بسم الله الرحمن الرحيم

جامعة السّودان المفتوحة

برنامج علوم الحاسوب وتقانة المعلومات

أساليب البرمجة (٣)

رمز المقرر: حسب ٢٠٢٣

تأليف: د. عبد الرحمن أحمد محمد عثمان

منشورات جامعة السودان المفتوحة الطبعة الأولى ٢٠١١م

جميع الحقوق محفوظة لجامعة السودان المفتوحة، لا يجوز إعادة إنتاج أيّ جزء من هذا الكتاب، وبأيّ وجه من الوجوه، إلاّ بعد الموافقة المكتوبة من الجامعة.

التحكيم العلمي ◄◄ د. أحمد صلاح الدين عبد الله

التصميم التعليمي ◄◄ د. عبد الباسط محمد شريف محمد

أ. أبو طالب محمد عيسى

مراجعة التصميم التعليمي ◄◄ أ. عمر الشيخ هجو

التدقيق اللغوي ◄◄ د. محمد حمدنا الله رملي

التصميم الفني ◄◄ أ. منى حمدان موسى محمد

مقدمة المقرر

الحمد لله حمداً كثيراً، الذي خلق كل شيء وقدره تقديراً. علّم الإنسان ما لم يكن يعلم، وكان فضله علينا عظيماً، ومن مظاهر فضله على الإنسان أن زوّده بهذه الطاقة المعنوية الرهيبة التي هي العقل والتي استطاع بواسطتها اكتشاف هذه القوة المادية الهائلة، وهو الحاسوب الذي أصبح الرمز الأول لهذه الحضارة، فقد أصبح الإنسان المعاصر بفضل هذا الاكتشاف قادراً على اختصار المسافات والأزمنة، وجعلها طوع إرادته. والصلاة والسلام على رسوله الأمين، المبعوث للناس هادياً ومبشراً ونذيراً. وصحابته الذين ورثوا منه العلم الغزير وحبّروه تحبيراً. صلوات ربي وسلامه عليه وعلى أهل بيته وسلم تسليماً كثيراً.

و بعد:

في الوقت الذي تزداد فيه الحياة تعقيداً، يواكب ذلك زيادة في المتطلبات التي ننتظرها من جهاز الكمبيوتر، ولم نعد نرضى بتنفيذ مهمة واحدة في الوقت نفسه، ولكننا نريد القيام بالكثير من الأعمال في الوقت نفسه، ومن الأمثلة على ذلك تفقد البريد الإلكتروني والعمليات المصرفية الإلكترونية، والمؤتمرات عبر الفيديو، وممارسة الألعاب، وحري عن الذكر قائمة طويلة أخرى من التطبيقات التي تعمل في خلفية الجهاز؛ مثل برنامج مقاومة الفيروسات. إننا نتوقع من الأجهزة التي نستخدمها في منازلنا ومكاتبنا أن تكون قادرة على أداء الكثير من المهام في وقت واحد. وعلاوة على ما ذكر، فإن جميع هذه التطبيقات هي بالفعل أكثر كثافة وإثارة للإعجاب مقارنة مع البرامج التي اعتدنا استخدامها في فترات سابقة. وفي هذه الأيام بتنا نطلب بالحصول على اللون الكامل، والحركة والفيديو بجودة عالية، والصور والرسومات المدهشة، ومن شأن كل هذه الاحتياجات أن تضع عبئاً هائلاً على المعالج التي هي أدمغة أجهزة الكمبيوتر المستخدمة لدينا.

عزيزي الدارس، يحتوي هذا الكتاب على أمثلة محلولة وتمارين متنوعة، كما تم وضع مواضيع هذا الكتاب في تسلسلٍ سلس، ولغةٍ ميسرة، مما يجعله يناسب طلاب الحاسوب في مختلف التخصصات.

ويشتمل هذا الكتاب على الوحدات التالية :

الوحدة الأولى وعنوانها : " عصر التعددية والبرمجة المتوازية " المعالجة المتوازية هي إحدى التوجهات الحديثة في صناعة المعالجات بعد الوصول إلى أقصى السرعات الممكنة في صناعة المعالجات التسلسلية، ومقاربتها لسرعة الضوء.

في هذه الوحدة حاولنا إعطاء القارئ فكرة واضحة حول مفهوم التعاون، ومن ثم تعاون الحواسيب لحل المسائل الكبيرة، كما توضح الوحدة الهدف من التعاون، كذلك تلخص الوحدة أسباب الاتجاه نحو التعددية .

الوحدة الثانية وعنوانها : " أنواع الحواسيب "، تتناول هذه الوحدة أنواع الحواسيب المتعددة، والتي تحتاج إلى أدوات وبرمجة خاصة بها . وفيها بيّنا أن الحواسيب المتعددة تنقسم إلى نوعين أساسيين؛ هما الحواسيب ذات الذاكرة المشتركة (Shared Memory)، الحواسيب ذات الذاكرة الموزعة (distributed memory) .

الحواسيب ذات الذاكرة المشتركة تشمل الحاسبات متعددة المعالجات (multiprocessors) والحواسيب متعددة النواة (multi-core) . أما الحواسيب ذات الذاكرة الموزعة الوحدة بينت أنها قد تكون متجانسة أو متباينة، المتجانسة مثل التوازي الكثيف للمعالجات (MPPs)، والتي منها الحواسيب العملاقة (super computer)، وتجمع المحطات (COWs) ومنها التجمعات (clusters) . أما الحواسيب المتباينة فمنها الشبكية (Grid) . ولكل واحد من هذه الأنواع صفاته الخاصة. كما إن معرفة الفروقات بين هذه الحواسيب يمكننا من التعرف على أساليب البرمجة المناسبة لكل نوع.

الوحدة الثالثة وعنوانها : "نماذج البرمجة المتوازية" ووفق هذا التقسيم هنالك عدة نماذج برمجية للبرمجة المتوازية، الوحدة تتناول هذا الجانب من خلال إلقاء نظرة

عامة لكل نموذج حيث تتناول نماذج الذاكرة المشتركة، ونموذج الخيوط و تبادل الرسائل، وكذلك تستعرض الوحدة نموذج توازي البيانات و نموذج الهجين.

الوحدة الرابعة وعنوانها : " مفهوم الخيوط "، وفيها تم شرح فكرة الخيوط، وقدمنا فيها تعريف الخيط (thread) . كذلك محتوى بنية معلومات الخيط، الوحدة تتناول أنواع الخيوط، وفصلت الفرق بين كل نوع، مع إعطاء مثال لكل نوع. كما تتناول بالشرح إنشاء الخيط، والتعامل مع الخيط، وحالات الخيط، الوحدة تتناول بالشرح أيضاً التحول بين العمليات Context Switch، واستخدامات الخيط في المعالج الواحد .

الوحدة الخامسة وعنوانها: " التزامن (Synchronizaation) " نتحدث في هذه الوحدة عن التزامن وهو كيف لعدة عمليات تعمل معاً أن تنسق عملها، بحيث لا تؤثر إحداها على الأخرى ويعتبر هذا المفهوم مهم جداً في البرمجة المتوازية، وعلى المبرمج أن يراعي المشاكل التي قد تنجم عن التزامن مثل مشاركة البيانات، وتعديلها من قبل أكثر من عملية في نفس الوقت، وفيها تم توضيح الأسباب التي تدعو العمليات المتوازية التعاون فيما بينها. وتشرح الوحدة مفهوم المنافسة (Competition)، ثم سرد لبعض الأمثلة التي تحدث فيها مشاكل نتيجة للإيقاف غير المتوقع لبعض العمليات بواسطة المجدول. واستعرضنا في هذه الوحدة بعض المشاكل الناجمة عن التزامن والتي تحدث في العمليات المتوازية.

الوحدة السادسة وعنوانها: " برمجة الخيوط باستخدام Pthreads "، نقدم في هذه الوحدة تعريف لمكتبات pthreads ، ونتناول استخدام مكتبات pthreads التي تعمل مع لغة c\c++، والتي تستخدم في معظم نظم تشغيل لينكس، كما نوضح كيفية إنشاء وإنهاء الخيوط والروتينات المستخدمة في إنشاء وإنهاء الخيوط . الوحدة تبحث كذلك في انضمام وانفصال الخيوط مع عرض الروتينات المستخدمة . وإدارة المكدسة، والروتينات المستخدمة، والوقاية من مشاكل المكدسة. نوضح كذلك في هذه الوحدة كيفية استخدام متغيرات Mutexs وكيفية إنشاء وتدمير الـ Mutexs ، كما نستعرض كيفية غلق وفتح Mutexs، ونستعرض كذلك في هذه الوحدة متغيرات

الحالة وهنا سنوضح كيفية إنشاء وتدمير متغيرات الحالة. والروتينات المستخدمة . كذلك الانتظار في متغيرات الحالة .

الوحدة السابعة وعنوانها : " البرمجة باستخدام **OpenMP** " في هذه الوحدة، سنتعلم كيف نكتب برامج متعددة الخيوط باستخدام مكتبة **OpenMP**، والتي تعتبر بيئة لبرمجة الخيوط تعتمد على موجهات المترجمات (**compiler directives**). وتختلف عن برمجة الخيوط التي تعلمناها مسبقاً في أن **OpenMP** هنا هي التي تنشيء الخيوط وليس المبرمج، وهذا يجعل برمجة الخيوط أكثر سهولة ويسراً.

في هذه الوحدة نقدم نبذة عن **OpenMP**، كذلك بعض موجهات **OpenMP** المهمة، كما تعدد الوحدة بعض دوال **OpenMP** المهمة . ثم نتناول تشارك البيانات، ونشرح كيف نستخدم **OpenMP** مع الأنظمة ذات الذاكرة المشتركة، ونوضح كيف يتم توزيع الحمل في **OpenMP**، وتتناول الوحدة بالشرح العقبات في برمجة الخيط من خلال الأمثلة، وأيضاً تبين الوحدة الامكانيات التي توفرها **OpenMP** للتزامن. ونتناول برنامج **OpenMP** حيث سنوضح الشكل العام للبرنامج . ونقوم بكتابة الصيغة العامة لموجهات **OpenMP**. وستجد الإجابة عن السؤال كم خيط نريد ؟ وفي الوحدة أيضاً نتعرف على الدالة التي تقوم بتحديد عدد الخيوط التي تعمل لديك الآن .

الوحدة الثامنة وعنوانها : برمجة التجمع **(Cluster Programming)**

وفيها تم إعطاء نبذه عن **MPI**، وهي المكتبة الأكثر انتشاراً المستخدمة في تبادل الرسائل . كما استعرضنا بعض الأمثلة البرمجية، وفيها نوضح مفاهيم البرمجة المتوازية في التجمع . ونتناول قياس الأداء، وفيه نستعرض أحد طرق قياس أداء البرنامج هو حساب الزمن من بداية البرنامج وحتى نهايته . تتناول الوحدة كذلك تبادل الرسائل في **MPI**، وهنا سنتحدث عن الاتصال المحجوز، الاختناق وبعض من مسبباته، الاتصال غير المتزامن (غير المحجوز) .

الوحدة التاسعة وعنوانها : الحوسبة الشبكية (Grid Computing)

في هذه الوحدة نتناول الـــ Grid Computing ومكوناتها، وأمثلة لبعض طبقات Gridالوسيطة مثل Globus. حيث تبحث الوحدة في المفاهيم الأساسية للـــGrid، وتبدأ بتعريف الـــ Grid ومكوناته وكذلك نتناول الفرق بين التجمع (cluster) والـــ Grid. الوحدة تتناول طبقة Globus Toolkit وهي من أشهر الطبقات الوسيطة، حيث قدمنا نبذة عن Globus Toolkit، واستعرضنا أمثلة استخدام وخدمات وبرمجيات Globus Toolkit

الوحـــدة العاشـــرة وعنوانهـــا: A Simple API for Grid (SAGA) Applications

في هذه الوحدة كـــذلك ســـنلقي نظـــرة عامـــة علـــى A Simple API for Grid Applications (SAGA) كمثال لواجهات تطبيقات مـــستقلة عـــن طبقـــة Grid الوسيطة، حيث تعرِّف الوحدة SAGA، كما تقدم تطبيق عملي لبدء اســـتخدام SAGA بتحميل شفرتها (source)، ومميزاتها، ونستعرض كذلك الحزم الوظيفية لـــ SAGA .

عزيزي الدارس، لقد كتبت هذه الوحدات بأسلوب سهل وواضح يتفق مع أسلوب التعليم الذاتي. لذا يتوقع منك أن تقرأ وتستعد لكل موضوع قبل مناقشته في اللقاء مع مشرفك الأكاديمي، لتكون مشاركتك وتفاعلك في اللقاء عاملاً مساعداً على نجاح الدرس وإثرائه.

كذلك اشتملت هذه الوحدات على عدد من الأسئلة التي وضعت لتساعدك في إثراء معلوماتك، وتعزيز ما تتعلمه، وتقييم درجة فهمك واستيعابك للمفاهيم والمعلومات والأفكار التي وردت في هذه الوحدات.

وختاماً نرجو أن نكون قد وفقنا في تقديم عمل مفيد للمكتبة العربية من خلال هذا الكتاب. وأسأل الله تعالى أن يجعله خالصاً لوجهه الكريم، وأن يجعله في ميزان حسناتنا يوم لا ينفع مال ولا بنون إلا من أتى الله بقلب سليم. والحمد لله أولاً وآخراً، وهو من وراء القصد، وهو يهدي السبيل.

الأهداف العامة للمقرر



عزيزي الدارس، بعد فراغك من دراسة هذا المقرر يؤمّل أن تكون قادراً على:

- توضيح المفاهيم الأساسية للبرمجة المتوازية .
- وصف الأنواع الأساسية للحاسبات المتعددة.
- الكتابة عن نماذج البرمجة المتوازية.
- ذكر نماذج البرمجة المتوازية.
- توضيح فكرة عمل الخيوط بأمثلة تشبيهة.
- شرح مفهوم التوازي.
- الكتابة عن مكتبات POSIX thread.
- الكتابة عن برامج متعددة الخيوط باستخدام مكتبة OpenMP.
- التحدث عن مكتبة "واجهة تبادل الرسائل" MPI.
- توضيح بالرسم الشكل العام لـــ SAGA.

محتويات المقرر

متسلسل	اسم الوحدة	الصفحة
الوحدة الأولى	عصر التعددية والبرمجة المتوازية	1
الوحدة الثانية	أنواع الحواسيب	25
الوحدة الثالثة	نماذج البرمجة المتوازية	47
الوحدة الرابعة	مفهوم الخيوط	65
الوحدة الخامسة	التزامن	93
الوحدة السادسة	برمجة الخيوط باستخدام Pthreads	133
الوحدة السابعة	البرمجة باستخدام OpenMP	191
الوحدة الثامنة	برمجة التجمع Clusters programming	221
الوحدة التاسعة	الحوسبة الشبكية (Grid Computing)	257
الوحدة العاشرة	A Simple API for Grid Applications (SAGA)	283


الوحدة الأُولى
عصر التعددية
والبرمجة المتوازية

محتويات الوحدة

الموضوع	الصفحة
المقدمة	3
تمهيد	3
أهداف الوحدة	4
توطئة	5
1. مفهوم التعاون	7
1.1 أمثلة تشبيهية للتعاون	7
2.1 تعاون الحواسيب	10
3.1 الهدف من التعاون (التوازي)	12
2. أسباب الإتجاه نحو التعددية	13
3. أمثلة للتطبيقات التي تحتاج إلى قوة معالجة عالية	14
4. لماذا ستصبح زيادة المعالج مستحيلة	15
5. تعريفات ظهرت في عصور التعددية	18
الخلاصة	20
لمحة مسبقة عن الوحدة التالية	21
إجابات التدريبات	22
مسرد المصطلحات	23
المراجع	24

المقدمة

تمهيد

عزيزي الدارس، مرحباً بك إلى الوحدة الأولى من مقرر أساليب البرمجة (3) وموضوعها عصر التعددية والبرمجة المتوازية، وتسعى هذه الوحدة إلى تزويدك بالمفاهيم الأساسية للبرمجة المتوازية.

تبدأ الوحدة بتوطئة تفسر فيها اتجاه شركات صناعة الحواسيب نحو إنتاج حواسيب ذات معالجات متعددة النواة.

القسم الأول يستعرض مفهوم التعاون، وهنا تقدم الوحدة طرق مختلفة لتقليل الوقت والجهد في المهام الكبيرة. وكذلك يستعرض القسم أمثلة تشبيهية للتعاون. وفي القسم نجد أيضاً تعاون الحواسيب، ويستعرض القسم أيضاً الهدف من التعاون (التوازي).

أما القسم الثاني فيوضح أسباب الاتجاه نحو التعددية، والقسم الثالث يستعرض أمثلة للتطبيقات التي تحتاج قوة معالجة عالية.

القسم الرابع يقدم إجابة عن السؤال لماذا ستصبح زيادة سرعة المعالج مستحيلة ؟

أما القسم الخامس والأخير، فيتناول تعريفات ظهرت في عصر التعددية.

أهداف الوحدة



عزيزي الدارس، بعد فراغك من دراسة هذه الوحدة ينبغي أن تكون قادراً على أن:

- **تفسر اتجاه شركات صناعة الحواسيب نحو إنتاج حواسيب ذات معالجات متعددة النواة.**
- **توضح فائدة التعددية حال استخدام أسلوب برمجة غير تسلسلي.**
- **تبين كيف تقلّل الوقت والجهد عند إنجازك لمهمة كبيرة.**
- **تعرض أمثلة تشبيهية للتعاون.**
- **تصف تعاون الحواسيب لتنفيذ مهمة بالتوازي.**
- **تذكر أسباب الاتجاه نحو التعددية.**
- **تعدد أمثلة لتطبيقات تحتاج إلى قوة معالجة عالية.**
- **تفسر لماذا ستصبح زيادة سرعة المعالجة مستحيلة.**
- **تعرف المفاهيم التالية:**

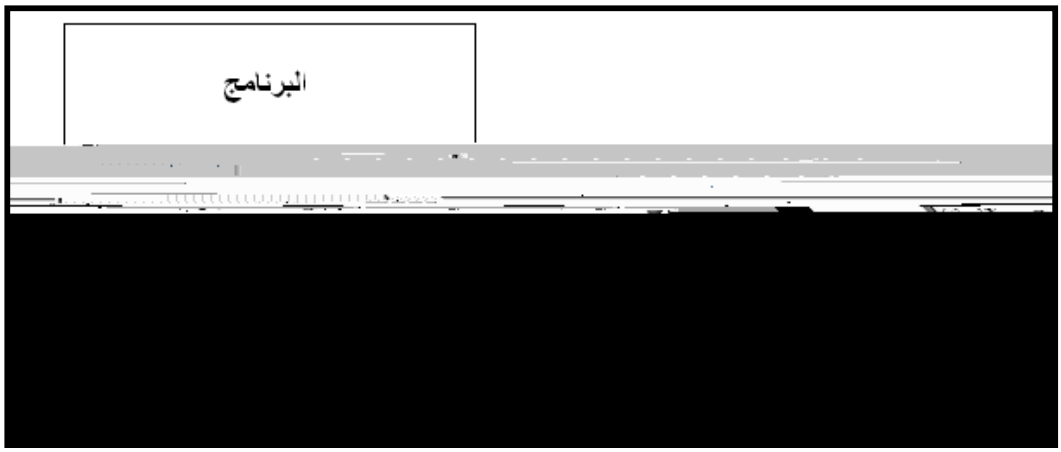
التجمع – الحوسبة الموزعة – الحوسبة المتنلقة – الحوسبة الشبكية – الحوسبة السحابية.

توطئة

معظم التطبيقات العملية الحديثة تطلب سرعة تنفيذ عالية بشره ونهم كبيران بحيث لا يشبعهما زيادة سرعة المعالج الواحد مهما بلغت. لهذا السبب أتجهت معظم شركات تصنيع الحواسيب نحو إنتاج حواسيب ذات معالجات متعددة النواة (multi-core processors)، فظهرت المعالجات ثنائية النواة (dual-core)، ثم تبعتها الآن المعالجات رباعية النواة (quad-core). الاتجاه نحو المعالجات متعددة النواة جعل مفهوم البرمجة المتوازية هو الأساس. فعلى مصنعي البرامج والمبرمجين الاتجاه إلى كتابة برامجهم بمفهوم جديد، وهو مفهوم المعالجة المتوازية التي تمكنهم من الاستفادة من هذا التعدد، وإلا فلن يكون هنالك فرق بين معالج ذي نواة واحدة ومعالج ذي مائة نواة.

❐ البرمجة التسلسلية

عزيزي الدارس، درسنا في كتاب أساليب البرمجة (1) وكتاب أساليب البرمجة (2) كيف نكتب برامج متسلسة تعمل على جهاز ذو معالج واحد؟ سواء أكانت هذه البرامج إجرائية (procedural) أو كائنية (object oriented). هذا النوع من البرمجة هي التي نعرف ونستخدم لأكثر من 50 عاماً، وكل الأجهزة والبرمجيات صممت لتنفذ هذه النوع من البرامج التسلسلية. فالبرنامج المتسلسل مكون من سلسلة من الأوامر (التعليمات) التي تنفذ بواسطة حاسوب واحد تسلسلياً (أي أمر تلو الآخر).



عادة ينفذ البرنامج المتسلسل على جهاز ذي معالج واحد، حيث لايستطيع المعالج تنفيذ أكثر من أمر في اللحظة الواحدة. لذلك إذا كان لدينا برنامج مكون من مائة أمر وكل أمر يحتاج إلى ن وحدة زمنية، فإن تنفيذ البرنامج كاملاً سيحتاج إلى (ن × 100) وحدة زمنية. وحتى لو نفذت مثل هذه البرامج على حاسوب متعدد المعالجات فلن يستفيد هذا البرنامج من هذه المعالجات، ويكون التنفيذ على معالج واحد منها وبقية المعالج غير مستفاد منها. لذلك قد لا تشعر بفائدة التعددية إذا لم تستخدم أسلوب برمجة غير تسلسلي.

1. مفهوم التعاون

إذا كان لديك عزيزي الدارس، مهمة كبيرة يحتاج إنجازها إلى وقت طويل وجهد كبير، ستحاول تقليل الوقت والجهد بطرق مختلفة؛ مثل :

أ- بذل جهد أكبر ونشاط أكثر.

ب- استخدام طرق ذكية لتقليل وقت المهمة.

ت- الاستعانة بآخرين لمساعدتك.

يجب أن تكون المهمة قابلة للتقسيم حتى يساعدك الآخرون في إنجازها. مثلا قراءة الكتاب لا تتم إلا تسلسلياً فلا يمكن لشخصين أو أكثر أن يستخدما الكتاب نفسه في الوقت نفسه، لكن مثلا لبناء حائط أو حصاد زرع، فيمكن لعدد من الأشخاص التعاون في ذلك وفى وقت واحد.

أيضا بعض المهام قد تفشل إذا لم تنجز بسرعة (لها قيد زمني). مثلاً لو كانت المهمة هي بناء منزل فلن يؤثر التأخير في إكمال المهمة. بعكس حصاد زرع الذي قد يفسد إذا لم يحصد في وقته.

إذن في حالة الزرع (والحالات المشابهة له) نحتاج الإسراع في إنجاز الحصاد ليس لتقليل الوقت والجهد فقط وإنما أيضاً لتلافي الضرر (فوات زمن الحصاد).

1.1 أمثلة تشبيهية للتعاون

أ) الحصاد

لحصاد زرع في مساحة حجمها 1000

المائة فدان ستستغرق 5 × 100 = 500 ساعة، تعتبر هذه المدة طويلة، ولتقليلها علينا القيام بإحدى من ثلاثة طرق:

تغيير هذا المزارع بمزارع آخر أقوى وأنشط: في هذه الحالة سنقلل الزمن بفرق بسيط، فمهما كان المزارع قوياً ونشيطاً فلن يستطيع إنجاز العمل في أقل من 4 أو ثلاث ساعات وبالتالي سيكون زمن الزراعة الكلي هو (300) ساعة.

استخدام طرق أخرى في الحل، كأن يحضر المزارع آلة تساعده في الزراعة، أو أن يستغل وقته بصورة جيدة، يحاول ألا يتوقف كثيراً أثناء الزراعة أو غيرها من الطرق التي تقلل زمن الزراعة.

الاستعانة بمزارعين آخرين يساعدوه في الزراعة، وكلما زاد عدد المزارعين كلما قل وقت الزراعة، فإذا أحضر 9 مزارعين ليصبح العدد الكلي (10) وافترضنا أن المزارعين بنفس قوة المزارع الأول (للتبسيط)، فإن العمل سينجز في (5 × 100) ÷ 10 = 50 ساعة، إذا كان لدينا مائة مزارع فإن كل مزارع سيزرع فدان في 5 ساعات وكل المزارعين سيعملون في وقت واحد، أي أن العمل سينجز في (5) ساعات فقط ((5 × 100) ÷ 100).

كيف ينجز الحاسوب المهام الكبيرة ؟

- **تغيير المعالج بآخر أسرع.**
- **استخدام طرق ذكية لتقليل زمن المعالجة؛ مثل:**
  - **الأنابيب الانسيابية (pipeline).**
  - **جدولة المعالج بواسطة نظام التشغيل (استغلال زمن المعالج بقدر ما يمكن).**
- **تقسيم المهمة على أكثر من معالج.**

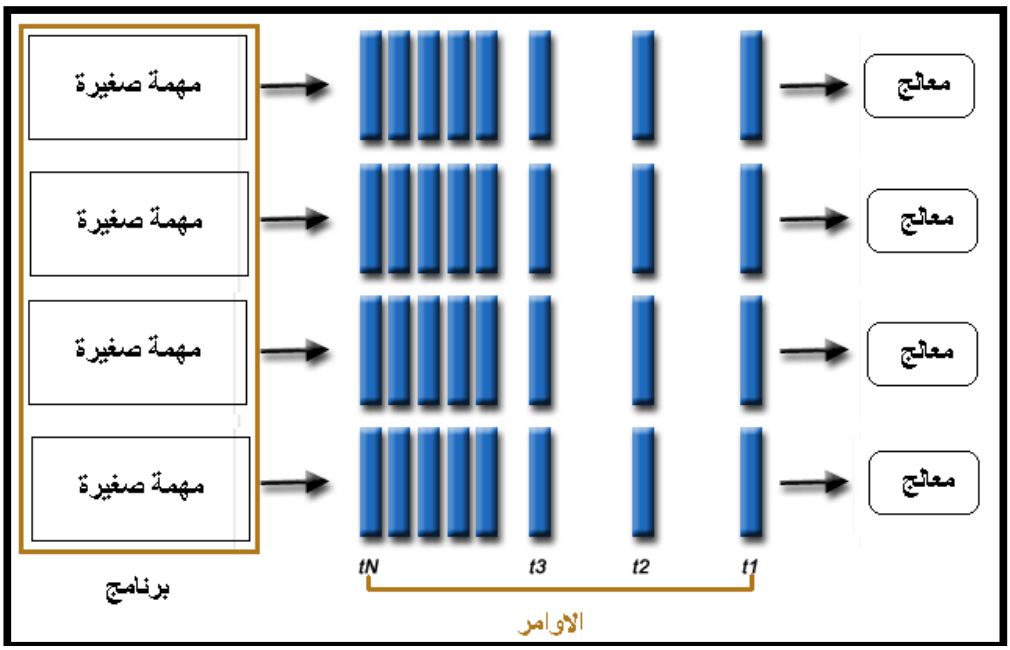
2.1 تعاون الحواسيب

المعالجة المتوازية هي تطبيق لفكرة الزراعة أعلاها، فإذا استبدلنا الزراعة بمسألة حاسوبية كبيرة تحتاج إلى وقت طويل للتنفيذ، واستعضنا عن المزارعين بحواسيب، فستتعاون هذه الحاسبات لتنفيذ هذه المسألة بالتوازي، حيث سيتم :

- تقسيم المسألة إلى أجزاء صغيرة.
- توزيع كل جزء على جهاز لتنفيذه.
- استلام النتائج بعد التنفيذ من الأجهزة بواسطة حاسوب منسق.

تعاون الحواسيب لحل المسائل الكبيرة

تقسم المشكلة الكبيرة إلى اجزاء صغيرة، بحيث ينفذ كل جزء في معالج بالتوازي (في وقت واحد).



فمثلا لو كان لدينا مهمة واحدة كبيرة، وقسمناها على m مهمة صغيرة، وكل مهمة تحتاج t زمن للتنفيذ، فالزمن الكلي لتنفيذ هذه المهام على معالج واحد هو:

$$\sum_{i=1}^{m} t_i$$

أما إذا وزعنا هذه المهام على عدد n معالج فإنه :

- **إذا كانت n أكبر أو تساوي عدد المهام m، فإن زمن التنفيذ الكلي سيكون:**

Max (t1,t2, …., tm)

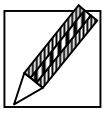
أما إذا كانت n أقل من m فإن على بعض المهام الانتظار حتى تكتمل مهام أخرى، حيث سيكون الزمن الكلي للتنفيذ هو:

round (m/n) * t

توزيع المهام بالتساوي بين المعالجات يسمى "load balancing".

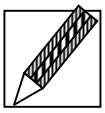
زمن تنفيذ أي مهمة (t) هو زمن تنفيذ المهمة بالمعالج، مضافاً إليه زمن الاتصال (delay time).

تدريب (1)



إذا كانت لدينا مهمة واحدة، و تم تقسيمها إلى (5) مهام صغيرة، وكل مهمة تحتاج إلى زمن تنفيذ بالثواني على النحو التالي: $t_1=3, t_2=2, t_3=5, t_4=7, t_5=1$، أحسب الزمن الكلي لتنفيذ هذه المهام على معالج واحد.

تدريب (2)



إذا كانت لدينا مهمة واحدة، وتم تقسيمها إلى (6) مهام صغيرة، و كل مهمة تحتاج إلى زمن تنفيذ بالثواني على النحو التالي :
t1= 4, t2= 10 , t3=12,t4= 7,t5 = 9, t6=11 أحسب الزمن الكلي إذا تم توزيع المهام على(7) معالجات.

3.1 الهدف من التعاون (التوازي)

- ☑ **حل المسائل الكبيرة بسرعة عالية.**
- ☑ **توفير الوقت (save time).**
- ☑ **توفير التزامنية (عمل عدة أشياء في وقت واحد).**
- ☑ **توفير بيئة للبرامج التي بطبيعتها متوازية.**
- ☑ **الاستفادة من الموارد الموجودة بالشبكة أو بالإنترنت.**
- ☑ **تقليل التكلفة باستخدام أجهزة شخصية متعددة رخيصة بدلاً من جهاز حاسوب واحد فائق السرعة (supercomputer) وباهظ الثمن. حيث يعتبر تكوين حاسوب فائق السرعة من حواسيب متفرقة هو بديل قليل التكلفة للحواسيب الفائقة والتي يكلف شراؤها مبالغ طائلة. فمثلا حاسوب فائق بسرعته 12.3 تيرا فلوب (teraflops) (ترليون عملية)) قد يكلف حوالي مائة مليون دولار.**
- ☑ **البرامج الكبيرة والتي تحتاج إلى ذاكرة كبيرة يمكنها الاستفادة من ذاكرة الحواسيب الموجودة بالشبكة.**
- ☑ **الاعتمادية (reliability).**
- ☑ **الموثوقية (يمكن تنفيذ البرنامج في أكثر من جهاز ومقارنة النتائج للتأكد من صحتها).**

أسئلة تقويم ذاتي

1. **فسر اتجاه شركات صناعة الحواسيب نحو انتاج حواسيب ذات معالجات متعددة النواة.**
2. **وضّح فائدة التعددية حال استخدام أسلوب برمجة غير تسلسلي.**
3. **بيّن كيف تقلل الوقت والجهد عند انجازك لمهمة كبيرة.**
4. **قدم بعض الأمثلة التشبيهية للتعاون.**
5. **صفْ تعاون الحواسيب لتنفيذ مهمة بالتوازي.**
6. **اذكر الهدف من التعاون (التوازي).**

2. أسباب الإتجاه نحو التعددية

مما أدى إلى الاتجاه إلى الحواسيب المتعددة هو التطور السريع في المكونات المادية والشبكات. فقد تحولت الحواسيب المركزية الغالية إلى حواسيب شخصية رخيصة، وتحولت الشبكات من بطيئة وكلفة إلى شبكات فائقة السرعة وغير مكلفة.

كانت الحواسيب (المركزية) باهظة الثمن، ولا يستطيع شخص عادي امتلاكها، فكان الحل التشارك في استخدامها عن طريق طرفيات (terminal) (لوحة مفاتيح وشاشة لكل مستخدم)، بهذه الطريقة يمكن لعدد كبير من المستخدمين الاستفادة من حاسوب واحد (mainframe). ثم أصبحت الحواسيب صغيرة وقليلة التكلفة وأصبح بمقدور كل شخص شراء حاسوب وبالتالي لا حاجة إلى برامج مشاركة زمنية، ولا حاجة إلى تزاحم حول حاسوب واحد.

ظهور الشبكات التي تربط الحواسيب الشخصية فيما بينها جعل من السهل التشارك في الموارد من عتاد وبرامج. هنا توسع الإنسان في طموحه، وبدأ يصمم ويستخدم

تطبيقات تحتاج قوة معالجة عالية (إمكانيات أعلى من طاقة الحواسيب الشخصية)، فكان الحل هو أحد خيارين:

- استخدام الحواسيب الفائقة (super computers)، وهي مكلفة جداً.
- استخدام عدة معالجات (حاسوب الفقراء الفائق Poor's man super computer)، فالذي لا يستطيع شراء حاسوب فائق يمكنه تجميع عدة معالجات لتكوين حاسوب فائق افتراضي (Virtual Super Computer)، هذا الحاسوب الافتراضي نسميه تجمع (Cluster).

أسئلة تقويم ذاتي

؟	اذكر أسباب الاتجاه نحو التعددية.

3. أمثلة للتطبيقات التي تحتاج قوة معالجة عالية

- مسائل النمذجة المحاكاة (Simulation and Modeling problems).
- التعرف على الكلام (speech recognition).
- التنبؤ بالطقس (Climate Modeling).
- محركات البحث على الإنترنت وخدمات الأعمال الالكترونية.
- التشخيص بمساعدة الحاسوب في الطب.
- الرسومات المتقدمة والواقع الافتراضي، وبخاصه في صناعة الترفيه.
- تقنيات الفيديو عبرالشبكات والوسائط المتعددة.
- المسائل التي تعتمد على الحسابات ومعالجة بيانات ضخمة.
- معالجة الصور والإشارات (Image and Signal Processing).
- تنقيب البيانات (Data Mining).
- الجينوم البشري (Human Genome).

أسئلة تقويم ذاتي



عدد بعض الأمثلة لتطبيقات تحتاج إلى قوة معالجة عالية.

4. لماذا ستصبح زيادة سرعة المعالج مستحلية؟

وضع "مور" فرضية تقول بأن المعالجات تتضاعف سرعتها كل (18) شهراً، هذه الفرضية أثبتت صحتها لقرابة ثلاثين عاماً. فاذا كنت تعتقد أن كتابة برامج على حواسيب متعددة صعبة، وأنك ستنتظر حتى يصل الحاسوب الشخصي للسرعة التي تريد (حسب قانون مور) فهذا لن يحصل للأسباب الآتية:

- **يستخدم مصنعو المعالجات خدع كثيرة لزيادة سرعة المعالج معتمدة على مفهوم المعالجة المتوازية وعلى هرمية الذاكرة (ذاكرة المسجلات، الذاكرة المخبأة، الذاكرة الرئيسية، والذاكرة الثانوية)، وهذا يعتبر من الأساليب الذكية لزيادة أداء الحاسوب، ولكن لا يعتبر زيادة فعلية لسرعة الحاسوب.**
- **الحصول على سرعة عالية من المعالجات أصبح أكثر صعوبة وتعقيداً.**
- **سرعة المفاتيح (switching speeds) ستصل نهايتها، وتصغير حجم الدوائر الإلكترونية محدود بسرعة الضوء. سنصل قريباً إلى نقطة يصبح فيها زيادة كفاءة الحاسوب الشخصي مستحيلا (performance limit). وستصبح الحواسيب المتعددة هي البديل المناسب.**
- **لدينا عقبات تقف أمام استمرارية زيادة سرعة المعالج (قريباً ستصل سرعة المعالجات نهايتها)، نذكر منها:**
  - **إن سرعة الحاسوب ككل مرتبطة بسرعة أبطأ جزء (تسلسلي) به. ونلاحظ من الشكل أعلاه أن سرعة المعالج تزيد بنسبة 60% سنوياً، بينما تزيد النسبة للذاكرة بـــ 7% فقط بالسنة. تعتبر الذاكرة مؤثراً رئيسياً على سرعة الحاسوب.**

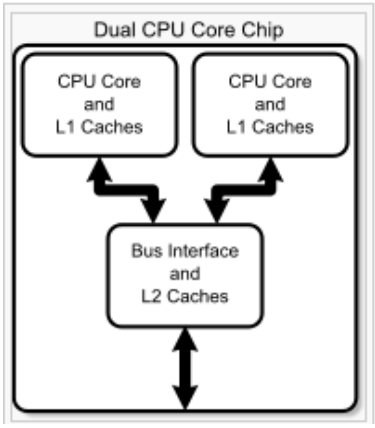
- فقدان جزء كبير من سرعة المعالج بسبب الزمن المستغرق في نقل المعلومة من الذاكرة إلى المعالج (ترتبط سرعة المعالج بسرعة وصول البيانات من الذاكرة).

• لجعل المعلومة تنتقل أسرع لابد من تقليل المسافة المقطوعة بين المعالج والذاكرة.

مثلا إذا كنت تمتلك حاسوباً بسرعة 1 GHz، فهذا يعني أن في كل نانو ثانية سيقوم المعالج بعمل دورة (cycle) واحدة (أو بمعنى آخر بليون دورة (clock cycle) في الثانية الواحدة). الضوء يستطيع عبور (30) سنتمتراً (حوالى قدم) في النانو ثانية (nanosecond) أو في 1 من البليون من الثانية. والمعلوم أن الاشارات الرقمية تنتقل بسرعة الضوء. لذلك الدارات الالكترونية التي تنتج سرعة المعالج (clock speeds) من الأفضل ألا يزيد حجمها (على الأقل عُشرها) عن 30 سنتمتراً. هذا يعني أن أكبر حجم للدارة الالكترونية يجب ألا يتعدى (3) سنتمتر.

إذا أردنا تصميم معالج بسرعة 100GHz، فهذا يعني أنه لابد لهذا المعالج من إنتاج الدورة الواحدة (clock cycle) في كل 1 من المائة من النانو ثانية. هذا يعني أن الإشارة تستطيع عبور 3 ملمتر في النانو ثانية (سرعة الضوء). لذلك من الأفضل أن يكون حجم قلب المعالج 0.3 ملمتر. وقد يكون من العسير جداً صناعة معالج بهذا الحجم لكنه – مع وجود التقنيات الحديثة – أمر ممكن.

عزيزي الدارس، من هنا نستنتج أنه كلما قلصنا حجم الدارات الالكترونية زادت سرعة المعالج، ولكن إلى متى سنظل نقلص في الحجم ؟ بالتأكيد سنصل نقطة يصبح فيها تقليص الحجم مستحيلاً، وبالتالي ستصبح زيادة سرعة المعالج وتحسين أداء الحاسوب ذي المعالج الواحد مستحيلاً. بمعنى آخر قد تصل سرعة المعالج الواحد إلى نهايتها، ولن نستطيع زيادة الميغاهيرتز (MHz)، وهذا واضح الآن في سوق الحواسيب، فقد بدأت حالياً الكثير من الشركات المنتجة للأجهزة تتجه نحو زيادة عدد المعالجات في الحاسوب بدلاً من زيادة سرعة المعالج، فظهر ما يسمى المعالجات متعددة النواة (multi-core CPU)، شكل رقم (1–6).



شكل رقم (1–6): معالج ثنائي القلب

مثال:

سرعة الضوء هي الحد الفاصل لزيادة سرعة الحاسوب.

افترض أننا نريد بناء حاسوب (تسلسلي) بذاكرة حجمها 1TB وبسرعة 1 Tflop.

إذا كانت المسافة بين الذاكرة الرئيسية والمعالج هي d، فعلى البيانات قطع هذه المسافة لتصل المعالج، ولابد من قطع هذه المسافة بسرعة 10^12 مرة في الثانية بسرعة الضوء (c=3e8 m/s)، إذن فيجب أن تكون:

d <= c/10^12 =.3 mm

إذن على هذا الحاسوب أن يوضع في صندوق حجمه 3. ملم

أيضا لبناء ذاكرة بحجم 1TB، سنحتاج إلى شبكة (10^6 في 10^6) من الكلمات (words)

إذا كان حجم هذه الشبكة هو 3. ملم × 3. ملم، إذن فالكلمة الواحدة يجب أن يكون حجمها 3 أنغوستراام × 3 أنغواستراام (Angstroms)، أو حجم ذرة (atom) صغيرة.

من الصعب تخيل أين سنضع الأسلاك؟

5. تعريفات ظهرت في عصور التعددية

- **التجمع (Cluster)**

حواسيب متجانسة (متشابهة في المكونات المادية ونظم التشغيل)، ومتلاصقة (في مكان واحد) وشبكة محلية عالية السرعة.

- **الحوسبة الموزعة (Distributed Processing)**

حواسيب متباينة وشبكة بطيئة، وقد تكون الحواسيب متباعدة جغرافيا.

- **الحوسبة المتنقلة (Mobile Computing)**

تنشأ مؤقتاً، عند انعقاد مؤتمر، في معرض، في الكلية بواسطة شبكات لاسلكية، وهواتف نقالة وحواسيب نقالة (lab tops).

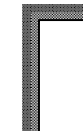
- **حوسبة شبكية Grid Computing**

موارد موزعة في مناطق مختلفة وبعيدة عن بعضها ويمكن استخدامها كحاسوب واحد إفتراضي كبير، وهي أشبه بذلك للتجمع (cluster) ولكنها تختلف في كون أجزائها موزعة جغرافيا ومتباينة في العتاد ونظم التشغيل. وهي أشمل من الحوسبة الموزعة، ويمكن أن تكون هجيناً بين كل الأنواع السابقة. حيث يمكنك الدخول على الحوسبة الشبكية من بعيد عبر الإنترنت مثلاً، والاستفادة منها في تنفيذ المهام الكبيرة. يمكن أن تكون الحوسبة الشبكية تربط معامل مختلفة بشبكة مدنية أو واسعة.

- **الحوسبة السحابية Cloud Computing**

عبارة عن موارد مختلفة متاحة عبر الإنترنت، مملوكة لجهات معينة تقوم بتأجيرها للناس، أو إتاحتها لهم مجاناً. يمكن الاستفادة منها في مجالات مختلفة، مثل تأجير معمل في فترة معينة، أو تأجير عدد من الجيغابياتات للتخزين، أو استخدام برامج لا تسطيع شراءها.

أسئلة تقويم ذاتي



1. **فسّر لماذا ستصبح زيادة سرعة المعالجة مستحيلة؟**
2. **عرِّف المفاهيم التالية:**

التجمع – الحوسبة الموزعة – الحوسبة المتنلقة – الحوسبة الشبكية – الحوسبة السحابية.

الخلاصة

بينت الوحدة أن اتجاه معظم شركات تصنيع الحواسيب نحو إنتاج حواسيب ذات معالجات متعددة النواة بسبب أن معظم التطبيقات العملية الحديثة تطلب سرعة تنفيذ عالية، حيث لا يشبعها زيادة سرعة المعالج مهما بلغت.

وضحت الوحدة أنه يمكن تقليل الوقت والجهد في المهام الكبيرة بطرق مختلفة، منها :

- بذل جهد أكبر ونشاط أكثر - استخدام طرق ذكية لتقليل وقت المهمة - الاستعانة بآخرين لمساعدتك.

- قدمت الوحدة أمثلة تشبيهية للتعاون :

- الحصاد : يمكن تقليل الفترة الزمنية

يمكن حصاد زرع في مساحة كبيرة بإحدى طريقتين :- بذل جهد أكبر لتقليل زمن الحصاد - استخدام طرق ذكية لتسريع الحصاد مثل استخدام حاصدة ميكانيكية.

- تجميع السيارات : تصنيع أجزاء السيارة بوساطة شخص واحد تعتبر بطيئة ولكن تقسيم هذه العملية إلى مهام صغيرة، وتوزيعها على وإعطاء كل مهمة إلى شركة يعتبر الحل الأمثل. كذلك البرامج الكبيرة المعقدة يمكن تقسيمها إلى برامج صغيرة وتوزيعها على عدة معالجات، بحيث ينفذ كل معالج المهام بالتوازي يجعل التنفيذ أسرع.

- الزراعة : لتقليل زمن زراعة حقل كبير علينا القيام بالتالي :-

تغيير المزارع بمزارع أقوى وأنشط. استخدام طرق أخرى في الحل، كاستخدام آلة تساعده في الزراعة الاستعانة بمزارعين آخرين يساعده في الزراعة.

لمحة مسبقة عن الوحدة التالية

تتناول الوحدة التالية أنواع الحواسيب المتعددة التي تحتاج إلـــى أدوات وبرمجـــة خاصة بها، وتسعى هذه الوحدة إلى تزويدك بالمفاهيم الأساسية عن الأنظمة المتعـــددة بأنواعها المختلفة، نعتقد أنك ستسعد بدراسة هذه الوحدة، نتمنى لك التوفيق والسداد.

إجابات التدريبات

تدريب (1)

الزمن الكلي = $\sum_{i=1}^{m} t_i$

= 3+2+5+7+1 = 18 ثانية

تدريب (2)

$\sum_{i=1}^{m} t_i$

الزمن الكلي = Max (4,10,12,7,9,11)

= 12 ثانية

مسرد المصطلحات

❖ **التجمع (Cluster)**

حواسيب متجانسة (متشابهة في المكونات المادية ونظم التشغيل)، ومتلاصقة (في مكان واحد) وشبكة محلية عالية السرعة.

❖ **الحوسبة الموزعة (Distributed Processing)**

حواسيب متباينة وشبكة بطيئة، وقد تكون الحواسيب متباعدة جغرافيا.

❖ **الحوسبة المتنقلة (Mobile Computing)**

تنشأ مؤقتاً، عند انعقاد مؤتمر، في معرض، في الكلية بواسطة شبكات لاسلكية، وهواتف نقالة، وحواسيب نقالة (lab tops).

❖ **حوسبة شبكية Grid Computing**

موارد موزعة في مناطق مختلفة، وبعيدة عن بعضها، ويمكن استخدامها كحاسوب واحد إفتراضي كبير، وهي أشبه بذلك للتجمع (cluster) ولكنها تختلف في كون أجزائها موزعة جغرافيا، ومتباينة في العتاد ونظم التشغيل. وهي أشمل من الحوسبة الموزعة، ويمكن أن تكون هجين بين كل الأنواع السابقة، حيث يمكنك الدخول على الحوسبة الشبكية من بعيد عبر الإنترنت مثلاً، والاستفادة منها في تنفيذ المهام الكبيرة. يمكن أن تكون الحوسبة الشبكية تربط معامل مختلفة بشبكة مدنية أو واسعة.

❖ **الحوسبة السحابية Cloud Computing**

عبارة عن موارد مختلفة متاحة عبر الإنترنت، مملوكة لجهات معينة تقوم بتأجيرها للناس، أو إتاحتها لهم مجاناً. يمكن الإستفادة منها في مجالات مختلفة، مثل تأجير معمل في فترة معينة، أو تأجير عدد من الجيغابياتات للتخزين أو استخدام برامج لا تسطيع شراءها.

المراجع

1. عبد الرحمن عثمان(2008م)، الـــنظم المزوعـــة، منشـــورات جامعـــة الســـودان المفتوحة.

SP Parallel Programming Workshop,
http://www.mhpcc.edu/training/workshop/parallel_intro
Speed of light,
http://www.physlink.com/education/AskExperts/ae391.cfm

الوحدة الثانية

أنواع الحواسيب



محتويات الوحدة

الموضوع	الصفحة
المقدمة	27
تمهيد	27
أهداف الوحدة	28
توطئة	29
1. الأنظمة متعددة المعالجات	32
2. الأنظمة متعددة الحواسيب	35
1.2 الأنظمة متعددة الحواسيب المتجانسة	35
1.1.2 التوازي الكثيف المعالجات	36
2.1.2 تجمع المحطات	39
2.2 الأنظمة متعددة الحواسيب المتباينة	40
1.2.2 الحوسبة الشبكية	40
الخلاصة	42
لمحة مسبقة عن الوحدة التالية	43
مسرد المصطلحات	44
المراجع	46

المقدمة

تمهيد

عزيزي الدارس،

مرحباً بك إلى الوحدة الثانية من مقرر (أساليب البرمجة3) سنتحدث في هذه الوحدة عن أنواع الحواسيب المتعددة، والتي تحتاج إلى أدوات وبرمجة خاصة بها. حيث إنّ الحواسيب المتعددة تنقسم إلى نوعين أساسيين؛ هما الحواسيب ذات الذاكرة المشتركة (shared memory) الحواسيب ذات الذاكرة الموزعة (distributed memory).

القسم الأول من الوحدة يتناول الحواسيب ذات الذاكرة المشتركة تشمل الحواسيب متعددة المعالجات (multiprocessors)، والحواسيب متعددة النواة (multi-core). أما القسم الثاني من الوحدة فيستعرض الحواسيب ذات الذاكرة الموزعة فقد تكون متجانسة أو متباينة. المتجانسة مثل التوازي الكثيف للمعالجات (MPPs) ومنها الحواسيب العملاقة (super computer)، وتجمع المحطات (COWs)، ومنها التجمعات (clusters). أما الحواسيب المتباينة فمنها الشبكية Grid.

ولكل واحد من هذه الأنواع صفاته الخاصة. كما إن معرفة الفروقات بين هذه الحواسيب يمكننا من التعرف على أساليب البرمجة المناسبة لكل نوع. لكن الأفضل (إن استطعت) أن تكتب برامج يمكنها العمل على معظم هذه الحواسيب.

أهداف الوحدة



عزيزي الدارس، بعد فراغك من دراسة هذه الوحدة ينبغي أن تكون قادراً على أن:

- **تذكر الأنواع الأساسية للحاسبات المتعددة.**
- **تعدد أنواع الأنظمة المتعددة.**
- **تبين الفرق الرئيس بين انظمة الذاكرة المشتركة، وأنظمة موزعة الذاكرة.**
- **تعدد طرق الربط لنظم ذات ذاكرة مشتركة، ونظم ذات ذاكرة موزعة.**
- **ترسم أشكال الربط المختلفة لنظم ذات ذاكرة مشتركة، ونظم ذات ذاكرة موزعة.**
- **تعرف الأنظمة متعددة المعالجات.**
- **تذكر عيوب نظام المعالجات المتعددة.**
- **تعرف الأنظمة المتعددة الحاسبات.**
- **تعرف التوازي الكثيق للمعالجات.**
- **تعرف تجمع المحطات.**
- **تقارن بين التوازي الكثيف للمعالجات وتجمع المحطات.**
- **تعرف الأنظمة متعددة الحاسبات المتباينة.**

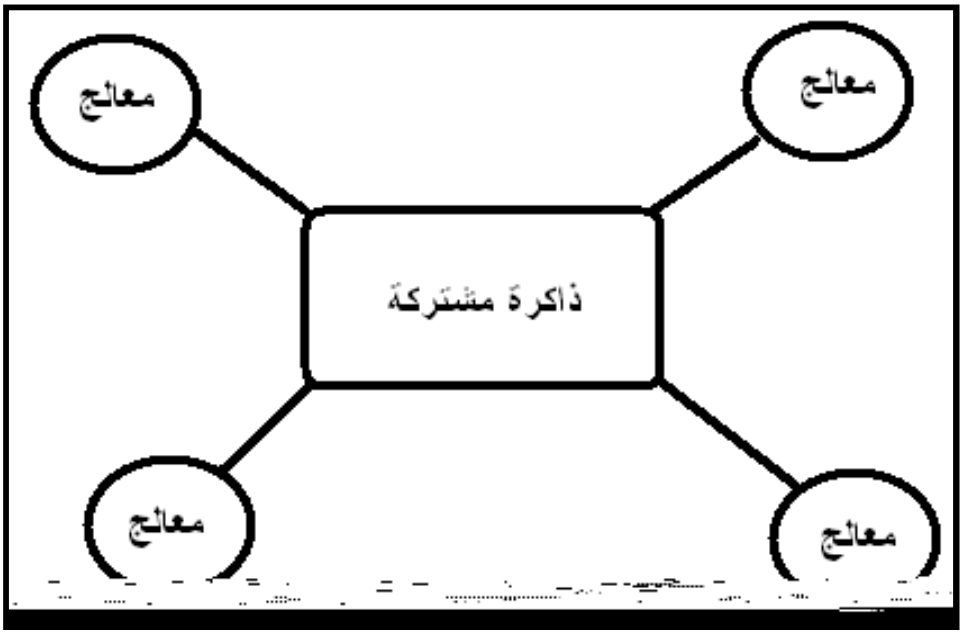
توطئة

الأنظمة المتعددة

الأنظمة المتعددة قسمان:

- **أنظمة ذات ذاكرة مشتركة، وتسمى الأنظمة متعددة المعالجات (multiprocessors).**
- **أنظمة موزعة الذاكرة، وتسمى الأنظمة متعددة الحواسيب (multicomputers).**

الفرق الرئيسي بين الاثنين هو أنّ في النوع الأول يشتمل على عنوان حقيقي (فيزيائي) واحد تتشارك فيه كل المعالجات CPUs، مثلاً إذا كتب معالج القيمة (7) في العنوان 2020 فإن أي معالج آخر سيحصل على ذات القيمة اذا احضر محتوى الذاكرة ذات العنوان (2020).



شكل رقم (2–1): نظم ذات ذاكرة مشتركة (shared memory).

1.الأنظمة متعددة المعالجات (multiprocessor)

عبارة عن مجموعة من المعالجات تتشارك في ذاكرة واحدة، بحيث يكون بمقدور أي معالج الوصول مباشرة لهذه الذاكرة المشتركة والتعامل معها.

قد ترتبط هذه المعالجات عن طريق ناقل مشترك (bus)، ترتبط على ناقل الذاكرة المشتركة نفسه. الناقل المشترك قد يكون شبكة أو ناقل في لوحة رئيسية (motherboard) تركب فيها كل المعالجات والذاكرة المشتركة.

وبما أن الذاكرة المشتركة واحدة، فإن كل معالج يضع قيمة في موقع ما بالذاكرة، تستطيع بقية المعالجات الوصول لتلك القيمة، والتعامل معها من قراءة أو تعديل أو غيرها.

عزيزي *الدارس*،، الذاكرة التي تمتلك هذه الخاصية نسميها (coherent). المشكلة الرئيسية في هذا النوع هو أن زيادة عدد المعالجات يخلق اختناقاً على الخط الناقل، مما يؤثر على الأداء بصورة واضحة جداً. إذن فما الحل؟

الحل هو وضع ذاكرة مخبأة (cache memory) بين الخط الناقل والمعالج، هذه الذاكرة المخبأة توضع بها البيانات الحديثة التي أخذت من الذاكرة الرئيسية، وبالتالي عند ما يطلب المعالج معلومة سيأخذها من هذه الذاكرة المخبأة إن وجدت فيها، وإذا لم توجد بها، فسيضطر لاستخدام الخط الناقل لإحضار المعلومة من الذاكرة الرئيسية.

هذه الطريقة قلّلت كثيراً من الزحام الذي كان موجوداً بالناقل الرئيسي. ولكن هل هذا الحلّ حلّ نهائي؟ لا! لقد ولّد هذا الحل مشكلة أخرى هي عدم توافقية البيانات (consistency problem).

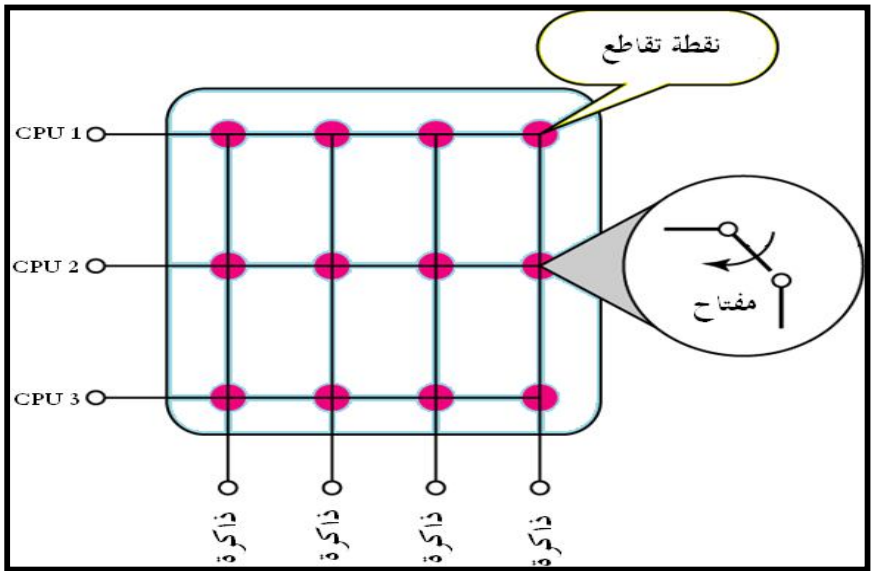
عزيزي الدارس، افترض أن معالجين قاما بنسخ قيمة من الذاكرة الرئيسية إلى ذاكرتيهما المخبأتين، ثم قام المعالج الأول بتعديل هذه القيمة بالذاكرة الرئيسية،

عند ما يطلب المعالج الثاني هذه القيمة سيحصل عليها من ذاكرته المخبأة (لن يأخذ القيمة الحديثة المعدلة بواسطة المعالج الأول)، هنا يظهر عدم توافقية القيم.

هذا يقود لذاكرة غير متماسكة (incoherent)، حيث يصعب برمجة النظام مع وجود هذه المشكلة.

من عيوب نظام المعالجات المتعددة هو عدم قدرته على التوسع (ليس به خاصية قابلية التوسع scalability) حتى مع استخدام الذاكرة المخبأة، فاذا أردنا بناء نظام متعدد المعالجات لأكثر من (256) معالجاً فلا بد من طريقة مختلفة لربط المعالجات مع الذاكرة الرئيسية المشتركة، لأن الخط الناقل المشترك سيزدحم ببيانات المعالجات وكثرتها تسبب اختناقاً فيه.

أحد الحلول هو تقسيم الذاكرة الرئيسية المشتركة إلى ذواكر كثيرة متعددة، وربط كل مجموعة من المعالجات مع واحدة من هذه الذواكر بطريقة مفاتيح متقاطعة (crosspoint switching) كما في الشكل أدناه.

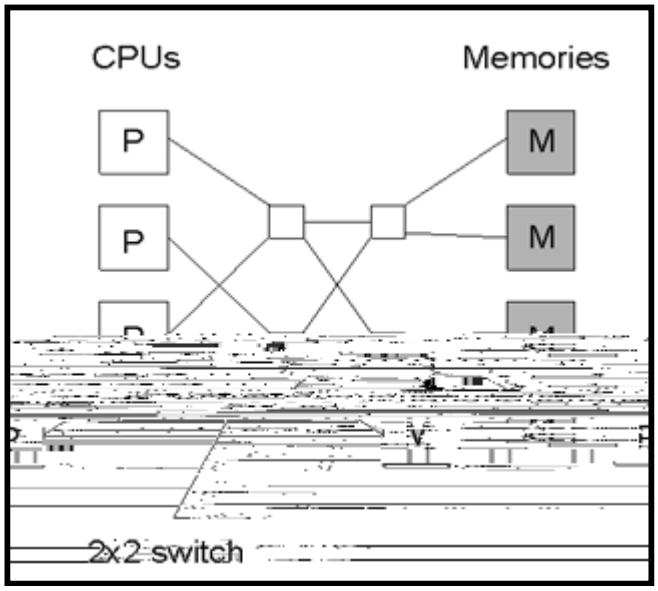


شكل رقم (2-4): مفاتيح تقاطع لتوصيل المعالجات مع الذواكر

كل مفتاح تقاطع (crosspoint switch) هو عبارة عن مفتاح يمكن فتحه و غلقه بواسطة جهاز معين. حيث يتم غلق هذا المفتاح في حالة طلب المعالج الوصول للذاكرة.

من مزايا هذا النظام أنه يسمح لأكثر من معالج أن يتعامل مع أكثر من ذاكرة في نفس الوقت نفسه، ولا ينتظر معالجُ معالجاً آخر للوصول لذاكرة معينة كما في الطريقة الأولى، إلا في حالة طلب معالجين الوصول لذاكرة معينة في الوقت نفسه، في هذه الحالة على معالج أن ينتظر الآخر حتى يفرغ من استخدام هذه الذاكرة.

عيوب هذا النوع أنه إذا أردت بناء مفاتيح لعدد ن معالج لربطها مع عدد (ن) ذاكرة مشتركة، فإنني أحتاج إلى ن × ن مفتاح للتوصيل وهذا كثير. الحل هو طريقة توصيل مختلفة تسمى مفتاح المقاطعات متعدد المستويات (omega network) كما في الشكل (3-5)، حيث يقل عدد المفاتيح كثيراً.



شكل رقم (2-5): مفاتيح مقاطعات متعددة المستويات

أسئلة تقويم ذاتي



1. عرِّف الأنظمة متعددة المعالجات (**multiprocessors**).

2. ما هو الحل للاختناق الذي ينشأ نتيجة لاستخدام المعالجات ذات الناقل المشترك ؟

2. الأنظمة متعددة الحواسيب Multicomputer

هو ربط أجهزة حاسوب مع بعضها البعض عن طريق شبكة عالية السرعة. وقد تكون الحواسيب المرتبطة مع بعضها متشابهة في العتاد ونظم التشغيل، وتسمى متجانسة (homogenous) أو قد تكون مختلفة في العتاد و/أو البرامج، وتسمى متباينة (heterogeneous). أدناه سنشرح كل نوع منها.

1.2 الأنظمة متعددة الحواسيب المتجانسة

Homogenous Multicomputer Systems

عزيزي الدارس، **مقارنة بتعدد المعالجات، يعتبر بناء نظام متعدد الحواسيب أسهل، حيث لكل حاسوب ذاكرته المحلية المرتبطة معه مباشرة. قد تكون هنالك مشكلة واحدة فقط هي كيف سيتم تبادل البيانات بين الحواسيب.**

هذه الأجهزة عادة ما تكون مستقلة عن بعضها البعض، لذلك لابد من رابط بينها يصل بعضها ببعض. هذا الرابط أو الشبكة لابد أن تكون جيدة الأداء. قد يكون الرابط ناقلاً مشتركاً تتصل عبره كل الأجهزة، مثل الإثيرنت السريع (Fast Ethernet)، الذي تصل سرعته (100) ميغابت في الثانية. استخدام الناقل (bus) سريع، ولكنه غير قابل للتوسع، فكلما زاد عدد الحواسيب في الشبكة قل الأداء. السبب في قلة الأداء هي طريقة عمل شكل الناقل (bus)، حيث يتم إذاعة الرسالة في الشبكة،

وهذه الطريقة تجعل حاسوباً واحداً فقط يستخدم الناقل في اللحظة المعينة. تخيل طوابير الحواسيب التي تنتظر دورها للوصول إلى الناقل.

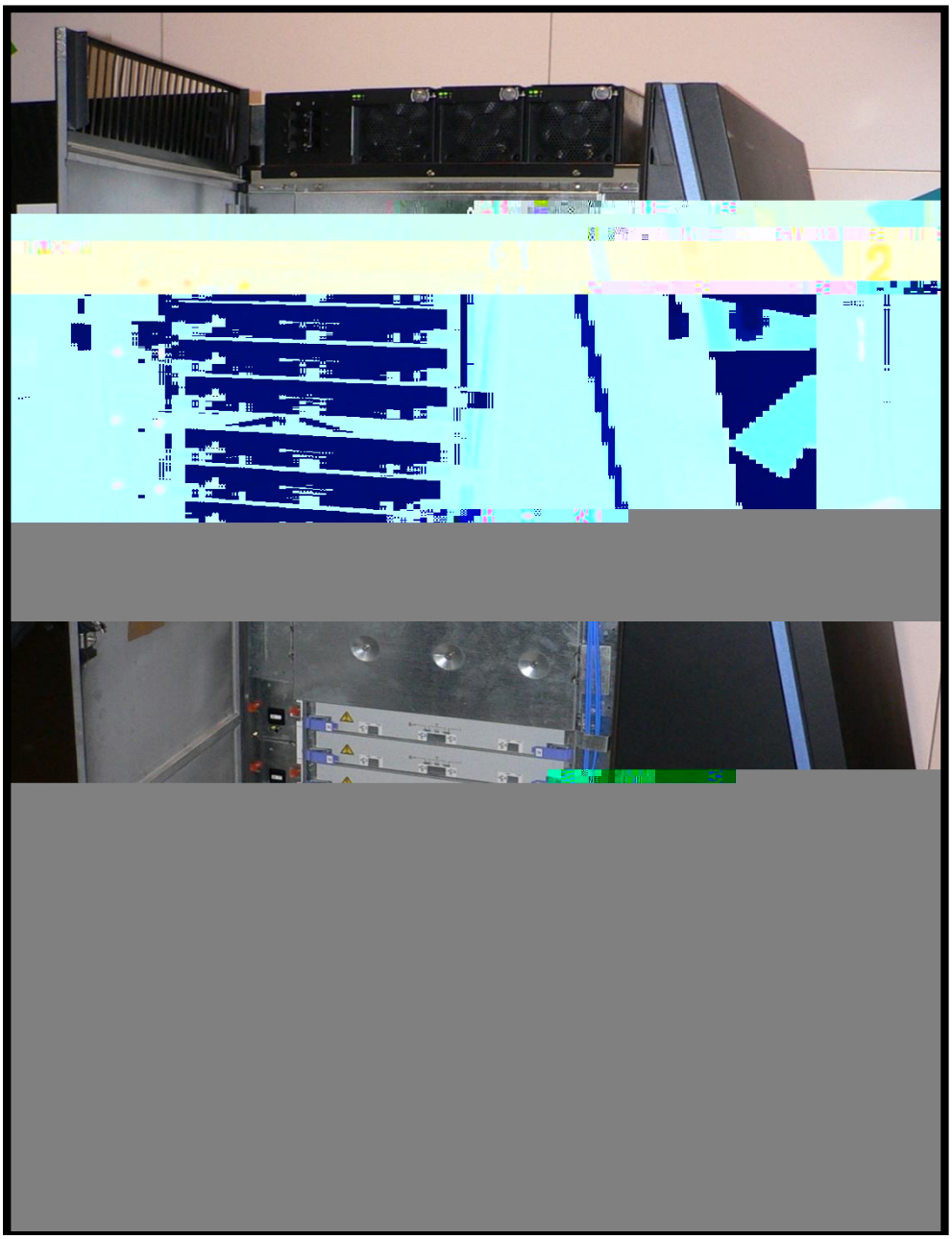
من هنا نستنتج *عزيزي الدارس*،، أنه إذا كان لدينا نظام به عدد حواسيب كبير (يفوق المائة جهاز مثلاً)، فلا بد من تغيير شكل الشبكة (network topology) من شكل الناقل (bus) إلى شل آخر يت ل عدداً كبيراً من

مثال

الحاسوب العملاق Blue Gene/L، تم تصميم في 1999 كحاسب كثيف متوازي لحل المسائل المعقدة التي تحتاج قوة معالجة كبيرة في مجال النشاطات البيولوجية، وتطوير أنواع جديدة من العقارات، وقد كان ذلك في 2003، وكان يتكون من 16,384 معالج بسرعة حوسبة (computational speed) تساوي 71 تيرافلوب (teraflops).

ثم تطور هذا الحاسوب بعدد 65,536 معالج في 2005، ووصل إلى سرعة 280.6 تيرافلوب.

ربط المعالجات مع بعضها (connection topology) كانت ثلاثية الاتجاهات. فكل معالج مرتبط بستة معالجات أخرى، وتسمى الروابط "شمال"، "شرق"، "جنوب"، "غرب"، "فوق"، و "تحت".



شكل رقم (2-6): حاسب عملاق (super computer) يسمى Blue Gene/L

2.1.2 تجمع المحطات Clusters of workstations (COWs)

عزيزي الدارس، هذا النوع عبارة عن حواسيب شخصية عادية مرتبطة بعضها ببعض بشبكة سريعة، مثل (Myrinet). تتميز MPPs على COWs برابط الاتصال، فأداء الرابط في النوع الأول عالي جداً، مقارنة مع رابط النوع الثاني (رغم ان النوع الثاني يعتبر عالياً مقارنة مع الشبكات العادية، مثل الإيثرنت السريع (fast Ethernet)).

تعتبر COWs أقل كلفة من MPPs، لذلك يمكننا القول بأنّ الأداء له ثمنه، فالذي لا يستطيع امتلاك MPPs يمكنه استخدام COWs، ويمكن تقليل تكلفة COWs أكثر إذا استخدمنا شبكة إيثرنت، لأنها أقل بكثير من Myrinet (وكذلك أبطأ بكثير).



شكل رقم (2-7): COWs تستخدم الإيثرنت السريع بجامعة السودان للعلوم والتكنولوجيا

أسئلة تقويم ذاتي



1. عرِّف الأنظمة متعددة الحواسيب **Multicomputer** .
2. اذكر أمثلة لأنظمة تعمل بنظام (**switched multicomputers**).
3. اذكر أمثلة لأنظمة تعمل بنظام الناقل المشترك.
4. كيف تتم معالجة التباين بين الأجهزة؟
5. عرِّف نظام **MPPs** .
6. عرِّف تجمع المحطات (**COWs**).
7. ما الفرق بين **MPPs** و **COWs** ؟

2.2 الأنظمة متعددة الحواسيب المتباينة

Heterogeneous Multicomputer Systems

عزيزي الدارس، هذا النوع من الأنظمة يتألف من حاسبات مختلفة من حيث نوع المعالج، وحجم الذاكرة، واتساع مخارج الدخل والخرج، طرق تمثيل البيانات. وقد يكون من عناصر هذا النظام جهاز متعدد المعالجات (**multiprocessor system**). أيضا قد تتصل أجزاء هذا النظام بشبكات متباينة، وتستخدم نظم تشغيل متباينة، وتعتبر الحوسبة الشبكية (**Grid Computing**) مثالاً لهذا النوع.

1.2.2 الحوسبة الشبكية (Grid Computing)

عزيزي الدارس، تعتبر الحوسبة الشبكية نوعاً من أنواع الأنظمة المتعددة المتباينة، فهي قد تتكون من آلاف الحواسيب المنتشرة في بقاع العالم، والتي تمتلك

الخلاصة

قسمت الوحدة الأنظمة المتعددة إلى:

- أنظمة ذات ذاكرة مشتركة، وتسمى الأنظمة متعددة المعالجات (multiprocessors).

- أنظمة موزعة الذاكرة وتسمى الأنظمة متعددة الحواسيب (multicomputers).

قد تتصل أي من النوعين أعلاه بإحدى الطرق التالية:

مفتاح (switch)- حلقة (ring)- شبكي (grid)- ناقل (bus)- مكعب (cube)- مكعب مركب (hypercube).

عرفت الوحدة الأنظمة المتعددة المعالجات: عبارة عن مجموعة من المعالجات تتشارك في ذاكرة واحدة، بحيث يكون بمقدور أي معالج الوصول مباشرة لهذه الذاكرة المشتركة والتعامل معها.

كذلك عرفت الوحدة الأنظمة متعددة الحواسيب Multicomputer عبارة عن ربط أجهزة حاسوب مع بعضها البعض عن طريق شبكة عالية السرعة.

تطرقت الوحدة إلى أمثلة تعمل بنظام (switched multicomputers) ؛ منها:

نظام: Massively parallel processors (MPPs) التوازي الكثيف للمعالجات حاسب يتألف من آلاف المعالجات التي لا تختلف كثيراً عن معالجات الحواسيب الشخصية، غير أن الرابط بينها يكون عالي الجودة والتصميم، بحيث لا يكون هنالك بطء به (low latency)، و أن يكون متسعاً بما فيه كفاية (high bandwidth).

تجمع المحطات Clusters of workstations (COWs):عبارة عن حواسيب شخصية عادية مرتبطة بعضها ببعض بشبكة سريعة مثل (Myrinet).

بينت الوحدة أن MPPs تتميز على COWs برابط الاتصال، فأداء الرابط في النوع الأول عالي جداً، مقارنة مع رابط النوع الثاني

مسرد المصطلحات

❖ **الأنظمة متعددة المعالجات (multiprocessor)**

عبارة عن مجموعة من المعالجات تتشارك في ذاكرة واحدة، بحيث يكون بمقدور أي معالج الوصول مباشرة لهذه الذاكرة المشتركة والتعامل معها.

❖ **مفتاح تقاطع (cross point switch)**

هو عبارة عن مفتاح يمكن فتحه و غلقه بواسطة جهاز معين، حيث يتم غلق هذا المفتاح في حالة طلب المعالج الوصول إلى الذاكرة.

❖ **الأنظمة متعددة الحاسبات Multicomputer**

هو ربط أجهزة حاسوب مع بعضها البعض عن طريق شبكة عالية السرعة. وقد تكون الحاسبات المرتبطة مع بعضها متشابهة في العتاد ونظم التشغيل، وتسمى متجانسة (homogenous) أو قد تكون مختلفة في العتاد و/أو البرامج، وتسمى متباينة (heterogeneous).

❖ **التوازي الكثيف للمعالجات Massively parallel processors (MPPs)**

نظام حاسوب يتألف من آلاف المعالجات التي لا تختلف كثيراً عن معالجات الحواسيب الشخصية، غير أن الرابط بينها يكون عالي الجودة والتصميم، بحيث لا يكون هنالك بطء به (low latency)، و أن يكون متسعاً بما فيه الكفاية (high bandwidth).

❖ **تجمع المحطات Clusters of workstations (COWs)**

هذا النوع عبارة عن حواسيب شخصية عادية مرتبطة بعضها ببعض بشبكة سريعة، مثل (Myrinet). (تتميز MPPs على COWs برابط الاتصال، فأداء الرابط في النوع الأول عالٍ جداً، مقارنة مع رابط النوع الثاني.)

❖ **الأنظمة متعددة الحاسبات المتباينة Heterogeneous Multicomputer Systems**

هذا **النوع من الأنظمة يتألف من حواسيب مختلفة من حيث نوع المعالج، وحجم الذاكرة، واتساع مخارج الدخل والخرج، وطرق تمثيل البيانات. وقد يكون من عناصر هذا النظام جهاز متعدد المعالجات (multiprocessor system). أيضا قد تتصل أجزاء هذا النظام بشبكات متباينة، وتستخدم نظم تشغيل متباينة. تعتبر الحوسبة الشبكية (Grid Computing) مثال لهذا النوع.**

❖ **الحوسبة الشبكية (Grid Computing)**

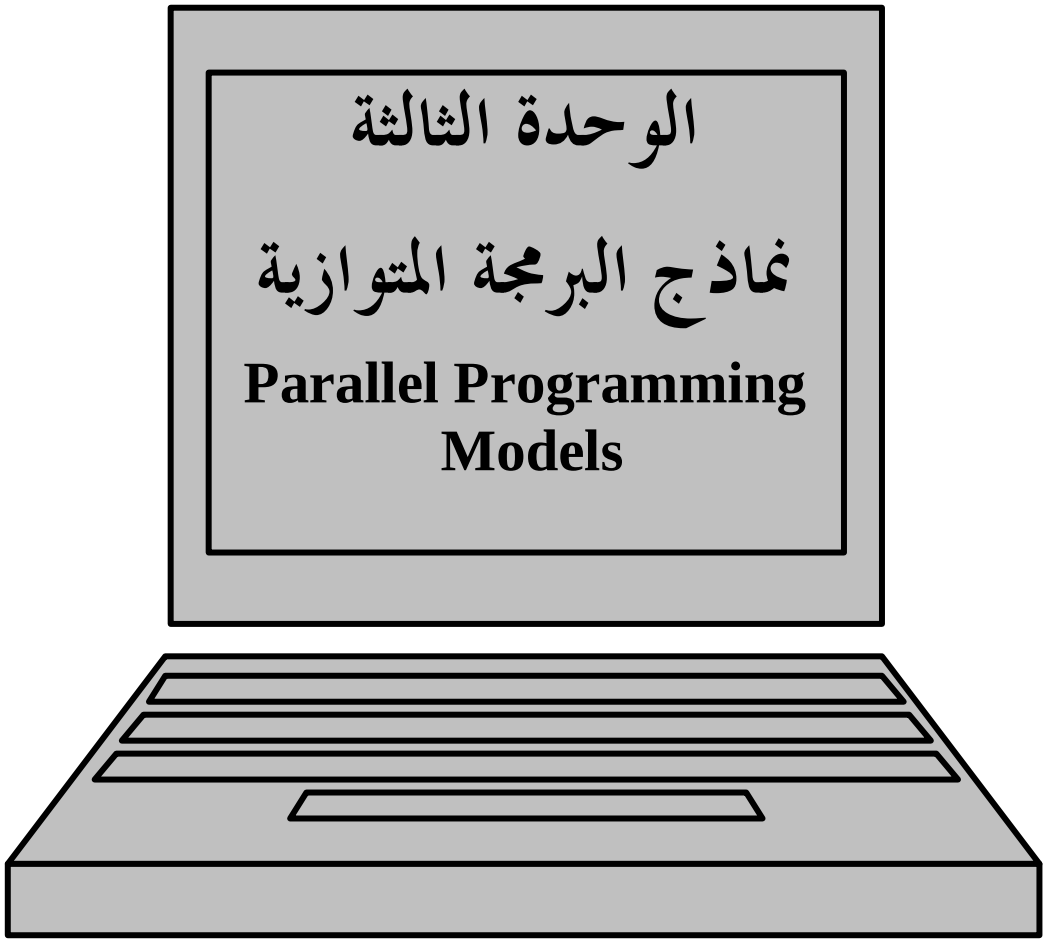
تعتبر الحوسبة الشبكية نوعاً من أنواع الأنظمة المتعددة المتباينة، فهي قد تتكون من آلاف الحواسيب المنتشرة في بقاع العالم، والتي تمتلك بواسطة جهات مختلفة. قد تحتوي هذه الأجهزة على حواسيب عملاقة (super computer)، وتجمع محطات (clusters) وأجهزة مكتبية وأجهزة محمولة (laptop)، وأجهزة جيبية، وهواتف نقالة، وغيرها من الأجهزة. تتصل هذه الأجهزة مع بعضها البعض عبر الإنترنت، أو عبر شبكات واسعة موفرة قوية، وسعة تخزينية عالية جدًّاً.

المراجع

POSIX Standard: www.unix.org/version3/ieee_std.html
"Pthreads Programming". B. Nichols et al. O'Reilly and Associates.

"Threads Primer". B. Lewis and D. Berg. Prentice Hall
Programming With POSIX Threads". D. Butenhof. Addison Wesley
www.awl.com/cseng/titles/0-201-63392-2

Pthreads tutorial , https://computing.llnl.gov/tutorials/pthreads/
POSIX thread (pthread) libraries,
http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html


الوحدة الثالثة
نماذج البرمجة المتوازية
Parallel Programming Models

محتويات الوحدة

الموضوع	الصفحة
المقدمة	49
تمهيد	49
أهداف الوحدة	50
توطئة	51
1. نموذج الذاكرة المشترك	52
2. نموذج الخيوط	53
3. نموذج تبادل الرسائل	55
4. نموذج توازي البيانات	57
5. الهجين	58
الخلاصة	60
لمحة مسبقة عن الوحدة التالية	60
إجابات التدريبات	61
مسرد المصطلحات	63
المراجع	64

المقدمة

تمهيد

مرحباً بك عزيزي الدارس، إلى الوحدة الثالثة من مقرر (أساليب البرمجة3) سنتحدث في هذه الوحدة عن نماذج البرمجة المتوازية، حيث سنتناول نماذج الذاكرة المشتركة في القسم الأول، وفي القسم الثاني نتناول نموذج الخيوط، وفي القسم الثالث نتناول تبادل الرسائل، وفي القسم الرابع نستعرض نموذج توازي البيانات، أما القسم الأخير نتناول فيه نموذج الهجين.

أهداف الوحدة



عزيزي الدارس، بعد فراغك من دراسة هذه الوحدة ينبغـــي أن تكـــون قادراً على أن:

- **تعدد أساليب برمجة الحاسبات.**
- **تذكر أقسام نماذج البرمجة المتوازية.**
- **تشرح مع استخدام الرسم نموذج الخيوط.**
- **تعدد خصائص نموذج تبادل الرسائل.**

توطئة

<u>*عزيزي الدارس،*</u> تحدثنا في الوحدة السابقة من هذا المقرر عن أنواع الحواسيب بشقيها المشتركة الذاكرة والموزعة الذاكرة، وقلنا إن كل نوع يحتاج إلى أسلوب برمجة خاصة به. يمكن تقسيم أنواع الحواسيب من وجهة أساليب البرمجة إلى ثلاثى أنواع رئيسية تختلف في طرق برمجتها؛ هي:

- **الذاكرة المشتركة (الأنظمة متعددة المعالجات (multiprocessor)، ومتعددة النواة (multi-core)).**
- **الأنظمة متعددة الحواسيب المتجانسة (مثل التجمع cluster).**
- **الأنظمة متعددة الحواسيب المتباينة، (مثل الشبكي Grid).**

وفق هذا التقسيم هنالك عدة نماذج برمجية تعمل مع كل نوع، ولكن أحيانا قد يستخدم نموذج برمجي لنوع مع نوع آخر، فمثلا هنالك أمثلة لنظم ذاكرة مشتركة افتراضية (virtual shared memory) تعمل على عتاد بذاكرة موزعة (distributed memory). وقد تجد بعض الأنظمة التي تطبق فكرة تبادل الرسائل (MPI) تعمل على عتاد له ذاكرة مشتركة (shared memory). ولكن ليس هذا هو الأساس.

تقسم نماذج البرمجة المتوازية إلى نماذج :

① الذاكرة المشتركة (shared memory)

② الخيوط (Threads)

③ تبادل الرسائل (message passing).

④ توازي البيانات (data parallel).

⑤ هجين (hybrid).

فيما يلي نلقي نظرة عامة لكل نموذج من النماذج أعلاه.

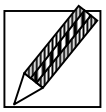
1. نموذج الذاكرة المشترك (Shared Memory Model)

في نموذج برمجة الذاكرة المشتركة تتشارك كل المهام مساحة عنونة مشتركة (common address space)، حيث تتم عمليات القراءة والكتابة بطريقة غير متزامنة (asynchronously). هذا الوصول غير المتزامن قد يسبب بعض المشاكل، لذلك وضعت آليات مختلفة، مثل الغلق أو السيمافورم (locks / semaphores) للتحكم في الوصول إلي الذاكرة المشتركة.

ميزة هذا النموذج من وجهة نظر المبرمج هي عدم الحاجة لاتصال بين المهام لتبادل البيانات. وبالتالي يمكن أن تكون برمجته مبسطة. ولكن عيبه الأساسي بالنسبة للأداء هو صعوبة فهم وإدارة البيانات محليا داخل المعالجات. فمسألة حفظ البيانات محليا في ذاكرة المعالج المخبأة (cache) يقود إلى عدم توافقية هذه البيانات مع البيانات الأصلية في الذاكرة المشتركة. والازدحام الذي قد يحدث في الناقل المشترك المؤدي للذاكرة المشتركة عندما تحتاج عدة معالجات البيانات نفسها في الوقت نفسه. يعتبر التحكم في البيانات ومحليتها (locality) في الذاكرة المخبأة، وضمان توافقيتها مع البيانات الأصلية في الذاكرة المشتركة يعتبر صعب، وليس من اختصاص المبرمج هنا.

في نظام الذاكرة المشتركة تقوم المترجمات بتحويل متغيرات برامج المستخدم إلى عناوين ذاكرة حقيقية (actual memory addresses)، وتكون عامة (global).

تدريب (1)



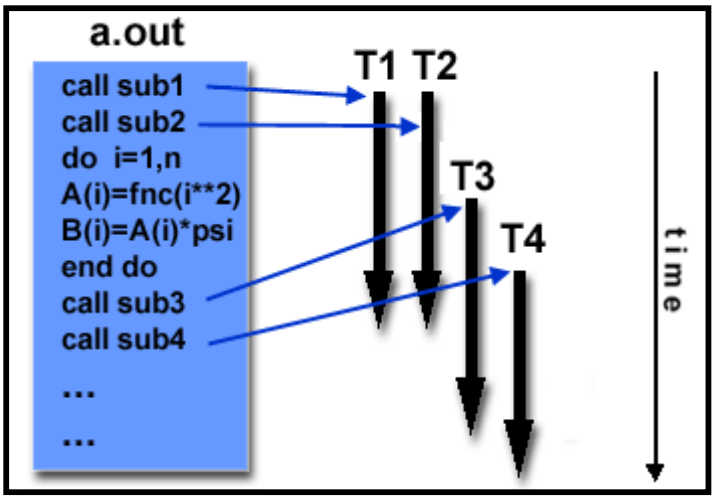
قارن بين نموذج الذاكرة المشترك و نموذج الخيوط.

2. نموذج الخيوط (Threads model)

عزيزي الدارس، في نموذج الخيوط يمكن أن تحتوي العملية الواحدة على عدة خيوط، (عمليات صغيرة) تنفذ بطريقة متزامنة (concurrent). قد تجد تفاصيل أوفي في باب مفهوم الخيوط.

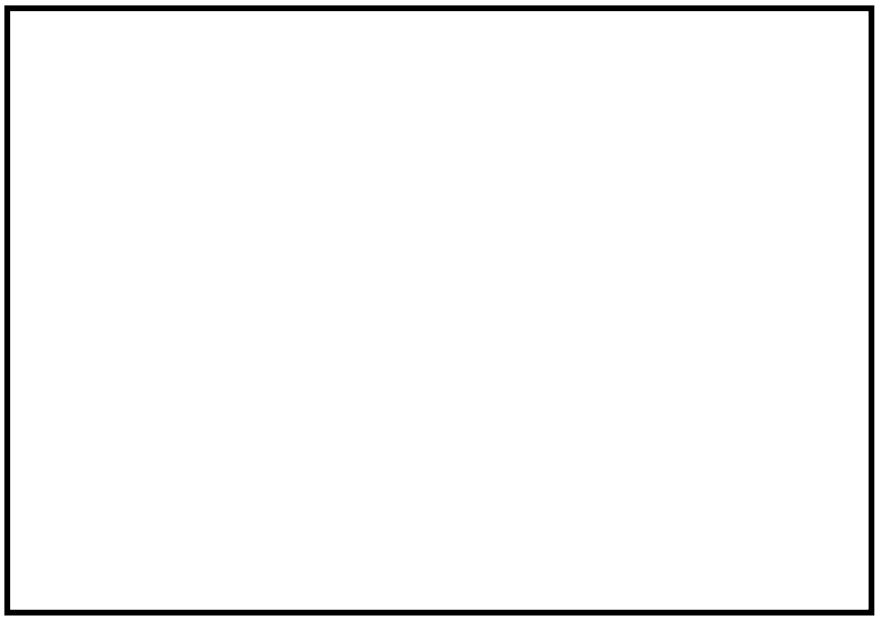
مثال تشبيهي

يمكن تشبيه الخيط ببرنامج يحتوي على عدة أجزاء (subroutines)، حيث يعتبر البرنامج الرئيسي هو a.out (كما موضح قي الشكل أدناه)، الذي سيتم تنفيذه بواسطة نظام التشغيل، فيتم تحميله بعد أن يحصل على كل الموارد التي يحتاج.



شكل رقم(): تشبيه الخيط

ينفذ a.out بعض الأعمال تسلسليا، بعدها ينشئ بعض المهام (الخيوط) التي يمكن لنظام التشغيل جدولتها وتنفيذها بالتوازي (schedule and run concurrently). كل خيط لديه بياناته المحلية، لكن كل الخيوط تتشارك في موارد البرنامج الرئيسي a.out وهذا التشارك يوفر تكرار هذا الموارد المشتركة لكل خيط. وكل خيط يمكنه الاستفادة من مقدرته الوصول إلى الذاكرة العامة، لأنه يتشارك فيها مع البرنامج الرئيسي.



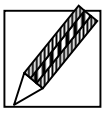
شكل رقم () يوضح نموذج تبادل الرسائل (Message passing)

من وجه نظر برمجية تطبيق تبادل الرسائل يحتاج مكتبة روتين فرعية يجب تضمينها في شفرة البرنامج. وعلى المبرمج تحديد الموازاة (parallelism).

تاريخياً، تتوفر مكتبات لتبادل الرسائل منذ الثمانينات، مثل Parallel Virtual machine (PVM)، وتختلف هذه المكتبات، بحيث لا يمكن للمبرمج تطوير تطبيقات متنقلة (portable).

في 1992 تم وضع الأهداف الأساسية لبدأ تطبيق واجهة تبادل رسائل معيارية Message Passing Interface (MPI). وقد تم إصدار الجزء الأول منها في 1994، ثم أصدر الجزء الثاني منها في 1996. تعتبر الآن MPI هي المكتبة الرئيسية الرسمية (defacto) التي تعتبر عن تبادل الرسائل، وقد سادت وألغت كل المكتبات الأخرى.

تدريب (3)



قارن بين نموذج الخيوط و نموذج تبادل الرسائل.

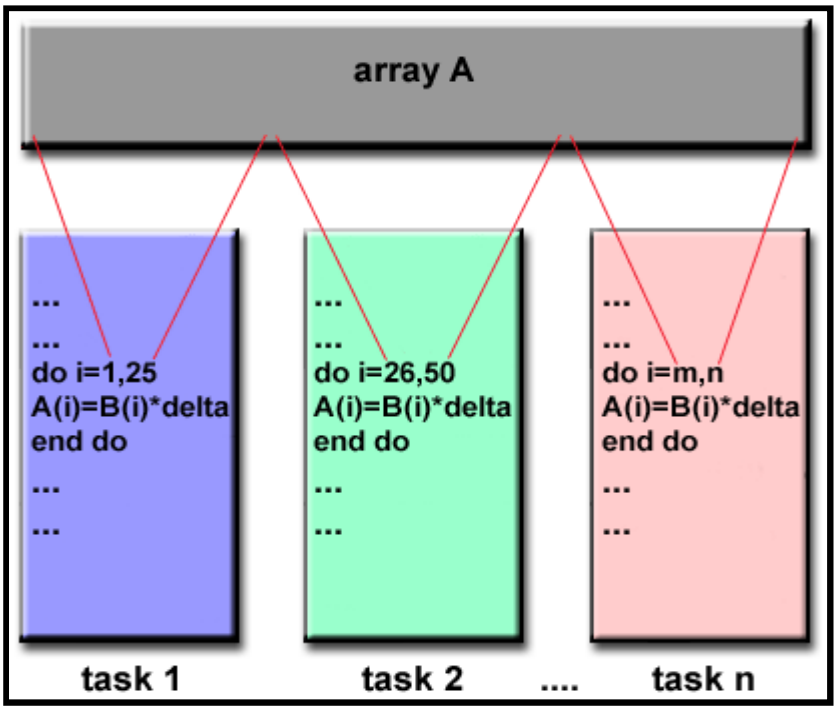
4. نموذج توازي البيانات (Data Parallel Model)

<u>*عزيزي الدارس،*</u> معظم العمل المتوازي يركز على تنفيذ عمليات على مجموعة بيانات مثل المصفوفات أو المكعبات.

هنالك مجموعة من المهام تعمل بشكل جماعي على بنية البيانات نفسها (data structure) لكن كل مهمة تعمل على جزء من هذه البنية. وتنفذ المهام العمل نفسه لكن على الجزء التي تعمل فيه مثلاً إضافة رقم لكل عنصر من عناصر المصفوفة.

بنية البيانات قد توجد في مكان واحد بالذاكرة في نظام الذاكرة المشتركة، ولكن في نظام الذاكرة الموزعة، فيجب وضع كل جزء من البنية في الذاكرة الخاصة التي تنفذ فيها المهمة بها (تقسيم بنية البيانات، وتوزيعها على ذواكر المهام).

البرمجة في توازي البيانات يتم بكتابة برنامج مع بانيات (مشيدات) توازي بيانات (data parallel constructs).



5. الهجين

عزيزي الدارس، يمكن استخدام اثنين أو أكثر من النماذج السابقة. مثلا إذا كان لدينا تجمع مكون من حاسبات ذات معالج متعدد النواة فهنا سيكون التعامل بين محطات التجمع بتبادل الرسائل، ولكن التعامل داخل المحطة الواحد (متعددة النواة) يتم بذاكرة مشتركة. لذلك يمكن في مثل هذا النظام استخدام نموذج هجين بين MPI و OpenMP أو POSIX threads. هذا الهجين أيضا يعتبر مناسب لتعدد المعالجات (multiprocessor).

أسئلة تقويم ذاتي



1. عدِّد نماذج البرمجة المتوازية؟
2. بيِّن العيوب الأساسية لنموذج الذاكرة المشترك.
3. اشرح بالرسم نموذج الخيوط.
4. ما هي الخصائص التي يتم بها تحديد نموذج تبادل الرسائل؟
5. ما هي المكتبة الرئيسية الرسمية والسائدة الآن لتبادل الرسائل؟

الخلاصة

وضحت الوحدة في نموذج برمجة الذاكرة المشتركة تتشارك كل المهام مساحة عنونة مشتركة حيث تتم عمليات القراءة والكتابة بطريقة غير متزامنة.

بينت الوحدة أن ميزة هذا النموذج من وجهة نظر المبرمج هي عدم الحاجة لاتصال بين المهام لتبادل البيانات. وبالتالي يمكن أن تكون برمجته مبسطة. ولكن عيبه الأساسي بالنسبة للأداء هو صعوبة فهم وإدارة البيانات محلياً داخل المعالجات.

وضحت الوحدة في نموذج الخيوط يمكن تحتوي العملية الواحدة على عدة خيوط (عمليات صغيرة) تنفذ بطريقة متزامنة (concurrent).

بينت الوحدة في نموذج تبادل الرسائل يتم تحديده بالخصائص التالية : مجموعة المهام تستخدم الذاكرة المحلية الخاصة بها أثناء التنفيذ ــ يمكن أن توجد عدة مهام في الحاسوب نفسه أو في حواسيب مختلفة ــ يتم تبادل البيانات عبر وسيلة اتصال، وذلك بإرسال واستقبال الرسائل ــ يتطلب نقل البيانات تعاون بين المهام، فمثلا عملية الإرسال من قبل مهمة لا تتم، ما لم تقابلها عملية استلام من مهمة أخرى.

وضحت الوحدة أن معظم العمل المتوازي يركز على تنفيذ عمليات على مجموعة بيانات مثل المصفوفات أو المكعبات.

بينت الوحدة أن هذه النماذج ليس بالضرورة أن تكون حكراً على نظام معين، وإنما يمكن استخدامها مع كل الأنواع. وكذلك يمكن استخدام أكثر من نموذج في نظام واحد.

لمحة مسبقة عن الوحدة التالية

تتناول الوحدة التالية مفهوم الخيوط، وأنواعها، حيث تبين الوحدة الخيوط التي تدار وتدعم بوساطة نظم التشغيل والخيوط التي تدار بوساطة المستخدم. كما تستعرض الوحدة بعضاً من استخدامات الخيط في المعالج الواحد.

إجابات التدريبات

تدريب (1)

في نموذج برمجة الذاكرة المشتركة تتشارك كل المهام مساحة عنونة مشتركة (common address space)، حيث تتم عمليات القراءة والكتابة بطريقة غير متزامنة (asynchronously). هذا الوصول غير المتزامن قد يسبب بعض المشاكل، لذلك وضعت آليات مختلفة، مثل الغلق أو السيمافورم (locks / semaphores) لتحكم في الوصول إلي الذاكرة المشتركة. و ميزة هذا النموذج من وجهة نظر المبرمج هي عدم الحاجة لاتصال بين المهام لتبادل البيانات. وبالتالي يمكن أن تكون برمجته مبسطة. ولكن يبه الأساسي بالنسبة للأداء هو صعوبة فهم وإدارة البيانات محلياً داخل المعالجات

بينما في نموذج الخيوط يمكن أن تحتوي العملية الواحدة على عدة خيوط (عمليات صغيرة) تنفذ بطريقة متزامنة (concurrent). فـــخيوط يمكن أن تتصل مع بعضها البعض من خلال الذاكرة العامة (global memory) بتحديث مواقع العناوين (address locations updating). وذلك يتطلب تزامناً لضمان عدم حدوث عملية تحديث للذاكرة العامة في أي زمن بأكثر من خيط واحد (فقط خيط واحد هو الذي يجب أن يحدث الذاكرة في أي لحظة معينة). فالـــخيوط يمكن أن تنشأ وتلغى، لكن البرنامج الرئيسي يظل باقيا لتوفير الموارد المشتركة للخيوط إلى أن يكتمل عمل التطبيق. وعموماً ترتبط الخيوط بمعمارية الذاكرة المشتركة ونظام التشغيل.

تدريب (2)

من وجه نظر برمجية:

أ. يحتاج تطبيق الخيوط إلى:

- **مكتبة روتينات فرعية (subroutines) يمكن استدعاؤها داخل البرنامج الذي يحتوي الخيوط.**
- **مجموعة من موجهات المترجمات (Compiler directives) تكون مضمنة داخل البرامج المتسلسلة والمتوازية. في كلتا الحالتين المبرمج هو المسئول من تحديد عملية التوازي.**

ب. يحتاج تطبيق تبادل الرسائل مكتبة روتين فرعية يجب تضمينها في شفرة البرنامج. وعلى المبرمج تحديد الموازاة (parallelism).

تدريب (3)

في نموذج الخيوط يمكن أن تحتوي العملية الواحدة على عدة خيوط (عمليات صغيرة) تنفذ بطريقة متزامنة (concurrent). فهنالك خيوط يمكن أن تتصل مع بعضها من خلال الذاكرة العامة(global memory) بتحديث مواقع العناوين (address locations updating). وذلك يتطلب تزامناً لضمان عدم حدوث عملية تحديث للذاكرة العامة في أي زمن بأكثر من خيط واحد (فقط خيط واحد هو الذي يجب أن يحدث الذاكرة في أي لحظة معينة).فالـــخيوط يمكن أن تنشأ وتلغى، لكن البرنامج الرئيسي يظل باقياً لتوفير الموارد المشتركة للخيوط إلى أن يكتمل عمل التطبيق، وعموما ترتبط الخيوط بمعمارية الذاكرة المشتركة ونظام التشغيل.

بينما نموذج تبادل الرسائل يتم تحديده بالخصائص التالية :

1. **مجموعة المهام تستخدم الذاكرة المحلية الخاصة بها أثناء التنفيذ.**
2. **يمكن أن توجد عدة مهام في الحاسوب نفسه أو في حواسيب مختلفة.**
3. **يتم تبادل البيانات عبر وسيلة اتصال وذلك بإرسال واستقبال الرسائل.**
4. **يتطلب نقل البيانات تعاوناً بين المهام، فمثلاً عملية الإرسال من قبل مهمة لا تتم، ما لم تقابلها عملية استلام من مهمة أخرى.**

مسرد المصطلحات

❖ **نموذج الذاكرة المشترك (Shared Memory Model)**

تتشارك كل المهام مساحة عنونة مشتركة (common address space)، حيث تتم عمليات القراءة والكتابة بطريقة غير متزامنة (asynchronously).

❖ **نموذج الخيوط (Threads model)**

تحتوي العملية الواحدة على عدة خيوط (عمليات صغيرة) تنفذ بطريقة متزامنة (concurrent).

❖ **نموذج توازي البيانات (Data Parallel Model)**

تنفيذ عمليات على مجموعة بيانات؛ مثل المصفوفات أو المكعبات.

المقدمة

تمهيد

توطئة

عزيزي الدارس، دعنا نضرب مثلاً تشبيهياً يوضح فكرة الخيوط قبل الدخول في تعريفها ومفهومها واستخدامها. لننظر إلى المسألة الحسابية البسيطة التالية:

5+6+7+8

فإذا طلبنا من أي تلميذ حل هذه المسألة، فإنه سيحلها غالبا في ثلاث خطوات هي :

- **حساب 5+6 والناتج هو 11.**
- **جمع العدد 7 للناتج 13 فيكون الناتج هو 18.**
- **جمع العدد 8 لناتج العملية السابقة 20 فتكون القيمة النهائية هي 26.**

العملية ذات الخيط الواحد (التوالي)

نلاحظ أن الخطوات هذه تتم بالتسلسل الواحد تلو الآخر (بالتوالي). فإذا افترضنا أن كل خطوة تحتاج لثانية من هذا التلميذ فإنه سينجز العمل في (3) ثوان (أي 11=6+5، 18=7+13، 26=8+18). هنا تتم الخطوات بالتوالي، ولا يستطيع التلميذ القيام بخطوتين في وقت واحد، وهذا يشبه العملية العادية (ذات الخيط الواحد).

العملية متعددة الخيوط (التوازي)

الآن لو أُحضر هذا التلميذ أثنين من زملائه ليساعدوه في حل هذه المسألة، فيمكن للتلميذ الأول أن يحسب 6+5، والتلميذ الثاني يحسب 8+7، في وقت واحد (الخطوتين في ثانية واحدة)، ويقوم التلميذ الثالث بجمع النتيجتين ليحصل على القيمة النهائية في ثانية، هنا نجد أن المسألة حُلّت في ثانيتين بدلاً من ثلاث ثوانٍ (أي 33% أسرع من الأولى). نلاحظ أن الخطوتين الأوائل قد تمتا في وقت واحد، لكن الخطوة الثالثة فلا يمكن أن تتم إلا بعد انتهاء الخطوتين السابقتين، لأنها تعتمد علي نتائجهما. من هنا نستنتج :

- **أن القيام بأكثر من عمل في وقت واحد يعني عملية لها أكثر من خيط.**
- **لا يمكن للخيوط خطو ن ي إذا كانت تعتمد على بعضها، لابد من أن**

لا ترتبط الخيوط بأي مورد، لذلك فإن إنشاءها وتدميرها أسهل من إنشاء وتدمير العمليات، وقد يكون إنشاء الخيط في بعض الأنظمة أسرع بمئة مرة من إنشاء العملية.

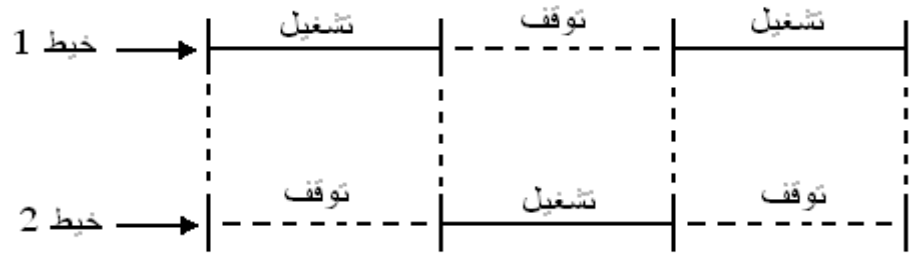
1.1 مثال تشبيهي

إذا تعاملت عزيزي الدارس، مع شركة معمارية، وطلبت منها تنفيذ عمل معين، فأنت في تعاملك مع الشركة لا تهتم ولا تعرف كم عامل سينفذ عملك، فالشركة تشبه العملية، والعمال هم (الخيوط) داخل الشركة (العملية). إذا نفذ العمل عاملٌ واحدٌ فأنت تتعامل مع الشركة كعملية تقليدية ذات خيط واحد (النظام القديم)، فعلى العامل هنا أن ينفذ العمل بالتوالي جزئية تلو الأخرى، أما إذا نفذ العمل عدد من العمال فأنت تتعامل مع شركة (عملية) ذات خيوط متعددة، حيث ينفذ العمال العمل بالتوازي، كل عامل يقوم بجزئية من العمل في وقت واحد. والفرق واضح بين الاثنين.

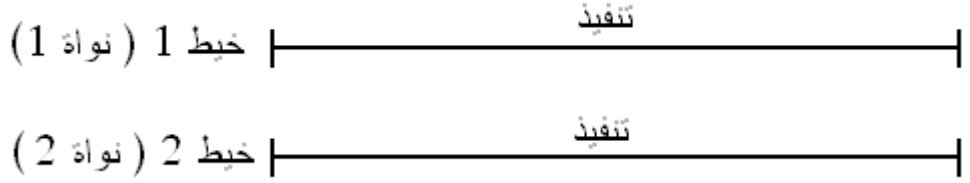
تستخدم معلومات الشركة حينما تتعامل معها (يشبه PCB)، وداخل الشركة توجد معلومات عن كل عامل (معلومات الخيط)، حيث يتشارك كل العمال في عنوان الشركة، ولكن يختلفون في عناوينهم الخاصة، والتي تميزهم عن بعضهم البعض.

2.1 الموازاة الخيالية والموازاة الحقيقية

إذا كنت عزيزي الدارس تمتلك حاسوباً ذا معالج واحد، وصممت برنامجاً يحتوي على خيطين مثلاً، فلن يستطيع المعالج تنفيذ الخيطين في الوقت نفسه حقيقة، وإنما يقوم بتشغيل خيط فترة زمنية، ثم توقيفه، ثم تشغيل الخيط الثاني، وهكذا يبدل بينهما فيما يبدو للمستخدم كأنه موازاة لسرعة التبديل بين تشغيل الخيطين.



لكن لو أردنا تنفيذ برنامج يحتوي على خيوط بصورة متوازية حقيقية، فيجب فعل ذلك على حاسوب متعدد النواة (multi-core) أو متعدد المعالجات (multi-processors). فمثلا لتنفيذ برنامج (عملية) مكون من خيطين في حاسوب ذي معالج ثنائي النواة (dual-core)، فيمكن تنفيذ كل خيط في نواة.



نشاط



1) قم بتنفيذ برنامج متعدد الخيوط على حاسوب ذي معالج واحد مرة، ثم على حاسوب ذي نواتين مرة ثانية وقارن زمن التنفيذ بين الاثنين، ولاحظ فرق سرعة التنفيذ؟

2) ما هو الفرق بين المعالج dual-core، والمعالج core2duo ؟ جرّب تنفيذ البرنامج على النوعين ؟

2.2 خيط النواة

عزيزي الدارس، هنا يتم دعم الخيط مباشرة بواسطة نظام التشغيل، فهو الذي يوفر طرق التعامل مع الخيط من إنشاء وإنهاء، جدولة وإدارة. ولأن إدارة الخيط تتم بواسطة نظام التشغيل، فهذا يجعل خيط النواة بطئياً نوعا ما مقارنة بخيط المستخدم. ولكن إذا تم توقيف خيط لسبب ما، فيمكن لخيوط أخرى في العملية نفسها أن تعمل دون أن تتأثر بحجز رفيقاتها. من أمثلة نظم التشغيل التي تدعم خيوط النواة، نظام التشغيل ويندوز، الذي يمتلك مكتبة تدعم الخيوط هي مكتبة win32. حيث يمكنك استخدام windows.h، ثم إنشاء الخيط بالأمر CreateThread().

هنالك نظم تشغيل تدعم النوعين، خيط المستخدم وخيط النواة.

3.2 خيوط جافا

من أشهر لغات البرمجة التي تطبق مفهوم الخيوط هي لغة جافا، فهي من اللغات التي توفر دعم الخيط على مستوى اللغة، ولها مكتبة كاملة لإنشاء وإدارة الخيوط.

1.3.2 إنشاء الخيط

أبسط طريقة لإنشاء الخيط في الجافا يكون بوراثة فئة الخيط المسماة (Thread) والموجودة بمكتبة جافا. مثلا الأمر التالي:

```
class th extends Thread
```

يمكنك من وراثة الصف Thread، حيث يكون لديك الصف th الذي يعتبر صفّ خيط. يحتوي الصف th على دالة تسمى run، هذه الدالة هي التي تنفذ الخيط، وقد ورثناها من الصف Thread. سنضع ما نريد تنفيذه داخل هذه الدالة، مثلا:

```
public void run(){
    System.out.print("Inside thread"):}
     }
```

بعدها ننشئ كائن من الصف th، مثلا:

```
th t1 = new th( )
```

الآن أصبح لدينا خيط هو t1، يمكن تشغيله بالأمر:

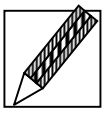
```
t1.start( )
```

✷تنبيه: الخيط t1 يعتبر الخيط الثاني في العملية ذلك لأن العملية في الأساس لديها خيط واحد يعتبر الخيط الرئيسي. فكل عملية تحتوي عادة على خيط رئيسي واحد، إذن الآن لدينا خيطان في العملية. تشغيل خيط العملية الأساسي يتم في main. البرنامج (5-1) هو برنامج صغير يوضح كيفية إنشاء خيط وتشغيله من داخل main.

```
class th extends Thread{//
public void run(){
     System.out.print("Inside thread"):}
}
public static void main(String args[ ]){
      th t1 = new th( ); // إنشاء خيط
      t1.start( ); // تشغيل الخيط
     System.out.print("Inside main");}
                                                    }
```

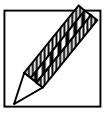
برنامج رقم (5-1)

تدريب (1)



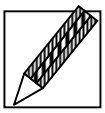
أكتب برنامجاً بلغة الجافا يستقبل من المستخدم اسمه، ثم يطبعه على الشاشة مع رسالة ترحيبية.

تدريب (2)



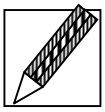
اكتب حلقة لطباعة الأعداد التي تقبل القسمة على (5) بين (1 و 50).

تدريب (3)



اكتب برنامجاً يقرأ من المستخدم عشرة أعداد، ثم يطبعها مرتبة تنازلياً.

تدريب (4)



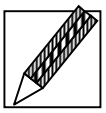
اكتب دالةً تقوم بحساب مضروب عدد صحيح وترجع قيمة المضروب كقيمة من النوع long.

2.3.2 التعامل مع الخيط

توفر جافا طرق (دوال) للتعامل مع الخيط، مثل:

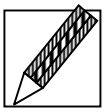
Suspend()	**توقيف الخيط، ويمكن تشغيله مرة ثانية بالأمر Resume()**
Sleep()	**توقيف (تنويم) الخيط لفترة زمنية محددة يعود ليعمل مرة أخرى بعد انقضائها.**
Resume()	**تشغيل الخيط مرة ثانية بعد توقيفه بالأمر suspend().**
Stop()	**إنهاء الخيط (قتله)، حيث يصبح الخيط ميت، ولا يمكن تشغيله مرة ثانية.**

تدريب (5)



اكتب دالةً تستقبل رقم الحد، وتقوم بحساب قيمة الحد في سلسلة فيبوناشي باستخدام النداء الذاتي. قيمة الحد في السلسلة يساوي حاصل جمع الحدين السابقين، وكل من الحد الأول والحد الثاني يساوي (1).

تدريب (6)



اكتب برنامج موضحاً فيه الصيغة العامة للتعامل مع الخيوط Threads وبعض الدوال التي تتعامل مع الخيوط , (Start, Suspend, Resume Stop).

أسئلة تقويم ذاتي



1. **عدِّد أنواع الخيوط، ووضّح الفرق بين كل نوع، مع إعطاء مثال لكل نوع؟**
2. **ما الفرق بين خيط المستخدم وخيط النواة؟**
3. **هل تعتبر Pthreads خيط مستخدم أم نواة؟**
4. **اشرح بصورة مبسطة إنشاء الخيط في لغة الجافا.**
5. **اذكر الطرق (الدوال) للتعامل مع الخيط في الجافا.**
6. **هل يعتبر الخيط في جافا خيط مستخدم؟**

3.3.2 حالات الخيط

للخيط حالات ينتقل بينها أثناء التنفيذ، الشكل (5-2)، هي:

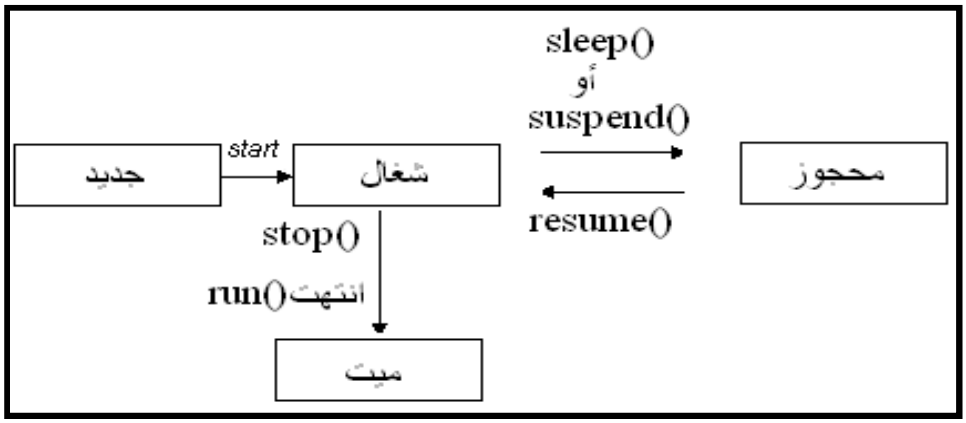
- **جديد (new): عند إنشائه بالأمر new. في البرنامج (5-1)، يعتبر الخيط th في حالة جديد عند استخدام الأمر:**

```
th t1 = new th();
```

- **شغال (منفذ) Runnable: الأمر ()start يحجز مساحة بالذاكرة للخيط، ويستدعى الطريقة ()run بالكائن t1 (في المثال (5-2))، هنا نقول أن حالة الخيط t1 هي Runnable.**

- **محجوز (blocked) أو غير شغال (not Runnable): إذا كان الخيط يقوم بعملية دخل أو خرج، أو تم استدعاء ()suspend أو ()sleep.**

- **ميت (Dead): يصبح الخيط ميتاً إذا انتهى عمل الطريقة (run() أو استدعينا stop().**



شكل رقم (5-2): حالات الخيط.

من الصعب معرفة حالة الخيط، ولكن قد تعرف هل الخيط حي يرزق أم لا باستخدام الطريقة (isAlive()، والتي ترجع قيمة منطقية (نعم (حي)، أم لا (مات)).

هنالك الكثير من الأمثلة البرمجية الموسعة عن برمجة الخيوط في جافا بالملاحق.

أسئلة تقويم ذاتي

1. عدّد أنواع الخيوط ووضح الفرق بين كل نوع مع إعطاء مثال من كل نوع ؟
2. اشرح بصورة مبسطة إنشاء الخيط في لغة الجافا.
3. اذكر الطرق (الدوال) للتعامل مع الخيط في الجافا.
4. اشرح بالرسم حالات الخيط.

3. التحول بين العمليات Context Switch

عزيزي الدارس، لكل عملية بنية بيانات تسمى (PCB)، بغض النظر أن العملية هل فيها خيط واحد أم أكثر. وكل الخيوط الموجودة في العملية الواحدة تشترك في بنية العملية (PCB)، بينما يكون لكل خيط بنيته الخاصة (الشكل (5-3)) التي يستخدمها عندما يتحول من حالة التنفيذ إلى حالة أخرى. بنية معلومات الخيط تتكون من:

- **المسجلات بما فيها مسجل عداد البرامج (PC).**
- **المكدسة (stack).**

نلاحظ هنا أن بنية الخيط صغيرة الحجم مقارنة مع بنية العملية، وبالتالي تحول المعالج بين الخيوط يكون أسرع وأبسط من تحول المعالج بين العمليات.

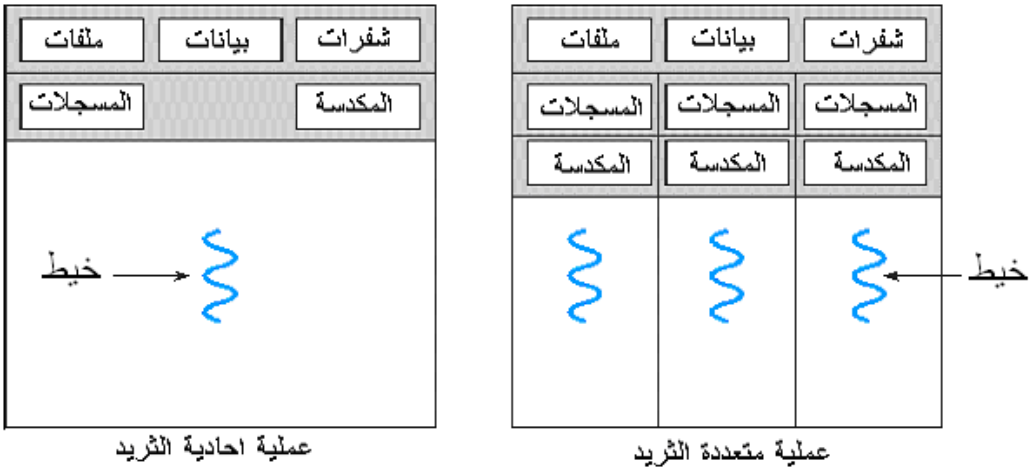
انتقال المعالج بين العمليات أو الخيوط تتم بإخراج عملية من المعالج (توقيفها أو حجزها)، وإدخال عملية أخرى للمعالج للتنفيذ، هنا نقول أن المعالج انتقل من تنفيذ العملية الأولى إلى تنفيذ العملية الثانية. هذا الانتقال أو التحول يسمى (context switching)، ويعتبر مكلفاً زمنياً، فلكي يتم الانتقال لابد من حفظ معلومات العملية (PCB) المنفذة حاليا (المنتقل منها)، وتحميل معلومات العملية (PCB) التي ستدخل المعالج (المنتقل إليها).

ويحدث الأمر نفسه في الانتقال بين الخيوط، لكن الفرق هنا أن معلومات الخيط التي ستحفظ ومعلومات الخيط الآخر التي ستحمل قليلة، وبالتالي فإن الزمن المستغرق لحفظ وتحميل بيانات الخيط أقل بكثير من الزمن المستغرق لحفظ وتحميل بيانات العملية، لذلك يعتبر زمن التحويل بين الخيوط أقل من زمن التحويل بين العمليات.

عملية حفظ بيانات العملية الموقفة وتحميل بيانات العملية المراد تشغيلها يستغرق وقتا من نظام التشغيل، ويسمى زمن التحويل (context switching time).
ويعتمد الزمن المستغرق في التحول من عملية إلى أخرى على العتاد المستخدم.

ملحوظة

في الأنظمة أحادية المعالج (حاسب بمعالج واحد)، فإن المعالج لا يستطيع تشغيل أكثر من عملية في اللحظة الواحدة، ولكن يمكن الاستفادة القصوى من المعالج بتحميل عدد من العمليات في الذاكرة، ثم جعل المعالج ينتقل بين هذه العمليات بسرعة عالية، وكلما احتاجت عملية انتظار حدث يقوم نظام التشغيل (مدير المعالجة) بإخراجها، ثم تشغيل عملية أخرى مكانها وهكذا يظل المعالج مشغولا. بينما يبدو للمستخدم أن المعالج يشغل عدداً من العمليات معاً (تعدد المهام).



شكل رقم (5-3): بنية العملية تكون مشتركة بين الخيوط، ولكل خيط بنيته الخاصة.

أسئلة تقويم ذاتي



1. ما هي مكونات بنية معلومات الخيط التي يحتاج لحفظها عند التحول من حالة التنفيذ إلى حالة أخرى؟
2. وضّح الفرق بين بنية العملية وبنية الخيط.
3. اشرح طريقة انتقال المعالج بين العمليات أو الخيوط.
4. وضّح كيف؛ يمكن الاستفادة القصوى من المعالج في الأنظمة أحادية المعالج (حاسوب بمعالج واحد)، لتنفيذ عدد من العمليات؟

4. استخدامات الخيط في المعالج الواحد

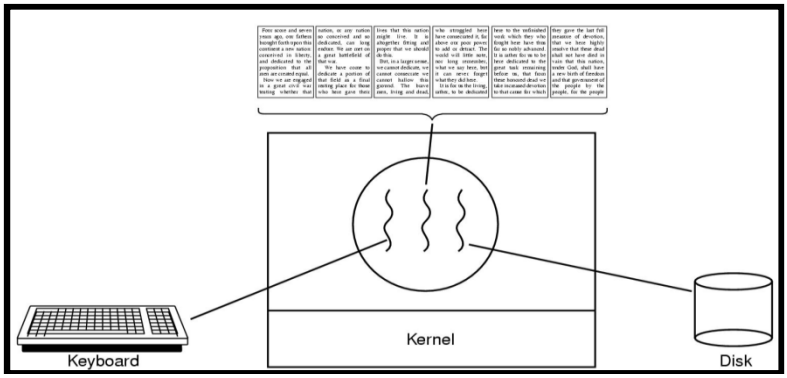
عزيزي الدارس، كل البرامج الحديثة التي نتعامل معها مكونة من مجموعة من الخيوط، فمحرر النصوص، والمتصفح، والرسام، وكل البرامج التي حولك تجدها مكونة من عدد من الخيوط. ذلك لأن معظم التطبيقات بها العديد من الأشياء التي تحدث في وقت واحد.

1.4 محرر النصوص

الشكل رقم (5-4) يوضّح مثال لعملية (برنامج محرر نصوص) يحتوي على ثلاثة خيوط، وكل خيط ينفذ مهمة معينة، فهنالك خيط يستجيب للمدخلات عبر لوحة المفاتيح والماوس، وينفذ الأوامر التي ترد عبرها مثل الانتقال لصفحة معينة. بينما هنالك خيط ثانٍ يقوم بعملية إعادة شكل الوثيقة، فإذا قمت بحذف جملة سيقوم هذا

الخيط بإعادة ترتيب النص كاملاً بعد حذف الجملة، وهنالك خيط ثالث يقوم بعملية الحفظ التلقائي (كل فترة).

مثلاً لو كان برنامج محرر النصوص يعمل بخيط واحد، فلن تستطيع الانتقال إلى صفحة مع ظهور شكل البيانات المعدلة في الوقت نفسه، وعندما يعمل الحفظ التلقائي لابد من توقف كل شيء حتى يكتمل الحفظ.



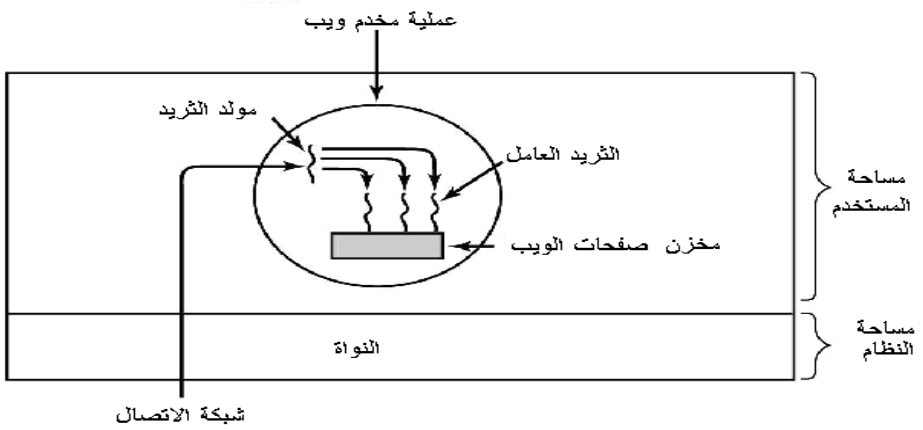
شكل رقم (4-5): برنامج محرر نصوص مكون من ثلاث خيوط.

2.4 مخدم الويب

وظيفة مخدم الويب (web server) هي الرد على طلبات المتصفحين، وإرسال ما يريدون من صفحات إلى أجهزتهم. ويوجد في كل جهاز مخدم ويب على الإنترنت وفيه عدد من المواقع، قد يتصل أكثر من شخص بهذا الجهاز، ويطلب عرض أكثر من صفحة في الوقت نفسه، هنا ينشي مخدم الويب خيط لكل متصل يتحاور معه ويرسل له طلباته.

ماذا لو كان مخدم الويب يعمل بخيط واحد ؟

إذا اتصل مائة عميل بهذا المخدم في وقت واحد، سيكون هنالك خيط واحد لخدمة هؤلاء العملاء، واحد تلو الآخر (بالتتالي)، فإذا افترضنا أن كل طلب يستغرق دقيقة واحدة، فإن العميل رقم مائة ستتم خدمته بعد ساعة وأربعون دقيقة. هل ستستخدم الإنترنت إذا كان المخدم يعمل بهذه الطريقة؟



شكل رقم (5-5): عملية مخدم ويب تنشئ خيطاً لكل طلب.

أسئلة تقويم ذاتي

1. **اشرح مستعيناً بالرسم من خلال أمثلة لبرامج استخدامات الخيط في المعالج الواحد.**

الخلاصة

عرفت الوحدة الخيط بأنه تقنية تسمح لعملية واحدة أن تقوم بعدة مهام في وقت واحد.

عرضت الوحدة بنية معلومات الخيط، والتي تحتوي على – المكدس – السجلات – عدد البرامج – حالة الخيط.

عددت الوحدة أنواع الخيط : خيط المستخدم، ويتم تطبيقه بوساطة مكتبات حيث توفر هذه المكتبات كل ما يتعلق بالتعامل مع الخيط، من إنشاء وإنهاء وجدولة وإدارة – خيوط الجافا وهي من أشهر اللغات التي تطبق مفهوم الخيط وأبسط طريقة لإنشاء

الخيط في الجافا يكون بوراثة فئة الخيط المسماة (Thread). توفر الجافا طرق (دوال) للتعامل مع الخيط.

عددت الوحدة حالات انتقال الخيط أثناء التنفيذ: جديد، شغال (منفذ) محجوز أو غير شغال، ميت

وضحت الوحدة أن لكل عملية بنية بيانات تسمى (PCB)، وتتكون بنية معلومات الخيط من : المسجلات بما فيها سجل عداد البرامج (PC) – المكدسة.

عرفت الوحدة محرر النصوص كواحدة من استخدامات الخيط في المعالج الواحد، ويحتوي محرر النصوص على ثلاثة خيوط: خيط للمدخلات غير لوحة المفاتيح والفأرة – خيط يقوم بعملية إعادة شكل الوثيقة – خيط يقوم بعملية الحفظ التلقائي (كل فترة).

كذلك عرضت الوحدة نموذج مخدم الويب، حيث يوجد في كل جهاز مخدم ويب على الإنترنت مثلاً فيه عدد من المواقع، حيث يقوم مخدم الويب بإنشاء خيط لكل متصل يتحاور معه ويرسل له طلباته.

لمحة مسبقة عن الوحدة التالية

تتناول الوحدة التالية التزامن، وهو كيف لعدة عمليات تعمل معـــاً أن تنشـــق عملها، بحيث لا تؤثر إحداها على الأخرى، ويعتبر مهماً جداً في البرمجة المتوازيـــة، نجد في الوحدة بعض الأمثلة التي تحدث فيها مشاكل نتيجة للإيقـــاف غيـــر المتوقـــع لبعض العمليات بوساطة المجدول.

إجابات التدريبات

تدريب (1)

```
class HelloUser
{
        public static void main (String args[])
        {
            String user;
            user = Input.readString( );
            System.out.println("Hello " + user);
        }
}
```

تدريب (2)

```
class FiveNumbers
{
        public static void main (String args[])
        {
System.out.println("The numbers are:  ");
for (int i=1; i<=50; i++)
{
    if (i%5 ==0)
    System.out.print( i + " ");
  }
}
```

تدريب (3)

```
class TenNumbers
{
    public static void main (String args[])
    {
int large = number[0], largeIndex;
for(int i=1; i<=9; i++)
```

```
        if (number[i]>large)
        {
                large = number[i];
                largeIndex = i;
        }
number [largeIndex] = number[0];
int number[ ] = new int[10];
int large, largeIndex=0;
System.out.println("Enter ten numbers: ");
for (int i=0; i<=9; i++)
{
        System.out.println("Number " + i);
        number[i]=Input.readInt( );
}
for (int k=0; k<=8; k++)
{
        large = number[k];
        for (int i= k+1; i<=9; i++)
        {
                if(number[i]>large)
                {
                        large = number[i];
                        largeIndex=i;
                }
        }
        number[largeIndex] = number[k];
        number[k] = large;
}
System.out.println("The sorted array is: ");
for(int i=0; i<=9; i++)
        System.out.println(number[i]);
}
```

تدريب(4)

```
Class FactNumbers
{
        Public static long factorial(int n)
{
        long fact = 1;
        if((n != 1) && (n != 0))
        {
                for(int i=2; i<= n; i++)
                        fact *= i;
        }
        return fact;
}
        public static void main(String args[ ])
        {
                long num = factorial(5);
                System.out.println("factorial 5 = " + num);
        }
}


}
```

تدريب (5)

```
public static int fibonacci(int f)
{
        int fib = 0;
        if ((f ==1) || (f == 2))
                fib =1;
        else
                fib = fibonacci(f – 1) + fibonacci(f – 2);
        return fib;
}
```

تدريب(6)

```
class th extends Thread{
public void run(){
     System.out.print("Inside thread"):}
}
public static void main(String args[ ]){
      th t1 = new th( ); //إنشاء خيط أول
      th t2 = new th( ); // إنشاء خيط ثاني
      t1.start( ); // تشغيل الخيط الأول
      t2.start( ); // تشغيل الخيط الثاني
      t1. Suspend( ); // توقيف الخيط الأول
      t1. Resume ( ); // تشغيل الخيط الأول مرة ثانية
      t2. Suspend( ); // توقيف الخيط الثاني
      t1. Stop( ); // قتل الخيط الأول
      t2. Resume( ); // تشغيل الخيط الثاني مرة ثانية
      t2. Stop( ); // قتل الخيط الثاني
     System.out.print("Inside main");}
                                                    }
```

مسرد المصطلحات

❖ الخيط (thread)

الخيط هو تقنية تسمح لعمليـــة واحـــدة أن تقـــوم بعـــدة مهـــام فـــي وقـــت واحـــد (concurrently).

❖ الموازاة الخيالية

عند تنفيذ برنامج يحتوي على عدد من الخيوط فـــي حاســـوب ذو معـــالج واحـــد لا يستطيع المعالج الواحد تنفيذ أكثر من خيط في الوقت نفســـه حقيقـــة، وإنمـــا يقـــوم بتشغيل خيط فترة زمنية، ثم توقيفه، ثم تشغيل الخيط الثاني، وهكذا يبدل بينهما فيمـــا يبدو للمستخدم كأنه موازاة لسرعة التبديل بين تشغيل الخيطين.

❖ خيط المستخدم (user thread)

قد يتعامل نظام التشغيل (خارجياً) مع العملية كعمليـــة واحـــدة دون معرفـــة الخيـــوط الداخلية التي توجد فيها، بينما داخلياً تنفذ العملية مجموعة من الخيوط في وقت واحد.

المراجع

POSIX Standard: www.unix.org/version3/ieee_std.html
"Pthreads Programming". B. Nichols et al. O'Reilly and Associates.

"Threads Primer". B. Lewis and D. Berg. Prentice Hall
Programming With POSIX Threads". D. Butenhof. Addison Wesley
www.awl.com/cseng/titles/0-201-63392-2

Pthreads tutorial , https://computing.llnl.gov/tutorials/pthreads/
POSIX thread (pthread) libraries,
http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html


الوحدة الخامسة
التزامن
Senchronization

محتويات الوحدة

الموضوع	الصفحة
المقدمة	96
تمهيد	96
أهداف الوحدة	97
توطئة	98
1. تعاون العمليات	99
1.1 الحجز غير المتوقع للعمليات	101
2. المنافسة	102
1.2 مشاكل المنافسة	103
2.2 حالة السباق والمنطقة الحرجة	105
3. مشاكل التزامن الكلاسيكية	106
1.3 مشكلة القراءة والكتابة	106
1.1.3 مشكلة القراءة والكتابة الأولى	107
2.1.3 مشكلة القراءة والكتابة الثانية	107
3.1.3 مشكلة القراءة والكتابة الثالثة	108
2.3 مشكلة المنتج والمستهلك	109
1.2.3 حل مشكلة المنتج والمستهلك	109
3.3 مشكلة عشاء الفلاسفة	114
4.3 مشكلة مدخني السجار	117
5.3 اللقاء	119
6.3 مشكلة الحلاق النائم	119

الخلاصة	**123**
لمحة مسبقة عن الوحدة التالية	**125**
إجابات التدريبات	**126**
مسرد المصطلحات	**130**
المراجع	**132**

المقدمة

تمهيد

عزيزي الدارس، مرحباُ بك إلى الوحدة الخامسة من مقرر أساليب البرمجة (3) وهي بعنوان التزامن، وهي تعرض لك من خلال ثلاثة أقسام رئيسة؛ يتناول القسم الأول فيها تعاون العمليات، وهنا سنقوم بالإجابة على السؤال التالي لماذا تحتاج العمليات المتوازية للتعاون فيما بينها ؟ كما يتناول القسم الحجز غير المتوقع للعمليات.

القسم الثاني يبحث في حالة المنافسة بين العمليات للوصول للبيانات ويقوم القسم باستعراض بعض الأمثلة التي تحدث فيها مشاكل نتيجة للإيقاف غير المتوقع لبعض العمليات.

أما القسم الثالث والأخير فيتناول مشاكل التزامن الكلاسيكية؛ وهنا يستعرض القسم مشاكل مشهورة، وهي مشكلة القراءة والكتابة (الأولى – الثانية – الثالثة) – مشكلة المنتج والمستهلك – مشكلة عشاء الفلاسفة، مشكلة مدخني السجاير، مشكلة الحلاق النائم.

أهداف الوحدة



عزيزي الدارس، بعد فراغك من دراسة هذه الوحدة ينبغي أن تكون قادراً على أن:

- **تعرف مفهوم التوازي.**
- **تفسر حاجة العمليات المتوازية للتعاون.**
- **تسرد بعض الأمثلة التي تحدث فيها مشاكل نتيجة للإيقاف غير المتوقع لبعض العمليات بوساطة المجدول.**
- **توضح حال السباق والمنطقة الحرجة وطريقة حلها.**
- **تستعرض بعض المشاكل الناجمة عن التزامن، والتي تحدث في العمليات المتوازية.**
- **تحل مشكلة المنتج والمستهلك.**
- **تضع حلاً للمشاكل التالية:**

القراءة والكتابة – المنتج والمستهلك – عشاء الفلاسفة – مدخني السجاير – الحلاق النائم.

توطئة

عزيزي الدارس، إذا كنت تبرمج على حاسب شخصي وتكتب برامج عادية، فسيكون همك الأكبر، هل المخرج أو الناتج الذي يظهره البرنامج هو المطلوب أم لا ؟

ولكن إذا صممت برنامج يتكون من أكثر من عملية، بحيث تنفذ هذه العمليات في معالجات مختلفة في نفس الوقت وتتشارك في البيانات. فستظهر هموم أخرى غير هموم النتائج الصحيحة، مثل؛ هل البيانات متوافقة؟ وهل ما قرأته من معلومات هو آخر ما تم تحديثه، وهل على عملية معينة الانتظار في لحظة معينة حتى يتزامن عملها مع بقية العمليات التي تتشارك معها في البيانات؟ وما إلى ذلك من مشاكل ترتبط بالتنفيذ المتوازي مع التشارك بين البيانات.

مفهوم التوازي Concurrency

نقصد بالتوازي تنفيذ أكثر من عملية في وقت واحد. قد تكون هذه العمليات المتوازية مستقلة عن بعضها البعض (independent processes) أو متعاونة مع بعضها البعض (cooperative processes). مثلا قد أقوم بتشغيل محرر النصوص (WinWord) لأكتب عليه، واستمع إلى مادة صوتية بواسطة مشغل الوسائط (media player)، ويكون المتصفح ينزل ملفات من الإنترنت، هذه العمليات تنفذ بالتوازي في وقت واحد، ولكنها مستقلة عن بعضها البعض.

التوازي قد يطبق على مستوى العمليات (concurrent processes) أو على مستوى الخيوط (concurrent threads).

> **العملية المستقلة هي التي لا تؤثر ولا تتأثر بالعمليات التي تعمل معها في الوقت نفسه. العملية تكون متعاونة إذا أثرت، أو تأثرت بالعمليات الأخرى التي تعمل معها بالتوازي.**

1. تعاون العمليات

عزيزي الدارس، إذا احتاجت العمليات المتوازية أن تتشارك في بعض البيانات، أو أن تؤدي عملاً مشتركاً، فلابد لها من أن تتعاون مع بعضها البعض. هذا التعاون يتطلب اتصال بين العمليات (أو الخيوط) وعملية تزامن (synchronization).

الاتصال بين العمليات يتم عن طريق متغيرات مشتركة (shared variables) أو بتبادل الرسائل (message passing). والتزامن نحتاجه ليتم الاتصال في الوقت المناسب، ويتحقق بوضع نقاط تزامن للعمليات، بحيث إذا سبقت عملية بقية العمليات، ووصلت نقطة تزامن، عليها أن تنتظر بقية العمليات الأخرى لتصل هذه النقطة، وإلا سيكون لدينا نتائج غير متوقعة.

لماذا تحتاج العمليات المتوازية التعاون فيما بينها ؟..... لعدة أسباب، مثل:

- **طلب عملية لخدمة من عملية أخرى: في هذه الحالة على العملية التي طلبت الخدمة انتظار العملية التي ُطلب منها الخدمة حتى تفرغ من أدائها. مثلا إذا طلبت عملية من نظام التشغيل معلومة معينة من القرص الصلب، ستظل هذه العملية في انتظار المعلومة حتى تصلها من القرص إلى الذاكرة، أو إذا طلب المتصفح من مخدم الويب بالإنترنت موقعاً معيناً فعلى المتصفح الانتظار حتى يرسل المخدم الموقع المطلوب، فيقوم المتصفح وقتها بعرضه على المستخدم.**

- **زيادة سرعة التنفيذ: وذلك بتقسيم مهمة واحدة كبيرة على عدة عمليات يتم تنفيذها بالتوازي. قد تسبق عملية في تنفيذها عمليات أخرى معها، فإذا تركنا هذه العملية لتكمل تنفيذها دون انتظار العمليات الأخرى، فقد يتسبب هذا في نتائج غير متوقعة أو غير صحيحة، لذلك لابد من وضع نقاط تزامن لا تتعداها العمليات السريعة، ما لم**

) pipelineتصل بقية العمليات هذه النقاط. مثلاً إذا استخدمنا الأنابيب الإنسيابية (بين عمليات، فهذا يعني أن مخرج العملية الأولى سيكون مدخلاً للعملية الثانية، في هذه الحالة لا يمكن للعملية الثانية أن تسبق العملية الأولى. مثلاً الأمر التالي (في ينكس):

$ ls | wc

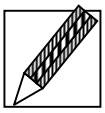
هو أمر لتنفيذ عمليتين؛ هما ls، التي تعرض محتويات الدليل الحالي، و wc لعد الكلمات، هاتان العمليتان ستنفذان في وقت واحد (بالتوازي)، وسيتم بينهم تزامن (ستنتظر العملية الثانية مخرجات العملية الأولى لتعمل عليها (تحسب عدد كلمات المخرج)).

- **قد يكون من الملائم للعمليات أن تعمل مع بعضها لإنجاز العمل، مثلا في نظام الوسائط المتعددة يمكن أن تقوم عملية بإحضار أجزاء المادة الوسائطية من القرص، ووضعها في ذاكرة مؤقتة (خازن (buffer))، بينما عملية أخرى تأخذ المادة من الخازن، وترسلها إلى الجهاز الذي يصدر الصوت/الصورة. هنا تعتبر العملية الأول منتج (producer) بينما العملية الثانية مستهلك (consumer). وعلى المنتج العمل على جعل الخازن ممتلئاً دوماً بحيث لا يتوقف المستهلك عن العمل (لا يجد الخازن فارغاً)، لأن المستهلك إذا وجد الخازن فارغا سيتوقف عن العمل وينتظر، وصول بيانات للخازن، مما يتسبب في تقطع مقطع الفيديو/الصوت أو عدم وضوحه.**

أيا كان نوع التعاون بين العمليات، فيجب أن يكون هنالك:

- **تزامن بينها.**
- **اتصال فيما بينها.**

تدريب (1)



اكتب برنامج بلغة الجافا لعدد من الخيوط يقوم بتعين رقم وحيد لكل خيط، ويتم التوليد بمتغير ساكن (Static Variable)، ويتم عمل الخيط بعداد تنازلي و عندما يصل إلى القيمة صفر يتوقف الخيط عن العمل.

1.1 الحجز غير المتوقع للعمليات

قد يقوم المجدول (في نظام التشغيل) بحجز أي عملية في أي وقت ولأي سبب، هذا الحجز سيوقف العملية التي تنفذ حاليا في المعالج، ويدخل عملية أخرى مكانها (بالطبع سيكون هنالك تحول (context switching)، حيث يتم حفظ معلومات العملية المحجوزة، ويتم تحميل معلومات العملية التي سيتم تنفيذها).

لا يستطيع المبرمج معرفة متى سيقوم المجدول بحجز برنامجه (عمليته)؟

هذا يعني أن البرنامج قد يتوقف في أي وقت (عندما يحجز)، ثم فيما بعد قد يواصل من آخر نقطة أُوقف فيها (بعد فك حجزه). قد يتسبب هذا الحجز في مشكلة إذا كانت العملية المحجوزة ضمن عمليات متعاونة.

أسئلة تقويم ذاتي



1. **ما هو مفهوم التزامن ؟**
2. **ما الفرق بين العملية المستقلة والعملية المتعاونة ؟**
3. **لماذا تحتاج العمليات المتوازية التعاون فيما بينها ؟**
4. **تتصل العمليات المتعاونة فيما بينها بطريقتين؛ ما هما ؟**

2. المنافسة (competition)

عزيزي الدارس، قد تكون للعمليات المتعاونة بيانات مشتركة تستطيع الوصول إليها. إذا لم يكن هنالك تحكم في طريقة الوصول لهذه البيانات ستكون النتيجة وصول غير منظم لها أو تنازع حولها وبالتالي نتائج غير صحيحة. هذا الوصول غير المنظم للبيانات المشتركة يسمى حالة السباق (race condition)، حيث تتسابق العمليات في تغيير قيمة البيانات المشتركة.

أيضا قد تكون العمليات مستقلة لكنها تتصل مع بعضها البعض لاستخدام موارد مشتركة، مثلاً افترض أن لدينا عمليتان، كل عملية تريد طباعة ملف على الطابعة (المشتركة)، لابد لهما من الاتصال بينهما ليقررا من يستخدم الطابعة أولا، وعلى العملية الثانية الانتظار حتى تفرغ الأولى من الطباعة. يسمى هذا تنافس العمليات (process competition).

1.2 مشاكل المنافسة

عزيزي الدارس، فيما يلي نسرد بعض الأمثلة التي تحدث فيها مشاكل نتيجة للإيقاف غير المتوقع لبعض العمليات بواسطة المجدول.

- مشكلة تعديل ملف على الخادم الملفات (file server)

افترض أن لدينا شبكة حواسيب، حيث يتم تخزين كل ملفات المستخدمين في خادم الملفات. إذا كان هنالك مستخدمان يريدان تعديل ملف مشترك بينهما في الوقت نفسه، سيقوم كل مستخدم بفتح الملف في جهازه (نسخة من الملف)، ثم يقوم كل منهما بتعديل نسخته، هذا التعديل لن يتم على الملف الأصيل بالخادم وإنما في نسخة كل مستخدم. وعندما يقوم المستخدم بالحفظ سيغير ذلك في الملف الأصلي. المشكلة هي أنه عندما يقوم المستخدم الثاني بحفظ نسخة ملفه في الخادم سيكتب فوق تعديلات المستخدم الذي حفظ ملفه أولا (overwrite). هذه المشكلة ينتج عنها مخرجات خاطئة.

الصحيح هو أنه عندما تعمل عمليتان في وقت واحد يجب أن تكون النتيجة متشابهة بغض النظر عن أي عملية انتهت أولاً. ولكن عندما يؤثر ترتيب تنفيذ العمليات في النتيجة يسمى هذا حالة السباق (race condition). لأن ذلك يمثل سباق بين العمليات لنعرف من ينتهي أولا.

- مشكلة نظام الحجز الموزع

إذا كان هنالك نظام حجز على خطوط طيران معينة، يعمل على عدة فروع. وكان هنالك مقعد واحد فارغ في رحلة ما، وجاء عميل لحجز هذا المقعد في الفرع (أ)، وفي الوقت نفسه كان هنالك عميل بالفرع ب، يريد حجز المقعد نفسه، وقام الموظفان في الفرعين بعملية طلب حجز للمقعد في الوقت نفسه، ماذا يحدث ؟

العملية الأولى في الفرع (أ) قامت بالتأكد من وجود مقعد فارغ، وهذا ما فعلته أيضا العملية التي نفذت في الفرع (ب)، فالعمليتان قامتا باختبار وجود مقعد فارغ، ثم

أجاب النظام للعمليتين "بنعم يوجد مقعد واحد فارغ"، في هذه الأثناء قام المجدول (بنظام التشغيل) بتوقيف إحدى العمليتين لسبب ما، فقامت العملية الأخرى بحجز المقعد، ثم بعد فك توقيف العملية الأولى ستكمل عملها وتنفذ الأمر الذي يلي اختبار وجود مقعد فارغ (فهي اختبرت المقعد، ووجدته فارغاً قبل توقيفها). فتقوم العملية بحجز المقعد الذي حجز من قبل، وبهذا يتم حجز المقعد مرتين.

- مشكلة الحساب المشترك

إذا كان هنالك حساب مفتوح ببنك ما، وكان هذا الحساب مشترك بين عميلين، فقد يتفق وأن يطلب العميلان سحب مبلغ من المال من الحساب المشترك في وقت واحد من فرعين مختلفين. هنا ستنفذ عمليتين على الحساب المشترك، حيث ستقوم العمليتين باختبار الرصيد (نفترض أنه كان 1500 جنيه)، ثم إذا أوقفت إحدى العمليتين بواسطة المجدول لسبب ما، ونفذت العملية الأخرى سحب المبلغ المطلوب (مثلا 100 جنيه)، سيكون الرصيد الآن 1400 جنيه. بعد أن يتم فك حجز العملية الثانية ستواصل من بعد آخر أمراً قد نفذته، وهو اختبار الرصيد (1500 جنيه)، وستنفذ السحب، وتخصم المبلغ من الرصيد (مثلا إذا كان المبلغ المراد سحبه هو 200 جنيه، فسيكون الرصيد المتبقي هو (1300 جنيه). وتعتبر هذه العملية خطأ، ولو عمل البنك بهذه الطريقة سيغلق أبوابه قريبا.

النتيجة الخاطئة التي وصلنا إليها كان سببها الوصول غير المنظم لقاعدة البيانات، وإجراء تعديل على الحساب المشترك بصورة غير منسقة (غير تزامني) مما نتج عنه خطأ يسمى حالة السباق (race condition). حل مشكلة مثل هذه يتم بالتأكد من أن عملية واحدة فقط في لحظة معينة تقوم بتعديل البيانات المشتركة.

2.2 حالة السباق (*race condition*) والمنطقة الحرجة (critical section)

المشاكل التي سردناها مسبقا معظمها تعاني من حالة السباق، ولنوضح حالة السباق وطريقة حلها سنأخذ المثال التالي:

إذا كان لدينا متغير x، مشترك بين عمليتين، وقامت كل عملية بالآتي:

- **قراءة قيمة المتغير x (read x).**
- **زيادة قيمة المتغير بواحد (x=x+1).**
- **حفظ قيمة المتغير الجديدة (write x).**

المشكلة ستظهر عندما تقوم أحد العمليات بقراءة قيمة المتغير، ثم تقوم العملية الثانية بقراءة قيمة المتغير قبل أن تقوم العملية الأولى بحفظ القيمة الجديدة في المتغير. فستكون النتيجة النهائية بعد تنفيذ العمليتين أن المتغير سيزيد واحداً بدلاً من 2.

الحل هو أن نمنع أي عملية أخرى من الوصول للمتغير x إذا كانت هنالك عملية تستخدم هذا المتغير.

المنطقة التي تتعامل مع المتغير المشترك في العملية نسميها المنطقة الحرجة (critical section). ستكون منطقة المتغير المشتركة هي المنطقة الحرجة، ولحل مشكلة حالة السباق يجب أن نتأكد من أن هنالك عملية واحدة تنفذ داخل المنطقة الحرجة. ولا يسمح للعملية الثانية بالدخول إلى المنطقة الحرجة، إلا بعد خروج العملية الأولى منها.

أسئلة تقويم ذاتي

1. ما المقصود بالمنافسة (process competition) ؟
2. ما المقصود بحالة السباق (race condition) ؟ وما هي مسبباتها؟ وكيف يتم حلها؟
3. قدّم واحداً من الأمثلة التي تحدث فيها مشاكل نتيجة للإيقاف غير المتوقع لبعض العمليات بواسطة المجدول.

3. مشاكل التزامن الكلاسيكية

عزيزي الدارس، هنا سنتطرق إلى بعض المشاكل الناجمة عن التزامن والتي تحدث في العمليات المتوازية، وهي مشاكل مشهورة، وتستخدم لاختبار أي نظام جديد تزامني مقترح.

1.3 مشكلة القراءة والكتابة (*reading and writing problem*)

في مشكلة الحجز الموزع ومشكلة الحساب المشترك بالبنك، تعتبر عملية التعديل على قاعدة البيانات، عملية كتابة، بينما عملية الحصول على معلومات من قاعدة البيانات تعتبر قراءة. مثلا معرفة المقاعد الفارغة في رحلة طيران، أو معرفة الرصيد لحساب بنك هي عمليات قراءة، بينما سحب مبلغ من الرصيد أو حجز مقعد في رحلة طيران هي عمليات كتابة.

إذا تم تنفيذ عمليتي كشف حساب لمعرفة الرصيد من حساب مشترك، فلن يسبب هذا مشكلة، لأننا نريد القراءة من قاعدة البيانات (يسمح بأكثر من عملية قراءة

في ذات الوقت). ولكن لا يسمح بتنفيذ أي عملية على البيانات المشتركة إذا كانت هنالك عملية كتابة تعمل على هذه البيانات، فإذا بدأت عملية في تعديل البيانات المشتركة، فعلى كل العمليات الأخرى، بما فيها عمليات القراءة، التوقف والانتظار حتى تفرغ هذه العملية من عملها.

توفر نظم قواعد البيانات، مثل أوراكل وأكسس إمكانية غلق السجلات التي نجري عليها التعديل (lock records)، واستخدامها بصورة احتكارية (exclusively) حتى نمنع الوصول إليها من قبل أي عملية أخرى أثناء التعديل، وبهذا نضمن توافقية البيانات (consistency).

إذا كانت قاعدة البيانات مشتركة، فلابد لعملية الكتابة من الوصول للبيانات المشتركة بصورة احتكارية (exclusive access).

إذن استخدام الاحتكار يعتبر أحد الحلول لمشكلة القراءة والكتابة.

1.1.3 مشكلة القراءة والكتابة الأولى

The first readers-writers problem

إذا كانت هنالك عملية كتابة، فلن نسمح لأي عملية أخرى بالوصول للبيانات المشتركة، ولكن إذا كانت العملية التي تتعامل مع البيانات عملية قراءة، فيمكن لأي عملية قراءة أخرى من الوصول للبيانات المشتركة. أي أن عمليات القراءة تنفذ بمجرد وصولها، بينما يتم توقف عملية الكتابة حتى لا يكون هنالك عملية قراءة.

2.1.3 مشكلة القراءة والكتابة الثانية

The second readers-writers problem

عزيزي الدارس، إذا كانت هنالك عملية قراءة، ثم جاءت عملية كتابة، ستنتظر الأخيرة حتى تفرغ عملية القراءة، في هذا الأثناء إذا جاءت عملية قراءة أخرى

(مسموح لعمليات القراءة أن تعمل مع بعضها)، ولكن ليس من اللائق أن تتخطى هذه العملية عملية الكتابة التي طلبت قبلها استخدام البيانات المشتركة. وإذا سمحنا لعملية القراءة الثانية بالعمل فقد تأتي عملية قراءة ثالثة ورابعة وخامسة وهكذا، مما يتسبب في حرمان عملية الكتابة من العمل (starvation)، لذلك يجب على عملية الكتابة إذا وصلت صف الانتظار ألا تنتظر أكثر من اللازم، وأن تبدأ العمل بمجرد وصولها. أي أن عمليات الكتابة تنفذ بمجرد وصولها، بينما تنتظر عمليات القراءة حتى لا تكون هنالك عملية كتابة.

3.1.3 مشكلة القراءة والكتابة الثالثة

The third readers-writers problem

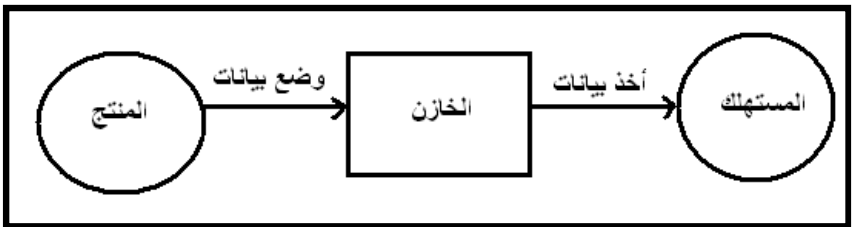
مشكلة القراءة والكتابة الأولى تتسبب في حرمان عمليات الكتابة، لأن عمليات القراءة قد تأتي تباعا وتظل عملية الكتابة في انتظار دائم لا ينتهي.

مشكلة القراءة والكتابة الثانية تتسبب في حرمان عمليات القراءة، لأن عمليات الكتابة عندما تأتي لصف الانتظار يسمح لها بالعمل عند أول فرصة لها، بالتالي قد تأتي عمليات الكتابة تباعا وتستأثر بالوصول للبيانات دون عمليات القراءة.

الحل الثالث جاء ليقول إنه يمنع احتكار الوصول للبيانات إذا تعدى فترة زمنية محددة، ولكل عملية قيد زمني في استخدامها للبيانات لا تتجاوزه، فإذا كانت هنالك عملية تستخدم البيانات احتكارياً فعليها أن تترك هذا الاحتكار، وتنهي عملها إذا تعدى الفترة المقررة للعملية.

2.3 مشكلة المنتج والمستهلك (producer-consumer)

مشكلة المنتج والمستهلك أحيانا تسمى مشكلة الخازن المحدود (bounded-buffer). هنا يكون لدينا عمليتان تستخدمان خازناً مشتركاً، العملية الأولى تضع به بيانات، بينما تأخذ العملية الأخرى البيانات منه. وعلى العمليتين التنسيق فيما بينهما، بحيث لا تحاول العملية الأولى وضع بيانات في الخازن إذا كان ممتلئاً، ولا تحاول العملية الثانية أخذ بيانات من الخازن إذا كان فارغاً.



شكل رقم (6–1): المنتج والمستهلك.

1.2.3 حل مشكلة المنتج والمستهلك

يمكن حل هذه المشكلة بإجبار عملية المنتج على التوقف عن إحضار البيانات للخازن إذا كان ممتلئا (sleep). وعندما يفرغ الخازن تقوم عملية المستهلك من إيقاظ المنتج ليبدأ في تعبئة الخازن. بالطريقة نفسها تتوقف عملية المستهلك (sleep) إذا كان الخازن فارغا، وعندما يحضر المنتج بيانات للخازن، يقوم بإيقاظها لتبدأ في أخذ البيانات من الخازن. يمكن تطبيق هذا الحل باستخدام السيمافور (semaphore) الذي يعتمد على الاتصال بين العمليتين (inter-process communication).

حل يقود إلى اختناق

البرنامج (6–1) يعتبر حلاً للمشكلة، حيث يمثل count عدد العناصر الموجودة بالخازن، و المتغير item يمثل الخازن. هذا البرنامج يعمل كما يلي:
ستختبر عملية المنتج الخازن، هل هو ممتلئ أم لا ؟ وذلك بالأمر:

if (count == BUFFER_SIZE)

إذا كان الأمر صحيحاً سيتوقف المنتج عن تعبئة الخازن، ويذهب لينام (sleep)، ولن يرجع لعمله، ما لم يوقظه المستهلك، أما إذا كان الخازن غير ممتلئ، فسيواصل المنتج في إحضار عناصر للخازن بالأمر:

putItemIntoBuffer(item)

ثم يزيد count بواحد كما يلي:

count = count + 1

المستهلك سيختبر هل الخازن فارغ أم لا؟ بالأمر:

if (count == 0)

إذا كان الأمر صحيحاً سيتوقف عن أخذ البيانات من الخازن، ويذهب لينام، ولن يرجع لعمله، ما لم يوقظه المنتج (عندما يحضر بيانات للخازن). أما إذا كانت هنالك بيانات بالخازن، فسيواصل المستهلك عمله بأخذ عنصر كل مرة من الخازن بالأمر:

item = removeItemFromBuffer()

ثم ينقص count بالأمر:

count = count – 1

البرنامج (6-1)

```
int count
procedure producer() {
   while (true) {
      item = produceItem()
      if (count == BUFFER_SIZE) {
         sleep()
      }
      putItemIntoBuffer(item)
      count = count + 1
     if (count == 1) {
         wakeup(consumer)
      }
   }
```

```
}
procedure consumer() {
   while (true) {
      if (count == 0) {
         sleep()
      }
      item = removeItemFromBuffer()
      count = count - 1
      if (count == BUFFER_SIZE - 1) {
         wakeup(producer)
      }
      consumeItem(item)
   }
}
```

هذا الحل سيقودنا إلى اختناق. لمعرفة كيف يحدث الاختناق دعنا نفترض السيناريو التالي:

- **اختبر المستهلك count فوجده يحتوي على صفر ((count == 0) if)، لذلك سيذهب لينام (sleep).**
- **ولكنه قبل أن يصل فراشه تمت مقاطعته (قام نظام التشغيل بتوقيف المستهلك قبل تنفيذ sleep).**
- **بعدها سيضيف المنتج عنصراً جديداً إلى الخازن (أصبحت count تحتوي 1).**
- **يحاول المنتج إيقاظ المستهلك، وبما أن المنتج أصلاً مستيقظاً فإن طلب الإيقاظ هذا لا فائدة منه (سيفقد).**
- **ولأن المستهلك كان قد اختبر الخازن (قبل إيقافه) ووجده فارغاً، فإنه بعد أن يتم تشغيله مرة أخرى سيواصل من آخر نقطة كان يعمل عندها وهي الذهاب للنوم (استدعاء sleep)، ولن يستيقظ مرة أخرى، لأن أمر الإيقاظ (wakeup) قد فُقد (قام به المنتج عند ما كان count=1).**

- **فيذهب المستهلك لينام، ولن يستيقظ مرة أخرى (فهو لا يعلم أن أمر الإيقاظ قد فقد).**
- **سيستمر المنتج في عمله بإحضار عناصر جديدة إلى الخازن حتى يمتلئ:**

if (count == BUFFER_SIZE)

- **بعدها سيذهب لينام بافتراض أن المستهلك سيوقظه مرة أخرى، ولكن هذا لن يحدث.**
- **وبما أن العمليتين ستكونان في الفراش، وكل عملية غارقة في نومها، وتنتظر الأخرى لتوقظها (ولا يوجد منبه)، فستظلان نائمتين هكذا إلى الأبد، مما ينتج عنه اختناق لا محالة.**

الحل باستخدام السيمافور (semaphore)

لحل مثل هذه المشكلة يمكننا استخدام ما يسمى بالسيمافور. السيمافور هو علامة كانت تستخدم قديما لتنظيم مرور القاطرات، حيث سيكون للسيمافور حالتان، إما تحت أو وفوق، عندما تصل قاطرة وتريد الدخول للمحطة سينظر السائق للسيمافور، فإذا كانت وضعيته تحت، فهذا يعني أن الطريق سالكة (تشبه إشارة المرور الخضراء)، ولأن خط السكة حديد لا يحتمل تلاقي قاطرتين، فسيرفع السيمافور إلى أعلى بعد دخول القاطرة بحيث لا يسمح لقاطرة أخرى أن تدخل (الخط مشغول)، ويشبه هذا إشارة المرور الحمراء.

بالمفهوم نفسه سنستخدم متغيراً يمثل السيمافور (عدداً صحيحاً يحتمل قيمتين)، حيث تمثل قيمة علامة تحت، وقيمة أخرى علامة فوق. ثم إذا بدأت عملية استخدام بيانات مشتركة ستضع القيمة التي تمثل علامة فوق داخل متغير السيمافور، وإذا أرادت عملية أخرى استخدام البيانات المشتركة ستختبر السيمافور وبما أن علامته فوق، فهذا يعني أنه لا يسمح لها باستخدام البيانات المشتركة الآن.

دائما تختبر أي عملية السيمافور قبل الوصول للبيانات المشتركة، واعتماداً على وضعه تتخذ العملية الإجراء اللازم. وعلى أي عملية تستخدم البيانات المشتركة أن تغير وضع السيمافور إلى فوق (ممنوع الاقتراب أو التصوير)، بعد أن تفرغ من البيانات المشتركة ستغير وضع السيمافور، إلى تحت (تصبح البيانات متاحة للعمليات الأخرى). أهم خاصية في نظام السيمافور هو أنه إذا وضعت العملية قيمة فوق في السيمافور، فلا يمكن لعملية أخرى الوصول إلى السيمافور حتى تكتمل العملية، أو يتم توقيفها (حجزها).

في البرنامج (6-2) سنستخدم السيمافور **fillCount**، والسيمافور **emptyCount** لتجنب تجاهل طلبات الإيقاظ، ولحل مشكلة الاختناق.

في هذا البرنامج يغلق المنتج السيمافور **emptyCount**، (لا يستطيع المستهلك تفريغ الخازن في هذه اللحظة)، ثم بعد وضع عنصر في الخازن يفتح السيمافور **fillCount**.

، بحيث لا **fillCount**عندما يريد المستهلك أخذ عنصر من الخازن سيغلق السيمافور يستطيع المنتج التعبئة في هذه اللحظة، ثم بعد أخذ عنصر من الخازن يفتح السيمافور **emptyCount**.

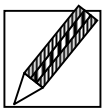
برنامج رقم (6-2)

```
Semaphore fillCount=0
semaphore emptyCount = BUFFER_SIZE
procedure producer() {
   while (true) {
      item = produceItem()
      down(emptyCount)
      putItemIntoBuffer(item)
      up(fillCount)
   }
```

```
 }
procedure consumer() {
   while (true) {
      down(fillCount)
      item = removeItemFromBuffer()
      up(emptyCount)
      consumeItem(item)
   }
```

حل السيمافور مناسب إذا كان لدينا منتج واحد ومستهلك واحد. ولكن إذا كان هنالك أكثر من منتج أو أكثر من مستهلك فإن هذا الحل قد ينتج عنه (race condition).

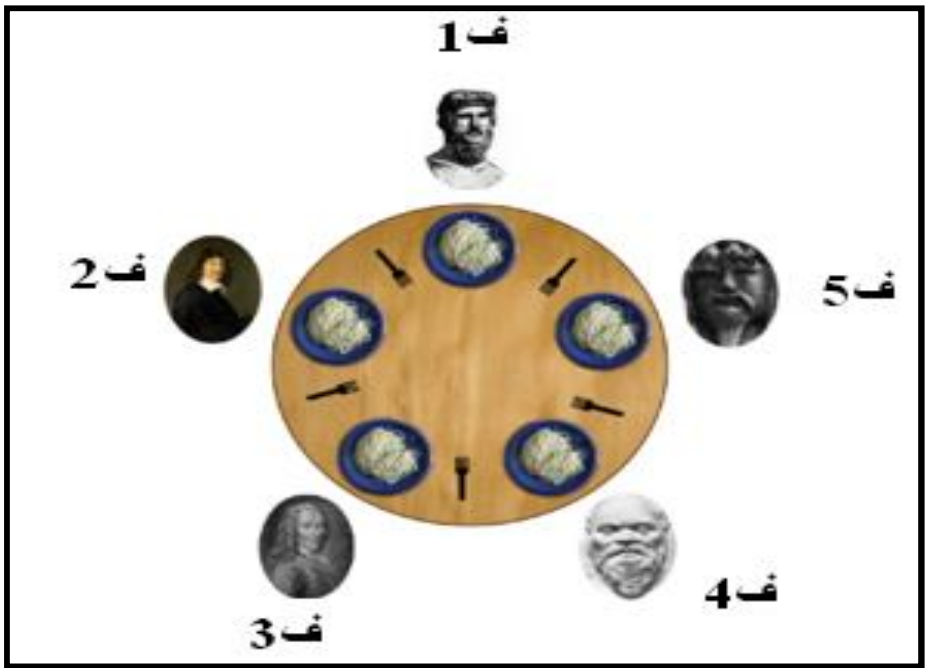
تدريب (2)



اكتب برنامجاً بلغة الجافا لحل مشكلة المنتج والمستهلك، موضحاً فيه مفهوم التزامن، وذلك باستغلال عدد من الخيوط لكائن واحد أو مصدر واحد.

3.3 مشكلة عشاء الفلاسفة (Dining philosophers problem)

هي مشكلة قديمة في التوازي، وهي عبارة عن مشكلة عمليات متعددة غير متزامنة. في عام 1971، أعد العالم ديكاسترا (Edsger Dijkstra) سؤالاً عن التزامن بين العمليات، وهو يتحدث عن (5) حاسبات تتنازع على (5) سواقات أشرطة ممغنطة مشتركة. وبعدها بفترة قليلة أسماها العالم توني هوار (Tony Hoare) مشكلة عشاء الفلاسفة. وهي تمثيل نظري لعملية الاختناق والحرمان، حيث يقوم كل فيلسوف بأخذ شوكة واحدة في البداية، ثم يبحث عن الشوكة الأخرى.



شكل رقم (6-2): عشاء الفلاسفة.

☞ تعريف المشكلة

هنالك (5) فلاسفة يجلسون على طاولة مستديرة، وعلى كل فيلسوف أن يقوم بأحد أمرين، إما أن يأكل أو يفكر. فإذا أكل لا يفكر، وإذا فكر لا يأكل.وهنالك خمسة شوك، شوكة بين كل فيلسوفين، بحيث يكون على يمين كل فيلسوف شوكة، وعلى يساره شوكة. ويحتاج كل فيلسوف أن يستخدم الشوكتين للأكل، ولابد من أن يستخدم الشوكتين اللتين على يمينه ويساره مباشرة، الشكل (6-2). إذا أخذ كل الفلاسفة شوكاتهم اليسرى، عندها لن يستطيع أي منهم الحصول على الشوكة اليمنى، وسيظل كل فيلسوف منتظراً الشوكة اليمنى حتى تفرغ، أي أن الفيلسوف ف1، سيكون في انتظار الفيلسوف ف2 ليترك شوكته، والفيلسوف ف2 سينتظر الفيلسوف ف3 ليترك شوكته، وهكذا (انتظار دائري)، مما يسبب إختناقاً (deadlock).

حل غير مجدٍ :

يمكن حل المشكلة بجعل الفيلسوف يلتقط شوكته اليسرى، ثم يختبر هل الشوكة اليمنى متاحة، فإن لم تكن كذلك يضع الفيلسوف شوكته اليسرى، وينتظر فترة زمنية محددة، ثم يعيد الكرة مرة أخرى. هذه الطريقة قد تفشل إذا قام كل الفلاسفة بإلتقاط شوكاتهم اليسرى في وقت واحد، فلن يجدوا شوكاتهم اليمنى، فسيضعوا شوكاتهم اليسرى، وينتظرون فترة زمنية محددة متساوية لكل الفلاسفة، ثم يلتقطوا شوكاتهم اليسرى مرة ثانية، ولن يجدوا اليمنى، لذلك سيضعو شوكاتهم اليسرى مرة أخرى، وينتظرون فترة زمنية متساوية، وهكذا يظلوا يكررون في هذه المحاولة إلى ما لا نهاية، وتظل المشكلة قائمة. هذا الأمر يحرم جميع الفلاسفة من الأكل (وتحصل مجاعة)؛ أو ما يسمى بالحرمان (starvation).

يمكن حل هذه المشكلة بجعل فترات انتظار الفلاسفة عشوائية، وبالتالي احتمال أنها تكون متساوية قد يكون ضعيفاً جداً، وهذا ما يطبق بالفعل فى كثير من الأنظمة، لكن أحياناً نكون بحاجة إلى حل لا يحتمل الفشل بسبب تطابق الفترات العشوائية. ذلك لأن هنالك من الأنظمة ما تدير أجهزة حساسة لا تقبل احتمال أي فشل؛ مثل أنظمة مراقبة المفاعل النووية.

حل آخر:

هنا نقوم باستخدام سيمافور ثنائي (semaphore)، حيث يقوم الفيلسوف قبل الحصول على شوكة بغلق السيمافور (تحت)، ثم بعد إعادة الشوكة يفتح السيمافور (فوق). هذه الطريقة تمكن فيلسوف واحد فقط أن يأكل في أي لحظة زمنية معينة، ولكن بما أن هنالك خمس شوك فمن المفترض أن يكون هنالك فيلسوفان قادرين على الأكل في اللحظة الواحدة.

حل ثالث:

هنا نتتبع حالة الفيلسوف هل هو:

- **يأكل.**
- **يفكر.**
- **جائع (يريد أن يحصل على شوكة).**

بحيث يستطيع الفيلسوف الجائع أن يأكل فقط إذا كان جاراه لا يأكلان.

4.3 مشكلة مدخني السجاير (Cigarette smokers problem)

هي أحد مشاكل التوازي التي تتحدث عن التزامن بين أربعة برامج. ظهرت في 1971. وقصتها كتالي :

افترض عزيزي الدارس، أن السيجارة تتكون من ثلاث أجزاء هي :

- **تبغ Tobacco.**
- **ورق paper.**
- **عود ثقاب match.**

فإذا كان هنالك ثلاثة مدخنين يجلسون حول طاولة، وكل مدخن لديه مخزون كافٍ من أحد أجزاء السيجارة. وهنالك شخص رابع غير مدخن مهمته اختيار اثنين من المدخنين الثلاثة عشوائيا، ليضع كل واحد من هؤلاء الاثنين قطعة واحدة مما يمتلك من مخزونه على الطاولة، فمثلا إذا اخترنا صاحب التبغ وصاحب الورق، فسيضع كل منهما ما يكفي لعمل سيجارة واحدة. ويطلب من المدخن الثالث عمل السيجارة، فيقوم بأخذ ما في الطاولة، ثم إضافة المكون الذي لديه لعمل السيجارة، مثلا إذا كان المدخن الثالث صاحب الثقاب، فسيأخذ الورقة، ويلف عليها التبغ الموجود بالطاولة، ثم يشعل السيجارة بعود ثقاب من عنده. عند ما تصبح الطاولة فارغة يقوم الشخص الرابع باختيار اثنين من المدخنين عشوائياً لتتم الخطوات السابقة نفسها. ويظل تكرار هذه الخطوات دون توقف. لا يتم تخزين سجاير، وإنما تصنع السيجارة، ثم تدخن مباشرة،

ثم تصنع أخرى وتدخن؛ وهكذا، فليس من المسموح به عمل أكثر من سيجارة في الوقت نفسه. إذا تم وضع ورق وتبغ على الطاولة، وكان صاحب أعواد الثقاب يدخن، ستظل هذه المكونات على الطاولة دون ان يلمسها احد حتى يكمل صاحب الثقاب التدخين، ثم يقوم بعمل السيجارة.

المشكلة تحاكي بأربعة برامج تمثل الأدوار الأربعة (ثلاثة مدخنين و شخص رابع للاختيار). مسموح باستخدام السيمافور للتزامن، وممنوع لأي واحد من البرامج الأربعة بعمل قفز شرطي، فقط يمكن استخدام الانتظار المشروط باستخدام wait.

الحل

- **تحل هذه المشكلة باستخدام سيمافر ثنائي أو mutexes.**
- **نعرف مصفوفة من السيمافورات الثنائية A واحدة لكل مدخن.**
- **سيمافور ثنائي للطاولة T.**
- **هيئ كل سيمافورات المدخنين بقيمة ابتدائية صفر.**
- **وأجعل قيمة سيمافور الطاولة يبدأ بواحد.**

.ستكون شفرة الشخص الذي يختار المدخنين هي :

```
while true{
wait(T)
choose smokers i and j nondeterministically
making the third smoker k
signal(A[k])
                                                    }
```

وشفرة المدخن i هي :

```
{ while true
wait(A[i])
make a cigarette
signal(T)
smoke the cigarette
}
```

5.3 اللقاء (Rendezvous)

نفترض عزيزي الدارس، أن شابين قد تواعدا على أن يلتقيا في منتزه لم يرياه من قبل، بالفعل قد حضر كلّ واحد على حد، ولكنهما اندهشا من كبر حجم المنتزه، وبالتالي صعوبة أن يجد أحدهما الآخر. هنا على كل شاب أن يختار أحد أمرين:

- أن ينتظر في مكان واحد ويتوقع أن الآخر سيبحث عنه.
- أن يبدأ بالبحث بإفتراض أن الآخر سينتظر في مكان واحد.

السؤال هنا أي استراتيجية يجب أن يختارا حتى يكون إحتمال اللقاء أكبر ؟

- إذا قرر الإثنين الإنتظار فبالتأكيد لن يجد أحدهما الآخر أبدا.
- إن قررا أن يبحثا عن بعضهما فهنالك فرصة في أن يتقابلا.
- إذا قرر أحدهما أن ينتظر والآخر أن يبحث فحتما سيلتقيا (من ناحية نظرية).

6.3 مشكلة الحلاق النائم (sleeping barber)

إذا افترضنا أن لدينا صالون حلاقة به حلاق واحد وكرسي حلاقة واحد وعدد من الكراسي للإنتظار. إذا لم يكن هنالك زبائن سيجلس الحلاق في كرسي الحلاقة وينام، وعندما يصل زبون فسيقوم بإيقاظ الحلاق. إذا جاء زبون آخر وكان الحلاق يحلق للزبون الأول فسيجلس الزبون الثاني على أحد كراسي الإنتظار. وكلما يصل زبون سيجلس في كرسي إنتظار إلى أن تمتليء كراسي الإنتظار. إذا جاء زبون ووجد كراسي الإنتظار مشغولة فعليه مغادرة الصالون.

<u>الحل</u>

لحل هذه المشكلة نستخدم ثلاث سيمافورات:

- سيمافور للزبائن المنتظرين
- سيمافور للحلاق (لنعرف هل هو نائم (عاطل) أم يعمل)

- **سيمافور لتحقيق والتأكد من المنع المتبادل (mutual exclusion): بحيث لا نسمح لشخصين بالحلاقة في وقت واحد.**

كلما يحضر زبون سيحاول الحصول على كرسي الحلاقة (mutex)، وسيظل كذلك حتى ينجح. على الزبون الذي يحضر للصالون أن يحسب عدد الزبائن المنتظرين، فإذا كان أقل من عدد الكراسي فسيجلس وإلا فسيغادر (يحاول الحصول على كرسي سواء أكان في غرفة الانتظار أو كرسي الحلاق نفسه).

إذا وجد الزبون كرسيًّاً سيجلس، وبالتالي ينقص عدد الكراسي الفارغة بواحد (critical section).

يقوم الزبون بإرسال إشارة إلى الحلاق لإيقاظه، وسيحرر mutex للسماح للزبائن (أو الحلاق) بالمقدرة على الحصول عليه. إذا كان الحلاق مشغولاً فعلى الزبائن الانتظار. سيظل الحلاق في إنتظار دائم، يتم ايقاظه بأي زبون من المنتظرين، وعندما يستيقظ سيرسل إشارات للزبائن المنتظرين بواسطة السيمافور للسماح لهم بالحلاقة، واحد كل مرة. هذه المشكلة تحتوي على حلاق واحد لذلك تسمى أحيانا (*single sleeping barber problem*). فيما يلي شفرات مبدئية (pseudo-code)،تضمن التزامن بين الحلاق والزبائن خالية من الاختناق، ولكنها قد تقود إلى حرمان الزبون.

نجد P و V هما دالتان توفرهما السيمافورات.

```
Semaphore Customers = 0
Semaphore Barber = 0
Semaphore accessSeats (mutex) = 1
int NumberOfFreeSeats = N // عدد المقاعد الفارغة
```

عملية أو خيط الحلاق:

```
while(true) { // تكرار غير منته
P(Customers) // محاولة الحصول على زبون، وإذا لا يوجد زبائن ينام
```

```
في هذه اللحظة تم إيقاظه، يريد تعديل عدد المقاعد المتاحة // P(accessSeats)
لقد أصبح هنالك مقعد فارغ// ++NumberOfFreeSeats
الحلاق جاهز ليبدأ الحلاقة// V(Barber)
لانريد إغلاق الكراسي // V(accessSeats)
    الحلاق الآن يحلق لزبون //
}
```

عملية أو خيط الزبون:

```
while(true) { // تكرار غير منته
  P(accessSeats) // محاولة الحصول على مقعد
  if ( NumberOfFreeSeats > 0 ) { // إذا كان هنالك مقعد فارغ
    NumberOfFreeSeats-- // إجلس على المقعد
    V(Customers) // أخبر الحلاق، الذي هو في انتظار زبون
    V(accessSeats) // لا نحتاج إلى إغلاق الكراسي
    P(Barber) // الآن حان دور الزبون، لكنه سينتظر إذا كان الحلاق مشغولاً
    // هنا سيتمكن الزون من الحلاقة
  } else { // لا توجد مقاعد فارغة
    V(accessSeats) // حرر إغلاق المقاعد
    // سيغادر الزبون دون أن يحلق
  }
}
```

أسئلة تقويم ذاتي



1. **اشرح مشكلة القراءة والكتابة.**
2. **اذكر خمسة من مشاكل التزامن الكلاسيكية.**
3. **اشرح فكرة عمل السيمافور (semaphore).**
4. **ما هي أهم خاصية في نظام السيمافور؟**

الخلاصة

- وضّحت الوحدة مفهوم التوازي بأنه تنفيذ أكثر من عملية في وقت واحد.
- قدمت الوحدة إجابة عن السؤال؛ لماذا تحتاج العمليات المتوازية التعاون فيما بينها ؟ لعدة أسباب، مثل : طلب عملية لخدمة من عملية أخرى – زيادة سرعة التنفيذ – قد يكون من الملائم للعمليات أن تعمل مع بعضها لإنجاز العمل.
- سردت الوحدة بعض الأمثلة التي تحدث فيها مشاكل نتيجة للإيقاف غير المتوقع لبعض العمليات بوساطة المجدول : مشكلة تعديل ملف على خادم الملفات – مشكلة نظام حجز الموزع – مشكلة الحساب المشترك هذه المشاكل تعاني من حالة السباق والمنطقة الحرجة ولحل مشكلة حلة السباق يجب أن نتأكد من أن هناك عملية واحدة تنفذ داخل المنطقة الحرجة. ولا يسمح للعملية الثانية بالدخول إلى المنطقة الحرجة إلا بعد خروج العملية الأولى منها.
- تطرقت الوحدة على بعض المشاكل الناجمة عن التزامن، والتي تحدث في العمليات المتوازية.
- بينت الوحدة أن استخدام الاحتكار تعتبر أحد الحلول لمشكلة القراءة والكتابة، فإذا كانت قاعدة البيانات مشتركة، فلا بد لعملية الكتابة من الوصول للبيانات المشتركة بصورة احتكارية.
- وضحت الوحدة أن حل السيمافور مناسب لمشكلة المنتج والمستهلك إذا كان لدينا منتج واحد ومستهلك واحد، ولكن إذا كان هناك أكثر من منتج أو أكثر من مستهلك فإن هذا الحل قد ينتج عنه حال سباق.
- عرضت الوحدة أكثر من حل لمشكلة عشاء الفلاسفة وهي حل غير مجدٍ، حيث يجعل الفيلسوف يلتقط شوكته اليسرى، ثم يختبر، هل الشوكة اليمنى متاحة فإن لم

تكن كذلك يطع الفيلسوف شوكته اليسرى، وينتظر فترة زمنية محدودة، ثم يعيد الكرة. هذه الطريقة تؤدي إلى ما يسمى بالحرمان. أما الحل الآخر يجعل فترات الانتظار عشوائية، وبالتالي احتمال أنها تكون متساوية قد يكون ضعيفاً جداً. حل ثالث وهو بتتبع حالة الفيلسوف؛ هل هو : يأكل – يفكر – جائع (يريد أن يحصل على شوكة)؟ بحيث يستطيع الفيلسوف الجائع أن يأكل فقط إذا كان جاراه لا يأكلان.

- كذلك عرضت الوحدة حلاً لمشكلة مدخني السجائر : تحل المشكلة باستخدام سيمافور ثنائي – نعرف مصفوفة من السيمافورات الثنائية A واحدة لكل مدخن – سيمافور ثنائي للطاولة T – هيئ كل سيمافرات المدخنين بقيمة ابتدائية صفر– واجعل قيمة سيمافور الطاولة يبدأ بواحد.

- حددت الوحدة في مشكلة لقاء الشابين أن على كل شاب أن يختار واحد من أمرين – أن ينتظر في مكان واحد، ويتوقع أن الآخر سيبحث عنه – أن يبدأ بالبحث بافتراض أن الآخر سينتظر في مكان واحد.

- قدمت الوحدة ثلاث سيمافورات لمشكلة الحلاق النائم – سيمافور للزبائن المنتظرين – سيمافور للحلاق (التعرف هل هو نائم "عاطل" أم يعمل) – سيمافور للتحقيق والتأكد من المنع المتبادل، بحيث لا تسمح لشخصين بالحلاقة في وقت واحد.

لمحة مسبقة عن الوحدة التالية

تأتي الوحدة التالية تكملة لما سبق لك أن درسته في الوحدات السابقة عن مفهوم الخيوط، التزامن وسوف نتحدث بالتفصيل عن كيفية كتابة برامج متعددة الخيوط تحتوي على بيانات مشتركة وكيف تمنع الوصول المتزامن لهذه البيانات، وعدم الوقوع في حالات السباق.وسوف تقوم الوحدة بتوضيح كيفية استخدام متغيرات **Mutex** والمتغيرات الشرطية لحماية البيانات المشتركة بين الخيوط.

إجابات التدريبات

تدريب (1)

```
// Very simple Threading example.
public class SimpleThread extends Thread {
private int countDown = 5;
private static int threadCount = 0;
private int threadNumber = ++threadCount;
public SimpleThread() {
System.out.println("Making " + threadNumber);
}
public void run() {
while(true) {
System.out.println("Thread " +
threadNumber + "(" + countDown + ")");
if(--countDown == 0) return;
}
}
public static void main(String[] args) {
for(int i = 0; i < 5; i++)
new SimpleThread().start();
System.out.println("All Threads Started");
}
} ///:~
```

لهذا البرنامج سوف تكون على النحو التالي : (سوف (One Run)و النتيجة من عملية تنفيذ واحدة تكون مختلفة من كلي عملية تنفيذ و أخرى)

```
Making 1
Making 2
Making 3
Making 4
Making 5
Thread 1(5)
Thread 1(4)
Thread 1(3)
Thread 1(2)
Thread 2(5)
Thread 2(4)
Thread 2(3)
Thread 2(2)
```

```
Thread 2(1)
Thread 1(1)
All Threads Started
Thread 3(5)
```

```
Thread 4(5)
Thread 4(4)
Thread 4(3)
Thread 4(2)
Thread 4(1)
Thread 5(5)
Thread 5(4)
Thread 5(3)
Thread 5(2)
Thread 5(1)
Thread 3(4)
Thread 3(3)
Thread 3(2)
Thread 3(1)
```

تدريب (2)

```
// multiple threads modifying shared object.
public class SharedCell {
// execute application
public static void main( String args[] )
{
HoldIntegerSynchronized sharedObject =
new HoldIntegerSynchronized();
// create threads
ProduceInteger producer =
new ProduceInteger( sharedObject );
ConsumeInteger consumer =
new ConsumeInteger( sharedObject );
// start threads
producer.start();
consumer.start();
}
} // end class SharedCell
```

و النتيجة سوف تكون على النحو التالي :

```
ProduceInteger setting sharedInt to 1
ConsumeInteger retrieving sharedInt value 1
ProduceInteger setting sharedInt to 2
ConsumeInteger retrieving sharedInt value 2
ProduceInteger setting sharedInt to 3
ConsumeInteger retrieving sharedInt value 3
ProduceInteger setting sharedInt to 4
ConsumeInteger retrieving sharedInt value 4
ProduceInteger setting sharedInt to 5
ConsumeInteger retrieving sharedInt value 5
ProduceInteger setting sharedInt to 6
ConsumeInteger retrieving sharedInt value 6
```

ProduceInteger setting sharedInt to 7
ConsumeInteger retrieving sharedInt value 7
ProduceInteger setting sharedInt to 8
ConsumeInteger retrieving sharedInt value 8
ProduceInteger setting sharedInt to 9
ConsumeInteger retrieving sharedInt value 9
ProduceInteger setting sharedInt to 10
ProduceInteger finished producing values
Terminating ProduceInteger
ConsumeInteger retrieving sharedInt value 10
ConsumeInteger retrieved values totaling: 55
Terminating ConsumeInteger

}

مسرد المصطلحات

❖ **التوازي Concurrency**

نقصد بالتوازي تنفيذ أكثر من عملية في وقت واحد. قد تكون هذه العمليات المتوازية مستقلة عن بعضها البعض (independent processes) أو متعاونة مع بعضها البعض (cooperative processes). التوازي قد يطبق على مستوى العمليات (concurrent processes) أو على مستوى الخيوط (concurrent threads).

❖ **حالة السباق (race condition).**

الوصول غير المنظم للبيانات المشتركة في ظل عدم وجود تحكم في طريقة الوصول لهذه البيانات، حيث تتسابق العمليات في تغيير قيمة البيانات المشتركة.

❖ **تنافس العمليات (process competition).**

عندما تكون العمليات مستقلة لكنها تتصل مع بعضها البعض لاستخدام موارد مشتركة، فعندما يحدث أن هنالك أكثر من عملية تريد استخدام أحد هذه الموارد المشتركة، لابد لهذه العمليات من الاتصال فيما بينها ليقرر من يستخدم هذا المورد المشترك أولاً؟ وعلى باقي العمليات الانتظار حتى تفرغ الأولى من المورد. يسمى هذا تنافس العمليات (process competition).

❖ **السيمافور (semaphore)**

السيمافور هو علامة أو وسيلة كانت تستخدم قديما لتنظيم مرور القاطرات.

استخدم المفهوم نفسه أو الفكرة في مجال البرمجة. فالسيمافور هو متغير يمثل (عدد صحيح يحتمل قيمتين)، حيث تمثل قيمة علامة تحت وقيمة أخرى علامة فوق. ثم إذا بدأت عملية استخدام بيانات مشتركة ستضع القيمة التي تمثل علامة (فوق) داخل متغير السيمافور، وإذا أرادت عملية أخرى استخدام البيانات المشتركة

ستختبر السيمافور، وبما أن علامته فوق، فهذا يعني أنه لا يسمح لها باستخدام البيانات المشتركة الآن. وبعد أن تفرغ العملية التي بدأت أولاً من استخدام البيانات المشتركة ستغير وضع السيمافور إلى تحت (تصبح البيانات متاحة للعمليات الأخرى). أهم خاصية في نظام السيمافور، هو أنه إذا وضعت العملية قيمة (فوق) في السيمافور، فلا يمكن لعملية أخرى الوصول إلى السيمافور حتى تكتمل العملية أو يتم توقيفها (حجزها).

المراجع

POSIX Standard: www.unix.org/version3/ieee_std.html
"Pthreads Programming". B. Nichols et al. O'Reilly and Associates.

"Threads Primer". B. Lewis and D. Berg. Prentice Hall
Programming With POSIX Threads". D. Butenhof. Addison Wesley
www.awl.com/cseng/titles/0-201-63392-2

Pthreads tutorial , https://computing.llnl.gov/tutorials/pthreads/
POSIX thread (pthread) libraries, http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html


الوحدة السادسة
برمجة الخيوط باستخدام
Pthreads

محتويات الوحدة

الموضوع	الصفحة
المقدمة	136
تمهيد	136
أهداف الوحدة	137
1. مكتبات POSIX thread (pthread)	138
1.1 تصميم برامج متعددة الخيوط	139
2.1 برمجة الخيط بأمان	141
3.1 واجهة التطبيقات البرمجية	142
2. إنشاء وإنهاء الخيوط	143
1.2 إنشاء خيط	143
2.2 إنهاء خيط	144
3.2 البروتينات المستخدمة في إنشاء وإنهاء الخيوط	144
1.3.2 إنشاء الخيط	145
2.3.2 صفات الخيط	146
3.3.2 إنهاء الخيط	146
4.2 تمرير قيم إلى الخيط	149
3. إنضمام وانفصال الخيوط	152
1.3 الإنضمام	153
2.3 الانفصال	154
4. إدارة المكدسة	158
1.4 الوقاية من مشاكل المكدسة	158

5. متغيرات الميوتكس	162
1.5 خطوات استخدام الميوتكس	163
2.5 إنشاء وتدمير الميوتكس	164
1.2.5 طريقة الاستخدام	164
3.5 غلق وفتح الميوتكس	165
6. متغيرات الحالة	165
1.6 إنشاء وتدمير متغيرات الحالة	172
1.1.6 طريقة الاستخدام	174
2.6 الانتظار والتأثير في متغيرات الحالة	174
1.2.6 طريقة الاستخدام	175
الخلاصة	175
لمحة مسبقة عن الوحدة التالية	181
إجابات التدريبات	184
مسرد المصطلحات	188
المراجع	190

المقدمة

تمهيد

عزيزي الدارس، مرحباً بك إلى الوحدة السادسة وهي بعنوان برمجة الخيوط باستخدام Pthreads تتكون هذه الوحدة من ستة أقسام رئيسة: القسم الأول يبحث في استخدام مكتبات pthreads التي تعمل مع لغة c\c++ والتي تستخدم في معظم نظم تشغيل لينكس، وسيكون بالملاحق شرح عملي عن كيفية كتابة وحفظ وترجمة وتنفيذ هذه البرامج على نظام لينكس (نسخة أوبونتو). ويمكن استخدام الخطوات نفسها في أي توزيعة لينكس أخرى، والقسم الثاني سنوضح فيه كيفية إنشاء وإنهاء الخيوط والروتينات المستخدمة، في إنشاء وإنهاء الخيوط، أما القسم الثالث فيبحث في انضمام وانفصال الخيوط مع عرض الروتينات المستخدمة. القسم الرابع يتناول إدارة المكدسة والروتينات المستخدمة، والوقاية من مشاكل المكدسة، وفي القسم الخامس سنوضح كيفية استخدام متغيرات Mutexs وكيفية إنشاء وتدمير الـ Mutexs ، كذلك نستعرض في هذا القسم كيفية غلق وفتح Mutexs، والقسم السادس والأخير نستعرض فيه متغيرات الحالة، وهنا سنوضح كيفية إنشاء وتدمير متغيرات الحالة والروتينات المستخدمة، كذلك يبحث القسم في الانتظار في متغيرات الحالة.

أهداف الوحدة



عزيزي الدارس، بعد فراغك من دراسة هذه الوحدة ينبغي أن تكــون قادراً على أن:

- **تعرف مكتبات POSIX thread.**
- **تذكر الصفات التي يجب أن تتصف بها البرامج التي تريد كتابتها حتى تكون مناسبة لاستخدام Pthreads.**
- **تعدد النماذج الشائعة للبرامج متعدد الخيوط.**
- **تعرف برمجة الخيط بأمان.**
- **تعدد مجموعات واجهة التطبيقات البرمجية.**
- **تذكر المعطيات التي يجب أن تحتويها الدالة المستخدمة في إنشاء خيط.**
- **تكتب الروتينات المستخدمة في إنشاء وإنهاء الخيوط.**
- **تعدد طرق إنهاء الخيوط.**
- **تكتب الروتينات المستخدمة في انفصال وانضمام الخيوط.**
- **تبين كيف يتم الوقاية من مشاكل المكدسة.**
- **تعدد خطوات استخدام mutex.**
- **تكتب الروتينات المستخدمة في انشاء وتدمير الــ mutexes.**
- **تكتب الروتينات المستخدمة في غلق وفتح الــ mutexes.**
- **تكتب الروتينات المستخدمة في إنشاء وتدمير المتغيرات الشرطية.**
- **توضح طريقة استخدام الروتينات المستخدمة فــي إنشــاء وتــدمير متغيرات الحالة.**
- **تكتب الروتينات المستخدمة في الانتظار والتأشير في متغيرات الحالة.**
- **توضح كيفية استخدام الروتينات في الانتظار والتأشير في متغيــرات الحالة.**

1. مكتبات POSIX thread (pthread)

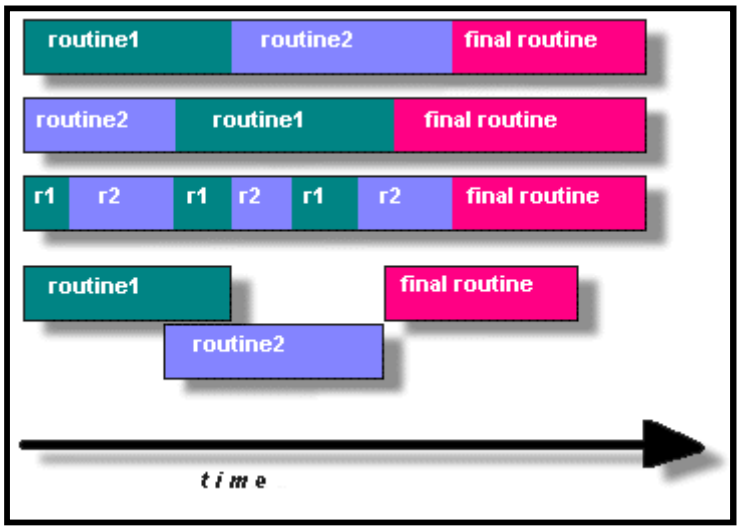
عزيزي الدارس، مكتبات POSIX thread هي واجهة تطبيقات برمجية لدعم الخيوط للغة C/C++. وهي تسمح لنا بكتابة برامج متعددة الخيوط تعمل بالتوازي. وهي مناسبة، وتظهر كفاءتها في الأنظمة متعددة المعالجات (multi-processor)، والأنظمة متعددة النواة (multi-core) حيث يمكن لكل خيط أن يعمل في معالج منفصل، مما يزيد سرعة التنفيذ خلال المعالجة المتوازية أو الموزعة. ويعتبر استخدام الخيوط أقل إزعاجاً من استخدام التفريخ (forking)، والذي يسمح بتنفيذ أكثر من عملية في وقت واحد. لأنه لن يحتاج تهيئة مساحة ذاكرة ظاهرية وبيئة لكل عملية جديدة. هنالك أمثلة برمجية للتفريخ في لينكس باستخدام لغة C، واستدعاء نداءات النظام (system calls) بالملاحق.

أيضا يمكن الاستفادة من هذه المكتبات في الأنظمة ذات المعالج الواحد (uniprocessor) حيث يمكن لخيط الإستفادة من المعالج نفسه والعمل فيه إذا كان الخيط المنفذ في انتظار دخل أو خرج أو أي شيء آخر ليس للمعالج دخل فيه. (تعدد المهام multi-task)

هنالك تقنيات برمجة متوازية أخرى، مثل MPI و PVM تستخدم في بيئات حوسبة موزعة بينما تقتصر الخيوط على أنظمة الحواسيب الواحدة (single computer system). كل الخيوط في العملية نفسها (البرنامج) تتشارك في مساحة العنونة نفسها (address space).

1.1 تصميم برامج متعددة الخيوط

للاستفادة من ميزات Pthreads يجب أن يكون البرنامج قادرا على تقسيم البرنامج إلى أجزاء (مهام)، بحيث يمكن تنفيذ كل مهمة على حدة، وبالتوازي مع تنفيذ بقية المهام. مثلا إذا كان لدينا routine1 و routine2، بحيث أن يتداخل ويتقاطع تنفيذهما، فهما مناسبان للبرمجة الخيطية (threading).



❐ عزيزي الدارس، يجب أن تكون البرامج التي نريد كتابتها تتصف بالصفات التالية حتى تكون مناسبة لاستخدام Pthreads:

① العمل الذي نريد تنفيذه، أو البيانات التي نريد العمل عليها يمكن تنفيذها في عدة مهام بالتوازي.

② أمكانية حجز عمليات دخل/خرج لتنتظر طويلاً.

③ إمكانية استخدام المعالج في بعض الأماكن، ولكن ليس فيها كلها.

④ يجب الاستجابة للأحداث غير المتزامنة (asynchronous events).

⑤ بعض الأعمال أهم من غيرها (مقاطعات الأفضلية (priority interrupts)).

يمكن استخدام Pthreads في التطبيقات المتسلسلة، مثل المتصفح الذي يعمل في حاسب ذي معالج واحد، لكنه "يبدو للمستخدم" أنه يقوم بعدة مهام في وقت واحد.

هناك العديد من النماذج الشايعة للبرامج متعددة الخيوط منها:

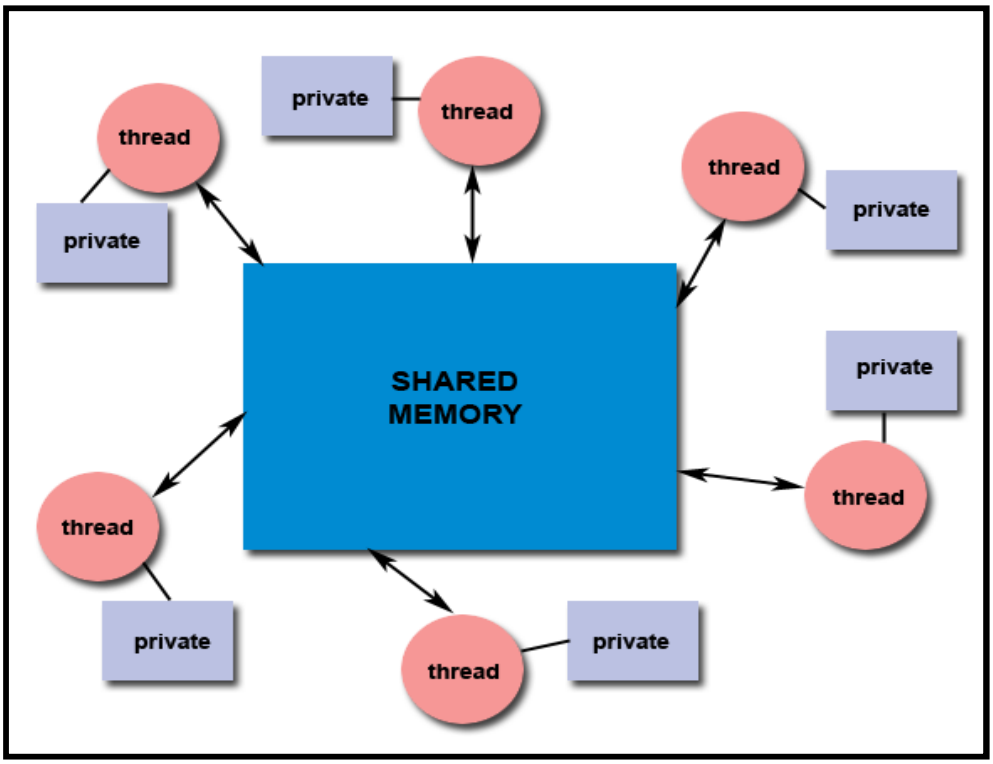
أ- نموذج المدير/العامل (Manager/worker): حيث يقوم الخيط المدير بتوزيع العمل على الخيوط الأخرى، والتي تمثل العمال.

ب- نموذج الأنبوب الانسيابي (pipeline): حيث يتم تقسيم المهمة إلى عمليات فرعية، بحيث تعتمد كل عملية على مخرجات العملية الأخرى، لكن تنفذ كل عملية في خيط منفصل (مثل تجميع السيارات).

ج- الند (peer): يشبة نموذج المدير/العامل، لكنه يختلف في أن المدير بعد توزيعه المهام على العمال يشارك هو في التنفيذ.

نموذج الذاكرة المشتركة :

1. كل الخيوط تصل إلى الذاكرة المشتركة (بيانات مشتركة (global)).
2. أيضا لكل خيط بياناته الخاصة.
3. المبرمج مسئول عن عمليات الوصول المتزامن (حماية البيانات المشتركة).



2.1 برمجة الخيط بأمان

هو مقدرة التطبيقات على تنفيذ عدة خيوط بالتوازي دون تخريب البيانات المشتركة أو توليد حالة سباق (race conditions).

مثلا إذا كان هنالك تطبيق يحتوي على عدة خيوط وكل خيط يستدعي روتين المكتبة نفسها (library routine)، فإذا افترضنا أن هذا الروتين يقوم بتعديل بيانات عامة، أو موقع في الذاكرة، فسيحاول كل خيط تعديل هذه البيانات في الوقت نفسه مما يسبب تخريبا في هذه البيانات، لذلك لابد من أن يكون هنالك نوع من الوصول المتزامن لهذه البيانات حتى نحميها من التخريب وحتى يصبح الخيط آمناً.

لذلك عليك في حال استخدام روتينات خارجية من ضمان سلامتها، وأنها آمنة، وإلا عليك تجنب استخدامها.

3.1 واجهة التطبيقات البرمجية (Pthreads API)

يمكن تقسيم الواجهة البرمجية إلى أربع مجموعات رئيسية هي:

❶ إدارة الخيط: وهي روتينات التي تعمل مباشرة مع الخيوط مثل creating, detaching, joining، إلخ.

❷ روتينات mutex : والتي تستخدم في التزامن، وتسمى (Mutex) من الكلمتين mutual exclusion، وهي معنية بتعديل الصفات المرتبطة مع الـــ mutexes.

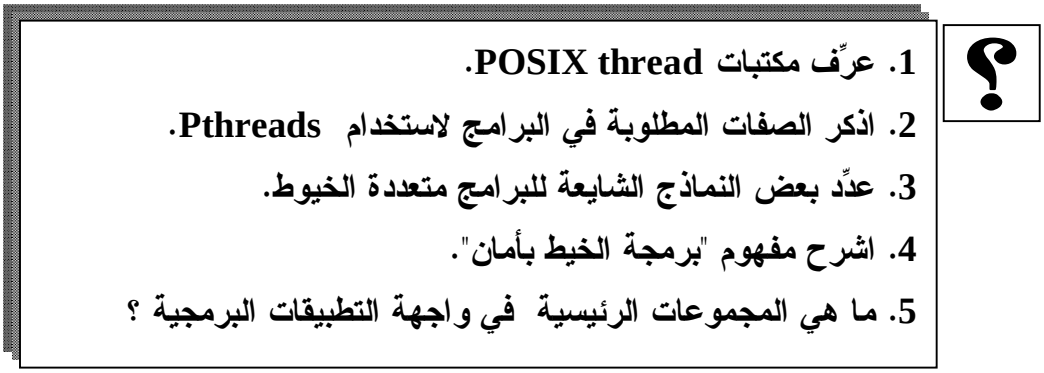
❸ متغيرات الحالة (condition variables): وهي الروتينات التي تهتم بالاتصالات بين الخيوط المتشاركة في mutex. وهي معتمدة على شروط يضعها المبرمج.

❹ التزامن (synchronization): روتينات لإدارة حجز القراءة والكتابة.

كل التعريفات في مكتبة الخيوط تبدأ بالكلمة pthread_. كما موضح في الأمثلة التالية:

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys
pthread_rwlock_	Read/write locks
pthread_barrier_	Synchronization barriers

أسئلة تقويم ذاتي



1. عرِّف مكتبات **POSIX thread**.
2. اذكر الصفات المطلوبة في البرامج لاستخدام **Pthreads**.
3. عدِّد بعض النماذج الشايعة للبرامج متعددة الخيوط.
4. اشرح مفهوم "برمجة الخيط بأمان".
5. ما هي المجموعات الرئيسية في واجهة التطبيقات البرمجية ؟

2. إنشاء و إنهاء الخيوط

1.2 إنشاء خيط

عزيزي الدارس، لإنشاء خيط نحتاج استخدام الدالة ()pthread_create، والتي تحتوي على المعطيات التالية:

- **المعطى الأول نحصل من خلاله على تعريف الخيط (thread identifier).**
- **المعطى الثاني مؤشر إلى مكان الكائن الذي يحدد صفات الخيط، إذا استخدمت null هنا فهذا يعني أنك تريد الصفات الافتراضية للخيط.**
- **المعطى الثالث هو مؤشر إلى مكان الدالة التي ينفذها الخيط.**
- **المعطى الأخير هو القيم (argument) التي نريد تمريرها إلى الدالة المنفذة داخل الخيط، إذا لم يكن لديك قيم تريد تمريرها إلى الدالة يمكنك كتابة null في هذه الخانة.**

مثلا لو كتبت شفرة الدالة بهذه الطريقة :

```
pthread_create(&th_ID, NULL, th_fun, &value);
```

فهذا يعني أنني أريد إنشاء خيط يخزن تعريف هذا الخيط في المتغير th_ID، ويستخدم هذا الخيط الصفات الافتراضية، وينفذ هذا الخيط الدالة th_fun التي تكون مدخلاتها value.

إذا أردت إنشاء ثلاثة خيوط، فعليك استدعاء الدالة pthread_create() ثلاث مرات.

أيضا نستدعي الدالة pthread_join() مع كل خيط قمنا بإنشائه لضمان انتهاء الخيوط قبل إنتهاء الدالة الرئيسية main.

2.2 إنهاء خيط

لإنهاء خيط نستخدم الدالة pthread_cancel()، لكن عليك التأكد من أن الخيط الذي تريد إنهاؤه لا يستخدم موارد قبل إنهائه. مثلاً إذا كان الخيط يحجز مساحة بالذاكرة، وقمنا باستدعاء دالة الإنهاء pthread_cancel() سنفقد مكان هذه الذاكرة، وستظل محجوزة بلا فائدة (memory leak).

3.2 الروتينات المستخدمة في إنشاء وإنهاء الخيوط

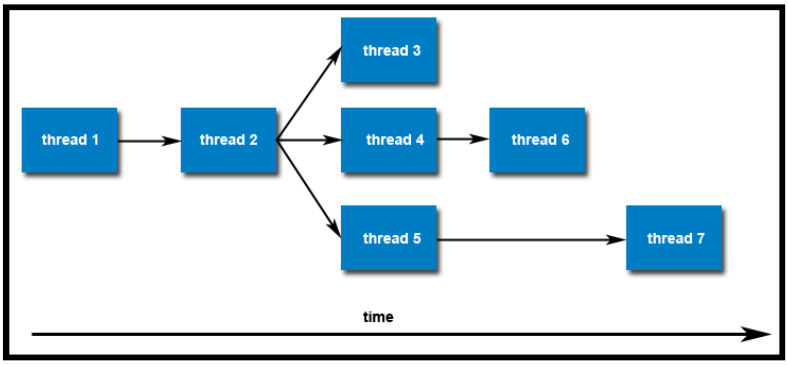
```
pthread create
(thread,attr,start_routine,arg)

pthread exit (status)

pthread attr init (attr)

pthread attr destroy (attr)
```

1.3.2 إنشاء الخيط

① الدالة الرئيسية main لديها خيط واحد افتراضي، بقية الخيوط على المبرمج أن يقوم بإنشائها بنفسه.

② الأمر `pthread_create` ينشيء خيط ويمكن استدعاؤه أكثر من مرة من أي مكان داخل الشفرة لإنشاء أكثر من خيط.

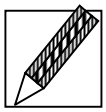
③ عند إنشاء الخيط يمكنه بدوره إنشاء خيوط أخرى، فليس هنالك هرمية بين الخيوط.



Quiz

بعد إنشاء الخيط كيف ستعرف متى سيجدول لينفذ بواسطة نظام التشغيل ؟

تدريب (1)



اكتب برنامج لغة C++ يوضح إنشاء و إنهاء خيطين يوضح إنشاء الخيط الأول ثم إنشاء الثاني و أيضا عملية الإنهاء بالترتيب نفسه.

2.3.2 صفات الخيط (Thread Attributes)

- افتراضياً ينشأ الخيط بصفات معينة، ويمكن للمبرمج تغير بعض هذه الصفات عبر كائن الصفات (thread attribute object).
- تستخدم `pthread_attr_init` و`pthread_attr_destroy` لتهيئة/تدمير كائن الصفات .
- هنالك روتينات أخرى تستخدم لمعرفة أو تغيير صفات معينة في كائن الصفات.

3.3.2 إنهاء الخيط (Terminating Threads)

هنالك عدة طرق لإنهاء الخيط؛ منها:

- رجوع الخيط من الروتين الذي بدأ فيه (الروتين الرئيسي الذي قام بتهيئة الخيط).
- استدعاء الخيط للروتين `pthread_exit`.
- إلغاء الخيط بخيط آخر وذلك باستدعاء الروتين `pthread_cancel`.
- إذا انتهت العملية بكاملها باستدعاء روتين مثل exec أو exit.

- يمكن استخدام الروتين `pthread_exit` للخروج من الخيط ويتم استدعاؤه في نهاية الخيط عند ما نريد عمل شيء آخر.
- إذا انتهى البرنامج الرئيسي `main()` قبل الخيوط التي أنشأها، وخرج بالأمر `pthread_exit()`، فإن الخيوط الأخرى ستظل تعمل.غير ذلك انتهاء `main()` سينهي معه كل الخيوط التي أنشأها.
- يستطيع المبرمج (اختياريا) تحديد حالة الانتهاء (termination status) والتي تكون مخزنة كمؤشر من النوع void في أي خيط قد يشارك في استدعاء الخيط.
- الروتين `pthread_exit()` لا يغلق الملفات المفتوحة داخل أي خيط، وستظل مفتوحة حتى بعد انتهاء الخيط.

ملحوظة

في الروتينات التي من المتوقع انتهاء تنفيذها بصورة طبيعة يمكننا الاستغناء عن `pthread_exit()`، ما لم نريد تمرير pass a return code back. لكن هنالك مشكلة غير منتهية عندما تكتمل `main()` قبل توزيع الخيوط (threads it spawned). فإذا لم تستدعي `pthread_exit()` ضمنياً، عندما تكتمل `main()` فإن العملية (وكل خيوطها) ستنتهي. باستدعاء `pthread_exit()` في `main()`، فإن العملية وكل خيوطها ستحفظ حية حتى ولو اكتمل تنفيذ كل شفرة موجودة في

مثال (إنشاء وإنهاء الخيط) :

هذا المثال ينشيء (5) خيوط باستدعاء الروتين `pthread_create()`. كل خيط سيطبع الرسالة "Hello World!" ثم ستنتهي الخيوط باستدعاء الروتين `pthread_exit()`.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread
#%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread
%ld\n", t);
      rc = pthread_create(&threads[t], NULL,
PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from
pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

4.2 تمرير قيم إلى الخيط (Passing Arguments to Threads)

الروتين `pthread_create()` بتمرير قيمة واحدة (one argument) لروتين بدء الخيط (thread start routine). ولتمرير أكثر من قيمة يمكنك إنشاء بنية تحتوي على كل القيم التي تريد تمريرها، ثم تمرر مؤشر هذه البنية في الروتين `pthread_create()`.

كل القيم يجب أن تمرر بالمرجع ((by reference)، وتحول إلى (void *).

مثال برمجي (تمرير قيم إلى الخيط) :

في البرنامج التالي نوضح كيف يمكن تمرير عدد صحيح (integer) لكل خيط .

الاستدعاء يتم باستخدام بنية بيانات (data structure)) واحدة لكل خيط

مثال1

```
long *taskids[NUM_THREADS];

for(t=0; t<NUM_THREADS; t++)
{
   taskids[t] = (long *)
malloc(sizeof(long));
   *taskids[t] = t;
   printf("Creating thread %ld\n", t);
   rc = pthread_create(&threads[t], NULL,
PrintHello, (void *) taskids[t]);
  ...
}
```

مثال 2

كيفية إعداد/تمرير أكثر من قيمة باستخدام بنية (structure). كل خيط يستلم نسخة واحدة من هذه البنية.

```
struct thread_data{
   int  thread_id;
   int  sum;
   char *message;
};

struct thread_data
thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
   struct thread_data *my_data;
  ...
   my_data = (struct thread_data *)
threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
  ...
}

int main (int argc, char *argv[])
{
  ...
   thread_data_array[t].thread_id = t;
   thread_data_array[t].sum = sum;
   thread_data_array[t].message =
messages[t];
```

```
   rc = pthread_create(&threads[t], NULL,
PrintHello,
         (void *) &thread_data_array[t]);
  ...
}
```

مثال 3

في هذا البرنامج نوضح كيف يمكن تمرير قيم غير صحيحة.فهو يمرر عنوان المتغير t الذي يتشارك مساحة ذاكرة متاحة لكل الخيوط. وخلال حلقات التكرار يتغير موقع هذه الذاكرة، وقد يحدث هذا التغيير قبل أن تتعامل معها الخيوط.

```
int rc;
long t;

for(t=0; t<NUM_THREADS; t++)
{
   printf("Creating thread %ld\n", t);
   rc = pthread_create(&threads[t], NULL,
PrintHello, (void *) &t);
  ...
}
```

أسئلة تقويم ذاتي



1. **اذكر الدالة المستخدمة لإنشاء خيط والمعطيات التي تحتويها هذه الدالة.**
2. **اكتب الروتينات المستخدمة في إنشاء وإنهاء الخيوط.**

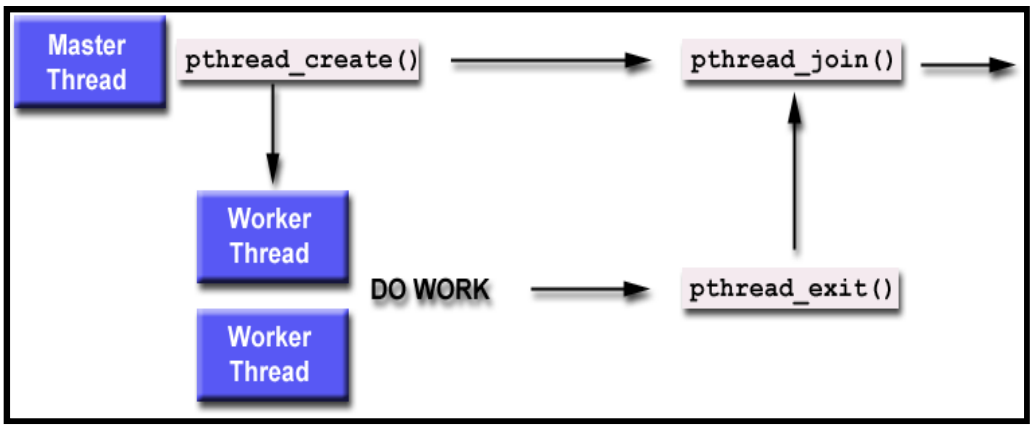
3. انضمام وانفصال الخيوط (Joining and Detaching (Threads

الروتينات المستخدمة:

pthread join (threadid,status) **pthread detach (threadid,status)** **pthread attr setdetachstate (attr,detachstate)** **pthread attr getdetachstate (attr,detachstate)**

1.3 الإنضمام (Joining)

- "Joining" هي إحدى الطرق لعمل تزامن (synchronization) بين الخيوط. مثلا:



- الروتين `pthread_join()` يحجز الخيط المستدعي (calling thread) حتى ينتهي الخيط المحدد (`threaded`).
- المبرمج يكون قادراً على الحصول على حالة انتهاء الخيط الهدف إذا كان محدداً عند استدعاء الروتين `pthread_exit()`.
- تتم عملية انضمام الخيط (joining thread) مرة واحدة فقط (أي استدعاء الروتين `pthread_join()` مرة واحدة لنفس الخيط). ويعتبر خطأ منطقي محاولة عمل أكثر من انضمام (multiple joins) للخيط نفسه.
- هنالك طرق أخرى للتزامن؛ مثل mutexes، والمتغيرات الفرعية (condition variables) سنقاشها لاحقا في هذا الباب.

) هل الخيط قابل للانضمام أم لا (Joinable or Not)؟

- **عند إنشاء الخيط هنالك أحد صفاته توضح هل هذا الخيط قابل للإنضمام (joinable) أم لا (detached). فقط الخيوط القابلة للإنضمام هل التي نضمها، أما إذا أنشيء الخيط كخيط منفصل (detached) فلا يمكن ضمه أبدا.**
- **في معايير POSIX الاخيرة تم تحديد أن الخيط يبرمج عند إنشائه أن يكون من النوع الفابل للضم (joinable).**
- **لتحديد حالة الخيط عند إنشائة هل سيكون joinable أو detached، يمكننا استخدام القيمة المدخلة `attr` في الروتين**
- **`pthread_create()` كالتالي:**

1. **Declare a pthread attribute variable of the `pthread_attr_t` data type**
2. **Initialize the attribute variable with `pthread_attr_init()`**
3. **Set the attribute detached status with `pthread_attr_setdetachstate()`**
4. **When done, free library resources used by the attribute with `pthread_attr_destroy()`**

2.3 الإنفصال (Detaching)

- **يمكن استخدام الروتين `pthread_detach()` لجعل الخيط منفصل حتى ولو تم انشائه منضمّاً (joinable)، وليس هنالك روتين عكسي.**

✷ تنبيه

إذا كان الخيط يجب أن ينضم، فعليك بإنشائه من البداية من النوع المنضم (joinable). هذا يسمح لك بالتنقلية (portability)، حيث لن تجد كل المكتبات تنشيء الخيط من البداية منضمّاً (joinable by default).

أما إذا عرفت مسبقاً أن الخيط لن يحتاج إلى انضمام مع خيط آخر، فيمكنك إنشاؤه في حالة المنفصل، حيث يوفر ذلك تحرير بعض موارد النظام.

مثال برمجي (Joining) :

في هذا البرنامج نوضح كيف يمكن لخيط أن ينتظر انتهاء خيط آخر باستخدام الروتين join. هنا قمنا بإنشاء الخيط من البداية بأنه قابل للانضمام، حتى نستطيع ضمه مؤخرًا .

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
4   #define NUM_THREADS

void *BusyWork(void *t)
{
   int i;
   long tid;
   double result=0.0;
   tid = (long)t;
   printf("Thread %ld
starting...\n",tid);
   for (i=0; i<1000000; i++)
   {
      result = result + sin(i) * tan(i);
   }
   printf("Thread %ld done. Result =
%e\n",tid, result);
   pthread_exit((void*) t);
}

int main (int argc, char *argv[])
{
   pthread_t thread[NUM_THREADS];
   pthread_attr_t attr;
```

```
   int rc;
   long t;
   void *status;

   /* Initialize and set thread detached
attribute */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr,
PTHREAD_CREATE_JOINABLE);

   for(t=0; t<NUM_THREADS; t++) {
      printf("Main: creating thread
%ld\n", t);
      rc = pthread_create(&thread[t],
&attr, BusyWork, (void *)t);
      if (rc) {
         printf("ERROR; return code from
pthread_create()
                is %d\n", rc);
         exit(-1);
         }
      }

   /* Free attribute and wait for the
other threads */
   pthread_attr_destroy(&attr);
   for(t=0; t<NUM_THREADS; t++) {
      rc = pthread_join(thread[t],
&status);
      if (rc) {
         printf("ERROR; return code from
pthread_join()
                is %d\n", rc);
         exit(-1);
```

```
        }
      printf("Main: completed join with
thread %ld having a status
            of %ld\n",t,(long)status);
      }

printf("Main: program completed.
Exiting.\n");
pthread_exit(NULL);
}
```

أسئلة تقويم ذاتي



1. **اكتب الروتينات المستخدمة في انضمام وانفصال الخيوط.**
2. **ما هو الإنضمام (Joining)؟**
3. **وضح كيف يتم تحديد، هل الخيط قابل للانضمام أم لا (Joinable or Not)؟**
4. **كيف يمكن استخدام الروتين جعل الخيط منفصل حتى ولو تم إنشاؤه منضمًّا (joinable). وليس هنالك روتين عكسي.**

4. إدارة المكدسة Stack Management

الروتينات المستخدمة:

```
pthread attr getstacksize (attr,
stacksize)

pthread attr setstacksize (attr,
stacksize)

pthread attr getstackaddr (attr,
stackaddr)

pthread attr setstackaddr (attr,
stackaddr)
```

1.4 الوقاية من مشاكل المكدسة

- **لم يتم تحديد سعة المكدسة في معايير POSIX، وإنما يختلف ذلك ويعتمد على التطبيق.**
- **من السهل الزيادة على حد المكدسة الافتراضي، مما يسبب إنهاء البرنامج، أو تخريب البيانات.**
- **البرامج الآمنة والمتنقلة يجب ألاّ تعتمد على سعة المكدسة الافتراضية، وإنما يجب حجز مساحة كافية للمكدسة لكل خيط باستخدام الروتين pthread_attr_setstacksize.**
- **الروتينات pthread_attr_getstackaddr والروتين pthread_attr_setstackaddr يمكن استخدامهما في البيئات التي يجب فيها وضع مكدس الخيط في منطقة معينة بالذاكرة.**

مثال برمجي على إدارة المكدسة:

في هذا المثال نوضح كيفية معرفة وتغيير حجم مكدسة الخيط.

```
#include <pthread.h>
#include <stdio.h>
#define NTHREADS 4
#define N 1000
#define MEGEXTRA 1000000

pthread_attr_t attr;

void *dowork(void *threadid)
{
   double A[N][N];
   int i,j;
   long tid;
   size_t mystacksize;

   tid = (long)threadid;
   pthread_attr_getstacksize (&attr,
&mystacksize);
   printf("Thread %ld: stack size = %li
bytes \n", tid, mystacksize);
   for (i=0; i<N; i++)
     for (j=0; j<N; j++)
      A[i][j] = ((i*j)/3.452) + (N-i);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
   pthread_t threads[NTHREADS];
   size_t stacksize;
   int rc;
```

```
   long t;

   pthread_attr_init(&attr);
   pthread_attr_getstacksize (&attr,
&stacksize);
   printf("Default stack size = %li\n",
stacksize);
   stacksize =
sizeof(double)*N*N+MEGEXTRA;
   printf("Amount of stack needed per
thread = %li\n",stacksize);
   pthread_attr_setstacksize (&attr,
stacksize);
   printf("Creating threads with stack
size = %li bytes\n",stacksize);
   for(t=0; t<NTHREADS; t++){
      rc = pthread_create(&threads[t],
&attr, dowork, (void *)t);
      if (rc){
         printf("ERROR; return code from
pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   printf("Created %ld threads.\n", t);
   pthread_exit(NULL);
}
```

روتينات متنوعة

HYPERLINK "https://computing.llnl.gov/tutorials/pthreads/man/pthread_self.txt" \t "W26"pthread_self ()

HYPERLINK "https://computing.llnl.gov/tutorials/pthreads/man/pthread_equal.txt" \t "W27"pthread_equal (thread1,thread2)

- **الروتين `pthread_self` يعطي (returns) رقم خيط فريد للخيط المستدعي (calling thread).**
- **الروتين `pthread_equal` يقارن بين رقمي خيطين (thread Ids) ويعطي 0 إذا كانا مختلفين وقيمة غير صفرية في غير ذلك.**
- **يعتبر رقم تعريف الخيط كائناً وبالتالي لا يمكن استخدام العلامة == لمقارنة قيم الأرقام التعريفية للخيوط (thread Ids).**

أسئلة تقويم ذاتي



كيف يتم الوقاية من مشاكل المكدسة؟

5. متغيرات الميوتكس (Mutex Variables)

- **Mutex هي اختصار لـــ "mutual exclusion".، وتعتبر متغيرات Mutex من أهم التطبيقات التي تستخدم لتحقيق التزامن بين الخيوط (thread synchronization) وحماية البيانات المشتركة عند حدوث أكثر من أمر كتابة عليها (multiple writes).**
- **يعمل المتغير mutex كغلق (lock) لمنع الوصول إلى البيانات المشركة. والمفهوم من وراء فكرة المتغير mutex المستخدمة في Pthreads هو أن خيطاً واحداً فقط هو من يغلق (يمتلك) متغير mutex في أي لحظة. أي إذا كان هنالك عدد من الخيوط تحاول غلق mutex واحدة فقط، هي التي ستنجح.ولن يستطيع أي خيط آخر غلق (امتلاك) هذا المتغير (mutex) ما لم يتركه الخيط الذي حصل عليه مسبقا.**
- **يمكن استخدام Mutexes في الوقاية من (منع) حدوث حالات سباق ("race" conditions).**

المثال التالي يوضح حالات سباق حدثت في حركة مالية ببنك :

الرصيد	الخيط الثاني	الخيط الأول
$1000		Read balance: $1000
$1000	Read balance: $1000	
$1000	Deposit $200	
$1000		Deposit $200
$1200		Update balance $1000+$200
$1200	Update balance $1000+$200	

- في المثال أعلاه يجب عمل mutex لغلق الرصيد ("lock the "Balance")، بينما يقوم الخيط باستخدام هذه البيانات المشتركة. المتغيرات التي تم تحديثها تنتمي إلى مقطع حرج (critical section).

1.5 خطوات استخدام mutex

- إنشاء وتهيئة المتغير mutex.
- عدة خيوط تحاول غلق الـــ mutex
- خيط واحد فقط ينجح في امتلاك الـــ mutex
- ينفذ الخيط المالك للـــ mutex بعض العمل
- يترك الخيط المالك الـــ mutex
- يحصل خيط آخر على الـــmutex ليؤدى عملاً ما.
- في النهاية يتم تدمير الـــ mutex.

- عندما تتنافس عدة خيوط على mutex، تظل لخيوط التي لم تحصل على الـــ mutex محجوزة في ذاك الوضع وتستدعى طلب الحجز بالأمر "trylock" بدلا من استدعائه بالأمر "lock".
- عند حماية بيانات مشتركة، فإن مهمة المبرمج التأكد من أن كل خيط يحتاج استخدام الـــ mutex قد فعل. فمثلا لوكان لدينا ثلاث خيوط تريد تعديل البيانات نفسها، ولكن خيط واحد فقط هو من استخدم الـــ mutex، فتصبح البيانات تالفة، ولا فائدة من استخدام الـــ mutex.

2.5 إنشاء وتدمير الـــ Mutexes

الروتينات المستخدمة:

```
pthread mutex init (mutex,attr)

pthread mutex destroy (mutex)

pthread mutexattr init (attr)

pthread mutexattr destroy (attr)
```

1.2.5 طريقة الاستخدام

- **يجب الإعلان عن المتغيرات الـــ Mutex بالنوع pthread_mutex_t، ثم تهيئتها قبل استخدامها. هنالك طريقتان لتهيئة المتغير mutex، هي:**

① ساكن (Statically)، عند الإعلان عنه. مثلاً:

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

② ديناميكي (Dynamically)، بالروتين pthread_mutex_init().

هنا يمكن إعداد صفات كائن mutex وهي attr.

عند التهيئة يكون المتغير mutex غير محجوز (unlocked).

3.5 غلق وفتح الـــ Mutexes

الروتينات المستخدمة:

```
pthread_mutex_lock (mutex)

pthread_mutex_trylock (mutex)

pthread_mutex_unlock (mutex)
```

1.3.5 طريقة الاستخدام

- **الروتين `pthread_mutex_lock()` يستخدم بواسطة الخيط للحصول على المتغير mutex (الغالق). إذا كان الـــ mutex مغلق بخيط آخر سيتم حجز هذا النداء حتى يتم فتح (unlocked) المتغير mutex.**
- **الروتين `pthread_mutex_trylock()` يحاول غلق المتغير mutex، لكنه يعطي عبارة الخطأ busy مباشرة إذا وجد المتغير mutex مغلقاً. This routine may be useful in preventing يكون هذا الروتين مناسب لتجنب شرط من شروط الاختناق (deadlock).**
- **الروتين `pthread_mutex_unlock()` يفتح الـــ mutex إذا استدعي بواسطة الخيط المالك للـــ mutex.وهو مطلوب استدعاؤه بعد أن ينتهي الخيط من استخدام البيانات المحمية. وقد يحدث خطأ في الحالات التالية:**
  - **إذا كان الـــ mutex أصلاً مفتوحاً (unlocked).**
  - **إذا كان الـــ mutex مملوكاً لخيط آخر.**

Quiz

عندما يكون هنالك أكثر من خيط ينتظر غلق mutex، فأي واحد مــن هــذه الخيــوط سيحصل على الغلق بعد أن يصبح متاحاً؟

مثال برمجي (استخدام Mutexes) :

هنا في هذا البرنامج نوضح كيفية استخدام متغيرات Mutexes في برنامج متعــدد الخيوط. البيانات الرئيسية متاحة لكل الخيوط عبر بينية عامة. كل خيط يعمــل علــى جزء من البيانات. الخيط الرئيسي سينتظر كل الخيوط لتنهــي عملياتهــا، ثــم يطبــع النتيجة النهائية.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

/*
The following structure contains the
necessary information
to allow the function "dotprod" to access
its input data and
place its output into the structure.
*/

typedef struct
 {
   double      *a;
   double      *b;
   double     sum;
   int     veclen;
 } DOTDATA;
```

```
/* Define globally accessible variables
and a mutex */

#define NUMTHRDS 4
#define VECLEN 100
   DOTDATA dotstr;
   pthread_t callThd[NUMTHRDS];
   pthread_mutex_t mutexsum;

/*
The function dotprod is activated when
the thread is created.
All input to this routine is obtained
from a structure
of type DOTDATA and all output from this
function is written into
this structure. The benefit of this
approach is apparent for the
multi-threaded program: when a thread is
created we pass a single
argument to the activated function -
typically this argument
is a thread number. All  the other
information required by the
function is accessed from the globally
accessible structure.
*/

void *dotprod(void *arg)
{

   /* Define and use local variables for
convenience */
```

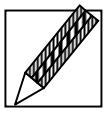
```
   int i, start, end, len ;
   long offset;
   double mysum, *x, *y;
   offset = (long)arg;

   len = dotstr.veclen;
   start = offset*len;
   end   = start + len;
   x = dotstr.a;
   y = dotstr.b;

   /*
   Perform the dot product and assign
result
   to the appropriate variable in the
structure.
   */

   mysum = 0;
   for (i=start; i<end ; i++)
    {
      mysum += (x[i] * y[i]);
    }

   /*
   Lock a mutex prior to updating the
value in the shared
   structure, and unlock it upon
updating.
   */
   pthread_mutex_lock (&mutexsum);
   dotstr.sum += mysum;
   pthread_mutex_unlock (&mutexsum);
```

```
   pthread_exit((void*) 0);
}

/*
The main program creates threads which do
all the work and then
print out result upon completion. Before
creating the threads,
the input data is created. Since all
threads update a shared structure,
we need a mutex for mutual exclusion. The
main thread needs to wait for
all threads to complete, it waits for
each one of the threads. We specify
a thread attribute value that allow the
main thread to join with the
threads it creates. Note also that we
free up handles when they are
no longer needed.
*/

int main (int argc, char *argv[])
{
   long i;
   double *a, *b;
   void *status;
   pthread_attr_t attr;

   /* Assign storage and initialize
values */
   a = (double*) malloc
(NUMTHRDS*VECLEN*sizeof(double));
   b = (double*) malloc
(NUMTHRDS*VECLEN*sizeof(double));
```

```
   for (i=0; i<VECLEN*NUMTHRDS; i++)
    {
     a[i]=1.0;
     b[i]=a[i];
    }

   dotstr.veclen = VECLEN;
   dotstr.a = a;
   dotstr.b = b;
   dotstr.sum=0;

   pthread_mutex_init(&mutexsum, NULL);

   /* Create threads to perform the
dotproduct  */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr,
PTHREAD_CREATE_JOINABLE);

for(i=0; i<NUMTHRDS; i++)
        {
/*
Each thread works on a different set
of data.
The offset is specified by 'i'. The
size of
the data for each thread is
indicated by VECLEN.
*/
pthread_create(&callThd[i], &attr,
dotprod, (void *)i);
}
```

```
pthread_attr_destroy(&attr);

        /* Wait on the other threads */
for(i=0; i<NUMTHRDS; i++)
        {
  pthread_join(callThd[i], &status);
}

   /* After joining, print out the
results and cleanup */
   printf ("Sum =  %f \n", dotstr.sum);
   free (a);
   free (b);
   pthread_mutex_destroy(&mutexsum);
   pthread_exit(NULL);
}
```

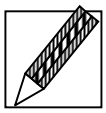
تدريب (2)



اكتب أمثلة لشفرات تستخدم ال *Mutexes* في المكتبات (libararies) التالية:

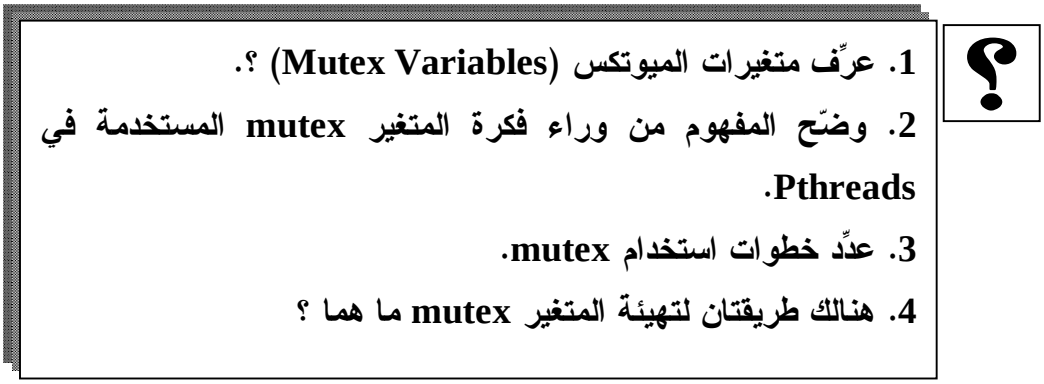
POSIX
Win32
OS/2

تدريب (3)



أكتب برنامجاً يوضح مفهوم انضمام الخيوط و استخدام روتينات غلق وفتح الـــ Mutexes.

أسئلة تقويم ذاتي



1. عرِّف متغيرات الميوتكس (Mutex Variables) ؟.
2. وضّح المفهوم من وراء فكرة المتغير mutex المستخدمة في Pthreads.
3. عدِّد خطوات استخدام mutex.
4. هنالك طريقتان لتهيئة المتغير mutex ما هما ؟

6. متغيرات الحالة (Condition Variables)

- تعتبر متغيرات الحالة نوعاً آخر يستخدم في تزامن الخيوط (synchronization)، بينما يعتمد التزامن في mutexes على التحكم في وصول الخيط للبيانات، حيث يعتمد التزامن في متغيرات الحالة على القيمة الفعلية للبيانات.
- بدون متغيرات الحالة يحتاج المبرمج لخيوط تختبر باستمرار (polling)؛ هل تم تحقق الشروط أم لا؟ وهذا قد يهدر الموارد ويستهلكها، لأن الخيط سيكون دوماً مشغولاً بهذه الأمور. فمتغير الحالة يسمح لنا بالتأكد من تحقق الشرط دون الحاجة لـــpolling. متغيرات الحالة تعمل مع غلق الmutex .
- طريقة استخدام المتغيرات الشرطية موضحة أدناه:

الخيط الرئيسي (Main Thread)

- **Declare and initialize global data/variables which require synchronization (such as "count")**
- **Declare and initialize a condition variable object**
- **Declare and initialize an associated mutex**
- **Create threads A and B to do work**

Thread A	Thread B
Do work up to the point where a certain condition must occur (such as "count" must reach a specified value) **Lock associated mutex and check value of a global variable** **Call `pthread_cond_wait()` to perform a blocking wait for signal from Thread-B. Note that a call to `pthread_cond_wait()` automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.** **When signalled, wake up. Mutex is**	**Do work** **Lock associated mutex** **Change the value of the global variable that Thread-A is waiting upon.** **Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.** **Unlock mutex.** **Continue**

automatically and atomically locked. o Explicitly unlock mutex o Continue	
Main Thread **Join / Continue**	

1.6 إنشاء وتدمير متغيرات الحالة

الروتينات المستخدمة:

```
pthread cond init (condition,attr)

pthread cond destroy (condition)

pthread condattr init (attr)

pthread condattr destroy (attr)
```

1.1.6 طريقة الاستخدام

- **يجب الاعلان عن متغيرات الحالة بالنوع pthread_cond_t، ويجب تهيئتها قبل استخدامها. وهنالك طريقتان لتهيئة المتغيرات؛ هما:**

❶ ساكنة (Statically)، عند الإعلان عنها، مثلا:

```
pthread_cond_t myconvar =
PTHREAD_COND_INITIALIZER;
```

❷ ديناميكية (Dynamically) باستخدام الروتين

`pthread_cond_init()`. قيمة تعريف المتغير الشرطي (ID) الذي ينشأ ترجع للخيط الذي يستدعيها عبر المدخل (condition parameter). ويمكن إعداد صفات كائن المتغير الفرعي attr .

- الروتين `pthread_condattr_init()` والروتين `pthread_condattr_destroy()` يستخدمان لإنشاء وتدمير كائنات الصفات للمتغيرات الفرعية(condition variable attribute objects).
- الروتين `pthread_cond_destroy()` يجب أن يستخدم لتحرير متغير الحالة الذي لا نحتاج إليه.

2.6 الانتظار والتأشير في متغيرات الحالة

Waiting and Signaling on Condition Variables

الروتنينات المستخدمة:

```
pthread cond wait (condition,mutex)
pthread cond signal (condition)
pthread cond broadcast (condition)
```

1.2.6 طريقة الاستخدام

- الروتين `pthread_cond_wait()` يحجز الخيط المستدعي (calling thread) حتى يتحقق الشرط المحدد (condition is signaled). هذا الروتين يجب استدعاؤه عندما يكون mutex مغلقاً، ويحرر mutex تلقائياً بينما هو منتظر بعد استلام الإشارة واستيقاظ الخيط، سيغلق mutex تلقائياً، ويستخدم بواسطة الخيط، وسيكون المبرمج مسئولاً عن فتح mutexعندما ينتهي الخيط منه.

- الروتين pthread_cond_signal() مسئول عن إرسال إشارة إلى (إيقاظ) خيط آخر (الخيط الذي ينتظر المتغير الشرطي). يجب أن يستدعى بعد غلق mutex، ويجب أن يفتح mutex ليكتمل الروتين pthread_cond_wait() عمله.
- الروتين pthread_cond_broadcast() يجب أن يستخدم بدلاً من pthread_cond_signal() إذا كان هنالك أكثر من خيط محجوز في حالة الانتظار(blocking wait state).
- استدعاء الروتين pthread_cond_signal() قبل الروتين pthread_cond_wait() يعتبر خطأ منطقياً (logical error).

✷تنبيه

غلق وفتح المتغير mutex بصورة حكيمة ضروري عند استخدام هذه الروتينات. مثلاً:

- فشل غلق الـ mutex قبل استدعاء pthread_cond_wait() قد يمنعه من الحجز.
- الفشل في فتح (unlock) الـ mutex بعد استدعاء pthread_cond_signal() قد لا يسمح للروتين pthread_cond_wait() الذي يعمل معها بالاكتمال (يظل محجوز).

مثال برمجي (استخدام متغيرات الحالة)

في هذا المثال نقوم بإنشاء ثلاثة خيوط، اثنين من هذه الخيوط تنفذان عمل وتحدثان المتغير count. الخيط الثالث ينتظر حتى يصل المتغير count قيمة معينة.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
int     thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *t)
{
  int i;
  long my_id = (long)t;

  for (i=0; i<TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

    /*
    Check the value of count and signal
waiting thread when condition is
    reached.  Note that this occurs while
mutex is locked.
    */
    if (count == COUNT_LIMIT) {

pthread_cond_signal(&count_threshold_cv);
      printf("inc_count(): thread %ld, count
= %d  Threshold reached.\n",
             my_id, count);
      }
    printf("inc_count(): thread %ld, count =
```

```
%d, unlocking mutex\n",
         my_id, count);
    pthread_mutex_unlock(&count_mutex);

    /* Do some "work" so threads can
alternate on mutex lock */
    sleep(1);
    }
  pthread_exit(NULL);
}

void *watch_count(void *t)
{
  long my_id = (long)t;

  printf("Starting watch_count(): thread
%ld\n", my_id);

  /*
  Lock mutex and wait for signal.  Note that
the pthread_cond_wait
  routine will automatically and atomically
unlock mutex while it waits.
  Also, note that if COUNT_LIMIT is reached
before this routine is run by
  the waiting thread, the loop will be
skipped to prevent pthread_cond_wait
  from never returning.
  */
  pthread_mutex_lock(&count_mutex);
  if (count<COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cv,
&count_mutex);
    printf("watch_count(): thread %ld
Condition signal received.\n", my_id);
    count += 125;
```

```
    printf("watch_count(): thread %ld count
now = %d.\n", my_id, count);
    }
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  int i, rc;
  long t1=1, t2=2, t3=3;
  pthread_t threads[3];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable
objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv,
NULL);

  /* For portability, explicitly create
threads in a joinable state */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,
PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr,
watch_count, (void *)t1);
  pthread_create(&threads[1], &attr,
inc_count, (void *)t2);
  pthread_create(&threads[2], &attr,
inc_count, (void *)t3);

  /* Wait for all threads to complete */
  for (i=0; i<NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
```

```
  printf ("Main(): Waited on %d  threads.
Done.\n", NUM_THREADS);

  /* Clean up and exit */
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit(NULL);
}
```

أسئلة تقويم ذاتي



1. اشرح طريقة عمل المتغيرات الشرطية موضحاً الفرق بينها وبين الـــ mutexes.
2. اكتب الروتينات المستخدمة في إنشاء وتدمير المتغيرات الشرطية.
3. وضّح طريقة استخدام الروتينات في إنشاء وتدمير المتغيرات الشرطية.

الخلاصة

- وضحت الوحدة أن مكتبات Pthreads هي واجهة تطبيقات برمجية لدعم الخيوط للغة c\c++، وهي تسمح بكتابة برامج متعددة الخيوط تعمل بالتوازي.
- عرضت الوحدة الصفات التي يجب أن تتصف بها البرامج التي نريد كتابتها حتى تكون مناسبة لاستخدام Pthreads : العمل الذي نريد تنفيذه أو البيانات التي نريد العمل عليها يمكن تنفيذها في عدة مهام بالتوازي- أمكانية حجز عمليات دخل/خرج لتنتظر طويلاً- إمكانية استخدام المعالج في بعض الأماكن، ولكن ليس فيها كلها- يجب الاستجابة للإحداث غير المتزامنة (asynchronous events)- بعض الأعمال أهم من غيرها (مقاطعات الأفضلية (priority interrupts)).
- قسمت الوحدة واجهة التطبيقات البرمجية إلى:- إدارة الخيط: وهي الروتينات التي تعمل مباشرة مع الخيوط مثل creating, detaching, joining، إلخ- روتينات mutex: والتي تستخدم في التزامن وتسمى (Mutex) من الكلمتين mutual exclusion، وهي معنية بتعديل الصفات المرتبطة مع الـ mutexes- المتغيرات الشرطية (condition variables): وهي الروتينات التي تهتم بالاتصالات بين الخيوط المتشاركة في mutex. وهي معتمدة على شروط يضعها المبرمج- التزامن (synchronization): روتينات لإدارة حجز القراءة والكتابة.
- بينت الوحدة أنه لإنشاء الخيط نحتاج لاستخدام الدالة (pthread_create، والتي تحتوي على المعطيات التالية:المعطى الأول نحصل من خلاله على تعريف الخيط (thread identifier). المعطى الثاني مؤشر إلى مكان الكائن الذي يحدد صفات الخيط، إذا استخدمت null هنا فهذا يعني أنك تريد الصفات الافتراضية للخيط. المعطى الثالث هو مؤشر إلى مكان الدالة التي ينفذها الخيط. المعطى الأخير هو

القيم (argument) التي نريد تمريرها إلى الدالة المنفذة داخل الخيط، إذا لم يكن لديك قيم تريد تمريرها إلى الدالة يمكنك كتابة null في هذه الخانة.

- كذلك بينت الوحدة أنه لإنهاء الخيط نستخدم الدالة pthread_cancel().
- عددت الوحدة عدة طرق لإنهاء الخيط :رجوع الخيط من الروتين الذي بدأ فيه (الروتين الرئيسي الذي قام بتهيئة الخيط)- استدعاء الخيط للروتين pthread_exit- إلغاء الخيط بخيط آخر، وذلك باستدعاء الروتين pthread_cancel- إذا انتهت العملية بكاملها باستدعاء روتين؛ مثل exec أو exit.
- وضحت الوحدة أن الانضمام (Joining) هي إحدى الطرق لعمل تزامن بين الخيوط، ويستخدم الروتين pthread_join() أما الانفصال فيستخدم فيه الروتين pthread_detach().
- حدّدت الوحدة الروتين pthread_attr_getstackaddr والروتين pthread_attr_setstackaddr يمكن استخدامهما في البيئات التي يجب فيها وضع مكدس الخيط في منطقة معينة بالذاكرة.
- اعتبرت الوحدة متغيراتMutex من أهم التطبيقات التي تستخدم لتحقيق التزامن بين الخيوط وحماية البيانات المشتركة عند حدوث أكثر من أمر كتابة عليها. يعمل Mutex كغلق (lock) لمنع الوصول إلى البيانات المشتركة. يمكن استخدام Mutexes في الوقاية من منع حدوث حالات سباق.
- عرضت الوحدة خطوات استخدام mutex : إنشاء وتهيئة المتغير mutex- عدة خيوط تحاول غلق الـ mutex - خيط واحد فقط ينجح في امتلاك الـ mutex - ينفذ الخيط المالك للـ mutex بعض العمل - يترك الخيط المالك الـ mutex - يحصل خيط آخر على الـmutex يؤدى عملا ما - في النهاية يتم تدمير الـ mutex.

- عرضت الوحدة طريقتين لتهيئة المتغير mutex هي : ساكن (Statically)،ديناميكي (Dynamically).
- اعتبرت الوحدة المتغيرات الشرطية نوعاً آخر يستخدم في تزامن الخيوط، ويعتمد التزامن في متغيرات الحالة على القيمة الفعلية للبيانات.
- قدمت الوحدة طريقتين لتهيئة المتغيرات : ساكن (Statically) ديناميكي (Dynamically).
- قدمت الوحدة الروتين pthread_condattr_init() والروتين pthread_condattr_destroy() لإنشاء وتدمير كائنات الصفات للمتغيرات الفرعية(condition variable attribute objects).

لمحة مسبقة عن الوحدة التالية

في الوحدة التالية سنتعلم كيف نكتب برامج متعددة الخيوط باستخدام مكتبة OpenMP والتي تعتبر بيئة لبرمجة الخيوط تعتمد على موجهات المترجمات (compiler directives)، وتختلف عن برمجة الخيوط التي تعلمناها مسبقاً في أن OpenMP هنا هي التي تنشيء الخيوط وليس المبرمج، وهذا يجعل برمجة الخيوط أكثر سهولة ويسراً.

إجابات التدريبات

تدريب (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *print_message_function( void *ptr );

main()
{
     pthread_t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int  iret1, iret2;

    /* Create independent threads each of which will execute
function */

     iret1 = pthread_create( &thread1, NULL,
print_message_function, (void*) message1);
     iret2 = pthread_create( &thread2, NULL,
print_message_function, (void*) message2);

     /* Wait till threads are complete before main continues.
Unless we  */
     /* wait we run the risk of executing an exit which will
terminate   */
     /* the process and all threads before the threads have
completed.   */

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);

     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);
     exit(0);
}

void *print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
}
```

سوف تكون النتيجة على التالي :

```
Thread 1
Thread 2
Thread 1 returns: 0
Thread 2 returns: 0
```

تدريب (2)

POSIX

```
pthread_mutex_lock(m)
......
pthread_mutex_unlock(m)
```

Win32

```
WaitForSingleObject(m)
ReleaseMutex(m)
```

OS/2

```
DosRequestMutexSem(m)
...

DosReleaseMutexSem(m)
```

تدريب (3)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main()
{
   int rc1, rc2;
   pthread_t thread1, thread2;

   /* Create independent threads each of which will execute
functionC */

   if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )
   {
      printf("Thread creation failed: %d\n", rc1);
   }

   if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )
   {
      printf("Thread creation failed: %d\n", rc2);
```

```
   }

   /* Wait till threads are complete before main continues.
Unless we  */
   /* wait we run the risk of executing an exit which will
terminate   */
   /* the process and all threads before the threads have
completed.   */

   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);

   exit(0);
}

void *functionC()
{
   pthread_mutex_lock( &mutex1 );
   counter++;
   printf("Counter value: %d\n",counter);
   pthread_mutex_unlock( &mutex1 );
}
```

سوف تكون النتيجة على النحو التالي :

Counter value: 1
Counter value: 2

مسرد المصطلحات

❖ **مكتبات POSIX thread (pthread)**

مكتبات POSIX thread هي واجهة تطبيقات برمجية لدعم الخيوط للغة C/C++. وهي تسمح لنا بكتابة برامج متعددة الخيوط، تعمل بالتوازي.

❖ **نموذج المدير/العامل (Manager/worker):**

من النماذج الشايعة للبرامج متعددة الخيوط حيث يقوم الخيط المدير بتوزيع العمل على الخيوط الأخرى، والتي تمثل العمال.

❖ **نموذج الأنبوب الانسيابي (pipeline):**

من النماذج الشايعة للبرامج متعددة الخيوط حيث يتم تقسيم المهمة إلى عمليات فرعية، بحيث تعتمد كل عملية على مخرجات العملية الأخرى، لكن تنفذ كل عملية في خيط منفصل (مثل تجميع السيارات).

❖ **الند (peer):**

من النماذج الشايعة للبرامج متعددة الخيوط، وهو يشبه نموذج المدير/العامل، لكنه يختلف في أن المدير بعد توزيعه المهام على العمال يشارك هو في التنفيذ.

❖ **الانضمام (Joining)**

هي إحدى الطرق لعمل تزامن (synchronization) بين الخيوط.

❖ **متغيرات الميوتكس (Mutex Variables)**

Mutex هي اختصار لـ "mutual exclusion". وتعتبر متغيرات Mutex من أهم التطبيقات التي تستخدم لتحقيق التزامن بين الخيوط (thread synchronization) وحماية البيانات المشتركة عند حدوث أكثر من أمر كتابة عليها (multiple writes).

❖ **متغيرات الحالة (Condition Variables)**

تعتبر متغيرات الحالة نوعاً آخر يستخدم في تزامن الخيوط (synchronization)، بينما يعتمد التزامن في mutexes على التحكم في وصول الخيط للبيانـــات، يعتمـــد التزامن في متغيرات الحالة على القيمة الفعلية للبيانات.

المراجع

POSIX Standard: www.unix.org/version3/ieee_std.html
"Pthreads Programming". B. Nichols et al. O'Reilly and Associates.

"Threads Primer". B. Lewis and D. Berg. Prentice Hall
Programming With POSIX Threads". D. Butenhof. Addison Wesley
www.awl.com/cseng/titles/0-201-63392-2

Pthreads tutorial , https://computing.llnl.gov/tutorials/pthreads/
POSIX thread (pthread) libraries, http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html


الوحدة السابعة
البرمجة باستخدام
OpenMP

محتويات الوحدة

الموضوع	الصفحة
المقدمة	193
تمهيد	193
أهداف الوحدة	194
توطئة	195
1. بعض موجهات Open MP الهامة	196
1.1 بعض دوال Open MP الهامة	196
2. تشارك البيانات	197
1.2 توزيع الحمل	199
2.2 عقبات في برمجة الخيط	201
3.2 تزامن الخيط Open MP	202
3. برنامج Open MP	204
1.3 موجهات Open MP	206
2.3 كم خيطاً نريد	210
3.3 معرفة عدد الخيوط	212
الخلاصة	213
إجابات التدريبات	215
مسرد المصطلحات	216
المراجع	219

المقدمة

تمهيد

عزيزي الدارس،

مرحباً بك عزيزي الدارس في الوحدة السابعة من مقرر أساليب البرمجة (3) والتي تحمل العنوان "البرمجة باستخدام OpenMP ".

لقد استخدمنا Pthreads كمكتبة لبرمجة الخيوط، وقبلها تعلمنا كيف نكتب برامج متعددة الخيوط معتمدة على لغة جافا (وهنالك مزيد من التفاصيل عن كتابة برامج متعددة الخيوط باستخدام جافا في الملاحق لمن يريد المزيد). سنختم هذه السلسلة المتعلقة بالخيوط في هذه الوحدة، حيث سنتعلم كيف نكتب برامج متعددة الخيوط باستخدام مكتبة OpenMP، والتي تعتبر بيئة لبرمجة الخيوط تعتمد على موجهات المترجمات (compiler directives). وتختلف عن برمجة الخيوط التي تعلمناها مسبقا في أن OpenMP هنا هي التي تنشيء الخيوط وليس المبرمج، وهذا يجعل برمجة الخيوط أكثر سهولة ويسراً.

تم تقسيم هذه الوحدة إلى ثلاثة أقسام رئيسة : الأول فيها يذكر بعض موجهات OpenMP، المهمة كذلك يعدد القسم بعض دوال OpenMP المهمة. في القسم الثاني تشارك البيانات نشرح كيف نستخدم OpenMP مع الأنظمة ذات الذاكرة المشتركة، وفي القسم نوضح كيف يتم توزيع الحمل في OpenMP، وعقبات في برمجة الخيط، وتزامن الخيط في OpenMP. أما القسم الثالث فهو يتناول برنامج OpenMP حيث سنوضح الشكل العام للبرنامج. ونقوم بكتابة الصيغة العامة لموجهات OpenMP. وفي القسم ستجد الإجابة عن السؤال كم خيطاً نريد ؟ وفي القسم أيضاً سنتعرف على الدالة التي تقوم بتحديد عدد الخيوط التي تعمل لديك الآن.

أهداف الوحدة



عزيزي الدارس، بعد فراغك من دراسة هذه الوحدة ينبغي أن تكون قادراً على أن:

- تذكر بعض موجهات OpenMP المهمة.
- تعدد بعض دوال OpenMP المهمة.
- تشرح كيف تستخدم OpenMP مع الأنظمة ذات الذاكرة المشتركة.
- توضح كيف يتم توزيع الحمل في OpenMP.
- تصف الشكل العام لبرامج OpenMP.
- تشرح بالأمثلة الصيغة العامة لموجهات OpenMP.
- تشرح كيف تعتمد OpenMP على عمليات التفريخ والتجميع.
- تكتب الدالة التي تتحكم في عدد الخيوط.
- تكتب الدالة التي تحدد عدد الخيوط التي تعمل لديك الآن.

توطئة

نبذة عن OpenMP

عزيزي الدارس، في الثمانينيات كان الحاسوب متعدد المعالجات (بعدد معالجات بسيطة) يكلف الآلاف الدولارات. ثم ظهرت أنواع متعددة المعالجات، مثل التي تحتوي على معالجين فقط وتكلف أقل من (5000) دولار. وقد تجد حاسوباً بثلاث معالجات يكلف أقل من 20000 دولار. كل هذا الآن لم نعد بحاجة إليه مع ظهور المعالجات متعددة النواة، والتي لا تزيد عن ألف دولار فقط وتعمل بأداء جيد. وقد انتشرت الآن في السوق في الحاسبات النقالة والحاسبات المكتبة والمخدمات.

يمكنك كتابة برامج متوازية تعمل على هذه الحاسبات باستخدام MPI مثلاً، و لكن يكون الأداء أفضل إذا استخدمنا لغات برمجة متوازية مخصصة للذاكرة المشتركة (Pthreads، OpenMP،..).

تعتبر OpenMP أحد الطرق المستخدمة لتوفير بيئة للبرمجة على الأجهزة ذات الذاكرة المشتركة، حيث توفر واجهة برمجة (API) للبرمجة المتوازية على الحاسبات متعددة المعالجات (multiprocessors) والحاسبات متعددة النواة (multi-core)، وهي تتألف من مجموعة من موجهات المترجمات (compiler directives)، ومكتبة من الدوال.

انتشرت OpenMP لسهولة استخدامها وإمكانية عملها مع العديد من اللغات؛ مثل C، ++C، و Fortran، حيث يمكن استخدامها مع هذه اللغات، أيضا نجدها مضمنة داخل Visual studio.Net.

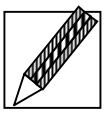
1. بعض موجهات OpenMP الهامة

Parallel	**تسبق شفرة (block of code) نريد تنفذه بالتوازي باستخدام عدد من الخيوط**
For	**تسبق حلقة تكراراتها غير معتمدة على بعضها (independent iterations) يمكن تقسيمها على عدة خيوط لتنفذ بالتوازي**
Parallel for	**استخدام موجهين مع بعض**
Sections	**تسبق سلسلة من الشفرات (series of blocks) قد تنفذ بالتوازي**
Parallel sections	**استخدام موجهين مع بعض**
Critical	**تسبق مقطع حرج**
single	**تسبق شفرة (code block) نريد تنفيذها بواسطة خيط واحد.**

1.1 بعض دوال OpenMP الهامة

Omp_get_num_procs	**تعطينا عدد المعالجات (CPUs) في الحاسب الذي ينفذ هذا الخيط**
Omp_get_num_threads	**تعطينا عدد الخيوط الفعالة (active threads) في هذه المنطقة (current region)**
Omp_get_thread_num	**يعطينا الرقم التعريفي للخيط (thread identification number)**
Omp_set_num_threads	**تسمح لك بتحديد عدد الخيوط التي تريد تنفيذها في الجزء من الشفرة**

تدريب (1)



اكتب نموذجاً مبسطاً يوضح كيفية استخدام ال parallel Region

أسئلة تقويم ذاتي



1. قدم نبذة مختصرة عن **OpenMP**.
2. اذكر بعض موجهات **OpenMP** الهامة.
3. عدِّد بعض دوال **OpenMP** الهامة.

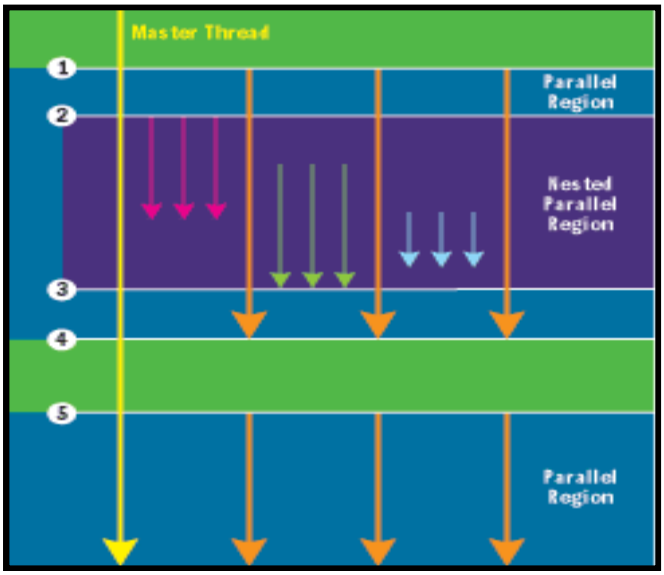
2. تشارك البيانات

عزيزي الدارس، تستخدم OpenMP مع الأنظمة ذات الذاكرة المشتركة حيث يمكن لعدة خيوط في عدة معالجات التشارك في متغيرات (shared variables)، ويستطيع أي معالج في النظام الوصول لمواقع الذاكرة نفسها (أي أن هنالك عنواناً حقيقياً (فيزيائي)، أي هنالك عنوان واحد تتشارك فيه كل المعالجات، مثلاً إذا كتب معالج القيمة (7) في العنوان 2020 فإن أي معالج آخر سيحصل على ذات القيمة إذا أحضر محتوى الذاكرة ذات العنوان 2020). هذا التشارك يمكِّن المعالجات من تبادل البيانات عبر المتغيرات المشتركة.

❐ تعتمد OpenMP على فكرة الخيط (thread).

من المعلوم أن أي برنامج يكون تحت التنفيذ (شغال) يسمى عملية (process)، ويقوم نظام التشغيل بحجز مساحة للعملية بالذاكرة. وكل عملية تحتوي على الأقل على خيط واحد، ويمكنها أن تحتوي على أكثر من خيط. كل عملية لديها مساحة بالذاكرة، ولا يمكن لعمليتين التشارك في مساحة واحدة بالذاكرة، لكن يمكن لمجموعة من الخيوط (في عملية واحدة) أن تتشارك في هذه المساحة بما في ذلك المتغيرات. وبالتالي يمكن لعدد من الخيوط أن تعمل معا في الوقت نفسه بالتوازي داخل العملية الواحدة.

طريقة الخيط هي الخيار المنطقي المتبع لبرمجة المعالجات المتوازية ذات الذاكرة المشتركة عن طريق التفريخ والتشارك (fork/join).



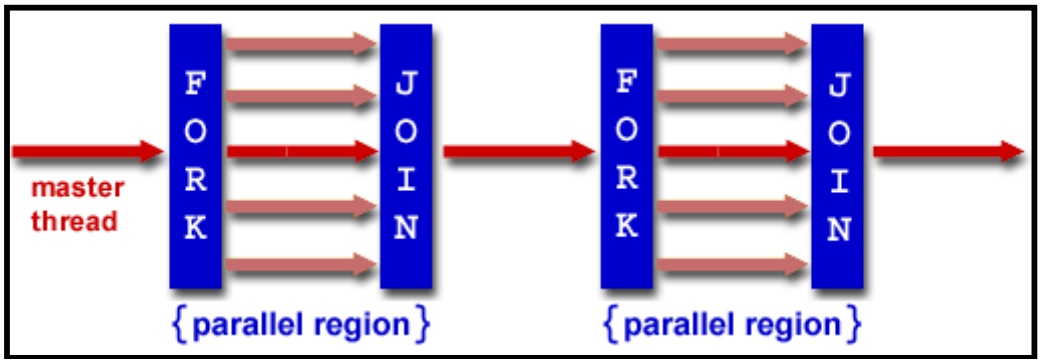
الخيط الرئيسي يعمل تسلسلياً إلى أن يجد أمر تفريخ لتوليد خيوط جديدة. هذه الخيوط المولدة يتم توزيعها على عدد من المعالجات لتنفذ بالتوازي فيها، ثم تجمع النتائج. يمكن توليد خيوط بعدد المعالجات الموجودة بالجهاز. ويستطيع المستخدم

تحديد عدد الخيوط التي يريدها في المتغير البيئي OMP_NUM_THREADS.

أو باستخدام الروتين

call omp_set_num_threads.

يمكن إنشاء وتوليد الخيوط أكثر من مرة، وكذلك تجميعها كما موضح بالشكل التالي:



1.2 توزيع الحمل (Load Balancing) في OpenMP

أ- من أهم المواضيع في التنفيذ المتوازي هو توزيع الجهد بين المعالجات، حيث يجب توزيع المهام بين المعالجات بالتساوي، بحيث يكون (تقريباً) لكل معالج جزء متساوٍ من العمل. توزيع العبء بين المعالجات بجعل كل الخيوط متساوية في زمن التنفيذ (لأن كل خيط سينفذ في معالج). وهو ما يسمى جدولة الخيط (thread scheduling). افتراضياً إذا انتهى خيط قبل بقية الخيوط في مجال شفرة (in region of code) فسينتظر البقية حتى تنتهي. وتعتبر هذه الطريقة غير جيدة لأنه سيكون هنالك معالجات لاتعمل (غير مستفاد منها) لأن عملها اكتمل قبل المعالجات الأخرى. ولحل هذه المشكلة توفر لنا OpenMP خيارات مختلفة لجدولة الخيط. أحد خيارات الجدولة في تكرار الحلقة (for loop)، هي:

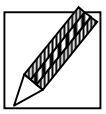
ب- كل الخيط ينفذ (ن) حلقة، وينتظر بقية الخيوط إلى أن تكمل (ن) حلقة. جدولة ساكنة (static scheduling).

ت- الجدولة الديناميكية (dynamic scheduling): نسند (ن) حلقة من الحلقات التي لم تنفذ للخيوط العاطلة (idle).

ث- Guided scheduling: كل خيط ينتهي من عمله نسند إليه عدد حلقات يساوي عدد الحلقات الباقية مقسوماً على عدد الخيوط.

قد يظهر أن الجدولة الديناميكة هي الأفضل، ولكن ذلك ليس صحيحاً دوماً، فعملية إسناد حلقات إضافية للخيط يبطئ عملية التنفيذ ككل، لذلك فإن عملية إيجاد (ن) مناسب للجدولة الديناميكية وصعب وعسير.

تدريب (2)



توفر لنا OpenMP خيارات مختلفة لجدولة الخيط. أحد خيارات الجدولة في تكرار الحلقة (for loop) هي الجدولة الساكنة (static scheduling)و الجدولة الديناميكية (dynamic scheduling) وضِّح الفرق بين الخيارين مع التوضيح بمثال(برنامج مبسط).

2.2 عقبات في برمجة الخيط

عزيزي الدارس، هنالك بعض المشاكل التي تقابلنا في برمجة الخيط لم تكن موجودة في البرمجة التقليدية (التسلسلية)، مثل مشكلة حالة السباق (race condition). وهذه المشكلة تظهر عندما يعدل أكثر من خيط قيمة متغير واحد مشترك.

<u>مثال (1):</u>

إذا أردنا تنفيذ هذا الكود

```
balance = getBalance(loanID)
rate = getRate(loanID)
setBalance(loanID, balance * (1.0 + rate))
```

عندما ينفذ خيطين (T1, T2) نفس الشفرة أعلاه، فلن تتوقع كيف سينفذ كل خيط الشفرة أعلاه، إحدى حالات التنفيذ المتوقعة؛ هي:

```
T1: balance = getBalance(loan1)
T1: rate = getRate(loan1)
T2: balance = getBalance(loan2)
T1: setBalance(loan1, balance * (1.0 + rate))
```

لأن كل خيط سينفذ شفرته في معالج مختلف وبالتالي قد لا تعرف ترتيب التنفيذ.

<u>مثال (2):</u>

T1	T2

Void foo(){
X++;
Y = x;
}
	Void foo(){
	Y++;
	X+=3;
	}

إذا بدأنا بقيمة y=0 و x=6، ما قيمة x و y بعد تنفيذ خيط T1 وخيط T2 ؟ حيث x و y هما متغيران مشتركان بين الخيوط أعلاه.

هنالك العديد من النتائج المحتملة :

فقد يكون 7، 9، 10 للمتغير x، وقد يكون 1، 7، 10 للمتغير y، ولاندري أي تباديل ستنتج من هذه القيم. هذه المشكلة (أعلاه) ناتجة من أن هنالك خيطين ينفذان شفرة تعرف بالمقطع الحرج (critical section).أحد طرق التزامن المستخدمة لحل هذه المشكلة هو عملية الغلق (lock). عندما يقوم خيط بعملية الغلق في المقطع الحرج سيمنع بقية الخيوط من تنفيذ شفرات على هذا القاطع. والخيط الذي يريد أن يعمل غلق عليه الانتظار حتى يحرر الغلق (أي لا يكون هنالك أي خيط آخر قام بعملية الغلق).

معالجة مشكلة حالة السباق (race condition) قد تقودنا إلى مشكلة جديدة هي الإختناق (deadloack). وينتج الإختناق إذا كان هنالك خيط T1 وخيط T2 :

- الخيط T1 قام غلق على المقطع الحرج S2.
- الخيط T2 قام بعملية غلق على المقطع الحرج S1

ثم أحتاج الخيط T1 استخدام المقطع الحرج S1 وفي الوقت نفسه أراد الخيط T2 استخدام القاطع الحرج S2، سيظل كل خيط في إنتظار الخيط الآخر أن يترك مقطعه الحرج، وسيظل الأمر هكذا، مما يسبب اختناق.

3.2 تزامن الخيط في OpenMP

توفر OpenMP إمكانيات للتزامن تيسر على المبرمج تجاوز مشاكل تشارك البيانات، فيمكنك تحديد المقطع الحرج بالامر #pragma critical. ويتم تنفيذ المقطع الحرج بخيط واحد فقط في اللحظة المعينة. هذا سيمنع الوقوع في مشكلة حالة السباق (race conditions) في هذا المقطع.

أيضاً يمكنك جعل بعض المتغيرات محلية بالأمر #pragma local(list) أو #pragma threadprivate (list)، حيث تمثل list قائمة المتغيرات التي نريدها أن تكون محلية.

التوجيه #pragma reduction (op: list) يجعل كل متغير في list محلي، ولكن عندما يتنهي الخيوط من تنفيذ شفرات المجال (region) الذي تم تعريف هذه المتغيرات به، ستصل نسخة من كل متغير محلي شارك في العملية. بهذه الطريقة سيستلم كل خيط نسخة نهائية من الحسابات (computational) التي تتم في هذا المجال (region).

أسئلة تقويم ذاتي



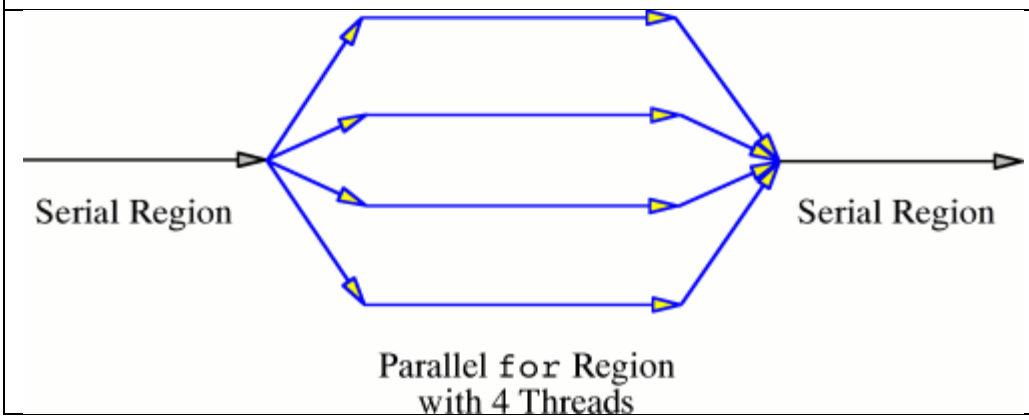
1. اشرح كيف تستخدم OpenMP مع الأنظمة ذات الذاكرة المشتركة.
2. وضِّح كيف يتم توزيع الحمل (Load Balancing) في OpenMP ؟
3. وضِّح مفهوم جدولة الخيط (thread scheduling).

3. برنامج OpenMP

الشكل العام للبرنامج:

```
C / C++ - General Code Structure
#include <omp.h>

main () {

int var1, var2, var3;

Serial code
   .
   .
   .

Beginning of parallel section. Fork a team of threads.
Specify variable scoping

#pragma omp parallel private(var1, var2) shared(var3)
 {

 Parallel section executed by all threads
    .
    .
    .

 All threads join master thread and disband

 }

Resume serial code
```

```
    .
    .
    .

}
```


Serial Region
Serial Region
Parallel for Region
with 4 Threads

1.3 موجهات OpenMP

OpenMP Directives

الصيغة العامة:

#pragma omp	directive-name	[clause,...]	Newline
Required for all OpenMP C/C++ directives.	**A valid OpenMP directive. Must appear after the pragma and before any clauses.**	**Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.**	**Required. Precedes the structured block which is enclosed by this directive.**

مثلا:

```
#pragma omp parallel default(shared)
private(beta,pi)
```

يوفر OpenMP التوازي على مستوى المهام (task parallelism)، وقد يستخدم حلقة loop لتوزيع المهام.

الشفرة التالية هي شفرة بلغة C لجمع 100 رقم، وهي شفرة برنامج تقليدي (تسلسلي)، حيث سيتم جمع كل عدد في حلقة:

```
int main(int argc, int *argv[]) {
  int i, intArray[100], sum = 0;
        initArray(intArray, 100);
        for (i=0;i<100;i++) {
         sum = sum + intArray[i];
        }
}
```

لتنفيذ الشفرة أعلاه بالتوازي على حاسوب به أربعة معالجات مثلاً، يمكننا جعل كل خيط يجمع (25) عدد (في معالج منفصل) وبالتالي سيكون لدينا أربعة خيوط، كل خيط ينفذ مهمة.

حيث سنقوم بالتالي:

① **تقسيم العملية (جمع 100 عدد) إلى أربع مهام صغيرة (جمع 25 عدد).**

② **توزيع كل مهمة على خيط (توليد خيوط – fork)**

③ **تنفيذ كل خيط في معالج**

④ **تجميع النتيجة (تجميع النتائج – join)**

⑤ **تدمير الخيوط بعد حصول الخيط الرئيسي على النتائج.**

لتعديل الشفرة التقليدية أعلاه لتنفذ بالتوازي على OpenMP سنقوم فقط بإضافة ثلاثة موجهات (directives) للشفرة أعلاه، وهي تجعل المترجم ينفذ الشفرة بالتوازي :

❶ **#pargma parallel for: تخبر المترجم بتنفيذ هذا التكرار بالتوازي**

❷ **#pragma threadprivate(i): إنشاء متغير محلي i في كل معالج (تختلف قيمة i في كل معالج، مثلا بالمعالج الأول تكون من 0 إلى 24 وبالمعالج الثاني تكون من 25 إلى 49 وبالثالث من 50 إلى 74، وفي الرابع من 75 إلى 99).**

❸ **#pragma reduction(+:sum): ننشئ متغيراً محليَّاً sum بكل معالج، ونحصل على قيمته النهائية، وبتجميع كل القيم النهائية للأربعة متغيرات sum، وبجمعها باستخدام العملية + لنحصل على الناتج الأخير.**

الشفرة المتوازية ستكون كما يلي:

```
int main(int argc, int *argv[]) {
 int i, intArray[100];
  int sum = 0;
 /* Store some values in intArray. */
  initArray(intArray, 100);
#pragma parallel for    /* Make for loop a parallel region*/
#pragma threadprivate(i)
      #pragma reduction(+: sum)
         for (i=0;i<100;i++) {
           sum = sum + intArray[i];
         }
       }
```

ثم قبل تنفيذ هذا البرنامج لابد من وضع القيمة 4 في المتغير البيئي OMP_NUM_THREADS لنوضح أننا نريد تنفيذ هذا التكرار بالتوازي في أربعة خيوط.

❐ عزيزي الدارس، ، تعتمد **OpenMP** على عملية التفريخ والتجميع كما ذكرنا مسبقاً (**fork/join**) وهي كالآتي:

① **يبدأ البرنامج بخيط واحد رئيسي.**

② **يُولّد هذا الخيط الرئيسي عدداً من الخيوط الفرعية (*fork*).**

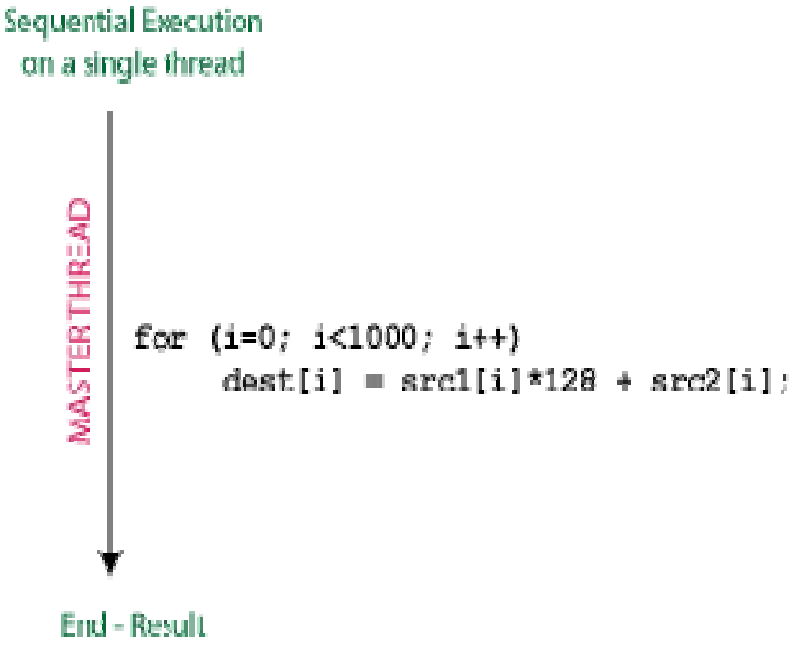
③ **تعمل الخيوط المولدة في معالجات مختلفة بالتوازي مع الخيط الرئيسي**

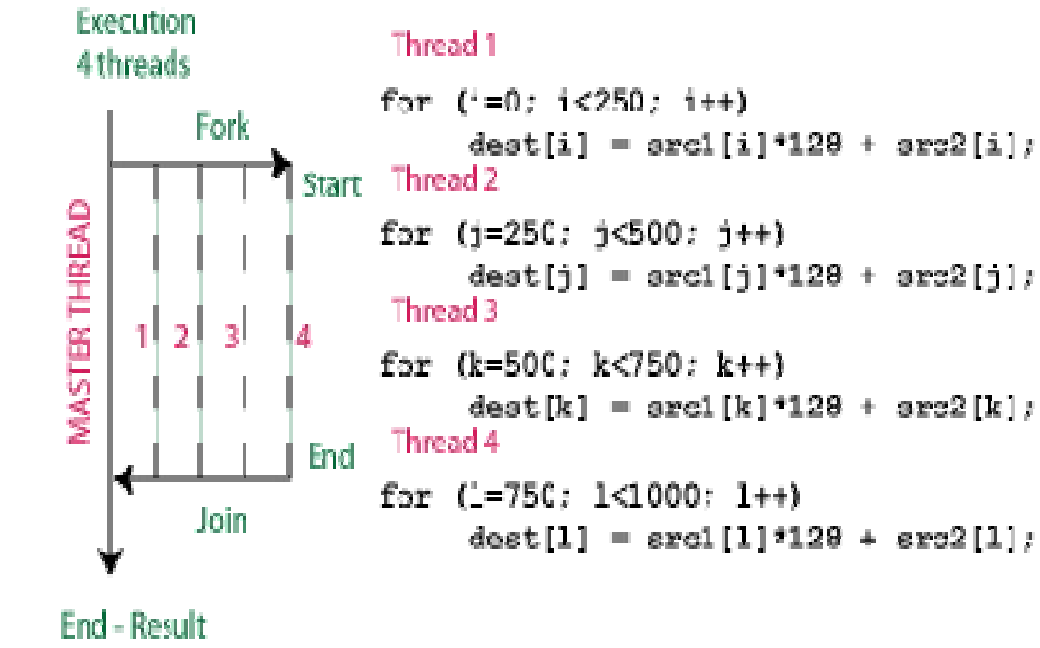
④ **إذا انتهى خيط سينتظر بقية الخيوط حتى تكتمل.**

⑤ **عندما تكتمل كل الخيوط، ويتم إكتمال العملية، تجمع النتائج، وتدمر كل الخيوط ما عدا الرئيسي.**

⑥ **عملية تقسيم الخيوط وتجميعها تحتوي على تزامن تقوم به OpenMP للحصول على النتيجة الرئيسية.**

مثال:


Sequential Execution
on a single thread
MASTER THREAD
for (i=0; i<1000; i++)
dest[i] = src1[i]*128 + src2[i];
End - Result


Execution
4 threads
Fork
Start
MASTER THREAD
1
2
3
4
End
Join
End - Result
Thread 1
for (i=0; i<250; i++)
dest[i] = src1[i]*128 + src2[i];
Thread 2
for (j=250; j<500; j++)
dest[j] = src1[j]*128 + src2[j];
Thread 3
for (k=500; k<750; k++)
dest[k] = src1[k]*128 + src2[k];
Thread 4
for (l=750; l<1000; l++)
dest[l] = src1[l]*128 + src2[l];

2.3 كم خيطاً نريد؟

عدد الخيوط التي تريدها يعتمد على عدد المعالجات لديك. انشاء الخيوط وتدميرها بعد تجميع النتائج لا يستغرق وقتاً طويلاً. إذا كانت المعالجات قليلة وعدد الخيوط أكبر سيستغرق التنفيذ وقتا أطول، لأن هنالك زمن إضافيًّاً استخدم لإنشاء الخيوط والانتقال بين الخيوط، لأن هذا الانتقال يتطلب عمليات حفظ وتحميل بياناته من وإلى الذاكرة.

القاعدة الهامة هي بما أن كل الخيوط تعمل على عملية واحدة، فالأولوية واحدة، وبالتالي يكفي تشغيل خيط واحد في كل معالج وكلما زادت عدد المعالجات زاد عدد الخيوط.

معظم موجهات المترجمات في *OpenMP* تستخدم المتغير البيئي (*environment varaible*) المسمى *OMP_NUM_THREADS* لتحديد عدد الخيوط التي تريد إنشاءها. يمكنك التحكم في عدد الخيوط بالدالة التالية:

```
// Get the number of processors in this system
int iCPU = omp_get_num_procs();
// Now set the number of threads
omp_set_num_threads(iCPU);
```

يمكن وضع أي قيمة في المتغير *iCPU* إذا كنت لا تريد استخدام الدالة *mp_get_num_procs()*. ويمكنك استدعاء الدالة *omp_set_num_threads* كما تريد في أجزاء عديدة من شفرتك. إذا لم تستخدم هذه الدالة سيستخدم *OpenMP* المتغير البيئي *OMP_NUM_THREADS*.

- **تنفيذ التكرار *for* بالتوازي**

لنعيد صياغة المثال السابق ليعمل على *visual c++* كما يلي:

```
omp_set_num_threads(threads);

 int i, intArray[100];
  int sum = 0;
 /* Store some values in intArray. */
  initArray(intArray, 100);

#pragma omp parallel for
 #pragma threadprivate(i)
      #pragma reduction(+: sum)
         for (i=0;i<100;i++) {
           sum = sum + intArray[i];
         }
       }
```

هنا استخدمنا الموجه #pragma omp parallel لجعل التكرار ينفذ بالتوازي حسب عدد الخيوط المحدد بواسطة الدالة (omp_set_num_threads(threads.

تمرين

نفذ هذا البرنامج بالطريقة المتسلسلة، وباستخدام OpenMP مرة في حاسب ذي معالج واحد، ومرة في حاسب متعدد النواة، وقارن الأداء بين أربع حالات. غير عدد الخيوط، ولاحظ الفرق في سرعة التنفيذ.

3.3 معرفة عدد الخيوط

عزيزي الدارس، يمكنك استدعاء الدالة التالية لمعرفة عدد الخيوط التي تعمل لديك الآن:

int omp_get_thread_num();.

باستخدام **OpenMP** يمكنك الاستفادة من الحاسبات متعددة النواة، أو متعددة المعالجات. بدون التوازي، حتى ولو كان لديك حاسوب ذو مائة معالج أو مائة نواة فلن تستفيد منه. وستكون كمن يستخدم سيارة رباعية الدفع في وسط مدينة مزدحمة ومعبدة الطرق.

أسئلة تقويم ذاتي



1. صِف الشكل العام لبرنامج **OpenMP** .
2. اشرح الصيغة العامة لموجهات **OpenMP**.
3. اشرح كيف تعتمد **OpenMP** على عملية التفريخ والتجميع (**fork/join**).
4. وضّح العلاقة بين عدد الخيوط وعدد المعالجات.
5. اكتب الدالة التي تتحكم في عدد الخيوط.
6. اكتب الدالة التي تحدد عدد الخيوط التي تعمل لديك الآن.

الخلاصة

- عرضت الوحدة بعض موجهات OpenMP المهمة :

Parallel	تسبق شفرة (block of code) نريد تنفذه بالتوازي باستخدام عدد من الخيوط
For	تسبق حلقة تكراراتها غير معتمدة على بعضها (independent iterations) يمكن تقسيمها على عدة خيوط لتنفذ بالتوازي
Parallel for	استخدام موجهين مع بعض
Sections	تسبق سلسلة من الشفرات (series of blocks) قد تنفذ بالتوازي
Parallel sections	استخدام موجهين مع بعض
Critical	تسبق مقطع حرج
single	تسبق شفرة (code block) نريد تنفيذها بواسطة خيط واحد.

- عرضت الوحدة بعض دوال OpenMP الهامة:

Omp_get_num_procs	تعطينا عدد المعالجات (CPUs) في الحاسب الذي ينفذ هذا الخيط
Omp_get_num_threads	تعطينا عدد الخيوط الفعالة (active threads) في هذه المنطقة (current region)
Omp_get_thread_num	يعطينا الرقم التعريفي للخيط (thread identification number)
Omp_set_num_threads	تسمح لك بتحديد عدد الخيوط التي تريد تنفيذها في الجزء من الشفرة

- بينت الوحدة OpenM مع الأنظمة ذات الذاكرة المشتركة، حيث يمكن لعدّة خيوط في عدة معالجات التشارك في متغيرات ويستطيع أي معالج في النظام الوصول لمواقع

الذاكرة نفسها أي أن هناك عنواناً واحداً تشارك فيه كل المعالجات، هذا التشارك يمكّن المعالجات من تبادل البيانات عبر المتغيرات المشتركة.

– قدمت الوحدة خيارات الجدولة في تكرار الحلقة (for loop) هي:

أ–كل الخيط ينفذ (ن) حلقة، وينتظر بقية الخيوط إلى أن تكمل (ن) حلقة. (جدولة ساكنة static scheduling).

ب– الجدولة الديناميكية (dynamic scheduling): نسند (ن) حلقة من الحلقات التي لم تنفذ للخيوط العاطلة (idle).

ت– Guided scheduling: كل خيط ينتهي من عمله نسند إليه عدد حلقات يساوي عدد الحلقات الباقية مقسوماً على عدد الخيوط.

– ذكرت الوحدة أن *OpenMP* تعتمد على عملية التفريخ والتجميع (*fork/join*) وهي كالآتي: يبدأ البرنامج بخيط واحد رئيسي– يولد هذا الخيط الرئيسي عدداً من الخيوط الفرعية (*fork*)– تعمل الخيوط المولدة في معالجات مختلفة بالتوازي مع الخيط الرئيسي – إذا انتهى خيط سينتظر بقية الخيوط حتى تكتمل – عندما تكتمل كل الخيوط ويتم اكتمال العملية تجمع النتائج وتدمر كل الخيوط ما عدا الرئيسي – عملية تقسيم الخيوط وتجميعها تحتوي على تزامن تقوم به *OpenMP* للحصول على النتيجة الرئيسية.

– وضّحت الوحدة أنه يمكن التحكم في عدد الخيوط بالدالة التالية:

```
// Get the number of processors in this system
int iCPU = omp_get_num_procs();
// Now set the number of threads
omp_set_num_threads(iCPU);
```

لمحة مسبقة عن الوحدة التالية

كل الوحدات السابقة ركزت على برمجة الذاكرة المشتركة، وذلك باستخدام الخيوط، ولكن ماذا بشأن الحواسيب التي ليست لها ذاكرة مشتركة. هذا ما سنتحدث عنه في الوحدة التالية، حيث سنتناول برمجة الذاكرة الموزعة باستخدام تبادل الرسائل، وسنستخدم مكتبة من أشهر المكتبات في هذا المجال، وهي مكتبة "واجهة تبادل الرسائل" (MPI).

إجابات التدريبات

تدريب (1)

```
#include <stdio.h>
#include <omp.h>
main(){
   int a[2]={-1,-1};
#pragma omp parallel
   {
      a[omp_get_thread_num()] = omp_get_thread_num();
   }
   printf("a[0] = %d, a[1] = %d",a[0],a[1]);
```

تدريب (2)

1. جدولة ساكنة (**static scheduling**) : كل خيط ينفذ (ن) حلقة وينتظر بقية الخيوط إلى أن تكمل (ن) حلقة.

2. الجدولة الديناميكية (**dynamic scheduling**): نسند (ن) حلقة من الحلقات التي لم تنفذ للخيوط العاطلة (**idle**).

مثال :

```
1.    //-------------------------------------------------------
-------
2.    #include <omp.h>
3.    #include <valarray>
4.    #define N 1000
5.    #define CHUNKSIZE 100
6.
7.    int main() {
8.      int i,chunk;
9.      std::valarray<double> a(N);
10.     std::valarray<double> b(N);
11.     std::valarray<double> c(N);
12.
13.     for(i=0; i<N; i++)  a[ i ]  = b[ i ] = i * 1.0;
```

```
14.     chunk = CHUNKSIZE;
15.
16.   #pragma omp parallel shared(a,b,c,chunk) private(i) {
17.   #pragma omp for schedule(dynamic,chunk) nowait
18.       for(i=0; i<N; i++) c[ i ] = a[ i ] + b[ i ];
19.     }
20.   }
21.   //-----------------------------------------------------------
-------
22.
```

* فى مكان *dynamic* يمكن أن نستخدم static بدلاً عنه

مسرد المصطلحات

❖ **OpenMP**

تعتبر OpenMP أحد الطرق المستخدمة لتوفير بيئة للبرمجة على الأجهزة ذات الذاكرة المشتركة، حيث توفر واجهة برمجة (API) للبرمجة المتوازية على الحاسبات متعددة المعالجات (multiprocessors) والحاسبات متعددة النواة (multi-core). وهي تتألف من مجموعة من موجهات المترجمات (compiler directives)، ومكتبة من الدوال.

انتشرت OpenMP لسهولة استخدامها وإمكانية عملها مع العديد من اللغات؛ مثل C، C++، و Fortran. حيث يمكن استخدامها مع هذه اللغات، أيضا نجدها مضمنة داخل Visual studio.Net.

❖ **توزيع الحمل (Load Balancing) في OpenMP**

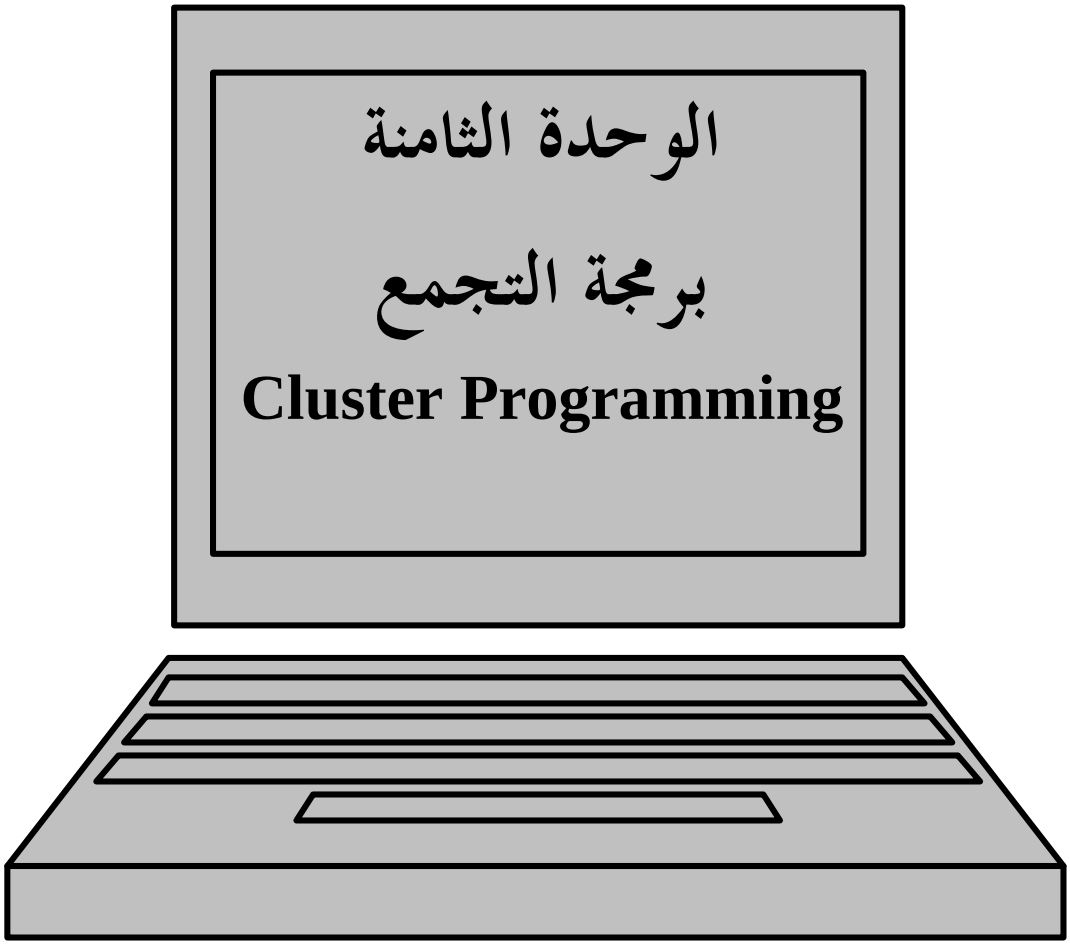
هو توزيع الجهد بين المعالجات، حيث يجب توزيع المهام بين المعالجات بالتساوي، بحيث يكون (تقريباً) لكل معالج جزء متساوٍ من العمل.

المراجع

POSIX Standard: www.unix.org/version3/ieee_std.html
"Pthreads Programming". B. Nichols et al. O'Reilly and Associates.

"Threads Primer". B. Lewis and D. Berg. Prentice Hall
Programming With POSIX Threads". D. Butenhof. Addison Wesley
www.awl.com/cseng/titles/0-201-63392-2

Pthreads tutorial , https://computing.llnl.gov/tutorials/pthreads/
POSIX thread (pthread) libraries,
http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html


الوحدة الثامنة
برمجة التجمع
Cluster Programming

محتويات الوحدة

الموضوع	الصفحة
المقدمة	223
تمهيد	223
أهداف الوحدة	224
توطئة	225
1. نبذة عن MPI	227
2. أمثلة برمجية	228
1.2 أول برنامج	228
2.2 البرنامج الثاني	229
3.2 البرنامج الثالث	234
3. الدالة	238
4. تبادل الرسائل في MPI	243
1.4 الاتصال الحجوز	243
2.4 الاختناق	245
3.4 الاتصال غير المتزامن (غير المحجوز)	247
الخلاصة	250
لمحة مسبقة عن الوحدة التالية	252
إجابات التدريبات	253
مسرد المصطلحات	255
المراجع	256

المقدمة

تمهيد

مرحباً بك عزيزي الدارس، في الوحدة الثامنة من مقرر "أساليب البرمجة 3" وهي بعنوان "برمجة التجمع clusters programming " تم تقسيم هذه الوحدة إلى أربعة أقسام رئيسة الأول منها يعطي نبذه عن MPI، وهي المكتبة الأكثر انتشاراً المستخدمة في تبادل الرسائل. في القسم الثاني نستعرض بعض الأمثلة البرمجية، وفيها نوضح مفاهيم البرمجة المتوازية في التجمع، ويأتي القسم الثالث تحت عنوان قياس الأداء، وفيه نستعرض أحد طرق قياس أداء البرنامج هو حساب الزمن من بداية البرنامج وحتى نهايته.

في القسم الرابع والأخير تتناول تبادل الرسائل في MPI، وهنا سنتحدث عن الاتصال المحجوز، الاختناق وبعض من مسبباته، الاتصال غير المتزامن (غير المحجوز).

أهداف الوحدة

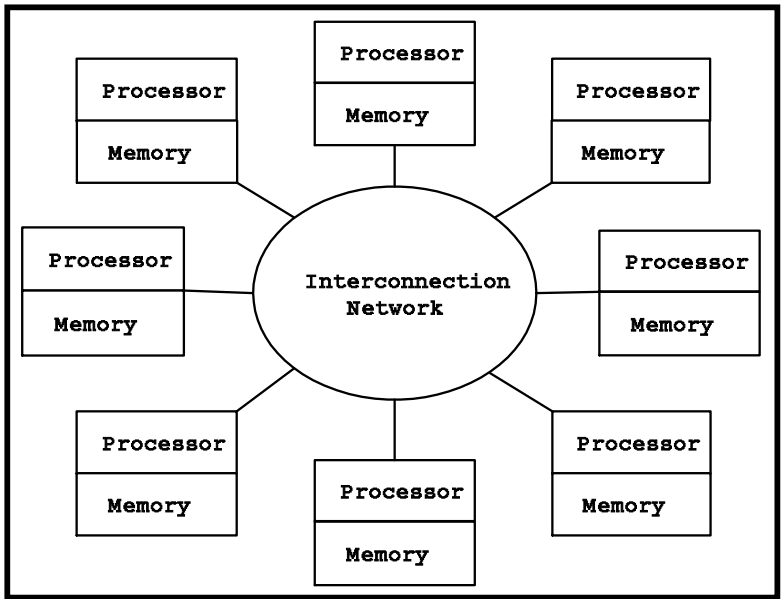


عزيزي الدارس، بعد فراغك من دراسة هذه الوحدة ينبغي أن تكــون قادراً على أن:

- **تذكر بعض من دوال MPI الرئيسة.**
- **تعدد أنواع البيانات في MPI.**
- **توضح أنواع عمليات الاختزال المبنية في MPI.**
- **تذكر دوال لقياس الزمن في MPI.**
- **تكتب الصيغة العامة لدالة الحاجز.**
- **تشرح دالتي الإرسال والاستقبال المستخدمتان في البرنامج.**
- **تذكر بعض من مسببات الاختناق.**
- **تكتب الصيغة العامة للدالة MPI_Send.**
- **تكتب الصيغة العامة للدالة MPI_Recv.**
- **تكتب الصيغة العامة للإرسال غير المحجوز.**
- **تكتب الصيغة العامة للاستقبال غير المحجوز.**

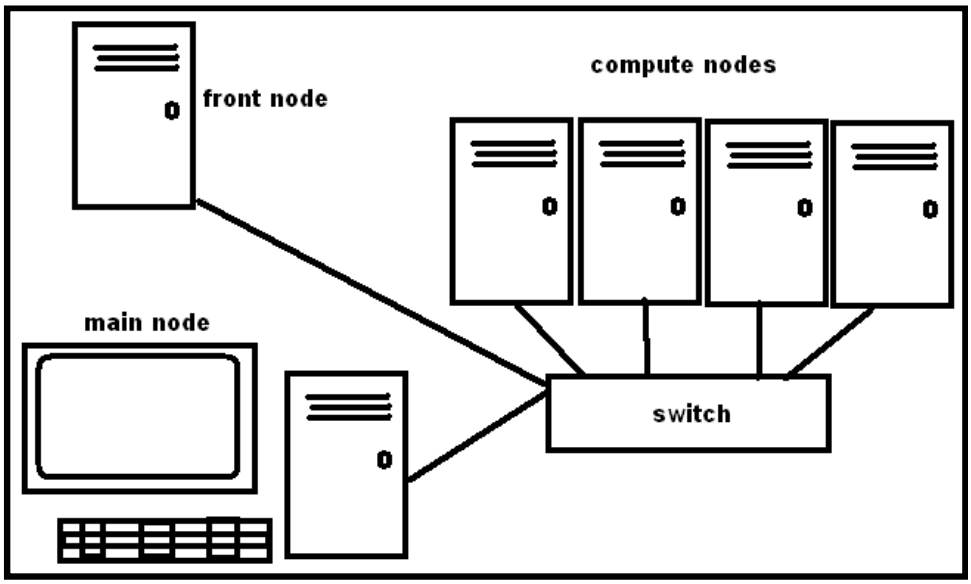
توطئة

عزيزي الدارس، كل الوحدات السابقة ركزت على برمجة الذاكرة المشتركة وذلك باستخدام الخيوط، سواء أكان ذلك باستخدام لغة برمجة تدعم ذلك مثل الجافا أو مكتبة تضاف للغة برمجة مثل Pthreads و OpenMP. ولكن ماذا بشأن الحواسيب التي ليس لها ذاكرة مشتركة (الحواسيب ذات الذاكرة الموزعة). هذه الوحدة جاءت لهذا الغرض. حيث سنتحدث هنا عن برمجة الذاكرة الموزعة باستخدام تبادل الرسائل (Message Passing) وسنستخدم مكتبة من أشهر المكتبات في هذا المجال، ألا وهي مكتبة "واجهة تبادل الرسائل"، والتي تسمى MPI أختصارات للكلمات Message Passing Interface.



ذكرنا في حديثنا عن العتاد أن الأجهزة متعددة الحواسيب (multicomputer) هي حواسيب مرتبطة ببعضها، ولكن لكل حاسوب ذاكرة خاصة بها، وإذا أرادت هذه الحواسيب التشارك في البيانات سيتم ذلك عبر تبادل الرسائل. من أمثلة الأجهزة متعدد الحواسيب، الحواسيب الفائقة السرعة (super computers)،

وهي كما نعلم باهظة الثمن **وتكلف ملايين الدولارات، ولا يمكن لجهات عادية امتلاكها. ولكن للفقراء حل، حيث يمكن ربط مجموعة من الحواسيب الشخصية العادية بشبكة سريعة وتهيئتها لتعمل كحاسوب فائق السرعة، وذلك بتثبيت بعض البرامج، والتي من بينها واجهة تبادل الرسائل MPI. أيضا من أمثلة الأجهزة متعددة الحواسيب، هذا الحاسوب الفائق السرعة الافتراضي يسمى أحيانا تجمع (cluster)، وقد نستخدم فيه الأجهزة القديمة الغير مستخدمة. وقد يسمى هنا تجمع Beowulf، حيث تستخدم نظم تشغيل مفتوحة المصدر ومجانية في ذلك، مما يجعل تكلفة التجمع قليلة جدًّا. التجمع يجب أن يكون في مكان واحد، حيث تربط الأجهزة بشبكة محلية سريعة ويكون هنالك حاسوب أو أكثر للتعامل مع التجمع، بينما تكون بقية الأجهزة تنفيذ ما يصلها من طلبات من هذه الحواسيب، وغالباً لا نحتاج إلى شاشات للمحطات العاملة (انظر الشكل أدناه).**



أيضا يمكنك تهيئة شبكة محلية عادية لتعمل كتجمع، ثم تستخدم بعض الأجهزة للتحكم وإرسال مهام للبقية، الفرق بين التجمع الحقيقي والتجمع الذي يستخدم أجهزة في شبكة محلية موجودة هو أن الأول يستفيد من إمكانيات كامل المحطات، بينما

الثاني يستخدم حواسيب يستخدمها غيره، وبالتالي سيستفيد من وقت تعطلها (عند مـــا لا يكون عليها أحد أو عندما يكون مستخدمها يكتب أو يتصفح موقع لايحتاج للمعـــالج في كثير عمل.

عادة يتكون التجمع من حواسيب متجانسة وتثبت فيها نظم تشغيل متشابها.

سيكون هنالك ملحق عن كيفية تهيئة واستخدام MPI في visual studio 2008.

1. نبذة عن MPI

هي المكتبة الأكثر انتشاراً المستخدمة في تبادل الرسائل، والتي تعمل مع العديد من اللغات مثل C و Fortran و Visual C++. Net. ويمكن استخدامها في برمجة التجمعات (clusters) والشبكات المحلية، وغيرها من الحاسبات المتعددة.

في أواخر الثمانينيات تم تطوير Parallel Virtual Machine (PVM) في المعمل القومي Oak Ridge، ثم بدأ وضع معايير MPI في 1992، ثم ظهرت النسخة الأولى منها في 1994، وكانت النسخة الثانية جاهزة في عام 1997، وأصبحت MPI اليوم هي المكتبة الرئيسية والأكثر انتشاراً واستخداماً في مجال تبادل الرسائل.

تعتبر MPI متنقلة وسهلة، حيث رغم أنها تحتوي على ما يزيد على 125 دالة، لكن يمكنك فعل الكثير بست دوال رئيسية منها؛ وهي :

MPI_INIT
MPI_FINALIZE
MPI_COMM_SIZE
MPI_COMM_RANK
MPI_SEND
MPI_RECV

سنتحدث في بقية هذه الوحدة عن هذه الدوال ودوال أخرى، مع توضيحها بأمثلة برمجية توضح مفاهيم البرمجة المتوازية في التجمع.

أسئلة تقويم ذاتي

1. **قدّم نبذة مختصرة عن MPI.**
2. **اذكر بعض من دوال MPI الرئيسة.**

2. أمثلة برمجية

1.2 أول برنامج

```
"mpi.h"#include
#include <stdio.h>
int main(int argc, char *argv[])
{
MPI_Init(&argc, &argv);
 printf("Hello, world!\n");
MPI_Finalize();
Return 0;
                                    }
```

شرح البرنامج أعلاه

- **#include "mpi.h" : توفر تعريفات وأنواع MPI الأساسية**
- **MPI_Init: تهيئة MPI لنستخدمها في البرنامج.**
- **MPI_Finalize: تحرير الموارد التي حجزها برنامج MPI**
- **تعتبر كل الروتينات الأخرى (غير روتينات MPI) محلية، وتنفذ نسخة في كل عملية، مثلاً printf تطبع رسالة في كل عملية.**
- **دوال MPI تعطي (return) MPI_SUCCESS في حالة تنفيذها بنجاح، أما إذا لم تنجح فستعطي رسالة خطأ.**
- **افتراضياً أي خطأ يتسبب في توقيف كل العمليات (abort).**

- المستخدم يمكن أن يكون لديه روتينات تناول الأخطاء الخاصة به (error handling routines).
- هنالك custom error handlers متاحة على الانترنت ويمكن تحميلها من هناك.

شرح دوال MPI المستخدمة في البرنامج أعلاه

- روتين تهيئة MPI_Init هو أول روتين يجب استدعائيه في برامج MPI، ويستدعى مرة واحدة فقط، الصيغة العامة لهذا الروتين؛ هي:

```
MPI_Init(&argc, &argv);
```

- روتين الانتهاء MPI_Finalize هو آخر روتين يتم استدعائية وهو يقوم بتحرير الموارد التي تم حجزها بواسطة البرنامج مثل الذاكرة. صيغة هذا الروتين؛ هي:

```
MPI_Finalize();
```

2.2 البرنامج الثاني

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
int rank, size;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
printf("I am %d of %d\n", rank, size);
MPI_Finalize();
return 0;
}
```

شرح البرنامج

عندما نهيئ MPI بالأمر MPI_Init ستصبح كل عملية فعالة (active process) عضو في مجتمع متصل (communicator) يسمى MPI_COMM_WORLD. حيث يعتبر هذا المجتمع (communicator) كائن يوفر بيئة لتبادل الرسائل بين العمليات الأعضاء فيه. MPI_COMM_WORLD هو المجتمع الافتراضي الذي يمكنك الحصول عليه مجانياً. وهو كافياً لمعظم برامجنا. ويمكن بناء مجتمعاتك الخاصة إذا أردت تجزيئة العمليات إلى مجموعات مستقلة.

العمليات داخل المجتمع مرتبة، فإذا كان لديك مجتمع بعدد (ع) عملية، فكل عملية سيكون لها رتبة مفردة غير متكررة بين (صفر و ن−1) قد تستخدام العملية رتبتها لمعرفة أي عمل أو بيانات هي مسئولة عنه.

تستخدم الدالة MPI_Comm_rank لمعرفة رتبة العملية في المجتمع، مثلا لمعرفة رتبة العملية في المجتمع MPI_COMM_WORLD نكتب الأمر التالي:

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

حيث سيكون رقم الرتبة في المتغير rank.

كذلك يمكن استدعاء الروتين MPI_Comm_Size لمعرفة عدد العمليات في المجتمع (communicator). مثلاً لمعرفة عدد العمليات الموجودة في المجتمع MPI_COMM_WORLD سنكتب الأمر:

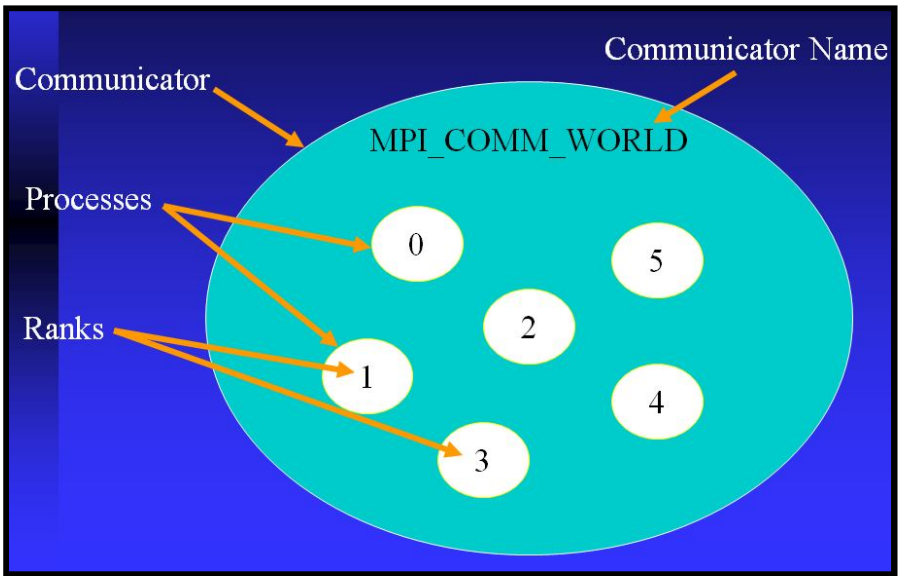
```
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

حيث سيكون عدد العمليات في المتغير size.

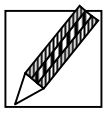
الأمر :

```
printf("I am %d of %d\n", rank, size);
```

يقوم بطباعة رتبة كل عملية، وعدد العمليات الموجودة في المجتمع MPI_COMM_WORLD.

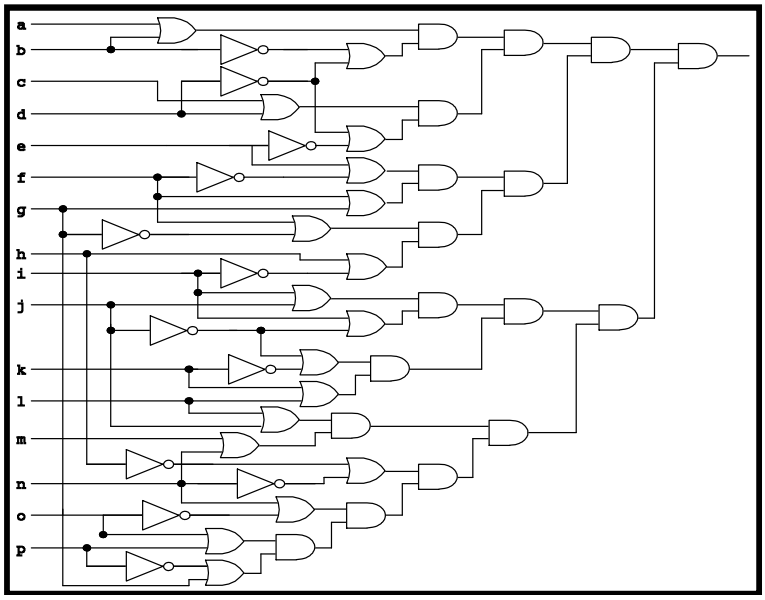


تدريب (1)



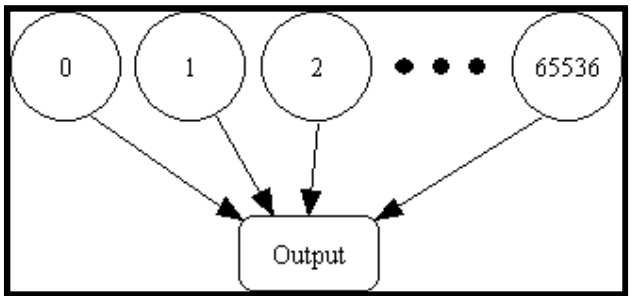
وضّح بمثال كيفية الإعلان، واستخدام الدوال init و finalize في لغة C و لغة C++.

3.2 البرنامج الثالث (Circuit Satisfiability)



في هذا البرنامج نريد أن نعرف هل الشكل أعلاه دائرة satisfiable أم لا؟ ونطلق على الدائرة بأنها satisfiable إذا كان مدخلاتها تنتج القيمة 1.

لحل هذه المسألة سنجرب كل الاحتمالات المدخلة، وبما أن مدخلات الدائرة هي (16) مدخل (من a إلى p كما في الشكل أعلاه)، وكل مدخل يمكن أن يكون صفراً أو واحداً، فإن هنالك 65,536 (2 مرفوعة للقوة 16) مدخل يمكن اختباره. يمكننا اختبار كل المدخلات في مهمة، وتنفيذ هذه المهام بالتوازي (الشكل أدناه). كل مهمة تجد أن نتيجة المدخلات التي اختبرتها هي (1) تقوم بطبع هذه المدخلات.



هذه المهام لا تحتاج إلى اتصال فيما بينها. وبما أن المهام كثيرة، فيجب توزيعها على المعالجات المتاحة بطريقة Cyclic (interleaved) Allocation، مثلا إذا كان لدينا (20) مهمة (6) عمليات، فيمكن للعملية الأولى (رقم صفر) أن تكون مسئولة عن المهام 0،6،12،18، والعملية الثانية تكون مسئولة عن المهام 1،7،13،19، والعملية الثالثة مسئولة عن 2،8،14، والعملية الرابعة عن 3،9،15، والعملية الخامسة عن 4،10،16، السادسة عن 5،11،17.

أيضا إذا كان لدينا (5) عمليات و (12) مهمة، فإن التوزيع سيكون كالتالي:

- P_0: 0, 5, 10
- P_1: 1, 6, 11
- P_2: 2, 7
- P_3: 3, 8
- P_4: 4, 9

الشفرة:

```
#include <mpi.h>
#include <stdio.h>

int main (int argc, char *argv[]) {
  int i;
  int id;
  int p;
  void check_circuit (int, int);

  MPI_Init (&argc, &argv);
  MPI_Comm_rank
(MPI_COMM_WORLD, &id);
  MPI_Comm_size
```

```
(MPI_COMM_WORLD, &p);

  for (i = id; i < 65536; i += p)
    check_circuit (id, i);

  printf ("Process %d is done\n", id);
  fflush (stdout);
  MPI_Finalize();
  return 0;
                                                    }
```

الدالة **check_circuit** المستخدمة في البرنامج أعلاه:

```
/* Return 1 if 'i'th bit of 'n' is 1; 0 otherwise */
#define EXTRACT_BIT(n,i) ((n&(1<<i))?1:0)
void check_circuit (int id, int z) {
  int v[16];       /* Each element is a bit of z */
  int i;
  for (i = 0; i < 16; i++) v[i] = EXTRACT_BIT(z,i);
  if ((v[0] || v[1]) && (!v[1] || !v[3]) && (v[2] || v[3])
    && (!v[3] || !v[4]) && (v[4] || !v[5])
    && (v[5] || !v[6]) && (v[5] || v[6])
    && (v[6] || !v[15]) && (v[7] || !v[8])
    && (!v[7] || !v[13]) && (v[8] || v[9])
    && (v[8] || !v[9]) && (!v[9] || !v[10])
    && (v[9] || v[11]) && (v[10] || v[11])
    && (v[12] || v[13]) && (v[13] || !v[14])
    && (v[14] || v[15])) {
    printf ("%d)
%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d\n",
id,
       v[0],v[1],v[2],v[3],v[4],v[5],v[6],v[7],v[8],v[9],
       v[10],v[11],v[12],v[13],v[14],v[15]);
    fflush (stdout);
                                                    }
                                                   }
```

- **تنفيذ البرنامج في معالج واحد**

```
% mpirun -np 1 sat
0) 1010111110011001
0) 0110111110011001
0) 1110111110011001
0) 1010111111011001
0) 0110111111011001
0) 1110111111011001
0) 1010111110111001
0) 0110111110111001
0) 1110111110111001
Process 0 is done
```

- **تنفيذ البرنامج في ثلاث معالجات**

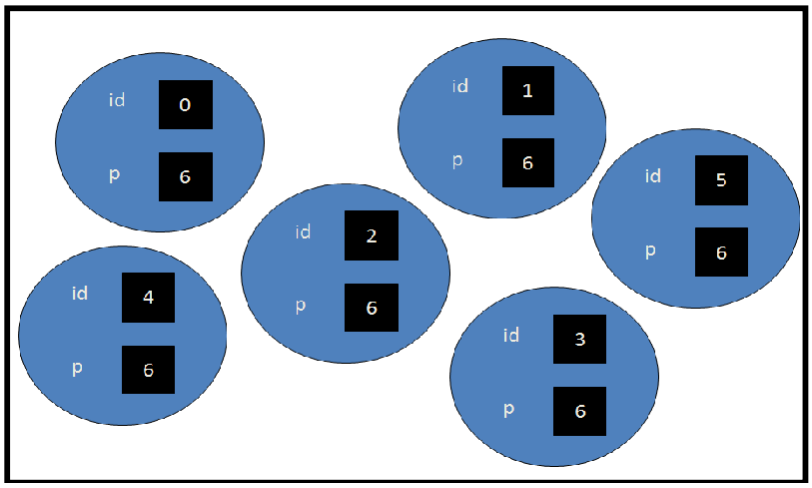
```
% mpirun -np 3 sat
0) 0110111110011001
0) 1110111111011001
2) 1010111110011001
1) 1110111110011001
1) 1010111111011001
1) 0110111110111001
0) 1010111110111001
2) 0110111111011001
2) 1110111110111001
Process 1 is done
Process 2 is done
Process 0 is done
```

- **شرح البرنامج**

```
int id; /* Process rank */
int p;  /* Number of processes
```

هذه المتغيرات يكون منها نسخة في كل عملية، الشكل أدناه بعد تنفيذ الأمرين:

```
MPI_Comm_rank (MPI_COMM_WORLD, &id);
  MPI_Comm_size (MPI_COMM_WORLD, &p);
```



توزيع المهام على العمليات (Cyclic Allocation of Work) بالأمر التالي:

```
for (i = id; i < 65536; i += p)
check_circuit (id, i);
```

في حالة تنفيذ البرنامج على معالج واحد بالأمر mpirun -np 1 sat % ستكون لدينا عملية واحدة (قيمة p هي 1) ورقم رتبة هذه العملية هو صفر (id=0). وبالتالي فستسند كل المهام لعملية واحدة.

أما عندما نحدد عدد العمليات بثلاث عمليات (mpirun -np 3 sat %) فستكون p=3، وبالتالي فستسند المهام 0، 3، 6،... للعلمية الأولى، وستسند المهام 1،4،7،... للعملية الثانية، بينما يكون نصيب العملية الثالثة المهام 2،5، 8،... وهذا ما يقوم به التكرار:

for (i = id; i < 65536; i += p).

تطوير البرنامج السابق

في البرنامج السابق إذا أردنا أن نضيف متغيراً يحسب عدد المرات التي تكون الدائرة satisifiable (عدد الحلول)، فلابد من التعديل في البرنامج لنضع هذا المتغير، وربطه مع كل العمليات، ليجمع منها الحلول، وإضافتها في هذا المتغير.

سنضيف متغيراً solutions لنحفظ فيه عدد حلول كل عملية، وسنضيف متغيراً آخر global_solutions لنضع فيه مجموع كل الحلول، وسنعلن عنهما كما يلي:

int solutions;
int global_solutions;

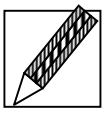
أيضا لابد من تعديل الدالة check_circut لتعطينا القيمة 1 إذا كانت نتيجة المدخلات (satisfy) وصفراً إذا كانت النتيجة غير ذلك (not satisfy)، لذلك ستكون الدالة:

check_circut(int, int)

سيكون البرنامج بعد التعديل كما يلي:

```
#include <mpi.h>
#include <stdio.h>

int main (int argc, char *argv[]) {
  int i;
  int id;
  int p;
 int solutions, global_solutions;
  int check_circuit (int, int);

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &id);
  MPI_Comm_size (MPI_COMM_WORLD, &p);

solutions = 0;
  for (i = id; i < 65536; i += p)
    solutions += check_circuit (id, i);

MPI_Reduce (&solutions, &global_solutions, 1, MPI_INT,
MPI_SUM, 0, MPI_COMM_WORLD);
  printf ("Process %d is done\n", id);
  fflush (stdout);
  MPI_Finalize();
if (id == 0) printf("There are %d different solutions \n",
global_solutions);
  return 0;
}
```

تدريب (2)



اكتب برنامج يوضح كيف يتم استخدام معالجة الأخطاء و الاستثناءات في الmpi.

3. الدالة MPI_Reduce

```
int MPI_Reduce (
  void        *operand,        /* addr of 1st reduction element */
  void        *result,         /* addr of 1st reduction result */
  int         count,           /* reductions to perform */
  MPI_Datatype type,           /* type of elements */
  MPI_Op      operator,        /* reduction operator */
  int         root,            /* process getting result(s) */
  MPI_Comm    comm             /* communicator */
)
```

- **المدخل الأول (operand): مكان العنصر الأول للاختزال (reduction)**
- **المدخل الثاني (result): مكان العنصر الثاني للاختزال (reduction)**
- **المدخل الثالث للدالة (count): عدد الاختزالات (reductions) التي نريد تنفيذها.**
- **المدخل الرابع (type): نوع المتغيرات التي نريد اختزالها (type of elements to be reduced).**
- **المدخل الخامس (operator): نوع الاختزال الذي نريد تنفيذه (هنالك جدول أدناه يسرد العمليات المبنية في OpenMP).**

- **المدخل السادس (root): نضع هنا رقم رتبة العملية التي سيكون بها كل نتائج الاختزالات (reductions).**
- **المدخل السابع (result): يشير لمكان أول نتيجة اختزال (frist reduction result). وهو ذي معنى للعملية الجذر (root process).**
- **المدخل الأخير (comm): اسم المجتمع (communicator) الذي يحتوي على العمليات المشاركة في الاختزال.**

أنواع البيانات في MPI

MPI_CHAR
MPI_DOUBLE
MPI_FLOAT
MPI_INT
MPI_LONG
MPI_LONG_DOUBLE
MPI_SHORT
MPI_UNSIGNED_CHAR
MPI_UNSIGNED
MPI_UNSIGNED_LONG
MPI_UNSIGNED_SHORT

أنواع عمليات الاختزال المبنية في MPI

MPI's built-in reduction operators
MPI_BAND
MPI_BOR
MPI_BXOR
MPI_LAND
MPI_LOR
MPI_LXOR
MPI_MAX
MPI_MAXLOC

MPI_MIN
MPI_MINLOC
MPI_PROD
MPI_SUM

قياس الأداء (benchmarking performance)

أحد طرق قياس أداء البرنامج هو حساب الزمن من بداية البرنامج حتى نهايته. لتقليل زمن التنفيذ يمكننا تجاهل الزمن المستغرق في تهيئة العمليات، وفتح الاتصالات بين العمليات وعمليات الدخل والخرج على الأجهزة المتسلسلة، وسنحسب الزمن المستغرق في التنفيذ الحقيقي (قلب البرنامج) بين قراءة البيانات وطباعة النتائج.

هنالك دالتين لقياس الزمن في MPI هما:

MPI_Wtick — timer resolution
MPI_Wtime — current time

لقياس الزمن المستغرق في أي جزء من الشفرة نضع قبل وبعد هذا الجزء الأمر MPI_Wtime التي تحسب الزمن بالثواني. نظريا نفترض أن كل العمليات تبدأ التنفيذ في وقت واحد، لكن عملياً ليس هذا ما يحدث. مثلا في البرنامج السابق، لأننا نستخدم MPI_Reduce، فإن كل العمليات (حتى التي بدأت مبكرة) لابد أن تنتظر بعضها البعض.

الدالة MPI_Barrier

لحل هذه المشكلة سنستخدم حاجز تزامن (barrier synchronization) قبل استدعاء MPI_Wtime الأولى. مع العلم بأن أي عملية لا يمكنها تخطي هذا الحاجز ما لم تصل بقية العمليات إليه. وبالتالي فإن الحاجز يؤكد من أنَّ كل العمليات ستكون في جزء الشفرة الذي نستخدمه للقياس في الوقت نفسه تقريبا.

الصيغة العامة لدالة الحاجز؛ هي:

MPI_Barrier(MPI_Comm comm)

يمكننا عمل قياس أداء في البرنامج السابق بتعديل شفرته لتكون كما يلي:

```
double elapsed_time;
…
MPI_Init (&argc, &argv);
MPI_Barrier (MPI_COMM_WORLD);
elapsed_time = - MPI_Wtime();
…
MPI_Reduce (&solutions, &global_solutions, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);
Elapsed_time += MPI_Wtime();
```

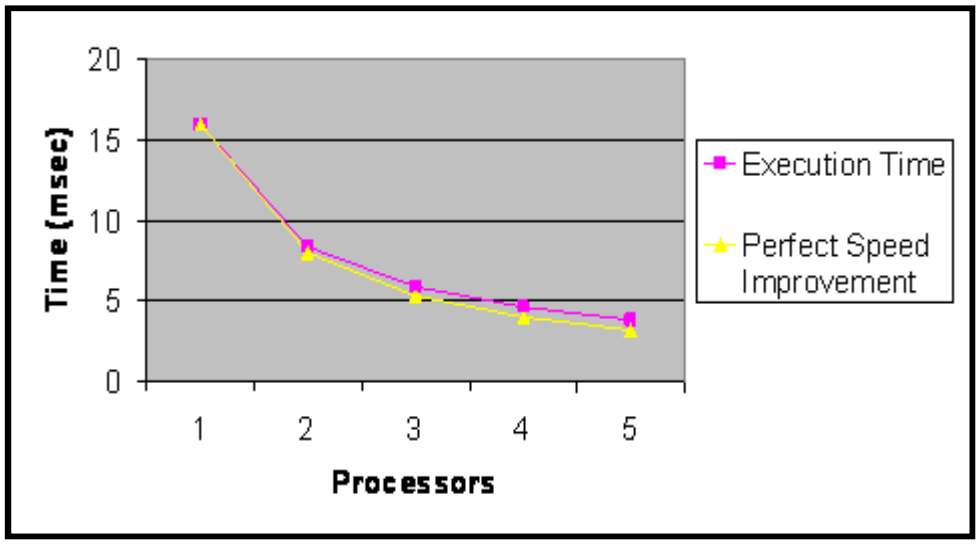
نستدعي الحاجز بعد تهيئة MPI،بالأمر MPI_Barrier

(MPI_COMM_WORLD);، ثم نستدعي دالة حساب الوقت بالأمر

elapsed_time = - MPI_Wtime();، بعد استدعاء MPI_Reduce سنوقف

المؤقت. بعد التنفيذ قد تحصل على نتيجة مثل هذه:

Processors	Time (sec)
1	15.93
2	8.38
3	5.86
4	4.60
5	3.77



الدالة MPI_Bcast

يمكن عملية من إذاعة (broadcast) عنصر بيانات أو أكثر (من النوع نفسه) لبقية العملية الموجودة معها في المجتمع، مثال:

```
MPI_Comm comm;
int array[100];
int root=0;
...
MPI_Bcast( array, 100, MPI_INT, root,
comm);
```

حيث تمت إذاعة (100) عدد صحيح من العملية الجذر(root) لبقية العمليات في المجتمع (comm).

أسئلة تقويم ذاتي



1. عدِّد انواع البيانات في MPI.
2. اذكر انواع عمليات الاختزال المبنية في MPI.
3. اذكر دوالاً لقياس الزمن في MPI.
4. اكتب الصيغة العامة لدالة الحاجز.

4. تبادل الرسائل في MPI

1.4 الاتصال المحجوز (blocking communication)

عزيزي الدارس، إرسال واستقبال الرسائل بين العمليات هي الطريقة الأساسية المستخدمة في MPI لتبادل البيانات. سنوضح كيف يتم تبادل الرسائل في البرنامج التالي، ثم نشرح دالتي الإرسال والاستقبال المستخدمين في البرنامج.

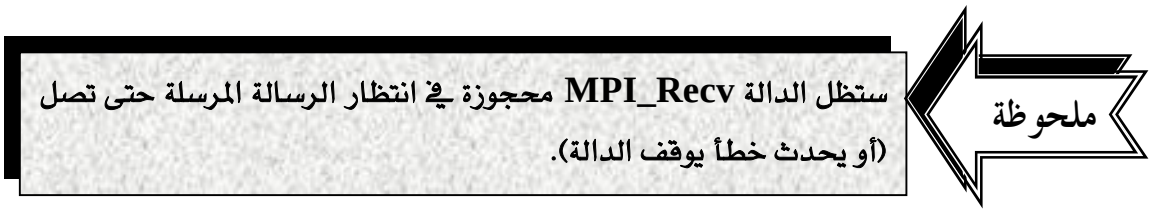
```
#include "mpi.h"
main( argc, argv )
int argc;
char **argv;
{
    char message[20];
    int myrank;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank
);
    if (myrank == 0)    /* code for process
zero */
```

```
    {
        strcpy(message,"Hello, there");
        MPI_Send(message, strlen(message),
MPI_CHAR, 1, 99, MPI_COMM_WORLD);
    }
    else                /* code for process
one */
    {
        MPI_Recv(message, 20, MPI_CHAR, 0,
99, MPI_COMM_WORLD, &status);
        printf("received :%s:\n", message);
    }
    MPI_Finalize();
}
```

في البرنامج أعلاه، ترسل العملية رقم صفر (myrank = 0) رسالة للعملية رقم (1) وذلك باستخدام الدالة MPI_SEND. حيث يتم مكان وحجم ونوع الخازن (buffer) بوساطة المدخلات الثلاثة الأولى للدالة MPI_Send، وهي (مكان الرسالة message، حجم الرسالةstrlen(message) ، نوع الرسالة MPI_CHAR). المدخل الرابع نحدد فيه وجهة الرسالة (رتبة العملية المرسل لها الرسالة) وهي 1. المدخل الخامس عنوان تعريفي للرسالة، وهو عدد صحيح (هنا رقم الرسالة هو 99). المدخل السادس يحدد نوع المجتمع الذي سترسل له الرسالة (هنا MPI_COMM_WORLD).

تستلم العملية رقم 1 (myrank = 1) الرسالة بواسطة الأمر MPI_Recv. المدخلات الثلاثة الأولى تحدد مكان وطول ونوع الرسالة المستلمة (مكان تخزين الرسالة message، طول الرسالة 20 ، نوع الرسالة MPI_CHAR). المدخل الرابع عنوان الرسالة المستلمة وهو 99 (لابد من أن يكون مطابقاً للعنوان المذكور في دالة الإرسال). المدخل الخامس المجتمع (الذي في دالة الإرسال نفسه). المدخل

السادس يعطينا معلومات عن الرسالة المستلمة، حيث تعتبر status مؤشراً لسجل يحتوي على تفاصيل معلومات، مثل رتبة العملية التي أرسلت الرسالة (Status->MPI_Source).



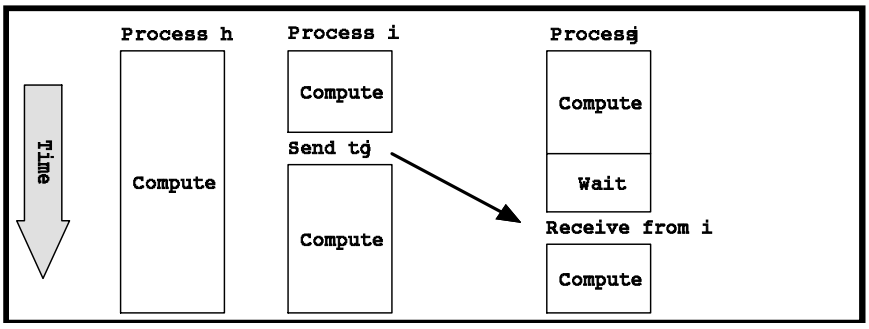
2.4 الاختناق (deadlock)

تعتبر العملية في حالة اختناق إذا تنتظرحدثاً لن يتم أبداً. لذلك من السهل الوقوع في حالة اختناق عند كتابة برنامج يستخدم MPI_Send و MPI_Recv.

مثلا إذا كان لدينا عمليتان تريدان حساب متوسط العددين a و b، وتمتلك العملية الأولى أحدث قيمة للمتغير a، بينما تمتلك العملية الثانية أحدث قيمة للمتغير B، وتحتاج كل عملية قراءة قيمة المتغير الآخر من العملية الأخرى (عمليتان يستلمان قبل الإرسال (Two processes: both receive before send)).

أيضا من مسببات الاحتناق ألاّ يتطابق رقم تعريف الرسالة (tag) في دالة الإرسال، مع رقم تعريف الرسالة في دالة الاستقبال.

أيضا قد يحدث اختناق إذا أرسلت عملية رسالة إلى عملية خطأ، أو عندما تحاول عملية استلام رسالة من عملية خطأ.



الصيغة العامة للدالة MPI_Send

```
int MPI_Send (
      void          *message,
      int            count,
      MPI_Datatype   datatype,
      int            dest,
      int            tag,
      MPI_Comm       comm
)
```

MPI_Recvالصيغة العامة للدالة

```
int MPI_Recv (
      void          *message,
      int            count,
      MPI_Datatype   datatype,
      int            source,
      int            tag,
      MPI_Comm       comm,
      MPI_Status    *status
                                                    )
```

3.4 الاتصال غير المتزامن (غير المحجوز)

Asynchronous (non-blocking) Communication

هنا لن تنتظر عمليتا الإرسال والاستقبال الإرسال إلى أن يكتمل الاتصال، ويمكن استخدام دالة الانتظار wait لمعرفة متى اكتمل العمل.

الصيغة العامة للإرسال غير المحجوز؛ هي:

- **MPI_Request request**
- **MPI_Isend(&buffer, count, datatype, dest,tag, COMM, &request)**

الصيغة العامة للاستقبال غير المحجوز هي:

- **MPI_Request request;**
- **int MPI_Irecv(&buf, count, datatype, source, tag, COMM, request)**

تستخدم الدالة MPI_Wait لإكمال الاتصال، وصيغتها العامة كالتالي:

- **MPI_Request request;**
- **MPI_Status status;**
- **MPI_Wait(&request, &status)**

حيث تظل MPI_Wait محجوزة حتى تكتمل الرسالة المحددة بـــ request.

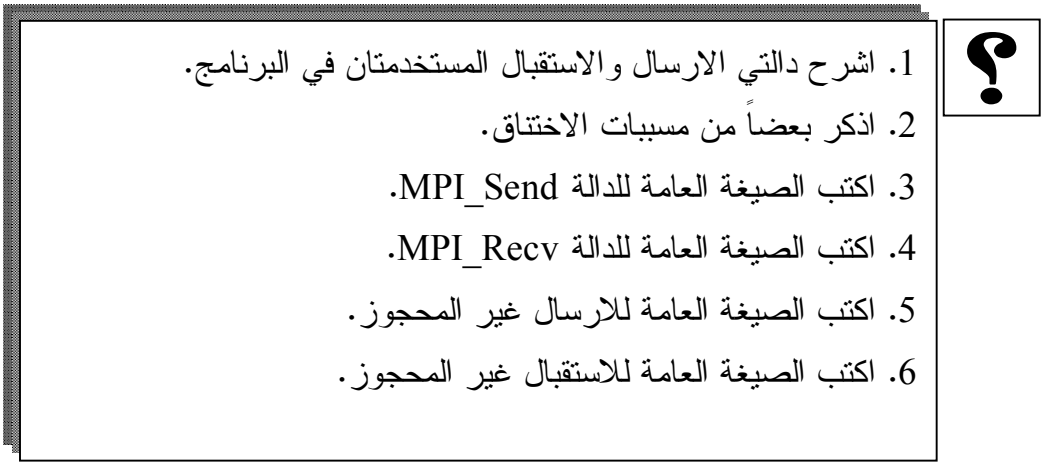
مثال برمجي يوضح استخدام الاتصال غير المحجوز

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
   int myid, numprocs, left, right;
   int buffer[10], buffer2[10];
   MPI_Request request, request2;
   MPI_Status status;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);
   right = (myid + 1) % numprocs;
   left = myid - 1;
   if (left < 0)
      left = numprocs - 1;
   MPI_Irecv(buffer, 10, MPI_INT, left, 123,
MPI_COMM_WORLD, &request);
   MPI_Isend(buffer2, 10, MPI_INT, right, 123,
MPI_COMM_WORLD, &request2);
   MPI_Wait(&request, &status);
   MPI_Wait(&request2, &status);
   MPI_Finalize();
   return 0;
}
```

بهذا اكتملت الوحدة، ولكن لم تكتمل قصة MPI، ذلك لأن الدوال الموجودة فيها كثيرة جداً ولكن كما ذكرنا فإن هنالك (6) دوال رئيسية شرحناها بالتفصيل في بداية الوحدة، وتقوم بالأعمال الأساسية في مجال البرمجة الموزعة، ثم شرحنا دوالّ أخرى إضافية رأينا أنك قد تحتاجها، وبهذه الطريقة تستطيع التعامل مع أي دالة من

دوالهه MPI حتى ولو لم تدرسها، وذلك من خلال معرفتك للصيغة العامة لها، وما الغرض منها. الغرض من هذه الوحدة أن يدلك على الطريق ولكن لن يوصلك.

أسئلة تقويم ذاتي



1. اشرح دالتي الارسال والاستقبال المستخدمتان في البرنامج.
2. اذكر بعضاً من مسببات الاختناق.
3. اكتب الصيغة العامة للدالة MPI_Send.
4. اكتب الصيغة العامة للدالة MPI_Recv.
5. اكتب الصيغة العامة للارسال غير المحجوز.
6. اكتب الصيغة العامة للاستقبال غير المحجوز.

الخلاصة

- بينت الوحدة أن التجمع عادة يتكون من حاسبات متجانسة تثبت فيها نظم تشغيل متشابهاً.

- وضحت الوحدة أن روتين تهيئة MPI_Init هو أو روتين يجب استدعاؤه في برامج MPI، ويستدعى مرة واحدة فقط والصيغة العامة لهذا الروتين؛ هي :

MPI _ Init (&argc, &argv);

- كذلك عرضت الوحدة روتين الانتهاء MPI _ Finalize هو آخر روتين يتم استدعاؤه وهو يقوم بتحرير الموارد التي تم حجزها بوساطة البرنامج، مثل الذاكرة، صيغة هذا الروتين؛ هي :

MPI _ Finalize1);

- عدّدت الوحدة أنواع البيانات في MPI

MPI_CHAR
MPI_INT
MPI_LONG
MPI_LONG_DOUBLE
MPI_SHORT
MPI_UNSIGNED_CHAR
MPI_UNSIGNED
MPI_UNSIGNED_LONG
MPI_UNSIGNED_SHORT

- كذلك عدّدت الوحدة أنواع عمليات الاختزال المبنية في MPI

MPI's built-in reduction operators
MPI_BAND
MPI_BOR
MPI_BXOR
MPI_LAND
MPI_LOR
MPI_LXOR

MPI_MAX
MPI_MAXLOC
MPI_MIN
MPI_MINLOC
MPI_PROD
MPI_SUM

- قدّمت الوحدة دالتين لقيـــاس الـــزمن فـــي MPI: MDI _ Wtick _ timer resolutr

MDI _ Wtime _ Current time

- كذلك عرضت الوحدة الصيغة العامة لدالة الحاجز؛ وهي :

MPI _ Berrter (MPI _ Comm comm)

- عرضت الوحدة الصيغة العامة للدالة

MPI_Barrier(MPI_Comm comm)

لمحة مسبقة عن الوحدة التالية

الوحدة التالية تتناول الـــ Grid Computing ومكوناتها، وأمثلة لبعض طبقات Grid
كمثال لواجهات تطبيقات مستقلة عن طبقة Grid الوسيطة.

إجابات التدريبات

تدريب (1)

• MPI_Init function:

•C: `int MPI_Init(int* argc, char*** argv)`

•C++: `void MPI::Init(int& argc, char**& argv)`

```
     void MPI::Init()
```

• MPI_Finalize function

•C:`int MPI_Finalize()`

•C++:`void MPI::Finalize()`

تدريب (2)

```
01 #include "mpi.h"
02 #include <iostream>
03 04 int main(int argc, char* argv[])
05 {
06 MPI::Init(argc, argv);
07MPI::COMM_WORLD.Set_errhandler(MPI::ERRORS_T
HROW_EXCEPTIONS);
08
09 try {
10 int rank = MPI::COMM_WORLD.Get_rank();
11 std::cout << "I am " << rank << std::endl;
12 }
13 catch (MPI::Exception e) {
14 std::cout << "MPI ERROR: " <<
e.Get_error_code() \
             << " -" << e.Get_error_string() \
```

```
            << std::endl;
16 }
17
18 MPI::Finalize();
19 return 0;
20 }
```

مسرد المصطلحات

❖ **MPI**

هي المكتبة الأكثر انتشاراً المستخدمة في تبادل الرسائل، والتي تعمل مع العديد من اللغات مثل C و Fortran و Visual C++. .Net. ويمكن استخدامها في برمجة التجمعات (clusters) والشبكات المحلية، وغيرها من الحاسبات المتعددة.

❖ **الاختناق (*deadlock*)**

تعتبر العملية في حالة اختناق إذا كانت تنتظر حدثاً لن يتم أبدا.

المراجع

Parallel Programming in C with MPI and OpenMP, Michael J. Quinn (2003).

Tutorial on Basic MPI Point-to-Point Communication in C, http://www.pdc.kth.se/training/Tutor/MPI/Pt2pt.C/index-frame.html

MPI tutorials, http://supercomputingblog.com/mpi-tutorials/


الوحدة التاسعة
الحوسبة الشبكية
Grid Computing

محتويات الوحدة

الموضوع	الصفحة
المقدمة	260
تمهيد	260
أهداف الوحدة	261
توطئة	262
1. المفاهيم الأساسية في الـــ Grid	262
1.1 تعريف الـــ Grid	263
2.1 مكونات الـــ Grid	263
1.2.1 واجهة المستخدم	264
2.2.1 الحماية	264
3.2.1 الوسيط	264
4.2.1 المجدول	265
5.2.1 إدارة البيانات	265
6.2.1 إدارة المهام والموارد	265
7.2.1 مكونات أخرى	266
3.2 الفرق بين التجمع والـــ Grid	268
2. طبقة Globus Toolkit	268
1.2 نبذة عن Globus Toolkit	269
2.2 اتحاد Globus Toolkit	269
3.2 أمثلة لاستخدام Globus Toolkit	269
4.2 خدمات Globus Toolkit	270

5.2 برمجيات Globus Toolkit	271
الخلاصة	278
لمحة مسبقة عن الوحدة التالية	280
مسرد المصطلحات	281
المراجع	282

المقدمة

تمهيد

عزيزي الدارس، مرحباً بك إلى الوحدة التاسعة في مقرر أساليب البرمجة (3)، وهي بعنوان الحوسبة الشبكية (Grid Computing)، الوحدة تتناول الـ Grid Computing ومكوناتها وأمثلة لبعض طبقات Grid الوسيطة، مثل Globus. تم تقسيم هذه الوحدة إلى قسمين القسم الأول منها يبحث في المفاهيم الأساسية للـ Grid حيث يبدأ هذا القسم بتعريف الـ Grid ومكوناته، وكذلك يتناول القسم الفرق بين التجمع (cluster) والـ Grid، القسم الثاني يتناول طبقة Globus Toolkit، وهي من أشهر الطبقات الوسيطة حيث يبدأ هذا القسم بنبذة عن Globus Toolkit ويستعرض القسم أمثلة استخدام وخدمات وبرمجيات Globus Toolkit.

أهداف الوحدة



عزيزي الدارس، بعد فراغك من دراسة هذه الوحدة ينبغي أن تكون قادراً على أن:

- تعرف الـــ Grid.
- تحدد الهدف الرئيس من بنية Grid التحتية.
- تعدد مكونات الطبقة الوسيطة(Grid middleware)بصورة عامة
- تبين الفرق بين التجمع (cluster) والـــ Grid.
- تكتب نبذة عن Globus Toolkit.
- تعدد المهام التي تقوم بها Globus Toolkit .
- تذكر بعضاً من تقنيات Globus Toolkit 4.0 (GT4.0).

1. المفاهيم الأساسية في الـــ Grid

1.1 تعريف Grid

عزيزي الدارس، مجموعة من ملايين الحاسبات المنتشرة في بقاع الأرض والتي تمتلك بآلاف الأشخاص المختلفين، والتي تحتوي على أجهزة متباينة في العتاد وفي البرامج، فمنها حاسبات مكتبة، حاسبات محمولة (laptops)، وحاسبات فائقة السرعة (supercomputers)، وخازنات بيانات، وأجهزة أخرى، مثل الهواتف النقالة وحساسات الإرصاد الجوي (meteorological sensors)، والتلسكوبات (telescopes)، وغيرها من الأجهزة.

كل هذه الأجهزة ترتبط مع بعضها البعض بشبكات مختلفة محلية وواسعة، سلكية ولاسلكية، عبر الإنترنت، والإنترانت، والاكسترانت.

تتاح لك كل هذه الأجهزة أينما كنت ووقتما أردت، وتستخدمها كأنها حاسب واحد عملاق سريع وقوي، ومتوفرة فيه الكثير من الأجهزة النادرة التي قد لا تستطيع شراؤها أو حتى الحلم برؤيتها.

تسمية وفكرة Grid أخذت من شبكة الكهرباء (electrical gird). الفكرة هي جعل الوصول والاستفادة من الـــ Grid بالبساطة والسهولة التي نستخدم بها شبكة الكهرباء.

OGSA و OGSI

OGSA تعرف المعايير القياسية للتركيبة (structure)، والخدمات التي توفرها بيئة Grid عامة. مواصفات Open Grid Services Interface (OGSI) هي مواصفات قياسية مصاحبة تعرف الواجهات والبروتوكولات التي تستخدم بين الخدمات المتوفرة في بيئة Grid. OGSI هي قياسات تمكن بيئات

Grid المختلفة والملتزمة بهذه المعايير من أن تتواصل مع بعضها (inter-operability).

2.1 مكونات Grid

الهدف الرئيسي من بنية Grid التحتية (طبقة Grid الوسيطة) هو توفير بيئة سلسلة للمستخدمين تمكنهم من الوصول إلى استخدام الموارد الموزعة كما لو كانت معهم في غرفتهم أو مكتبهم.

مكونات الطبقة الوسيطة (Grid middleware) بصورة عامة؛ هي :

أ- واجهة المستخدم (Portal – User Interface).

ب- التأمين (security).

ت- الوسيط (Broker).

ث- المجدول (scheduler).

ج- إدارة البيانات (data management).

ح- مدير الموارد والأعمال (job & resource management).

خ- أخرى.

هذه المكونات ليس بالضرورة أن تكون في كل Grid، وإنما يعتمد ذلك على تصميم ال Grid (Grid design) والهدف الذي من أجله صمم. فيما يلي سنلقي نظرة على كل مكون من المكونات أعلاه.

1.2.1 واجهة المستخدم

عزيزي الدارس، كما ترى الطاقة الكهربية في شكل مفاتيح ومقابس، كذلك يجب ألاّ ترى تعقيدات Grid. لذلك يمكن لواجهة المستخدم للـــ Grid أن تأتي في

نماذج وأشكال متعددة، وتكون محددة بنوع التطبيق، وفي بعض الأحيان تكون في شكل بوابة (portal).

بوابة Grid (grid portal) توفر واجهة للمتسخدم لتشغيل تطبيقات تستفيد من الموارد والخدمات التي توفرها الـــ Grid. من هذا المنظور يرى المستخدم Grid كمصدر قوة حاسوبية خيالية (virtual)، كما يرى مستهلك الطاقة الكهربية المفاتيح والمقابس كواجهة لمولد خيالي كبير.

2.2.1 الحماية (security)

من متطلبات Grid الرئيسية هو الحماية. تحتوي أي بيئة Grid على طرق توفر تأمين هذه البيئة، تشمل الحماية، التحقق (authentication)، الصلاحيات (authorization)، تشفير البيانات (encryption)،...إلخ.

3.2.1 الوسيط (broker)

بعد التحقق من المستخدم يستطيع وضع تطبيقه في الـــ Grid. وقد يضيف المستخدم مدخلات (parameters) أخرى مع التطبيق.

الخطوة التالية هي التعرف على الموارد المناسبة المتاحة في الـــ Grid والتي يمكن للتطبيق استخدامها والاستفادة منها. هذه الخطوة يقوم بها الوسيط (broker).

4.2.1 المجدول (scheduler)

بعد التعرف على الموارد، يجب جدولة المهام لتنفذ في هذه الموارد. إذا كانت المهام غير معتمدة على بعضها فلن نحتاج مجدولاً خاصاً. ولكن إذا كانت المهام تحتاج حجز موارد خاصة، أو تريد أن تعمل بالتوازي (concurrently)، فلابد من استخدام مجدول المهام (job scheduler) لتنسيق التنفيذ بين المهام (jobs).

وقد يكون هنالك عدة مستويات من الجدولة، وبالتالي عدة مجدولات، فمثلا قد يكون هنالك تجمع (cluster)ضمن الـــ Grid، في هذه الحالة سيكون هنالك مجدول

للتجمع (cluster) يساعده في إدارة محطات التجمع (nodes of cluster). في المستوى الأعلى هنالك مجدول آخر (أحيانا يسمى meta scheduler)، يقوم بجدولة العمل الذي سينفذ في التجمع، بينما يقوم مجدول التجمع بجدولة هذا العمل داخل التجمع.

5.2.1 إدارة البيانات (data management)

إذا كان لدينا بيانات (بما فيها أجزاء التطبيق نفسه)، فلابد من تحويلها أو جعلها متاحة لمحطات الـ Grid التي سننفذ فيها التطبيق. وللقيام بذلك لابد من طريقة آمنة ومضمونة (reliable) لنقل الملفات والبيانات إلى محطات الـ Grid.

6.2.1 إدارة المهام والموارد (job & resource management)

قلب Grid هي مجموعة من الخدمات (core set of services) التي تساعد في أداء العمل الحقيقي داخل بيئة الـ Grid.

يوفر Grid Resource Allocation Manager (GRAM) العديد من الخدمات التي تقوم بالتالي:

- وضع المهمة (launch a job) في مورد معين.
- تختبر حالة المهمة.
- تستلم النتائج عندما تكتمل المهمة.

7.2.1 مكونات أخرى

قد نحتاج تسهيلات (facilities) أخرى تضاف إلى بيئة الـ Grid ينظر إليها عند تصميم وتنفيذ التطبيق. مثلا الاتصال بين العمليات (inter-process communication) و خدمات المحاسبة وتحمل التكاليف (accounting/chargeback). تعتبر من التسهيلات التي قد تطلب كثيرا.

3.1 الفرق بين التجمع (cluster) والـ Grid

نستخدم في التجمع (cluster) برمجة تبادل الرسائل وأدوات؛ مثل MPI، بينما نستخدم في برمجة الحاسبات متعددة المعالجات (multiprocessor)، والحاسبات متعددة النواة (multi-core) برمجة الخيوط وأدوات مثل OpenMP و Pthreads. علما بأن الكثير من الحاسبات المستخدمة في التجمعات اليوم هي حاسبات متعددة النواة مشتركة الذاكرة.

ولأن معظم حاسبات التجمع محلية، فتصبح حماية الشبكة ليست بمسألة تهم المستخدم كثيراً. وغالباً يكفي استخدام SSH للتعامل مع حاسبات التجمع، وعادة ما تكون التجمعات تجمعات لينكس، وقد يثبت عليها بعض البرامج المساعدة، مثل نظام ملفات الشبكة (Network File System (NFS)).

في الـ Grid نهتم بإدارة موارد موزعة جغرافياً. ويحتاج المستخدم حسابات على كل الموارد. وغالبا ما يمثل موقع تجمع. وكون أن البيئة موزعة فهذا يحوجنا إلى تقنيات مثل خدمات الويب (web services)، وبروتوكولات الإنترنت، وحماية الشبكة، بالإضافة إلى كيفية الاستفادة من الموارد الموزعة. الحماية مهمة جدّاً لأن المشروع الذي يستخدم الـ Grid قد يكون لديه معلومات سرية، وطبيعة البيئة الموزعة تفتح الباب للخروقات الأمنية والتخريب.

الحماية (security) مهمة جدّاً في الـ Grid، بينما تقل أهميتها في التجمع (Cluster).

هنالك أشياء متشابهة بين التجمع والـــ Grid فالأثنان يستخدمان :

- عدة موارد بشكل جماعي.
- استخدام مجدولات مهام (job schedulers) لتوزيع العمل.

ولكن يكون مجدول المهام في التجمع واحداً ويقوم بتوزيع المهام على الموارد المحلية؛ بينما يكون مجدول المهام في الـــ Grid يعمل مع مجدولات التجمع، بحيث يتداخل عملهما مع بعض.

أسئلة تقويم ذاتي



1. عرّف الـــ Grid؟
2. حدّد الهدف الرئيس من بنية Grid التحتية.
3. عدّد مكونات الطبقة الوسيطة (Grid middleware) بصورة عامة.
4. بيِّن الفرق بين التجمع (cluster) والـــ Grid؟

2. طبقة Globus Toolkit

عزيزي الدارس، هنالك العديد من الجامعات والمراكز البحثية والتجارية التي لديها طبقات Grid وسيطة، وفيما يلي بعض منها:

- **Aneka: The Next Generation Enterprise Grid Technology**
- **Alchemi**
- **Cosm P2P Toolkit**
- **Globus**
- **Gridbus**
- **Grid Datafarm**
- **GridSim: Toolkit for Grid Resource Modeling and Scheduling Simultation**
- **Jxta Peer to Peer Network**
- **Legion: A Worldwide Virtual Computer**
- **NorduGrid middleware**
- **PUNCH**
- **Simgrid**
- **Storage Resource Broker (SRB)**
- **ProActive**
- **Unicore**
- **Vishwa**

يعتبر Globus من أشهر الطبقات الوسيطة، وأكثر تأثيراً من بين الطبقات التي ذكرنا مسبقاً. وقد تم تطوير نسخة متعددة منه.

1.2 نبذة عن Globus Toolkit

هي أداة برمجية مفتوحة المصدر تستخدم في بناء نظم الـــ Grid وتطبيقاتها (Grid systems and applications). وقد تم بناء هذه الأداة بواسطة اتحاد Globus، وجهات أخرى من جميع أنحاء العالم. وهنالك الكثير من المشاريع والشركات التي تستخدم هذه الأداة البرمجية في حوسبتها الشبكية الخاصة.

أول نسخة ظهرت لبرنامج Globus Toolkit كان في 1998، ثم ظهرت النسخة الثانية في عام 2002، الآن النسخة الخامسة متاحة (GT5.0)، والتي تمكنك من بناء حوسبة شبكية بخدمات مكتوبة بلغة C. تعمل هذه الخدمات على بيئات ينكس ولينكس.

2.2 اتحاد Globus (Globus Alliance)

تجمع مكون من منظمات وأفراد يقومون بتطوير التقنيات الأساسية المستخدمة في الـــ Grid والتي تسمح للناس بالتشارك في قوة حاسوبية وقواعد بيانات وأجهزة وأدوات موزعة في مؤسسات ومعاهد (قد تكون متباعدة جغرافيا) بصورة آمنة، ودون المساس بخصوصية هذه المؤسسات والمعاهد.

3.2 أمثلة لاستخدام Globus toolkit

عزيزي الدارس، قام فريق عمل من الفيزيائيين باستخدام Globus Toolkit و MPICH-G2 للاستفادة من قوة عدد من الحواسيب فائقة السرعة (super computers) لمحاكاة تأثير الجاذبية على تصادم الثقب الأسود (simulate the gravitational effects of black hole collisions).

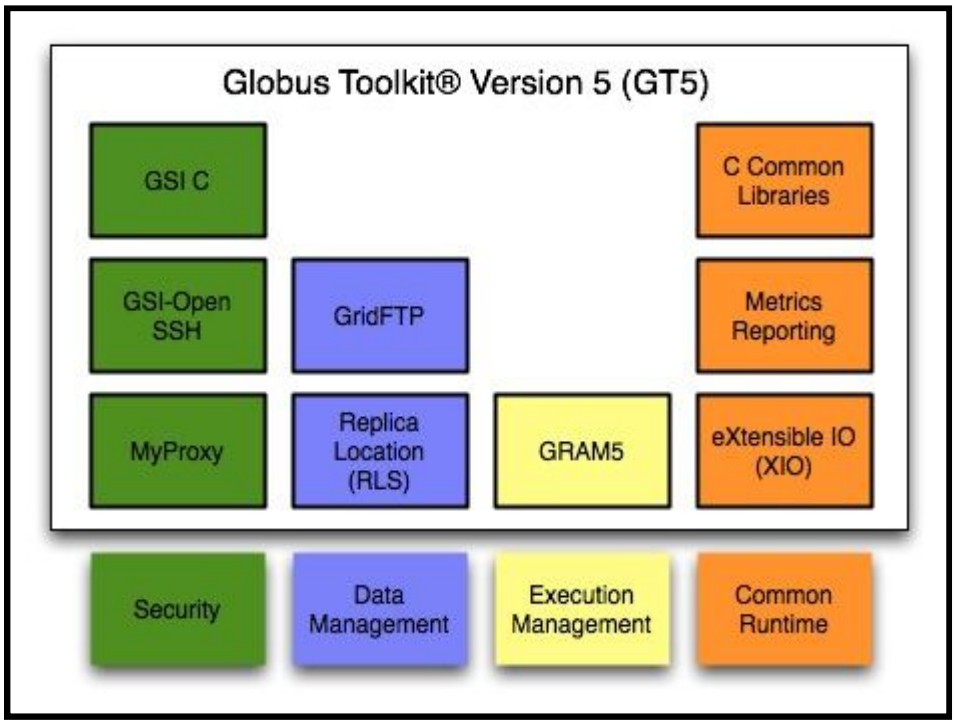
فريق العمل مكون من باحثين من معمل أريجونا الوطني، جامعة شيكاغو، جامعة إيليون، ومعهد لانك للجاذبية بألمانيا. وقد حصل هذا العمل على جائزة غوردون بيل (Gordon Bell prize).

4.2 خدمات Globus Toolkit

تقوم Globus toolkit بالعديد من المهام مثل:

- مراقبة الموارد (resource monitoring).
- اكتشاف الموارد (discovery).
- إدارة الموارد (management).
- الحماية (security).
- إدارة الملفات (file management).

إن Globus Toolkit بالإضافة لكونها جزءاً أساسياً في بناء الكثير من المشاريع الهندسية العالمية، والتي تكلف حوالي نصف بليون دولار، أيضاً تعتبر من الدعامات الأساسية التي تعتمد عليها الكثير من شركات التقنية في بناء منتجات حوسبة شبكية ضخمة (significant commercial Grid products).



5.2 برمجيات Globus Toolkit

تتكون Globus Toolkit من المكونات (البرمجيات) التي تدعم :

- الحماية (security).
- البنية التحتية للمعلومات (information infrastructure)
- إدارة الموارد (resource management).
- إدارة البيانات (data management).
- الاتصال (communication).
- اكتشاف الأخطاء (fault detection).
- التنقلية (portability).

تحزم هذه المجموعة من المكونات حيث يمكن استخدامها مستقلة، أو مع بعضها في تطوير التطبيقات. لدى كل منظمة أسلوب عملها الفريد، لذلك التعاون بين المنظمات المختلفة يعرقلها عدم التوافقية في الموارد، مثل أرشفة البيانات والحاسبات والشبكات. لذلك جاءت Globus Toolkit بتصور يزيل هذه العقبات التي تمنع التعاون السلس بين المنظمات المتباينة. فهي خدمات أساسية، وواجهات وبروتوكولات تسمح للمستخدم بالوصول إلى موارد بعيدة واستخدامها كما لو كانت في غرفته

نمت Globus Toolkit، وتطورت عبر استراتيجية المصادر المفتوحة، مثلها في ذلك مثل نظام التشغيل لينكس، بحيث يبعدها هذا عن التملك الشخصي، ويساعد في تطويرها بسرعة كما يفعل مجتمع المصادر المفتوحة دوماً.

تقنيات GT4.0

Globus Toolkit 4.0 (GT4.0) لديها العديد من التقنيات مثل:

- المكونات الأساسية للتشغيل (common Runtime Components)
- الحماية (security).
- إدارة البيانات (data management).
- خدمات المعلومات (information services).
- GRAM.

تفاصيل هذه التقنيات متوفرة على موقع Globus الرسمي (globus.org)

MPICH-G2

تعتبر MPI مناسبة لبرمجة المسائل الكبيرة، والتي يمكن تقسيمها إلى عمليات تنفذ بالتوازي، وتتبادل هذه العمليات المعلومات فيما بينها عند ما تحتاج لذلك.

البيئة البرمجية التي تتيحها MPI تقوم بالتفاصيل التي تهتم ببدء وإنهاء العمليات، تنسيق التنفيذ وتوفير وسائل للتزامن؛ مثل الحواجز (barriers) والميتافور (metaphors)، تمرير البيانات بين العمليات.

جاءت MPICH-G2 كامتداد يبنى على MPI، وتستخدم MPICH-G2 قدرات الـــ Grid مثل:

- بدء العمليات في أنظمة بعيدة (remote systems).
- إرسال البرامج والبيانات إلى الأنظمة البعيدة.
- الحماية.
- اختيار طريقة الاتصال المناسبة، اعتماداً على هل ستمرر الرسالة لأنظمة محلية أم بين أنظمة بعيدة.

كتابة تطبيقات مخصصة للـــ Grid

يمكن كتابة تطبيقات مخصصة تستخدم مزايا Grid معين، مثل:

- نقل الملفات بين المواقع حسب الحاجة.
- العثور على الموارد المناسبة لمهمة معينة.
- بدء مهام جديدة من داخل التطبيق.

هذا يعتبر صعباً جداً وخاصة إذا استخدمت واجهات التطبيقي البرمجية لطبقة Grid معين (Grid middleware APIs). حيث غالباً ما تختلف هذه الواجهات (APIs) بين نسخة وأخرى. هنالك حالياً جهود لتطوير واجهات تطبيق برامج عالية المستوى قياسية يستطيع استخدامها أي شخص، ولكن هذا العمل يحتاج الكثير.

استخدام Grid Middleware APIs

برمجة الـــ Grid هي إدراج واجهات طبقة الـــ Grid الوسيطة البرمجية في شفرة التطبيق لإنجاز العمليات الخارجية؛ مثل:

- **إدخال وإخراج الملفات (file input/output).**
- **بدء المهام ومراقبتها.**
- **مراقبة واكتشاف الموارد.**

واجهات Globus البرمجية

هنالك الكثير من واجهات APIs التي يوفرها Globus تستخدم لاستدعاء خدمات وروتينات Globus. هنالك المئات، بل الألآف من روتينات C، وجافا الموجودة في موقع Globus والتي يمكنك استدعائها داخل تطبيقاتك. مثلا في 2005 قام شخصان بكتابة واجهة تطبيق برمجية لنقل ملفات من موقع إلى موقع آخر، تتكون هذه الواجهة من حوالي (50) سطراً من الشفرة.

واجهات APIs المستقلة عن طبقة Grid

إن الاعتماد على طبقة Grid وسيطة معينة في كتابة تطبيقات يجعل هذه التطبيقات غير متنقلة، وحتي بين النسخة والأخرى نجد عدم توافقية. مثلا Globus يراجع كل عدة شهور، وتكون هنالك تغييرات كبيرة كل سنة أو سنتين. مثلا نسخة Globus الثالثة تم تركها تماما بعد عام واحد فقط من صدورها.

أيضا هنالك الكثير من الطبقات الوسيطة غير Globus، مثل UNICORE و gLite وغيرها الكثير.

مفهوم التطبيقات البرمجية APIs عالية المستوى هو جعل طبقة APIs فوق طبقة Grid الوسيطة، وليس داخلها مما يجعلها مستقلة عن الطبقة الوسيطة، ولا تتأثر

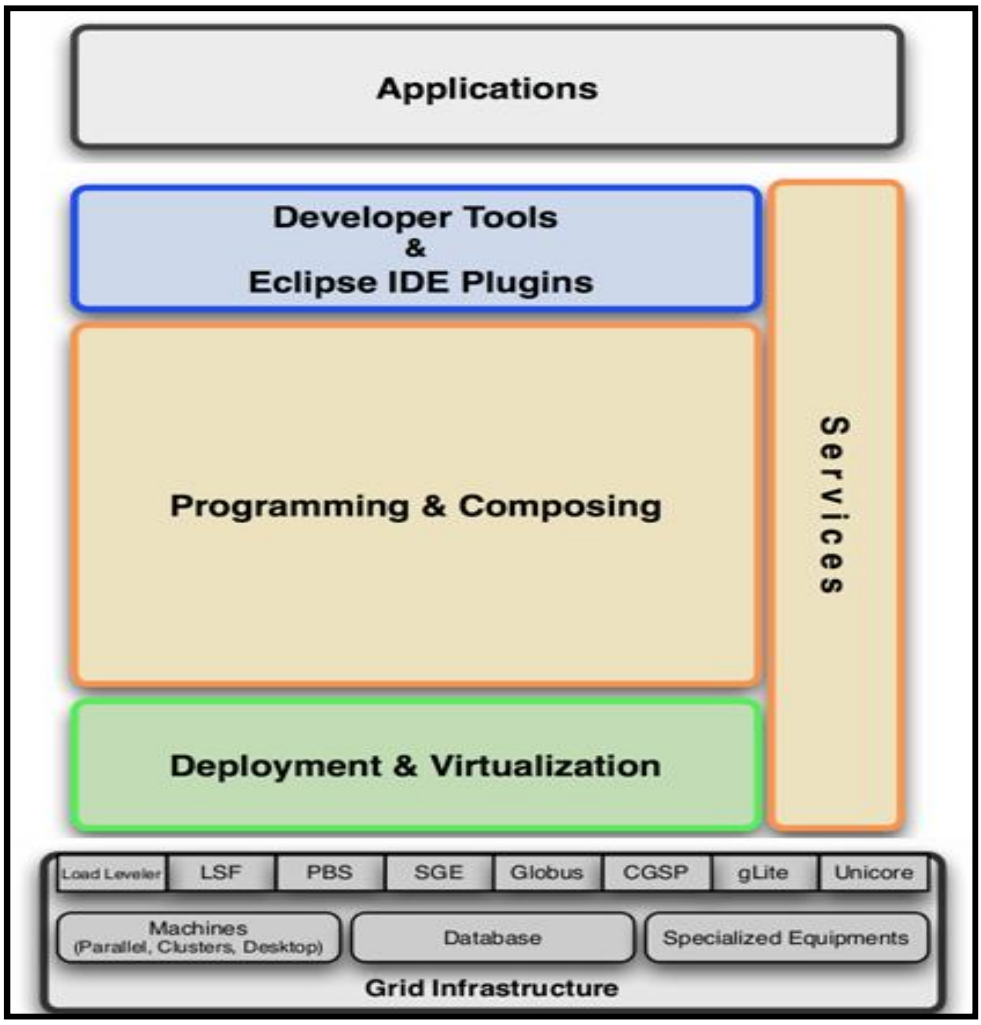
بالتغييرات التي تحدث في الطبقة الوسيطة، وبالتالي تكون مستقرة وثابتة بالنسبة لمبرمجي الـــ Grid.

Grid Application Toolkit (GAT) أحدث الأمثلة التي طبقت هذا المفهوم. طورت في 2003–2005. ثم تلتها جهود قام بها Grid community لوضع معايير قياسية لواجهات APIs، والتي قادت إلى مشروع Simple API for Grid Applications (SAGA). حيث أصبح من الممكن كتابة شفرة لنقل ملف من عدة سطور C++.

أيضا ظهرت واجهات تطوير برمجية متكاملة تحتوي على واجهات تطبيقات برمجية وبيئة تطوير مثل ProActive التي تعتبر أداة لبرمجة الـــ Grid باستخدام لغة جافا.

ProActive

تعتبر ProActive مكتبة جافا مفتوحة المصدر للحوسبة المتوازية والموزعة ومتعددة الخيوط. وتمتلك بيئة عمل تتصف بالتنقلية والحماية. توفر ProActive أداة توفر كتابة تطبيقات موزعة في الشبكات المحلية، التجمعات، الـــ Internet Grids، وشبكات الانترانت.


Applications
Developer Tools
&
Eclipse IDE Plugins
Programming & Composing
Deployment & Virtualization
Services
Load Leveler
LSF
PBS
SGE
Globus
CGSP
gLite
Unicore
Machines
(Parallel, Clusters, Desktop)
Database
Specialized Equipments
Grid Infrastructure

أسئلة تقويم ذاتي



1. اكتب نبذة عن Globus Toolkit.
2. عرّف اتحاد Globus (Globus Alliance).
3. عدّد المهام التي تقوم بها Globus Toolkit .
4. اذكر بعض من تقنيات Globus Toolkit 4.0 (GT4.0).

الخلاصة

- عرفت الوحدة الـ Grid : مجموعة من ملايين الحاسبات المنتشرة في بقاع الأرض ترتبط مع بعضها بشبكات مختلفة محلية وواسعة، سلكية، لا سلكية، عبر الإنترنت، والإنترانت، والإكسترانت.

- عرضت الوحدة مكونات الـ Grid : واجهة المستخدم – التامين – الوسيط – المجدول – إدارة البيانات – مدير الموارد والأعمال – أخرى.

- وضحت الوحدة الفرق بين التجمع cluster والـ Grid في أن التجمع clusterيستخدم برمجة تبادل الرسائل وأدوات؛ مثل MPI؛ بينما نستخدم في برمجة الحاسبات متعددة المعالجات والحاسبات متعددة النواة برمجة الخيوط وأدوات؛ مثل OpenMP و Pthreads.

أما في الـ Grid نهتم بإدارة موارد موزعة جغرافياً، ويحتاج المستخدم حسابات على كل الموارد. وغالبا ما يمثل موقع تجمع.

الحماية (security) مهمة جدا في الـ Grid، بينما تقل أهميتها في التجمع (Cluster).

- كذلك وضحت الوحدة أوجه الشبهة بين التجمع cluster والـ Grid فالاثنان يستخدمان:عدة موارد بشكل جماعي - استخدام مجدولات مهام (job schedulers) لتوزيع العمل.

عرفت الوحدة Globus Toolkit هي أداة برمجية مفتوحة المصدر تستخدم في بناء نظم الـ Grid وتطبيقاتها. وقد تم بناء هذه الأداة بواسطة اتحاد Globus وجهات أخرى من جميع أنحاء العالم.

- كذلك عرفت الوحدة اتحاد Globus بأنه تجمع مكون من منظمات وأفراد يقومون بتطوير التقنيات الأساسية المستخدمة في الـ Grid والتي تسمح للناس بالتشارك في

قوة حاسوبية وقواعد بيانات، وأجهزة وأدوات موزعة في مؤسسات ومعاهد بصـــورة آمنة، ودون المساس بخصوصية هذه المؤسسات والمعاهد.

- عدّدت الوحدة مهام Globus Toolkit بـــ: مراقبة الموارد - اكتشاف المـــوارد- إدارة الموارد - الحماية - إدارة الملفات.

تتكون Globus Toolkit من المكونات (البرمجيات) التي تدعم :الحماية - البنيـــة التحتية للمعلومات - إدارة الموارد - إدارة البيانات - الاتصال - اكتشاف الأخطاء - التنقلية.

- بينت الوحدة أنه يمكن كتابة تطبيقات مخصصة تستخدم مزايا Grid معين؛ مثـــل: نقل الملفات بين المواقع حسب الحاجة- العثور على الموارد المناسبة لمهمة معينـــة- بدء مهام جديدة من داخل التطبيق.

لمحة مسبقة عن الوحدة التالية

نتناول الوحدة التالية وهي الوحدة الأخيرة في هذا المقرر موضوع SAGA حيث تعرف SAGA وتذكر مميزاتها، وتعدِّد أشكال الطرق في نموذج المهمة، وتوضح مفهوم الحزم الوظيفية.

مسرد االمصطلحات

❖ **Grid**

مجموعة من ملايين الحاسبات المنتشرة في بقاع الأرض، والتي تمتلك بآلاف الأشخاص المختلفين، والتي تحتوي على أجهزة متباينة في العتاد وفي البرامج، كل هذه الأجهزة ترتبط مع بعضها البعض بشبكات مختلفة محلية وواسعة، سلكية ولاسلكية، عبر الإنترنت والإنترانت والإكسترانت.

تتاح لك كل هذه الأجهزة أين ما كنت ووقت ما أردت وتستخدمها كأنها حاسب واحد عملاق سريع وقوي، ومتوفرة فيه الكثير من الأجهزة النادرة التي قد لا تستطيع شراءها، أو حتى الحلم برؤيتها.

تسمية وفكرة Grid أخذت من شبكة الكهرباء (electrical gird). الفكرة هي جعل الوصول والاستفادة من الـــ Grid بالبساطة والسهولة التي نستخدم بها شبكة الكهرباء.

❖ **Globus Toolkit**

هي أداة برمجية مفتوحة المصدر تستخدم في بناء نظم الـــ Grid وتطبيقاتها (Grid systems and applications). وقد تم بناء هذه الأداة بواسطة اتحاد Globus وجهات أخرى من جميع أنحاء العالم. وهنالك الكثير من المشاريع والشركات التي تستخدم هذه الأداة البرمجية في حوسبتها الشبكية الخاصة.

❖ **اتحاد Globus (Globus Alliance)**

تجمع مكون من منظمات وأفراد يقومون بتطوير التقنيات الأساسية المستخدمة في الـــ Grid والتي تسمح للناس بالتشارك في قوة حاسوبية وقواعد بيانات، وأجهزة وأدوات موزعة في مؤسسات ومعاهد (قد تكون متباعدة جغرافيا) بصورة آمنة، ودون المساس بخصوصية هذه المؤسسات والمعاهد.

المراجع

Grid Computing Info Centre (GRID Infoware) , http://www.gridcomputing.com, (2010).

Grid Computing: A Practical Guide to Technology and Applications, Ahmar Abbas, Charles River Media, INC, , (2004).

Enabling Applications for Grid Computing with Globus, Bart Jacob, Luis Ferreira, Norbert Bieberstein, Candice Gilzean, Jean-Yves Girard, Roman Strachowski, Seong (Steve) Yu, (2003), IBM

Globus Toolkit, http://www.globus.org/toolkit, (2010).

Grid Computing Techniques and Applications, Wilkinson , Barry, Taylor and Francis Group, (2010).

ProActive, http://proactive.inria.fr, (2010).


الوحدة العاشرة
SAGA
Asimple API for Grid
Applications

محتويات الوحدة

الموضوع	الصفحة
المقدمة	285
تمهيد	285
أهداف الوحدة	286
A Simple API for Grid Applications (SAGA).1	287
1.1 ما هي SAGA	287
2.1 تطبيقات SAGA	287
3.1 عملي	288
4.1 مميزات SAGA	289
5.1 ما الذي لا تقوم به SAGA	289
2. الحزم الوظيفية (functional packages)	291
1.2 Namespace Package API	292
2.2 واجهات حزمة الملفات	293
3.2 واجهات حزمة العمل	294
4.2 واجهات حزمة النداء البعيد	296
الخلاصة	298
مسرد المصطلحات	299
المراجع	300

المقدمة

تمهيد

عزيزي الدارس، مرحباً بك إلى الوحدة التاسعة في مقرر أساليب البرمجة (3) وهي بعنوان A Simple API for Grid Applications (SAGA)، وهي تعرض لك من خلال قسمين في القسم الأول من الوحدة سنلقي نظرة عامة على (SAGA) كمثال لواجهات تطبيقات مستقلة عن طبقة Grid الوسيطة، حيث يعرِّف القسم SAGA، كما يقدّم القسم تطبيقاً عملياً لبدء استخدام SAGA بتحميل شفرتها (source)، ومميزاتها، أما القسم الثاني والأخير، فيستعرض الحزم الوظيفية، وهو يتناول العديد من حزم SAGA.

أهداف الوحدة



عزيزي الدارس، بعد فراغك من دراسة هذه الوحدة ينبغي أن تكون قادراً على أن:

- **تعرِّف SAGA**
- **تصف بالرسم الشكل العام لـــ SAGA.**
- **تذكر مميزات SAGA.**
- **تعدِّد أشكال الطرق في نموذج المهمة.**
- **توضِّح مفهوم الحزم الوظيفية (functional packages).**
- **تعدِّد حزم SAGA**

A Simple API for Grid Applications (SAGA).1

عزيزي الدارس، في هذا القسم سنلقي نظرة عامة على SAGA كمثال لواجهات تطبيقات مستلقة عن طبقة Grid الوسيطة. لا تعتبر هذه المقدمة عن SAGA كافية لكتابة برامج متقدمة، ولكنها تفتح الطريق وتعطي لمحة بسيطة عن شكل الحزم والطرق المستخدمة في SAGA، وإمكانية التوسع في ذلك عبر موقع SAGA الرسمي، والذي أثبتناه في المراجع. وقد تركنا مهمة تحميل وتثبيت وكتابة أول برنامج يعمل على SAGA على القارئ، حيث يوجد مرجع سريع بالموقع يوضح كل هذه الخطوات.

1.1 ما هي SAGA

هي واجهات تطبيق برمجية عالية المستوى توفر الوظائف الأساسية لبناء تطبيقات موزعة، وتوفر أدوات وبيئة عمل مستقلة عن البنية التحتية لطبقة الـــ Grid.

عرفت هذه الواجهات بواسطة Open Grid Forum (OGF) بغرض تطوير تطبيقات عالية المستوى لمن لا يرد معرفة أي شيء عن Grid computing، ولكن يريد استخدام موارد موزعة عبر Grid.

تعريفات Grid API سهلة وبسيطة، بحيث يمكنك استخدامها دون الحاجة لمعرفة تفاصيل التقنيات التحية، أو الطبقة الوسطى المستخدم في الـــ Grid.

2.1 تطبيقات SAGA

حالياً يوجد تطبيقان وعدة روابط مع لغات؛ هي:

تطبيق C++ الذي طور في جامعة ولاية لويزيانا (Louisiana State University CCT /)، حيث يمكن الحصول على معلومات أوفى من الموقع http://saga.cct.lsu.edu/implementaions/saga-using-c.

تطبيق جافا، وطوّر في الجامعة الحرة بهولندا (Vrije Universiteit Amsterdam)،، بالموقع

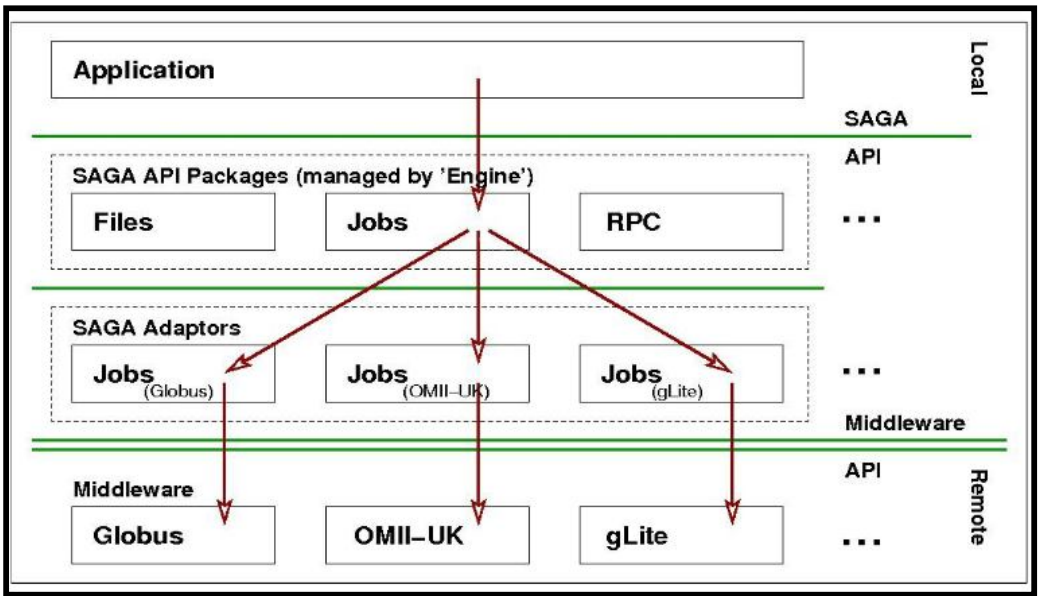
.http://saga.cct.lsu.edu/implementations/saga-using-java

3.1 عملي

لتبدأ استخدام SAGA عليك بتحميل شفرتها (source) من الموقع :
http://saga.cct.lsu.edu/software
ثم متابعة المرجع السريع (quick refernce) بالموقع، وتتبع خطوات بناء SAGA وتثبيتها، ثم كتابة أول برنامج..

الشكل العام

الشكل أدناه يوضح موقع واجهات SAGA، حيث تثبت في البداية طبقة Grid الوسيطة (Globus أو gLite أو غيرها)، بحيث تدير هذه الطبقة موارد Grid المنتشرة. ثم تثبت عليها واجهات SAGA التي تتعامل مع طبقة Grid الوسيطة عبر المحولات SAGA (SAGA adaptors)، بينما تكون حزم SAGA هي الواجهات التي نستدعيها داخل تطبيقاتنا.

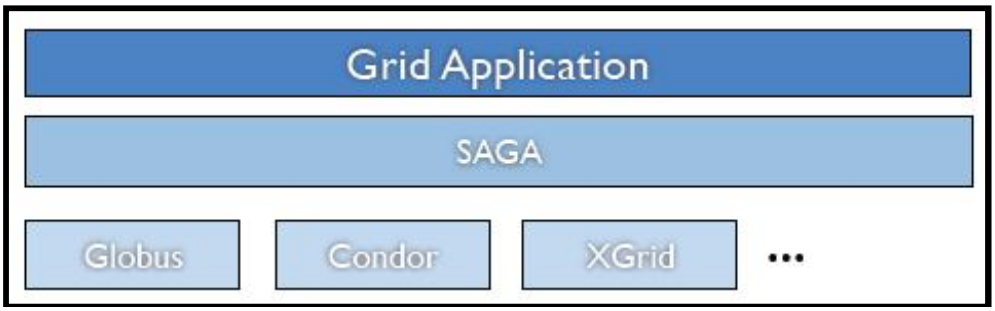


4.1 مميزات SAGA

- تسمح لك بكتابة برامج Grid متنقلة، وغير مرتبطة طبقة Grid معينة. فهي مستقلة عن الطبقات الوسيطة والمنصة (middleware and platform independent).
- تسمح بتطوير تطبيقات Grid بسرعة (rapid application development) عبر واجهاتها البرمجية البسيطة وعالية المستوى (simple high-level API)..

5.1 ما الذي لا تقوم به SAGA

- ليس بديل لطبقة Grid الوسيطة.
- ليست مرتبطة بلغة برمجة معينة.
- لا تغطي كل ميزات الـــ Grid.



تناول الأخطاء

توفر SAGA فئة استثناءات (exception class) تحتوي على تعريفات شفرات تستخدم بواسطة جميع طرق SAGA لتوضح الأخطاء.

```
Success              = 0      NotImplemented       = 1
IncorrectURL         = 2      IncorrectSession     = 3
AuthenticationFailed = 4      AuthorizationFailed  = 5
PermissionDenied     = 6      BadParameter         = 7
IncorrectState       = 8      AlreadyExists        = 9
DoesNotExist         = 10     ReadOnly             = 11
Timeout              = 12     NoAdaptor            = 13
NoAdaptorInfo        = 14     Unexpected           = 15
NoSuccess            = 16
```

نموذج المهمة (Task Model)

كل كائنات SAGA تطبق (implement) نموذج المهمة حيث يكون لكل طريقة ثلاثة أشكال (flavors)؛ هي:

① نسخة تزامنية (synchronous version)، التطبيق.

② نسخة غير متزامنة (asynchronous version)، نسخة تزامنية لفت في مهمة (خيط) وابتدأت.

③ نسخة المهمة (Task version)، نسخة تزامنية لفت في مهمة، لكنها لم تبدأ

المحول (adapter) حيث يمكنه تطبيق نسخته غير المتزامنة..

نموذج المراقبة (Monitoring Model)

هنالك بعض كائنات SAGA التي تطبق واجهة المراقبة (monitoring interface).

أسئلة تقويم ذاتي



1. عرِّف SAGA
2. صف بالرسم الشكل العام لـــ SAGA.
3. اذكر مميزات SAGA.
4. عدّد أشكال الطرق في نموذج المهمة.

2. الحزم الوظيفية (functional packages)

هي مجموعة من الكائنات (objects) والطرق (methods) المخصصة لمزايا محددة في طبقة Grid.الوسيطة، وتطبق في شكل مكتبات C++ ديناميكية. (dynamic libraries)

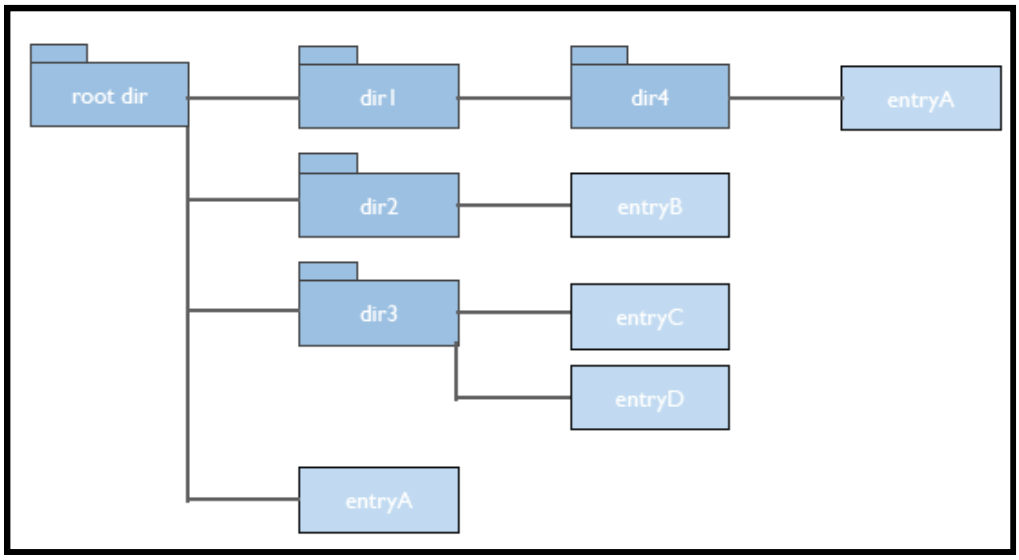
هنالك العديد من حزم SAGA، مثل:

1. Namespace Package
2. File Package
3. Job Package
4. Logical File Package (replica)
5. RPC Package
6. Stream Package□

Namespace Package API 1.2

هي بنية البيانات الأساسية في SAGA، وهي تركيبة هرمية من دلائل (directories) و مداخل (entries)، وقد يحتوي الدليل على دلائل أخرى أو مداخل. والمدخل يمكن أن يكون كائناً SAGA.

مثال لهرمية namespace



يدعم Namespace API البحث في الدلائل والمداخل، ولديه مؤشر إلى المداخل والدلائل. ويحتوي على عدة طرق مثل:

- get_url(), get_cwd(), get_name(), exists(), is_link()
- read_link(), change_dir(), list(), find()
- get_num_entries(), get_entry()
- copy(), link(), move(), remove()

مثال برمجي يوضح استخدام بعض طرق Namespace API

```
01: // Recursive Namespace structure listing
02: //
03: void list_dir( std::string & url) {
04: saga::ns_dir dir (url);
```

```
05:
06: for ( int i = 0; i < dir.get_num_entries (); i++ )
07: {
08: string name = dir.get_entry (i);
09:
10: std::cout << name << std::endl
11:
12: if( dir.is_dir(name) )
13: {
14: list_dir(name);
15: }
16: }
```

2.2 واجهات حزمة الملفات (File Package API)

هي namespace تحتوي على كائنات للملفات. وهو تمديد للـــ namespace API مع طرق للدخل والخرج. حيث تحتوي على طرق الدخل والخرج القياسية؛ مثل read(), write(), seek(). وتدعم هذه الحزمة الدخل والخرج المبعثر (scattered I/O)، حيث تكون عمليات القراءة والكتابة باستخدام متجهات تخزين (buffer vectors). أيضا توفر إمكانية تجميع عدة عمليات دخل وخرج في نداء واحد (single call). وتعتبر هذه الواجهات فعالة للملفات الكبيرة، وغير فعالة للملفات الصغيرة، تعتمد على دعم من الطبقة الوسطى (needs middleware support). الطرق المستخدمة في هذه الحزمة هي read_v(), write_v():

مثال برمجي

يقوم بقراءة أول 10 بايتات لملف إذا كان حجم الملف أكبر من 10 بايتات.

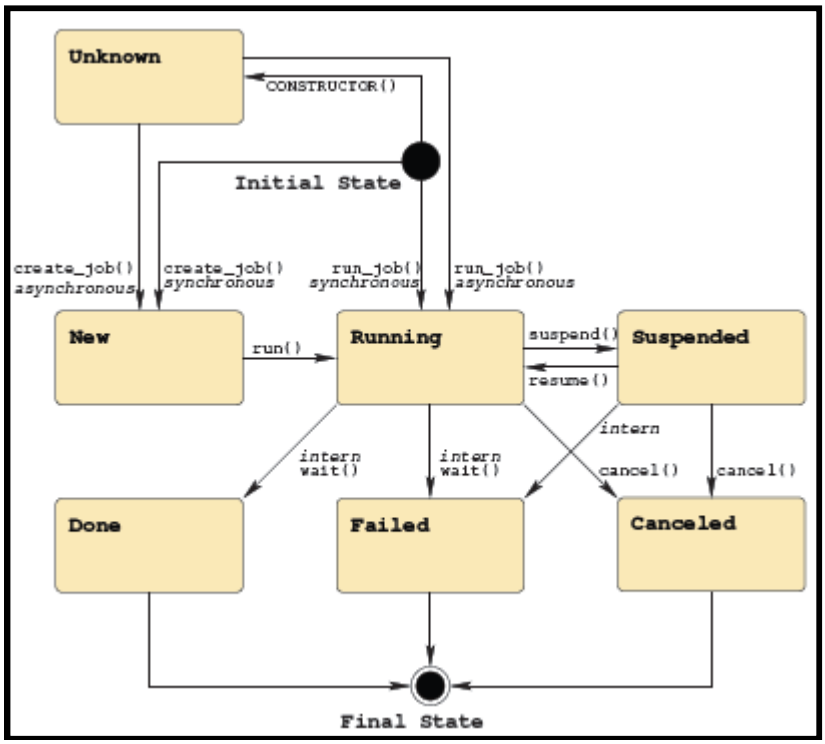
```
01: saga::file my_file
("griftp://gridhub/~/result.dat);
02:
03: off_t size = my_file.get_size ();
04:
05: if ( size > 10 )
06: {
```

```
07: char buffer[11];
08: long bufflen;
09:
10: my_file.read (10, buffer, &bufflen);
11:
12: if ( bufflen == 10 )
13: {
14: std::cout << buffer << std::endl;
15: }
16: }□
```

3.2 واجهات حزمة العمل (Job Package API

هي تمديد لمنوذج المهام (Extends the SAGA task model API)، ويتم فيها وصف المهامة، وإضافة مهمة والتحكم فيها (Job submission & control). حيث نستخدم طرق؛ مثل create_job(), run_job() لإضافة مهمة. ويمكن التحكم في المهمة بالطرق run(), cancel (), suspend(), resume(). أيضا هنالك طرق أخرى في هذه الحزمة؛ مثل checkpoint(), migrate().

Job state model - extends task state model



Job I/O redirection support

- **stdin, stdout and stderr available as C++ streams**
- **Methods: get_stdin(), get_stderr(), get_stdout()**
- **Job reconnection support**
- **(Re-)connect to running job based on JobID**
- **Job doesn't need necessarily to be a "SAGA Job"**

مثال برمجي

يوضح كيف يمكن إضافة مهمة بسيطة وإنتظارها حتى تكتمل.

```
01: //
02: //
03: saga::job_description jobdef;
04: jobdef.set_attribute ("Executable", "job.sh");
05:
06: saga::job_service js;
```

```
07: saga::job job = js.create_job ("remote.host.net",
jobdef);
08:
09: job.run();
10:
11: while( job.get_state() == saga::job::Running )
12: {
13: std::cout << "Job running with ID: "
14: << job.get_attribute("JobID") << std::endl;
15: sleep(1);
16: }□
```

4.2 واجهات حزمة النداء البعيد (RPC Package API)

عزيزي الدارس، يعتمد نداء الإجراء البعيد على معايير OGF GridRPC. الطريقة تصاغ في SAGA-URL كالتالي:

gridrpc://server.net:1234/my_function

ويكون هنالك مصفوفة من البنيات (array of structures) تحتوي على المدخلات (parameters) والقيم الراجعة (return values).

مثال برمجي

هنا نقوم باستدعاء إجراء بعيد.

```
01: //
02: //
03: rpc rpc ("gridrpc://fs0.das2.cs.vu.nl/matmul1");
04:
05: std::vector <saga::rpc::parameter> params (2);
06:
07: params[0].buffer = // ptr to matrix A
08: params[0].size = sizeof (buffer);
09: params[0].mode = saga::rpc::InOut;
10:
11: params[1].buffer = // ptr to matrix B
12: params[1].size = sizeof (buffer);
13: params[1].mode = saga::rpc::In;
```

```
14:
15: rpc.call (&params);
16:
17: // A now contains the result
```

أسئلة تقويم ذاتي



1. وضِّح مفهوم الحزم الوظيفية (**functional packages**) ؟.
2. عدِّد حزم **SAGA**.

الخلاصة

- عرفت الوحدة SAGA بأنها واجهات تطبيق برمجية عالية المستوى توفر الوظائف الأساسية لبناء تطبيقات موزعة، توفر أدوات وبيئة عمل مستقلة عن البنية التحتية لطبقة الـ Grid.

- عددت الوحدة مميزات SAGA بأنها تسمح بكتابة برامج Grid متنقلة وغير مرتبطة بطبقة Grid معينة.

- تسمح بتطوير تطبيقات Grid بسرعة عبر واجهاتها البرمجية البسيطة وعالية المستوى.

- بينت الوحدة أن كل كائنات SAGA تطبق نموذج المهمة حيث يكون لكل طريقة ثلاثة أشكال، هي: ① نسخة تزامنية التطبيق. ② نسخة غير متزامنة. ③ نسخة المهمة.

- عرفت الوحدة الحزم الوظيفية بأنها مجموعة من الكائنات (objects) والطرق (methods) المخصصة لمزايا محددة في طبقة Grid الوسيطة، وتطبق في شكل مكتبات C++ ديناميكية. (dynamic libraries)

- عددت الوحدة حزم SAGA، وهي :

Namespace Package API : وهي بنية البيانات الأساسية في SAGA. وهي تركيبة هرمية من دلائل (directories) و مداخل (entries).

واجهات حزمة الملفات (File Package API) : هي namespace تحتوي على كائنات للملفات.

واجهات حزمة العمل (Job Package API) : هي تمديد لنموذج المهام (Extends the SAGA task model API) ويتم فيها وصف المهامة، وإضافة مهمة والتحكم فيها.

واجهات حزمة النداء البعيد (RPC Package API) : ويعتمد نداء الإجراء البعيد على معايير OGF GridRPC.

مسرد المصطلحات

❖ **SAGA**

هي واجهات تطبيق برمجية عالية المستوى توفر الوظائف الأساسية لبنـــاء تطبيقـــات موزعة، وتوفر أدوات وبيئة عمل مستقلة عن البنية التحتية لطبقة الـــ Grid.

المراجع

SAGA, http://saga.cct.lsu.edu.

SAGA Quick reference, http://saga.cct.lsu.edu/software/cpp/documentation/quickstart-guide.

SAGA overview and Introduction, http://www.slideshare.net/ISSGC/saga-issgc09

الملاحق

ملحق (أ)

أمثلة برمجية على خيوط جافا

<u>لماذا نحتاج لأكثر من خيط ؟</u>

- **تجاوب جيد وسريع مع المستخدم**
- **الاستفادة من العتاد (تعدد المعالجات)**
- **القيام بأعمال أخرى بينما ننتظر تنفيذ عمليات بطيئة (مثل عمليات الدخل والخرج وتحميل البرامج عبر الإنترنت).**

<u>جافا تدعم الخيط</u>

تحتوي جافا على مكتبة مخصصة للبرمجة بمفهوم الخيط وتدعم ذلك دعم كامل عبر الفئة Thread.

<u>مثال (1)</u>

كل برنامج يحتوي على خيط واحد على الأقل، في جافا تعتبر الدالة main هي هذا الخيط (الخيط الرئيسي)، ويمكننا إضافة اكثر من خيط إلى main، بحيث تنفذ هذه الخيوط معا في وقت واحد. الخيط الرئيسي main ينفذ بواسطة JVM. لتوضيح ذلك دعنا نكتب برنامج عادي (بخيط واحد)، ثم نعدل البرنامج بإضافة خيط ثاني مع الخيط الرئيسي، ليصبح لدينا برنامج مكون من خيطين.

برنامج جافا عادي (خيط واحد هو main)

```
public class singleTh {
public static void main(String[] args) {
for ( int i = 0; i < 10; i++ ) {
System.out.println("Main
thread");
}
}
}
```

مخرج هذا البرنامج سيكون:

```
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
```

الآن سنعدل هذا البرنامج بإضافة خيط آخر إليه، ولكن قبل ذلك دعونا نوضح كيف نضيف خيط إلى البرنامج.

إضافة خيط:

هنالك 4 خطوات لإضافة خيط للبرنامج هي:

1. نرث صفات الفئة Java.Lang.Thread ، وهي الفئة التي تدعم الخيط في الجافا.

```
Public duoTh extends Thread {
// ….
```

}

2. نعيد كتابة الدالة **run()** الموجودة بالفئة **Thread**، داخل الفئة **duoTh** أعلاه (هنا نقول أننا عملنا **overriding** للدالة **run()**).

```
Public void run(){
      for ( int i = 0; i < 10; i++ ) {
  System.out.println("Second thread");
                                    }
```

3. ننشئ كائن من الفئة **duoTH** أعلاه.

```
duoTh dth = new duoTh();
```

4. ننفذ الخيط (الكائن المنشأ في (3)) بالأمر **start()**

```
dth.start();
```

بتجميع هذه الخطوات سيكون برنامجنا كما يلي:

```
public class duoTh extends Thread {
public void run(){
for ( int i = 0; i < 10; i++ ) {
System.out.println("Second thread");
}
}
public static void main(String[] args) {
duoTh dth = new duoTh();
dth.start();
for ( int i = 0; i < 10; i++ ) {
System.out.println("Main thread");
}
}
}
```

سيكون مخرج البرنامج أعلاه كما يلي:

```
Main thread
Main thread
Main thread
Main thread
Main thread
Second thread
Second thread
```

```
Second thread
Second thread
Second thread
Second thread
Second thread
Second thread
Second thread
Second thread
Main thread
Main thread
Main thread
Main thread
Main thread
```

إذا نفذت البرنامج مرة ثانية فقد يكون المخرج مختلف (مثلا كما يلي):

```
Main thread
Second thread
Second thread
Main thread
Second thread
Second thread
Second thread
Second thread
Second thread
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Second thread
Second thread
Second thread
Main thread
Main thread
```

ملحوظة

- **هنا يقوم الخيط الرئيسي main بالعمل وتشغيل الخيط dTh معه في نفس الوقت بواسطة الأمر dTh.start().**

- **الخيط الرئيسي يظهر الرسالة Main thread عشرة مرات.**
- **الخيط الثاني dTh يظهر الرسالة Second thread عشرة مرات.**
- **ترتيب التنفيذ غير متوقع.**

<u>ملخص</u>

في هذا المثال وضحنا كيف يمكنك إنشاء برنامج جافا يتكون من أكثر من خيط وذلك بالخطوات الأربعة التالية:

1. **إنشاء فئة ترث الكائن Thread.**
2. **إعادة كتابة الدالة ()run في الفئة أعلاه ((overriding run()).**
3. **إنشاء كائن من الفئة أعلاه.**
4. **تنفيذ الكائن كخيط منفصل بالأمر ()start.**

□

Pthread Library Routines Reference

For convenience, an alphabetical list of Pthread routines, linked to their corresponding man page, is provided below.

pthread_atfork
pthread_attr_destroy
pthread_attr_getdetachstate
pthread_attr_getguardsize
pthread_attr_getinheritsched
pthread_attr_getschedparam
pthread_attr_getschedpolicy
pthread_attr_getscope
pthread_attr_getstack
pthread_attr_getstackaddr
pthread_attr_getstacksize
pthread_attr_init
pthread_attr_setdetachstate
pthread_attr_setguardsize
pthread_attr_setinheritsched
pthread_attr_setschedparam
pthread_attr_setschedpolicy
pthread_attr_setscope
pthread_attr_setstack
pthread_attr_setstackaddr
pthread_attr_setstacksize
pthread_barrier_destroy
pthread_barrier_init
pthread_barrier_wait
pthread_barrierattr_destroy
pthread_barrierattr_getpshared
pthread_barrierattr_init
pthread_barrierattr_setpshared
pthread_cancel
pthread_cleanup_pop
pthread_cleanup_push
pthread_cond_broadcast

pthread_cond_destroy
pthread_cond_init
pthread_cond_signal
pthread_cond_timedwait
pthread_cond_wait
pthread_condattr_destroy
pthread_condattr_getclock
pthread_condattr_getpshared
pthread_condattr_init
pthread_condattr_setclock
pthread_condattr_setpshared
pthread_create
pthread_detach
pthread_equal
pthread_exit
pthread_getconcurrency
pthread_getcpuclockid
pthread_getschedparam
pthread_getspecific
pthread_join
pthread_key_create
pthread_key_delete
pthread_kill
pthread_mutex_destroy
pthread_mutex_getprioceiling
pthread_mutex_init
pthread_mutex_lock
pthread_mutex_setprioceiling
pthread_mutex_timedlock
pthread_mutex_trylock
pthread_mutex_unlock
pthread_mutexattr_destroy
pthread_mutexattr_getprioceiling
pthread_mutexattr_getprotocol
pthread_mutexattr_getpshared
pthread_mutexattr_gettype
pthread_mutexattr_init
pthread_mutexattr_setprioceiling
pthread_mutexattr_setprotocol

pthread_mutexattr_setpshared
pthread_mutexattr_settype
pthread_once
pthread_rwlock_destroy
pthread_rwlock_init
pthread_rwlock_rdlock
pthread_rwlock_timedrdlock
pthread_rwlock_timedwrlock
pthread_rwlock_tryrdlock
pthread_rwlock_trywrlock
pthread_rwlock_unlock
pthread_rwlock_wrlock
pthread_rwlockattr_destroy
pthread_rwlockattr_getpshared
pthread_rwlockattr_init
pthread_rwlockattr_setpshared

ملحق (ج)

تنفيذ العمليات باستخدام fork()

أحيانا تحتاج كمبرمج خبير تنفيذ عمليات تعمل على التوازي دون الإعتماد على مكتبة خيوط أو عمليات مثل Pthreads، في هذه الحالة لابد من أن تكون ملما بنداءات النظام (system calls) الموجودة في نظام التشغيل. وكيفية استخدامها.

من أهم النداءات التي تمكننا من تشغيل أكثر من عملية هي نداء النظام fork() والذي يولد أكثر من عملية.

في هذا الملحق نسرد بعض البرامج التي تستخدم نداءات النظام في لينكس للتحكم في العمليات من داخل برامج C.

<u>خطوات تنفيذ البرنامج:</u>

قبل البدء في سرد أمثلة البرامج نوضح هنا خطوات كتابة وترجمة وتنفيذ برامج C على لينكس (توزيعة أوبونتو):

- **أفتح الطرفية**
- **أكتب الأمر gedit (محرر لكتابة شفرة البرنامج).**
- **أكتب البرنامج أدناه في المحرر.**
- **أحفظ الملف بالاسم creatProcess.c**
- **ترجم البرنامج بالأمر:**
  - **cc -o creatProcess creatProcess.c**
- **نفذ البرنامج بعد الترجمة وتصحيح الأخطاء بالأمر:**
  - **./ creatProcess**

برنامج (1)

```
#include <stdio.h>
#include <sys/types.h>
#define MAX_COUNT 200
void ChildProcess(void); /* child process prototype */
void ParentProcess(void); /* parent process prototype */
void main(void) {
        pid_t pid;
        pid = fork();
        if (pid == 0)
        ChildProcess();
        else ParentProcess();
 }
void ChildProcess(void) {
        int i=5;
        printf(" This line is from child, value = %d\n", i);
        printf(" *** Child process is done ***\n");
 }
void ParentProcess(void) {
```

```
 int i=9;
printf("This line is from parent, value = %d\n", i);
printf("*** Parent is done ***\n");
}
```

جرب البرنامج أعلاه ووضح ما هو المخرج ؟

برنامج (2)

البرنامج التالي يوضح كيفية استخدام استدعاءات النظام بواسطة برنامج C++، حيث تم إنشاء عملية باستدعاء fork()، ثم اختبار القيمة الراجعة منها، للتميز بين العملية الإبن والعليمة الأب.

```
#include <iostream>
#include <string>

// Required by for routine
#include <sys/types.h>
#include <unistd.h>

using namespace std;

int globalVariable = 2;

main()
{
   string sIdentifier;
   int    iStackVariable = 20;

   pid_t pID = fork();
   if (pID == 0)                // child
   {
      // Code only executed by child process

      sIdentifier = "Child Process: ";
      globalVariable++;
      iStackVariable++;
    }
    else if (pID < 0)            // failed to fork
    {
        cerr << "Failed to fork" << endl;
```

```
        exit(1);
        // Throw exception
    }
    else                                    // parent
    {
      // Code only executed by parent process

      sIdentifier = "Parent Process:";
    }

    // Code executed by both parent and child.

    cout << sIdentifier;
    cout << " Global variable: " << globalVariable;
    cout << " Stack variable: "  << iStackVariable << endl;
}
```

ملحق (د)

استخدام OMPI و OpenMP على Visual studio.Net 2005

في هذا الملحق سنوضح كيف يمكن إعداد visual C++ الممضمنة في Visual studio 2005 لتنفذ برامج OpenMP وبرامج MPI، في البداية سنوضح خطوات تهيئة واستخدام OpenMP ثم تليها خطوات تثبيت واستخدام MPI.

OpenMP على Visual C++

كانت بداية OpenMP في 1997 كواجهة تطبيقية (API) لتصميم تطبيقات multithreaded. وكانت تعمل مع C، C++، و Fortran. ولكن النسخة الحالية OpenMP 2.0 و Visual C++ 2005 تعملان بكامل المواصفات القياسية لهذه الواجهة.

كل موجهات visual c++ للتعامل مع OpenMP تبدأ بـ: #pragma omp

تفعيل OpenMP في Visual studio

افتح visual studio



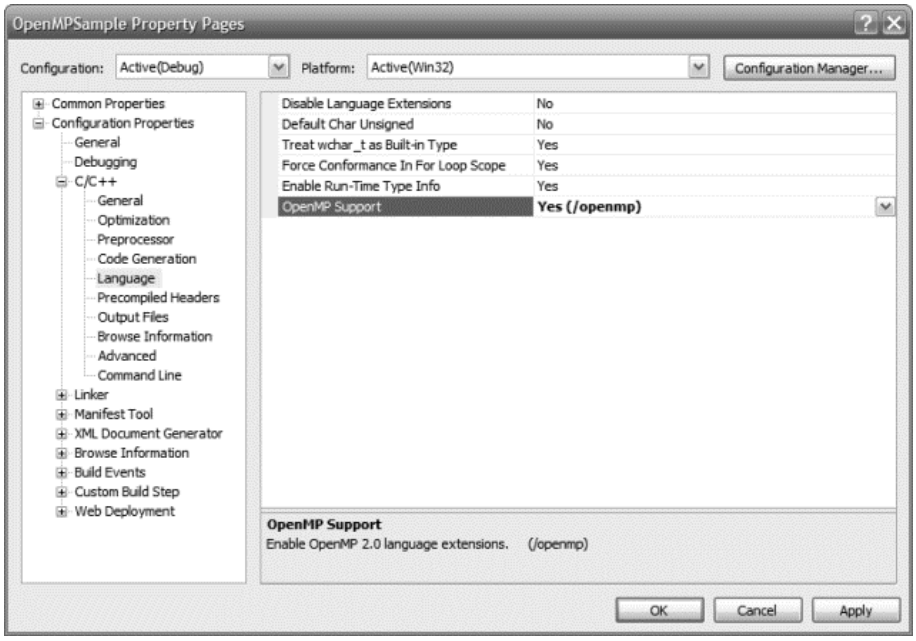
لإستخدام OpenMP على Visual studio نحتاج فقط :

- **Visual studio professional أو أعلى**
- **جهاز متعدد المعالجات (multiprocessor) أو متعدد القلوب (multi core) لتشعر بالفرق في الأداء.**
- **خوارزمية لتنفيذها بالتوازي.**

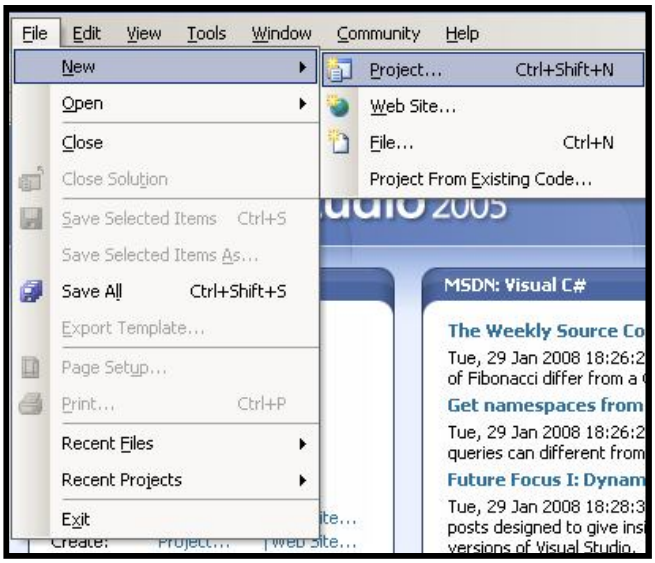
يستخدم فيجوال ستديو واجهة OpenMP 2.0 وهي موجودة بالملف vcomp.dll .

لإستخدام OpenMP عليك القيام بالخطوات التالية:

1. **تضيف الملف omp.h لمشروعك**
2. **تفعل OpenMP في خصائص مشروعك (project properties).**



انشيء مشروع جديد



أكتب البرنامج التالي:

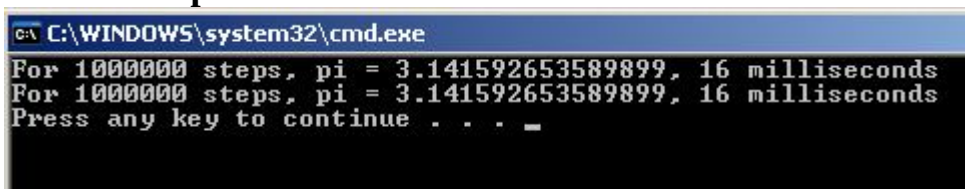
```
// cpp_compiler_options_openmp.cpp
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

volatile DWORD dwStart;
volatile int global = 0;

double test2(int num_steps) {
   int i;
   global++;
   double x, pi, sum = 0.0, step;

   step = 1.0 / (double) num_steps;
```

```
   #pragma omp parallel for reduction(+:sum)
private(x)
   for (i = 1; i <= num_steps; i++) {
      x = (i - 0.5) * step;
      sum = sum + 4.0 / (1.0 + x*x);
   }

   pi = step * sum;
   return pi;
}

int main(int argc, char* argv[]) {
   double   d;
   int n = 1000000;

   if (argc > 1)
      n = atoi(argv[1]);

   dwStart = GetTickCount();
   d = test2(n);
   printf_s("For %d steps, pi = %.15f, %d
milliseconds\n", n, d, GetTickCount() - dwStart);

   dwStart = GetTickCount();
   d = test2(n);
   printf_s("For %d steps, pi = %.15f, %d
milliseconds\n", n, d, GetTickCount() - dwStart);
}
```

التنفيذ:



Result – output:



جرب هذا البرنامج في اجهزة متعددة المعالجات أو معالجات متعددة القلوب ولاحظ الفرق في النتيجة.

ملحق (هـــ)

خطوات تنفيذ البرنامج PThreads

خطوات كتابة وترجمة وتنفيذ برامج C على لينكس (توزيعة أوبونتو):

- **أفتح الطرفية**
- **أكتب الأمر gedit (محرر لكتابة شفرة البرنامج).**
- **أكتب البرنامج أدناه في المحرر.**
- **أحفظ الملف بالاسم creatProcess.c**
- **نترجم البرنامج كالآتي:**

```
C compiler: cc -lpthread pthread1.c
```

أو

```
C++ compiler: g++ -lpthread pthread1.c
```

- **ثم ننفذ البرنامج كما يلي:**

```
./a.out
```

برنامج

أكتب هذا البرنامج في بعد فتح محرر gedit ثم أحفظه بالاسم pthread1.c ثم ترجمه ونفذه حسب الخطوات الموضحة أعلا. من المفترض أن يكون المخرج كالموضح أدنى البرنامج.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *p_message( void *ptr );
```

```
main(){
     pthread_t thread1, thread2;
     char *m1 = "Thread 1";
     char *m2 = "Thread 2";
     int  r1, r2;
/*  ننشئ خيطين، كل خيط ينفذ نفس الدالة لكن بمعطى مختلف  */
/*  إنشاء الخيط الأول  */
r1 = pthread_create( &thread1, NULL, p_message,
(void*) m1);
/*  إنشاء الخيط الثاني  */
r2 = pthread_create( &thread2, NULL, p_message,
(void*) m2);
/*  نجعل الدالة الرئيسية تنتظر حتى يكتمل تنفيذ الخيطين  */
     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);
     printf("Thread 1 returns: %d\n",r1);
     printf("Thread 2 returns: %d\n",r2);
     exit(0);
}
/*  الدالة التي تنفذ داخل كل خيط مع مدخل مختلف  */
void *p_message ( void *ptr )
{     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
}
```

مخرج البرنامج:

```
Thread 1
Thread 2
Thread 1 returns: 0
Thread 2 returns: 0
```