

البرمجة المتوازية

Parallel Programming



عبد الرحمن أحمد محمد عثمان

جامعة أم القرى - كلية الحاسوب بالقنفذة

جامعة الشيخ عبدالله البدري - برس - السودان

النسخة الورقية حقوقها محفوظة لجامعة السودان المفتوحة (2011)

ومنه النسخة الإلكترونية المجانية الأولى (2016) - نسخة غير مدققة

مقدمة

أصبحت البرمجة المتوازية ضرورة من ضرورات العصر. فمعظم الحواسيب المتوفرة الآن في السوق متعددة النواة وتحتاج إلى هذا النوع من البرمجة للاستفادة القصوى منها.

أيضا ظهر عصر البيانات الضخمة جعل من الصعب تحميل وتحليل بيانات كبيرة الحجم في حاسب واحد، فلا بد من التوازي. لهذا السبب جاء هذا الكتاب كمدخل للطالب ليضعه في بداية الطريق.

تطرقنا في هذا الكتاب لمفاهيم التوازي ثم افردنا أبواب كاملة للجانب العملي. حيث تم توضيح كيف يمكن كتابة برامج متوازية باستخدام على كل من:

Java •

pThreads •

OpenMP •

.MPI •

في الجانب العملي غطينا أساسيات التي تمكنك من معرفة كيف كتب برنامجا متوازيا.

هذا الكتاب موجه للذين يعرفون البرمجة المتسلسلة العادلة وليس لهم دراية بالبرمجة المتوازية، فهو لا يتطرق إلى أساليب البرمجة العادلة..

لتنفيذ البرامج استخدمنا:

Netbeans لتنفيذ برمج Java •

فيجوال C++ (Visual studio 2010) لتنفيذ برمج pThread، OpenMP، و MPI •

في الختام لا يخلو عمل من خطأ أو نسيان أو نقص، لذلك ارجو من القارئ الكريم ارسال ملاحظاته على الايميل التالي:

parallelprogramming7@gmail.com

تنبيه:

هذه النسخة غير مكتملة وتحتاج إلى تنقية وإضافات ولكن نسبة حاجة كبير من الطلاب لها قررت أن أرفعها ليستفاد منها، ومساعدتكم يكمننا تنفيتها وإضافة ما تحتاجونه فيها باذن الله .

والله الموفق ،،،

توجد كتب أخرى ذات بالموضوع متاحة على الروابط التالية:

1. مدخل للنظم الموزعة (بالعربي):

https://www.researchgate.net/publication/307122357_Distributed_Systems_In_Arabic

2. نظم التشغيل (بالعربي):

https://www.researchgate.net/publication/306960494_Operating_Systems_In_Arabic

جدول المحتويات المختصر

9	عصر التعديلة والبرمجة المتوازية
20	أنواع الحاسوبات
29	نماذج البرمجة المتوازية
36	مفهوم الخيط
44	التزامن (synchronization)
57	خيوط جافا
67	برمجة الخيوط باستخدام Pthreads
87	البرمجة باستخدام OpenMP
100	برمجة الذاكرة الموزعة باستخدام MPI
الملاحق	
123	سرعة المعالج إلى أين؟
126	تفعيل OpenMP في فيجوال استديو 2010
129	اعداد MPI في فيجوال استديو 2010
133	تشغيل pThread في فيجوال استديو 2010
138	المراجع

جدول المحتويات المفصل

9.....	الباب الأول.....
9.....	عصر المتعددة والبرمجة المتوازية.....
9.....	1. البرمجة التسلسلية.....
10.....	1.2. مفهوم التعاون.....
12.....	1.3. تعاون الحاسوب.....
13.....	1.4. المدفوع من التعاون (التوافزي).....
14.....	1.5. أساليب الاتجاه نحو المتعددية.....
14.....	1.6. أمثلة التطبيقات التي تحتاج قوة معالجة عالية.....
15.....	1.7. لماذا تتسم بزيادة سرعة المعالج مستحلكية.....
17.....	1.8. تعريفاته ظهرت في عصر المتعددية.....
19.....	1.9. النلاطة.....
20.....	الباب الثاني.....
20.....	أدوات الحاسوب.....
20.....	2.1. الأنظمة المتعددة.....
22.....	2.2. الأنظمة متعددة المعالجات (multiprocessor).....
24.....	2.3. الأنظمة متعددة الحاسوبات (Multicomputer).....
24.....	2.3.1. الأنظمة متعددة الحاسوبات المتجانسة (Homogenous Multicomputer Systems).....
27.....	2.3.2. الأنظمة متعددة الحاسوبات المترابطة (Heterogeneous Multicomputer Systems).....
27.....	2.4. ملخص.....
29.....	الباب الثالث.....
29.....	نماذج البرمجة المتوازية.....
29.....	3.1. مقدمة.....
29.....	3.2. تصنيفه فللين.....
30.....	3.3. نماذج البرمجة المتوازية.....
34.....	3.4. النلاطة.....
36.....	الباب الرابع.....
36.....	مفهوم النط.....
36.....	4.1. تمهيد.....
37.....	4.2. تعريفه النط (thread).....

38.....	4.3 . المواراة الخيالية والمواراة الحقيقة.
39.....	4.4 . أنواع الخيوط.....
39.....	4.4.1 . خيط المستخدم (user thread).....
40.....	4.4.2 . خيط النواة.....
40.....	4.5 . التحول بين العمليات.....Context Switch
41.....	4.6 . استخداماته الخيط في المعالج الواحد.....
41.....	4.6.1 . مدرر النصوص.....
41.....	4.6.2 . مندم الوريد.....
42.....	4.7 . ملخص.....
44.....	الباب السادس.....
44.....	التزامن (synchronization).....
44.....	5.1 . مفهوم التوازي.....Concurrency
44.....	5.2 . تعاون العمليات.....
45.....	5.3 . النزاع (competition).....
46.....	5.3.1 . مشاكل النزاع.....
47.....	5.4 . مشاكل التزامن اللاسلكية.....
47.....	5.4.1 . مشكلة القراءة والكتابة (reading and writing problem).....
48.....	5.4.2 . مشكلة المنتج والمستهلك (producer-consumer).....
51.....	5.4.3 . مشكلة عشاء الفلسفه (Dining philosophers problem).....
53.....	5.4.4 . مشكلة مدخني السيادي (Cigarette smokers problem).....
54.....	5.4.5 . اللقاء (Rendezvous).....
54.....	5.4.6 . مشكلة العلاق النائم (sleeping barber).....
56.....	5.5 . ملخص.....
57.....	الباب السادس.....
57.....	خيوط جافا.....
57.....	6.1 . إنشاء الخيط.....
57.....	إنشاء خيط بالطريقة الأولى.....
58.....	الطريقة الثانية.....
60.....	6.2 . دورة حياة خيط جافا.....
60.....	6.3 . التعامل مع الخيط.....
61.....	6.4 . حالات الخيط.....
63.....	6.5 . مثال برمجي.....

67.....	الباب السادس
67.....	برمجة الخيوط باستخدام Pthreads
67.....	7.1. مكتبات POSIX thread (Pthread)
67.....	7.2. نموذج الذاكرة المشتركة
68.....	7.3. واجهة التطبيقات البرمجية (Pthreads API)
68.....	7.4. إنشاء خيط
69.....	7.5. إنتهاء خيط
70.....	7.6. صفات الخيط (Thread Attributes)
72.....	7.7. تمرير قيمة إلى الخيط (Passing Arguments to Threads)
74.....	7.8. الإنضمام (Joining)
74.....	7.9. الإنفصال (Detaching)
76.....	7.10. إدارة المكادمة
78.....	7.11. متغيرات الميورتكس (Mutex Variables)
81.....	7.12. السيمافور
83.....	7.13. المتغيرات الشرطية (Condition Variables)
86.....	7.14. ملخص
87.....	الباب الثامن
87.....	برمجة باستخدام OpenMP
87.....	8.1. نقطة من OpenMP
87.....	8.2. أول برنامج
91.....	8.3. تحديد عدد الخيوط
91.....	8.4. تشارك البيانات
92.....	8.5. تغريد الخيوط (Fork/Join)
93.....	8.6. توزيع العمل (Load Balancing)
94.....	8.7. عقابات في برمجة الخيط
94.....	8.8. تزامن الخيط في OpenMP
95.....	8.9. الاندماج (Reductions)
96.....	8.10. جدولة التكرار (الحلقة) (Loop Scheduling)
98.....	8.11. ملخص
100.....	الباب التاسع
100.....	برمجة الذاكرة الموزعة باستخدام MPI
100.....	9.1. مقدمة

101.....	9.2. نسخة من MPI.....
102.....	9.3. أول برنامج.....
104.....	9.4. برنامج Circuit Satisfiability.....
108.....	9.5. تبادل الرسائل في MPI.....
111.....	9.5. البقتاق (deadlock).....
112.....	9.6. قياس الأداء (benchmarking performance).....
114.....	9.7. الحالة MPI_Bcast.....
117.....	9.8. الحالة MPI_Scatter.....
118.....	9.9. الحالة MPI_Gather.....
119.....	9.10. الحالة MPI_Allgather.....
120.....	9.11. حالة الاتصال MPI_Reduce.....
121.....	9.12. الملخص.....
122.....	الملاحق.....
123.....	ملحق (أ).....
123.....	سرعة المعالج إلى أين؟.....
126.....	ملحق (بـ).....
126.....	تفعيل OpenMP في فيجوال استديو 2010.....
129.....	ملحق (ج).....
129.....	إعداد MPI في فيجوال استديو 2010.....
133.....	الملحق (د).....
133.....	تشغيل pThread في فيجوال استديو 2010.....
138.....	المراجع.....

الباب الأول

عصر التعددية والبرمجة المتوازية

معظم التطبيقات العملية الحديثة تطلب سرعة تنفيذ عالية بشره ونهم كبير بحيث لا يشعها زيادة سرعة المعالج الواحد مهما بلغت. لهذا السبب أتجهت معظم شركات تصنيع الحواسيب نحو إنتاج حاسبات ذات معالجات متعددة النواة (multi-core processors)، فظهرت المعالجات ثنائية النواة (dual-core)، ثم تبعتها الآن المعالجات رباعية النواة (quad-core). الإتجاه 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 الإستفادة من هذا التعدد، وإلا فلن يكون هنالك فرق بين معالج ذو نواة واحدة ومعالج ذو مائة نواة. أيضا هنالك نظم حاسبات متعددة المعالجات كما سنتوضح ذلك بالتفصيل في حديثنا عن المكونات المادية للبرمجة المتوازية.

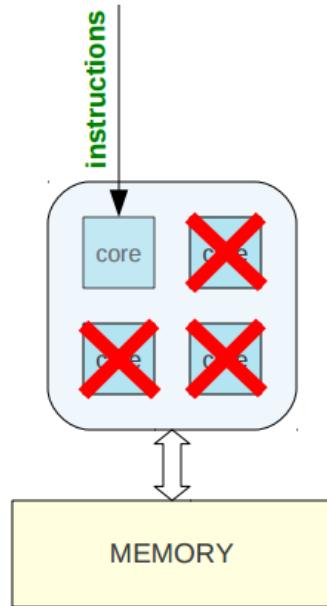
في صياغ حديثنا عن البرمجة المتوازية ستتجد إننا بحاجة للتمييز بين بعض المصطلحات التي ظهرت في عصر التعددية. سنتحدث عنها هنا باختصار لأننا قد نحتاج أن نتعامل معها في هذا الكتاب.

- تعدد المهام (Multitasking): القدرة على تنفيذ أكثر من مهمة واحدة في نفس الوقت.
- تعدد الخيوط (Multithreading): تنفيذ خيوط متعددة في نفس الوقت.
- تعدد المعالجة (Multiprocessing): مثل تعدد المهام، ولكن هنا يشترك أكثر من معالج في التنفيذ. بينما يستخدم معالج واحد في تعدد المهام.
- المعالجة المتوازية (Parallel Processing): أحيانا يقصد بها استخدام عدة معالجات في نظام كمبيوتر واحد.

1.1. البرمجة التسلسلية

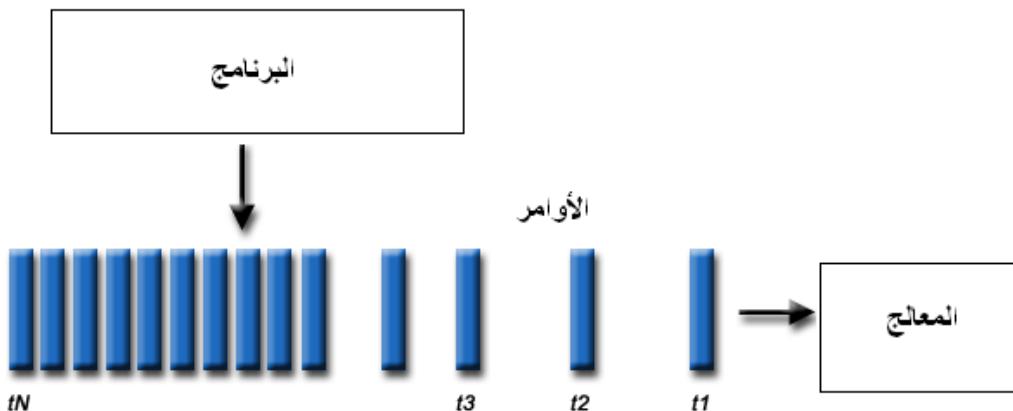
تعتبر البرمجة التقليدية المتسلسلة والتي تنفذ على جهاز ذو معالج واحد (أو نواة واحدة)، سواء كانت إجرائية (procedural) أو كائنية (object oriented)، هي المعروفة لدى الجميع ويستخدمها الناس لأكثر من 50 عاماً.

معظم الأجهزة والبرمجيات صممت لتنفذ هذه النوع من البرامج التسلسلية. فالبرنامج المتسلسل مكون من سلسلة من الأوامر (التعليمات) التي تنفذ بواسطة حاسب واحد تسلسليا (أي أمر تلو الآخر).



شكل 1-1: التنفيذ التسلسلي [1].

عادة ينفذ البرنامج المتسلسل على معالج واحد (ذو نواة واحدة)، حيث ينفذ المعالج أمرًا واحدًا 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 ($n \times 100$) وحدة زمنية. البرامج التسلسلية لا تستفيد من تعدد المعالجات وتعدد الأنوية في المعالج الواحد. فالتنفيذ سيتم على معالج (نواة) واحد وبقية المعالجات (الأنيوية) عاطلة غير مستفاد منها، شكل 1-1، 2-1.



شكل رقم 1-2: تنفيذ الأوامر تسلسلياً [2].

إذا لديك دراية بمفهوم التعاون (النفير)، يمكنك تجاوز فقرة مفهوم التعاون والانتقال إلى 1.3.

1.2. مفهوم التعاون

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

1. بذل جهد أكبر ونشاط أكثر.
2. نستخدم طرق ذكية لتقليل وقت المهمة.

3. الإستعانة بآخرين لمساعدتك.

يجب أن تكون المهمة قابلة للتقسيم حتى يساعدك الآخرين في إنجازها. مثلاً قراءة الكتاب لا تتم إلا تسلسلياً فلا يمكن لشخصين أو أكثر أن يستخدما نفس الكتاب في نفس الوقت، لكن مثلاً لبناء حائط أو حصاد زرع فيمكن العدد من الأشخاص التعاون في ذلك وفي وقت واحد.

أيضاً 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) تحتاج الإسراع في إنجاز الحصاد ليس لتقليل الوقت والجهد فقط وإنما أيضاً لتلافي الضرر (فوات زمن الحصاد).

أمثلة تشبيهية للتعاون

1. الحصاد

لحصاد زرع في مساحة حجمها 10 فدان مثلاً، إذا كان لدينا عشرة عمال، فيمكن لكل عامل حصاد مساحة فدان واحد، وبالتالي لو كان العمال بنفس القوة وكل عامل يحتاج يوم واحد لحصاد الفدان الذي خصص له، فإن العمال العشرة سيحصدون كل المساحة في يوم واحد. لكن لو كان عدد العمال 5 فسينجز العمل في يومين، ولو كان لدينا عامل واحد فقط فسيحتاج إلى عشرة أيام لحصاد المساحة كاملة.
في المثال أعلاه نجد أنه:

لو كانت عملية الحصاد على عامل واحد فسيحتاج إلى عشر أيام لإنجاز العمل، ولكن يمكنه تقليل الفترة الزمنية بإحدى طريقتين هما:

1. بذل جهد أكبر لتقليل زمن الحصاد.
 2. استخدام طرق ذكية لتسريع الحصاد مثل استخدام حاصدة ميكانيكية أو آلة كهربائية أو غيرها.
- في حالة وجود أكثر من عامل يمكننا تقسيم العمل بين العمال، وقد بذل عامل جهد أكبر من العمال الآخرين أو قد يستخدم طرق ذكية لتقليل زمن عمله وبالتالي إنجاز العمل في أقصر فترة زمنية ممكنة.

2. تجميع السيارات

يتم تجميع أجزاء السيارة بالتوازي. فالملاكيتة، الانوار الامامية، المبكل، المقود،... تصنع بواسطة شركات مختلفة في نفس الوقت وتأتي منتجاتها إلى خط التجميع في الوقت المناسب.

عملية تجميع أجزاء السيارة بواسطة شخص واحد تعتبر بطيئة وقد تكون مستحبة. تقسيم هذه عملية إلى مهام صغيرة وإعطاء كل مهمة إلى شركة يتغير الحل الأمثل.

بنفس القدر البرامج الكبيرة والمعقدة يمكن تقسيمها إلى برامج صغيرة وتوزيعها على عدة معالجات بحيث ينفذ كل معالج المهام بالموازي مما يزيد سرعة التنفيذ.

3. الزراعة

إذا كان لدينا حقل كبير مساحته 100 فدان يريد زراعته، فإذا كان زراعة الفدان الواحد تحتاج 5 ساعات إذا قام بها مزارع واحد يدوياً، فهذا يعني أن زراعة المائة فدان ستس取 5 × 100 = 500 ساعة، تعتبر هذه المدة طويلة ولتقليلها علينا القيام بإحدى من ثلاثة:

تغيير هذا المزارع بمزارع آخر أقوى وأنشط: في هذه الحالة سنقلل الزمن بفرق بسيط، فمهما كان المزارع قوي ونشيط فلن يستطيع إنجاز العمل في أقل من 4 أو ثلاثة ساعات وبالتالي سيكون زمن الزراعة الكلي هو 300 ساعة. استخدام طرق أخرى في العمل كأن يحضر المزارع آلة تساعدته في الزراعة أو أن يستغل وقته بصورة جيدة، يحاول أن لا يتوقف كثيراً أثناء الزراعة أو غيرها من الطرق التي تقلل زمن الزراعة.

الإستعانتة بمزارعين آخرين يساعدوه في الزراعة وكلما زاد عدد المزارعين كلما قل وقت الزراعة، فإذا أحضر 9 مزارعين ليصبح العدد الكلي 10 وافتضنا أن المزارعين بنفس قوة المزارع الأول (للتبسيط)، فإن العمل سينجز في $(5 \times 10) \div 10 = 50$ 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 $((5 \times 100) \div 100)$.

1.3. تعاون الحاسوبات

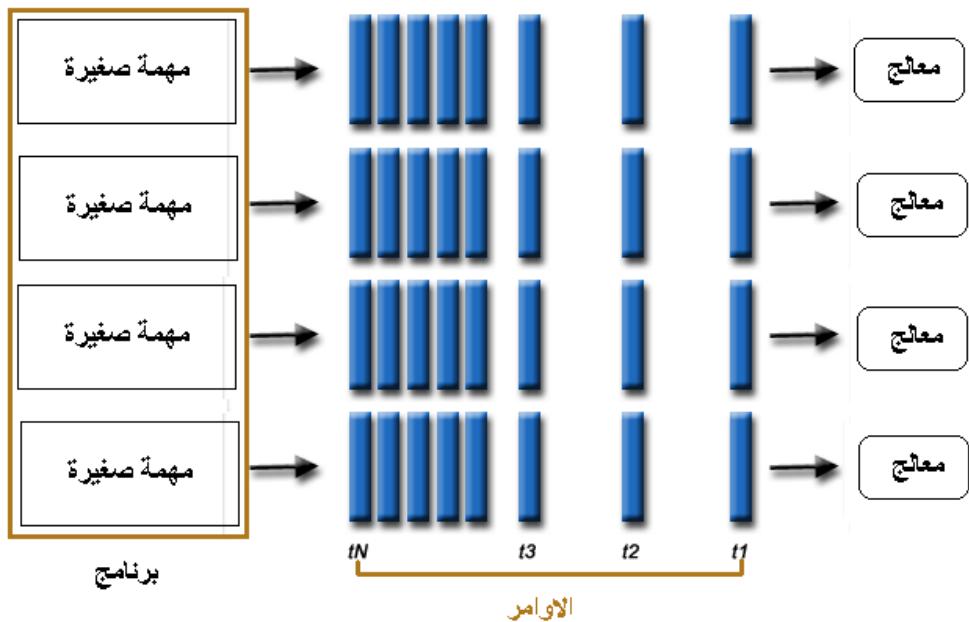
كيف ينجذب الحاسوب المهام الكبيرة؟

- تغيير المعالج بأخر أسرع.
- استخدام طرق ذكية لتقليل زمن المعالجة مثل:
 - الأنابيب الانسياحية (pipeline).
 - جدولة المعالج بواسطة نظام التشغيل (استغلال زمن المعالج بقدر ما يمكن).
- تقسيم المهمة على أكثر من معالج (نواة)، شكل 1-3.

المعالجة المتوازية هي تطبيق لفكرة الزراعة أعلاه ، فإذا استبدلنا الزراعة بمسألة حاسوبية كبيرة تحتاج وقت طويل للتنفيذ واستعاضنا عن المزارعين بمحاسبات، فستتعاون هذه الحاسوبات لحل هذه المسألة بالتوازي، حيث سيتم:

- تقسيم المسألة إلى أجزاء صغيرة.
- يوزع كل جزء على جهاز لتنفيذها.
- استلام النتائج بعد التنفيذ من الأجهزة بواسطة حاسب منسق.

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



شكل رقم 1-3: التنفيذ المتوازي [2].

فمثلا لو كان لدينا مهمة واحدة كبيرة وقسمناها على m مهمة صغيرة، وكل مهمة تحتاج t زمن للتنفيذ، فالזמן 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:

$$\text{Max } (t_1, t_2, \dots, t_m)$$

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

$$\text{round } (m/n) * t$$

توزيع المهام بالتساوي بين المعالجات يسمى ”load balancing“ . زمن تنفيذ أي مهمة (t) هو زمن تنفيذ المهمة بالمعالج مضافا إليه زمن الاتصال (delay time).

1.4. 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 بطيئتها 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 حاسب فائق السرعة من حاسيبات متفرقة هو بديل قليل التكلفة للحاسبات السوبر والتي يكلف شراءها مبالغ طائلة. فالحاسبات السوبر مكلفة وقد تصل اسعارها إلى ملايين الدولارات.
- البرامج الكبيرة والتي تحتاج ذاكرة كبيرة يمكنها الاستفادة من ذاكرة الحاسيبات الموجودة بالشبكة.
- الإعتمادية (reliability)، فوجود أكثر من جهاز يضمن سير العمل ولو تعطلت بعض الأجهزة..
- الموثوقة (يمكن تنفيذ البرنامج في أكثر من جهاز ومقارنة النتائج للتأكد من صحتها).

1.5. أسباب الإتجاه نحو التعديدية

ما أدى إلى الإتجاه إلى الحاسيبات المتعددة هو التطور السريع في المكونات المادية والشبكات. فقد تحولت الحاسيبات من الحاسيبات المركزية الغالية إلى حسابات شخصية رخيصة، وتحولت الشبكات من بطيئة وكفالة إلى شبكات فائدة السرعة وغير مكلفة.

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

ظهور الشبكات التي تربط الحاسيبات الشخصية فيما بينها جعل من السهل التشارك في الموارد من عتاد وبرامج. هنا توسيع الإنسان في طموحه وبدأ يصمم ويستخدم تطبيقات تحتاج قوة معالجة عالية (إمكانات أعلى من طاقة الحاسيبات الشخصية) ، فكان الحل هو أحد خيارات:

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

1.6. 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 الانترنت وخدمات الاعمال الالكترونية .
- التشخيص بمساعدة الكمبيوتر في الطب .
- الرسوميات المتقدمة والواقع الافتراضي وبخاصة في صناعة الترفيه .
- تقنيات الفيديو عبر الشبكات والوسائل المتعددة .
- المسائل التي تعتمد على الحسابات ومعالجة بيانات ضخمة.
- معالجة الصور والإشارات (Image and Signal Processing).
- تنقيب البيانات (Data Mining).
- الجينوم البشري (Human Genome).

1.7. لماذا ستصبح زيادة سرعة المعالج مستحلاً

وضع مور فرضية تقول بأن المعالجات تتضاعف سرعتها كل 18 شهراً، هذه الفرضية أثبتت صحتها لقرابة ثلاثة عاماً. فإذا كنت تعتقد أن كتابة برامج على حاسبات متعددة صعبة وأنك ستنتظر حتى يصل الحاسوب الشخصي للسرعة التي تريده (حسب قانون مور) فهذا لن يحصل للآتي:

- يستخدم مصنفو 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.
- لذلك الحصول على سرعة عالية من المعالجات أصبح أكثر صعوبة وتعقيداً.
- سرعة المفاتيح (switching speeds) تتصل نهايتها وتتغير حجم الدوائر الإلكترونية محدود بسرعة الضوء. ستصل قريباً إلى نقطة يصبح فيها زيادة كفاءة الحاسوب الشخصي مستحيلاً (performance limit). وستصبح الحاسبات 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 (سلسلة) به. ونلاحظ من الشكل أعلاه أن سرعة المعالج تزيد بنسبة 66% سنوياً، بينما تزيد النسبة للذاكرة بـ 7% فقط بالسنة. تعتبر الذاكرة مؤثراً رئيسياً على سرعة الحاسوب.
 - ستفقد جزءاً كبيراً 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.
- مثلاً إذا كنت تمتلك حاسباً بسرعة 1 GHz، فهذا يعني أن في كل نانو ثانية سيقوم المعالج بعمل بدورة (cycle) واحدة (أو بمعنى آخر بليون دورة clock cycle) في الثانية الواحدة. الضوء يستطيع عبور 30 سنتيمتراً (حوالى قدم) في النانو ثانية (nanosecond) أو في 1 من البليون من الثانية. وللمعلوم أن الاشارات الرقمية تنتقل بسرعة الضوء. لذلك الدارات الالكترونية التي تنتج سرعة المعالج (clock speeds) من الأفضل ألا يزيد حجمها (على الأقل عشرها) عن 30 سنتيمتراً. هذا يعني أن أكبر حجم للدارة الالكترونية يجب ألا يتعدى 3 سنتيمتر.

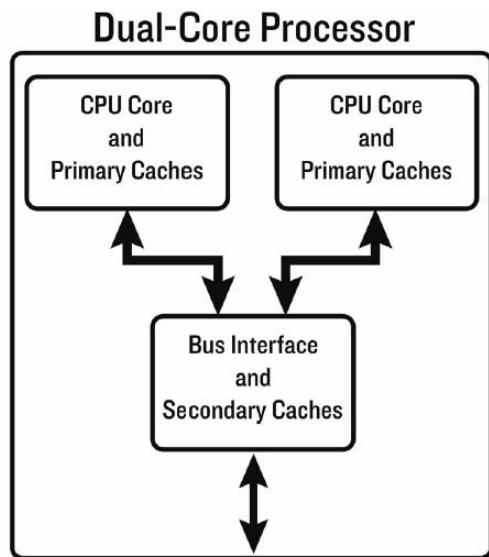
إذا أردنا تصميم معالج بسرعة 100GHz، فهذا يعني أنه لابد لهذا المعالج من إنتاج الدورة الواحدة (clock) في كل 1 من المائة من النانو ثانية. هنا يعني أن الإشارة تستطيع عبور 3 ملمتر في النانو ثانية (سرعة الضوء). لذلك من الأفضل أن يكون حجم قلب المعالج 0.3 ملمتر. وقد يكون من العسير جداً صناعة معالج بهذا الحجم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

الفقرة أعلاه مثار جدل ونقاش كثير، يمكن الاطلاع على الملحق الذي يشير إلى بعض المصادر التي تتحدث عن هذا الموضوع، من أراد الزيادة.

هذا الكلام يدل على 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 الحاسب ذي المعالج الواحد مستحيلاً.

يعنى آخر قد تصل سرعة المعالج الواحد إلى خطتها (هل هذا أكيد ؟ لا أجرم (انظر الملحق الخاص بهذا الموضوع)), ولن نستطيع زيادة الميجاهايرتز (MHz).

الواضح الآن أن سوق الحاسوب بدأ يتجه نحو التعددية، فقد بدأت حالياً الكثير من الشركات المنتجة للأجهزة تتجه نحو زيادة عدد المعالجات (أو زيادة النواة في المعالج الواحد) في الحاسوب بدلاً من زيادة سرعة المعالج، فظهر ما يسمى المعالجات متعددة النواة (multi-core CPU)، شكل رقم (4-1).



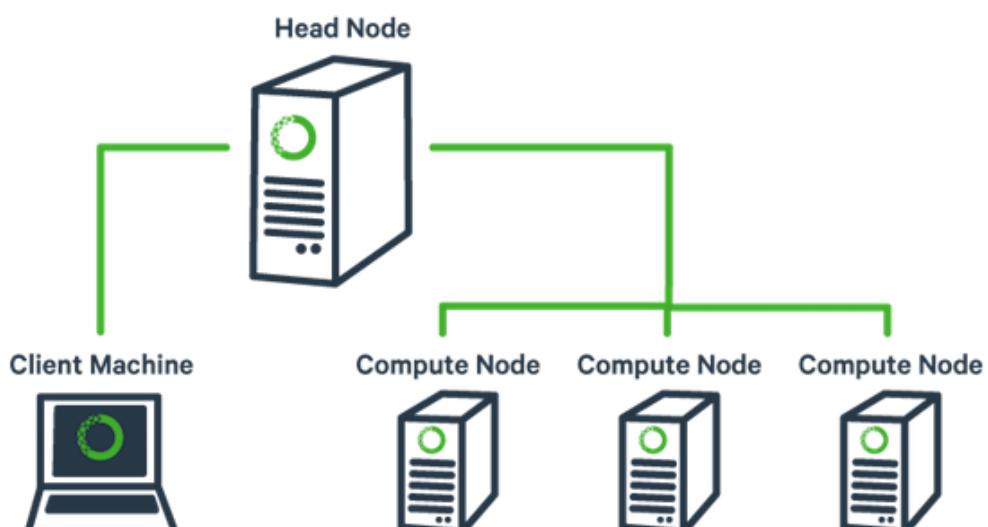
شكل رقم (4-1): معالج ثباتي النواة [4].

سرعة الضوء هي الحد الفاصل لزيادة سرعة الحاسوب. افترض أننا نريد بناء حاسوب (سلسلة) بذاكرة حجمها 1TB وسرعة 1 TFlop، إذا كانت المسافة بين الذاكرة الرئيسية والمعالج هي d ، فعلى البيانات قطع هذه المسافة لتصل إلى المعالج، ولابد من قطع هذه المسافة بسرعة 10^{12} مرة في الثانية بسرعة الضوء ($c=3e8 \text{ m/s}$)، إذن فيجب أن تكون: $d <= c/10^{12} = 3 \text{ mm}$. إذن على هذا الحاسوب أن يوضع في صندوق حجمه 0.3 ملم أيضاً لبناء ذاكرة بحجم 1TB، سنحتاج إلى شبكة (10^6 في 10^6) من الكلمات (words) إذا كان حجم هذه الشبكة هو 3. ملم \times 3. ملم، إذن فالكلمة الواحدة يجب أن يكون حجمها 3 أنجوسترام \times 3 أنجوسترام (Angstroms) أو حجم ذرة (atom) صغيرة. من الصعب تخيل أين سنضع الأسلاك [5].

1.8. تعريفات ظهرت في عصر التعددية

التجمع (Cluster)

حاسوب متتجانسة (متتشابهة في المكونات المادية ونظم التشغيل)، ومتلائمة (في مكان واحد) ومرتبطة بشبكة محلية عالية السرعة، تدار بواسطة جهاز رئيسي يقوم بتوزيع المهام على المخطبات واستلام النتائج منها، غالباً ما تستخدم التجمع عبارة، شكل 1-5.



شكل رقم 1-5: تجمع [6].

الحوسبة الشبكية (Grid Computing)

موارد موزعة في مناطق مختلفة وبعيدة عن بعضها ويمكن استخدامها كحاسِب واحد إفتراضي كبير، وهي أشبه بذلك للتجمع (cluster) ولكنها تختلف في كون أجزاءها موزعة جغرافيا وقد تكون متباينة في نوعية الأجهزة ونظم التشغيل والبرامج. ويمكن أن تحتوي على تجمعات داخلها. يستطيع المستخدم الدخول على الحوسبة الشبكية من بعيد عبر الانترنت مثلًا والاستفادة منها في تنفيذ المهام الكبيرة. يمكن أن تكون الحوسبة الشبكية تربط معامل مختلفة بشبكة مدنية أو واسعة ويمكن أن تمتد عبر مدن ودول وقارات.

تعتبر الحوسبة الشبكية نوع من أنواع النظم الموزعة وهي بذلك تختلف عن التجمع في كون محطاتها متباينة ومتباعدة ويعتبر أن تنفذ مهام وتطبيقات متباينة، شكل 1-6.

عرف Buuya الحوسبة الشبكية على أنها نوع من النظم الموزعة والمتوازية، تسمح بتبادل و اختيار وتخمير الموارد موزعة جغرافيا، بشكل ديناميكي في وقت التشغيل اعتماداً على التوفير والقدرة والأداء والتكلفة ومتطلبات جودة خدمة المستخدمين [7].



شكل رقم 1-6: الحوسبة الشبكية تجعل أجهزة متباينة ومتباعدة جغرافياً كأنها جهاز واحد [8].

الحوسبة السحابية (Cloud Computing)

موارد مختلفة متاحة عبر الإنترنت، مملوكة لجهات معينة تقوم بتأجيرها للناس أو إتاحتها لهم مجاني. يمكن الإستفادة منها في مجالات مختلفة مثل تأجير معلم لفترة معينة، أو تأجير عدد من مساحات تخزينية فيما يسمى التخزين السحابي أو استئجار برامج لإنجاز لفترة محدودة.

1.9. الخلاصة

ذكرنا في مقدمة حديثنا أن حل المسائل الكبيرة يتم بإحدى ثلاث طرق هي:

- بذل جهد أكبر ونشاط أكثر (إستبدال المعالج بأخر أسرع).
- نستخدم طرق ذكية لتقليل وقت المهمة (الأنابيب الإنسانية وغيرها).
- الإستعانة بآخرين لمساعدتك (استخدام أكثر من معالج للحل).

وأثبتنا في هذا الباب أن أسرع هذه الطرق هو استخدام أكثر من معالج للتنفيذ للأسباب التالية:

- توقف إنتاج المعالجات ذات المعالج الواحد (لن يكفي المعالج الواحد مهما بلغ من السرعة).
- الطرق الذكية تزيد سرعة المعالجة إلى حد أقصاه مائة ضعف تقريباً (زيادة ليست بالمؤثرة).
- إذن الحل هو استخدام أكثر من معالج لإنجاز المهام.

الباب الثاني

أنواع الحاسوبات

ستتحدث في هذا الباب عن أنواع الحاسوبات المتعددة والتي تحتاج إلى أدوات وبرمجة خاصة بها. تنقسم الحاسوبات المتعددة نوعين:

- الحواسيب ذات الذاكرة المشتركة (shared memory): تشمل الحاسوبات متعددة المعالجات (multi-core) والحواسيب متعددة النواة (multiprocessors).
- الحواسيب ذات الذاكرة الموزعة (distributed memory): قد تكون متجانسة أو متباعدة. المتتجانسة مثل التوازي الكثيف للمعالجات (MPPs) ومنها الحاسوبات العملاقة (super computer)، وتجمع المخطبات (COWs) ومنها التجمعات (clusters). أما الحاسوبات المتباعدة فمنها الشبكية (Grid).

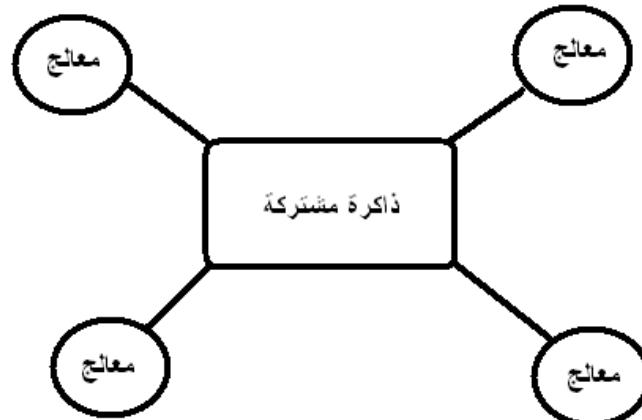
لكل واحد من هذه الأنواع صفاته الخاصة. معرفة الفروقات بين هذه الحاسوبات يمكننا من التعرف على أساليب البرمجة المناسبة لكل نوع.

2.1. الأنظمة المتعددة

الأنظمة المتعددة قسمان:

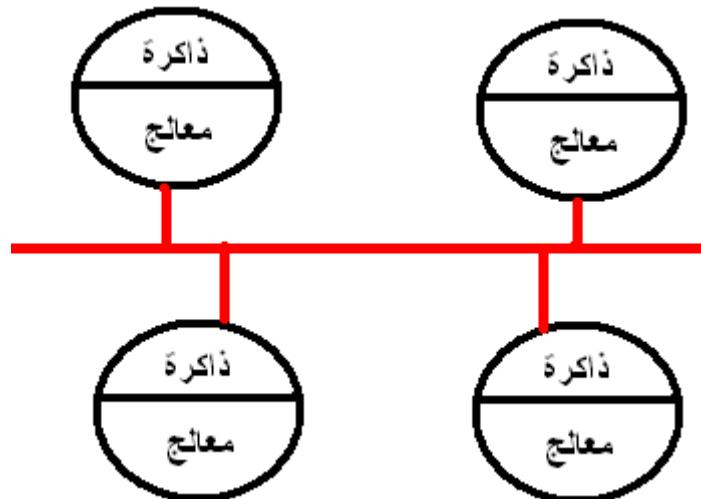
- أنظمة ذات ذاكرة مشتركة وتسمى الأنظمة متعددة المعالجات (multiprocessors).
- أنظمة موزعة الذاكرة وتسمى الأنظمة متعددة الحاسوبات (multicomputers).

الفرق الرئيسي بين الاثنين هو أنه في النوع الأول هنالك عنوان حقيقى (فيزيائى) واحد تشارک 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)

أما في النظام الثاني فلكل جهاز ذاكرته الخاصة، أي إذا كتب معالج ما قيمة في العنوان رقم 2020، فلن يحصل معالج آخر بالنظام على هذه القيمة إذا نظر في 2020. ذلك لأن العنوان 2020 للمعالج الأول هو بذاكرة المعالج الأول بينما العنوان 2020 للمعالج الثاني هو في ذاكرة المعالج الثاني.



شكل رقم (2-2): نظم ذات ذاكرة موزعة (distributed memory).

قد تتصل أي من النوعين أعلاه بإحدى الطرق التالية:

1. عن طريق مفتاح (switch)

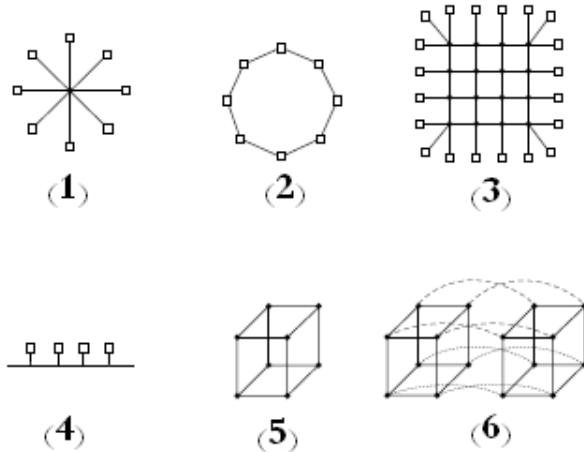
2. في شكل حلقة (ring)

3. في شكل شبكي (grid)

4. في شكل ناقل (bus)

5. مكعب (cube)

6. مكعب مركب (hypercube)



شكل رقم (2-3): أشكال ربط مختلفة [9].

2.2. الأنظمة متعددة المعالجات (multiprocessor)

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

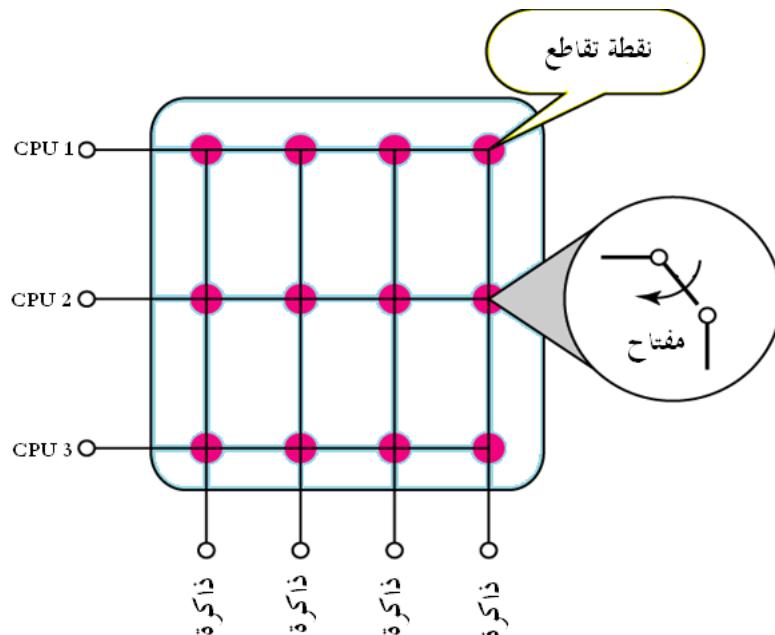
قد ترتبط هذه المعالجات عن طريق ناقل مشترك (bus)، وترتبط على نفس الناقل الذاكرة المشتركة. الناقل المشترك قد يكون شبكة أو ناقل في لوحة رئيسية (motherboard) ترکب فيها كل المعالجات والذاكرة المشتركة. وبما أن الذاكرة المشتركة واحدة، فإن كل معالج يضع قيمة في موقع ما بالذاكرة ، تستطيع بقية المعالجات الوصول لتلك القيمة والتعامل معها من قراءة أو تعديل أو غيرها.

الذاكرة التي تمتلك هذه الخاصية نسميتها (coherent). المشكلة الرئيسية في هذا النوع هو أن زيادة عدد المعالجات يخلق اختلافاً 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 يتطلب 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 قللت كثيراً من الرحام الذي كان موجوداً بالناقل الرئيسي. ولكنها خلقت مشكلة أخرى هي مشكلة عدم توافقية البيانات (consistency problem).

افتراض أن معالجين قاماً بنسخ قيمة من الذاكرة الرئيسية إلى ذاكرتيهما المخباة، ثم قام المعالج الأول بتعديل هذه القيمة بالذاكرة الرئيسية، عند ما يتطلب المعالج الثاني هذه القيمة سيحصل عليها من ذاكرته المخبأة (لنأخذ 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 التوسيع (ليس به خاصية قابلية التوسيع scalability) حتى مع استخدام الذاكرة المخبأة ، فإذا أردنا ببناء نظام متعدد المعالجات لأكثر من 256 معالج فلابد من طريقة مختلفة لربط المعالجات مع الذاكرة الرئيسية المشتركة، لأن الخط الناقل المشترك سيزدحم ببيانات المعالجات وكثيراً ما تسبب اختناقًا فيه.

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

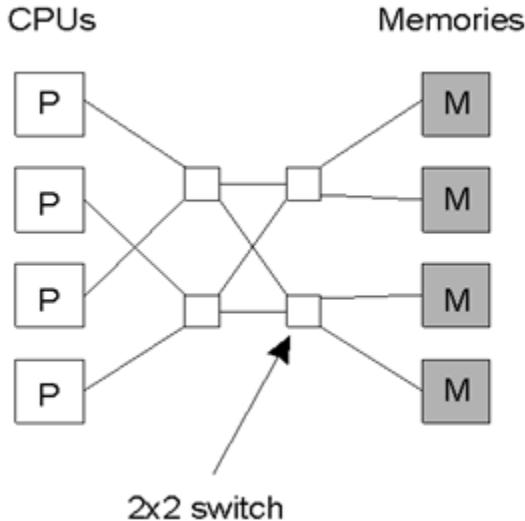


شكل رقم (4-2): مفاتيح مقاطع لتوصيل المعالجات مع الذواكر.

كل مفتاح مقاطع يمكن فتحه وغلقه 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، و لا يتضرر معالج آخر للوصول لذاكرة معينة كما في الطريقة الأولى إلا في حالة طلب معالجين الوصول لذاكرة معينة في نفس الوقت، في هذه الحالة على معالج أن يتضرر الآخر حتى يفرغ من استخدام هذه الذاكرة.

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



شكل رقم (5-2): مفاتيح مقاطعات متعددة المستويات [9].

2.3. الأنظمة متعددة الحاسوبات (Multicomputer)

هو ربط أجهزة حاسوب مع بعضها البعض عن طريق شبكة عالية السرعة. وقد تكون الحاسوبات المرتبطة مع بعضها متاشحة في المكونات المادية ونظم التشغيل. وتسمى متاجنسة (homogenous) أو قد تكون مختلفة في المكونات المادية و/أو البرامج وتسمى متباينة (heterogeneous).

2.3.1. الأنظمة متعددة الحاسوبات المتاجنسة (Homogenous Multicomputer Systems)

مقارنة ببعضها البعض، يعتبر بناء نظام متعدد الحاسوبات أسهل، حيث لكل حاسب ذاكرة محلية المرتبطة معه مباشرة. قد تكون هنالك مشكلة واحدة فقط هي كيف سيتم تبادل البيانات بين الحاسوبات.

هذه الأجهزة عادة ما تكون مستقلة عن بعضها البعض، لذلك لا بد من رابط بينها يصل بعضها البعض. هذا الرابط أو الشبكة لا بد أن تكون جيدة الأداء. قد يكون الرابط ناقلاً مشتركاً تتصل عبره كل الأجهزة مثل الإيثرنت السريع (Fast Ethernet)، الذي تصل سرعته 100 ميجابت في الثانية. استخدام الناقل (bus) السريع ولكنه غير قابل للتوسيع ، فكلما زاد عدد الحاسوبات في الشبكة قل الأداء. السبب في قلة الأداء هي طريقة عمل الناقل (bus) حيث يتم إذاعة الرسالة في الشبكة، وهذه الطريقة تجعل حاسباً واحداً فقط يستخدم الناقل في اللحظة المعينة. تخيل طوابير الحاسوبات التي تنتظر دورها للوصول إلى الناقل.

من هنا نستنتج أنه إذا كان لدينا نظام به عدد حاسوبات كبير (يفوق المائة جهاز مثلاً)، فلا بد من تغيير شكل الشبكة (network topology) من شكل الناقل (bus) إلى شكل آخر يتحمل عدداً كبيراً من الحاسوبات.

الشكل الأفضل للشبكة هنا هو النظام المبني على الرابط المفتاحي (switch). حيث بدلاً من إذاعة الرسالة يتم توجيهها (routing).

هنالك أمثلة كثيرة تعمل بنظام (switched multicomputers) منها:

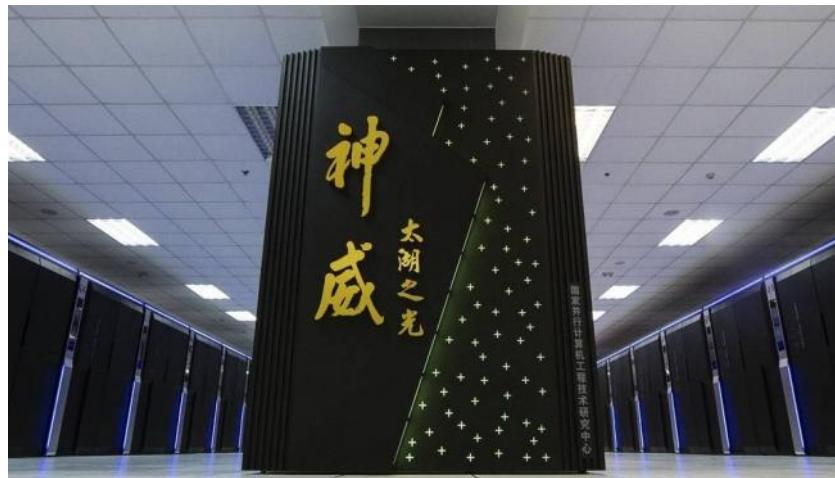
- التوازي الكثيف للمعالجات (MPPs).

- تجمع المحطات (Cluster Of Workstations (COWs)).
ادناه سننطرق لكل واحدة من النوعين أعلاه مع توضيح أمثلة لهما.

التواري الكثيف للمعالجات (MPPs) (Massively parallel processors (MPPs))
نظام حاسب يتتألف من آلاف المعالجات التي لا تختلف كثيراً عن معالجات الحواسيب الشخصية، غير أن الرابط بينها يكون عالي الجودة والتصميم، بحيث لا يكون به بطء (low latency) وأن يكون متسعاً بما فيه الكفاية (high bandwidth). سميت بالكيفية (massive) لكثره المعالجات التي يتكون منها هذا النوع، فإن لم تكن مئات فستكون آلاف، وكلمة التوازي (parallel) أنت من كون المعالجات تعمل داخل النظام بالتوازي.
يعتبر هذا النوع من النظم مناسب لتنفيذ تطبيقات تعمل بالتوازي مثل البحث في عدة قواعد بيانات، وفي تطبيقات نظم دعم القرار (decision support system)، وتطبيقات مستودعات البيانات (warehouses).

الحاسب العملاق **TaihuLight**

ينتسب الحاسب العملاق Sunway TaihuLight أسرع حاسب في العالم حسب تصنيف top500.org في يونيو 2016. سرعة هذا الحاسب هي 93 بتفلوب (petaflop/s): كواحدة مليون عملية حسابية في الثانية على مؤشر LINPACK. وهو يستخدم في حل المسائل المعقدة التي تحتاج قوة معالجة كبيرة.

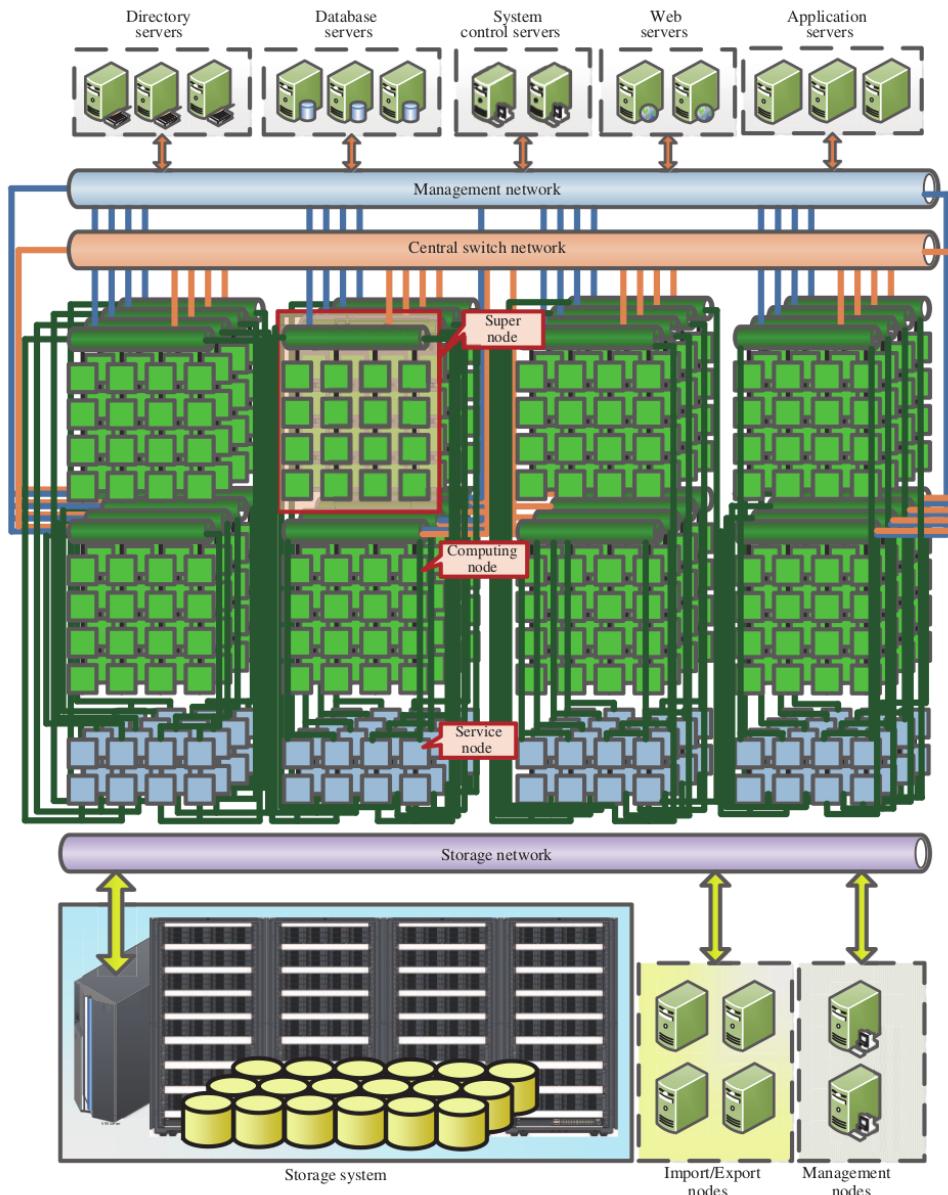


شكل رقم (2-6): الحاسب العملاق . TaihuLight

اسرع حاسب في العالم (يونيو 2016)

[/http://www.startlr.com/sunway-taihulight-the-fastest-supercomputer-in-the-world](http://www.startlr.com/sunway-taihulight-the-fastest-supercomputer-in-the-world)

يكون 40,960 مقطة (nodes)، مكوناً 10,649,600 Sunway TaihuLight (computing cores) نواة .



https://en.wikipedia.org/wiki/Sunway_TaihuLight#/media/File:General_architecture_of_the_Sunway_TaihuLight_system.png

تجمع المحطات (COWs)

هذا النوع عبارة عن حواسيب شخصية عاديّة مرتبطة بعضها البعض بشبكة سريعة مثل (Myrinet). تميّز COWs على MPPs برابط الاتصال، فأداء الرابط في النوع الأول عالي جداً مقارنة مع رابط النوع الثاني (رغم أن النوع الثاني يعتبر عالياً مقارنة مع الشبكات العاديّة مثل الإيثرنت السريع (fast Ethernet)). تعتبر COWs أقل كلفة من MPPs، لذلك يمكننا القول إن الأداء له ثمنه، فالذى لا يستطيع امتلاك MPPs يمكنه استخدام COWs، ويمكن تقليل تكلفة COWs أكثر إذا استخدمنا شبكة إيثرن特 لأنها أقل بكثير من (وكذلك أبطأ بكثير).



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

2.3.2. الأنظمة متعددة الحاسوبات المتباينة (Heterogeneous Multicomputer Systems)

هذا النوع من الأنظمة يتتألف من حاسوبات مختلفة من حيث نوع المعالج ، حجم الذاكرة، اتساع مخراج 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. تعتبر الحوسبة الشبكية (Grid Computing) مثال لهذا النوع.

تعتبر الحوسبة الشبكية نوع من أنواع الأنظمة المتعددة المتباينة، فهي قد تتكون منآلاف الحاسوبات المنتشرة في بقاع العالم والتي تمتلك بواسطة جهات مختلفة. قد تحتوي هذه الأجهزة على حاسوبات 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 الأنترنت أو عبر شبكات واسعة موفرة قوة وسعة تخزينية عالية جدا.

2.4. ملخص

تحدثنا في هذه الباب عن الحاسوبات المتعددة بأنواعها المختلفة وقد بينا أن الذي لا يستطيع شراء حاسب عملاق، فيمكنه تركيب حاسب متجانسة لإنشاء تجمع أوربط حسابات متباينة ومتباعدة جغرافيا فيما يسمى الحوسبة الشبكية.

العمليات التي تعمل على حاسوبات الذاكرة المشتركة تتبادل البيانات فيما بينها بالمشاركة في التغيرات وتستخدم الخيوط (threads) في برمجتها. بينما تتبادل البيانات في الذاكرة الموزعة يتم تبادل الرسائل (message passing) وتم برمجتها بأدوات مثل MPI أو SAGA.

تمارين

1. اذكر الفرق الجوهرى بين نوعي الاجهزة المستخدمة للمعالجة المتوازية ؟

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

3. اذكر أمثلة لأنظمة تعمل بنظام (switched multicomputers) ؟

4. اذكر أمثلة لأنظمة تعمل بنظام الناقل المشترك ؟

5. كيف تتم معالجة التباين بين الأجهزة

6. عرف تجمع المحطات (COW) ؟

7. عرف الشبكية (Grid) ؟

8. ما الفرق بين COW و Grid ؟

9. لماذا يجب أن نعرف الفروق بين أنواع الاجهزة المتعددة ؟

الباب الثالث

نماذج البرمجة المتوازية Parallel Programming Models

3.1. مقدمة

تحدثنا في الباب السابق عن أنواع الحاسوب بشقها المشتركة الذاكرة والموزعة الذاكرة، وقلنا أن كل نوع يحتاج أسلوب برمجة خاصة به. يمكن تقسيم أنواع الحاسوب من وجهة أساليب البرمجة إلى ثلاث أنواع رئيسية تختلف في طرق برمجتها هي:

- الذاكرة المشتركة (الأنظمة متعددة المعالجات (multi-core) ومتحدة النواة (multiprocessor))
- الأنظمة متعددة الحاسوب المتباينة (مثل التجمع (cluster)).
- الأنظمة متعددة الحاسوب المتباينة (مثل الشبكي (Grid)).

وفق هذا التقسيم هنالك عده نماذج برمجية تعمل مع كل نوع ولكن أحيانا قد يستخدم نموذج برمجي لنوع مع نوع آخر، فمثلا هنالك أمثلة لنظم ذاكرة مشتركة إفتراضية (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.

3.2. تصنیف فلاين

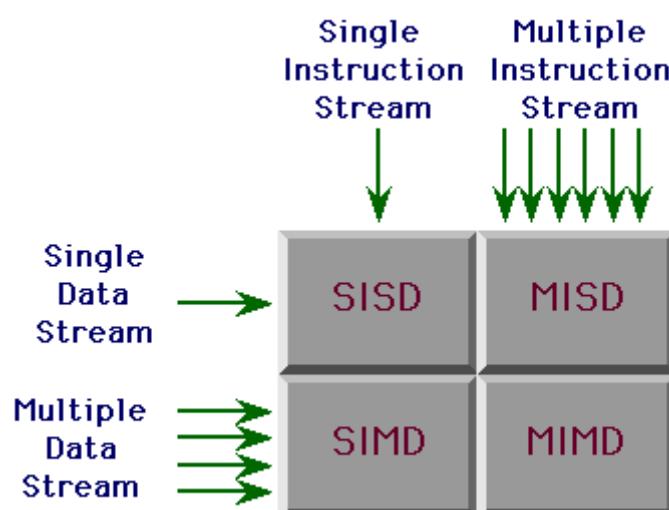
قسم فلاين انواع الحاسوب من ناحية توافي إلى اربعة أصناف حسب الشكل 1-3.

SISD: الحاسوب بمعالج عادي (به نواة واحدة)، تنفيذ تسلسلي عادي.

SIMD: اجراء نفس العمل على بيانات مختلفة (مثل جمع مجموعات مختلفة من الاعداد)، غالبا ما يكون على حاسوب متعدد النواة (multi-core).

MISD: اجراء اعمال مختلفة على نفس البيانات (نادر وجود مثل هذا النوع).

MIMD: اعمال مختلفة على بيانات مختلفة، مثل التجمع (cluster) الحوسبة الشبكية.



شكل 3-1: تصنيف فلاين.

من <http://users.cis.fiu.edu/~prabakar/cda4101/Common/notes/figs/flynn-taxonomy.gif>

3.3. نماذج البرمجة المتوازية

قسم Blaise Barney في [2]، البرمجة المتوازية إلى النماذج مختلفة بما فيها تصنيفات فلاين الاربعة. وهي كما يلي:

1. نموذج الذاكرة المشتركة.
2. نموذج الخيوط (Threads).
3. نموذج الذاكرة الموزعة (تبادل الرسائل (message passing)).
4. نموذج توازي البيانات (data parallelism).
5. النموذج الهجين (hybrid).
6. برنامج واحد على بيانات متعددة (SIMD)
7. برامج متعددة على بيانات متعددة (MIMD)

هناك تشابه وتدخل بين هذه النماذج. ويفضل تصنيف النماذج اما على اساس الذاكر وهي نوعين (مشتركة وموزعة). أو على اساس تصنيف فلاين.

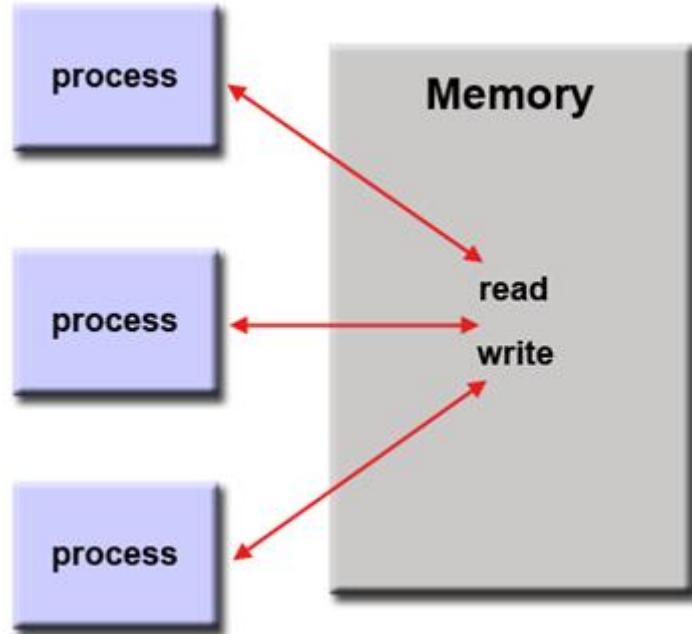
فيما يلي نلقي نظرة عامة لكل نموذج من النماذج أعلاه.

3.3.1. نموذج الذاكرة المشتركة (Shared Memory Model)

في نموذج برمجة الذاكرة المشتركة تشارك كل المهام مساحة عنونة مشتركة (common address space)، حيث تتم عمليات القراءة والكتابة بطريقة غير متزامنه (asynchronously). هذا الوصول غير المتزامن قد يسبب بعض المشاكل لذلك وضعت آليات مختلفه مثل الغلق او السيمافور (locks / semaphores) للتحكم في الوصول الى الذاكرة المشتركة.

ميزة هذا النموذج من وجهة نظر المبرمج هي عدم الحاجة لاتصال بين المهام لتبادل البيانات. وبالتالي يمكن أن تكون برمجته بسيطة. ولكن عييه الاساسي بالنسبة للاداء 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. والإزدحام الذي قد يحدث في الناقل المشترك المؤدي للذاكرة المشتركة عندما تحتاج عدة معالجات نفس البيانات في نفس الوقت. 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 حقيقة (actual memory addresses) وتكون عامة (global).



شكل رقم 3-1: الذاكرة المشتركة [2].

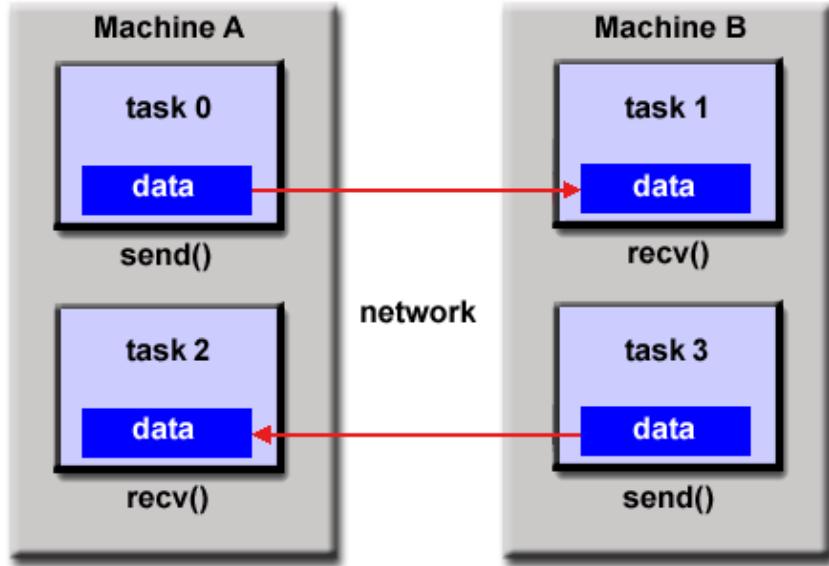
3.3.2. نموذج الخيوط (Threads model)

في نموذج الخيوط يمكن أن تحتوي العملية الواحدة على عدة خيوط (عمليات صغيرة) تنفذ بطريقة متزامنة (concurrent). غالباً ما تستخدم الخيوط في برمج المعالجات متعددة النواة (Multi-core).

3.3.3. نموذج تبادل الرسائل (Message Passing Model)

يتم تحديده بالخصائص التالية :

1. مجموعة المهام تستخدم الذاكرة المحلية الخاصة بها أثناء التنفيذ.
2. يمكن أن تتوارد عدة مهام في نفس الحاسوب أو في حاسوبات مختلفة.
3. يتم تبادل البيانات عبر وسيلة اتصال وذلك بارسال واستقبال الرسائل .
4. يتطلب نقل البيانات تعاون بين المهام، فمثلاً عملية الارسال من قبل مهمة لا تتم ما لم تقابلها عملية استلام من مهمة أخرى.



شكل رقم 3-2: نموذج تبادل الرسائل [2].

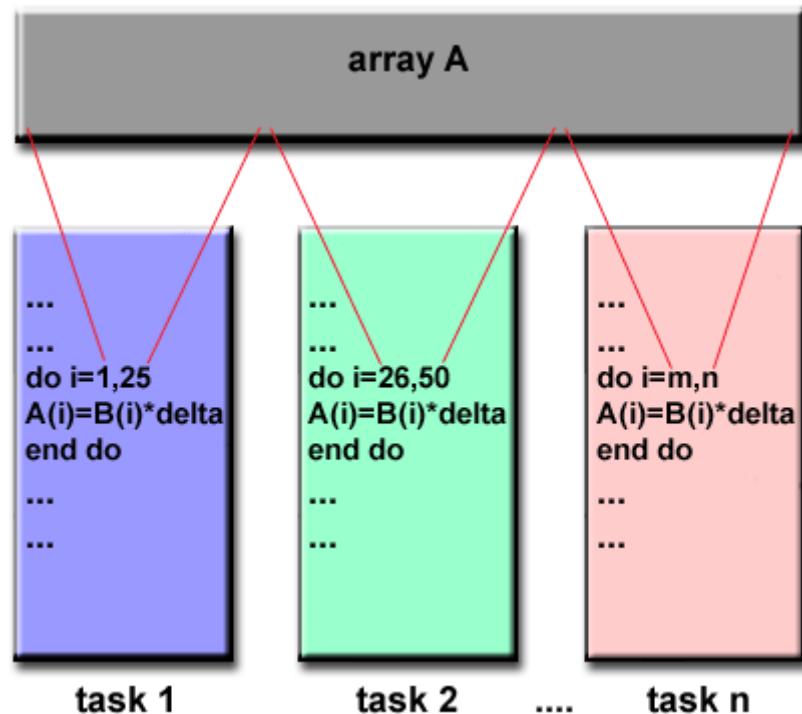
من وجه نظر برمجيه تطبيق تبادل الرسائل يحتاج مكتبة روتين فرعية يجب تضمينها في شفرة البرنامج. وعلى المبرمج تحديد الموازاة (parallelism).

تاريجيا ، توفر 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 الاهداف الاساسية 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 انتشرت انتشار واسع.

3.3.4. نموذج توازي البيانات (Data Parallel Model)

معظم العمل المتوازي يركز على تنفيذ عمليات على مجموعة بيانات مثل المصفوفات أو المكعبات. هنالك مجموعة من المهام تعمل بشكل جماعي على نفس بنية البيانات (data structure) لكن كل مهمة تعمل على جزء من هذه البنية. وتنفذ المهام نفس العمل لكن على الجزء التي تعمل فيه مثلا إضافة رقم لكل عنصر من عناصر المصفوفة.

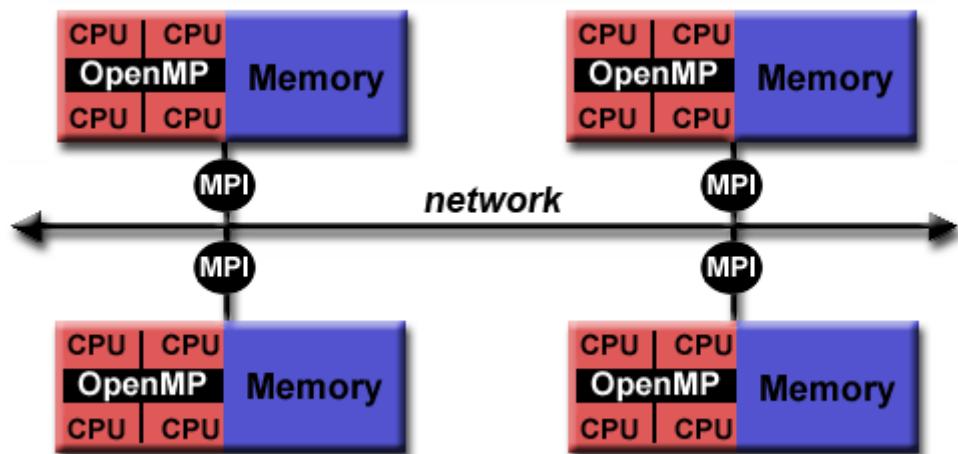
بنية البيانات قد توجد في مكان واحد بالذاكرة في نظام الذاكرة المشتركة، ولكن في نظام الذاكرة الموزعة فيجب وضع كل جزء من البنية في الذاكرة الخاصة التي تنفذ فيها المهمة بها (تقسيم بنية البيانات وتوزيعها على ذواكر المهام). البرمجة في توازي البيانات يتم بكتابة برنامج مع بنيات (مشيدات) توازي بيانات (constructs).



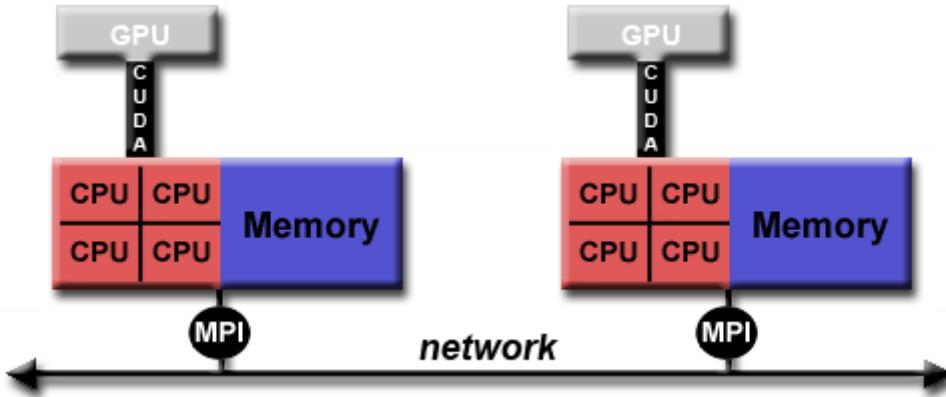
شكل رقم (3-3): توازي البيانات [2].

3.3.5. الهجين

يمكن استخدام اثنين أو أكثر من النماذج السابقة. مثلاً إذا كان لدينا تجمع مكون من حاسوب ذات معالج متعدد النواة فهنا سيكون التعامل بين محطات التجمع بتبادل الرسائل، ولكن التعامل داخل المخطة الواحدة (متعددة النواة) يتم بذاكرة مشتركة. لذلك يمكن في مثل هذا النظام استخدام غووج هجين بين MPI و OpenMP أو POSIX threads. هذا الهجين أيضاً يتعبر مناسب لـ تعدد المعالجات (multiprocessor).



شكل رقم 3-4 (أ): النموذج الهجين [2].



شكل رقم 3-4 (ب): النموذج المجين [2].

3.3.6. برنامج واحد على بيانات متعددة

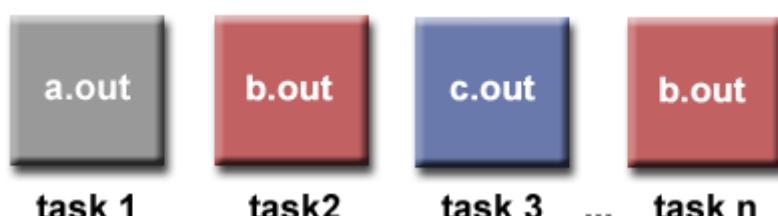
مهمة واحدة تنفذ على عدد من البيانات، مثل استخراج الاعداد الأولية من مجموعات مختلفة من الارقام (مثلاً استخراج الاعداد الأولية بين 1 و مليون، استخراج الاعداد الأولية بين مليون واثنين مليون،....). بحيث يمكن لكل معالج القيام بنفس العمل على بيانات مختلفة.



شكل رقم 3-5: برنامج واحد على بيانات متعددة [2].

3.3.7. برامج متعددة على بيانات مختلفة

برامج مختلفة تعمل في وقت واحد على بيانات مختلفة.



شكل رقم 3-6: برامج مختلفة على بيانات مختلفة [2].

3.4. الخلاصة

تحدثنا في هذا الباب عن أنواع نماذج البرمجة التي يمكن استخدامها في برمجة الأنظمة المتعددة. وبيننا أن هذه النماذج ليس بالضرورة أن تكون حكر على نظام معين وإنما يمكن استخدامها مع كل الأنواع. وكذلك يمكن استخدام أكثر من نموذج في نظام واحد.

ستكون في الأبواب القادمة بإذن الله تفاصيل أوفى عن هذه النماذج وستكون هنالك أبواباً مخصصة لبعض النماذج
نوضح فيها أساليب برمجة هذه النماذج وكيفية تصميم برامج من خلالها.

الباب الرابع مفهوم الخيط

Thread Concept

4.1. تمهيد

إذا لديك دراية بمفهوم الخيوط، يمكنك تجاوز هي الفقرة التمهيدية والانتقال إلى 4.2.

دعنا نضرب مثل تشبيهي يوضح فكرة الخيوط قبل الدخول في تعريفها ومفهومها واستخدامها. لتنظر إلى المسألة الحسابية البسيطة التالية:

$$95 = 14 + 12 + 13 + 7 + 15 + 7 + 19 + 8$$

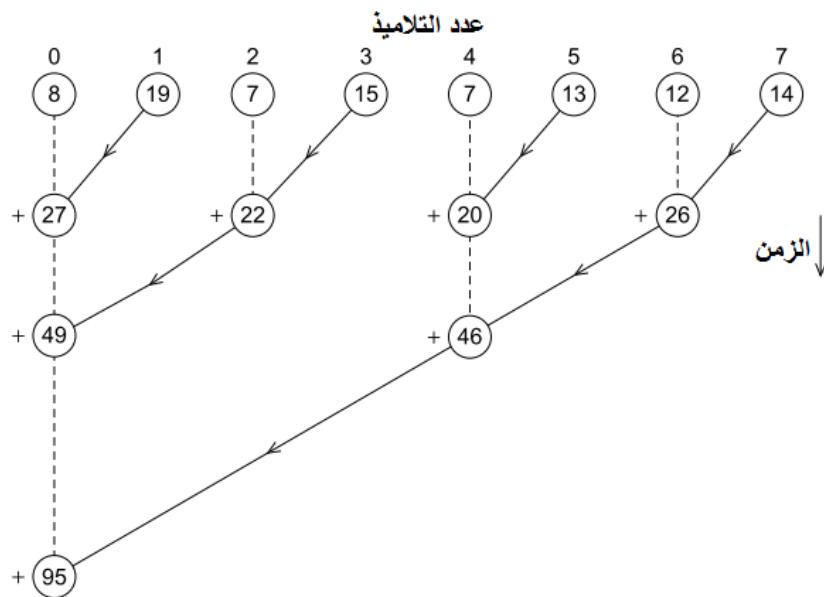
فيما إذا طلبنا من أي تلميذ حل هذه المسألة، فإنه سيحلها غالباً في سبع خطوات هي:

1. جمع $19 + 8$
2. جمع الناتج إلى 7
3. جمع الناتج إلى 15
4. جمع الناتج إلى 7
5. جمع الناتج إلى 13
6. جمع الناتج إلى 12
7. جمع الناتج إلى 14، والحصول على النتيجة النهائية، الشكل 1-4.

$$\begin{array}{r} (8) \\ + \\ (6) \\ + \\ (7) \\ + \\ (15) \\ + \\ (7) \\ + \\ (13) \\ + \\ (12) \\ + \\ (14) \\ \hline 95 \end{array}$$

شكل رقم 4-1: عملية حسابية تنفذ تسلسلياً بواسطة تلنيد واحد.

ولكن لو طلبت من ثمانية تلاميذ القيام بذلك بالتوالي في وقت واحد، فإن العملية ستتجدد في خطوات أقل كما مبين في الشكل 4-2.



شكل رقم 4-2: ثمانية تلاميذ يتعاونون في عملية حسابية [10].

4.1.1. العملية ذات الخيط الواحد (التوازي)

نلاحظ في الشكل 4-2، أن الخطوات تتم بالتوال و هنا يشبه العمليات ذات الخيوط المتعددة (كل تلميذ يقابل خيط). أما في الشكل 4-1، فإن الخطوات تتم بالتوازي، ولا يستطيع التلميذ القيام بخطوتين في وقت واحد، وهذا يشبه العملية العادية (ذات الخيط الواحد).

4.1.2. العملية متعددة الخيوط (التوازي)

في مثال التلاميذ الثمانية يقوم كل تلميذ بعملية حسابية منفصلة عن ما يقوم به زملاؤه، في نفس الوقت الذي يقوم به زملاؤه بعمليات حسابية أخرى منفصلة عن بعض البعض. من هنا نستنتج:

- أن القيام بأكثر من عمل في وقت واحد يعني عملية لها أكثر من خيط.
- لا يمكن للخيوط أن تعمل بالتوازي إذا كانت تعتمد على بعضها، فلا بد من أن تكون مستقلة عن بعضها لتعمل معاً في وقت واحد، وإنما نستفيد من وجود أكثر من خيط في العملية.

4.2. تعريف الخيط (thread)

الخيط هو تقنية تسمح لعملية واحدة أن تقوم بعدة مهام في وقت واحد (concurrently). أي أن العملية لن تحتاج أن تنتهي من عمل حتى تبدأ عمل آخر.

العملية (process) في وضعها الطبيعي تنفذ عمل واحد، ونطلق عليها عملية أحادية الخيط (single thread). ولكن ماذا لو أردنا من العملية أن تنفذ أكثر من عمل في نفس الوقت، هنا لابد من أن نجعل العملية متعددة الخيوط (multiple threads)، الشكل (3-4). حيث تشغّل كل الخيوط التي توجد في العملية في وقت واحد بالتوالي مما يحقق التزامنة (concurrency). تشارك خيوط العملية الواحدة في بنية معلومات وحدة (PCB)، لكن لكل خيط بنية معلوماته الخاصة به التي تختلف عن بقية بنية معلومات الخيوط الأخرى التي معه في نفس العملية. بنية معلومات الخيط تحتوي على :

- المكبس (stack).

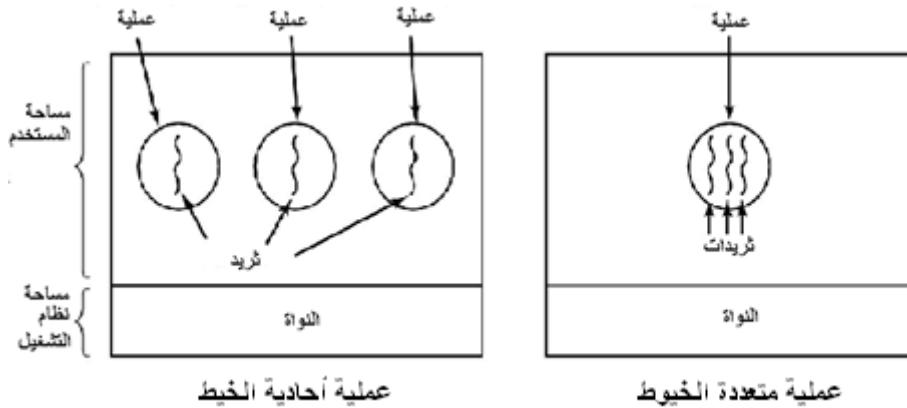
○ المسجلات (register).

○ عداد البرامج.

○ حالة الخيط.

أحياناً نطلق على الخيط عملية خفيفة (lightweight process). ذلك لأن الخيط يمتلك الكثير من خصائص العمليات.

لا ترتبط الخيوط بأي مورد لذلك فإن إنشاءها ودميرها أسهل من إنشاء ودمير العمليات. وقد يكون إنشاء الخيط في بعض الأنظمة أسرع بمئات مرة من إنشاء العملية.



شكل رقم (3-4): على اليسار: ثلاث عمليات كل واحدة تحتوي على خيط واحد.

على اليمين: عملية واحدة تحتوي على ثلاث خيوط.

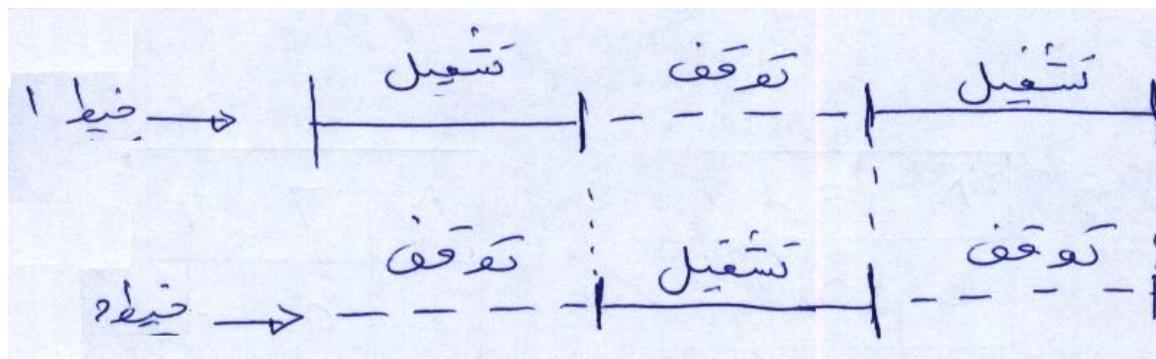
مثال تشبيهي

إذا تعاملت مع شركة ما 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 عامل واحد فأنت تعامل مع الشركة كعملية تقليدية ذات خيط واحد (النظام القديم)، فعلى العامل هنا أن ينفذ العمل بالتوالي جزء تلو الآخر، أما إذا نفذ العمل عدد من العمال فأنت تعامل مع شركة (عملية) ذات خيوط متعددة، حيث ينفذ العمال العمل بالتزامن، كل عامل ينفذ جزئية معينة في وقت واحد. والفرق واضح بين الاثنين.

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

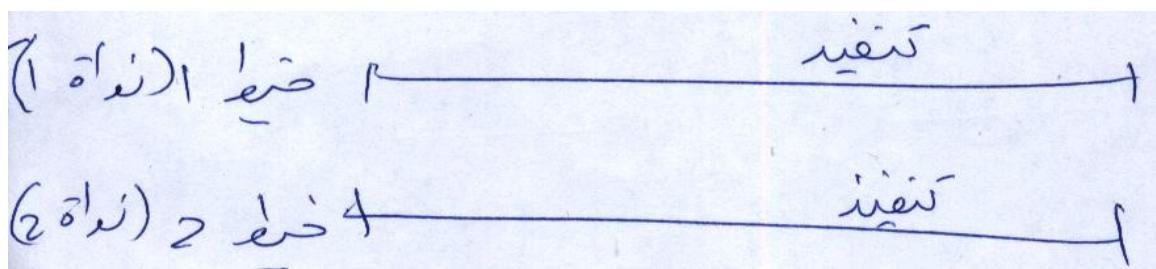
4.3. الموازاة الخيالية والموازاة الحقيقة

إذا كنت تمتلك حاسب ذو معالج واحد (ونواة واحدة)، وصممت برنامج يحتوي على خطيدين مثلاً، فلن يستطيع المعالج تنفيذ الخطيدين في نفس الوقت حقيقة، وإنما يقوم بتشغيل خيط فترة زمنية ثم توقيفه ثم تشغيل الخيط الثاني فترة زمنية ثانية، وهكذا يتبادلان في استخدام المعالج. هذه العملية تبدو للمستخدم كأنه موازاة حقيقة لسرعة التبديل بين تشغيل الخطيدين، شكل رقم 4-4.



شكل رقم 4-4: موازاة خيالية

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



شكل رقم 4-5: موازاة حقيقة

هل هناك فرق بين المعالج dual-core والمعالج core2duo ؟ وما هو؟

4.4. أنواع الخيوط

تحتلت الخيوط في طريقة الإدارة والدعم، فهناك خيوط تدار وتدعى بواسطة نظام التشغيل وهناك ما يدار بواسطة المستخدم، أدناه سنفصل الفرق بين كل نوع مع إعطاء مثال لكل واحد.

4.4.1. خيط المستخدم (user thread)

قد يتعامل نظام التشغيل (خارجيًا) مع العملية كعملية واحدة دون معرفة الخيوط الداخلية التي توجد فيها، بينما داخلياً تنفذ العملية مجموعة من الخيوط في وقت واحد. هذا النوع من الخيوط يسمى خيط المستخدم، أي أن إنشاء الخيط والتعامل معه يتم في مستوى المستخدم (user level) ولا شأن لنظام التشغيل به، فهو يتعامل مع العمليات دون التمييز بين التي بها خيط واحد والتي بها مائة خيط. يتم تطبيق خيط المستخدم بواسطة مكتبات تدعم الخيوط، مثل POSIX Threads (Pthreads)، حيث توفر هذه المكتبة كل ما يتعلق بالتعامل مع الخيط من إنشاء، وإنهاء، جدولة، وإدارة، دون أي مساعدة أو تدخل من نظام التشغيل، ودون الإرتباط بلغة برمجة معينة. عيب هذا النوع أن نظام التشغيل لا يتعامل مع الخيوط الموجودة بالعملية، لذلك عندما يحجز عملية معينة، فإنه يحجز

العملية ككل بما فيها من خيوط ولا يشترى أي خيط داخل العملية من الحجز، رغم أن بعض الخيوط قد تكون قادرة على العمل. أما ميزة فهي سرعة وسهولة إنشاء وإدارة الخيوط.

أيضاً من خيوط المستخدم OpenMP التي بنيت على موجهات المترجم (compiler directives).

4.4.2. خيط النواة

هنا يتم دعم الخيط مباشرة بواسطة نظام التشغيل فهو الذي يوفر طرق التعامل مع الخيط من إنشاء وإلغاء، جدولة وإدارة. ولأن إدارة الخيط يتم بواسطة نظام التشغيل فهذا يجعل خيط النواة يطبع نوعاً ما مقارنة بخيط المستخدم. ولكن إذا تم توقف خيط بسبب ما، فيمكن لخيط آخر في نفس العملية أن تعمل دون أن تتأثر بجزء رقيقها. من أمثلة نظم التشغيل التي تدعم خيوط النواة، نظام التشغيل ويندوز، الذي يمتلك مكتبة تدعم الخيوط هي مكتبة windows.h. حيث يمكنك استخدام CreateThread()

هناك نظم تشغيل تدعم النوعين، خيط المستخدم وخيط النواة.

4.5. التحول بين العمليات Context Switch

لكل عملية بنية بيانات تسمى (PCB)، بغض النظر أن العملية هل فيها خيط واحد أم أكثر. وكل الخيوط الموجودة في العملية الواحدة تشارك في بنية العملية (PCB)، بينما يكون لكل خيط بنية الخاصة (الشكل (6-4)) التي يستخدمها عندما يتحول من حالة التنفيذ إلى حالة أخرى. بنية معلومات الخيط تتكون من:

- المسجلات بما فيها مسجل عدد 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.

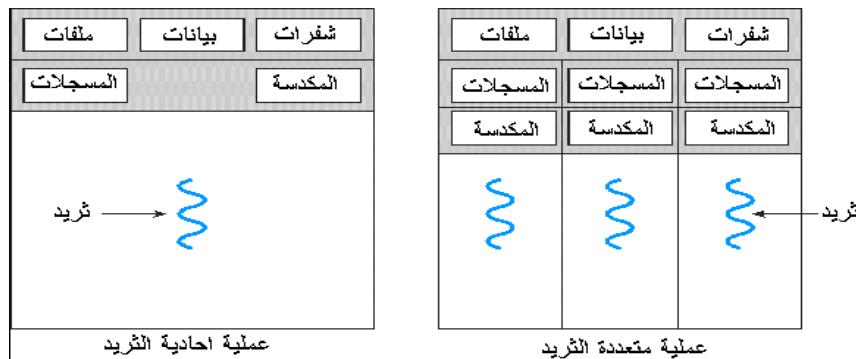
انتقال المعالج بين العمليات أو الخيوط يتم بإخراج عملية من المعالج (توقفها أو حجزها)، وإدخال عملية أخرى للمعالج للتنفيذ، هنا نقول أن المعالج انتقل من تنفيذ العملية الأولى إلى تنفيذ العملية الثانية. هذا الانتقال أو التحول يسمى (context switching) ويغير مكلف زمنياً، فلكلّي يتم الانتقال لأبد من حفظ معلومات العملية (PCB) المنفذة حالياً (المتّقل منها)، وتحميل معلومات العملية (PCB) التي ستدخل المعالج (المتّقل إليها).

ويحدث نفس الأمر في الانتقال بين الخيوط، لكن الفرق هنا أن معلومات الخيط التي ستحفظ ومعلومات الخيط الآخر التي ستتحمل 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 وقتاً 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 مشغولاً. بينما يجد للمستخدم أن المعالج يشغل عدد من العمليات معاً (تعدد المهام).



شكل رقم (4-6): بنية العملية تكون مشتركة بين الخيوط ، ولكل خيط بيته الخاصة.

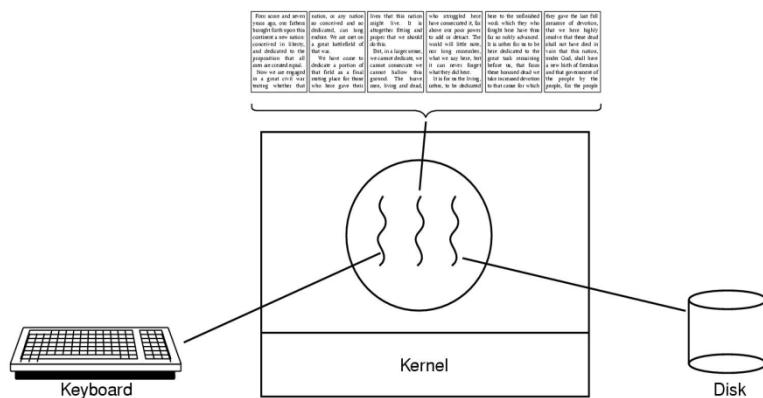
4.6. استخدامات الخيط في المعالج الواحد

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

4.6.1. محرر النصوص

الشكل رقم (4-7) يوضح مثال لعملية (برنامج محرر نصوص) يحتوي على ثلاثة خيوط ، وكل خيط ينفذ مهمة معينة، فهناك خيط يستجيب للمدخلات عبر لوحة المفاتيح والماوس وينفذ الأوامر التي ترد عبرها مثل الانتقال لصفحة معينة. بينما هناك خيط ثاني يقوم بعملية إعادة شكل الوثيقة ، فإذا قمت بحذف جملة سيقوم هذا الخيط بإعادة ترتيب النص كاملاً بعد حذف الجملة، وهناك 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 نفس الوقت ، وعندما يعمل الحفظ التلقائي لا بد من توقف كل شيء حتى يكتمل الحفظ.



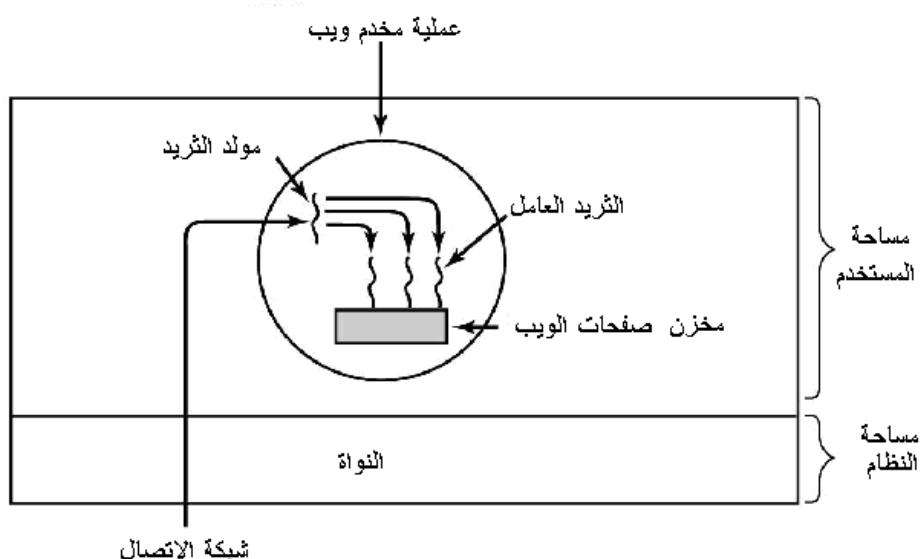
شكل رقم (4-7): برنامج محرر نصوص مكون من ثلاثة خيوط.

4.6.2. مخدم الويب

وظيفة مخدم الويب (web server) هي الرد على طلبات المتصفحين وإرسال ما يريدون من صفحات إلى أجهزتهم. يوجد في كل جهاز مخدم ويب على الإنترنت مثلاً وفيه عدد من المواقع، وقد يتصل أكثر من شخص بجهاز في وقت واحد وكل يريد الرد على ما يطلب في نفس الوقت، هنا ينشي مخدم الويب خيط لكل متصل ليرد عليه ويرسل له طلباته، شكل 4-8.

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 (بال التالي)، فإذا أفترضنا أن كل طلب يستغرق دقيقة واحدة، فإن العميل رقم مائة ستم خدمته بعد ساعة وأربعون دقيقة. هل ستستخدم الإنترنت إذا كان المخدم يعمل بهذه الطريقة؟



شكل رقم (4-8): عملية مخدم ويب تنشي خيط لكل طلب.

4.7 ملخص

تحدثنا في هذا الباب عن مفهوم الخيوط، وذكرنا أن الخيط يمكن أن يطبق داخل نظام التشغيل أو في لغة برمجة معينة أو في مستقل في مكتبة منفصلة (خيط المستخدم). ووضحنا بصورة مبسطة كيف يمكن استخدام خيط اللغة باستهدا جافا لنوضح مفهوم إنشاء وإخاء والتحكم في الخيوط من تشغيل وتوقيف وغيرها. سنتحدث عن برمجة الخيوط بصورة موسعة تشمل عملية التزامن بين الخيوط وكيفية التعامل مع بيانات مشتركة بين الخيوط ولكن لأن هذا يرتبط بمفهوم التزامن، فلا بد من باب يسبقه عن التزامن ومفهومه والمشكلات التي قد تترجم عنه.

تمرين غير محلولة

.1 عرف الخيط ؟

.2 ما الفرق بين الخيط والعملية ؟

.3 لا ترتبط الخيوط بأي مورد، ووضح ذلك ؟

.4 إنشاء الخيط وتدميرها أسهل من إنشاء العمليات لماذا ؟

.5 "إنشاء الخيط أسرع بعشرة مرات من إنشاء العملية" ، أثبتت أن هذه العبارة صحيحة (أبحث في الإنترنت عن ذلك) ؟

.6 ما الفرق بين خيط المستخدم وخيط النواة ؟

.7 هل يعتبر الخيط في جافا خيط مستخدم ؟

.8 هل تعتبر Pthreads خيط مستخدم أم نواة ؟

.9 أذكر مثال لنظام تشغيل يدعم خيط المستخدم ومثال لنظام تشغيل يدعم خيط النواة ؟

.10 ما هي مكونات بنية معلومات الخيط التي يحتاج لحفظها عند التحول من حالة التنفيذ إلى حالة أخرى؟

.11 أذكر حالات الخيط المختلفة ؟

.12 أذكر دوال الخيط في جافا والتي تستخدم في تشغيل وتوقف وإنفاس الخيط ؟

الباب الخامس التزامن (synchronization)

إذا كنت تبرمجة على حاسوب شخصي وتكتب برمج عادي، فسيكون هك الأكبر هل المخرج أو الناتج الذي يظهره البرنامج وهل هو المطلوب أم لا؟

ولكن إذا صممت برنامج يتكون من أكثر من عملية بحيث تنفذ هذه العمليات في معالجات مختلفة في نفس الوقت وتشترك في البيانات. فستظهر هموم أخرى غير هموم الناتج الصحيحة، مثل هل البيانات متواقة؟ وهل ما قراءته من معلومات هو آخر ما تم تحميله، وهل على عملية معينة الانتظار في لحظة معينة حتى يتزامن عملها مع بقية العمليات التي تشارك معها في البيانات. و ما إلى ذلك من مشاكل التي ترتبط بالتنفيذ المتوازي مع التشارك بين البيانات.

5.1. 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). مثلاً قد أقوم بتشغيل محرر النصوص (WinWord) لأكتب عليه، واستمع إلى مادة صوتية بواسطة مشغل الوسائط (media player)، ويكون المتصفح ينزل ملفات من الانترنت، هذه العمليات تنفذ بالتوازي في وقت واحد ولكنها مستقلة عن بعضها البعض. التوازي قد يطبق على مستوى العمليات (concurrent processes) أو على مستوى الخيوط (threads).

العملية المستقلة هي التي لا تؤثر ولا تتأثر بالعمليات التي تعمل معها في نفس الوقت. العملية تكون متعاونة إذا أثرت و/أو تأثرت بالعمليات الأخرى التي تعمل معها بالتوازي.

5.2. تعاون العمليات

إذا احتاجت العمليات المتوازية أن تتشارك في بعض البيانات أو أن تؤدي عملاً مشتركاً، فلا بد 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 يحتاجه 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. مثلاً إذا طلبت عملية من نظام التشغيل معلومة معينة من القرص الصلب، ستظل هذه العملية في انتظار المعلومة حتى تصاحلها من القرص إلى الذاكرة، أو إذا طلب المتصفح من مخدم الويب بالانترنت موقع معين 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 لتکمل تنفيذها دون انتظار العمليات الأخرى فقد يتسبب هذا في نتائج غير متوقعة أو غير صحيحة، لذلك لابد من وضع نقاط تزامن لا تتعادلها العمليات السريعة ما لم تصل بقية العمليات هذه النقاط.

مثلاً إذا استخدمنا الأنابيب الانسية (pipeline) بين عمليات فهذا يعني أن مخرج العملية الأولى سيكون مدخل للعملية الثانية، في هذه الحالة لا يمكن للعملية الثانية أن تسبق العملية الأولى. مثلاً الأمر التالي (في ينكش):

```
$ 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 ل تعمل عليها (تحسب عدد كلمات المخرج)).

- قد يكون من الملائم للعمليات أن تعمل مع بعضها لإنجاز العمل، مثلاً 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 جهاز الذي يصدر الصوت/الصورة. هنا تعتبر العملية الأول منتج (producer) بينما العملية الثانية مستهلك (consumer). وعلى المنتج العمل على جعل الخازن متلها دوماً بحيث لا يتوقف المستهلك عن العمل (لا يجد الخازن فارغاً)، لأن المستهلك إذا وجد الخازن فارغاً سيتوقف عن العمل ويتنظر وصول بيانات للخازن مما يتسبب في تقطيع مقطع الفيديو/الصوت أو عدم وضوحيه.

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

- تزامن بينها.

- اتصال فيما بينها.

الحجز غير المتوقع للعمليات

قد يقوم المجدول (في نظام التشغيل) بمحجز أي عملية في أي وقت ولأي سبب، هذا المحجز سيوقف العملية التي تنفذ حالياً في المعالج، ويدخل عملية أخرى مكانها (بالطبع سيكون هنالك تحول (context switching)، حيث يتم حفظ معلومات العملية الحجوزة و يتم تحميل معلومات العملية التي سيتم تنفيذها).

لا يستطيع المبرمج معرفة متى سيقوم المجدول بمحجز برنامجه (عمليته).

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

5.3. النزاع (competition)

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.

أيضاً قد تكون العمليات مستقلة لكنها تتصل مع بعضها لاستخدام موارد مشتركة، مثلاً إفترض أن لدينا عمليتان، كل عملية تزيد طباعة ملف على الطابعة (المشتراك)، لابد لها من الإتصال بينهما ليقررا من يستخدم الطابعة أولاً، وعلى العملية الثانية الانتظار حتى تفرغ الأولى من الطباعة. هذا يسمى هذا تنافس العمليات (process competition).

5.3.1. مشاكل النزاع

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

5.3.1.1. مشكلة تعديل ملف على الخادم الملفات (file server)

إفترض أن لدينا شبكة حواسيب حيث يتم تخزين كل ملفات المستخدمين في خادم الملفات. إذا كان هنالك مستخدمان يريدان تعديل ملف مشترك بينهما في نفس الوقت، سيقوم كل مستخدم بفتح الملف في جهازه (نسخة من الملف)، ثم يقوم كل منهما بتعديل نسخته، هذا التعديل لن يتم على الملف الأصلي 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 أولاً (overwrite). هذه المشكلة يتبع عنها مخرجات خطأ.

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

5.3.1.2. 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 A، وفي نفس الوقت كان هنالك عميل بالفرع B ، يريد حجز نفس المقعد، وقام الموظفان في الفرعين بعملية طلب حجز للمقعد في نفس الوقت، ماذا يحدث ؟

العملية الأولى في الفرع A قامت بالتأكد من وجود مقعد فارغ، وهذا ما فعلته أيضاً العملية التي نفذت في الفرع B، فالعمليتان قاماً باختبار وجود مقعد فارغ، ثم أحاجب النظام للعمليتين "نعم يوجد مقعد واحد فارغ" ، في هذه الإثناء قام المجدول (بنظام التشغيل) بتوقيف إحدى العمليتين لسبب ما، فقمت العملية الأخرى بمحرر المقعد، ثم بعد ذلك توقيف العملية الأولى ستكمل عملها وتنفذ الأمر الذي يلي اختبار وجود مقعد فارغ (فهي اختبرت المقعد ووجده فارغاً قبل توقيفها). فتقوم العملية بمحرر المقعد الذي حجز من قبل، وهذا يتم حجز المقعد مرتين.

5.3.1.3. مشكلة الحساب المشترك

إذا كان هنالك حساب مفتوح بينك ما، وكان هذا الحساب مشترك بين عميلين، فقد يتفق وأن يطلب العميلين 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 دولار)، ثم إذا أوقفت إحدى العمليتين بواسطة المجدول لسبب ما، ونفذت العملية الأخرى سحب المبلغ المطلوب (مثلاً 100 دولار)، سيكون الرصيد الآن 1400 دولار. بعد أن يتم فك حجز العملية الثانية ستواصل من بعد آخر أمر كانت قد نفذته، وهو اختبار الرصيد (1500 دولار)، وستنفذ السحب وتحصل المبلغ من الرصيد (مثلاً إذا كان المبلغ المراد سحبه هو 200 دولار ، فسيكون الرصيد المتبقى هو 1300 دولار). وتعتبر هذه العملية خطأ، ولو عمل البنك بهذه الطريقة سيغلق أبوابه قريباً.

النتيجة الخطأة 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.

حالة السباق (race condition) (والم منطقة الحرجة (critical section))

المشاكل التي سررناها مسبقاً معظمها تعاني من حالة السباق، ولتوسيع حالة السباق وطريقة حلها سنأخذ المثال التالي:

إذا كان لدينا متغير X، مشترك بين عميلين، 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 تفيد العمليتان أن المتغير سيزيد بواحد بدلًا من 2. الحل هو أن نمنع أي عملية أخرى من الوصول للمتغير x إذا كانت هناك عملية تستخدم هذا المتغير. المنطقة التي تعامل مع المتغير المشترك في العملية نسميتها المنطقة الحرجة (critical section). ستكون منطقة المتغير المشترك هي المنطقة الحرجة، وحل مشكلة حالة السباق يجب أن تتأكد من أن هناك عملية واحدة تنفذ داخل المنطقة الحرجة. ولا يسمح للعملية الثانية بالدخول إلى المنطقة الحرجة إلا بعد خروج العملية الأولى منها.

4.5. مشاكل التزامن الكلاسيكية

هنا سنطرق على بعض المشاكل الناجمة عن التزامن والتي تحدث في العمليات المتوازية وهي مشاكل مشهورة وتستخدم لاختبار أي نظام جديد تزامني مفترض.

4.5.1. 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. مثلاً 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 تفيد عملية 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 هناك عملية كتابة تعلم 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 تجري 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، فلا بد لعملية الكتابة من الوصول للبيانات المشتركة بصورة احتكارية (exclusive access).

إذن استخدام الاحتكار يعتبر أحد الحلول لمشكلة القراءة والكتابة.

4.5.1.1. مشكلة القراءة والكتابة الأولى

The first readers-writers problem

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

5.4.1.2 مشكلة القراءة والكتابه الثانية

The second readers-writers problem

إذا كانت هناك عملية قراءة ، ثم جاءت عملية كتابة، ستنظر الأخيرة حتى تفرغ عملية القراءة، في هذه الائتماء إذا جاءت عملية قراءة أخرى (ممسموح لعمليات القراءة أن تعمل مع بعضها)، ولكن ليس من اللائق أن تخطى هذه العملية عملية الكتابة التي طلبت قبلها استخدام البيانات المشتركة. وإذا سمحنا لعملية القراءة الثانية بالعمل فقد تأتي عملية القراءة الثالثة ورابعة وخامسة وهكذا ، مما يتسبب في حرمان عملية الكتابة من العمل (starvation)، لذلك يجب على عملية الكتابة إذا وصلت صف الانتظار لا تنتظر أكثر من اللازم وأن تبدأ العمل بمجرد وصولها. أي أن عمليات الكتابة تتفقد بمجرد وصولها، بينما تنتظر عمليات القراءة حتى لا تكون هناك عملية كتابة.

5.4.1.3 مشكلة القراءة والكتابه الثالثة

The third readers-writers problem

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

مشكلة القراءة والكتابه الثانية تتسبب في حرمان عمليات القراءة لأن عمليات الكتابة عندما تأتي لصف الانتظار يسمح لها بالعمل عند أول فرصة لها، وبالتالي قد تأتي عمليات الكتابة تباعاً وتستأثر بالوصول للبيانات دون عمليات القراءة. الحل الثالث جاء ليقول أنه يمنع احتكار الوصول للبيانات إذا تعدى فترة زمنية محددة، وكل عملية قيد زمني في استخدامها للبيانات لا تتجاوزه، فإذا كانت هناك عملية تستخدم البيانات احتكارياً فعليها أن تترك هذا الاحتياط وتنهي عملها إذا تعدى الفترة المقررة للعملية.

5.4.2 مشكلة المنتج والمستهلك (producer-consumer)

مشكلة المنتج والمستهلك أحياناً تسمى مشكلة الخازن المحدود (bounded-buffer). هنا يكون لدينا عمليتان مستخدمان خازن مشترك، 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 (5-1): المنتج والمستهلك.

حل مشكلة المنتج والمستهلك

يمكن حل هذه المشكلة بإجبار عملية المنتج على التوقف عن إحضار البيانات للخازن إذا كان ممتلئاً (sleep). وعندما يفرغ الخازن تقوم عملية المستهلك من إيقاظ المنتج ليبدأ في تعبئة الخازن. بنفس الطريقة توقف عملية المستهلك (sleep) إذا كان الخازن فارغاً، 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

الشفرة (5-1) تعتبر حل 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 صحيحا سيتوقف المنتج عن تعبئة الخازن وينهض 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 صحيحا سيتوقف عنأخذ البيانات من الخازن وينهض لينام ولن يرجع لعمله ما لم يوقظه المنتج (عندما يحضر بيانات للخازن). أما إذا كانت هناك بيانات بالخازن فسيواصل المستهلك عمله بأخذ عنصر كل مرة من الخازن بالأمر:

```
item = removeItemFromBuffer()
```

ثم ينقص count بالأمر:

```
count = count - 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)
}
}

شفرة 5-1

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 عنصر جديد إلى الخازن (أصبحت count تحتوي 1).
 - يحاول المنتج إيقاظ المستهلك، وبما أن المنتج أصلاً مستيقظ فإن طلب الإيقاظ هذا لافتادة منه (سيفقد).
 - ولأن المستهلك كان قد اختبر الخازن (قبل إيقافه) وووجهه فارغاً، فإنه بعد أن يتم تشغيله مرة أخرى سيواصل من آخر نقطة كان يعمل فيها وهي الذهاب للنوم (استدعاء sleep)، ولن يستيقظ مرة أخرى، لأن أمر الإيقاظ (wakeup) قد فقد 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 يمتهن:
- ```
if(count == BUFFER_SIZE)
```
- بعدها سيذهب لينام بافتراض أن المستهلك سيوقفه مرة أخرى، ولكن هذا لن يحدث.
  - وبما أن العمليات ستكونان في الفراش، وكل عملية غارقة في نومها وتنتظر الأخرى لتنوتها (ولا يوجد منها)، فستظلان نائمتان هكذا إلى الأبد مما يتبع عنها اختناق لا محالة.

#### الحل باستخدام السيمافور (semaphore)

حل مثل هذه المشكلة يمكننا استخدام ما يسمى بالسيمافور. السيمافور هو علامة كانت تستخدم قديماً لتنظيم مرور القاطرات، حيث سيكون للسيمافور حالتين، إما تحت أو فوق، عندما تصل قاطرة وتريد الدخول للمحطة سيُنظر السائق للسيمافور فإذا كانت وضعيته تحت فهذا يعني أن الطريق سالكة (تشبه إشارة المرور الحضراء)، وأن خط السكة حديد لا يتحمل تلاقي قاطرتين، فسيُرِفُّ السيمافور إلى أعلى بعد دخول القاطرة بحيث لا يسمح لقاطرة أخرى أن تدخل (الخط مشغول) ويُشير هذا إشارة المرور الحمراء. بنفس المفهوم سنستخدم متغير يمثل السيمافور (عدد صحيح يحمل قيمتين)، حيث تمثل قيمة علامة تحت وقيمة أخرى علامة فوق. ثم إذا بدأت عملية استخدام بيانات مشتركة ستُضَعُّفُّ 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.

دائماً 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 (منوع الاقتراب أو التصوير)، بعد أن تفرغ من البيانات المشتركة ستتغير 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 تكمل العملية أو يتم توقيفها (حجزها).

في الشفرة (6-2) سنستخدم السيمافور `fillCount` والسيمافور `emptyCount` لتجنب تجاهل طلبات الإيقاظ وحل 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`.

عندما يريد المستهلكأخذ عنصر من الخازن سيفعل السيمافور `fillCount`، بحيث لا يستطيع المنتج التعبئة في هذه اللحظة، ثم بعد أخذ عنصر من الخازن يفتح السيمافور `emptyCount`.

```
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)
 }
}
```

شفرة 2-5

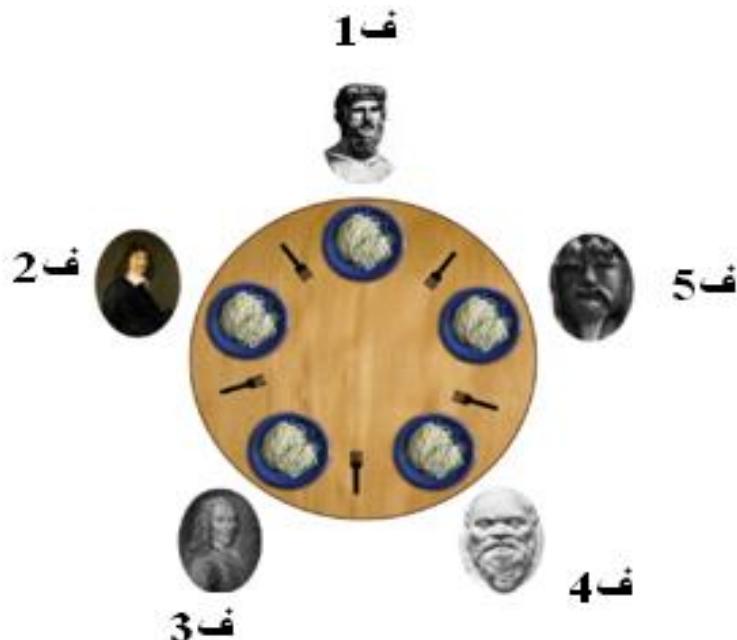
حل السيمافور مناسب إذا كان لدينا منتج واحد ومستهلك واحد. ولكن إذا كان هنالك أكثر من منتج أو أكثر من مستهلك فإن هذا الحل قد ينتهي بهم المطاف (race condition).

#### 5.4.3. مشكلة عشاء الفلسفه (Dining philosophers problem)

هي مشكلة قديمة في التوزي، وهي عبارة عن مشكلة عمليات متعددة غير متزامنة. في عام 1971 ، أعد العالم ديكاسترا (Edsger Dijkstra) سؤال عن التزامن بين العمليات وهو يتحدث عن 5 حاسوبات تتنازع على 5 سواقات أشرطة مغنة مشتركة. وبعدها بفترة قليلة أسمىها العلم توني هوار (Tony Hoare) مشكلة عشاء الفلسفه. وهي تمثيل نظري لعملية الاختناق والحرمان، حيث يقوم كل فيلسوف بأخذ شوكة واحدة في البداية ثم يبحث عن الشوكة الأخرى. تعريف المشكلة

هنالك 5 فلسفه 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 شوكة، شوكة بين كل فيلسوفين، بحيث يكون على يمين كل فيلسوف شوكة وعلى يساره شوكة.

ويحتاج كل فيلسوف أن يستخدم الشوكيتين اللتين على يمينه ويساره مباشرة، الشكل (5-2). إذا أخذ كل الفلسفة شوكاً تم اليسرى، عندها لن يستطيع أي منهم الحصول على الشوكة اليمنى، وسيظل كل فيلسوف متظراً الشوكة اليمنى حتى تفرغ، أي أن الفيلسوف ف1، سيكون في انتظار الفيلسوف ف2 ليترك شوكته، والفيلسوف ف2 سيتظر الفيلسوف ف3 ليترك شوكته، وهكذا (انتظار دائري)، مما يسبب إختناق (deadlock).



شكل رقم (5-2): عشاء الفلسفه.

#### حل غير مجددي:

يمكن حل المشكلة بجعل الفيلسوف يتقطط شوكته اليسرى ثم يختبر هل الشوكة اليمنى متاحة، فإن لم تكن كذلك يضع الفيلسوف شوكته اليسرى ويتناول فترات زمنية محددة ثم يعيد الكرة مرة أخرى. هذه الطريقة قد تفشل إذا قام كل الفلسفة بإلقاء شوكته اليسرى في وقت واحد، فلن يجدوا شوكتهم اليسرى، فسيضعوا شوكتهم اليسرى ويتناولوا فترات زمنية محددة متساوية لكل الفلسفة، ثم يتقططاً شوكتهم اليسرى مرة ثانية، ولن يجدوا اليمين لذلك سيضعوا شوكتهم اليسرى مرة أخرى ويتناولوا فترات زمنية متساوية، وهكذا يظلوا يكررون في هذه المحاولة إلى ما لا نهاية وتظل المشكلة قائمة. هذا الأمر يحرم جميع الفلسفه من الأكل (وتحصل مجاعة) أو ما يسمى الحرمان (starvation).

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

#### حل آخر:

هنا نقوم باستخدام سيمافور ثنائى (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 فيلسوفين قادرین على الأكل في اللحظة الواحدة.

#### حل ثالث:

هنا تتبع حالة الفيلسوف هل هو:

- يأكل.

- يفكـر.

- جائع ( يريد أن يحصل على شوكة).

حيث يستطيع الفيلسوف الجائع أن يأكل فقط إذا كان جاريه لا يأكلان.

#### 5.4.4 مشكلة مدخني السجائر (Cigarette smokers problem)

هي أحد مشاكل التوازي التي تتحدث عن التزامن بين أربعة برامح. ظهرت في 1971. وقصتها كالتالي:  
إفترض أن السيجارة تتكون من ثلاثة أجزاء هي:

- .Tobacco

- ورق paper

- عود ثقاب match

فإذا كان هناك ثلاثة مدخنين يجلسون حول طاولة، كل مدخن لديه مخزون كافي من أحد أجزاء السيجارة. وهناك شخص رابع غير مدخن مهمته اختيار أثنين من المدخنين الثلاثة عشوائيا، ليضع كل واحد من هؤلاء الإثنين قطعة واحدة مما يمتلك من مخزونه على الطاولة، فمثلاً إذا اختارنا صاحب التبغ وصاحب الورق، فسيضع كل منهما ما يكفي لعمل سيجارة واحدة. ويطلب من المدخن الثالث عمل السيجارة فيقوم باخذ ما في الطاولة ثم إضافة المكون الذي لديه لعمل السيجارة، مثلاً إذا كان المدخن الثالث صاحب الثقاب، فسيأخذ الورقة ويلف عليها التبغ الموجودين بالطاولة ثم يشعل السيجارة بعود ثقاب من عنده. عند ما تصبح الطاولة فارغة يقوم الشخص الرابع باختيار أثنين من المدخنين عشوائياً لتم نفس الخطوات السابقة. ويظل تكرار هذه الخطوات دون توقف. لا يتم تخزين سجائر وإنما تصنع السيجارة ثم تدخن مباشرة، ثم تصنع أخرى وتدخن وهكذا، فليس من المسموح عمل أكثر من سيجارة في نفس الوقت. إذا تم وضع ورق وتبغ على الطاولة وكان صاحب اعود الثقاب يدخن ستظل هذه المكونات على الطاولة دون ان يلمسها احد حتى يكمل صاحب الثقاب التدخين ثم يقوم بعمل السيجارة.

المشكلة تحاكي باربعة برامح تمثل الادوار الاربعة (ثلاثة مدخنين و شخص رابع للاختيار). مسموح باستخدام السيمافور للتزامن، ومنع لأي واحد من البرامح الاربعة بعمل فرز شرطي، فقط يمكن استخدام الانتظار المشروط باستخدام wait وحل

- تحل هذه المشكلة باستخدام سيمافور ثنائي أو mutexes.

- نعرف مصفوفة من السيمافورات الثنائية A ووحدة لكل مدخن.

- سيمافور ثنائي للطاولة T.

- هي كل سيمافورات المدخنين بقيمة إبتدائية 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 ١ هي:

```

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

```

#### 5.4.5. اللقاء (Rendezvous)

نفترض أن شابين قد توعدا على أن يلتقيا في منتهه لم برياه من قبل، وبالفعل حضر كل واحد على حدا، ولكنهما إندهشا من كبر حجم المنتزه، وبالتالي هنالك صعوبة في أن يجد أحدهما الآخر. هنا على كل شاب أن يختار واحد من أمرتين:

- أن يتضمن في مكان واحد ويتوقع أن الآخر سيبحث عنه.

- أن يبدأ بالبحث بإفتراض أن الآخر سيتمنى 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 فرصه في ان يتقابلان.

- إذا قرر أحدهما أن يتضمن الآخر أن يبحث فتحتما سيلتقيا (من ناحية نظرية).

#### 5.4.6. مشكلة الحلاق النائم (sleeping barber)

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

الحل

حل هذه المشكلة نستخدم ثلاثة 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. كل ما 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 والا فسيغادر (يحاول الحصول على كرسي سواء في غرفة الإنتظار أو كرسي الحلاق نفسه).

إذا وجد الزبون كرسي سيعجلس وينقص عدد الكراسي الفارغة بواحد (critical section).  
 يقوم الزبون بإسال إشارة إلى الحلاق لإيقاظه، وسيحرر mutex للزبائن (أو الحلاق) بالقدرة على الحصول عليه. إذا كان الحلاق مشغول فعلى الزبائن الانتظار. سيجلل الحلاق في إنتظار دائم، يتم إيقاظه بأي زبون من المنتظرين، وعندما يستيقظ سيرسل إشارات للزبائن المنتظرين بواسطة السيمافور للسماح لهم بالحلاقة ، واحد كل مرة. هذه المشكلة تختوي على حلاق واحد لذلك تسمى أحياناً (single sleeping barber problem)، فيما يلي شفرات مبدئية (pseudo-code)، تضمن التزامن بين الحلاق والزبائن خالية من الاختناق، ولكنها قد تقود إلى حرمان الزبائن.  
 نجد P و V هما دالتين 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 منتهي //
 محاولة الحصول على زبون وإذا لا يوجد زبائن ينام //
 في هذه اللحظة تم إيقاظه، يريد تعديل عدد المقاعد المتاحة //
 NumberOfFreeSeats++ //
 لقد أصبح هناك مقعد فارغ //
 الحلاق جاهز ليبدأ الحلاقة //
 V(accessSeats) //
 لأن يريد إغلاق الكراسي //
 الحلاق الآن يخلق لزبون //
}
```

عملية أو خيط الزبون:

```
while(true) {
 تكرار لا منتهي //
 محاولة الحصول على مقعد //
 P(accessSeats) //
 إذا كان هناك مقعد فارغ // {
 NumberOfFreeSeats-- //
 إجلس على المقعد //
 آخر الحلاق ، الذي هو في انتظار زبون //
 V(accessSeats) //
 لا يحتاج إلى إغلاق الكراسي //
 الآن حان دور الزبون، لكنه سينتظر إذا كان الحلاق مشغول //
 هنا سيمكن الزبون من الحلاقة //
 } else {
 لا توجد مقاعد فارغة //
 V(accessSeats) //
 حرر إغلاق المقاعد //
 سيعادر الزبون دون أن يخلق //
 }
}
```

## 5.5. ملخص

تحدثنا في هذا الباب عن التزامن وهو كيف لعدة عمليات تعمل معاً تنسق عملها بحيث لا تؤثر إحداها على الأخرى.  
ويعتبر هذا المفهوم مهم جداً في البرمجة المتوازية وعلى المبرمج أن يراعي المشاكل التي قد تنتجه عن التزامن مثل مشاركة البيانات وتعديلها من قبل أكثر من عملية في نفس الوقت.

### تمارين غير محلولة

.1 ما هو مفهوم التزامن؟

.2 ما المقصود بالنزاع؟

.3 ما الفرق بين العملية المستقلة والعملية المتعاونة؟

.4 لماذا تحتاج العمليات المتوازية التعاون فيما بينها؟

.5 ما المقصود بحالة السباق (race condition)؟ وما هي مسبباتها؟ وكيف يتم حلها؟

.6 عرف المنطقية الحرجة (critical section)؟

.7 عرف اللقاء؟ Rendezvous

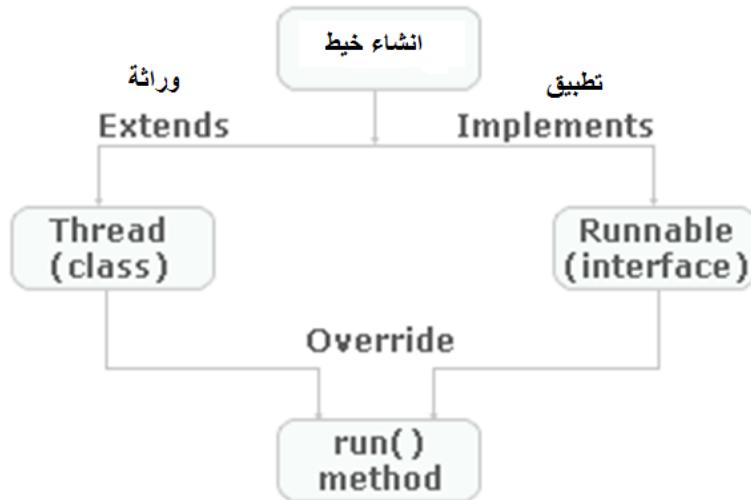
.8 أذكر خمسة من مشاكل التزامن الكلاسية؟

.9 تتصل العمليات المتعاونة فيما بينها بطريقتين، ما هما؟

## الباب السادس

# خيوط جافا

من أشهر لغات البرمجة التي تطبق مفهوم الخيوط على مستوى اللغة هي لغة جافا، فهي من اللغات التي توفر دعم الخيط على مستوى اللغة، ولها مكتبة كاملة لإنشاء وإدارة الخيوط. سنتطرق في هذا الباب لترجمة العالجات متعددة النواة باستخدام لغة جافا. وسنعطي أمثلة برمجية في هذا المجال.



شكل رقم 6-1: طرقين لإنشاء الخيط .

### 6.1. إنشاء الخيط

هناك طريقتين لإنشاء الخيط في الجافا، شكل 5-1، هما:

1. بوراثة فئة الخيط المسممة Thread يعني (Extends Thread).
2. عن طريق تطبيق الواجهة المسممة Runnable يعني (implement Runnable).

كل برامج في جافا لديها على الأقل خيط واحد رئيسي هو main.

### إنشاء خيط بالطريقة الأولى

مثلاً إذا أردنا إنشاء خيط اسمه th سنقوم بكتابة الأمر التالي:

```
class th extends Thread
```

هذا الأمر ينشئ فئة اسمها th يرث صفات الفئة Thread والتي توجد ضمن مكتبات جافا التي تدعم الخيوط. هذه الفئة تحتوي على دالة (طريقة) تسمى run، هذه الدالة هي المسؤولة عن تنفيذ الخيط، وقد ورثناها من الفئة Thread. سنضع ما نريد تنفيذه داخل هذه الدالة، مثلاً:

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

بعدها ننشئ كائن من الصف th، مثلاً:

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

الآن أصبح لدينا خيط هو t1، يمكن تشغيله بالأمر:

```
t1.start()
```

تبينه: 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 (main). إذن الآن لدينا خيطين في العملية. خيط العملية الأساسي main، والخيط t1.

البرنامج (1-5) هو برنامج صغير يوضح كيفية إنشاء خيط وتشغيله.

```
class th extends Thread {
 @Override
 public void run(){
 System.out.println("Inside thread");
 }
}

class threadTest{
 public static void main(String args[]){
 th t1 = new th(); // إنشاء خيط
 t1.start(); // تشغيل الخيط
 System.out.println("Inside main");
 }
}
```

برنامـج 1-5

```
run:
Inside main
Inside thread
BUILD SUCCESSFUL (total time: 0 seconds)
```

مخرج البرنامج 1-5

### الطريقة الثانية

تحتوي الفئة Runnable على دالة واحدة هي run وهي معرفة داخل هذه الفئة كالتالي:

```
public void run()
```

داخل هذه الدالة يمكن وضع ما تريده تنفيذه داخل الخيط.

داخل هذه الدالة تستطيع استدعاء دوال أخرى، تعريف متغيرات، استخدام فئات أخرى، مثل الدالة main.

```

class th2 implements Runnable {
 @Override
 public void run() {
 System.out.println(" Inside thread ... ");
 }
}

class threadTest2 {
 public static void main(String [] args) {
 System.out.println("Inside main ... ");
 Thread t = new Thread(new th2());
 t.start();
 }
}

```

اللقطة 2 تطبيق للفئة th2

تكتب ما تريد تنفيذه في الخيط هنا

كان يمثل الخيط

تمرير الفئة التي تطبق Runnable إلى فئة الخيط

تنفيذ الخيط

برنامح 2-5.

```

run:
Inside main ...
Inside thread ...
BUILD SUCCESSFUL (total time: 0 seconds)

```

مخرج البرنامج 2-5.

سؤال

كيف ننشئ أكثر من خطيدين، كما في الشكل 5-2؟ ووضح كيف يتم ذلك بالطريقتين السابقتين؟

مساعدة: يمكنك إنشاء أكثر من خيط بالأوامر التالية:

th A = new th();

th B = new th();

th C = new th();

وللتتنفيذ نكتب:

A.start();

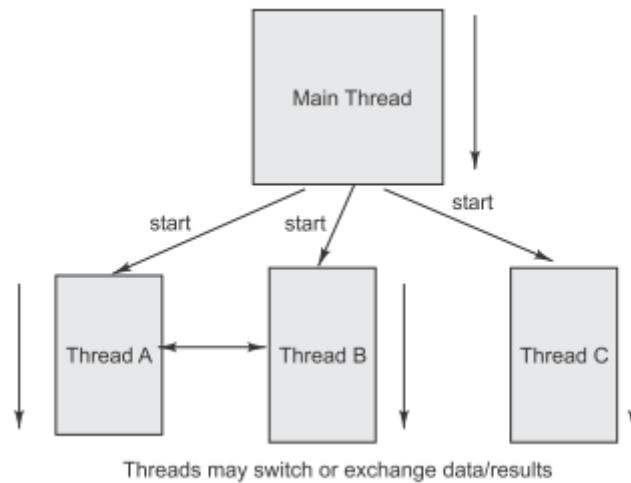
B.start();

C.start();

#### ćمارين عملية

1. عدل البرنامج أعلاه ليطبق نموذج SIMD، بحيث تنفذ الثلاث خيوط نفس المهمة على بيانات مختلفة.

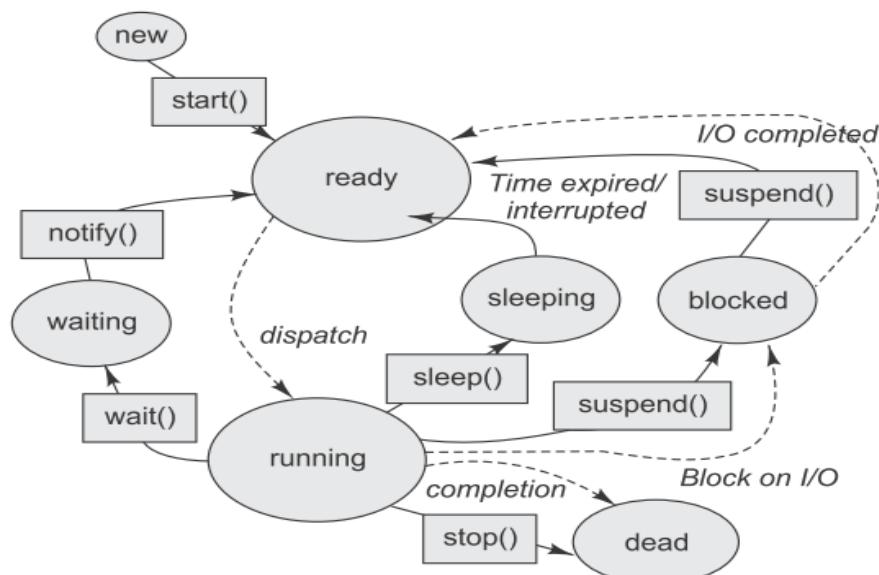
2. عدل البرنامج مرة ثانية بتطبيق نموذج MIMD، بحيث تنفذ الثلاث خيوط مهام مختلفة على بيانات مختلفة



شكل رقم 5-2: خيط رئيسي وثلاث خيوط فرعية [12].

## 6.2. دورة حياة خيط جافا

يمثل الخيط بحالات مختلفة بدأ من إنشائه ثم تشغيله ثم اكتماله. وقد يتوقف أثناء التنفيذ لسبب ما. ثم يواصل التنفيذ مرة أخرى بعد انتهاء السبب. وقد يتم إنتهاءه قبل اكتماله بالأمر `stop()`. هذه الحالات والسبب في انتقال الخيط من حالة إلى أخرى موضحة في الشكل 5-3.



شكل رقم 5-3: دورة حياة الخيط في جافا [12].

## 6.3. التعامل مع الخيط

فيما يلي بعض الدوال التي تتحكم في دورة حياة المبنية في الشكل 5-3.

| تشغيل الخيط                                                     | Start()   |
|-----------------------------------------------------------------|-----------|
| إنهاء الخيط، حيث يصبح الخيط ميت ولا يمكن تشغيله مرة ثانية.      | Stop()    |
| توقف الخيط، ويمكن تشغيله مرة ثانية بالأمر <code>resume()</code> | suspend() |

|                                                                                                                               |          |
|-------------------------------------------------------------------------------------------------------------------------------|----------|
| تشغيل الخيط مرة ثانية بعد توقفه بالأمر (.suspend())                                                                           | resume() |
| وضع الخيط في حالة الانتظار حتى يتم ايقاظه بالأمر (notify())                                                                   | wait()   |
| ايقاظ الخيط الذي كان في حالة انتظار                                                                                           | notify() |
| انتظار الخيط لفترة زمنية محددة، مثلا (st.sleep(4000) يجعل الخيط ينتظر 4 ثواني. ثم يعود ليعمل مرة أخرى بعد انقضاء هذه الثواني. | Sleep()  |

ما الفرق بين suspend, wait, sleep و ?

#### 6.4. حالات الخيط

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

- جديد (new): عند إنشائه، كما في الأمر التالي:

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

- جاهز (Ready): الأمر start() يحجز مساحة بالذاكرة للخيط، ويستدعي الطريقة run() بالكائن t1، هنا يكون الخيط جاهز بالذاكرة جاهز للتنفيذ.

شغال (Running): اذا قام المجدول بادخاله للمعالج يصبح الخيط شغال.

محجوز (blocked): يصبح الخيط محجوزاً إذا تم توقفه بأحد الأوامر التالية:

suspend() أو sleep(), stop().

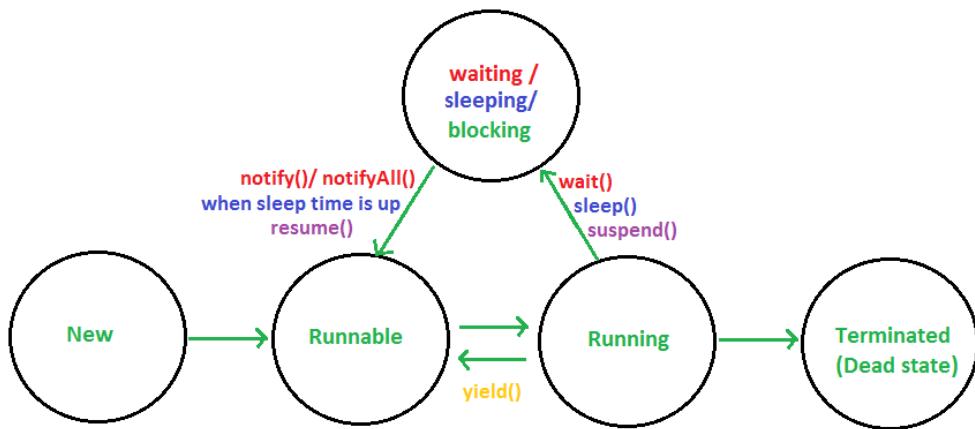
أو احتاج إلى عملية دخل/خرج أو نتهي عمله.

إذا انتهت عمله أو اوقف بالأمر stop()، فإنه سيموت (dead).

إذا اوقف بالأمر suspend() فيمكن ان يتتحول الى حالة جاهز بالأمر resume()، أما اذا اقف بالأمر sleep() ففيه تستخدم قيمة تحدد زمن الوقوف بعدها يرجع الخيط إلى حالة جاهز، مثلا (sleep(4000) توقف الخيط مدة اربعة ثواني ثم يرجع الى وضع جاهز.

• ميت (dead): يصبح الخيط ميتاً إذا انتهى عمل الخيط أو استدعينا الأمر stop()

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



**Fig. THREAD STATES**

شكل رقم 5-4: اختصار حالات الخيط الموجودة في الشكل 5-5 [13].

احيانا يطلق على كلمة **Runnable**

ما الفرق بين استخدام طريقي انشاء الخيط في جافا (وراثة Thread وتطبيق Runnable)؟ وأيهما أفضل؟

ذكر Subham Mittal في [14] خمس فروق بين الطريقتين لخصها في الجدول 5-1.

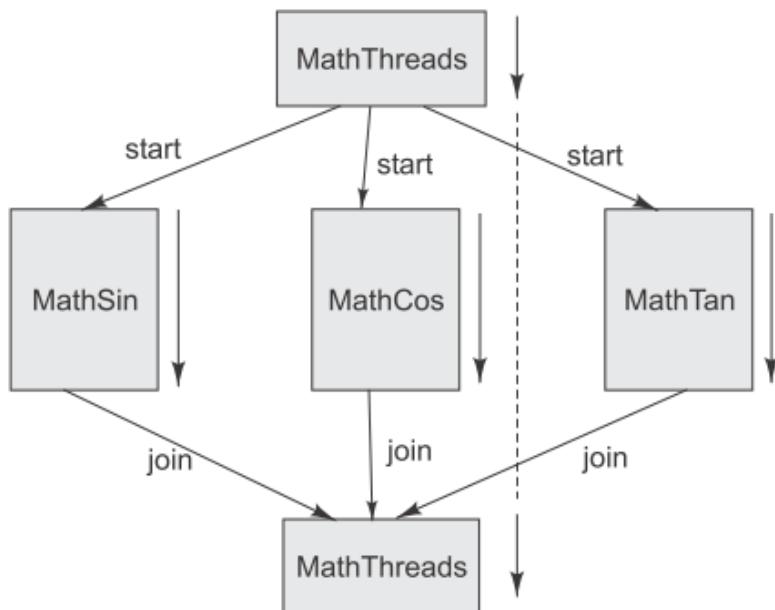
جدول رقم 5-1: الفرق بين طريقي انشاء الخيوط في جافا.

| وراثة Thread         | تطبيق Runnable         |                               |
|----------------------|------------------------|-------------------------------|
| لا توجد وراثة متعددة | انشاء اي عدد من الفئات | خيارات الوراثة                |
| لا                   | نعم                    | اعادة الاستخدام (Reusability) |
| سيئ                  | جيد                    | Object Oriented Design        |
| لا                   | نعم                    | Loosely Coupled               |
| نعم                  | لا                     | Function Overhead             |

جافا لا تدعم الوراثة المتعددة (multiple inheritance).

## 6.5. مثال برمجي

في البرنامج 5-3 نريد إنشاء ثلاثة خيوط بالإضافة إلى الخيط الرئيسي، كما موضح في الشكل 5-4. وذلك للقيام بعمليات حسابية مختلفة في وقت واحد، فإذاً الخيوط يحسب جيب الزاوية ( $\sin$ ) ( $\text{Sin}$ )، والآخر يحسب جيب تمام الزاوية ( $\cos$ ) ( $\text{Cos}$ )، والثالث يحسب ظل الزاوية ( $\tan$ ) ( $\text{Tan}$ ). ثم يقوم الخيط الرئيسي بجمع هذه النتائج وإظهارها للمستخدم.



شكل رقم 5-4: برنامج يتكون من أربعة خيوط [12].

```
package threads_chapter;
class MathSin extends Thread {
 public double deg;
 public double res;
 public MathSin(int degree) {
 deg = degree;
 }
 @Override
 public void run() {
 System.out.println("Executing sin of " + deg);
 double Deg2Rad = Math.toRadians(deg);
 res = Math.sin(Deg2Rad);
 }
}
```

```
 System.out.println("Exit from MathSin. Res = " + res);
 }
}

class MathCos extends Thread {
 public double deg;
 public double res;
 public MathCos(int degree) {
 deg = degree;
 }
 @Override
 public void run() {
 System.out.println("Executing cos of " + deg);
 double Deg2Rad = Math.toRadians(deg);
 res = Math.cos(Deg2Rad);
 System.out.println("Exit from MathCos. Res = " + res);
 }
}

class MathTan extends Thread {
 public double deg;
 public double res;
 public MathTan(int degree) {
 deg = degree;
 }
 public void run() {
 System.out.println("Executing tan of " + deg);
 double Deg2Rad = Math.toRadians(deg);
 res = Math.tan(Deg2Rad);
 System.out.println("Exit from MathTan. Res = " + res);
 }
}

class MathThreads {
 public static void main(String args[]) {
 MathSin st = new MathSin(45);
```

```

MathCos ct = new MathCos(60);
MathTan tt = new MathTan(30);
st.start();
ct.start();
tt.start();
try { // wait for completion of all thread and then sum
 st.join();
 ct.join(); //wait for completion of MathCos object
 tt.join();
 double z = st.res + ct.res + tt.res;
 System.out.println("Sum of sin, cos, tan = " + z);
} catch (InterruptedException IntExp) {
}
}
}

```

.[12] 3-5 برنامج

```

run:
Executing tan of 30.0
Exit from MathTan. Res = 0.5773502691896257
Executing cos of 60.0
Executing sin of 45.0
Exit from MathCos. Res = 0.5000000000000001
Exit from MathSin. Res = 0.7071067811865475
Sum of sin, cos, tan = 1.7844570503761732
BUILD SUCCESSFUL (total time: 0 seconds)

```

3-5 مخرج برنامج

نلاحظ في البرنامج 3-5 اننا استخدمنا الامر (join)، وهو يستخدم لجعل المستدعى لهذا الامر الانتظار إلى يكتمل تنفيذ الخيط المستدعى. فمثلا لو أننا استدعيينا الامر () main st.join() داخلا (الخيط الرئيسي)، فهذا يجعل تنتظراً اكمال الخيط st قبل تنفيذ أي عملية تلي هذا الاستدعاء. في البرنامج سينتظر الخيط الرئيسي اكتمال الخيوط الثلاثة قبل أن يقوم بعملية جمع النتائج. فالأوامر :

```

st.join();
ct.join();

```

```
tt.join();
```

تعني أننا ننتظر اكتمال الخيوط st، ct، و tt قبل تنفيذ الاوامر (التي تلي الاستدعاء) وهي:

```
double z = st.res + ct.res + tt.res;
System.out.println("Sum of sin, cos, tan = " + z);
```

ما نوع النموذج الذي يطبقه البرنامج 3-5 ؟ SIMD أم MIMD ؟ وضح ؟

تمرين عملي

أكتب برنامج من خمسة خيوط (من نموذج SIMD)، لاستخراج الاعداد الاولية بين 1 و 100000000 ؟ بحيث يقوم كل خيط باستخراج الاعداد الأولية في جزء معين ؟

- الخيط الاول يستخرج الاعداد الاولية بين 1 و 25000000
- الخيط الثاني يستخرج الاعداد الاولية بين 25000000 و 50000000
- الخيط الثالث يستخرج الاعداد الاولية بين 50000000 و 75000000
- الخيط الرابع يستخرج الاعداد الاولية بين 75000000 و 100000000
- الخيط الخامس هو الرئيسي ويقوم بتحميم النتائج من الخيوط الاربعة ثم اظهار الناتج.

تمارين برمجية

أكتب البرامج التالية باستخدام جافا:

برنامـج (1)

يقوم بطباعة سلسلة فيبوناكي (Fibonacci series) أي:

1,2,3,5,8,13,...

يجب أن يعمل البرنامج كالتالي:

ينفذ المستخدم البرنامج ثم يدخل رقم يمثل عدد الأعداد التي تريده من البرنامج أن يولدها من سلسلة فيبوناكي، هنا ينشئ البرنامج خيط يقوم بعملية توليد الأعداد.

Enter number of Fibonacci numbers to generate ? 5

1,2,3,5,8

برنامـج (2)

يقوم بطباعة الأعداد الأولية من 1 إلى رقم معين. حيث يطلب البرنامج من إدخال الرقم الذي تريده توليد أعداد أولية أقل أو تساويه، ثم ينشئ البرنامج خيط يقوم بعملية توليد الأعداد الأولية.

Enter number to generate primes less than or equal to ? 5

2,3,5

## الباب السابع

# برمجة الخيوط باستخدام Pthreads

بعد أن تحدثنا في الأبواب السابقة عن مفهوم الخيوط و التزامن ، وصرينا أمثلة بسيطة عن استخدامات الخيوط، في هذا الباب سنتحدث بالتفصيل عن كيفية كتابة برامج متعددة الخيوط (ذاكرة مشتركة) تحوي على بيانات مشتركة وكيف نمنع الوصول المتزامن لهذه البيانات وعدم الوقوع في حالات السباق (race conditions).

سنتستخدم مكتبات Pthreads التي تعمل مع لغة C/C++ والتي تستخدم في معظم نظم تشغيل لينكس، ويمكن استخدام نفس الخطوات على نسخة pThreads-win32 لتنفيذها على نظام ويندوز (موضح في الملحق خطوات استخدام win32 في ويندوز مع فيجوال استديو 2010).

### 7.1. مكتبات POSIX thread (Pthread)

مكتبات POSIX thread هي واجهة تطبيقات برمجية لدعم الخيوط للغة C/C++. وهي تسمح لنا بكتابة برامج متعددة الخيوط تعمل بالتوظاقي . 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 تقديرية مساحة ذاكرة ظاهرية وبيئة لكل عملية جديدة.

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

هناك تقنيات برمجة متوازية أخرى مثل MPI و PVM تستخدم في بيئات حوسية موزعة بينما تقتصر الخيوط على أنظمة الحواسيب الواحدة (single computer system). كل الخيوط في نفس العملية (البرنامج) تشارك في نفس مساحة العنونة (address space). هناك العديد من النماذج الشائعة للبرامج متعددة الخيوط منها:

1. نموذج المدير/العامل (Manager/worker): حيث يقوم الخيط المدير بتوزيع المهام على الخيوط الأخرى والتي تمثل العمال.

2. نموذج الأنابيب الأنسيابي (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).

3. الند (peer): يشبه نموذج المدير/العامل، لكنه مختلف في أن المدير بعد توزيعه المهام على العمال يشارك هو في التنفيذ.

### 7.2. نموذج الذاكرة المشتركة

- كل الخيوط تصل إلى الذاكرة المشتركة (بيانات مشتركة (global)).
- لكل خيط بيئاته الخاصة.
- المبرمج مسئول عن عمليات الوصول المتزامن (حماية البيانات المشتركة).

برمجة الخيط بأمان

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

مثلاً إذا كان هناك تطبيق يحتوي على عدة خيوط وكل خيط يستدعي نفس روتين المكتبة (library routine)، فإذا افترضنا أن هذا الروتين يقوم بتعديل بيانات عامة أو موقع في الذاكرة، فسيحاول كل خيط تعديل هذه البيانات في نفس الوقت مما يسبب

تخريبًا في هذه البيانات، لذلك لا بد من أن يكون هناك نوع من الوصول المتزامن لهذه البيانات حتى تخفيها من التخريب حتى يصبح الخطيط آمنا.

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

### 7.3. واجهة التطبيقات البرمجية (Pthreads API)

يمكن تقسيم الواجهة البرمجية إلى أربعة مجموعات رئيسية هي:

- إدارة الخطيط: وهي روتينات التي تعمل مباشرة مع الخيوط مثل creating, detaching, joining, الخ.
- روتينات mutex: والتي تستخدم في التزامن وتسمى (Mutex) من الكلمتين mutual exclusion، وهي معنية بتعديل الصفات المرتبطة مع المutexes.
- المتغيرات الشرطية (condition variables): وهي الروتينات التي تحتم بالاتصالات بين الخيوط المترابطة في mutex.
- التزامن (synchronization): روتينات لإدارة حجز القراءة والكتابة.

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

| Routine Prefix                  | Functional Group                                 |
|---------------------------------|--------------------------------------------------|
| <code>_pthread</code>           | Threads themselves and miscellaneous subroutines |
| <code>_pthread_attr</code>      | Thread attributes objects                        |
| <code>_pthread_mutex</code>     | Mutexes                                          |
| <code>_pthread_mutexattr</code> | Mutex attributes objects.                        |
| <code>_pthread_cond</code>      | Condition variables                              |
| <code>_pthread_condattr</code>  | Condition attributes objects                     |
| <code>_pthread_key</code>       | Thread-specific data keys                        |
| <code>_pthread_rwlock</code>    | Read/write locks                                 |
| <code>_pthread_barrier</code>   | Synchronization barriers                         |

### 7.4. إنشاء خطيط

لإنشاء خطيط تحتاج استخدام الدالة `pthread_create()` والتي تحتوي على المعلميات التالية:

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

مثلاً 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 ثالث خيوط فعليك استدعاء الدالة (`pthread_create()`) ثلث مرات.

أيضاً نستدعي الدالة (`join()`) مع كل خيط قمنا بإنشائه لضمان إنتهاء الخيوط قبل إنتهاء الدالة الرئيسية `main`.



شكل رقم 7-1: إنشاء خيط.

## 7.5. إنهاء خيط

لإنهاخ خيط نستخدم الدالة (`pthread_cancel()`، لكن عليك التأكد من أن الخيط الذي تريد إنهاخه لا يستخدم موارد قبل إنهاخه. مثلاً إذا كان الخيط يحجز مساحة بالذاكرة وقمنا باستدعاء دالة الإنهاخ (`pthread_cancel()`) ستفقد مكان هذه الذاكرة وستظل محجوزة بلا فائدة (`memory leak`).

|                                                   |
|---------------------------------------------------|
| pthread_create (thread, attr, start_routine, arg) |
| pthread_exit (status)                             |
| pthread_attr_init (attr)                          |
| pthread_attr_destroy (attr)                       |
| الروتينيات المستخدمة في إنشاء وإنهاخ الخيوط.      |

تنبيهات عن إنشاء الخيط:

- الدالة الرئيسية `main` لديها خيط واحد إفتراضي، بقية الخيوط على المبرمج أن يقوم بإنشائها بنفسه.
- الأمر `pthread_create` ينشيء خيط ويمكن استدعائه أكثر من مرة من أي مكان داخل الشفرة لإنشاء أكثر من خيط.
- عند إنشاء الخيط يمكنه بدوره إنشاء خيوط أخرى، فليس هنالك هرمية بين الخيوط.

بعد إنشاء الخيط كيف سترى متى سيجدول وينفذ بواسطة نظام التشغيل؟

## 7.6. صفات الخيط (Thread Attributes)

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

### إنهاء الخيط (Terminating Threads)

- هناك عدة طرق لإنهاء الخيط منها:
  - رجوع الخيط من الروتين الذي بدأ فيه (الروتين الرئيسي الذي قام بتاهية الخيط).
  - استدعاء الخيط للروتين `pthread_exit`.
  - إلغاء الخيط بخيط آخر وذلك باستدعاء الروتين `pthread_cancel`.
  - إذا انتهت العملية بكمالها باستدعاء روتين مثل `exit` أو `exec`.
- يمكن استخدام الروتين `pthread_exit` للخروج من الخيط ويتم استدعاؤه في نهاية الخيط عند ما نريد عمل شيء آخر.
- إذا إنتهى البرنامج الرئيسي `main()` قبل الخيوط التي أنشأها وخرج بالامر `pthread_exit()`، فإن الخيوط الأخرى ستظل تعمل. أما إذا إنتهت `main()` فستنتهي معها كل الخيوط التي أنشأها.
- يستطيع المبرمج (اختيارياً) 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 واستظل مفتوحة حتى بعد إنتهاء الخيط.
- ملحوظة: في الروتينات التي من المتوقع انتهاء تفيذها بصورة طبيعية يمكننا الاستغناء عن `pthread_exit()`، ما لم نريد تمرير `threads` (threads). لكن هناك مشكلة غير منتهية عندما تكمل `main()` قبل توزيع الخيوط (pass a return code back). فإذا لم تستدعي `pthread_exit()` (it spawned `main()` it spawned `main()` فإن العملية (وكل خيوطها) ستنتهي. باستدعاء `pthread_exit()` في `main()`، فإن العملية وكل خيوطها ستتحفظ حية حتى ولو أكتمل تفيذ كل `main()`. شفرة الموجودة في

```
#include "stdafx.h"
#include <pthread.h>
#include <stdio.h>
void * entry_point(void *arg)
{
 printf("Hello world!\n");
 return NULL;
}
int main(int argc, char **argv)
{
 pthread_t thr;
 if(pthread_create(&thr, NULL, &entry_point, NULL))
 {
 printf("Could not create thread\n");
 return -1;
 }
 if(pthread_join(thr, NULL))
 {
 printf("Could not join thread\n");
 return -1;
 }
 return 0;
}
```

برنامـج 0-7: برنامـج بسيـط ينشـي خـيط واحـد.

Hello world!

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

```
#include "stdafx.h"
#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);

 return 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 %d\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);
}
```

[برنامـج 7-1. من](https://computing.llnl.gov/tutorials/pthreads/samples/hello.c)

مخرج البرنامج:

```
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
In main: creating thread 2
In main: creating thread 3
Hello World! It's me, thread #1!
Hello World! It's me, thread #2!
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
```

شرح البرنامج 7-1

الدالة `PrintHello` هي دالة نريد تفريذها داخل خيط في الدالة الرئيسية `main`. لذلك فإن الامر:  
`pthread_create(&threads[t], NULL, PrintHello, (void *)t);`

سيقوم بإنشاء 5 خيوط (من 0 إلى 4). وكل خيط سينفذ الدالة `PrintHello` وعبر لها القيمة (0 إلى 4). ويتم ذلك كالتالي:  
عندما تكون قيمة `t=0` فإنه سيتم إنشاء الخيط `[0, threads[0]`، وتطبع الدالة `PrintHello` رسالة الترحيب مع القيمة المرسلة لها (وهي محتوى المتغير `t` يعني 0). في نفس الوقت تطبع الدالة الرئيسية رسالتها بالأمر:

```
printf("In main: creating thread %ld\n", t);
```

وستكون في هذه الحالة 0 .t=0، In main: creating thread 0  
سيتكرر هذا الأمر لبقية قيم t (1,2,3, و 4).

## 7.7. تمرير قيم إلى الخيط (Passing Arguments to Threads)

نلاحظ في البرنامج 7-1 أن القيمة التي تمريرها للدالة هي مخزنة في t، وهي قيمة واحدة. ذلك لأن الدالة (pthread\_create()) تسمح بتمرير قيمة واحدة (one argument) للدالة التي تنفذ الخيط. وتتمرير أكثر من قيمة يمكنك استخدام بنية بيانات (data structure) تحتوي كل القيم التي تريد تمريرها، ثم وضع مؤشر هذه البنية في الدالة (pthread\_create()). البرنامج 7-2 يوضح كيف يمكن تمرير عدد صحيح (integer) لكل خيط. الاستدعاء يتم باستخدام بنية بيانات (data structure) واحدة لكل خيط.

```
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]);
 ...
}
```

برنامج 7.2

في البرنامج 7-3 نوضح كيفية اعداد/تمرير أكثر من قيمة باستخدام بنية (structure). كل خيط يستلم نسخة واحدة من هذه البنية.

```
/****************************************************************************
* FILE: hello_arg2.c
* DESCRIPTION:
* A "hello world" Pthreads program which demonstrates another safe way
* to pass arguments to threads during thread creation. In this case,
* a structure is used to pass multiple arguments.
* AUTHOR: Blaise Barney
* LAST REVISED: 01/29/09

#include "stdafx.h"
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 8

char *messages[NUM_THREADS];

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

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
 int taskid, sum;
 char *hello_msg;
```

```

struct thread_data *my_data;

//sleep(1);
my_data = (struct thread_data *) threadarg;
taskid = my_data->thread_id;
sum = my_data->sum;
hello_msg = my_data->message;
printf("Thread %d: %s Sum=%d\n", taskid, hello_msg, sum);
pthread_exit(NULL);
return NULL;
}

int main(int argc, char *argv[])
{
pthread_t threads[NUM_THREADS];
int *taskids[NUM_THREADS];
int rc, t, sum;

sum=0;
messages[0] = "English: Hello World!";
messages[1] = "French: Bonjour, le monde!";
messages[2] = "Spanish: Hola al mundo";
messages[3] = "Klingon: Nuq neH!";
messages[4] = "German: Guten Tag, Welt!";
messages[5] = "Russian: Zdravstvyyte, mir!";
messages[6] = "Japan: Sekai e konnichiwa!";
messages[7] = "Latin: Orbis, te saluto!";

for(t=0;t<NUM_THREADS;t++) {
 sum = sum + t;
 thread_data_array[t].thread_id = t;
 thread_data_array[t].sum = sum;
 thread_data_array[t].message = messages[t];
 printf("Creating thread %d\n", t);
 rc = pthread_create(&threads[t], NULL, PrintHello, (void *)
 &thread_data_array[t]);
 if (rc) {
 printf("ERROR; return code from pthread_create() is %d\n", rc);
 exit(-1);
 }
}
pthread_exit(NULL);
}

```

برنامه 3-7

مخرج البرنامج:

```

Creating thread 0
Creating thread 1
Thread 0: English: Hello World! Sum=0
Creating thread 2
Thread 1: French: Bonjour, le monde! Sum=1
Creating thread 3
Thread 2: Spanish: Hola al mundo Sum=3
Creating thread 4
Thread 3: Klingon: Nuq neH! Sum=6
Creating thread 5
Thread 4: German: Guten Tag, Welt! Sum=10
Creating thread 6
Thread 5: Russian: Zdravstvyyte, mir! Sum=15
Creating thread 7
Thread 6: Japan: Sekai e konnichiwa! Sum=21
Thread 7: Latin: Orbis, te saluto! Sum=28

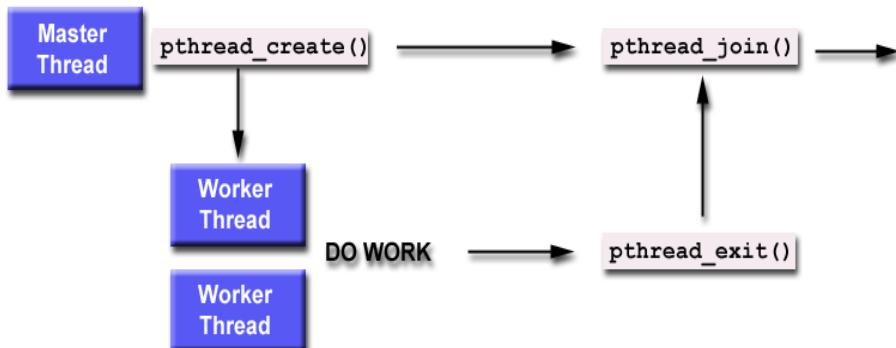
```

|                                                       |
|-------------------------------------------------------|
| <b>pthread_join</b> (threadid,status)                 |
| <b>pthread_detach</b> (threadid,status)               |
| <b>pthread_attr_setdetachstate</b> (attr,detachstate) |

|                                                                 |
|-----------------------------------------------------------------|
| <b>pthread_attr_getdetachstate (attr, detachstate)</b>          |
| الدوال المستخدمة في انضمام وانفصال الخيوط (Joining & Detaching) |

## 7.8. الانضمام (Joining)

- "Joining" هي احدى الطرق لعمل تزامن (synchronization) بين الخيوط، الشكل 7-2.
- الروتين `pthread_join()` يحجز الخيط المستدعي (calling thread) حتى يتنهي الخيط المحدد (threaded).
- المبرمج يكون قادراً على الحصول على حالة انتهاء الخيط الهدف إذا كان محدداً عند استدعاء الروتين `pthread_exit()`.
- تتم عملية إنضمام الخيط (joining thread) مرة واحدة فقط (أي استدعاء الروتين `pthread_join()` مرة واحدة لنفس الخيط). ويعتبر خطأ منطقياً محاولة عمل أكثر من إنضمام (multiple joins) لنفس الخيط.
- هنالك طرق أخرى للتزامن مثل mutexes و المتغيرات الفرعية (condition variables).



شكل رقم 7-2: استخدام `pthread_join()` للتزامن [15].

هل الخيط قابل للإنضمام 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 أنشأنا الخيط كخيط منفصل (detached) فلا يمكن ضمه أبداً.
- في معايير POSIX الأخيرة تم تحديد أن الخيط يجب أن يكون من النوع القابل للضم (joinable) عند إنشائه.
- لتحديد حالة الخيط عند إنشائه هل سيكون joinable أو detached ، يمكننا استخدام القيمة المدخلة attr في الروتين `pthread_create()`.

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()`

## 7.9. الإنفصال (Detaching)

- يمكن استخدام الروتين `pthread_detach()` لجعل الخيط منفصل حتى ولو تم إنشائه منضم (joinable). وليس هنالك روتين عكسي.

تنبيه

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

أما إذا عرفت مسبقاً أن الخيط لن يحتاج إنضمام مع خيط آخر فيمكنك إنشائه في حالة المنفصل حيث يوفر ذلك تحرير بعض موارد النظام.

البرنامج 7-4 يوضح كيف يمكن لخيط أن ينتظر إنتهاء خيط آخر باستخدام الروتين `join`. حيث تم إنشاء الخيط من البداية بأنه قابل للإنضمام، حتى نستطيع ضمه مؤخراً.

```

* FILE: join.c
* DESCRIPTION:
* This example demonstrates how to "wait" for thread completions by using
* the Pthread join routine. Threads are explicitly created in a joinable
* state for portability reasons. Use of the pthread_exit status argument is
* also shown. Compare to detached.c
* AUTHOR: 8/98 Blaise Barney
* LAST REVISED: 01/30/09

#include "stdafx.h"
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_THREADS 4

void *BusyWork(void *t)
{
 float i; // changed from int to float
 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);
 return NULL; // added: not found before
}

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);
}

```

البرنامج 4-7

مخرج البرنامج:

```

Main: creating thread 0
Main: creating thread 1
Main: creating thread 2
Thread 0 starting...
Thread 1 starting...
Main: creating thread 3
Thread 2 starting...
Thread 3 starting...
Thread 3 done. Result = -3.153838e+006
Thread 1 done. Result = -3.153838e+006
Thread 0 done. Result = -3.153838e+006
Main: completed join with thread 0 having a status of 0
Main: completed join with thread 1 having a status of 1
Thread 2 done. Result = -3.153838e+006
Main: completed join with thread 2 having a status of 2
Main: completed join with thread 3 having a status of 3
Main: program completed. Exiting.
-

```

## 7.10. إدارة المكادسة Stack Management

الروتينات المستخدمة:

```

pthread_attr_getstacksize (attr,
stacksize)
pthread_attr_setstacksize (attr, stacksize)
pthread_attr_getstackaddr (attr,
stackaddr)
pthread_attr_setstackaddr (attr,
stackaddr)

```

الوقاية من مشاكل المكادسة :

- لم يتم تحديد سعة المكادسة في معايير POSIX، وإنما يختلف ذلك ويعتمد على التطبيق.
- من السهل الزيادة على حد المكادسة الافتراضي مما يسبب إخاء البرنامج أو تخريب البيانات.
- البرامج الآمنة والمتقللة يجب أن لا تعتمد على سعة المكادسة الافتراضية وإنما يجب جزء مساحة كافية للمكادسة لكل خيط باستخدام الروتين .pthread\_attr\_setstacksize

- الروتينات pthread\_attr\_setstackaddr و pthread\_attr\_getstackaddr يمكن استخدامهما في البيئات التي يجب فيها وضع مكبس الخيط في منطقة معينة بالذاكرة.
- البرنامج 7-5 تطبيق على إدارة المكداة، حيث تمت تجربة كيفية معرفة وتغيير حجم مكداة الخيط.

```
#include "stdafx.h"
#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);
 return 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 %d threads.\n", t);
 pthread_exit(NULL);
}
```

[5-7 برنامج](#)

مخرج البرنامج

```
Default stack size = 0
Amount of stack needed per thread = 9000000
Creating threads with stack size = 9000000 bytes
Created 4 threads.
Thread 1: stack size = 9000000 bytes
Thread 2: stack size = 9000000 bytes
Thread 0: stack size = 9000000 bytes
Thread 3: stack size = 9000000 bytes
```

|                                             |
|---------------------------------------------|
| <code>pthread_self()</code>                 |
| <code>pthread_equal(thread1,thread2)</code> |

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

### 7.11. متغيرات الميوتوكس (Mutex Variables)

- كلمة Mutex هي اختصار ل "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 المشتركة. والمفهوم من وراء فكرة المتغير mutex المستخدمة في Pthreads هو أن خيط واحد فقط هو من يغلق (يعتلk) متغير mutex في أي لحظة. أي إذا كان هناك عدد من الخيوط تحاول غلق mutex واحدة فقط هي التي ستنتج. ولن يستطيع أي خيط آخر غلق (إمتلاك) هذا المتغير (mutex) ما لم يتمكن الخيط الذي حصل عليه مسبقا.
- يمكن استخدام Mutexes في الوقاية من (منع) حدوث حالات سباق ("race" conditions).

جدول رقم 7-2: حالات سباق حدثت في حركة مالية بينك.

| Thread 1                    | Thread 2                    | Balance |
|-----------------------------|-----------------------------|---------|
| Read balance: \$1000        |                             | \$1000  |
|                             | Read balance: \$1000        | \$1000  |
|                             | Deposit \$200               | \$1000  |
| Deposit \$200               |                             | \$1000  |
| Update balance \$1000+\$200 |                             | \$1200  |
|                             | Update balance \$1000+\$200 | \$1200  |

- في الجدول 7-2 يجب عمل mutex لغلق الرصيد ("Balance") بينما يقوم الخيط باستخدام هذا البيانات المشتركة. المتغيرات التي تم تحريرها تتبع إلى مقطع حرج (critical section). الجدول 7-3 يوضح الفرق بين استخدام mutex وعدم استخدامه.

جدول 7-3: [16] Mutex

| Without Mutex               | With Mutex                                                                                                               |
|-----------------------------|--------------------------------------------------------------------------------------------------------------------------|
| <code>int counter=0;</code> | <code>/* Note scope of variable and mutex are the same */<br/>pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;</code> |

```

/* Function C */
void functionC()
{
 counter++
}

/* Function C */
void functionC()
{
 pthread_mutex_lock(&mutex1);
 counter++;
 pthread_mutex_unlock(&mutex1);
}

```

### Possible execution sequence

| Thread 1    | Thread 2    | Thread 1    | Thread 2                                                               |
|-------------|-------------|-------------|------------------------------------------------------------------------|
| counter = 0 | counter = 0 | counter = 0 | counter = 0                                                            |
| counter = 1 | counter = 1 | counter = 1 | Thread 2 locked out.<br>Thread 1 has exclusive use of variable counter |
|             |             |             | counter = 2                                                            |

خطوات استخدام mutex كما يلي:

- إنشاء وتحية(mutex).
- عدة خيوط تحاول غلق الـ mutex.
- خيط واحد فقط ينجح في امتلاك الـ mutex.
- ينفذ الخيط المالك للـ mutex التعامل مع البيانات.
- يترك الخيط المالك الـ mutex.
- يحصل خيط آخر على الـ mutex بؤدي عملا ما.
- في النهاية يتم تدمير الـ mutex.
- عندما تتنافس عدة خيوط على mutex، تظل الخيوط التي لم تحصل على الـ mutex محجوزة في ذلك الوضع وتستدعي طلب الحجز بالأمر "trylock" بدلاً من استدعائه بالأمر "lock".
- عند حماية بيانات مشتركة فإن مهمة المبرمج التأكد من أن كل خيط يحتاج استخدام الـ mutex قد فعل. فمثلاً لو كان لدينا ثلات خيوط تريد تعديل نفس البيانات، ولكن خيط واحد فقط هو من استخدم الـ mutex، فستظل البيانات مخربة ولا فائدة من استخدام الـ mutex.

Mutexes إنشاء وتدمير الـ

الروتينات المستخدمة:

```

pthread_mutex_init
(mutex,attr)
pthread_mutex_destroy
(mutex)
pthread_mutexattr_init (attr)
pthread_mutexattr_destroy
(attr)

```

طريقة الاستخدام:

- يجب الإعلان عن المتغيرات الـ `Mutex` بال النوع `pthread_mutex_t`، ثم تعيينها قبل استخدامها. هنالك طريقتين 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 كائناً

`.attr` وهي `mutex`

عند التهيئة يكون المتغير `mutex` غير محجوز (unlocked).

غلق وفتح الـ `Mutexes`

الروتينات المستخدمة:

|                                         |
|-----------------------------------------|
| <code>pthread_mutex_lock (mutex)</code> |
| <code>pthread_mutex_trylock</code>      |
| <code>(mutex)</code>                    |
| <code>pthread_mutex_unlock</code>       |
| <code>(mutex)</code>                    |

طريقة الاستخدام:

- الروتين (`pthread_mutex_lock()`) يستخدم بواسطة الخيط للحصول على المتغير `mutex` (المغلق). إذا كان الـ `mutex` مغلق بخيط آخر سيتم حجز هذا النداء حتى يتم فتح (unlock) المتغير `mutex`.
- الروتين (`pthread_mutex_trylock()`) يحاول غلق المتغير `mutex`، لكنه يعطي عبارة الخطأ `busy` مباشرةً إذا وجد المتغير `mutex` مغلق. يكون هذا الروتين مناسب لتجنب شرط من شروط الاختناق (deadlock).
- الروتين (`pthread_mutex_unlock()`) يفتح الـ `mutex` إذا استدعي بواسطة الخيط المالك للـ `mutex`. وهو مطلوب استدعاؤه بعد أن يتنهي الخيط من استخدام البيانات المحمية. وقد يحدث خطأ في الحالات التالية:
  - إذا الـ `mutex` مفتوح (unlocked).
  - إذا كان الـ `mutex` مملوك لخيط آخر.

عندما يكون هنالك أكثر من خيط يتنتظر غلق `mutex`، فإني واحد من هذه الخيوط سيحصل على الغلق بعد أن يصبح متاحاً؟

هنا في هذا البرنامج نوضح كيفية استخدام متغيرات `mutex` في برنامج متعدد الخيوط. البيانات الرئيسية متاحة لكل الخيوط عبر بيئة عامة. كل خيط يعمل على جزء من البيانات. الخيط الرئيسي سيتظر كل الخيوط لتنهي عملياتها ثم يطبع نتيجة النهاية.

```
#include "stdafx.h"
#include <pthread.h>
#include <stdio.h>
#include <math.h>

#define ITERATIONS 10000

// A shared mutex
pthread_mutex_t mutex;
double target;

void* opponent(void *arg)
```

```

{
 for(int i = 0; i < ITERATIONS; ++i)
 {
 // Lock the mutex
 pthread_mutex_lock(&mutex);
 target -= target * 2 + tan(target);
 // Unlock the mutex
 pthread_mutex_unlock(&mutex);
 }

 return NULL;
}

int main(int argc, char **argv)
{
 pthread_t other;

 target = 5.0;

 // Initialize the mutex
 if(pthread_mutex_init(&mutex, NULL))
 {
 printf("Unable to initialize a mutex\n");
 return -1;
 }

 if(pthread_create(&other, NULL, &opponent, NULL))
 {
 printf("Unable to spawn thread\n");
 return -1;
 }

 for(int i = 0; i < ITERATIONS; ++i)
 {
 pthread_mutex_lock(&mutex);
 target += target * 2 + tan(target);
 pthread_mutex_unlock(&mutex);
 }

 if(pthread_join(other, NULL))
 {
 printf("Could not join thread\n");
 return -1;
 }

 // Clean up the mutex
 pthread_mutex_destroy(&mutex);

 printf("Result: %f\n", target);

 return 0;
}

```

[برنامـج 6-7 من](http://pages.cs.wisc.edu/~travitch/pthreads_primer.html)

[http://pages.cs.wisc.edu/~travitch/pthreads\\_primer.html](http://pages.cs.wisc.edu/~travitch/pthreads_primer.html)

مخرج البرنامج:

Result: -1.#IND00

## 7.12. السيمافور

```
#include "stdafx.h"
#include <semaphore.h>
```

```

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

#define THREADS 20

sem_t OKToBuyMilk;
int milkAvailable;

void* buyer(void *arg)
{
 // P()
 sem_wait(&OKToBuyMilk);
 if(!milkAvailable)
 {
 // Buy some milk
 ++milkAvailable;
 }
 // V()
 sem_post(&OKToBuyMilk);

 return NULL;
}

int main(int argc, char **argv)
{
 pthread_t threads[THREADS];

 milkAvailable = 0;

 // Initialize the semaphore with a value of 1.
 // Note the second argument: passing zero denotes
 // that the semaphore is shared between threads (and
 // not processes).
 if(sem_init(&OKToBuyMilk, 0, 1))
 {
 printf("Could not initialize a semaphore\n");
 return -1;
 }

 for(int i = 0; i < THREADS; ++i)
 {
 if(pthread_create(&threads[i], NULL, &buyer, NULL))
 {
 printf("Could not create thread %d\n", i);
 return -1;
 }
 }

 for(int i = 0; i < THREADS; ++i)
 {
 if(pthread_join(threads[i], NULL))
 {
 printf("Could not join thread %d\n", i);
 return -1;
 }
 }

 sem_destroy(&OKToBuyMilk);

 // Make sure we don't have too much milk.
 printf("Total milk: %d\n", milkAvailable);

 return 0;
}

```

[http://pages.cs.wisc.edu/~travitch/pthreads\\_primer.html](http://pages.cs.wisc.edu/~travitch/pthreads_primer.html) . من 7- البرنامج مخرج:

Total milk: 1

## 7.13. المتغيرات الشرطية (Condition Variables)

- تعتبر المتغيرات الشرطية نوع آخر يستخدم في تزامن الخيوط (synchronization) (mutexes). بينما يعتمد التزامن على التحكم في وصول الخيط للبيانات، يعتمد التزامن في المتغيرات الشرطية على القيمة الفعلية للبيانات.
- بدون متغيرات شرطية يحتاج المبرمج لخيوط تختبر باستمرار (polling) هل تم تحقق الشروط أم لا ، وهذا قد يهدى الموارد ويستهلكها لأن الخيط سيكون دوماً مشغولاً بهذه الأمور. المتغير الشرطي يسمح لنا التأكد من تتحقق الشرط دون الحاجة لـ polling . المتغيرات الشرطية تعمل مع غلقmutex .
- طريقة استخدام المتغيرات الشرطية موضحة أدناه:

```
pthread_cond_init (condition,attr)
pthread_cond_destroy (condition)
pthread_condattr_init (attr)
pthread_condattr_destroy (attr)
```

إنشاء وتدمير المتغيرات الشرطية

طريقة الاستخدام:

- يجب الإعلان عن المتغيرات الشرطية بال النوع `t` `pthread_cond_t`، ويجب تحديتها قبل استخدامها. وهناك طريقتين للتهيئة المتغيرات هما:

1. ساكنة (Statically)، عند الإعلان عنها، مثلا:

```
pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;
```

2. ديناميكية (Dynamically) باستخدام الروتين (`pthread_cond_init()`). قيمة تعريف المتغير الشرطي (`condition` parameter) الذي ينشأ ترجع للخيط الذي يستدعىها عبر المدخل (`ID`). ويمكن إعداد صفات كائن المتغير الفرعى `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 المتغير الشرطي الذي لا يحتاج إليه.

```
pthread_cond_wait (condition,mutex)
pthread_cond_signal (condition)
pthread_cond_broadcast (condition)
```

الإنتظار والتأشير في المتغيرات الفرعية

طريقة الاستخدام:

- الروتين (`pthread_cond_wait()`) يحجز الخيط المستدعى (calling thread) حتى يتحقق الشرط المحدد (`condition` is signaled). هذا الروتين يجب استدعائه عندما يكون `mutex` مغلق، ويحرر `mutex` تلقائياً بينما هو منتظر. بعد استلام الاشارة واستيقاظ الخيط، سيغلق `mutex` تلقائياً ويستخدم بواسطة الخيط. وسيكون المبرمج مسؤولاً عن فتح `mutex` عندما يتنهى الخيط منه.

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

تنبيه

- غلق وفتح المتغير `mutex` بصورة حكيمية ضروري عند استخدام هذه الروتينات. مثلاً:
- فشل غلق لا `mutex` قبل استدعاء `pthread_cond_wait()` قد يمنعه من الحجز.
  - الفشل في فتح (`unlock`) لا `mutex` بعد استدعاء (`pthread_cond_signal()`) قد لا يسمح للروتين `pthread_cond_wait()` الذي يعمل معها بالاكتمال (يظل محجوز).

مثال برمجي (استخدام المتغيرات الشرطية)

في هذا المثال نقوم بإنشاء ثلاثة خيوط، اثنين من هذه الخيوط يقومان بتحديث المتغير `count`. الخيط الثالث يتنتظر حتى يصل المتغير `COUNT` قيمة معينة.

```

* FILE: condvar.c
* DESCRIPTION:
* Example code for using Pthreads condition variables. The main thread
* creates three threads. Two of those threads increment a "count" variable,
* while the third thread watches the value of "count". When "count"
* reaches a predefined limit, the waiting thread is signaled by one of the
* incrementing threads. The waiting thread "awakens" and then modifies
* count. The program continues until the incrementing threads reach
* TCOUNT. The main program prints the final value of count.
* SOURCE: Adapted from example code in "Pthreads Programming", B. Nichols
* et al. O'Reilly and Associates.
* LAST REVISED: 10/14/10 Blaise Barney

#include "stdafx.h"
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

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

int count = 0;
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);
 }
 }
}

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

 for (i=0; i < TCOUNT; i++) {
 pthread_mutex_lock(&count_mutex);
 if (count == COUNT_LIMIT) {
 pthread_cond_signal(&count_threshold_cv);
 }
 pthread_mutex_unlock(&count_mutex);
 }
}
```

```

 printf("inc_count(): thread %ld, count = %d Threshold reached. ",
 my_id, count);
 pthread_cond_signal(&count_threshold_cv);
 printf("Just sent signal.\n");
 }
 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);
return 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);
 while (count < COUNT_LIMIT) {
 printf("watch_count(): thread %ld Count= %d. Going into wait...\n",
my_id, count);
 pthread_cond_wait(&count_threshold_cv, &count_mutex);
 printf("watch_count(): thread %ld Condition signal received. Count= %d\n",
my_id, count);
 printf("watch_count(): thread %ld Updating the value of count...\n",
my_id, count);
 count += 125;
 printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
 }
 printf("watch_count(): thread %ld Unlocking mutex.\n", my_id);
 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 and joined with %d threads. Final value of count = %d.
Done.\n",
 NUM_THREADS, count);

 /* Clean up and exit */
}

```

```

pthread_attr_destroy(&attr);
pthread_mutex_destroy(&count_mutex);
pthread_cond_destroy(&count_threshold_cv);
pthread_exit (NULL);

}

```

برنامـج 8-7

مخرج البرنامج

```

Starting watch_count(): thread 1
inc_count(): thread 2, count = 1, unlocking mutex
inc_count(): thread 2, count = 2, unlocking mutex
inc_count(): thread 2, count = 3, unlocking mutex
inc_count(): thread 2, count = 4, unlocking mutex
inc_count(): thread 2, count = 5, unlocking mutex
inc_count(): thread 2, count = 6, unlocking mutex
inc_count(): thread 2, count = 7, unlocking mutex
inc_count(): thread 2, count = 8, unlocking mutex
inc_count(): thread 2, count = 9, unlocking mutex
inc_count(): thread 2, count = 10, unlocking mutex
watch_count(): thread 1 Count= 10. Going into wait...
inc_count(): thread 3, count = 11, unlocking mutex
inc_count(): thread 3, count = 12 Threshold reached. Just sent signal.
inc_count(): thread 3, count = 12, unlocking mutex
inc_count(): thread 3, count = 13, unlocking mutex
inc_count(): thread 3, count = 14, unlocking mutex
inc_count(): thread 3, count = 15, unlocking mutex
inc_count(): thread 3, count = 16, unlocking mutex
inc_count(): thread 3, count = 17, unlocking mutex
inc_count(): thread 3, count = 18, unlocking mutex
inc_count(): thread 3, count = 19, unlocking mutex
inc_count(): thread 3, count = 20, unlocking mutex
watch_count(): thread 1 Condition signal received. Count= 20
watch_count(): thread 1 Updating the value of count...
watch_count(): thread 1 count now = 145.
watch_count(): thread 1 Unlocking mutex.
Main(): Waited and joined with 3 threads. Final value of count = 145. Done.

```

## 7.14 ملخص

يعتبر هذا الباب مختص في برمجة المتقدمة للخيوط باستخدام Pthreads. وقد تطرقنا في الباب على كيفية حماية البيانات المشتركة من الوصول المتزامن من قبل الخيوط التي تعمل معاً وذلك باستخدام mutex و المتغيرات الشرطية.

## الباب الثامن البرمجة باستخدام OpenMP

لقد استخدمنا Pthreads كمكتبة لبرمجة الخيوط وقبلها تعلمنا كيف نكتب برامج متعددة الخيوط معتمدة على لغة جافا. سنتعلم هذه السلسلة المتعلقة بالخيوط في هذا الباب، حيث ستتعلم كيف نكتب برنامج متعدد الخيوط باستخدام مكتبة OpenMP والتي تعتبر بيئة لبرمجة الخيوط تعتمد على موجهات المترجمات (compiler directives). وختلف عن برمجة الخيوط التي تعلمناها مسبقاً في أن OpenMP هنا هي التي تنشيء الخيوط وليس المبرمج، وهذا يجعل برمجة الخيوط أكثر سهولة ويسراً.

### 8.1. نبذة عن OpenMP

تعتبر OpenMP أحد الطرق المستخدمة لتوفير بيئة للبرمجة على الأجهزة ذات الذاكرة المشتركة، حيث توفر واجهة تطبيقات برمجية (API) للبرمجة المتوازية على الحاسوبات متعددة المعالجات (multiprocessors) والحاصلات متعددة النواة (multi-core). وهي تتكون من مجموعة من موجهات المترجمات (compiler directives) ومكتبة من الدوال. إنتشرت OpenMP لسهولة استخدامها وإمكانية عملها مع العديد من اللغات مثل C، C++، و Fortran. حيث يمكن استخدامها مع هذه اللغات، أيضاً بجانبها ضمن Visual studio .Net. قبل الدخول في تفاصيل openMP، سلقي نظرة على البرنامج البسيط رقم 6-1.

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran on many architectures, including Unix and Microsoft Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.  
From <http://berenger.eu/blog/c-cpp-openmp-vs-pthread-openmp-or-posix-thread/>

### 8.2. أول برنامج

```
#include "stdafx.h"
#include <stdio.h>
#include "omp.h"
Void main()
{
 int E=5;
 #pragma omp parallel/
 {
 printf("E is equal to %d\n",E);
 }
}
```

برنامج 6-1

هذا البرنامج مكتوب بلغة فيجوال C++ مع OpenMP. في معالج يتكون من 4 نواة سيكون مخرج البرنامج كالتالي:

```
E is equal to 5
Press any key to continue . . .
```

اختلاف هذا البرنامج عن برمج فيجوال C++ العادي في سطرين:

```
#include "omp.h"
```

و

```
#pragma omp parallel
```

السطر الأول هو لتضمين ترويسة OpenMP، والثاني هو موجه يستخدم لانشاء فقرة متوازية (parallel section) عدد من الخيوط (بما عدد الخيوط).

```
#include "stdafx.h"
#include <stdio.h>
#include "omp.h"
Void main()
{
 #pragma omp parallel
 {
 int E = omp_get_thread_num();
 printf("E is equal to %d\n", E);
 }
}
```

برنامـج 2-6

الدالة `omp_get_thread_num()` في البرنامج تعطينا رقم الخيط. لذلك سيكون مخرج البرنامج 2-6 كما يلي:

```
E is equal to 0
E is equal to 1
E is equal to 2
E is equal to 3
Press any key to continue . . .
```

عدد الخيوط الافتراضية عند استدعاءك للامر `#pragma omp parallel` سيكون عدد الخيوط بعدد المعالجات (أو النواة) المتاحة في نظامك.

إذا أردت إنشاء عدد معين من الخيوط يمكنك ذلك بالأمر:

```
omp_set_num_threads(n);
```

حيث `n` تمثل عدد الخيوط.

مثلا لو عدلنا البرنامج 2-6 بإضافة الأمر الذي يحدد عدد الخيوط، ليصبح 3-6.

```
#include "stdafx.h"
#include <stdio.h>
#include "omp.h"
Void main()
{
 omp_set_num_threads(7);
 #pragma omp parallel
 {
 int E = omp_get_thread_num();
 printf("E is equal to %d\n", E);
 }
}
```

البرنامـج 3-6

سيكون المخرج كما يلي:

```
E is equal to 4
E is equal to 0
E is equal to 3
E is equal to 5
E is equal to 1
E is equal to 2
E is equal to 6
Press any key to continue . . .
```

حيث أصبح لدينا 7 خيوط، وهي التي تم تحديدها في البرنامج 6-3 بالامر `omp_set_num_threads(7)` يمكننا استبدال الامرین:

```
omp_set_num_threads(7);
```

```
#pragma omp parallel
```

بالامر:

```
#pragma omp parallel num_threads(7)
```

```
#include "stdafx.h"
#include <stdio.h>
#include "omp.h"
Void main()
{
 #pragma omp parallel num_threads(9)
 {
 int E = omp_get_thread_num();
 printf("E is equal to %d\n", E);
 }
 printf("After threads execution...");
```

البرنامج 6-4

البرنامج 6-4 ينفذ 9 خيوط بالتوازي. ثم بعد أن يكتمل تنفيذه، ينفذ الامر التالي:

```
printf("After threads execution...");
```

المخرج سيكون كما يلي:

```
E is equal to 6
E is equal to 5
E is equal to 0
E is equal to 4
E is equal to 7
E is equal to 1
E is equal to 3
E is equal to 2
E is equal to 8
After threads execution..
```

نلاحظ أن تنفيذ الخيوط قد يكون غير مرتب ويختلف من مرة إلى أخرى. مثلاً فيما يلي مخرج آخر لنفس البرنامج:

```
E is equal to 0
E is equal to 6
E is equal to 4
E is equal to 1
E is equal to 7
E is equal to 2
E is equal to 3
E is equal to 8
E is equal to 5
After threads execution..
```

## لماذا يختلف مخرج البرنامج من مرة لأخرى؟

تنبيه

- الامر `int ID = omp_get_thread_num();` يعطينا رقم تعريف الخيط.
  - الامر `int nthrds = omp_get_num_threads();` يعطينا عدد الخيوط.
- نلاحظ هذا في البرنامج 5-6.

```
#pragma omp parallel num_threads(9)
{
 int nthrds = omp_get_num_threads();
 int E = omp_get_thread_num();
 printf("E is equal to %d, %d\n", E, nthrds);
}
printf("After threads execution...");
```

برنامـج 5-6 .

مخرج البرنامج اعلاه سيكون:

```
E is equal to 0, 9
E is equal to 6, 9
E is equal to 4, 9
E is equal to 3, 9
E is equal to 2, 9
E is equal to 5, 9
E is equal to 1, 9
E is equal to 7, 9
E is equal to 8, 9
After threads execution..
```

هنا لك دالة تستخدم لمعرفة عدد المعالجات الموجودة في جهازك هي: `(omp_get_num_procs()` .  
إذا أضفناها الى البرنامج ومع قليل من التعديل يصبح شكل البرنامج كما في 6-6 .

```
#pragma omp parallel num_threads(9)
{
int nprocs = omp_get_num_procs();
int nthrds = omp_get_num_threads();
int ID = omp_get_thread_num();
printf("Thread no%d of total no of threads%d, no of processors=%d\n", ID, nthrds,
nprocs);
}
printf("After threads execution...");
```

البرنامـج 6-6 .

مخرج البرنامج سيكون كما يلي:

```
Thread no 0 of total no of threads 9, no of processors = 4
Thread no 3 of total no of threads 9, no of processors = 4
Thread no 4 of total no of threads 9, no of processors = 4
Thread no 2 of total no of threads 9, no of processors = 4
Thread no 1 of total no of threads 9, no of processors = 4
Thread no 5 of total no of threads 9, no of processors = 4
Thread no 6 of total no of threads 9, no of processors = 4
Thread no 7 of total no of threads 9, no of processors = 4
Thread no 8 of total no of threads 9, no of processors = 4
After threads execution...Press any key to continue . . .
```

حيث نلاحظ أن العدد الكلي للخيوط 9، عدد المعالجات (النواة) 4.

### 8.3. تحديد عدد الخيوط

عدد الخيوط التي تريده يعتمد على عدد المعالجات لديك. إنشاء الخيوط وتدميره بعد تجميع النتائج لا يستغرق وقتا طويلا. إذا كانت المعالجات قليلة وعدد الخيوط أكبر سيستغرق التنفيذ وقتا أطول. لأن هناك زمن إضافي يضيع في إنشاء الخيوط والانتقال بينها. لأن هذا الانتقال يتطلب عمليات حفظ وتحميل بياناته من وإلى الذاكرة.

القاعدة العامة هي بما أن كل الخيوط تعمل على عملية واحدة فالأولوية واحدة، وبالتالي يمكن تشغيل خيط واحد في كل معالج وكلما زادت عدد المعالجات زيد عدد الخيوط.

معظم موجهات المترجمات في OpenMP تستخدم المتغير البيئي (environment variable) المسمى OMP\_NUM\_THREADS لتحديد عدد الخيوط التي تريده إنشاءها.

يمكنك إنشاء خيوط بعدد المعالجات بالخطوات التالية:

```
int nProc = omp_get_num_procs();
omp_set_num_threads(nProc);
```

يمكن وضع أي قيمة في المتغير nProc إذا كنت لا تريدين استخدام الدالة mp\_get\_num\_procs(). ويمكنك استدعاء الدالة omp\_set\_num\_threads كاما تريدين في أجزاء عديدة من شفرتك. إذا لم تستخدم هذه الدالة سيسخدم OpenMP المتغير البيئي OMP\_NUM\_THREADS.

جدول 6-1: بعض دوال OpenMP التي تم استخدامها في البرامج السابقة.

|                                                                    |                     |
|--------------------------------------------------------------------|---------------------|
| تعطينا عدد المعالجات الموجودة في الحاسوب الذي ينفذ هذا الخيط       | omp_get_num_procs   |
| تعطينا عدد الخيوط الفعالة في هذه المنطقة (current region)          | omp_get_num_threads |
| يعطينا الرقم التعريفي للخيط (thread identification number)         | omp_get_thread_num  |
| تسمح لك بتحديد عدد الخيوط التي تريده تنفيذها في جزء معين من الشفرة | omp_set_num_threads |

### 8.4. تشارك البيانات

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 لنفس موقع الذاكرة (هناك عنوان حقيقي (فيزيائي) مشترك، مثلاً إذا كتب معالج القيمة 7 في العنوان 2020 فإن أي معالج آخر سيحصل على ذات القيمة إذا احضر محتوى الذاكرة ذات العنوان 2020. هذا التشارك يمكن المعالجات من تبادل البيانات عبر المتغيرات المشتركة.

من المعلوم أن أي برنامج يكون تحت التنفيذ (شغال) يسمى عملية (process)، ويقوم نظام التشغيل بمحجز مساحة للعملية بالذاكرة. وكل عملية تحتوي على الأقل على خيط واحد ويمكنها أن تحتوي على أكثر 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 ان تعمل معاً في نفس الوقت بالتزامن داخل العملية الواحدة.

اي متغير يتم تعريفه خارج المنطقة المتوازية يعتبر متغير مشترك بين كل الخيوط التي سيتم إنشاؤها داخل المنطقة المتوازية. أما إذا عرفنا متغير داخل المنطقة المتوازية فستكون هناك نسخة من هذا المتغير لكل خيط خاص به ويختلف محتواها عن بقية النسخ الموجودة في الخيوط الأخرى. مثلاً لو تم إنشاء اربعة خيوط وكان هناك متغير واحد محلي داخل المنطقة المتوازية فسيكون لدينا اربعة نسخ من هذا المتغير، لكل خيط نسخته الخاصة به، انظر البرنامج 6-7.

```

#include "stdafx.h"
#include <stdio.h>
#include "omp.h"
void main()
{
 int x = 78; // مشترك
 #pragma omp parallel
 {
 int ID; // محلي
 ID = omp_get_thread_num();
 printf("x = %d, inside thread no. ID = %d \n", x, ID);
 }
 printf("x after parallel region = %d\n ", x);
}

```

البرنامج 7-6.

مخرج البرنامج 7-6 سيكون كما يلي:

```

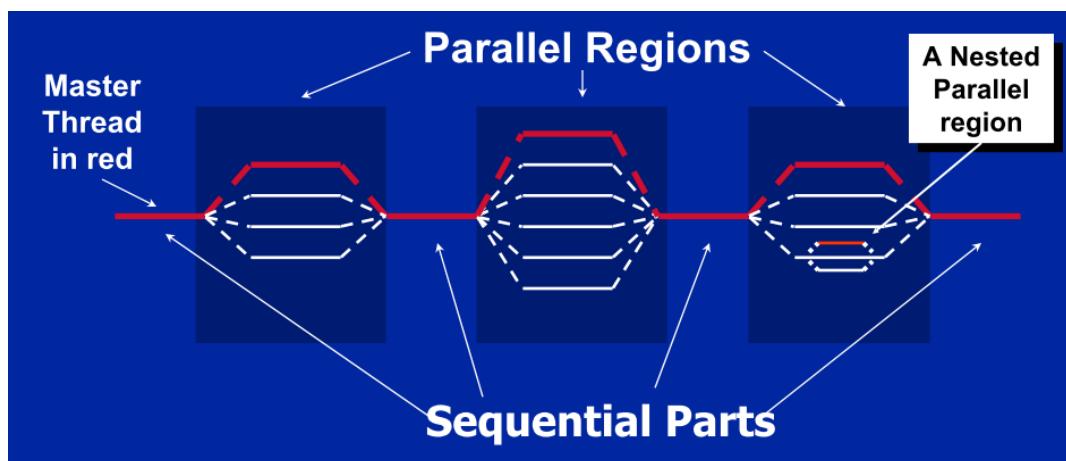
x = 78, inside thread no. ID = 0
x = 78, inside thread no. ID = 1
x = 78, inside thread no. ID = 3
x = 78, inside thread no. ID = 2
x after parallel region = 78
Press any key to continue . . .

```

يعتبر المتغير X متغير مشترك بين كل الخيوط لأننا انشأناه خارج المنطقة المتوازية. بينما المتغير ID هو متغير محلي انشئ داخل المنطقة المتوازية ولكن خيط نسخته الخاصة به وختلف قيمتها عن بقية الخيوط.

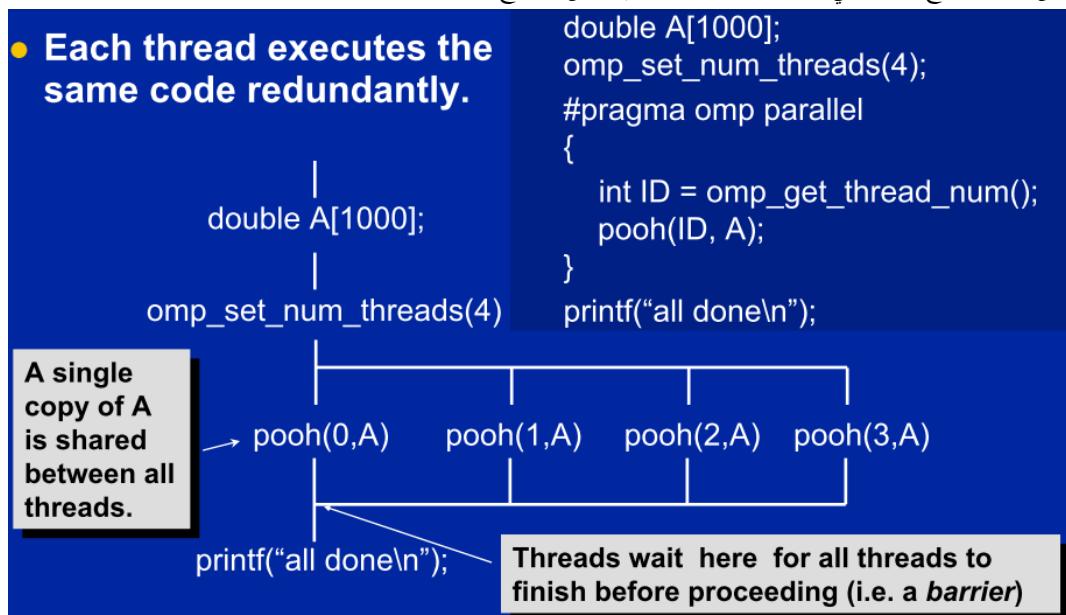
## 8.5. تفريخ الخيوط (Fork/Join)

يمكنك انشاء اكثرا من فقرة متوازية في برنامج OpenMP. عندما تنشئ اكثرا من خيط تسمى هذه العملية التفريخ (fork)، وعندما يرجع البرنامج للتنفيذ التسلسلي العادي يسمى Join، شكل 6-1. ونلاحظ من الشكل أن التفريخ يمكن أن يكون متداخل (يقوم خيط مفرخ بتفريخ خيوط أخرى).



شكل رقم 6-1: موازاة [17]

الشكل 6-2 يوضح مثال برمجي لإنشاء متوازي داخل البرنامج.



شكل رقم 6-2: منطقة توازي (Parallel Region) . [17]

الخيط الرئيسي يعمل تسلسلياً إلى أن يجد أمر تفريح لتوليد خيوط جديدة. هذه الخيوط المولدة يتم توزيعها على عدد من المعالجات لتنفذ بالتوازي، ثم تجمع النتائج. يمكن توليد خيوط بعدد المعالجات الموجودة بالجهاز. ويستطيع المستخدم تحديد عدد الخيوط التي يريدتها في التغير البيئي OMP\_NUM\_THREADS. أو باستخدام الروتين:

call omp\_set\_num\_threads.

## 8.6. توزيع الحمل (Load Balancing)

من أهم المواضيع في التنفيذ المتوازي هو توزيع العبء بين المعالجات. حيث يجب توزيع المهام بين المعالجات بالتساوي، بحيث يكون (تقريباً) لكل معالج جزء متساوي من العمل. توزيع العبء بين المعالجات يجعل كل الخيوط متساوية في زمن التنفيذ (لأن كل خيط سينفذ في معالج). وهو ما يسمى جدولة الخيط (thread scheduling). إفتراضياً إذا انتهى خيط قبل بقية الخيوط في منطقة شفرة (region of code) فسينتظر بقية حتى تنتهي. وتعتبر هذه الطريقة غير جيدة لأنها سيكون هناك معالجات لا تعمل (غير مستفاد منها)، لأن عملها إكتمل قبل المعالجات الأخرى. ولحل هذه المشكلة توفر لنا OpenMP خيارات مختلفة لجدولة الخيط. أحد هذه الخيارات المتاحة في تكرار الحلقة (for loop) هي:

1. كل الخيط ينفذ (n) حلقة وينتظر بقية الخيوط إلى أن تكمل (n) حلقة. (جدولة ساكنة (static scheduling).
2. الجدولة الديناميكية (dynamic scheduling): نسند (n) حلقة من الحلقات التي لم تنفذ للخيوط العاطلة (idle).
3. Guided scheduling: كل خيط ينتهي من عمله نسند إليه عدد حلقات يساوي عدد الحلقات الباقية مقسوماً على عدد الخيوط.

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

## 8.7. عقبات في برمجة الخيط

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

| T1                                 | T2                                |
|------------------------------------|-----------------------------------|
| Void foo(){<br>x++;<br>y = x;<br>} | Void foo(){<br>y++;<br>x+=3;<br>} |

إفترض أن  $y=0$  و  $x=6$ ، ما قيمة  $x$  و  $y$  بعد تنفيذ الخيطين T1 و T2؟ حيث  $x$  و  $y$  هما متغيران مشتركان بين الخيوط أعلاه.

هناك العديد من النتائج المحتملة:

فقد يكون 7، 9، 10 للمتغير  $x$  وقد يكون 1، 7، 10 للمتغير  $y$  ولا يندر أي تباديل ستنتهي من هذه القيم. هذه المشكلة (أعلاه) ناجمة من أن هناك خيطان ينفذان شفرة تعرف بالمنطقة الحرجة (critical section). أحد طرق التزامن المستخدمة لحل هذه المشكلة هو عملية الغلق (lock). عندما يقوم خيط بعملية الغلق في المنطقة الحرجة سيمتع بقية الخيوط من تنفيذ شفرات على هذه المنطقة. والخيط الذي يريد أن يعمل غلق عليه الإنتظار حتى يحرر الغلق (أي لا يكون هناك أي خيط آخر عمل غلق).

معالجة مشكلة حالة السباق (race condition) قد تقودنا إلى مشكلة جديدة هي الإختناق (deadlock). وينتج الإختناق

إذا:

- قام الخيط T1 بغلق لمنطقة الحرجة S2.
- قام الخيط T2 بعملية غلق لمنطقة الحرجة S1

ثم أحتج الخيط T1 إلى استخدام المنطقة الحرجة S1 وفي نفس الوقت أراد الخيط T2 استخدام المنطقة الحرجة S2، سيظل كل خيط في إنتظار الخيط الآخر أن يتراكم منطقته الحرجة ليستخدمها، وسيظل الأمر هكذا، مما يسبب إختناق.

## 8.8. تزامن الخيط في OpenMP

توفر OpenMP إمكانيات للتزامن تمكن المبرمج من تجاوز مشاكل تشارك البيانات. فيمكنك وضع البيانات المراد تنظيم الوصول إليها داخل منطقة حرجة (#pragma critical). وتنظيم الوصول إليها بحيث يسمح لخيط واحد فقط بالدخول على هذه المنطقة في اللحظة الواحدة فيما يسمى mutual exclusion.

يجب تجنب اعتماد بيانات خيط على بيانات خيط آخر (البرنامج 7-7) قد يعطي نتائج خطأ بسبب اعتمادية البيانات.

|                                                                                                         |                   |
|---------------------------------------------------------------------------------------------------------|-------------------|
| #pragma omp parallel for<br>for (i=2; i < 10; i++)<br>{<br>factorial[i] = i * factorial[i-1];<br>}<br>. | [18] البرنامج 7-7 |
|---------------------------------------------------------------------------------------------------------|-------------------|

تجنب تشارك البيانات، البرنامج 7-8 قد يعطي نتائج خاطئة لانه من المتوقع ان يعدل خيط في البيانات المشتركة بينما يكون هناك خيط قد اخذ نسخة قبل التعديل (عدم توافقية).

```
#pragma omp parallel for
for (i=0; i < 100; i++)
{
 temp = array[i];
 array[i] = do_something(temp);
}
```

برنامـج 7-8: تشارـك بـيانـات بين الخـيوط [18].

حل الاشكال في البرنامج 7-8 يمكننا تعريف المتغير temp كمتغير خاص (محلي). انظر البرنامج بعد التعديل في 7-9.

```
// This works. The variable temp is now private
#pragma omp parallel for
for (i=0; i < 100; i++)
{
 int temp; // variables declared within a
parallel construct
 // are, by definition, private
 temp = array[i];
 array[i] = do_something(temp);
}

///////////////
// This also works. The variable temp is declared private
#pragma omp parallel for private(temp)
for (i=0; i < 100; i++)
{
 temp = array[i];
 array[i] = do_something(temp);
}
```

برنامـج 7-9: تجـنب الاخـطاء السـابـقة [18].

## 8.9. الاختزال (Reductions)

اذا اردنا جمع مصفوفة اعداد، فيمكننا القيام بذلك كما يلي:

```
sum = 0;
for (i=0; i < 100; i++)
{
```

```

sum += array[i]; // this

variable needs to be shared to generate
//
the correct results, but private to avoid
//
race conditions from parallel execution
}

```

نلاحظ اننا نريد ان نجعل المتغير `sum` مشترك للحصول على نتائج صحيحة وفي نفس الوقت نريد ان يكون خاص لمنع الوصول المتعدد إليه وتجنب حالة السباق. حل مثل هذا الاشكال يمكننا استخدام الفقرة `reduction`. البرنامج 7-10 يوضح كيفية استخدامها.

```

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

```

برنامـج 7-10: استخـدام [18] reduction

بهذه الطريقة تستطيع OpenMP توفير نسخ خاصة من المتغير `sum` لكل خيط، وعندما يخرج اي خيط، يقوم بجمع كل القيم مع بعض ووضع الناتج في نسخة المتغير العامة. أي انه سيكون لدينا متغير محلي `sum` بكل خيط يخزن فيه النتيجة النهائية التي تحصل عليها، وتجميع كل القيم النهائية لكل الخيوط وبجمعها باستخدام العملية `+ نحصل على الناتج الأخير.`

Load balancing, the equal division of work among threads, is among the most important attributes for parallel application performance. Load balancing is extremely important, because it ensures that the processors are busy most, if not all, of the time. Without a balanced load, some threads may finish significantly before others, leaving processor resources idle and wasting performance opportunities [18].

## 8.10. جدولـة التكرار (الحلـقة) (Loop Scheduling)

يمكـنا استخـدام فـقرة الجـدولـة مع التـكرـار `for`. الصـيـغـةـ العـامـةـ هـذـاـ الـأـمـرـ تكونـ كـالتـالـيـ:

```
#pragma omp parallel for schedule(kind [, chunk size])
```

حيـثـ توـفـرـ OpenMPـ انـوـاعـ مـخـلـفـةـ مـنـ الجـدولـةـ، انـظـرـ الجـدولـ 7-2ـ.

جدـولـ 7-2: انـوـاعـ الجـدولـةـ [18]

| Kind    | Description                                                                                                                                                                                                                                                                                                                           |
|---------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| static  | Divides the loop into equal-sized chunks or as equal as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size. By default, chunk size is loop count / number of threads. Set chunk to 1 to interleave the iterations.                                |
| dynamic | Uses an internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue. By default, chunk size is 1. Be careful when using this scheduling hint because of the extra overhead required.                  |
| guided  | Similar to dynamic scheduling, but the chunk size starts off large and shrinks in an effort to reduce the amount of time threads have to go to the work-queue to get more work. The optional chunk parameter specifies the minimum size chunk to use. By default, the chunk size is approximately the loop count / number of threads. |
| runtime | Uses the OMP_SCHEDULE environment variable to specify which one of the three loop-scheduling types should be used. OMP_SCHEDULE is a string formatted exactly the same as would appear on the parallel construct.                                                                                                                     |

تمرين

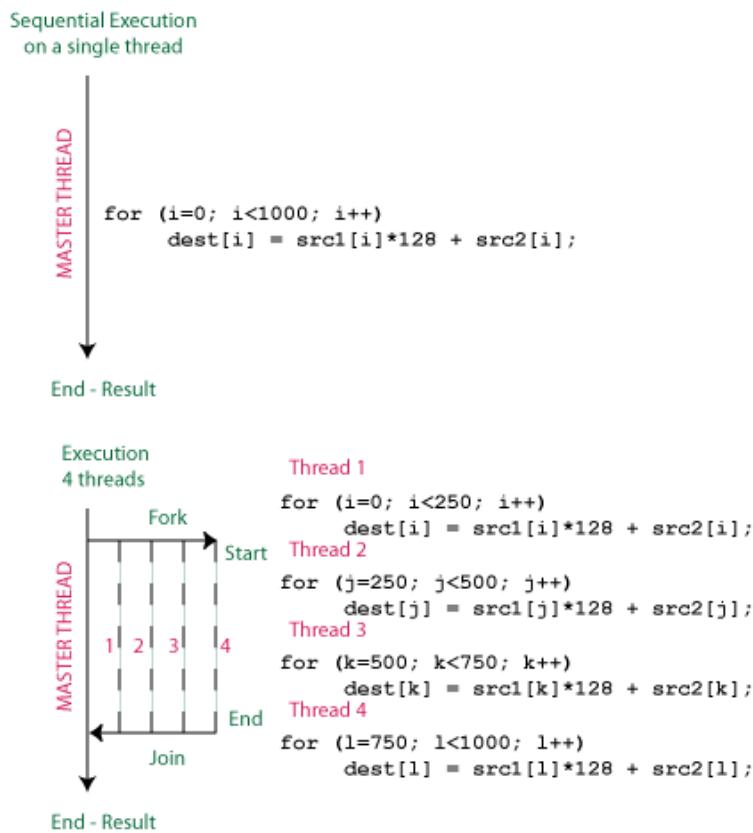
يُوفر OpenMP التوازي على مستوى المهام (task parallelism) وقد يستخدم حلقة loop لتوزيع المهام. البرنامج التالي يقوم بجمع أعداد مصنفة تسلسلياً.

```
const int numbers=1000;
int sum=0, total[numbers];
for (int x=0; x<numbers; x++)
 total[x] = x * 5;
for (int x=0; x<numbers; x++)
 sum = sum + total[x];
```

المطلوب تصميم برنامج لتنفيذ الشفرة أعلاه بالتوازي على حاسب به أربعة معالجات (نواة)، الشكل 6-3.

يمكنك توزيع المهام كالتالي:

1. تقسيم العملية (جمع 1000 عدد) إلى أربعة مهام صغيرة (جمع كل 250 عدد).
2. توزيع كل مهمة على خيط بالامر #pragma parallel for
3. تجميع النتيجة #pragma reduction(+:sum)



شكل رقم 7-3: جمع مصفوفة في اربعة خيوط [19].

تعتمد OpenMP على عملية التفريخ والتجميع كما ذكرنا مسبقا (fork/join) وهي كالتالي:

1. يبدأ البرنامج بخيط واحد رئيسي
2. يولد هذا الخيط الرئيسي عدد من الخيوط الفرعية (fork).
3. تعمل الخيوط المولدة في معالجات مختلفة بالموازي مع الخيط الرئيسي
4. إذا انتهى خيط سينتظر بقية الخيوط حتى تكتمل.
5. عندما تكتمل كل الخيوط ويتم إكمال العملية تجمع النتائج وتدمير كل الخيوط ما عدا الرئيسي.
6. عملية تقسيم الخيوط وتحميصها تحتوي ترامن تقوم به OpenMP للحصول على النتيجة الرئيسية.

#### تمرين

نفذ هذا البرنامج على بالطريقة المتسلسلة وباستخدام OpenMP مرة في حاسب ذو معالج واحد ومرة في حاسب متعدد النواة وقارن الأداء بين الأربع حالات. غير عدد الخيوط لاحظ الفرق في سرعة التنفيذ.

### 8.11. ملخص

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

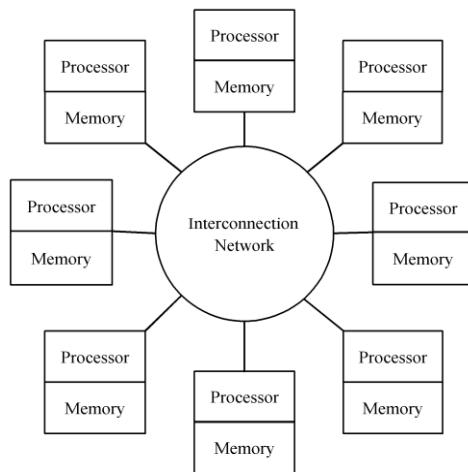
درستنا في هذا الباب كيف كتابة برامج متعددة الخيوط باستخدام مكتبة OpenMP، والتي يمكن تطبيقها عملياً على بيئة visual studio 2010. يوجد توضيح في الملحق عن كيف تفعيل OpenMP في مشروع فيجوال C++ والتي تأتي مدمجة في Visual studio 2010 .  
بيئة تطوير

## الباب التاسع

# برمجة الذاكرة الموزعة باستخدام MPI

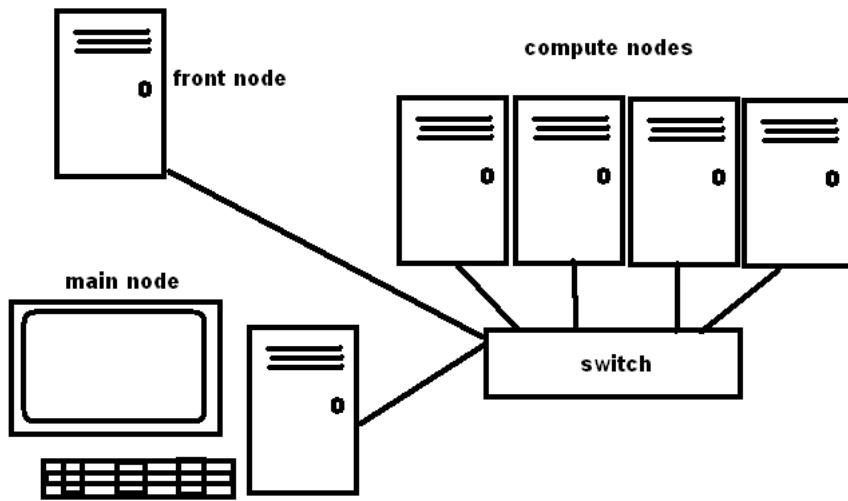
### 9.1. مقدمة

كل الأبواب السابقة ركزت على برمجة الذاكرة المشتركة وذلك باستخدام الخيوط سواء كان ذلك باستخدام لغة برمجة تدعم ذلك مثل الجافا أو مكتبة تضاف للغة برمجة مثل OpenMP، أو Pthreads. ولكن ماذا بشأن الحواسيب التي ليس لها ذاكرة مشتركة (الحواسيب ذات الذاكرة الموزعة). هذا الباب جاء لهذا الغرض. حيث ستحدث هنا عن برمجة الذاكرة الموزعة باستخدام تبادل الرسائل (Message Passing) وستستخدم مكتبة من أشهر المكتبات في هذا المجال إلا وهي مكتبة "واجهة تبادل الرسائل" والتي تسمى MPI اختصاراً للكلمات Message Passing Interface. معظم البرامج المستخدمة هنا في هذا الباب من كتاب Quinn الذي يتحدث عن البرمجة المتوازنة {Quinn, 2003 #29} ، فمن أراد المزيد يمكنه الرجوع لهذا الكتاب، فهو من أفضل الكتب التي قابلتني في هذا المجال.



شكل رقم 9-1: الذاكرة الموزعة .

ذكرنا في حديثنا عن المكونات المادية أن الأجهزة متعددة الحواسيب (multicomputer) هي حواسيب مرتبطة بعضها ولكن لكل حاسب ذاكرة خاصة. وإذا أرادت هذه الحواسيب التشارك في البيانات فسيتم ذلك عبر تبادل الرسائل. من أمثلة الأجهزة متعدد الحواسيب، الحاسوبات الفائقة السرعة (super computers)، وهي كما نعلم باهظة الثمن ومكلفة ولا يمكن لجهات عادية إمتلاكها. لذلك هنالك حل أقل كفاءة ولكنه أقل كلفة، حيث يمكن ربط مجموعة من الحواسيب الشخصية العادي بشبكة سريعة وتحتها تعمل كحاسوب فائق السرعة، وذلك بتثبيت بعض البرامج والتي من بينها واجهة تبادل الرسائل MPI. أحياناً نطلق على مثل هذا النوع من التجمع بالحاسوب الفائق السرعة الإفتراضي، 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 جداً. التجمع يفضل أن يكون في مكان واحد حيث تربط الأجهزة بشبكة محلية سريعة ويكون هنالك حاسب رئيسي أو أكثر للتعامل مع التجمع بينما تقوم بقية الأجهزة بتنفيذ ما يصلها من طلبات من الحاسب الرئيسي، غالباً لاحتاج شاشات للمحطات العاملة (أنظر الشكل 9-2).



شكل رقم 9-2: تجمع حاسوبات موزعة الذاكرة (cluster). [6]

أيضاً يمكنك تجربة 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 حاسوبات يستخدمها غيره وبالتالي سيستفيد من وقت تعطليها (عند ما لا يكون عليها أحد أو عندما يكون مستخدماً يكتب أو يتتصفح موقع ولا يحتاج للمعالج في كثير عمل).

عادةً يتكون التجمع من حاسوبات متعددة وثبت فيها نظم تشغيل متشاركة.

## 9.2. نبذة عن MPI

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

في أواخر الثمانينيات تم تطوير Parallel Virtual Machine (PVM) في المعهد 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 بستة دوال رئيسية منها وهي :

|               |
|---------------|
| MPI_INIT      |
| MPI_FINALIZE  |
| MPI_COMM_SIZE |
| MPI_COMM_RANK |
| MPI_SEND      |
| MPI_RECV      |

سنتحدث في بقية هذا الباب عن هذه الدوال ودوال أخرى مع توضيحها بأمثلة برمجية توضح مفاهيم البرمجة المتوازية.

### 9.3. أول برنامج

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

void main(int argc, char* argv[])
{
 MPI_Init(&argc, &argv);
 printf("Aslamu Alikum...");
 MPI_Finalize();
}
```

برنامـج 9-1

مخرج البرنامج:

Aslamu Alikum ...

#### شرح البرنامج 9-1 :

- الترويسة mpi.h : توفر تعريفات وانواع MPI الاساسية
- محيـة MPI: MPI\_Init لمستخدمها في البرنامج.
- MPI: تحرير الموارد التي حجزها برنامج MPI\_Finalize
- تعتـر كل الروتينات الأخرى (غير روـتينات MPI) محلية وتتفـد نسخـة في كل عملية ، مثلاـ printf تطبع رسالة في كل عملية . دوال MPI تعطي MPI\_SUCCESS (return) في حالة تنفيذـها بنجـاح، أما إذا لم تنجـح فستعطي رسالة خطـأ.
- افتراضـياً أي خطـأ يتسبـب في توقـيف كل العمـليات (abort) .
- . المستـخدم يمكن أن يكون لديه روـتينات تناول الاخطـاء الخاصة به (error handling routines)
- هـنالك custom error handlers متـاحة على الانترنت ويمكن تحمـيلها من هـنـاك.

#### شرح دوال MPI المستـخدمـة في البرنامج أعلاه

- روـتين محـيـة MPI\_Init هو أول روـتين يجب استـدعـائيـه في بـرامـج MPI ويـستـدـعـي مـرة وـاحـدة فقط، الصـيـغـةـ العـامـةـ لهذاـ الروـتينـ هيـ:
- MPI\_Init(&argc, &argv);
- روـتين الإـنـهـاء MPI\_Finalize هو آخر روـتين يتم استـدعـائـه وهو يـقـوم بـتحـريـرـ المـوارـدـ التي تمـ حـجزـهاـ بواسـطـةـ البرـنـامـجـ مثلـ الـذاـكـرـةـ. صـيـغـةـ هـذـاـ الروـتينـ هيـ:
- MPI\_Finalize();

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

void 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();
}

```

برنامـج 2-9

لتنفيذ البرنامج (mpi\_app مثلا):

1. عمل Build Solution للمشروع (الانشاء ملف تنفيذي المشروع)
  2. افتح نافذة الاوامر ثم اذهب الى مكان المشروع
  3. ثم اكتب الامر mpiexec مع عدد المعالجات قبل استدعاء البرنامج
- مثلا لتنفيذ البرنامج mpi\_app الذي يوجد في المجلد:
- C:\xxx\Projects\mpi\_app\Debug
4. انتقل لهذا المجلد ثم اكتب الامر التالي (التنفيذ على 4 عمليات):
- Mpiexec -n 4 mpi\_app

مخرج البرنامج 2-9 بدون استخدام mpiexec مرة، وبعمليتين مرة ثانية، وباريعة عمليات في المرة الثالثة.

```

>mpi_app
I am 0 of 1
>mpiexec n 2 mpi_app
I am 1 of 2
I am 0 of 2
>mpiexec n 4 mpi_app
I am 1 of 4
I am 0 of 4
I am 2 of 4
I am 3 of 4

```

شرح برنامج 2-9:

عندما تحيط MPI بالأمر MPI\_Init ستصبح كل عملية فعالة (active process) عضو في مجتمع متصل (communicator) يسمى MPI\_COMM\_WORLD. حيث يعتبر هذا المجتمع (communicator) كائناً يوفر بيئة للتبادل الرسائل بين العمليات الأعضاء فيه.

MPI\_COMM\_WORLD هو المجتمع الافتراضي الذي يمكنك الحصول عليه مجاني. وهو كافٍ لمعظم برامجنا. ويمكن بناء مجتمعاتك الخاصة إذا أردت تجزيء العمليات إلى مجموعات مستقلة.

العمليات داخل المجتمع مرتبة، فإذا كان لديك مجتمع بعدد ع عمليات، فكل عملية سيكون لها رتبة مفردة غير متكررة بين صفر و  $n-1$ . قد تستخدم العملية رتبتها لمعرفة أي عمل أو بيانات هي مسؤولة عنه.

تستخدم الدالة MPI\_Comm\_rank لمعرفة رتبة العملية في المجتمع، مثلاً 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:

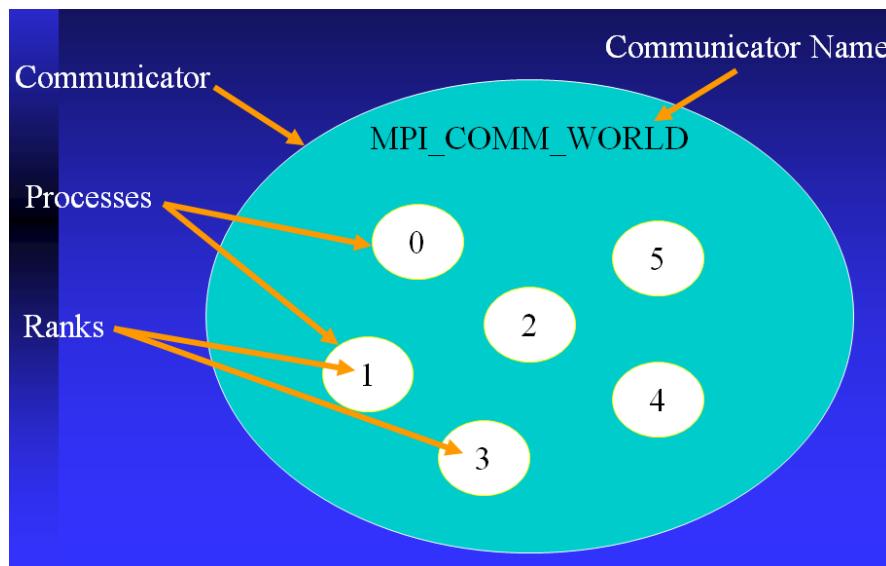
```
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

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

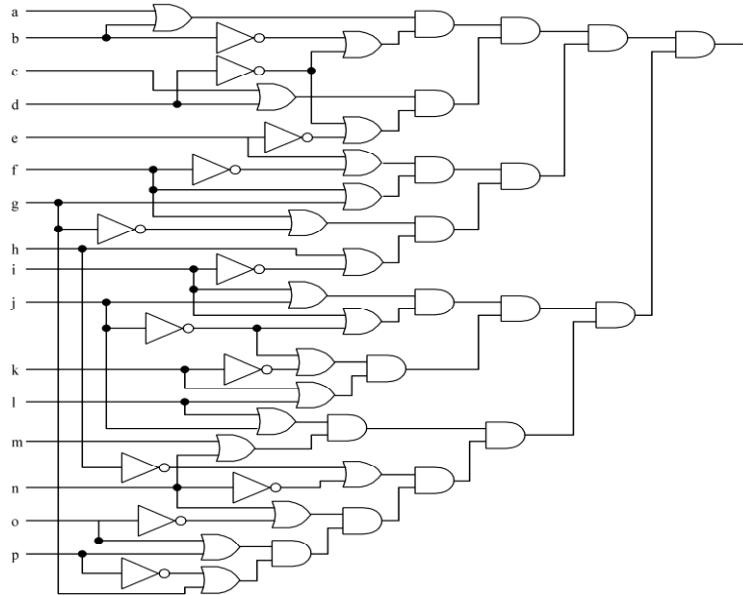


. [20] شكل رقم 9-3: رتبة العمليات في communicator

## 9.4. برنامج Circuit Satisfiability

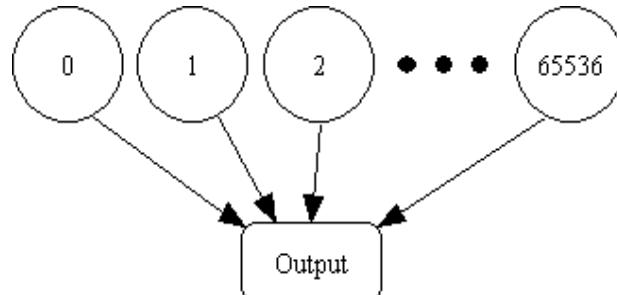
في هذا البرنامج نريد أن نعرف هل الدائرة أعلاه satisfiable أم لا ؟ ونطلق على الدائرة بأنها satisfiable إذا كان

مدخلاتها تنتج القيمة 1 .



شكل رقم 9-4: برنامج [20]. Circuit Satisfiability

حل هذه المسألة سنجريب كل الاحتمالات المدخلة، وبما أن مدخلات الدائرة هي 16 مدخل (من a إلى p كما في الشكل 9-4)، وكل مدخل يمكن أن يكون صفر أو واحد، فإن هنالك  $2^{16} = 65,536$  مدخل يمكن اختباره. يمكننا اختبار كل جزء من المدخلات في مهمة وتنفيذ هذه المهام بالتوازي (الشكل 9-5). كل مهمة تحد أن نتيجة المدخلات التي اختبرتها هي 1 تقوم بطبع هذه المدخلات.



شكل رقم 9-5 [20]

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

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

```

/*
 * This MPI program determines whether a circuit is
 * satisfiable, that is, whether there is a combination of
 * inputs that causes the output of the circuit to be 1.
 * The particular circuit being tested is "wired" into the
 * logic of function 'check_circuit'. All combinations of
 * inputs that satisfy the circuit are printed.
 */
#include "stdafx.h"
#include <stdio.h>
#include "mpi.h"

void main (int argc, char *argv[]) {
 int i;
 int id; /* Process rank */
 int p; /* Number of processes */
 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 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\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);
 }
}

```

[20] Circuit Satisfiability : 3-9 برماج

مخرج البرنامج

تنفيذ البرنامج بارعة عمليات

```

3) 1110111110011001
2) 0110111110011001
1) 1010111110011001
2) 011011111011001
3) 111011111011001
1) 101011111011001
2) 0110111110111001
3) 1110111110111001
1) 1010111110111001
Process 2 is done
Process 1 is done
Process 3 is done
Process 0 is done

```

### شرح البرنامج

يستخدم المتغير id ب تخزين رتبة العملية

المتغير p يخزن فيه عدد العمليات

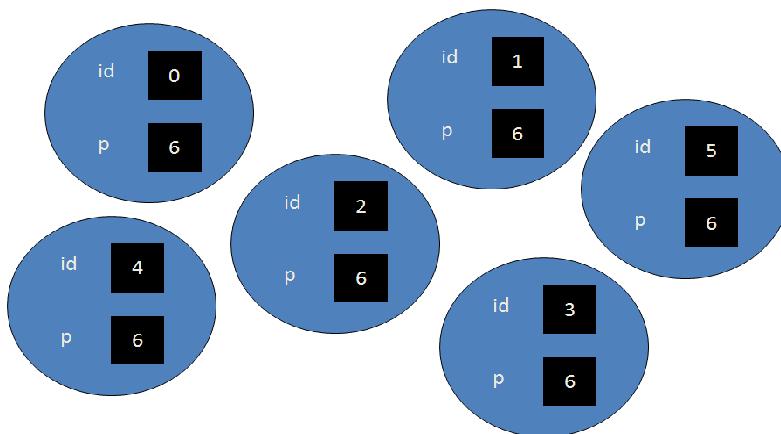
هذه المتغيرات يكون منها نسخة في كل عملية.

الشكل أدناه بعد تنفيذ الأمرين:

```

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

```



شكل رقم 9-6: ست عمليات [20]

توزيع المهام على العمليات 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) ورتبة هذه العملية هو صفر (id=0). وبالتالي فستتسع كل المهام لعملية واحدة.

أما عندما نحدد عدد العمليات بثلاث عمليات (mpirun -np 3 sat) فستكون p=3 (%). وبالتالي فستتسع كل المهام لعملية الأولى، وستتسع 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)

## 9.5. تبادل الرسائل في MPI

إرسال وإستقبال الرسائل بين العمليات هي الطريقة الأساسية المستخدمة في MPI لتبادل البيانات. هنالك دالتين أساسيتين

للإرسال والاستقبال. الصيغ العامة لهما كما يلي:

الصيغة العامة لدالة الارسال:

```
MPI_Send(
 void* data,
 int count,
 MPI_Datatype datatype,
 int destination,
 int tag,
 MPI_Comm communicator)
```

الصيغة العامة لدالة الاستقبال:

```
MPI_Recv(
 void* data,
 int count,
 MPI_Datatype datatype,
 int source,
 int tag,
 MPI_Comm communicator,
 MPI_Status* status)
```

المعطى (المدخل) الأول هي بيانات الخازن (data buffer). المدخل الثاني والثالث يحدد عدد ونوع العناصر التي ستكون في الخازن. يجب على الدالة المرسلة ارسال نفس العدد والتوعية من العناصر، وعلى الدالة المستلمة ان تستلم على الاكثر هذا العدد من العناصر. المدخل الرابع والخامس يحددان رتبة العملية المرسلة/المستقبلة علامة الرسالة (tag of the message). المدخل السادس يحدد التواصل (communicator). المدخل السابع (يوجد فقط في الدالة المرسلة) يوفر معلومات عن الرسالة المستلمة.

```
// Author: Wes Kendall
// Copyright 2011 www.mpitutorial.com
// This code is provided freely with the tutorials on mpitutorial.com. Feel
// free to modify it for your own use. Any distribution of the code must
// either provide a link to www.mpitutorial.com or keep this header intact.

//
// MPI_Send, MPI_Recv example. Communicates the number -1 from process 0
// to process 1.

#include "stdafx.h"
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
 // Initialize the MPI environment
 MPI_Init(NULL, NULL);
 // Find out rank, size
 int world_rank;
 MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
 int world_size;
 MPI_Comm_size(MPI_COMM_WORLD, &world_size);
```

```

// We are assuming at least 2 processes for this task
if (world_size < 2) {
 fprintf(stderr, "World size must be greater than 1 for %s\n", argv[0]);
 MPI_Abort(MPI_COMM_WORLD, 1);
}

int number;
if (world_rank == 0) {
 // If we are rank 0, set the number to -1 and send it to process 1
 number = -1;
 MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (world_rank == 1) {
 MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
 printf("Process 1 received number %d from process 0\n", number);
}
MPI_Finalize();
}

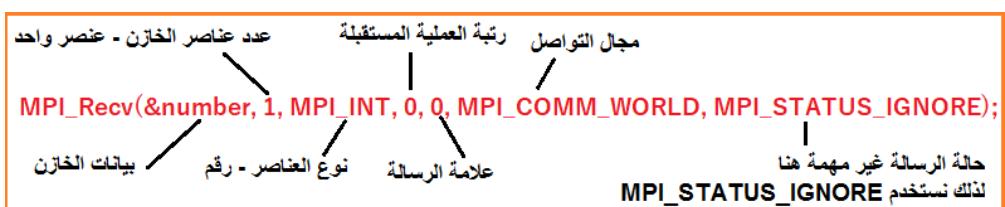
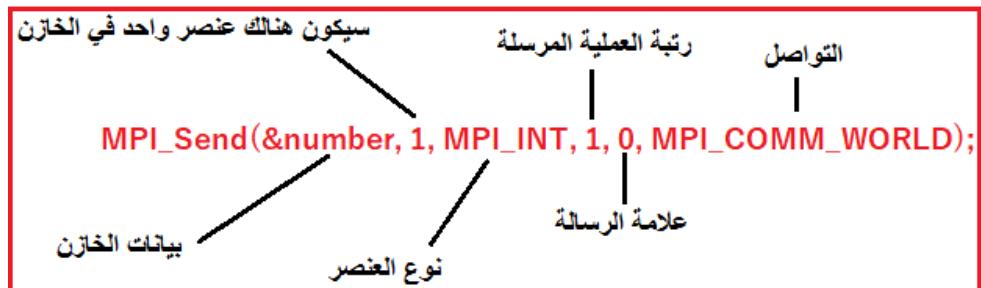
```

[21] برنامج 9-5: ارسال/استقبال

مخرج البرنامج أعلاه هو:

Process 1 received number -1 from process 0

شرح الارسال والاستقبال في البرنامج 9-5



علامة الرسالة (tag) في الراسل والمستقبل يجب أن تكون متطابقة.

مجال الاتصال في الراسل والمستقبل يجب أن يكون متطابق.

جدول 9-1: انواع البيانات في MPI . [21]

| MPI datatype           | C equivalent           |
|------------------------|------------------------|
| MPI_SHORT              | short int              |
| MPI_INT                | int                    |
| MPI_LONG               | long int               |
| MPI_LONG_LONG          | long long int          |
| MPI_UNSIGNED_CHAR      | unsigned char          |
| MPI_UNSIGNED_SHORT     | unsigned short int     |
| MPI_UNSIGNED           | unsigned int           |
| MPI_UNSIGNED_LONG      | unsigned long int      |
| MPI_UNSIGNED_LONG_LONG | unsigned long long int |
| MPI_FLOAT              | float                  |
| MPI_DOUBLE             | double                 |
| MPI_LONG_DOUBLE        | long double            |
| MPI_BYTE               | char                   |

```
#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 9-5، ترسل العملية رقم صفر (myrank = 0) رسالة للعملية رقم 1 وذلك باستخدام الدالة MPI\_SEND. حيث يتم تحديد مكان الحازن (buffer) بواسطة المدخل الاول، وتحديد حجم (عدد) ونوع البيانات المرسلة بالمدخل الثاني والثالث للدالة MPI\_Send وهي (مكان الرسالة message، حجم الرسالة strlen(message)، نوع الرسالة MPI\_CHAR). المدخل الرابع يحدد فيه وجهة الرسالة (رتبة العملية المرسل لها الرسالة) وهي 1. المدخل الخامس عنوان تعريفي للرسالة (علامة الرسالة) 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 الثلاث الأولى تحدد مكان وطول ونوع الرسالة المستلمة (مكان تخزين الرسالة message، طول الرسالة 20 ، نوع الرسالة MPI\_CHAR). المدخل الرابع (علامة الرسالة) أو عنوان الرسالة المستلمة وهو 99 (لابد من أن يكون مطابق للعنوان المذكور في دالة الإرسال). المدخل الخامس المجتمع (نفس الذي في دالة الإرسال). المدخل السادس يعطينا معلومات عن الرسالة المستلمة حيث تعتبر status مؤشر لسجل يحتوي تفاصيل معلومات مثل رتبة العملية التي أرسلت الرسالة (Status->MPI\_Source).

ستظل الدالة MPI\_Recv محجوزة في انتظار الرسالة المرسلة حتى تصل (أو يحدث خطأ يوقف الدالة).  
يسمى هذا النوع من الاتصال بالإتصال المحجز أو المتزامن (blocking communication).

## 9.6. الإختناق (deadlock)

تعتبر العملية في حالة اختناق إذا كانت تتنتظر حدوث لن يتم أبداً. لذلك من السهل الوقوع في حالة اختناق عند كتابة برنامج يستخدم MPI\_Recv و MPI\_Send.

مثلاً إذا كان لدينا عمليتين تريدان حساب متوسط العددين a و b ، ومتلك العملية الأولى أحدث قيمة للمتغير a، بينما متلك 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).

أيضاً من مسببات الاختناق أن لا يتطابق رقم تعريف الرسالة (tag) في دالة الإرسال مع رقم تعريف الرسالة في دالة الاستقبال.

أيضاً قد يحدث إختناق إذا أرسلت عملية رسالة إلى عملية خطأ، أو عندما تحاول عملية إستلام رسالة من عملية خطأ.

في الاتصال غير المتزامن (غير المحجوز) Asynchronous (non-blocking) Communication، لن تنتظر عملية الإرسال والاستقبال حتى يكتمل الاتصال، وعken استخدام دالة الإنتظار wait لمعرفة متى أكتمل 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;
- 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 محوّزة حتّى تكتمل الرسالة المحددة بـ .request  
مثال برمجي يوضح استخدام الاتصال غير المحوّز

```
#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;
}
```

برنامـج 7-9

## 9.7. قياس الأداء (benchmarking performance)

أحد طرق قياس أداء البرنامج هو حساب الزمن من بداية البرنامج حتّى نهايته. لتقليل زمن التنفيذ يمكننا تجاوز الزمن المستغرق في تخيّلة العمليات وفتح الاتصالات بين العمليات وعمليات الدخول والخرج على الأجهزة المتسلسلة، وسنحسب الزمن المستغرق في التنفيذ الحقيقي (قلب البرنامج) بين قراءة البيانات وطباعة النتائج.  
هناك دالّتين لقياس الزمن في MPI هما:

MPI\_Wtick — timer resolution

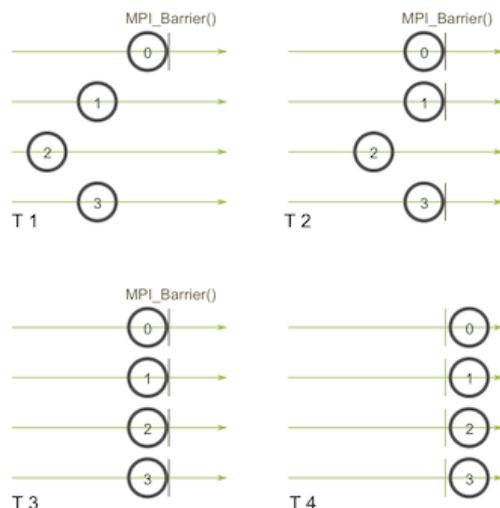
MPI\_Wtime — current time

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

### MPI\_Barrier الدالة

حل هذه المشكلة سنتستخدم 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 يضمن أن كل العمليات ستكون في جزء الشفرة الذي نستخدمه لقياس في نفس الوقت تقريباً. الصيغة العامة للدالة الحاجز هي:

`MPI_Barrier(MPI_Comm comm)`



شكل رقم 9-7 : حجز العمليات للتزامن [21].

يمكّنا قياس أداء في البرنامج السابق تعديل 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();
```

البرنامج 10-9

نستدعي الحاجز بعد تحية 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 التي في الجدول 9-3.

جدول رقم 9-3: مقارنة الاداء.

| المعالج | الزمن بالثانية |
|---------|----------------|
| 1       | 15.93          |
| 2       | 8.38           |
| 3       | 5.86           |
| 4       | 4.60           |
| 5       | 3.77           |

### تمرين

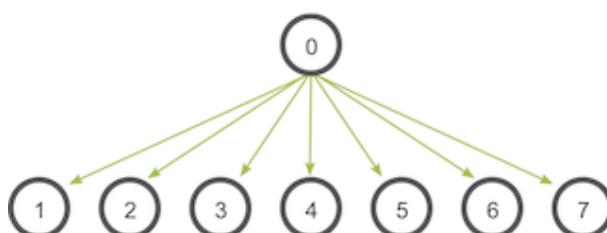
أكمل البرنامج 9-10 ثم قارن النتائج مع الموجودة في الجدول .

## 9. الدالة MPI\_Bcast

تستخدم هذه الدالة لبث (broadcast) عناصر بيانات (من نفس النوع) من عملية لبقية العمليات الموجودة معها في مجتمع الاتصال. مثلا الامر:

```
MPI_Bcast(array, 100, MPI_INT, root, comm);
```

يقوم ببث 100 عدد صحيح من العملية root لبقية العمليات في الموجودة معها في مجتمع الاتصال (comm).



الصيغة العامة للدالة:

```
MPI_Bcast(
 void* data,
 int count,
 MPI_Datatype datatype,
 int root,
 MPI_Comm communicator)
```

حيث يتم تعيين المتغير data بما يريد به.

```
// Author: Wes Kendall
// Copyright 2011 www.mpitutorial.com
// This code is provided freely with the tutorials on mpitutorial.com. Feel
// free to modify it for your own use. Any distribution of the code must
// either provide a link to www.mpitutorial.com or keep this header intact.
//
// An example of a function that implements MPI_Bcast using MPI_Send and
// MPI_Recv
```

```

//

#include "stdafx.h"

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

void my_bcast(void* data, int count, MPI_Datatype datatype, int root,

 MPI_Comm communicator) {

 int world_rank;

 MPI_Comm_rank(communicator, &world_rank);

 int world_size;

 MPI_Comm_size(communicator, &world_size);

 if (world_rank == root) {

 // If we are the root process, send our data to everyone

 int i;

 for (i = 0; i < world_size; i++) {

 if (i != world_rank) {

 MPI_Send(data, count, datatype, i, 0, communicator);

 }
 }
 } else {
 // If we are a receiver process, receive the data from the root

 MPI_Recv(data, count, datatype, root, 0, communicator, MPI_STATUS_IGNORE);
 }
}

int main(int argc, char** argv) {

 MPI_Init(NULL, NULL);

 int world_rank;

 MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

 int data;

 if (world_rank == 0) {

 data = 100;

 printf("Process 0 broadcasting data %d\n", data);

 my_bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);

 } else {

 my_bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);

 printf("Process %d received data %d from root process\n", world_rank, data);
 }
}

 MPI_Finalize();
}

```

برنامـج 9-8: تطبيق MPI\_Bcast عن طريق دالـيـة الـاـرسـال وـالـاستـقبـال

مخرج البرنامج عند تنفيذه بالامر :mpiexec -n 4

```

Process 2 received data 100 from root process
Process 0 broadcasting data 100
Process 1 received data 100 from root process
Process 3 received data 100 from root process

```

المخرج لـعشرة عمليات (mpiexec -n 10)

```

Process 4 received data 100 from root process
Process 2 received data 100 from root process
Process 6 received data 100 from root process
Process 5 received data 100 from root process
Process 7 received data 100 from root process
Process 1 received data 100 from root process
Process 9 received data 100 from root process
Process 0 broadcasting data 100
Process 3 received data 100 from root process
Process 8 received data 100 from root process

```

مقرنة الـبـث عن طـرـيق دـالـيـة الـاـرسـال وـالـاستـقبـال مع الـبـث باـسـتـخـدـام دـالـة الـبـث MPI\_Bcast  
الـبرـنـامـج التـالـي يـسـتـخـدـم طـرـيقـتـيـن للـبـث اـحـدـهـما عن طـرـيق دـالـيـة الـاـرسـال وـالـاستـقبـال، وـالـطـرـيقـةـ الثـانـيـة باـسـتـخـدـام دـالـة الـبـث المـخـصـصـة لـذـلـكـ .MPI\_Bcast وهي

```

// Author: Wes Kendall
// Copyright 2011 www.mpitutorial.com
// This code is provided freely with the tutorials on mpitutorial.com. Feel
// free to modify it for your own use. Any distribution of the code must
// either provide a link to www.mpitutorial.com or keep this header intact.
//
// Comparison of MPI_Bcast with the my_bcast function
//
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <assert.h>

void my_bcast(void* data, int count, MPI_Datatype datatype, int root,
 MPI_Comm communicator) {
 int world_rank;
 MPI_Comm_rank(communicator, &world_rank);
 int world_size;
 MPI_Comm_size(communicator, &world_size);

 if (world_rank == root) {
 // If we are the root process, send our data to everyone
 int i;
 for (i = 0; i < world_size; i++) {
 if (i != world_rank) {
 MPI_Send(data, count, datatype, i, 0, communicator);
 }
 }
 } else {
 // If we are a receiver process, receive the data from the root
 MPI_Recv(data, count, datatype, root, 0, communicator, MPI_STATUS_IGNORE);
 }
}

int main(int argc, char** argv) {
 if (argc != 3) {
 fprintf(stderr, "Usage: compare_bcast num_elements num_trials\n");
 exit(1);
 }

 int num_elements = atoi(argv[1]);
 int num_trials = atoi(argv[2]);

 MPI_Init(NULL, NULL);

 int world_rank;
 MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

 double total_my_bcast_time = 0.0;
 double total_mpi_bcast_time = 0.0;
 int i;
 int* data = (int*)malloc(sizeof(int) * num_elements);
 assert(data != NULL);

 for (i = 0; i < num_trials; i++) {
 // Time my_bcast
 // Synchronize before starting timing
 MPI_Barrier(MPI_COMM_WORLD);
 total_my_bcast_time -= MPI_Wtime();
 my_bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);
 // Synchronize again before obtaining final time
 MPI_Barrier(MPI_COMM_WORLD);
 total_my_bcast_time += MPI_Wtime();

 // Time MPI_Bcast
 MPI_Barrier(MPI_COMM_WORLD);
 total_mpi_bcast_time -= MPI_Wtime();
 MPI_Bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);
 MPI_Barrier(MPI_COMM_WORLD);
 total_mpi_bcast_time += MPI_Wtime();
 }

 // Print off timing information
}

```

```

if (world_rank == 0) {
 printf("Data size = %d, Trials = %d\n", num_elements * (int)sizeof(int),
 num_trials);
 printf("Avg my_bcast time = %lf\n", total_my_bcast_time / num_trials);
 printf("Avg MPI_Bcast time = %lf\n", total_mpi_bcast_time / num_trials);
}

MPI_Finalize();
}

```

[21] 9-9 . برنامج

مخرج البرنامج اعلاه هو:

```

>mpiexec -n 10 bcast_test 100000 10
Data size = 400000, Trials = 10
Avg my_bcast time = 0.016367
Avg MPI_Bcast time = 0.024465

```

جدول رقم 9 : مقارنة النتائج في عدد مختلف من المعالجات [21].

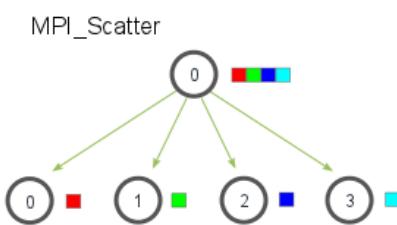
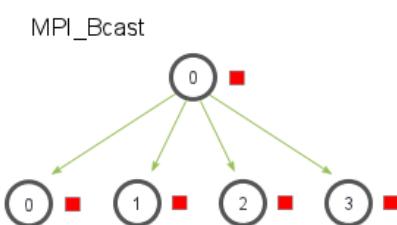
| Processors | my_bcast | MPI_Bcast |
|------------|----------|-----------|
| 2          | 0.0344   | 0.0344    |
| 4          | 0.1025   | 0.0817    |
| 8          | 0.2385   | 0.1084    |
| 16         | 0.5109   | 0.1296    |

تمرين

جرب ان تنفذ البرنامج اعلاه على نظامك لنفس عدد المعالجات التي بالجدول وقارن بين النتائج المبينة هنا والتي بنظمتك.

### 9.9 الدالة MPI\_Scatter

تشبه في عملها MPI\_Bcast لكن الفرق ان MPI\_Bcast تبث نسخة من نفس البيانات، بينما تبث MPI\_Scatter تبث بيانات مختلفة.

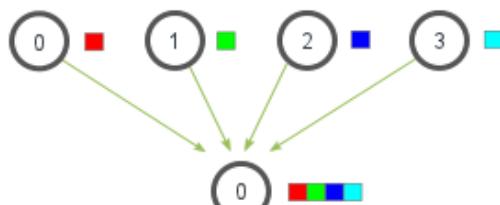


[21] 9-9 . شكل رقم

## MPI\_Gather . الدالة 9.10

تعمل عكس MPI\_Scattr، تقوم بتجمیع عناصر من عمليات مختلفة.

MPI\_Gather



.[21] 10-9 شکل رقم

```
// Author: Wes Kendall
// Copyright 2012 www.mpitutorial.com
// This code is provided freely with the tutorials on mpitutorial.com. Feel
// free to modify it for your own use. Any distribution of the code must
// either provide a link to www.mpitutorial.com or keep this header intact.
//
// Program that computes the average of an array of elements in parallel using
// MPI_Scatter and MPI_Gather
//
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>
#include <assert.h>

// Creates an array of random numbers. Each number has a value from 0 - 1
float *create_rand_nums(int num_elements) {
 float *rand_nums = (float *)malloc(sizeof(float) * num_elements);
 assert(rand_nums != NULL);
 int i;
 for (i = 0; i < num_elements; i++) {
 rand_nums[i] = (rand() / (float)RAND_MAX);
 }
 return rand_nums;
}

// Computes the average of an array of numbers
float compute_avg(float *array, int num_elements) {
 float sum = 0.f;
 int i;
 for (i = 0; i < num_elements; i++) {
 sum += array[i];
 }
 return sum / num_elements;
}

int main(int argc, char** argv) {
 if (argc != 2) {
 fprintf(stderr, "Usage: avg num_elements_per_proc\n");
 exit(1);
 }

 int num_elements_per_proc = atoi(argv[1]);
 // Seed the random number generator to get different results each time
 srand(time(NULL));

 MPI_Init(NULL, NULL);

 int world_rank;
 MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
 int world_size;
 MPI_Comm_size(MPI_COMM_WORLD, &world_size);

 // Create a random array of elements on the root process. Its total
```

```

// size will be the number of elements per process times the number
// of processes
float *rand_nums = NULL;
if (world_rank == 0) {
 rand_nums = create_rand_nums(num_elements_per_proc * world_size);
}

// For each process, create a buffer that will hold a subset of the entire
// array
float *sub_rand_nums = (float *)malloc(sizeof(float) * num_elements_per_proc);
assert(sub_rand_nums != NULL);

// Scatter the random numbers from the root process to all processes in
// the MPI world
MPI_Scatter(rand_nums, num_elements_per_proc, MPI_FLOAT, sub_rand_nums,
 num_elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Compute the average of your subset
float sub_avg = compute_avg(sub_rand_nums, num_elements_per_proc);

// Gather all partial averages down to the root process
float *sub_avgs = NULL;
if (world_rank == 0) {
 sub_avgs = (float *)malloc(sizeof(float) * world_size);
 assert(sub_avgs != NULL);
}
MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Now that we have all of the partial averages on the root, compute the
// total average of all numbers. Since we are assuming each process computed
// an average across an equal amount of elements, this computation will
// produce the correct answer.
if (world_rank == 0) {
 float avg = compute_avg(sub_avgs, world_size);
 printf("Avg of all elements is %f\n", avg);
 // Compute the average across the original data for comparison
 float original_data_avg =
 compute_avg(rand_nums, num_elements_per_proc * world_size);
 printf("Avg computed across original data is %f\n", original_data_avg);
}

// Clean up
if (world_rank == 0) {
 free(rand_nums);
 free(sub_avgs);
}
free(sub_rand_nums);

MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();
}

```

. [21] ١٠-٩ برنا مج .

مخرج البرنامج

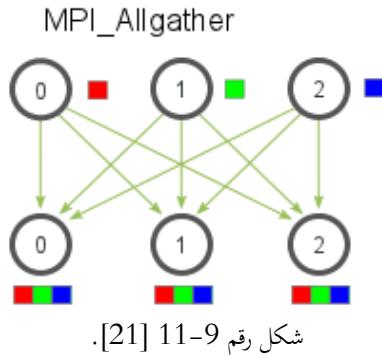
```

>mpiexec -n 4 mpi_avr 100000
Avg of all elements is 0.500438
Avg computed across original data is 0.500433

```

## MPI\_Allgather الدالة 9.11

جمع كل العناصر لكل العمليات.



## 9.12 دالة الاختزال MPI\_Reduce

تشبة MPI\_Gather فهي تأخذ مصفوفة من العناصر المدخلة من كل عملية ثم ترجع مصفوفة من العناصر المخرجية، العناصر المخرجية تحتوي النتيجة المختزلة.

الصيغة العامة للدالة

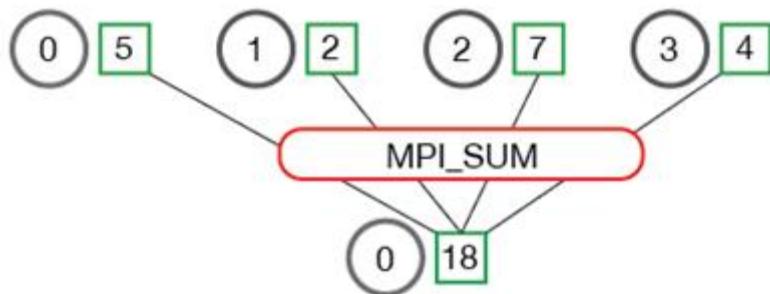
```
MPI_Reduce(
 void* send_data, // operand (1st reduction element
 void* recv_data, // result (address of 1st reduction result
 int count, // reduction to perform
 MPI_Datatype datatype, // type of elements
 MPI_Op op, // reduction operator
 int root, // process getting result(s)
 MPI_Comm communicator) // communicator
```

- المدخل الأول (send\_data): مصفوفة العناصر التي تريد كل عملية اختزالها.
- المدخل الثاني (recv\_data): مصفوفة العناصر التي تخزن فيها نتائج الاختزال (خاصة بالعملية root).
- المدخل الثالث للدالة (count): عدد العناصر المختزلة، حيث تحدد حجم recv\_data التي ستحسب بانها . sizeof(datatype) \* count
- المدخل الرابع (datatype): نوع العناصر التي نريد اختزالها.
- المدخل الخامس (op): نوع الاختزال الذي نريد تنفيذه (انظر الجدول 9-3).
- المدخل السادس (root): العملية الجذر (رقم رتبة العملية التي سيكون بها كل نتائج الاختزالات).
- المدخل الأخير (comm): مجتمع الاتصال (communicator) الذي يحتوي على العمليات المشاركة في الاختزال.

جدول رقم 9-3: أنواع الاختزال [21].

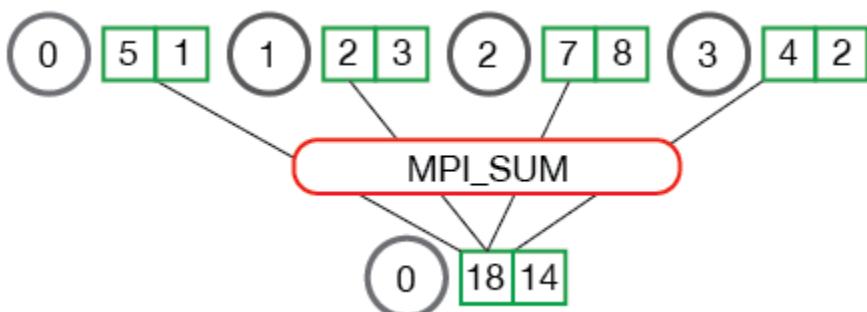
|                   |                                        |
|-------------------|----------------------------------------|
| <b>MPI_MAX</b>    | أكبر عنصر                              |
| <b>MPI_MIN</b>    | أصغر عنصر                              |
| <b>MPI_SUM</b>    | مجموع العناصر                          |
| <b>MPI_PROD</b>   | حاصل ضرب كل العناصر                    |
| <b>MPI_LAND</b>   | تنفيذ logical and على كل العناصر       |
| <b>MPI_LOR</b>    | تنفيذ logical or على كل العناصر        |
| <b>MPI_BAND</b>   | تنفيذ Bitwise and على كل بิตات العناصر |
| <b>MPI_BOR</b>    | تنفيذ Bitwise or على كل بิตات العناصر  |
| <b>MPI_MAXLOC</b> | ارجاع أكبر قيمة والعملية التي تمتلكها  |

الشكل 9-11 تحتوي كل عملية رقم واحد، وتم تنفيذ MPI\_Reduce بمعامل الاختزال MPI\_SUM فكانت النتيجة جمع الاربعة ارقام من العمليات الاربعة وتخزينها في العملية التي استدعت دالة الاختزال.



شكل رقم 9-12: توضيح لعمل MPI\_Reduce .[21]

في الشكل 9-13 يتم جمع كل خانة مع بعضها، ويتم اختزالها في الخانة المشابهة في العملية الجذر ( $18=4+7+2+5$ ). ( $14=2+8+3+1$ ).



شكل رقم 9-13: كل عملية تحتوي رقمين [21].

### 9.13. الملخص

بمذا إكتمل الباب ولكن لم تكتمل قصة MPI، ذلك لأن الدوال الموجودة فيها كثيرة جدا ولكن كما ذكرنا فإن هناك 6 دوال رئيسية شرحناها بالتفصيل في بداية الباب وتقوم بالاعمال الأساسية في مجال البرمجة الموزعة، ثم شرحنا دوال أخرى إضافية رائنا أنك قد تحتاجها، وبهذه الطريقة تستطيع التعامل مع أي دالة من دوالة MPI حتى ولو لم تدرسها وذلك من خلال معرفتك للصيغة العامة لها وما الغرض منها. هذا الباب أن يدللك على طريق MPI ولكنه لن يوصلك، يمكن الاستزادة من المصادر المثبتة في هذا الكتاب..  
معظم البرامج والشرحوات التي استخدمتها في هذا الباب هي من مرجعين رئيسين هما :

- كتاب Michael Quinn ومؤلفه هو Parallel Programming in C with MPI and OpenMP

• موقع MPI Tutorial الذي اعدت بواسطة Wesley Bland، Dwaraka Nath، Wes Kendall

ورابط الموقع هو [mpitutorial.com](http://mpitutorial.com). ويمكنك الحصول على البرامج التي استخدمناها من الموقع وزيادة من الرابط <https://github.com/wesleykendall/mpitutorial/tree/gh-pages>.

**الملاحق**

## ملحق (١) سرعة المعالج إلى أين؟

تقابل الميتر Hz الدورات (cycles) التي تحدث في الثانية . kHz (kilohertz, 103 Hz), MHz (megahertz, 106 Hz), GHz (gigahertz, 109 Hz) and THz (terahertz, 1012 Hz)

يتم تنسيق كافة المكونات في جهاز الكمبيوتر الشخصي بساعة النظام. تولد ساعة النظام إشارة متقلبة بين الفولتية العالية والمنخفضة. وتنسق رقائق الكمبيوتر المختلفة.

كان أول جهاز كمبيوتر استخدمناه هو 286 ثم ظهر بعد 386، 486، بسرعة 66 ميجا هيرتز، ثم ظهرت معالجات بسرعة 500 ميجا هيرتز، 1 غيغا هيرتز، واستمرت صعوداً إلى حوالي 4 غيغا هيرتز، ثم لم تزيد كثيراً بعد ذلك. فما هو السبب ؟

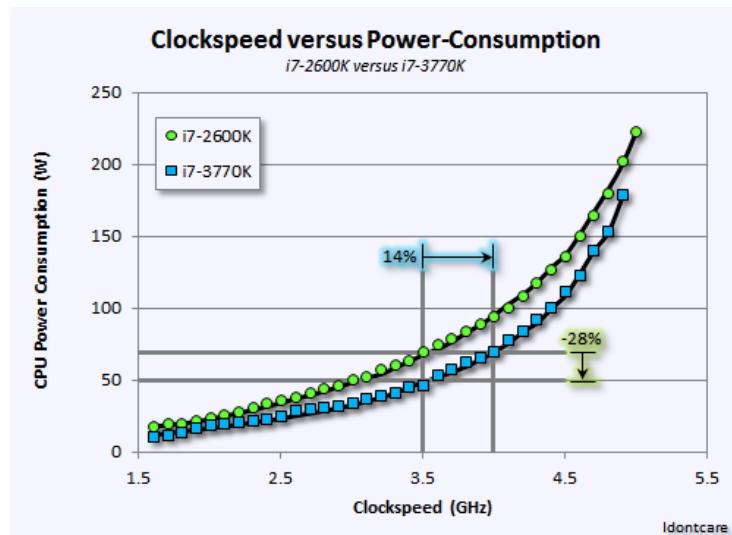
البعض ( <http://computer.howstuffworks.com/question307.htm> ) يعزى ذلك إلى :

1. تأخير النقل في الرقيقة chip

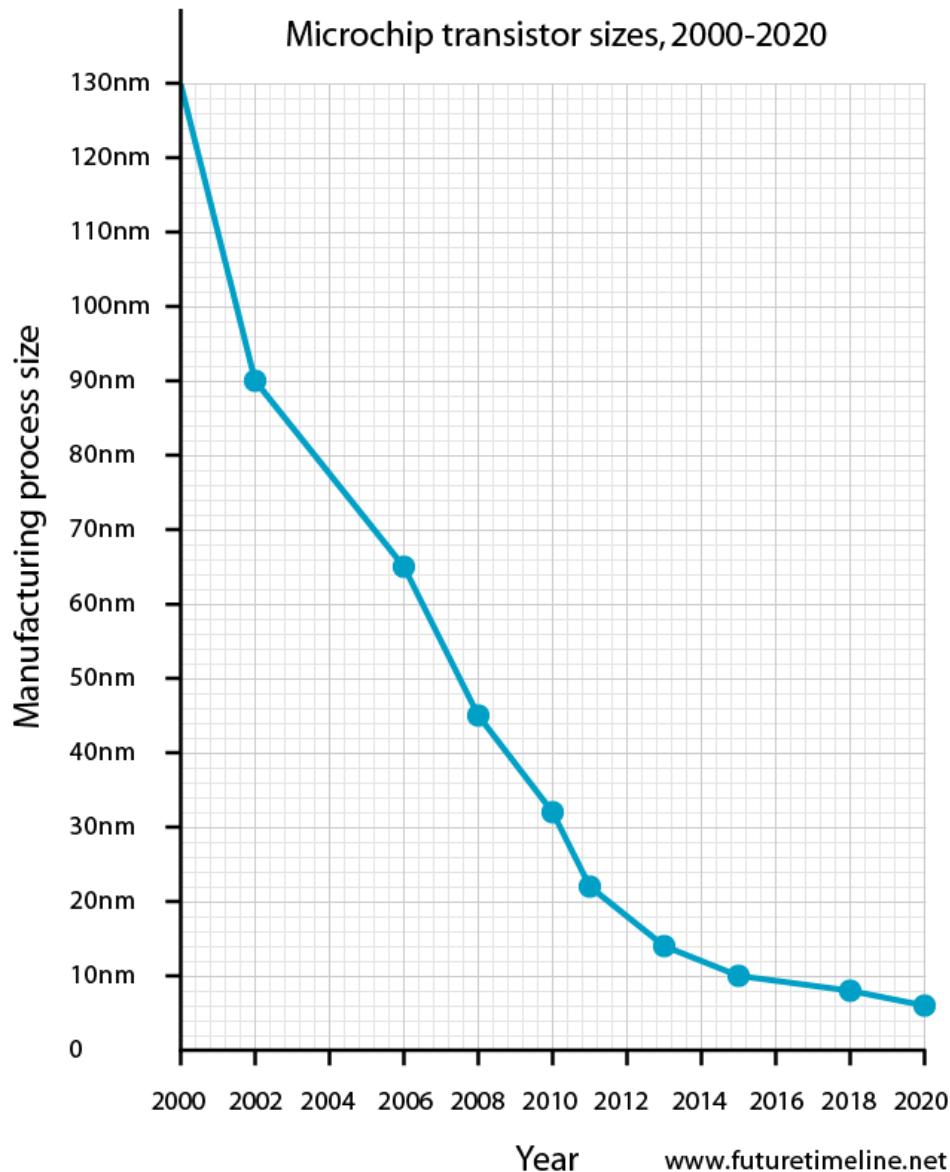
2. زيادة الحرارة heat build-up on the chip

يمكنك الاطلاع على الآراء المذكورة في الواقع التالية:

<https://www.comsol.com/blogs/havent-cpu-clock-speeds-increased-last-years/>



<https://www.quora.com/Why-havent-CPU-clock-speeds-increased-in-the-last-5-years> من



موقع ذكر <http://www.tomshardware.com/forum/250325-28-limit-clock-speed>  
أيه من الناحية النظرية، فإن الحد العلوي لتأخير الانتشار في إرسال الإلكترونات من نقطة واحدة إلى أخرى، والتي ستكون سرعة الكهرباء. الكهرباء تنتقل قريبة من سرعة الضوء، والضوء يتنقل بقطع القدم في الفيمتو ثانية ( $E-15$  10..... $10^{-15}$  ثانية).....

يمكنك قراءة الرد على السؤال التالي:

How fast (ie: clockspeed in Hz) can a single processor in a computer be? The highest I've read is 4 GHz. What's the limit?

في الموقع التالي:

<http://www.physlink.com/education/askexperts/ae391.cfm>  
في الموقع <http://www.geek.com/chips/intel-predicts-10ghz-chips-by-2011-564808>  
اذا افترضنا ان قانون مور ساري الى 2011 فإنه سيكون لدينا معالج بسرعة 128 غيغاهرتز في 2011.

assuming that early 2001 is a time when 1 GHz processors are rampant let's see what we get if we apply Moore's Law:

early 2001: 1 GHz

mid 2002: 2 GHz

early 2004: 4 GHz

mid 2005: 8 GHz

early 2007: 16 GHz

mid 2008: 32 GHz

early 2010: 64 GHz

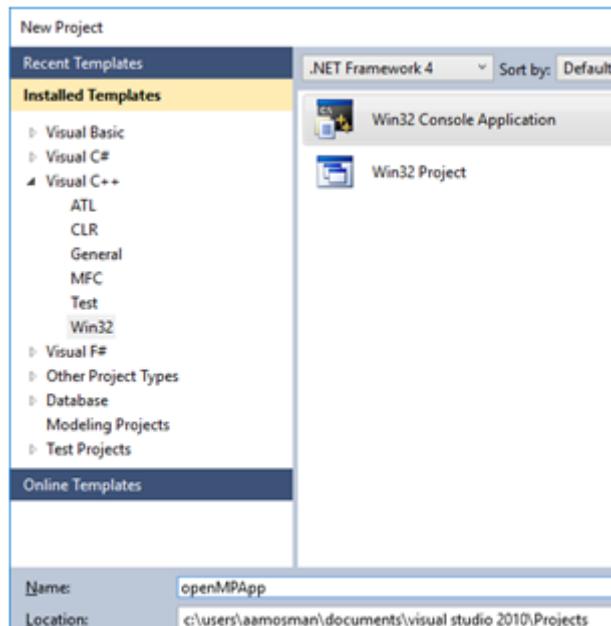
mid 2011: 128 GHz

## ملحق (ب) تفعيل OpenMP في فيجوال استديو 2010

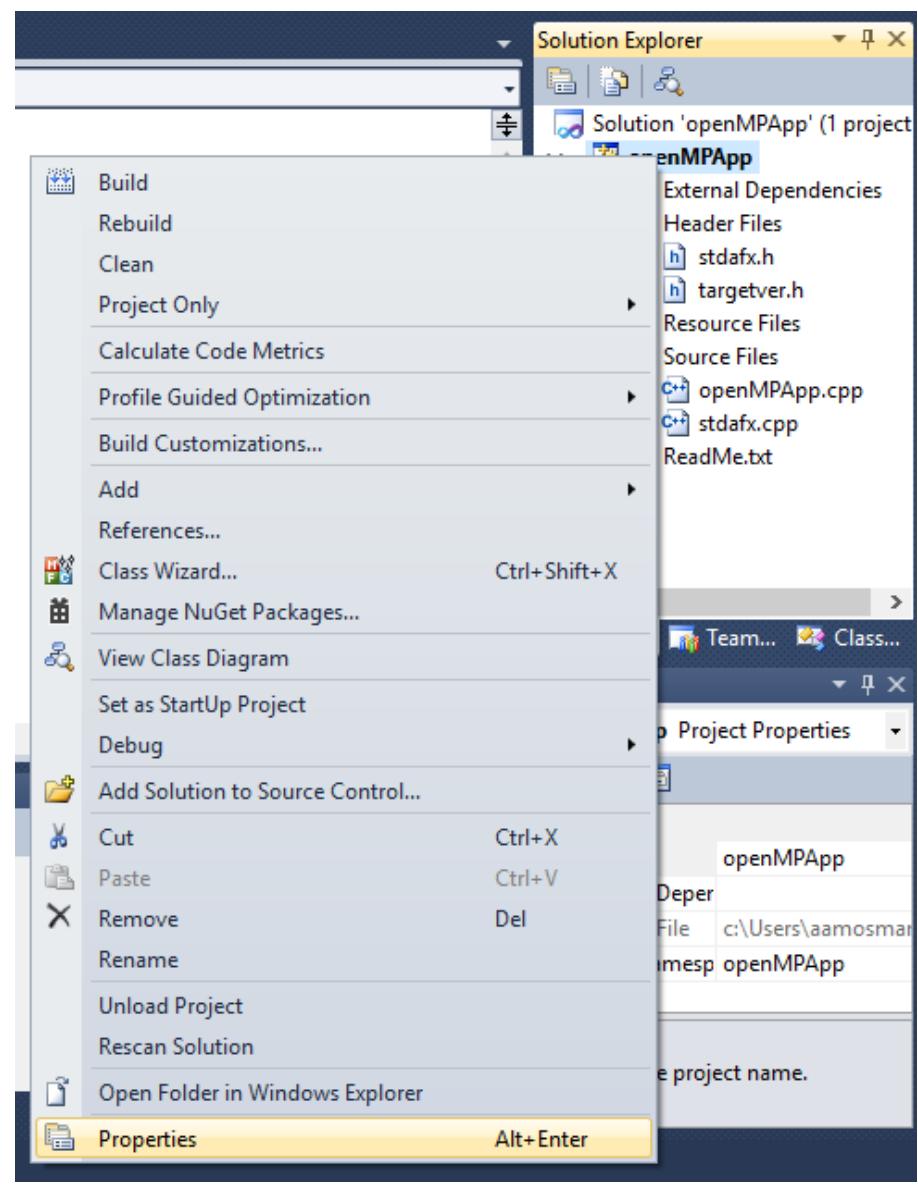
1. لاستخدام openMP في فيجوال استديو:
2. انشاء مشروع جديد (Create your project)
3. تفعيل OpenMP (Enable OpenMP)
4. تضمين الترويسة omp.h في البرنامج (Include omp.h)

الخطوات التالية توضح ذلك بالتفصيل:

Start Page - Microsoft Visual Studio



:properties يمين الماوس على المشروع ثم اختصار



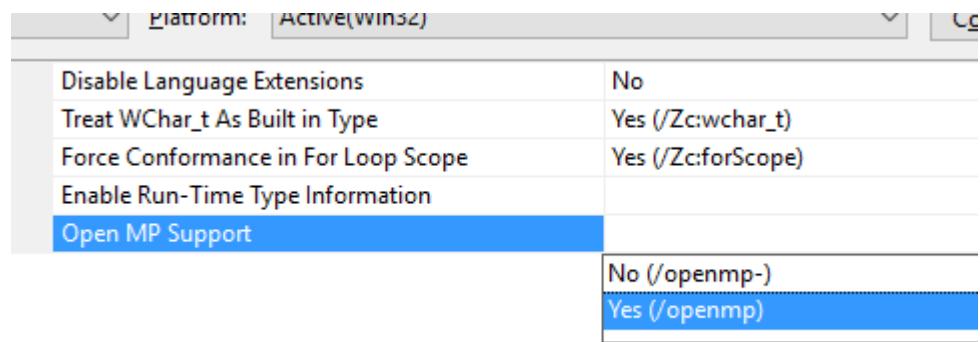
في النافذة التالية:

openMPApp Property Pages

Configuration: Active(Debug) Platform: Active(Win32)

|                            |                                     |                    |
|----------------------------|-------------------------------------|--------------------|
| > Common Properties        | Disable Language Extensions         | No                 |
| ✓ Configuration Properties | Treat WChar_t As Built in Type      | Yes (/Zc:wchar_t)  |
| General                    | Force Conformance in For Loop Scope | Yes (/Zc:forScope) |
| Debugging                  | Enable Run-Time Type Information    |                    |
| VC++ Directories           | Open MP Support                     |                    |
| C/C++                      |                                     |                    |
| General                    |                                     |                    |
| Optimization               |                                     |                    |
| Preprocessor               |                                     |                    |
| Code Generation            |                                     |                    |
| <b>Language</b>            |                                     |                    |
| Precompiled Headers        |                                     |                    |

: اختار yes



أول برنامج

عدل البرنامج الموجود في المشروع ليكون كالتالي:

```
#include "stdafx.h"
#include <stdio.h>
#include "omp.h"

Void main()
{
 int E=5;
 #pragma omp parallel
 {
 printf("E is equal to %d\n",E);
 }
}
```

عند التنفيذ (وذلك بالضغط على Ctrl+F5) سيكون المخرج كما يلي:

A screenshot of a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The command 'E:\Program Files\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.29.30133\bin\Hostx64\x64\cl.exe /EHsc /O2 /std:c++17 /MP /openmp test1.cpp' is entered. The output shows the variable 'E' being printed four times as 'E is equal to 5'. The prompt at the bottom says 'Press any key to continue . . .'

## ملحق (ج) اعداد MPI في فيجوال استديو 2010

تحميل واعداد البرامج التالية (من شركة ميكروسوفت)

msmpisdk.msi

MSMPiSetup.exe

رابط التحميل هو

<https://www.microsoft.com/en-us/download/details.aspx?id=52981>

1. افتح مشروع جديد

2. اختار visual c++ -> win32 -> win32 Console Application

3. ضع اسم لمشروعك

4. افتح خصائص المشروع

5. افتح MPI Cluster Debugger ثم غير حقل Debugger to launch إلى Debugging

6. غير Run Environment إلى localhost/4 (اذا اردت تشغيل 4 عمليات).

7. افتح General ثم C/C++

غير مجلدات المكتبات والانكلود الى مكان الذي اعددت فيه MSMPI

غالباً ما تكون

C:\Program Files (x86)\Microsoft SDKs\MPI\Include

C:\Program Files (x86)\Microsoft SDKs\MPI\Include\x64

C:\Program Files (x86)\Microsoft SDKs\MPI\Include\x86

C:\Program Files (x86)\Microsoft SDKs\MPI\Lib

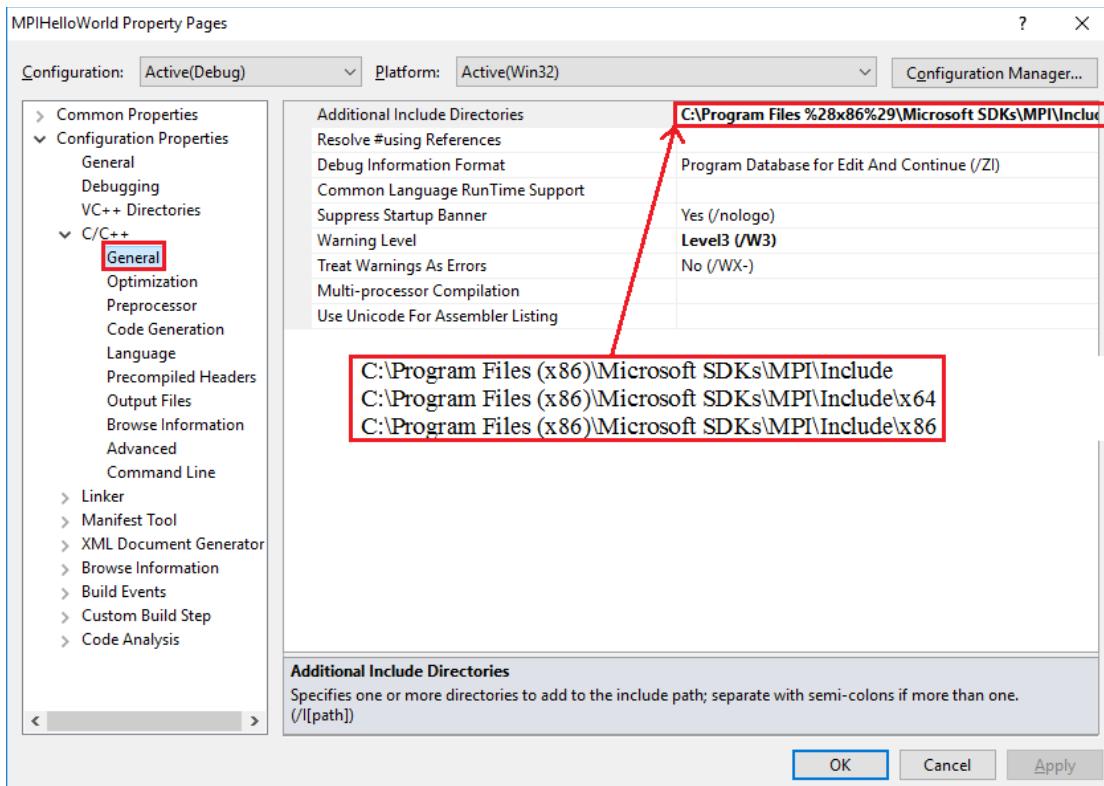
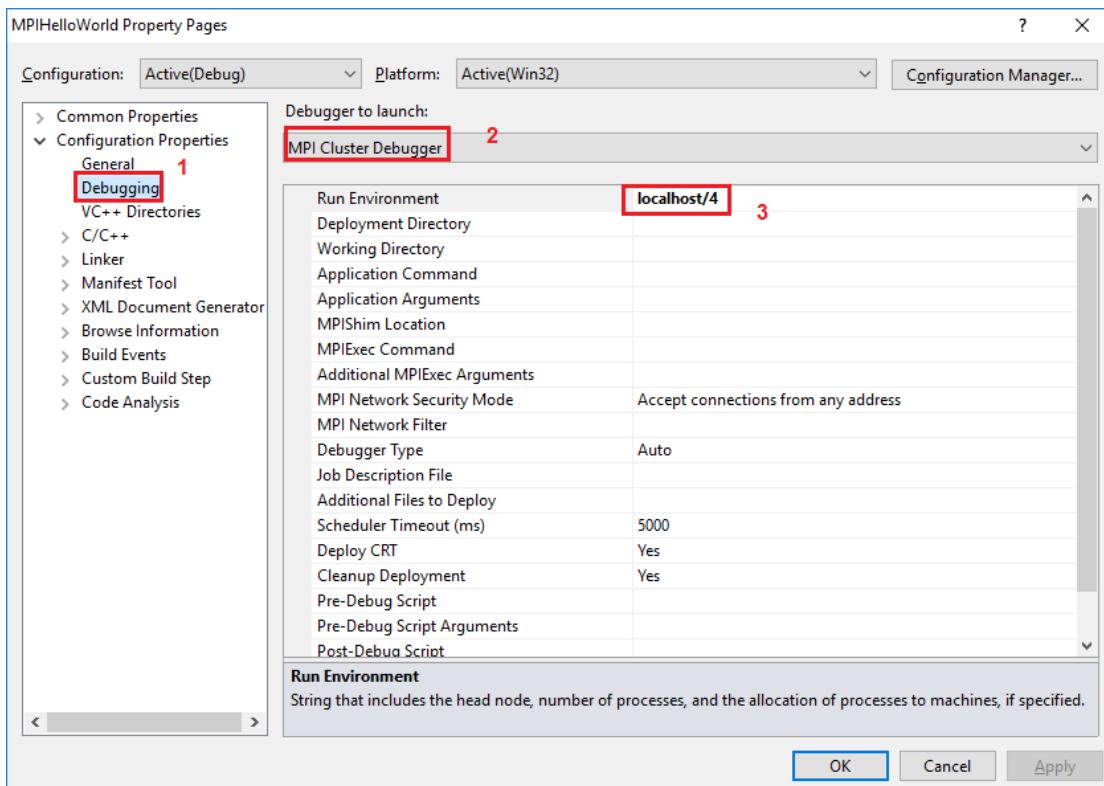
C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x64

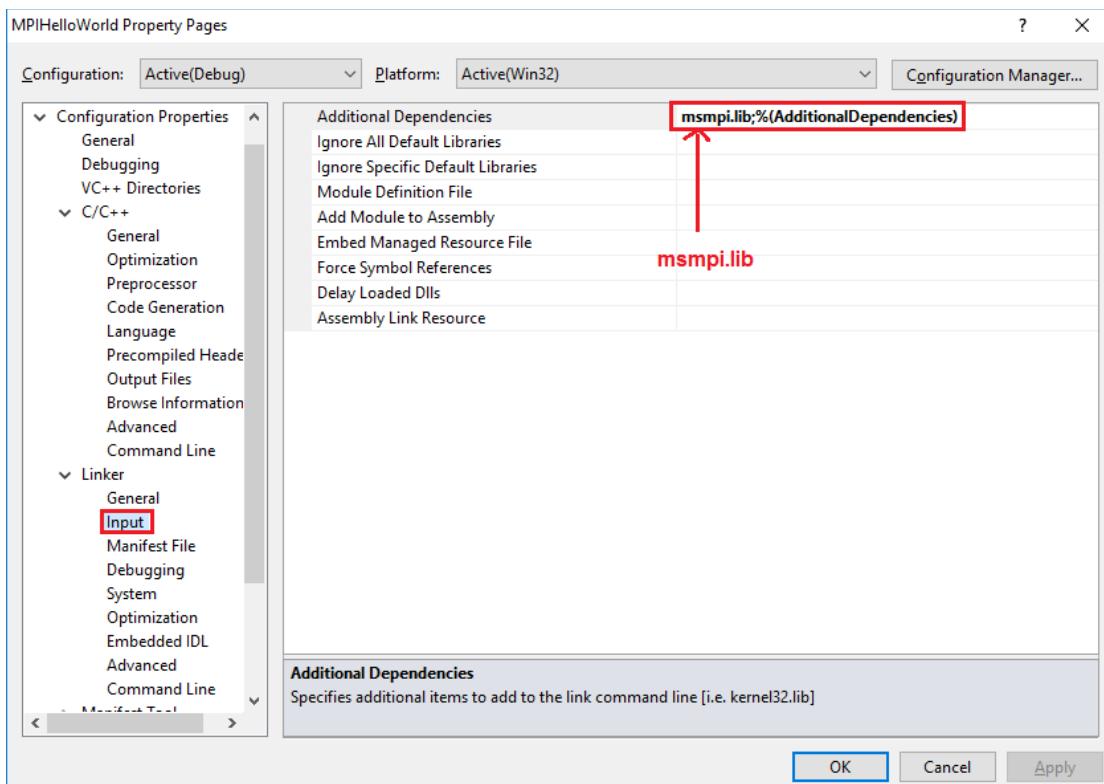
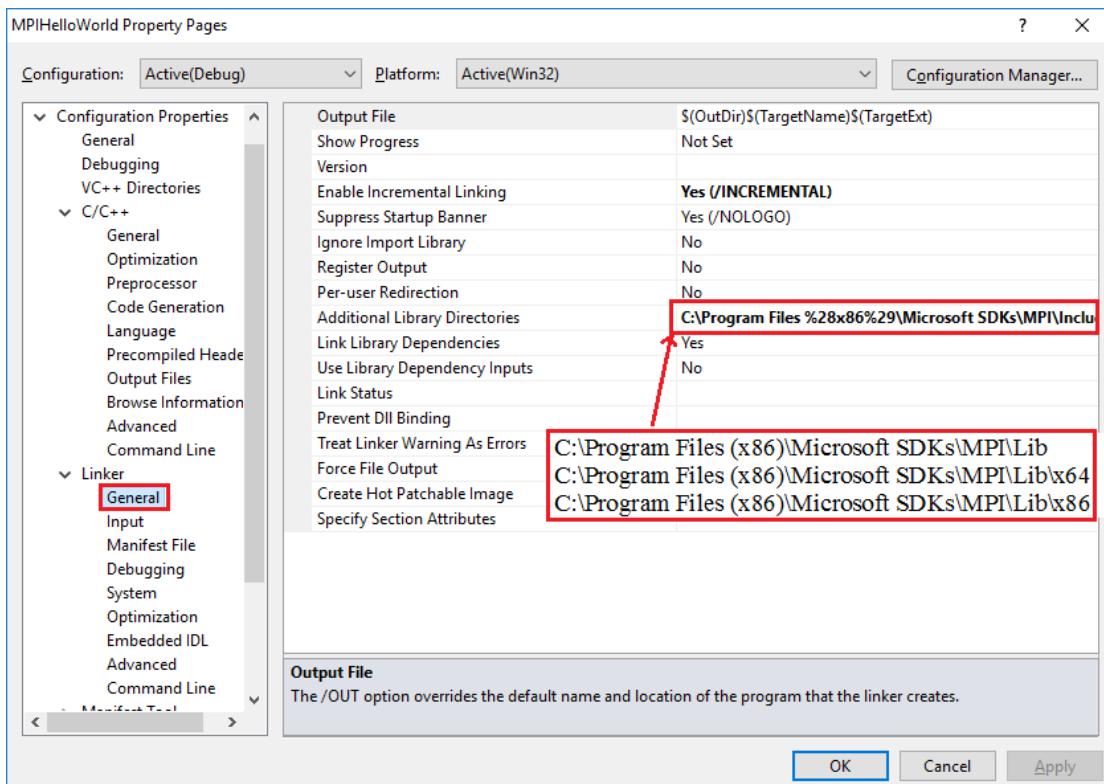
C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x86

8. افتح Link ثم اكتب input MSMPI.LIB

9. في البرنامج ضمن الترويسة MPI.H

تفاصيل الخطوات اعلاه كالتالي:





تعديل البرنامج في المشروع ليصبح كما يلي:

```
#include "stdafx.h"
#include <stdio.h>
#include "mpi.h"
int _tmain(int argc, char* argv[])
{
 MPI_Init(&argc, &argv);
 printf("Aslamu Alikum...");
 MPI_Finalize();
 return 0;
}
```

اخذar Build -> Rebuild Solution أو Build -> Build Solution

نفذ بالامر Ctrl+F5 أو من خلال نافذة الاوامر (وهو الافضل)

سيكون المخرج كما يلي:

```
Aslamu Alikum...
```

## الملحق (٤) تشغيل pThread في فيجوال استديو 2010

1. تحميل pThreads-win32 من الرابط : [www.sourceware.org/pthreads-win32](http://www.sourceware.org/pthreads-win32) اختار [ftp://sourceware.org/pub/pthreads-win32/dll-latest](http://sourceware.org/pub/pthreads-win32/dll-latest)

### Index of /pub/pthreads-win32/dll-lat

| Name               | Size    | Date Modified        |
|--------------------|---------|----------------------|
| [parent directory] |         |                      |
| ANNOUNCE           | 14.0 kB | 5/27/12, 12:00:00 AM |
| BUGS               | 6.3 kB  | 5/27/12, 12:00:00 AM |
| CONTRIBUTORS       | 5.5 kB  | 5/27/12, 12:00:00 AM |
| COPYING            | 5.8 kB  | 5/27/12, 12:00:00 AM |
| COPYING.LIB        | 26.3 kB | 5/27/12, 12:00:00 AM |
| ChangeLog          | 196 kB  | 5/27/12, 12:00:00 AM |
| FAQ                | 17.7 kB | 5/27/12, 12:00:00 AM |
| MAINTAINERS        | 99 B    | 5/27/12, 12:00:00 AM |
| NEWS               | 42.4 kB | 5/27/12, 12:00:00 AM |
| PROGRESS           | 142 B   | 5/27/12, 12:00:00 AM |
| README             | 22.4 kB | 5/27/12, 12:00:00 AM |
| README.Borland     | 2.5 kB  | 5/27/12, 12:00:00 AM |
| README.CV          | 87.7 kB | 5/27/12, 12:00:00 AM |
| README.NONPORTABLE | 31.1 kB | 5/27/12, 12:00:00 AM |
| README.Watcom      | 2.3 kB  | 5/27/12, 12:00:00 AM |
| README.WinCE       | 220 B   | 5/27/12, 12:00:00 AM |
| WinCE-PORT         | 7.9 kB  | 5/27/12, 12:00:00 AM |
| dll/               |         | 3/18/13, 12:00:00 AM |
| include/           |         | 2/5/15, 12:00:00 AM  |
| lib/               |         | 3/18/13, 12:00:00 AM |
| md5.sum            | 799 B   | 2/5/15, 12:00:00 AM  |

المجلدات

2. من الملفات التي حملتها انسخ الملف pthreadVC2.dll إلى مجلد فيجوال C++ (مثلا):

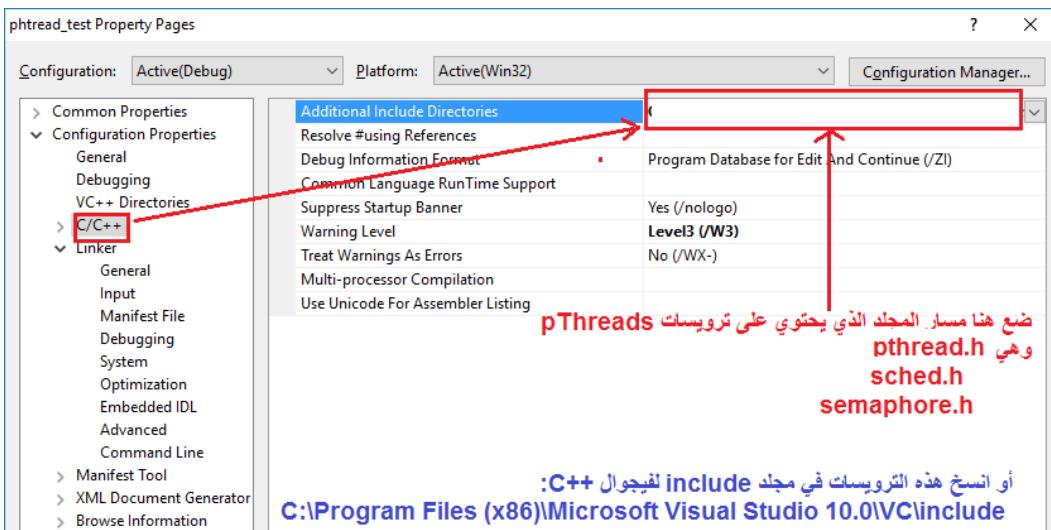
C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin

3. انسخ الملف pthreadVC2.lib إلى المجلد:

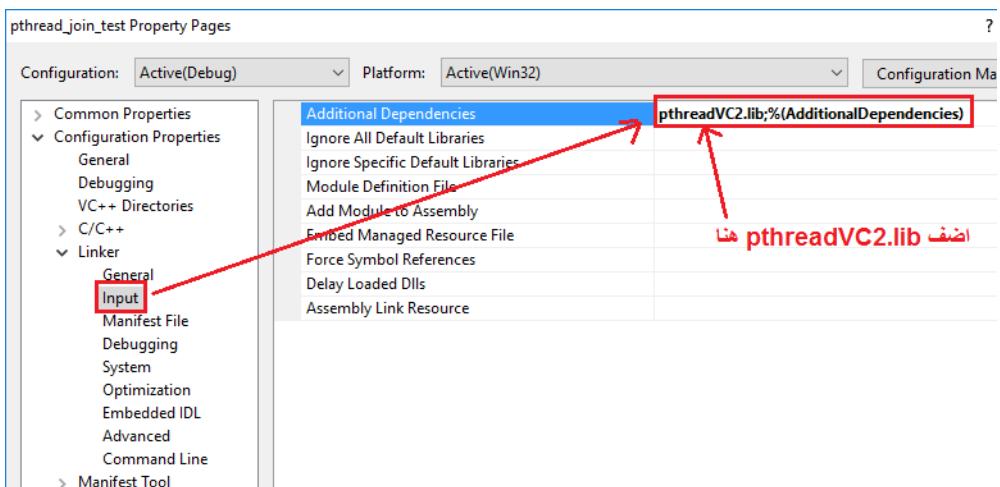
C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\lib

4. انشي مشروع جديد في فيجوال C++ واختار win32 console Application ثم سميء باي اسم وانقر ok

في خصائص المشروع :



5. أضف الملف pthreadVC2.lib إلى Additional dependencies كما في الشكل أدناه:



6. اكتب اي برنامج للتجربة.

```
#include "stdafx.h"
#include <pthread.h>
#include <stdio.h>
#include <iostream>
using namespace std;
#define NUM_THREADS 5
void *PrintHello(void *threadid)
{
 cout << "Hello World! Thread ID, \n";
 return 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 %d\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();
 }
pthread_exit(NULL);
}
```

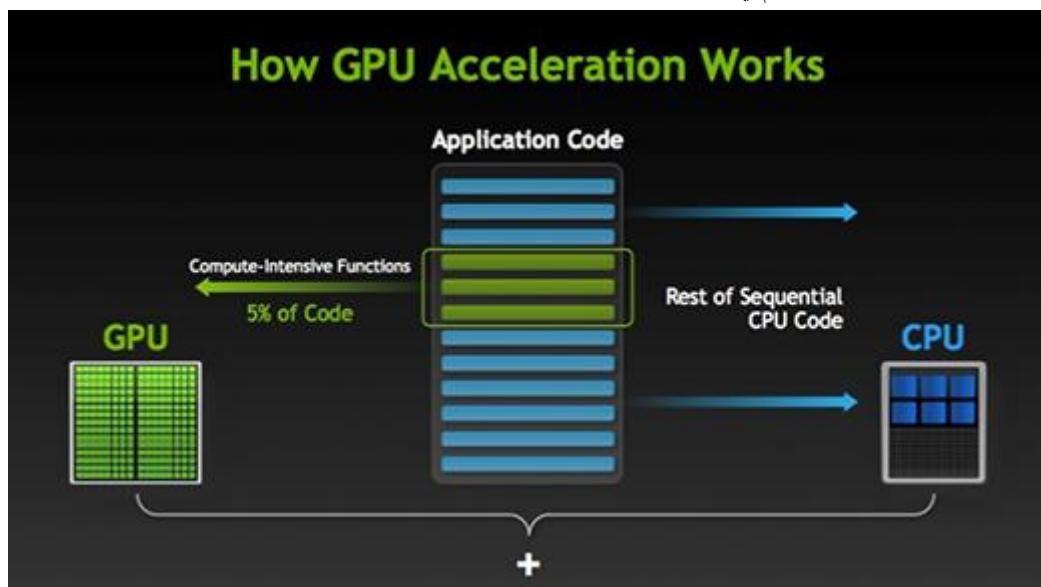
عند التنفيذ بالضغط على Ctrl+F5 سيكون المخرج كما يلي:

```
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
Hello World! Thread ID,
In main: creating thread 3
Hello World! Thread ID,
Hello World! Thread ID,
In main: creating thread 4
Hello World! Thread ID,
Hello World! Thread ID,
=
```

## ملحق (و) وحدات معالجة الرسوميات Graphics processing unit(GPU)

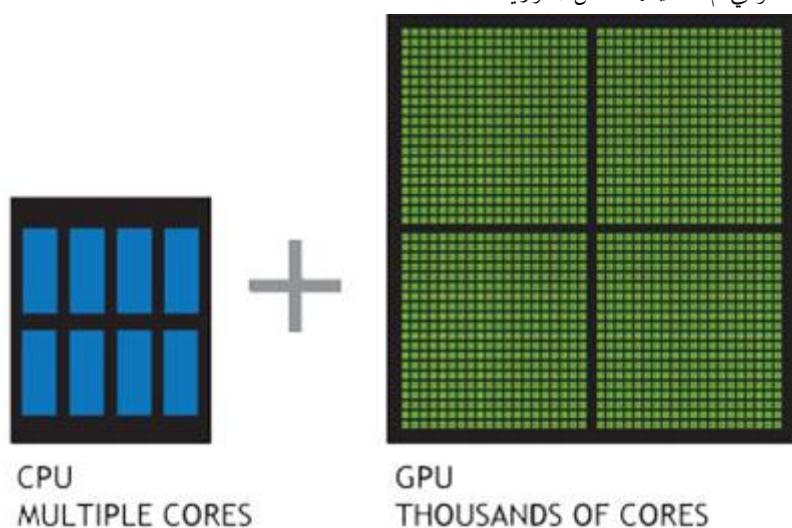
وحدة معالجة الرسوميات (GPU) هي المعالج الملتصق إلى بطاقة الرسوميات المخصصة لحساب عمليات الفاصلية العامة وما شابه ذلك. وهي تُستخدم أساساً لألعاب الفيديو ثلاثية الأبعاد وبرامج معالجة الرسوميات ثلاثية الأبعاد. وقدرها على تنفيذ عدد من العمليات بالتوازي يجعلها تعمل بشكل سريع ويستفاد منها في تنفيذ البرامج الكبيرة بالتوازي مع المعالج CPU.

مسرع الحوسبة (GPU-accelerated computing) يستخدم GPU مع المعالج CPU في زيادة سرعة تنفيذ بعض التطبيقات التي تحتاج قوة معالجة كبيرة. ويتم ذلك عن طريق توزيع جزء من المهام إلى GPU بينما تنفذ بقية المهام في CPU.



<http://www.nvidia.com/object/what-is-gpu-computing.html>

يحتوي المعالج من عدد محدود من الأنوية. المصممة لتعمل تسلسلياً، بينما GPU تمتلك معمارية متوازية ضخمة تضمآلاف الأنوية الصغيرة والفعالة والتي تم تصميماً لها لتعمل بالتوازي.



<http://www.nvidia.com/object/what-is-gpu-computing.html>



## المراجع

- .1 university, T.A.M. *Introduction to OpenMP*. 2016 [cited 2016 6 Sep.]; Available from: [http://sc.tamu.edu/shortcourses/SC-openmp/OpenMPSlides\\_tamu\\_sc.pdf](http://sc.tamu.edu/shortcourses/SC-openmp/OpenMPSlides_tamu_sc.pdf).
- .2 Barney, B. *Introduction to Parallel Computing*. 2016 [cited 2016 28 Aug.]; Available from: [https://computing.llnl.gov/tutorials/parallel\\_comp/](https://computing.llnl.gov/tutorials/parallel_comp/)
- .3 Bharadwaj, P. *How fast (ie: clockspeed in Hz) can a single processor in a computer be?* [cited 2016 29 Aug.]; Available from: <http://www.physlink.com/education/AskExperts/ae391.cfm>.
- .4 Grubb, A. *Market Watch: Dual-Core Processors*. 2006 [cited 2016 6 Sep.]; Available from: <http://sqlmag.com/database-performance-tuning/market-watch-dual-core-processors>.
- .5 M'ehaut, C.o.J.-F.c. *Speed of Light (Fundamental)*. Parallel Systems 2009; Available from: [http://mescal.imag.fr/membres/arnaud.legrand/teaching/2009/01\\_parallel\\_architectures.pdf](http://mescal.imag.fr/membres/arnaud.legrand/teaching/2009/01_parallel_architectures.pdf).
- .6 Anaconda. *Anaconda for cluster management*. [cited 2016 27 Aug.]; Available from: <https://docs.continuum.io/anaconda-cluster/>
- .7 Centre:, G.C.I. *What is Grid*. 2002 [cited 30 2016 Aug.]; Available from: <http://www.gridcomputing.com/gridfaq.html>.
- .8 Luis Ferreira, et al., *Introduction to Grid Computing with Globus*, 2003, IBM.
- .9 Tanenbaum, A.S. and M. Van Steen, *Distributed systems*. 2007: Prentice-Hall.
- .10 Pacheco, P.S., *An Introduction to Parallel Programming*. 2011: Elsevier.
- .11 Schildt, H., *Java: A Beginner's Guide, 3rd Edition*. 2005: McGraw-Hill.
- .12 Buuya, R., S.T. Selvi, and X. Chu, *Object-oriented Programming with Java: Essentials and Applications*. 2009: Tata McGraw-Hill.
- .13 Mittal, A. *Top 80 Thread- Java Interview Questions and Answers (Part 1)*. 2015 [cited 2016 1 Sep.]; Available from: <https://dzone.com/articles/threads-top-80-interview>.
- .14 Mittal, S. *Five Difference Between Implements Runnable And Extends Thread In Java*. 2015 [cited 2016 2 Spe.]; Available from: <http://javahungry.blogspot.com/2015/05/implements-runnable-vs-extends-thread-in-java-example.html>.
- .15 Barney, B. *POSIX Threads Programming*. 2016 [cited 2016 8 Sep.]; Available from: <https://computing.llnl.gov/tutorials/pthreads/>
- .16 Ippolito, G. *POSIX thread (pthread) libraries*. 2004 [cited 2016 8 Sep.]; Available from: <http://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html>.
- .17 Tim Mattson, L.M. *A “Hands-on” Introduction to OpenMP*. [cited 2016 4 Sep.]; Available from: <http://openmp.org/wp/>
- .18 Gerber, R. *Getting Started with OpenMP*. 2012 [cited 2016 5 Sep.]; Available from: <https://software.intel.com/en-us/articles/getting-started-with-openmp>.
- .19 Barney, B. *OpenMP*. 2016 [cited 2016 4 Sep.]; Available from: <https://computing.llnl.gov/tutorials/openMP/>
- .20 Quinn, M.J., *Parallel Programming*. Vol. 526. 2003: TMH CSE.
- .21 Wes Kendall, D.N ,Wesley Bland. *MPI Tutorial*. 2016 [cited 2016 7 Sep.]; Available from: <http://mpitutorial.com>.