

السنة الرابعة

نظم تشغيل 1



المحاضرة الأولى

Introduction to OS

د. المثني خضر

كلية الهندسة المعلوماتية

مفردات المقرر والمراجع

- ❑ مقدمة إلى نظم التشغيل
- ❑ بنية نظام التشغيل
- ❑ مفهوم الإجراءات (Process)
- ❑ النياسب (Threads)
- ❑ جدولة وحدة المعالجة المركزية (CPU Scheduling)
- ❑ تزامن الإجراءات (Process Synchronization)
- ❑ مشكلة التوقف التام (Deadlock)

❖ المراجع

- Operating System Concepts Book (النظري)
- مفاهيم نظم التشغيل للمؤلف عبد الرحمن أحمد محمد عثمان

المحاضرة الأولى: المفردات الرئيسية

1. تعريف نظام التشغيل
2. مهام نظام التشغيل حسب الحاجة
3. مكونات النظام الحاسوبي
4. المتحكم (controller)
5. المقاطعات (Interrupts)
6. الخط الزمني لمقاطعات أجهزة الدخل والخرج
7. Direct Memory Access Structure
8. هرمية الذاكرة
9. واجهة نداء النظام (system call interface)
10. Single & Multi Processing
11. Multiprogramming vs multitasking
12. أنماط عمل نظام التشغيل
13. مهام نظام التشغيل
14. هيكلية نواة نظام التشغيل Kernel Data Structures
15. بيئات الحوسبة Computing Environments
16. نظم التشغيل الأكثر شيوعاً

تعريف نظام التشغيل

• التعريف الأبسط لنظام التشغيل هو أنه برنامج يعمل كوسيط بين المستخدم و أجهزة الحاسوب المادية (Hardware) و يؤمن بيئة مناسبة لتنفيذ البرامج

• أهداف نظام التشغيل الرئيسية هي:

- تنفيذ تطبيقات المستخدم.
- توفير بيئة مناسبة وملائمة للاستخدام convenient.
- الاستفادة القصوى من الموارد وذلك بجعلها تعمل بشكل فعال efficient.

بعض التعاريف الخاصة بنظام التشغيل :

➤ Kernel : هو جزء من نظام التشغيل الذي لا يعمل بدونه (لب نظام التشغيل) و الذي يجب أن يبقى قيد التشغيل طالما الحاسب يعمل و عند تعرضه إلى ضرر يؤدي إلى توقف نظام التشغيل و عدم قدرة الحاسب على العمل.

➤ System programs : هي مجموعة من البرامج التي تؤمن بيئة مناسبة لتنفيذ و تشغيل البرامج, و تقديم خدمات للمستخدمين , ليست جزء من الـ kernel , تأتي مع نظام التشغيل أو يمكن تثبيته لاحقاً مثل (compiler - Assembler - command line) .

➤ Application programs : هي جميع البرامج التي لا ترتبط مباشرة مع نظام التشغيل مثل : (VLC – Adobe – call of duty – Notepad) .

➤ Middleware : هي مجموعة من الـ Software frameworks التي تزود مطورين البرامج بخدمات إضافية مثل (Databases – Multimedia- Graphics) وهي مضمنة ضمن أنظمة التشغيل ذات الاستخدام العام وأيضاً توجد في الحوسبة المحمولة mobile computing.

مهام نظام التشغيل حسب الحاجة

• تنقسم الحواسيب إلى أنواع مختلفة و منها :

□ **Mainframe computers** : عبارة عن جهاز حاسوب مركزي واحد تتصل به عدة طرفيات تسمى (dump terminal) هذه الطرفيات عبارة عن شاشة ولوحة مفاتيح، ليس بها ذاكرة أو معالج وإنما تستخدم معالج وذاكرة الحاسب المركزي.

□ **Minicomputers** : هي فئة من الحواسيب متعددة المستخدمين و تقع في المدى الأوسط من الطيف الحاسوبي بين الأنظمة متعددة المستخدمين الكبرى و الأنظمة أحادية المستخدم الصغرى (الحاسب الشخصي).

□ **Servers** : هو نظام حاسوبي متصل بشبكة أي أنه عقدة فيها و متخصص في أداء وظيفة معينة و تلبية الطلبات التي ترده من حواسيب أخرى على الشبكة و عادة ما يكون عتاده المادي ذات إمكانيات متفوقة و تصميمات خاصة لتحمل العمل لفترات طويلة بلا انقطاع و لمواجهة الأعطال بكفاءة أكبر.

□ **Embedded system** : هي أنظمة لمعالجة المعلومات و تكون مضمنة في منتج أكبر و هي عادة ما تكون غير مرئية مباشرة للمستخدم و تكون واجهاتها قليلة او معدومة أحيانا مثل الأنظمة الموجودة في السيارات الحديثة.

مهام نظام التشغيل حسب الحاجة

□ **المستخدمون** يريدون أن يكون نظام التشغيل سهل الاستخدام و أن يعطي أداء عالي من دون الاهتمام بما يقوم من استهلاك لموارد الحاسب.

□ بينما **الحواسيب المشتركة** مثل ال mainframe أو ال minicomputer على نظام التشغيل إدارة الجهاز المركزي لخدمة الكم الكبير من المستخدمين بحيث يكون زمن الاستجابة لكل مستخدم سريع (أقل من ثانية)، وحتى يشعر كل مستخدم أن الجهاز المركزي يخدمه لوحده.

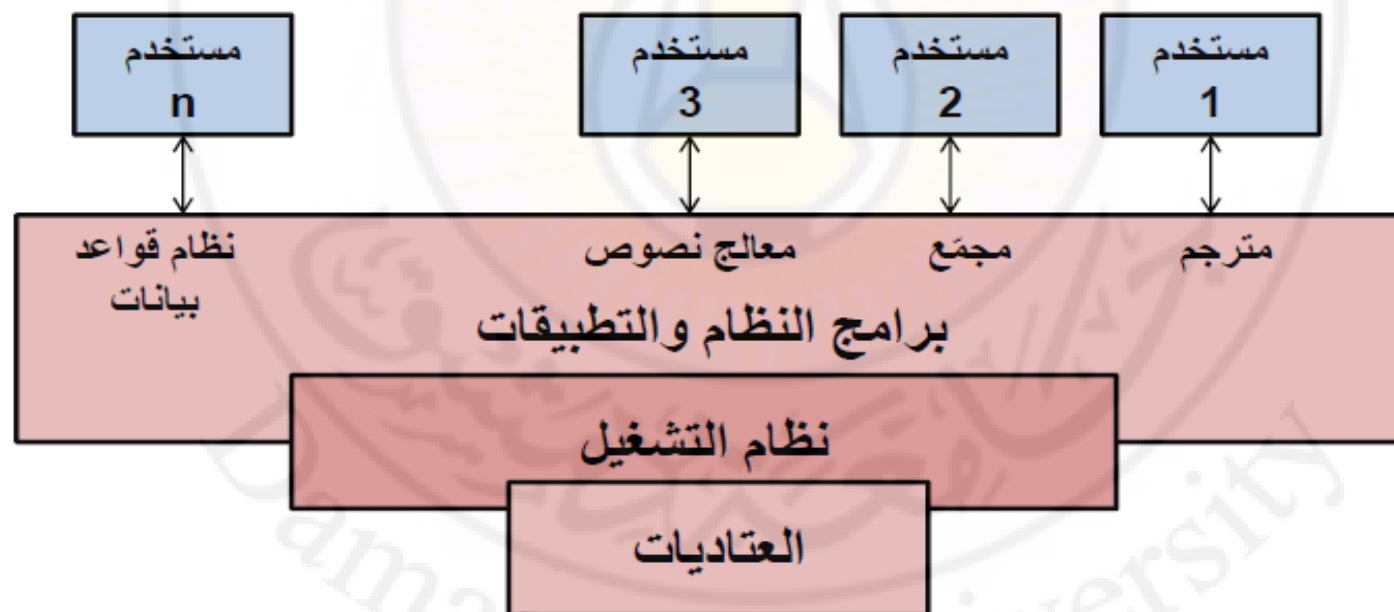
□ مستخدم **الأنظمة المخصصة** مثل ال workstations لديها موارد مخصصة ولكنها غالباً تتشارك في الموارد من ال servers , وعلى أنظمة التشغيل تأمين أدوات لإدارة الخدمات التي يقدمها السيرفر.

□ **الأجهزة المحمولة** mobile devices مثل الهواتف الذكية وال tablets هي أجهزة تمتلك موارد ضعيفة ويتم تطوير هذه الأجهزة من أجل تحسين الاستخدامية و عمر البطارية , تكون أنظمة التشغيل في هذه الحالة تستجيب لل touch screens و التعرف الصوتي voice recognition.

□ بعض الحواسيب تمتلك واجهات قليلة أو لا تمتلك واجهات أساساً , مثل الأنظمة الموجودة في embedded computers (**حواسيب مضمّنة**) , وهذه الأنظمة تعمل بشكل أساسي دون أي تدخل من المستخدم .

مكونات النظام الحاسوبي

- يقسم **النظام الحاسوبي** إلى أربعة مكونات: العتاديات (hardware)، نظام التشغيل، البرامج التطبيقية، والمستخدمين.
- العتاديات: وحدة المعالجة، الذاكرة، أجهزة الإدخال والإخراج (موارد الحاسوب).
- البرامج التطبيقية: معالجات النصوص، مترجم، نظام قاعدة بيانات، تمثل الأساليب التي تستخدم بها موارد الجهاز لحل مشاكل المستخدمين.
- يتولى نظام التشغيل مهمة مراقبة وتنسيق استخدام العتاديات بين مختلف البرامج التطبيقية لمختلف المستخدمين.



المتحكم (controller)

□ لكل جهاز أو لكل طرفية يوجد **متحكم controller** أو ما يسمى أيضا ب **Adapter** يتحكم به, قد يكون مدمجا بالجهاز أو أحيانا قد يكون خارجيا (مثل كرت الشاشة هو controller يتحكم بالشاشة) ولكنه ليس جزء من نظام التشغيل .

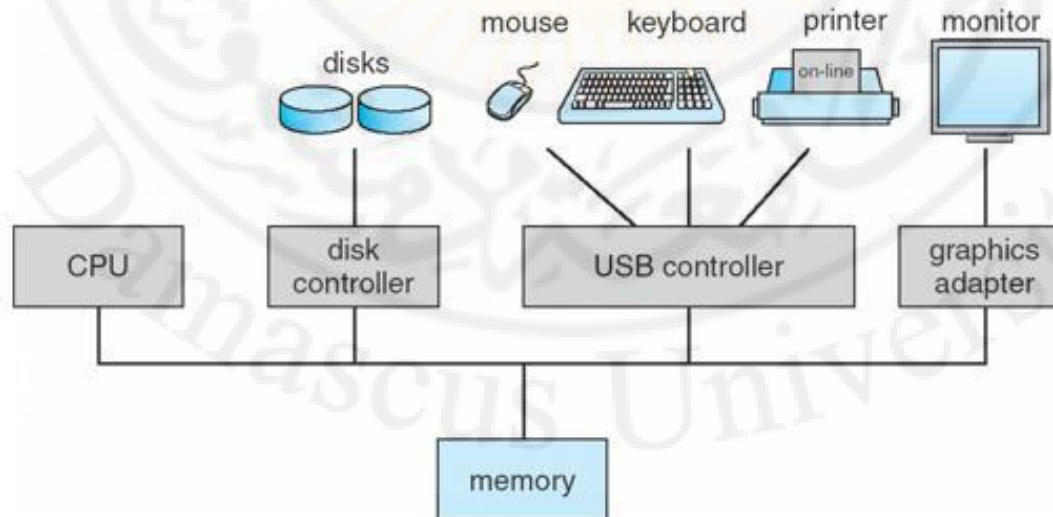
□ ويتعامل نظام التشغيل مع ال Controller عن طريق software يدعى **Device driver** .

□ كل متحكم يحتوي على مخزن محلي مؤقت (**Buffer**) .

□ يوجد أكثر من CPUs, device controllers متصلة من خلل خطوط النقل المشتركة (**Bus**) توفر الوصول إلى الذاكرة المشتركة .

□ التنفيذ **المتزامن** لوحدات المعالجة المركزية والأجهزة يجعل منهم يتنافسون على دورات الذاكرة حيث أن الذاكرة يمكنها أن تتعامل مع جهاز / cpu واحد في الدورة الواحدة .

□ يقوم المتحكم بإعلام المعالج أنه قد أنهى عمله عن طريق التسبب في مقاطعة **interrupt** .



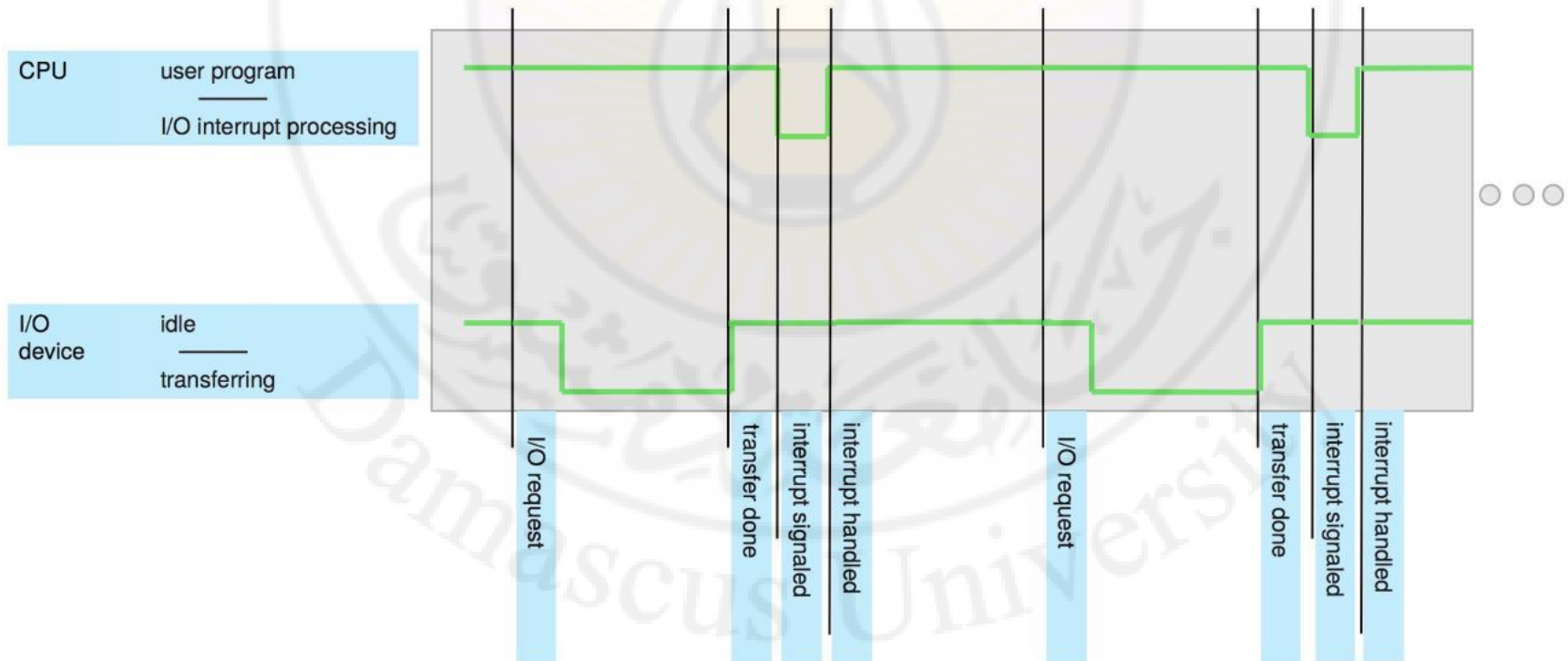
المقاطعات (Interrupts)

- المقاطعة : هي إشارة أو طلب ترسل من المكونات المادية أو المكونات البرمجية إلى المعالج من أجل تنفيذ **خدمة مستعجلة** (interrupt service routine) .
- هنالك مجموعة من الخدمات تختلف باختلاف نظام التشغيل، وهي عبارة عن دوال خاصة بالمقاطعات، فكل مقاطعة تقابلها مجموعة من الدوال.
- عندما يتم إرسال المقاطعة إلى المعالج يوقف ما كان يعمل فيه، ثم ينفذ دالة معينة من دوال المقاطعة (تحدد الدالة برقم في سجل معين)، ثم **يرجع المعالج مرة أخرى ليوصل ما أوقفه** قبل استلامه للمقاطعة.
- هنالك **أنواع مختلفة من المقاطعات** :
- Hardware interrupt : تستخدمها ال Hardware devices للتخاطب مع نظام التشغيل عندما تريد طلب عمل ما منه وهي عبارة عن نبضة كهربائية .
- Software interrupt : وهي عبارة عن Trap أو Exception و تحدث لثلاثة أسباب :
 - عند حصول خطأ برمجي ما . (division by zero)
 - طلب معين من برنامج ليقوم بتنفيذه نظام التشغيل (System call)
 - مشاكل برمجية أخرى (Infinity loop) أو عمليات تقوم بتعديل بعضها البعض .

الخط الزمني لمقاطعات أجهزة الدخل والخرج

• القمم في وقت وحدة المعالجة المركزية هي عندما تكون وحدة المعالجة المركزية مشغولة بالقيام بمهمة , و الوديان هي عندما يتم إعلام وحدة المعالجة المركزية بمقاطعة من ال I/O وعليه المضي قدما في إعداد المخازن المؤقتة للجولة المقبلة .

• القمم في ال I/O تكون عندما ينتظر خرج او طلبات دخل وخرج , والوديان عندما متحكم الجهاز يقوم بكتابة مخازن الذاكرة بشكل مستقل من وحدة المعالجة المركزية الى الجهاز الفعلي وغالبا ما تتم الكتابة عن طريق ال (DMA) الوصول المباشر للذاكرة.



الخط الزمني لمقاطعات أجهزة الدخل والخرج

أثناء المقاطعة يحافظ نظام التشغيل على الحالة الحالية لوحدة المعالجة المركزية من خلال تخزين السجلات registers وعداد البرنامج program counter ويقوم بتحديد نوع المقاطعة التي حصلت.

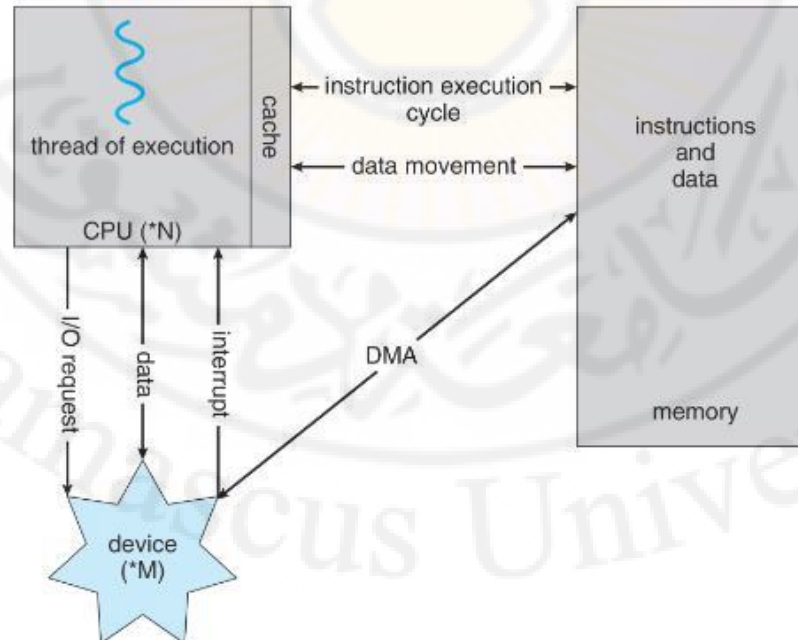
• عندما يطلب أحد البرامج عملية دخل أو خرج , فنظام التشغيل يقوم بعملية مقاطعة و نكون أمام حالتين:
أما الانتظار: * يعود التحكم إلى برنامج المستخدم فقط بعد إتمام عمليات ال I/O
* تعليمات الانتظار تجعل وحدة المعالجة المركزية بحالة خمول حتى المقاطعة التالية.
* حلقة الانتظار تؤدي إلى التنارع بين الموارد للوصول إلى الذاكرة.
* من الممكن أن يتواجد طلب I/O واحد على الأكثر في الوقت الواحد.

أو الاستمرار: * يعود التحكم إلى برنامج المستخدم دون انتظار انتهاء عمليات ال I/O.
* يفهرس نظام التشغيل جداول أجهزة I/O لتحديد حالة الجهاز وتعديل سطور الجدول لتضمين المقاطعة.
* Device-status table جدول حالة الجهاز يحتوي على دخل لكل جهاز I/O يشير إلى نوعه و عنوانه وحالته.

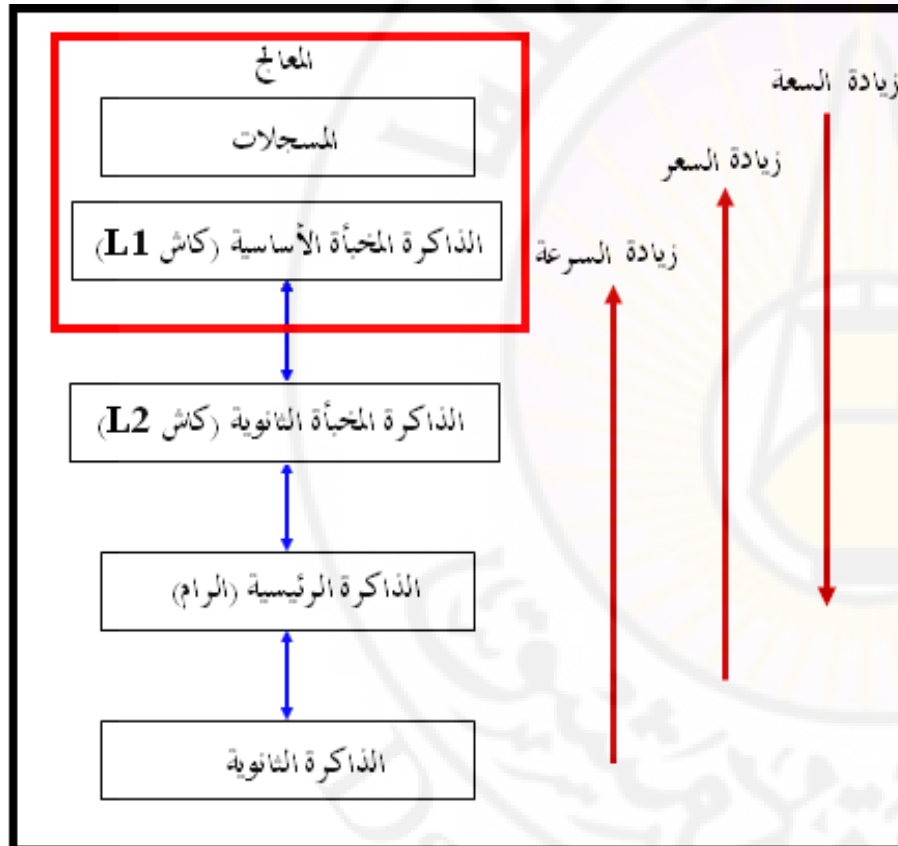
Direct Memory Access Structure

هي عملية نقل مباشر بين أجهزة الدخل والخرج والذاكرة الرئيسية دون تدخل المعالج و تُدار هذه العملية عن طريق ال device controller الخاص بالذاكرة الرئيسية.

- تستخدم لأجهزة I/O عالية السرعة القادرة على إرسال المعلومات بسرعة قريبة من سرعة ال RAM.
- Device controller ينقل blocks of data من المخازن المؤقتة buffer بشكل مباشر الى الذاكرة الرئيسية دون تدخل ال CPU.
- يتم إنشاء مقاطعة واحدة فقط لكل block، بدلا من المقاطعة واحدة لكل بايت .



هرمية الذاكرة



- تتكون **هرمية الذاكرة** من التالي:
 - المسجلات registers
 - الذاكرة المخبأة (الكاش) cache
 - الذاكرة الرئيسية (الرام) main memory
 - الأقراص الممغنطة magnetic disk
 - الأقراص الضوئية optical disks
 - الأشرطة الممغنطة magnetic tape
- هنالك عوامل تؤثر في هرمية الذاكرة هي :
التطاير – السرعة – السعة – السعر.

- أعلى الهرم: يخزن البيانات تخزيناً مؤقتاً يزول عند إغلاق الحاسب (**متطايرة**) volatile
- أسفل الهرم: يخزن البيانات تخزيناً دائماً لا يزول (**غير متطايرة**) non-volatile

واجهة نداء النظام (system call interface)

هو عبارة عن Programming Interface يسمح للبرمجيات و التطبيقات بأن تطلب خدمات من نظام التشغيل, و هو أيضاً عبارة عن مجموعة من الاكواد المخزنة في جدول , يحتوي هذا الجدول على حقلين : الحقل الأول رقم المقاطعة والحقل الثاني هو أكواد المقاطعة .

• كيفية عمل نداءات النظام

• يتم تنفيذ نداء النظام في وضع النواة kernel mode ولكل استدعاء نظام رقم مرتبط به:

1. يرسل هذا الرقم إلى النواة ليعرف نظام التشغيل ما هو استدعاء النظام المطلوب.
2. عندما يرسل المستخدم هذا الرقم فهو في الحقيقية يستدعي روتين مكتبة (library routine) ، فيقوم الروتين بإرسال مقاطعة (issues a trap) لنظام التشغيل، ثم يمرر رقم الاستدعاء ومعطياته إلى النواة باستخدام مسجلات معينة .
3. تقوم النواة بتنفيذ الروتين وترسل النتائج للمستخدم عبر مسجل معين.
4. إذا كانت النتائج كبيرة الحجم لا يستطيع المسجل تخزينها، سترسل بطريقة أخرى مثل استدعاء الروتين copy_to_user لتخزينها في موقع ما بالذاكرة.

Single & Multi Processing

Single Processor: وهو معالج يستخدم عندما يكون للحاسب أو الجهاز المراد برمجته هدف عام وحيد يعمل عليه ويهتم به .

Multi Processors: تعرف الأنظمة الخاصة بها بالأنظمة التفرعية (Parallel Systems) أو (tightly-coupled systems) والمعنى بالنهاية واحد , حيث يكون لدينا أكثر من معالج ضمن نطاق اتصال متقارب ويتشاركون موارد الحاسب (ملحقات – ذاكرة – نواقل Bus، ...)

وتم استخدام هذا النوع من المعالجات في البداية مع الخدمات services ثم أخذ ينتشر شيئاً فشيئاً مع باقي أجهزة الحاسب كالحواسب المكتبية أو حتى المحمولة واليوم أصبحنا نرى أيضاً أن المعالجات المتعددة وصلت إلى الهواتف الذكية

من ميزات المعالجات المتعددة :

- زيادة الإنتاجية . أي عمل أكثر خلال زمن أقل .
- اقتصادي من حيث الكلفة والتوفير .
- وثوقية أكثر: أي عندما يعمل نظام معين على أكثر من معالج فإن فشل معالج ما بشكل فجائي لن يؤدي إلى توقف عمل النظام ككل بل على الأكثر من الممكن أن يؤدي ذلك إلى بطء عام،

المعالجات المتعددة Multi Processors

يوجد نوعان للمعالجات المتعددة :

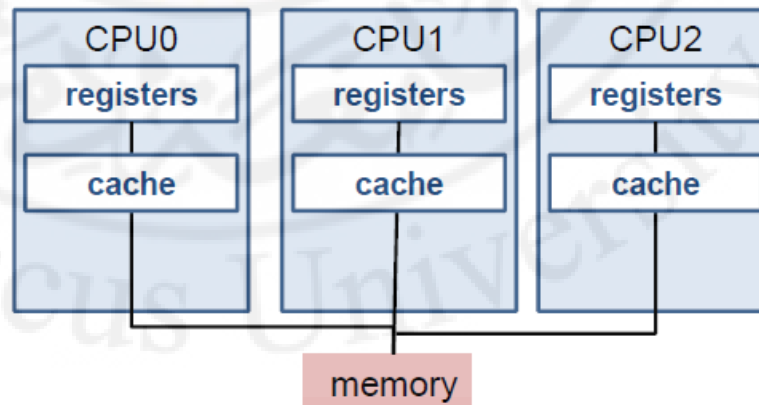
المعالجات المتماثلة Symmetric multiprocessing (SMP)

هنا كل المعالجات متساوية وينفذ كل معالج مهمة منفصلة، وتتصل هذه المعالجات مع بعضها عند الحاجة. يحجز كل معالج مسجلاته الخاصة (Registers) بالإضافة إلى ذاكرة مخبئية خاصة به، بينما من ناحية مورد الذاكرة الفيزيائية (Physical Memory) تقوم المعالجات بالتشارك به.

المعالجات غير المتماثلة Asymmetric multiprocessor

يوجد معالج رئيسي واحد يتحكم في النظام وينفذ مهمة رئيسية كبيرة، بينما بقية المعالجات تنفذ ما يأمرها به المعالج الرئيسي، فهو قد يقسم المهمة الكبيرة إلى أجزاء صغيرة يوزعها على بقية المعالجات ثم يجمع النتائج بعد التنفيذ.

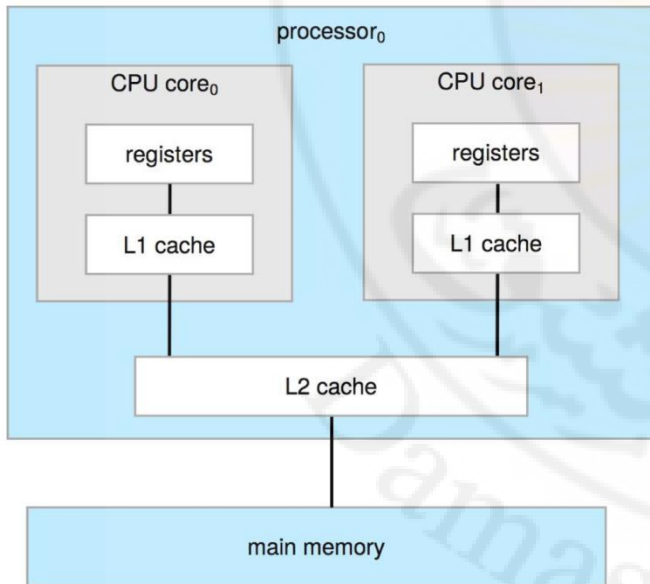
Symmetric multiprocessing



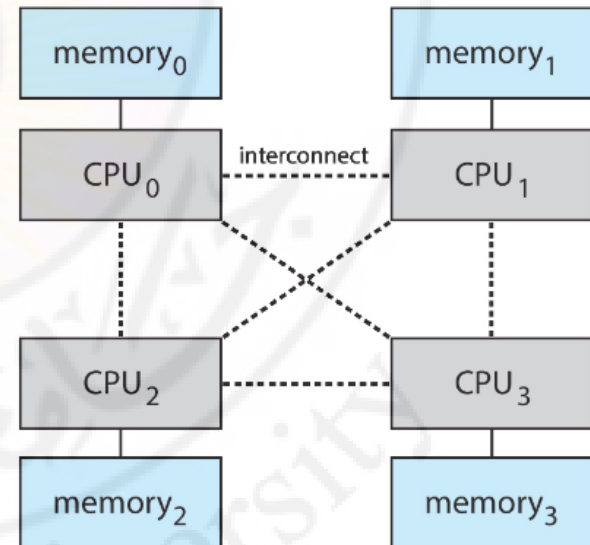
التصميم ثنائي النوى (Dual-core Design)

إن النوع السابق من المعالجات المتعددة كان يستخدم رقاقة **chip لكل معالج** بينما في النمط الجديد – المعالجات متعددة الأنوية - **multicore** فأصبحنا نستخدم رقاقة واحدة لأكثر من معالج بنفس الوقت (من الممكن **multichip & multicore**) مما وفر الطاقة المستهلكة وزاد الفعالية وبالتالي حصلنا على سرعة أكبر .

السرعة في هذا النوع هي بسبب أن التواصل بين المعالجات الموجودة على رقاقة واحدة هو أسرع من التواصل بين المعالجات الموجودة على أكثر من رقاقة



One chip for two core



Four chips for four cpu's

Multiprogramming vs multitasking

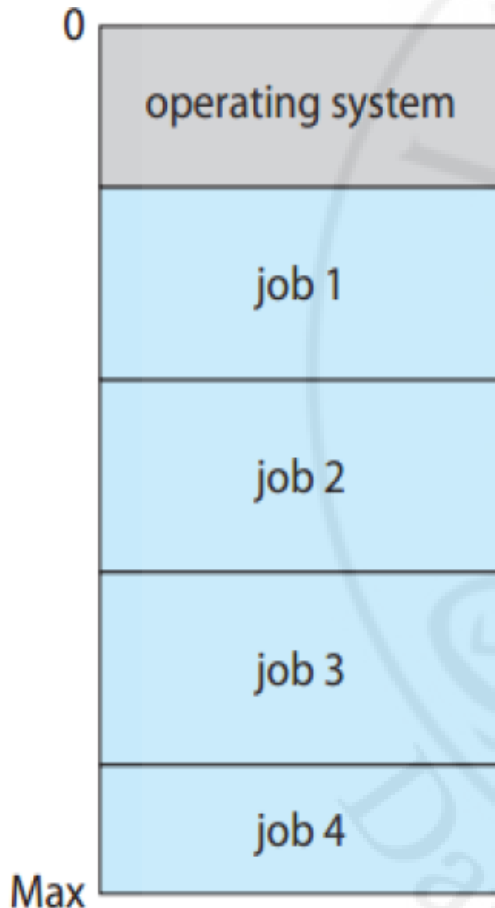
Multiprogramming

□ هي ضرورية لزيادة الفعالية، لأنه لا يمكن للمعالج و لأجهزة الدخل والخرج أن يكونين مشغولين في آن واحد طوال الوقت , فلابد من أوقات أن يكونا فيها ينتظران أعمال أو في حالة فراغ فعندها يجب ملئ هذا الفراغ بأعمال أخرى.

□ وتقوم هذه التقنية بوضع المهام في الذاكرة ومن ثم تقوم بسحبها واحدة تلو الأخرى بطريقة ال **job scheduling** لينفذها المعالج, وفي حالة المعالج كان في حالة انتظار فإنه يقوم بتنفيذ عمل آخر وذلك بأمر من نظام التشغيل, أي أن المعالج يقوم باستغلال الوقت المهدور في انتظار عمليات أخرى .

□ من الأمثلة لهذا النوع نظام التشغيل ويندوز XP

□ على عكس المعالج في تقنية non-multiprogrammed system فإن المعالج سيبقى في حالة خمول وذلك عند انتظاره لتنفيذ مهمة ما, من الأمثلة لهذا النوع نظام التشغيل (DOS).



الشكل العام للذاكرة RAM في
نظام multiprogrammed

Multiprogramming vs Multitasking

Timesharing (Multi-tasking)

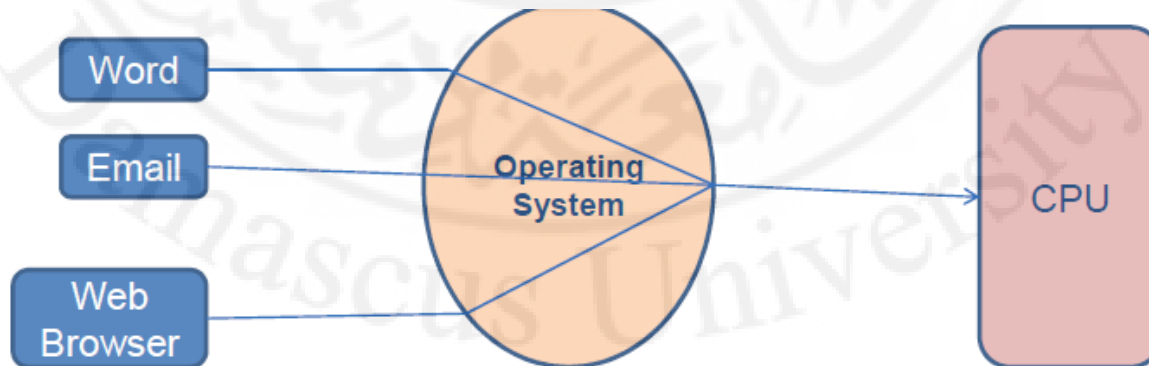
□ يقوم المعالج بخدمة العديد من المهام في وقت واحد وذلك بإعطاء كل مهمة حصة زمنية قصيرة تسمى (quantum).

□ ويتنقل المعالج بين المهام بسرعة عالية جدا لدرجة أن كل مهمة تعمل وكأنها تستخدم المعالج لوحدها.

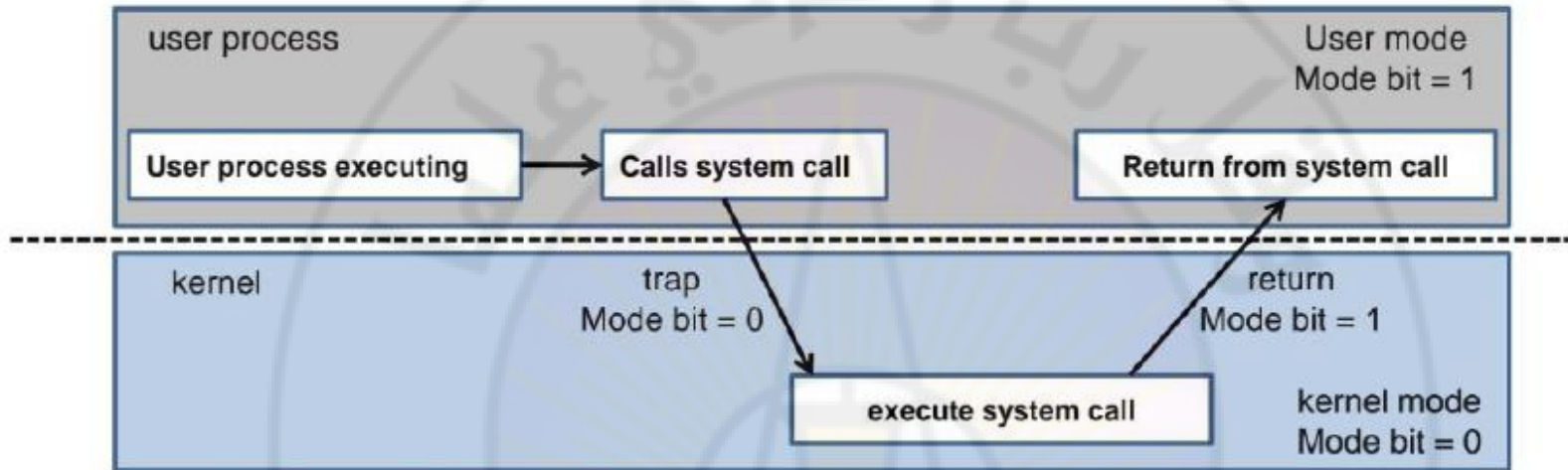
□ وشرط في ال Time sharing أن يكون لكل مستخدم على الأقل مهمة واحدة في الذاكرة وفي حال طلب تنفيذ أكثر من عملية في آن واحد فهذا يأتي دور ال **CPU Scheduling** لتنظيم هذه العملية.

□ وفي حال عدم اتساع الذاكرة للعمليات فيتم وضع بعض العمليات في ال Virtual memory والتي تسمح أن يتم تنفيذ جزء من العمليات خارج الذاكرة.

□ والميزة الرئيسية التي تقدمها الذاكرة الافتراضية بأنها تسمح للمستخدمين بتنفيذ برامج حجمها أكبر من الذاكرة الفيزيائية



أنماط عمل نظام التشغيل



- تسمح عملية **Dual-mode** بحماية نظام التشغيل ومكوناته الأخرى.
 - نمط المستخدم (تتعامل مع البرامج وواجهات المستخدم) - user mode
 - نمط النواة (تتعامل مع العتاديات) - system mode – supervisor mode – privileged mode.
- يهتم **نمط المستخدم** بالواجهة الفعلية بين المستخدم والنظام، ويتحكم بتشغيل التطبيقات والنفاز إلى الملفات.
- يهتم **نمط النواة** بكل ما يجري في الخلفية (background)، ويتحكم بالنفاز إلى الموارد، ومعالجة تعليمات البرامج والتحكم بوظائف العتاديات.
- يُحدّد النمط من خلال بت يسمى mode bit (نمط النواة 0، نمط المستخدم 1).

المؤقت (timer)

- من مهام نظام التشغيل التحكم في المعالج ومنع برامج المستخدمين من الاستئثار بهذا المورد الهام والعمل لمدة طويلة داخل المعالج.
- إذا كان هنالك برنامج ينفذ في تكرار غير منتهي **infinite loop** أو استدعى دالة خدمة ولم يُعيد السيطرة لنظام التشغيل، فهذا يسبب إهدار لزمان المعالج. هنا لابد لنظام التشغيل من آلية تمكنه من توقيف مثل هذه البرامج، وكان المؤقت **timer** هو الحل.

كيف يعمل المؤقت:

- قبل تشغيل برنامج المستخدم يتأكد نظام التشغيل من أن المؤقت مرتبط مع مقاطعة.
- يتم إعداد المؤقت ليصدر المقاطعة بعد فترة محددة (لا تزيد عن ثانية).
- يكون هنالك عداد وساعة لحساب هذه الفترة.
- يقوم نظام التشغيل بتجهيز العداد.
- كل دقة ساعة تنقص العداد.
- عندما يصل العداد صفر تصدر المقاطعة وينتقل التحكم تلقائياً إلى نظام التشغيل.
- لنظام التشغيل حق التصرف في اعتبار أن هذه المقاطعة خطأ جسيم **fatal error** أو قد يعطي البرنامج مهلة أكثر (زيادة زمن).

مهام نظام التشغيل

• يقوم نظام التشغيل بالمهام التالية :

1. **Process management** إدارة العمليات : عن طريق مدير العمليات .
2. **Memory management** إدارة الذاكرة الرئيسية : عن طريق مدير الذاكرة.
3. **File-system Management** إدارة الملفات : عن طريق مدير الملفات.
4. **Device Management** إدارة الأجهزة : عن طريق مدير الأجهزة الذي يقسم إلى
 - **Mass-Storage Management** إدارة التخزين.
 - **I/O Management** إدارة أجهزة الدخل والخرج .
5. **Protection and Security** الأمن والحماية .

سنقوم بدراسة كل منها بالتفصيل

Process management إدارة العمليات

• ما هي الـ process: هي عبارة عن برنامج source code في حيز التنفيذ.

ومن هنا جاء التصنيف التالي لكل من الـ process والـ program :

- حيث اعتبر الـ program أنه passive entity وذلك لأنه عبارة عن تعليمات موجودة في الذاكرة الثانوية "hard disk" حيث تكون قائمة أي لم توضع ضمن حيز التنفيذ.
- في حين اعتبرت الـ process بأنها active entity وذلك لأنها عبارة عن مجموعة من البرامج الموجودة في الذاكرة الرئيسية "RAM" حيث تكون فعالة أي يتم تنفيذها من قبل المعالج.

• مصطلح process termination أو عملية إنهاء تنفيذ الـ process ما هي إلا عملية تحرير لجميع الموارد التي تستهلكها العملية المنهية والقابل استخدامها لعمليات أخرى .

- إن نظام التشغيل مسؤول عن عدد من الأنشطة المرتبطة بعملية إدارة العمليات, من هذه الأنشطة:
 - إنشاء و إنهاء الـ process المفصلة من قبل user أو الخاصة بالـ system بحد ذاته .
 - الإيقاف المؤقت لتنفيذ الـ process واستئناف التنفيذ من حيث تم إيقافه .
 - إيجاد الآلية المناسبة من أجل تزامن العمليات process synchronization.
 - إيجاد الآلية المناسبة من أجل التواصل بين العمليات process communication.
 - إيجاد الآلية المناسبة من أجل deadlock handling.

Memory management إدارة الذاكرة الرئيسية

• بما أن ال process هي مجموعة من instructions، فلا بد أن تكون هذه التعليمات أو جزء منها ضمن الذاكرة RAM عندما يبدأ التنفيذ .

من عمليات إدارة الذاكرة هي تحديد ما يوجد ضمن الذاكرة ومتى يوجد، مما يقوم بتحسين استخدام المعالج CPU وتحسين استجابة الحاسوب للمستخدمين .

• من الأنشطة التي تعنى فيها ال Memory Management أيضا:

- تتبع جميع أجزاء الذاكرة التي تعمل في الوقت الراهن ومعرفة هوية العمليات العاملة بها .
- تحديد أي ال processes و أي data يجب أن تدخل إلى ال RAM، وأيها يجب أن نحرر الذاكرة منها .
- تخصيص وتحرير مساحات من الذاكرة حسب الحاجة .

File-system Management إدارة الملفات

- نظام التشغيل يؤمن طريقة موحدة ومنطقية لتخزين المعلومات , ويتم تحقيق هذا الأمر من خلال:
 - عملية تجريد الخصائص الفيزيائية للمعلومات ليتم تخزينها بوحدات منطقية على شكل ملفات files.
 - كل بيئة تخزينية يتم التحكم بها من قبل جهاز معين (disk drive, tape drive ,,etc).
 - تنوع الخصائص التي تتمتع بها كل بيئة تخزينية والمتمثلة ب access speed, capacity, data transfer rate

• وبما أن المعلومات في وحدات التخزين تعرض بشكل ملفات, تطلب هذا الأمر وجود File-System manager (مدير الملفات), وتتمثل الأنشطة التي يقوم بها بالتالي:

- 1) إنشاء وحذف الملفات وال-directories.
- 2) النقل والتعديل بالملفات وال-directories.
- 3) تنظيم الملفات ضمن directories.
- 4) التحكم بصلاحيات الوصول في معظم الأنظمة لتحديد من يمنح سماحية الوصول وإلى ماذا يمنح .
- 5) عملية وضع الملفات على وحدات التخزين الثانوية secondary storage.
- 6) عمليات ال Backup للملفات على وحدات التخزين المستقرة الغير متطايرة " non-volatile

Mass-Storage Management إدارة التخزين.

• عادة تستخدم الأقراص من أجل تخزين البيانات التي لا تتسع ضمن الذاكرة الرئيسية أو من أجل حفظ البيانات لمدة زمنية طويلة , وبالتالي فإن الإدارة السليمة لعمليات التخزين هي ذات أهمية كبيرة حيث أن سرعة كافة العمليات المنفذة ضمن الحاسوب متعلقة بالأنظمة الفرعية لهذه الأقراص والخوارزميات المطبقة ضمنها.

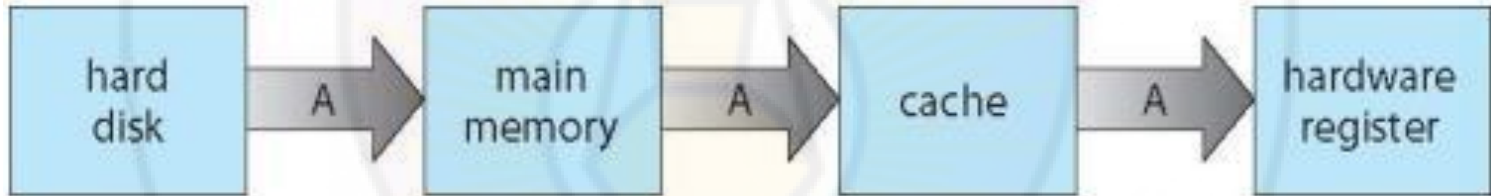
- الأنشطة التي يديرها نظام التشغيل في هذا الخصوص :
- إدارة عمليات تحرير المساحة free space management.
- عمليات ال mounting و ال unmounting .
- عمليات الحجز من أجل التخزين storage allocation.
- وعمليات الجدولة للأقراص disk scheduling.
- عملية تجزئة وحماية القرص partitioning & protecting.

ملاحظة : بعض أجهزة التخزين ليس من الضروري أن تعمل بسرعة , مثل أجهزة التخزين ما بعد الثانوية tertiary storage والتي تشمل , magnetic storage , optical storage ولكنها ما تزال تحتاج الإدارة من قبل نظام التشغيل .

Caching ذاكرة الكاش

- هي ذاكرة سريعة جدا تقارب سرعة المعالج , صغيرة الحجم و مرتفعة الثمن.
- يتم أولاً فحص التخزين الأسرع (ذاكرة التخزين المؤقت الكاش) فيما كان يحتوي على المعلومات التي نريدها ونميز حالتين :

- ✓ إذا احتوت الكاش ما نريد , فالمعلومات تستخدم مباشرة من الكاش (عملية سريعة).
- ✓ إن لم تحتوي الكاش على ما نريد، تنسخ البيانات من الذاكرة الأبطأ إلى الكاش وتستخدم هناك.



- البيانات التي تعتمد تعددية المهام multitasking لابد أن تكون حريصة في استخدامها **لأحدث قيم البيانات** وذلك بغض النظر عن مكان تخزين هذه البيانات في التسلسل الهرمي لوحدة التخزين .
- البيانات ذات المعالجات المتعددة لابد من أن تؤمن عملية الترابط فيما يخص ال cache أو ما يدعى **cache coherency** بحيث تصل أحدث قيمة موجودة على ال cache إلى المعالجات المركزية العاملة.
- في بيئات النظم الموزعة تعتبر مسألة الكاش أكثر تعقيدا من سابقتها وذلك لإمكانية وجود عدة نسخ من المعطيات التي يتم التعامل معها.

I/O Management إدارة أجهزة الدخل والخرج .

أحد مهام نظام التشغيل هي إخفاء خصوصيات الأجهزة عن المستخدمين.

يعد I/O Manager مسؤولاً عن :

1. إدارة الذاكرة فيما يخص ال I/O ومن العمليات التي يديرها:

Buffering وهي عملية تخزين البيانات بشكل مؤقت إلى أن يتم نقلها من أجل التنفيذ .

Caching وهي عملية تخزين أجزاء من البيانات الواردة في وحدات أسرع بغية تحسين الأداء .

Spooling وهي عملية التضارب الحاصل بين خرج مهمة ما ودخل مهمة أخرى .

2. واجهة عامة للـ device driver .

3. يعد أيضاً مسؤولاً عن ال drivers لمختلف الأجهزة المادية .

Protection and Security الأمن والحماية

Protection: تعرف بأنها أي آلية تتحكم بسماحية الوصول للموارد المتاحة من قبل ال processes أو ال users.

Security: تعرف بأنها عملية الدفاع من قبل النظام ضد أي هجمات داخلية أو خارجية.

من أوائل خطوات الأمان هي عملية تصنيف المستخدمين لتحديد من يمكنه فعل ماذا , تتم هذه العملية على المراحل التالية:

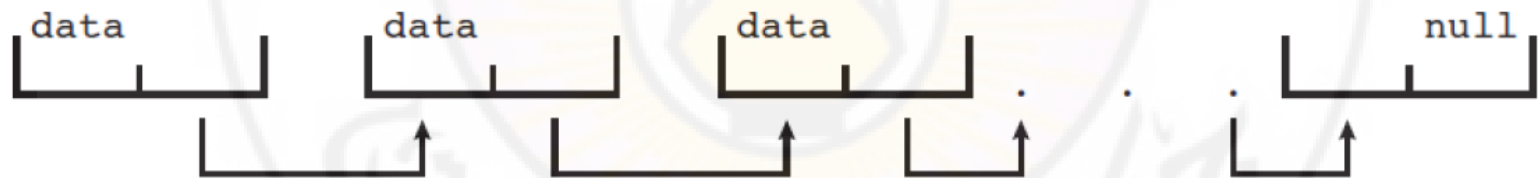
- 1 ■ تعريف هوية المستخدم " user IDs, security Ids " والتي تتضمن الاسم و associated number الخاص بالمستخدم .
- 2 ■ بعد ذلك يتم ربط id المستخدم بجميع الملفات والعمليات وذلك ليتم إعطاؤه صلاحيات الوصول المناسبة .
- 3 ■ تعريف المجموعات " group ID " والتي تسمح بتعريف مجموعة من المستخدمين وإدارة إمكانياتهم في التحكم , ومن ثم ربط هذه المجموعة بصلاحيات الوصول إلى الملفات وال processes المشتركة .
- 4 ■ وأخيرا القيام بزيادة الصلاحيات Privilege escalation والتي تسمح للمستخدم بتغيير ال id الفعالة الخاصة به في النظام ليحصل على صلاحيات أوسع .

هيكلية نواة نظام التشغيل Kernel Data Structures

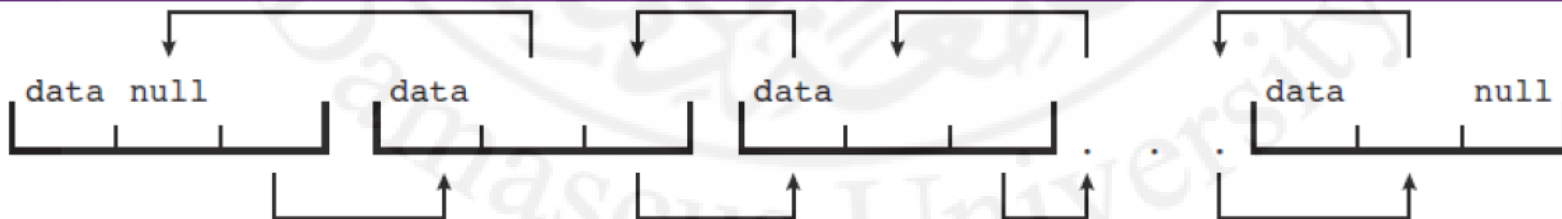
الـ kernel هي نواة نظام التشغيل و هي الجزء الذي يبقى عاملاً في الحاسب طالما لم يتم إغلاقه , وهي الدماغ المسؤول عن إدارة نظام التشغيل ويسمح للنظام وبعض التطبيقات بالتحكم بالمعالج والذاكرة , وتقوم بتنسيق وإدارة الدخل والخرج للنظام , وتنظم العلاقة بين النظام والأجهزة الأخرى الخارجية والداخلية منها .

• بنية المعطيات الخاصة بالـ Kernal شبيهة تماماً ببنى المعطيات البرمجية الأساسية , نذكرها سريعاً:

Single linked list:

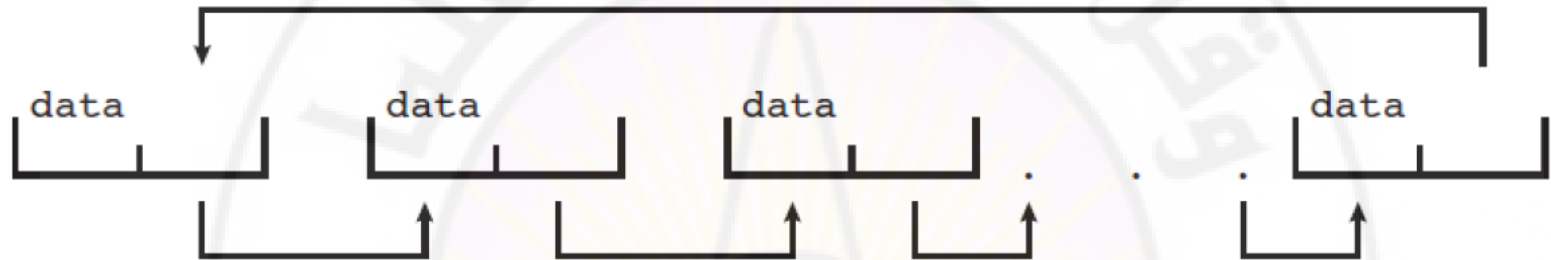


Doubly linked list:



هيكلية نواة نظام التشغيل Kernel Data Structures

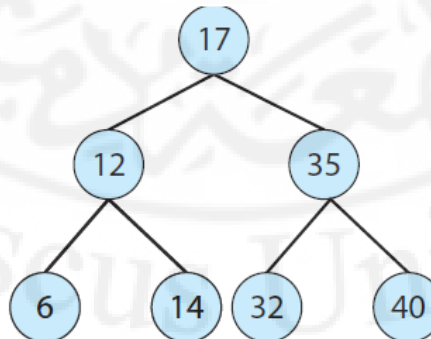
circular linked list:



Binary search tree:

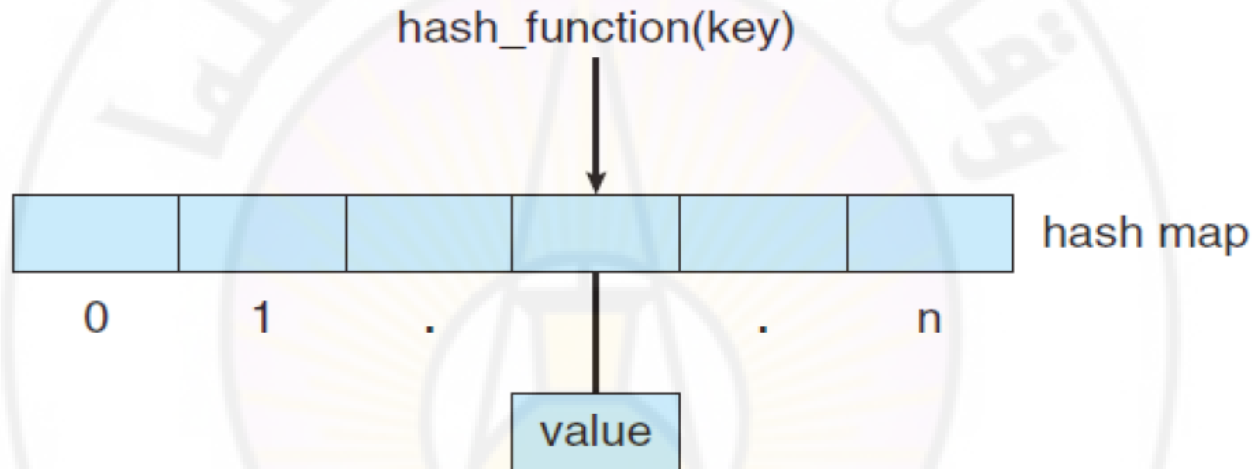
• حيث تكون الأفرع اليسارية $\leq$ الأفرع اليمينية .

• تعقيد البحث ضمن هذه البنية يساوي إلى $O(n)$ أما في الأشجار الثنائية المتوازنة يكون التعقيد مساو إلى $O(\lg n)$



هيكلية نواة نظام التشغيل Kernel Data Structures

Hash map :



• من أهم مواصفات ال **Hash function** بأنه يقوم بعملية التشفير باتجاه واحد, أي لا يمكن التراجع أو فك التشفير.

• عند ادخال كلمة سر معينة بغرض تسجيل دخول مستخدم ما, يتم تشفير كلمة السر هذه ومن ثم تقارن بالكلمة المشفرة والموجودة بال **hash map** إذا حصل التطابق يتم تسجيل الدخول بشكل سليم.

هيكلية نواة نظام التشغيل Kernel Data Structures

Bitmap :

- هي أيضا أحد بنى المعطيات المستخدمة والتي يمكن تعريفها بأنها سلسلة محرفيه مؤلفة من عدد n من ال binary digits والتي تمثل حالة n عنصر من الموارد .
- وبالتالي فإن الموقع i ضمن bitmap مرتبط بحالة المورد i .

• مثال: وليكن لدينا ال Bitmap التالية 001011101

- ❖ إذا بدأنا العد من الصفر ومن اليسار، وباعتبار أن 0 تعني أن المورد متاح وال 1 تعني أن المورد غير متاح .
- ❖ فنجد أن الموارد 2,4,5,6,8 جميعها موارد غير متاحة.
- ❖ أما الموارد 0,1,3,7 فهي موارد متاحة.

بيئات الحوسبة Computing Environments

يمكن تصنيف بيئات الحوسبة إلى الأقسام التالية :

1. التقليدية Traditional
2. المتنقلة Mobile
3. المخدم – زبون Client-Server
4. الند للند Peer-to-Peer
5. البيئة الافتراضية Virtualization
6. الحوسبة السحابية Cloud Computing
7. أنظمة الزمن الحقيقي Real –Time Embedded Systems
8. النظم الموزعة Distributed systems

التقليدية Traditional

- آلات قائمة بذاتها عامة الأهداف , نستطيع القول انها هي الأجهزة العادية التي نستخدمها كالحواسيب الشخصية .
- البوابات (Portals) هي التي توفر وصول الويب للأنظمة الداخلية .
- الأجهزة الشبكية (thin clients) (Network computers) :تعمل كمحطة الويب .
- thin clients :عبارة عن حاسوب عادي لكنه مصمم لاستخدام الشبكة بطريقة خاصة , حاسوب لا تهم ما هي مواصفاته لأن كل ال process التي سيقوم بها سيؤديها على ال server الذي يتمتع بمواصفات عالية .

المتنقلة Mobile

- Mobile computers حاسوب يتصل بالشبكة عن طريق ال wireless وأغلب الحواسيب الآن مجهزة لهذا الأمر .

ما يتميز به ال Mobile عن ال Traditional:

ميزات إضافية لنظام التشغيل مثل GPS و Gyroscope

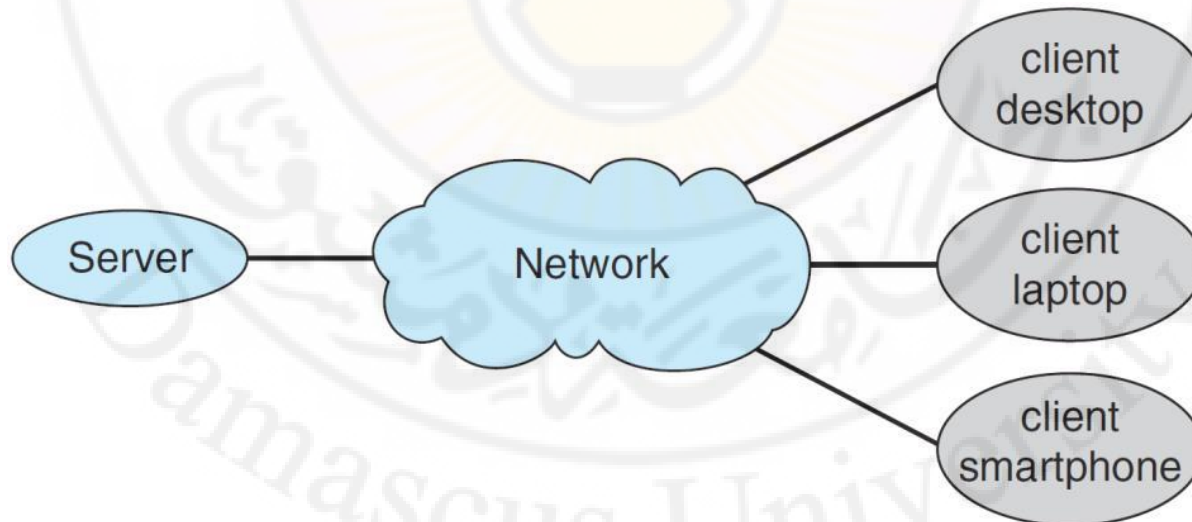
مناسب لأنواع جديدة من التطبيقات مثل augmented reality

يستخدم IEEE 802.11 اللاسلكية أو شبكات الهاتف المحمول للاتصال .

• رواد ال Computing Environments – Mobile هما Apple iOS و Google Android

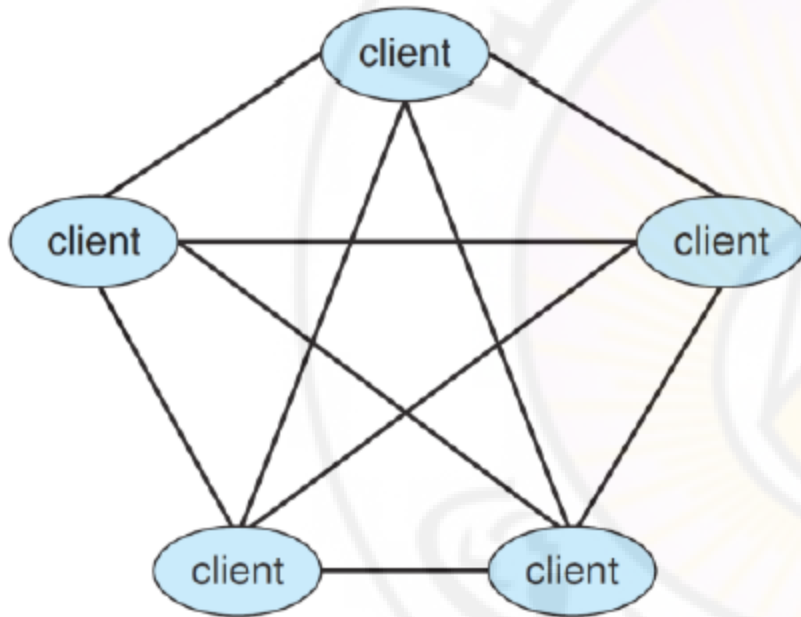
المخدم – زبون Client-Server

- ال server هو حاسوب بإمكانيات عالية مجهز ليخدم مجموعة خدمات محددة لمجموعة من ال clients عن طريق الشبكة.
- Dumb terminals تستبدل بالحواسيب الذكية.
- معظم الأنظمة الآن هي مخدمات تستجيب للطلبات المقدمة من قبل ال clients.
- Compute-server system يقدم واجهة بالخدمات المتاحة لل client.
- File-server system يقدم واجهة لل client لتخزين و استرجاع الملفات.



النند للنند Peer-to-Peer

• في هذا النموذج المستخدم يعمل بشكل مباشر مع المستخدم الآخر فلا يوجد server و client.



- يعتبر كل العقد (الأجهزة) أنداد .
- يجب أن يتم وصل كل عقدة مع كل عقدة أخرى .
- كل node تتصرف ك client أو ك server أو كليهما .
- كل عقدة تضع خدماتها في الشبكة بتقنية الـ central lookup.
- كل عقدة تبث طلبها لخدمة معينة , وتستجيب للطلبات عبر الخدمة discovery protocol.

• مثال عنها : Napster و Gnutella و Voice over IP (VoIP) كالكسايب.

البيئة الافتراضية Virtualization

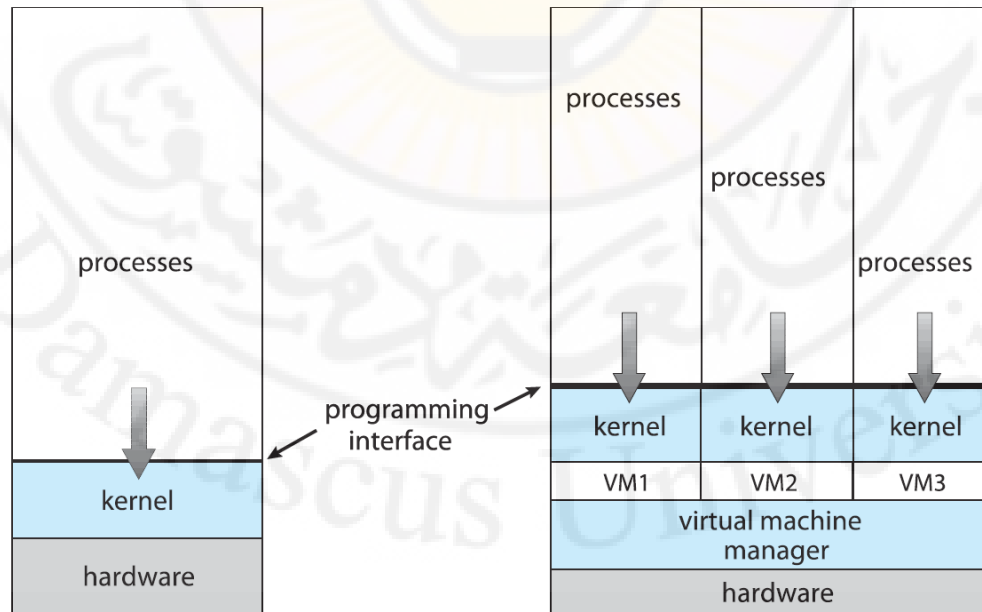
□ نظام التشغيل الأساسي للجهاز يستضيف نظام تشغيل جديد و يعمل كأنه أساسي أيضاً

□ مثلاً ال VMware يُشغل ال Linux كضيف , كل تطبيقات ال Linux يمكن تشغيلها دون أن يؤثر هذا على الحاسوب او نظام التشغيل الأساسي , وكل تطبيقات نظام التشغيل الأساسي يمكن تشغيلها

□ **VMM (virtual machine Manager)**: يوفر خدمات المحاكاة الافتراضية .

□ عندما أضع نظام تشغيل جديد فوق نظام التشغيل الحالي للجهاز فنظام التشغيل الأساسي للجهاز يرى النظام الجديد كتطبيق عادي , لكن المستخدم عند تطبيقه لأي أمر على نظام التشغيل الجديد يتوقع ان يتم تنفيذه تماماً كما لو أنه طبق الأمر على نظام التشغيل الأساسي .

□ إذاً نظام التشغيل الأساسي يجب أن يمتلك أليات ليتعامل مع النظام الجديد ليبدو و كأنه آلة مستقلة تماماً , هذا الأمر بحاجة لإدارة و هنا يأتي دور ال VMM



الحوسبة السحابية Cloud Computing

- خدمات الحوسبة و التخزين و حتى التطبيقات أصبحت متوفرة كخدمة عبر الشبكة .
- تعتبر امتداد لل Virtualization كونها تعتمد ال Virtualization أساس للمهام التي تقدمها.
- بيئات الحوسبة السحابية Cloud Computing تتألف من أنظمة التشغيل التقليدية، بالإضافة إلى VMM.

أدوات إدارة السحابة :

1. اتصال الإنترنت يتطلب الأمن مثل firewalls .
2. موازنات التحميل (Load balancers) تعمل على فصل حركة مرور البيانات عبر عدة تطبيقات.

- نستطيع القول أن Virtualization هي تقنية تقسيم الموارد و إعطائها ل OS مختلفة
- أما ال cloud فهي تحدد لمن أقدم المورد فهي تقوم بإدارة لل Virtualization.
- أمثلة عن الكلاود (IBM cloud – Google drive – Dropbox)

أنظمة الزمن الحقيقي Real –Time Embedded Systems

□ هي أنظمة موجودة في أجهزة تحكم مثل: أجهزة تجميع السيارات, منصات إطلاق الصواريخ, Robotics

□ أكثر أشكال الأنظمة انتشاراً

❖ متنوع الأهداف و الاستخدامات

❖ بعض الأجهزة لديها أنظمة التشغيل، و البعض يقوم بالمهام دون نظام التشغيل

□ لديها قيود زمنية محددة جيداً

❖ يجب أن تتم المعالجة ضمن القيد

❖ تكون العملية الصحيحة فقط إذا تحققت القيود

❖ مثلاً آلات سحب المال , لا يجب أن ينقص الرصيد قبل أن تخرج النقود من الآلة , يجب أن تخرج بطاقة المستخدم بعد خروج المال و ان يصدر صوت تنبيه من الآلة بنفس الوقت .

النظم الموزعة Distributed systems

- الحوسبة الموزعة : هي عملية توزيع data موجودة على جهاز لتكون على عدة أجهزة و كل جهاز يقوم باستخدامها بشكل مختلف لانجاز مهمة ما .
- تعرف بأنها عملية جمع لأنظمة منفصلة و قد تكون غير متجانسة و ربطها لتعمل معاً .
- الشبكة هي طريقة التواصل , و استخدام بروتوكول ال **TCP/IP** هو الأكثر شيوعاً :

Local Area Network (LAN)

Wide Area Network (WAN)

Metropolitan Area Network (MAN)

Personal Area Network (PAN)

- من مزايا نظام التشغيل الموزع :
 - يسمح مخطط التواصل Communication scheme بتبادل الرسائل بين الأنظمة .
 - يقدم انطباع أو إيهام بالعمل على نظام واحد .
 - توزيع الحمل بين هذه الحاسبات (load balancing).
- من العيوب: زمن الاتصال بين أجزاء النظام قد يؤثر على أداء النظام ككل.

نظم التشغيل الأكثر شيوعاً

- **ميكروسوفت ويندوز windows** : نسبة عالية من الحاسبات تستخدم ويندوز والتي تعمل على معالجات إنتل والمعالجات المتوافقة معها. توجد منها عدة نسخ، مثل ويندوز XP وويندوز فيستا تستخدم في الحواسيب الشخصية ، ومثل ويندوز advanced server و ويندوز 2000 التي تستخدم في الخوادم servers .
- **ماكنتوش Mac** : صمم ليعمل على أجهزة أبل ماکنتوش التي تنتشر في دور الطباعة والنشر، وهو قوي وسهل الاستخدام وقد أخذت ويندوز فكرة النوافذ وسطح المكتب من هذا النظام حيث كان أول نظم تشغيل يدعم الواجهات الرسومية و الايقونات والقوائم على سطح مكتب.
- **يونيكس UNIX** : صمم في عام 1974 بواسطة Dennis Ritchie و Ken Thompson بينما كانا يعملان في معامل AT & T Bell . كان الهدف نظام تشغيل صغير ومتنقل. ثم أنتشر في الجامعات مراكز البحوث في عام 1980
- ومن نسخ ينكس المشهورة: BSD - V Unix
- وهو أول نظام تشغيل يكتب بالكامل بلغة برمجة عالية high level language ، فهو مكتوب بلغة C
- **توزيعات لينكس Linux** : هي نسخة مصغرة من ينكس صممت لتعمل على الحاسبات الشخصية، وهو مفتوح المصدر حيث يتيح حرية تعديل الشفرة وإعادة التوزيع . ويوجد منها مئات التوزيعات أحدثها وأشهرها توزيعه أوبونتو التي تدعم كل لغات العالم واجهة وكتابة بما فيها اللغة العربية .

السنة الرابعة

نظم تشغيل 1



المحاضرة الثانية

OS Structures

د. المثني خضر

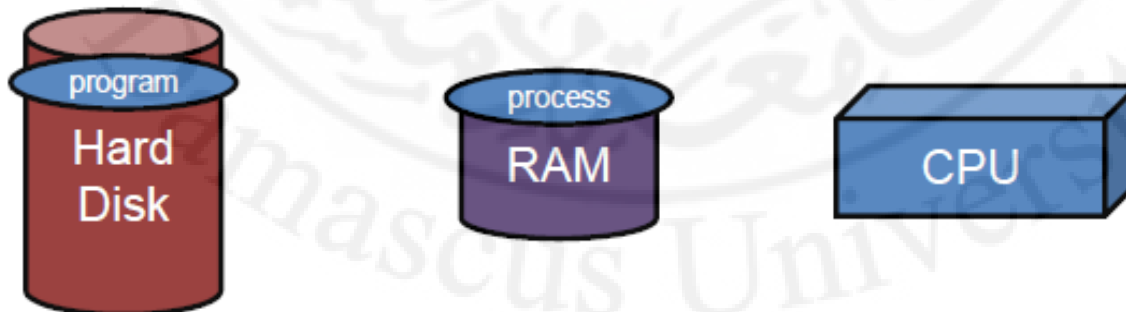
كلية الهندسة المعلوماتية

المحاضرة الثانية: المفردات الرئيسية

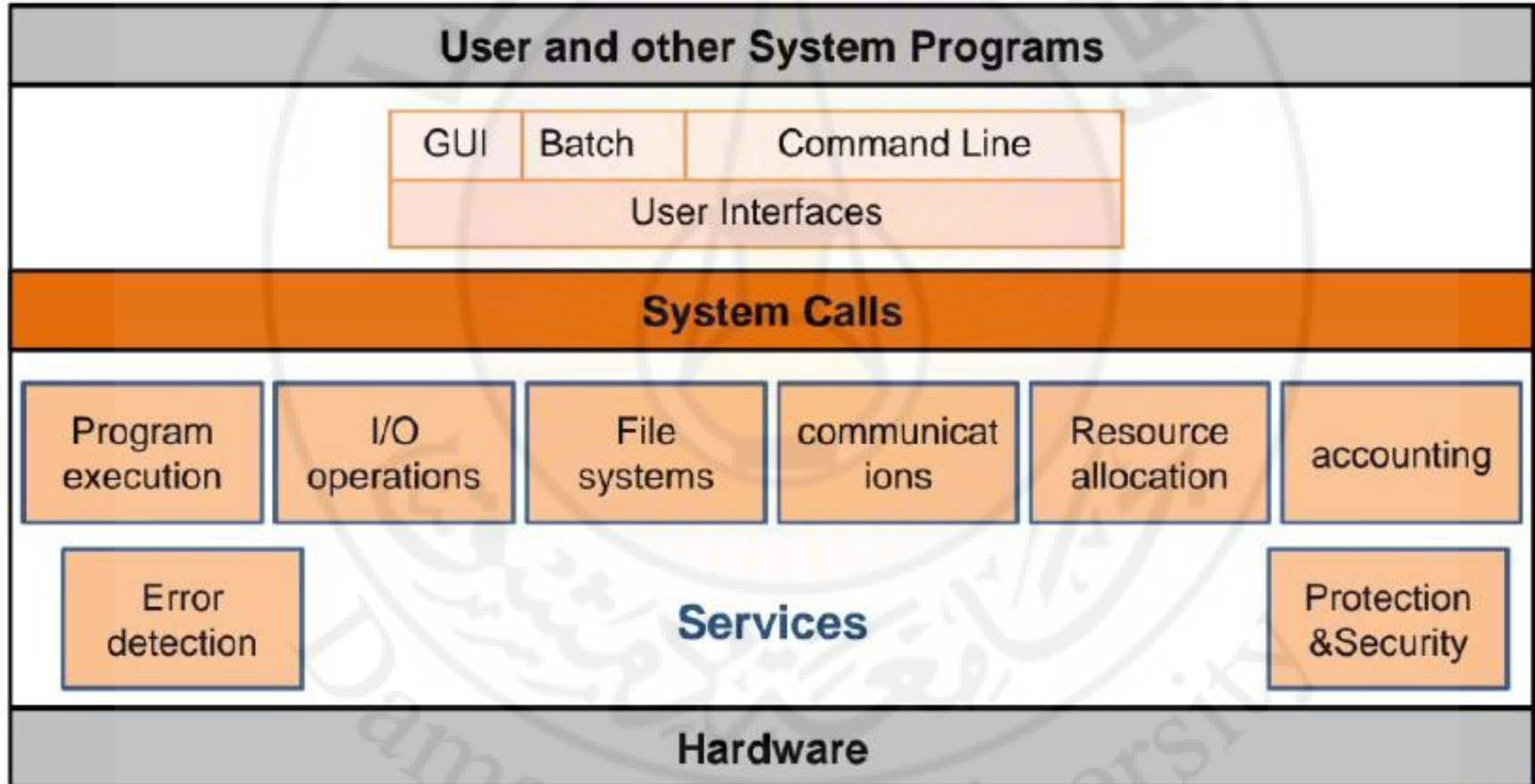
1. إقلاع نظام التشغيل (Booting)
2. خدمات نظام التشغيل
3. تنفيذ البرامج
4. واجهات المستخدمين User Interfaces
5. نداء النظام (System call)
6. نظام الـ Arduino & نظام الـ Free BSD
7. برامج النظام System programs
8. Linkers & Loaders
9. برامج التطبيقات Applications
10. تصميم نظام التشغيل Operating System Design
11. تحقيق نظام التشغيل Implementation of OS
12. هياكل نظام التشغيل OS Structure
13. Hybrid Systems & Modules
14. Debugging & Tracing

إقلاع نظام التشغيل (Booting)

- عند الضغط على زر الإقلاع، تتولى **BIOS** مهمة إقلاع النظام.
- توجد BIOS ضمن الذاكرة **ROM** وتعرف باسم firmware على اللوحة الأم.
- ينجز BIOS اختبار التأكد من عمل جميع العتاديات (RAM، أجهزة التخزين)؛ يسمى هذا الإجراء: **Power-On Self Test (POST)**.
- سوف تبحث BIOS عن قطاع صغير من hard disk يسمى MBR (boot loader).
- **Boot Loader**: هو برنامج محمل النظام، مهمته تحميل نواة النظام إلى الذاكرة **RAM**.
- يحوي MBR قائمة أو map بجميع أجزاء (partitions) القرص الصلب.
- بعد إيجاد MBR، ينفذ Boot Loader تعليمات أساسية لإقلاع نظام التشغيل.



- يزود نظام التشغيل بيئة لتنفيذ الخدمات والبرامج إلى المستخدمين وبرنامج أخرى.



- تزود كل مجموعة من خدمات نظام التشغيل وظائف مفيدة إلى المستخدم:
- **واجهة المستخدم:** تملك معظم أنظمة التشغيل واجهة مستخدم UI. تتنوع بين سطر أوامر Command-Line، وواجهة رسومية GUI، Batch.
- **تنفيذ البرنامج:** يجب أن يكون النظام قادراً على تحميل برنامج إلى الذاكرة وتشغيل هذا البرنامج، وتنفيذه إما بشكل طبيعي أو الإشارة إلى وجود خطأ.
- **عمليات الإدخال والإخراج:** يمكن أن يتطلب البرنامج المنفذ عملية إدخال أو إخراج، والتي يمكن أن تتضمن ملفاً أو جهاز I/O.
- **معالجة ملفات النظام:** تحتاج البرامج إلى قراءة وكتابة ملفات ومجلدات، إنشاء وحذف، البحث، استعراض معلومات عن الملفات، إدارة السماحيات عليها.
- **الاتصالات:** يمكن تبادل المعلومات بين العمليات؛ إما على جهاز واحد أو أكثر من جهاز عبر الشبكة. يتم ذلك إما باستخدام الذاكرة المشتركة أو تمرير الرسائل.
- **اكتشاف الخطأ:** يجب أن يتخذ نظام التشغيل حدثاً ملائماً لكل نموذج من الأخطاء ضمان استمرار العمل بشكل صحيح. (مشكلة التوقف التام).

- مجموعة أخرى من الوظائف تضمن أداءً فعالاً للنظام نفسه من خلال مشاركة الموارد:
- **تخصيص الموارد:** عندما تعمل مجموعة من المستخدمين أو مجموعة من المهام بشكل متزامن؛ فإنه يجب تخصيص الموارد بينها.
 - دورات CPU، الذاكرة الرئيسية، أجهزة الإدخال والإخراج، تخزين ملف.
- **المحاسبة (accounting):** يحدّد ما هو المسموح استخدامه من موارد لكل حساب مستخدم.
- **الأمن والحماية:**
 - Protection: التحكم بنفاذ جميع المستخدمين إلى الموارد.
 - Security: يتطلّب حماية النظام إجراء التوثيق (authentication)، وحماية أجهزة الإدخال والإخراج من محاولات النفاذ غير المسموح بها.

تنفيذ البرامج

من المعلوم أن البرنامج هو سلسلة من الأوامر (التعليمات) التي تنفذ متسلسلة (أمر تلو الآخر)، ويقوم المعالج بتنفيذ كل أمر في **خمس مراحل** هي:

➤ **إحضار الأمر (fetch)** : هنا نحضر الأمر من الذاكرة الرئيسية ونخزنه في سجل داخل المعالج.

➤ **فهم وتفسير الأمر (decode)** : هنا يقوم المعالج بالعمل على الأمر لفهمه ومعرفة ما إذا كان يحتاج بيانات إضافية من الذاكرة أم لا؟ مثلاً إذا كان الأمر (أجمع) فهذا يعني أننا نحتاج إحضار الرقمين المراد جمعهم من الذاكرة الرئيسية.

➤ **إحضار البيانات الإضافية (fetch operands)** : هنا نحضر البيانات التي يحتاجها الأمر من الذاكرة الرئيسية (إذا كان الأمر يحتاج بيانات إضافية حسب الخطوة 3 أعلاه) ونضعها في سجلات داخل المعالج.

➤ **التنفيذ (execute)** : ينفذ الأمر بواسطة وحدة الحساب والمنطق التي تعتبر جزءاً من المعالج.

➤ **التخزين** : هنا تخزن نتائج في سجلات داخل المعالج.

يكرر المعالج الخطوات أعلاه على كل أمر من أوامر البرنامج حتى ينتهي التنفيذ،

واجهات المستخدمين User Interfaces

يوجد ثلاثة أنواع من واجهات المستخدمين :

CLI موجه أوامر – GUI واجهة المستخدم الرسومية – touchscreen شاشة اللمس

Command Line Interpreter CLI

□ يسمح للمستخدم بإدخال الأوامر بشكل مباشر .

□ أحياناً يكون ال implementation الخاص بالأمر ضمن ال Kernel، وأحياناً يكون ضمن برنامج في النظام .

□ يوجد عدة أنواع من ال CLIs (shells)، مهمتها بشكل أساسي جلب تعليمات من المستخدم لتنفيذها .

□ في بعض الأحيان تكون ال commands هي Built-in وذلك يعني أن ال code الخاص بتنفيذ الأوامر مضمن ضمن ال CLI

□ وقد تكون ال command هي external وتكون بذلك هي عبارة عن أسماء برامج , في هذه الحالة لا يكون الكود الخاص بالأمر مضمن بال CLI وإنما تقوم ال shell بتحميل الكود الخاص بالأمر من مكانه المحدد إلى الذاكرة الرئيسية , و بالتالي يكون إضافة برامج جديدة لا يحتاج إلى إحداث تغييرات في الكود المضمن في ال shell.

واجهات المستخدمين User Interfaces

GUI واجهة المستخدم الرسومية

□ واجهة المستخدم , مجازاً هي سطح المكتب Desktop

• عادةً يوجد فأرة , لوحة المفاتيح , شاشة

• فيها أيقونات تدل على الملفات و البرامج و العمليات إلخ ...

• الضغوطات على أزرار الفأرة المختلفة على الأيقونات تسبب إجراءات مختلفة كتقديم معلومات حول الأيقونة المختارة , أو فتح المجلد أو تنفيذ برنامج ...

□ العديد من الأنظمة الآن تشمل الواجهات CLI و GUI

• Microsoft Windows فيها GUI و CLI والتي هي cmd.

• Apple Mac OS X فيها GUI "aqua" مع UNIX kernel و Shell.

• Unix و Linux فيهما CLI أما ال GUI فهي اختيارية (CDE, KDE, GNOME)

touchscreen شاشة اللمس

■ • الفأرة غير موجودة أو غير مرغوبة .

■ • الإجراءات و الاختيارات تكون بإشارات (اللمس) .

■ • Virtual keyboard لإدخال النصوص .

■ • يدعم التعليمات الصوتية .

نداء النظام (System call)

- وهي واجهة برمجية تطلب الخدمات التي يقدمها نظام التشغيل , حيث أن كل الخدمات السابقة لنظام التشغيل لا يمكن الوصول لها مباشرة وإنما من خلال ال System call.
- تكتب عادة بلغات عالية المستوى . C , ++ C
- عادةً ما تستخدم التطبيقات المكتوبة بلغات عالية المستوى واجهات برمجية API تستدعي من خلالها ال System calls حيث أن التطبيقات لا تقوم بإستدعاءها مباشرة .
- واجهات التطبيقات APIs الأكثر شيوعاً هي :

- .Win32 API for Windows
- POSIX API for POSIX-based systems يستخدم في UNIX, Linux, Mac OS X.
- Java API for the Java virtual machine (JVM)

مثال:

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count)
```

return value function name parameters

حيث أن ال read تعتبر API تطلب system call تطلب service قراءة ملف/دخول من نظام التشغيل .

System Call Implementation

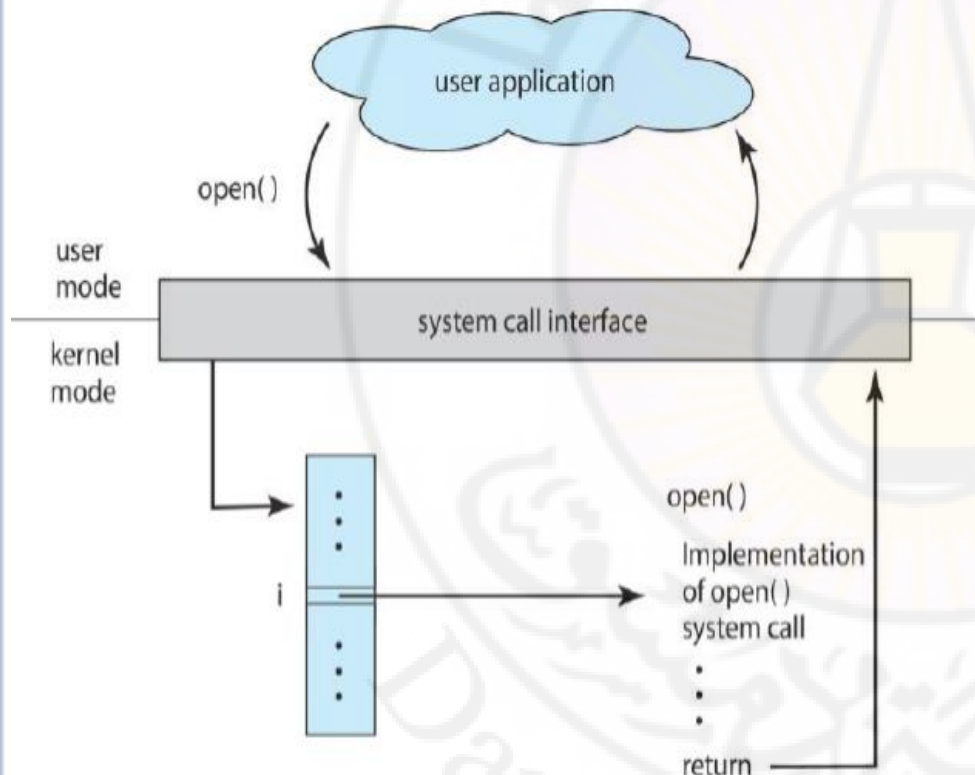
عادةً ترتبط كل System call برقم , و يتم الاحتفاظ بال System-call interface بجدول مفهرس بهذه الأرقام .

تقوم الـ system call interface باستدعاء الـ system call من نواة نظام التشغيل kernel وإرجاع حالة الـ system call أي قيمة إرجاع return values.

المستدعي لا يهتم معرفة كيفية تنفيذ الـ System calls يهتم فقط أن يرى تفاعل نظام التشغيل مع الأمر الذي تم طلبه, لذلك يحتاج فقط لاتباع الـ API ومعرفة ما يمكن لنظام التشغيل فعله كنتيجة للاستدعاء.

معظم تفاصيل واجهة النظام يتم إخفاؤها من خلال واجهات التطبيقات API حيث تدار هذه العملية من خلال الـ run-time support library التي هي عبارة عن مجموعة من الـ functions الموجودة في المكتاب ومضمنة في الـ compiler .

System Call Implementation



نلاحظ بدايةً أن المستخدم يكون في ال user mode وعند استدعائه لل API: Open(), يتم التحويل إلى ال kernel mode ويتم استدعاء ال system call والتي يكون رقمها i وتنفذ ال action الخاص بال open(), ثم تقوم بإعادة قيمة الاستدعاء إلى المستخدم.

System Call Parameter Passing

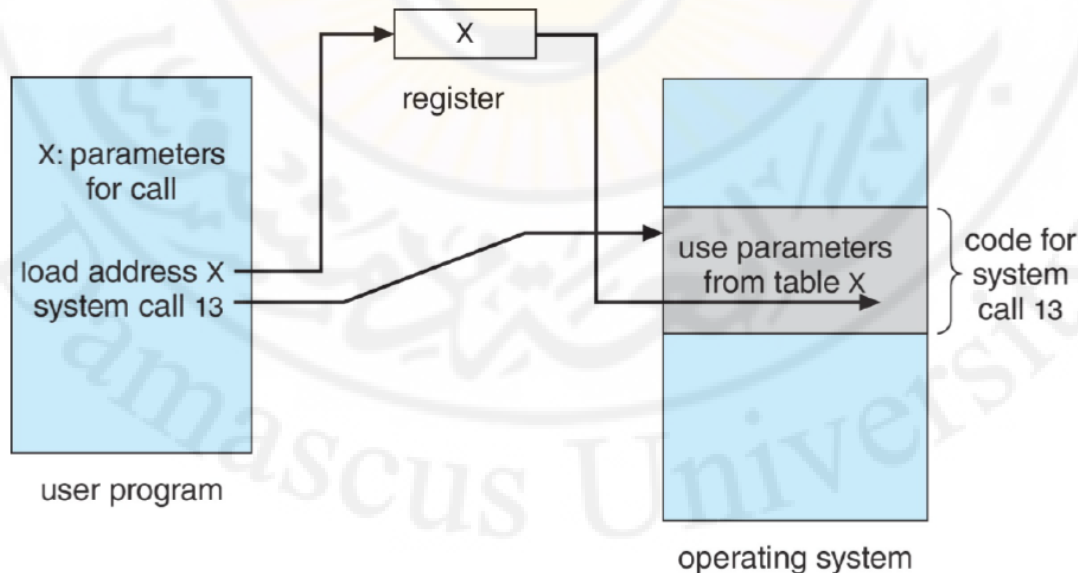
• هناك ثلاث طرق عامة تستخدم لتمرير المعلومات إلى نظام التشغيل :

1. أبسط طريقة هي تمرير المعلومات من خلال السجلات parameters in registers لكن عدد ال registers الممكن استخدامها محدود.

2. تخزين المعلومات parameters في كتلة Block أو جدول Table في الذاكرة , ثم يتم تمرير عنوان هذا block / table ك parameter in a register و هذه هي الطريقة المتبعة في Linux و Solaris .

3. إدخال ال parameters في stack من قبل البرنامج , ثم تخرج من ال stack من قبل نظام التشغيل .

• نلاحظ أن تمرير المعلومات في stack أو Block تسمح بتمرير عدد غير محدد من ال parameters



Examples of Windows and Unix System Calls

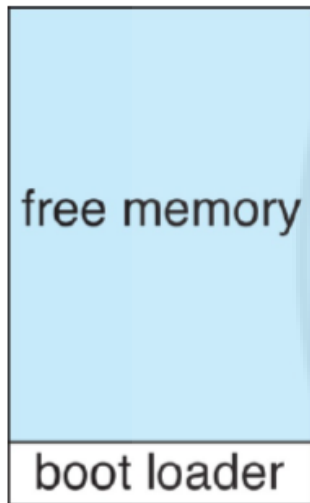
EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communications	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

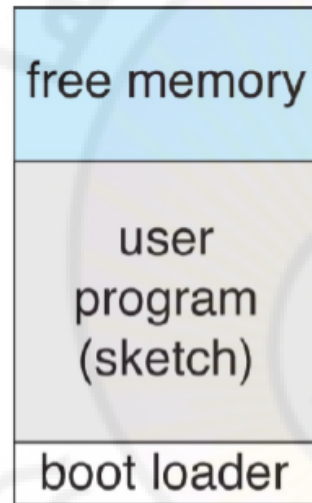
نظام الـ Arduino

□ يستخدم في مجالات عدة منها (robotics -home automation)



(a)

At system startup



(b)

Running a program

□ يحتوي على Microcontroller للتحكم بالجهاز.

□ (Single-tasking) أي لا يقوم بـ multi-tasking.

□ لا يحتوي على نظام تشغيل و يعالج فقط Binary Data.

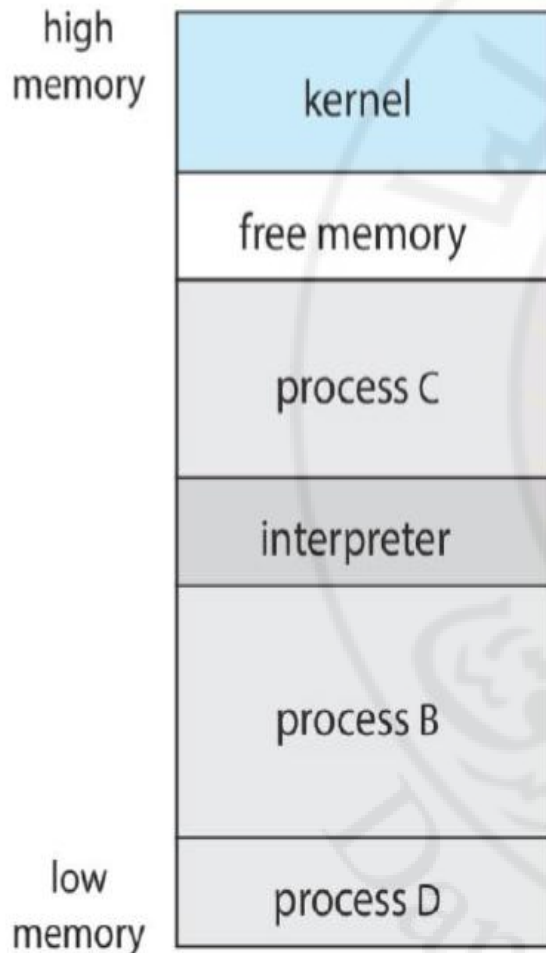
□ يتم تحميل البرامج (sketch) عن طريق مدخل USB الى الذاكرة.

□ مساحة تخزين وحيدة .

□ نظام إقلاع (boot loader) يحمل البرنامج عند تشغيل الجهاز

□ يعاد تحميل الـ shell عند انتهاء تنفيذ البرنامج و الخروج منه .

نظام الـ Free BSD

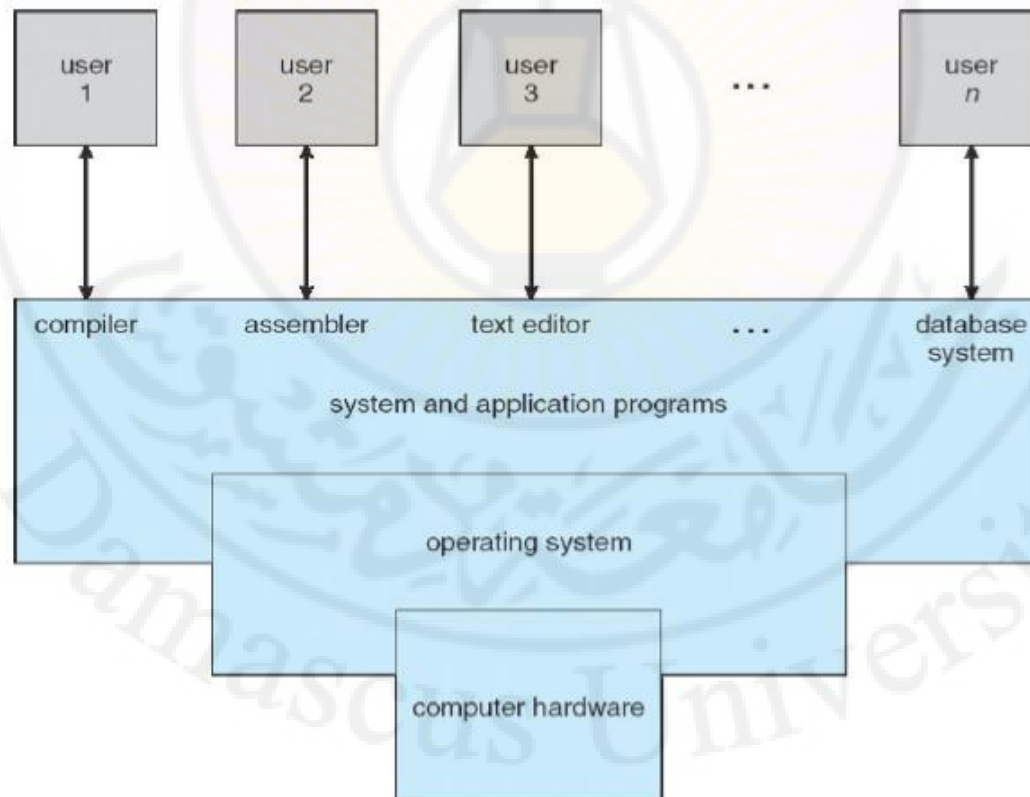


- هو نظام تشغيل شبيه بـ Unix مفتوح المصدر.
- يدعم الـ multi-tasking وهو نوع من concurrent processing.
- (User login) يستدعي خيارات الـ المستخدم من الـ shell.
- تقوم الـ shell باستدعاء system call و هو fork() لإنشاء process، و تستدعي الـ system call يدعى exec() لتحميل الـ program الى الـ process.
- تنتظر الـ shell العملية (process) حتى تنتهي او بإمكانها متابعة تلقي تعليمات المستخدم user-commands.
- تنتهي العملية و لها حالتان:
 - Code = 0 لا يوجد خطأ.
 - Code > 0 يوجد خطأ.

برامج النظام System programs

إن نظام التشغيل يؤمن بيئة **لتنفيذ** البرامج عبر System calls , أما ال system program (التي تكون من نظام تشغيل وكومبايلر و assembler و text editor) تؤمن بيئة **لتنفيذ** البرامج **وتطويرها**

• يتعامل المستخدمون مع نظام التشغيل من قبل برامج النظام، وليس من قبل استدعاءات النظام الفعلية , حيث قد تكون واجهة بسيطة إلى استدعاءات النظام, أو قد تكون أكثر تعقيداً .



برامج النظام System programs

• يمكن تقسيم ال system programs إلى :

File management -1

إدارة الملفات بشكل عام كإنشاء، حذف، نسخ، إعادة تسمية، طباعة، تفريغ، الوضع بالقوائم، بشكل عام إدارة الملفات والمجلدات .

Status information -2

توفر الحصول على معلومات : التاريخ والوقت وكمية الذاكرة المتوفرة، مساحة القرص، وعدد المستخدمين , توفر برامج أخرى معلومات مفصلة عن الأداء , معلومات عن التسجيل logging، وتصحيح أخطاء debugging, عادة هذه البرامج تنسق وتطبع الخرج إلى ال terminal أو غيرها من أجهزة الإخراج , بعض الأنظمة فيها registry يستخدم لتخزين و استعادة معلومات التكوين configuration information.

File modification -3

عن طريق محررات النصوص لإنشاء الملفات وتعديلها , و يوجد أوامر خاصة للبحث في محتويات الملفات أو إجراء تغييرات في النص .

Programming-language support -4

Compilers, assemblers, debuggers , interpreters

5- تحميل و تنفيذ البرامج .

وذلك من خلال المحملات (absolute loader/relocatable loader/ linkage editors /overlay loaders) و عملية ال debugging على مستوى النظام .

برامج النظام System programs

Communications -6

توفير إمكانية إنشاء تواصل افتراضي بين ال process, المستخدمين , نظام الحاسوب.
كإسماح للمستخدمين بإرسال الرسائل إلى شاشات بعضهم البعض، وتصفح صفحات الويب، وإرسال رسائل البريد الإلكتروني، وتسجيل الدخول عن بعد، ونقل الملفات من جهاز إلى آخر.

Background Services -7

- تنفيذ عند الإقلاع boot time بعضها ينفذ من أجل بدء تشغيل النظام، ثم تنتهي، و بعضها الآخر يعمل من إقلاع النظام system boot إلى إيقاف التشغيل .
- توفير facilities مثل فحص القرص، جدولة العمليات، تسجيل الأخطاء، والطباعة .
- يتم تشغيلها في سياق المستخدم user context وليس ضمن سياق النواة kernel context.
- تعرف باسم الخدمات services والنظم الفرعية subsystems أو daemons.

Applications -8 برامج التطبيقات

- لا تتعلق بالنظام , يتم تشغيلها من قبل المستخدمين , لا تعتبر عادة جزءاً من نظام التشغيل .
- تبدأ عملها بأمر من سطر الأوامر، أو بالنقر فوق الماوس ، أو بالإصبع .

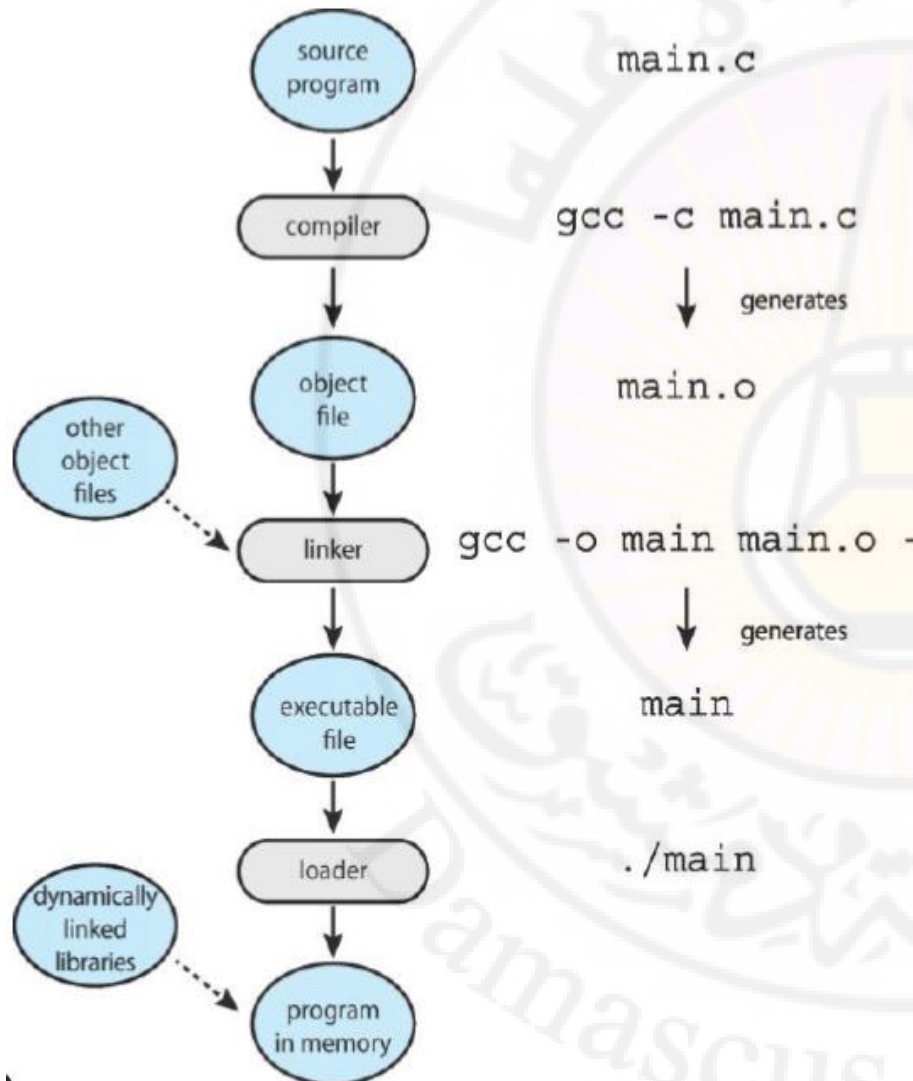
Linkers & Loaders

الـ compiler هو : software يحول الـ source code الى ملف تنفيذي
يتواجد الـ Linker في الـ compiler أما الـ loader يوجد بنظام التشغيل

- يقوم الـ **compiler** بداية بتحويل الملف المصدري source code إلى Object files .
- يقوم الـ **Linker** بعد ذلك بتجميع هذه الـ Object files و جعلها ملف تنفيذي واحد binary executable file و يحضر المكتبات الخاصة التي يحتاجها مثل math library .
- يتم تخزين البرنامج على ذاكرة ثانوية كملف تنفيذي binary executable file .
- ثم يتم تحميله الى الذاكرة الرئيسية عن طريق الـ **Loader** ليتم تنفيذه.
- الملفات التنفيذية لها صيغة معينة معروفة و لذلك يعرف نظام التشغيل كيفية تحميلها و تنفيذها .
- تقوم عملية الـ **Relocation** بإعطاء العناوين النهائية لأجزاء البرنامج و تعديل الـ data و الـ code لتطابق هذه العناوين.

في الأنظمة الحديثة لا يتم اعتبار المكتبات التي يتم استخدامها في أغلب البرامج مثل stdio.h كبقية المكتبات الخاصة, أي لا يقوم الـ Linker بربطها مع الملفات التنفيذية بل يتم اعتبارها كـ dynamically linked libraries (DLL in windows) و تكون مشتركة من قبل جميع البرامج التي تستخدم نفس النسخة من هذه المكتبات ويتم تحميلها مرة واحدة فقط من قبل الـ loader

Linkers & Loaders



■ هنا ال main.c هي برنامج وليس function ال main , بداية سيتم إجراء compile للبرنامج وتحويله ل object file , ثم يقوم ال linker بربط عدة object files بالإضافة الى ال libraries إلى ال object file الناتج عن ال compiling لينتج ملف تنفيذي واحد , ثم يقوم ال loader بتحميله مع ال DLL الى الذاكرة.

برامج التطبيقات Applications

□ عادةً التطبيقات التي تم إجراء عملية ال compilation لها على نظام تشغيل معين لا تعمل على نظام آخر , ذلك لأن كل نظام تشغيل يقدم system call خاصة به تختلف عن غيرها من الأنظمة ولكل نظام له file formats تختلف عن الأنظمة الأخرى .

□ يمكن للتطبيقات أن تعمل على أكثر من نظام تشغيل في الحالات التالية:

1. إذا كانت مكتوبة ب (interpreted language) مثل python أو Ruby وكان ال interpreter يعمل على أكثر من نظام تشغيل .
2. إذا كانت مكتوبة بلغة تتضمن VM (virtual machine) التي تقوم بتنفيذ البرنامج , مثل اللغة Java.
3. إذا كانت مكتوبة بلغة مثل C ويتم إجراء عملية ال compilation على كل نظام تشغيل مختلف على حدا , وذلك يتطلب أن يكون ال compiler موجود على النظامين و بنفس النسخة (بهذه الطريقة من الضروري تداول ال source code مما قد يؤدي إلى سرقة) .

□ Application Binary Interface (ABI) : هو بنية مثل ال API يحدد كيف تتعامل أجزاء الكود التنفيذية مع نظام التشغيل على بنية الجهاز (معالج, رام...الخ) .

تصميم نظام التشغيل Operating System Design

- يتأثر التصميم بخيارات الأجهزة الصلبة hardware، و نوع النظام المطلوب .
- يتم بدء التصميم من خلال تحديد الأهداف والمواصفات .

أهداف المستخدم وأهداف النظام:

أهداف المستخدم : يجب أن يكون نظام التشغيل مريح للاستخدام وسهل التعلم وموثوق به وآمن وسريع.

أهداف النظام : يجب أن يكون نظام التشغيل سهل التصميم والتنفيذ والصيانة، فضلاً عن المرونة والموثوقية والخلو من الأخطاء، و الكفاءة.

مبدأ هام للتفريق :

Policy: ما الذي سيتم عمله؟ ؟ أي أنها تقرير ما الذي سيتم تنفيذه .

Mechanism: كيف نفعل ذلك؟ أي أنها تحدد آلية القيام بشيء ما .

• إن فصل السياسة عن الآلية هو مبدأ مهم جداً ، فهو يسمح بأقصى قدر من المرونة في حال تم تغيير قرارات الـ policies في وقت لاحق .

• إن تحديد و تصميم نظام التشغيل هو مهمة خلاقة للغاية من مهام هندسة البرمجيات.

تحقيق نظام التشغيل OS Implementation of OS

□ اختلفت كثيرا بين كل عقد و آخر :

- أنظمة التشغيل القديمة كانت بلغة assembly.
- ثم لغات برمجة النظام أصبحت Algol , PL/1.
- الآن C ، C ++.

□ في الواقع عادة يكون OS مزيج من اللغات :

- أدنى مستويات بلغة التجميع assembly.
- الجسم الرئيسي Main body مكتوب ب C.
- برامج الأنظمة مكتوبة ب C ، C++ ، و ال scripting languages مثل PERL, Python, shell scripts.

□ اللغات التي يكون مستواها أعلى عالية يكون من السهل نقلها إلى الأجهزة الأخرى ولكنها أبطأ .

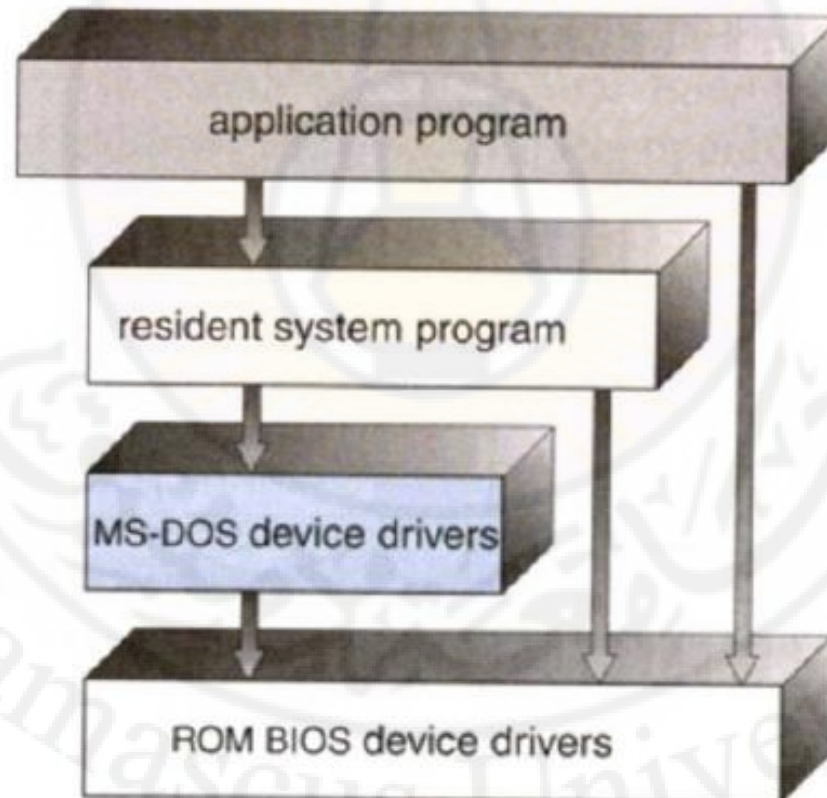
□ برامج المحاكاة Emulation يمكن أن تسمح لنظام التشغيل بان تعمل على أجهزة بأنظمة أخرى مثل VMware

هياكل نظام التشغيل Operating System Structure

- البنية البسيطة MS-DOS.
- في UNIX تمت تجزئة البنية إلى قسمين: النواة وبرامج النظام.
- النموذج الطبقي **Layered**: يتكون النظام من عدة طبقات.
- بنية النظام **Microkernel**: نقل جزء من مهام النواة إلى فضاء المستخدم، والإبقاء على المهام الأساسية (جدولة CPU، إدارة الذاكرة، الاتصالات).
 - Mach OS مثال عنها.
 - نظام التشغيل Mac OS يعتمد على Mach OS.

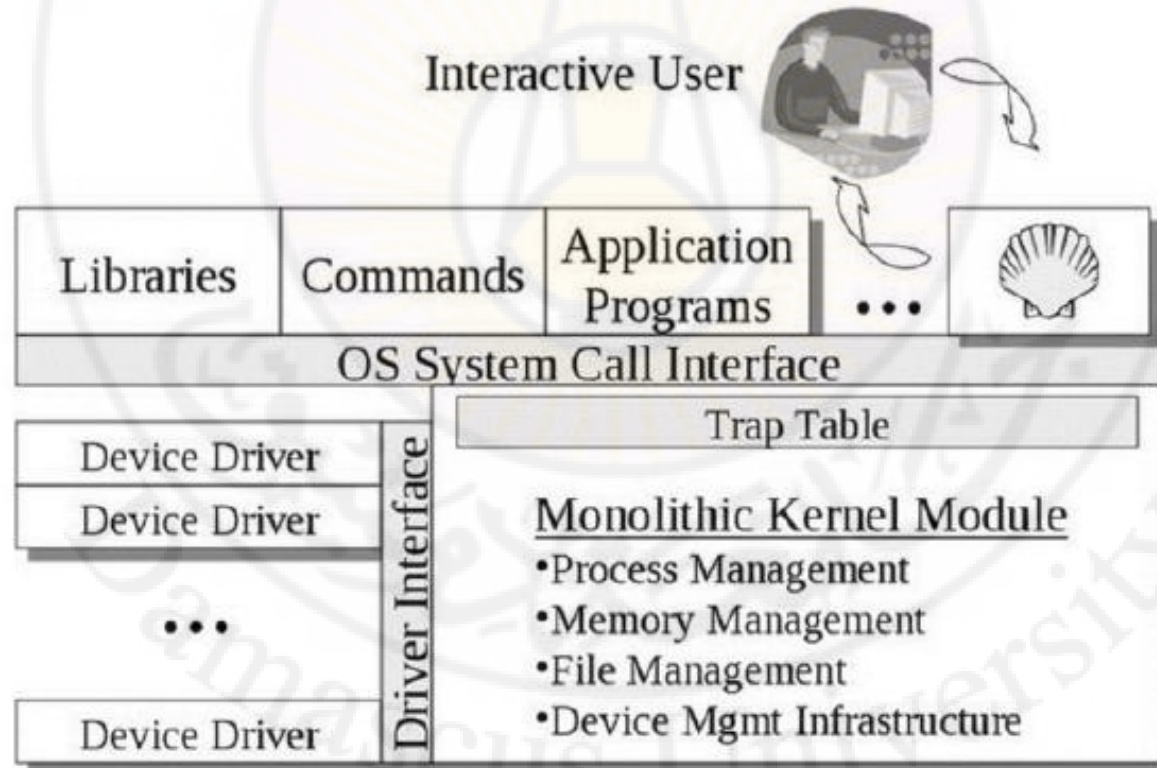
١- البنية البسيطة، مثال MS-DOS

- ٠ لا يملك نظام التشغيل بنية معرفة جيداً.
- ٠ لا يوجد نمط تنفيذ CPU (نمط المستخدم ونمط النواة)، وبالتالي يمكن أن تسبب الأخطاء في التطبيقات تعطل النظام بأكمله.



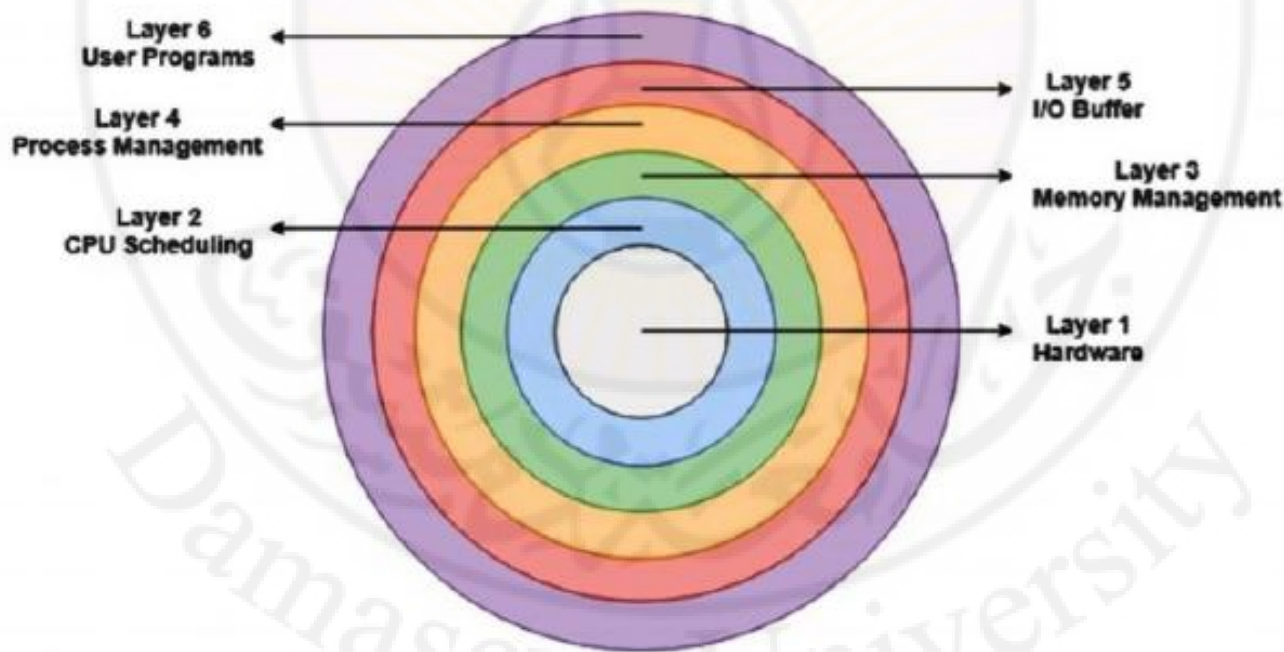
٢- MONOLITHIC APPROACH

- يتم تجزئة النظام في **UNIX** إلى جزأين منفصلين:
✓ برامج النظام – النواة.
- تتألف النواة من كل شيء تحت واجهة استدعاء النظام وفوق العتاديات.
- تزود النواة : جدولة **CPU**، نظام ملفات، إدارة الذاكرة، ووظائف نظام التشغيل الأخرى.



٣- السلوك الطبقي (LAYERED)

- يتم تجزئة النظام إلى عدة طبقات. وتملك كل طبقة مهمة محدّدة يتوجب عليها تنفيذها. (Windows NT مثال عنها)
- تمثّل الطبقة الأدنى طبقة العتاديات.
- تمثّل الطبقة الأعلى التالية «طبقة النواة»، وتحتوي المهام الأساسية للنظام.
- تمثّل Layer N واجهة المستخدم.



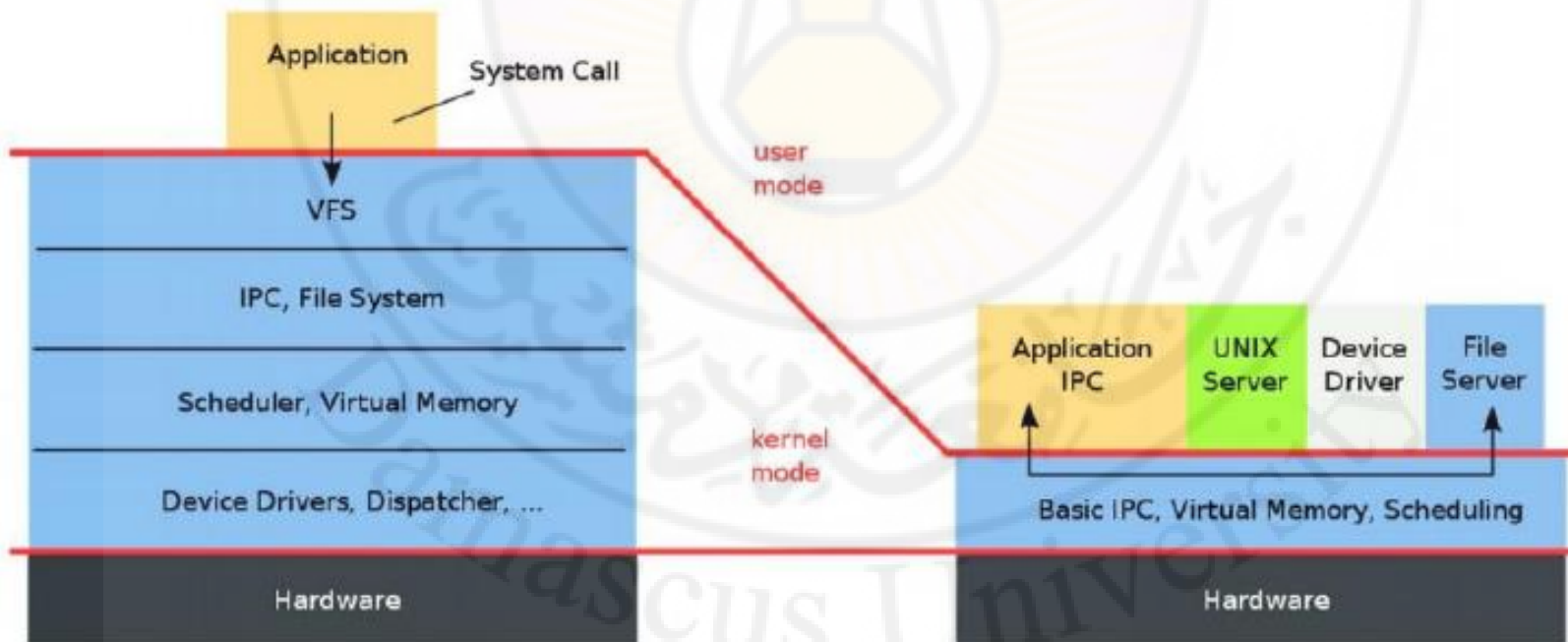
Layers of operating system

MICROKERNEL - ٤

- يعمل على هيكلة نظام التشغيل عن طريق إزالة جميع الأجزاء غير الأساسية من النواة وتنفيذها كبرامج على مستوى النظام والمستخدم (نمط المستخدم).
- Mach هو مثال عن النواة المصغرة، ويعتمد نظام **Mac OS** عليها.

Monolithic Kernel
based Operating System

Microkernel
based Operating System



Hybrid Systems & Modules

- يوجد العديد من أنظمة التشغيل الحديثة التي تستخدم مبدأ وحدات النواة القابلة للتعديل loadable kernel modules (LKMs).
- حيث تستخدم مبدأ ال Object-oriented و يكون كل عنصر أساسي منفصل عن الآخر, و يتم التواصل بين ال modules عن طريق interface معروفة, وحسب الحاجة يتم تحميل أيها إليها إلى داخل ال kernel.
- معظم أنظمة التشغيل الحديثة ليست نموذجاً وحيداً من النماذج السابقة و إنما هي هجين يجمع بين الأنماط المتعددة Hybrid Systems, من الأمثلة عليها :

1. Linux - Solaris : تعتبر من النمط "monolithic" بالاضافة الى كونها من النمط "module" من أجل التحميل الديناميكي للوظائف.
2. Windows : يكون في معظمه من النمط "monolithic" بالاضافة لكونه "microkernel" من أجل different subsystem personalities
3. Apple Mac OS : هو نظام هجين مما يلي :
(layered - Aqua UI - Cocoa programming environment) و من داخله نواة تتألف من : ال Mach microkernel , أجزاء من BSD Unix , أدوات إدخال و إخراج , وحدات قابلة للتحميل ديناميكياً (Kernel extensions).

Debugging & Tracing

- **Debugging:** هو إيجاد الأخطاء bugs وحلها، و ضبط أداء البرنامج performance tuning.
 - يقوم نظام التشغيل بحفظ معلومات الأخطاء في ملفات تدعى Log files يقوم هو بتوليدها .
 - حدوث فشل في تطبيق معين يولد core dump file يحتوي على مكان العملية في الذاكرة .
 - حدوث فشل في نظام التشغيل يمكن أن يولد crash dump file يحتوي على مكان نواة النظام kernel في الذاكرة .
 - ضبط أداء البرامج يمكن أن يحسن أداء النظام، عادة نستخدم trace listings of activities مثل task manager في نظام windows للقيام بذلك .
 - من خلال ال profiling نحصل على معلومات إحصائية عن الأداء .
- **Tracing:** يجمع معلومات حول حدث او عملية معينة، مثل خطوات المتضمنة في استدعاء System-call معين . تتضمن الأدوات المهام التالية:

1. Strace: والتي تتابع ال system call التي تم إستدعاؤها من قبل process .
2. GDB: source-level debugger .
3. Perf: وهي مجموعة من الأدوات الخاصة بأداء النظام لينوكس .
4. Tcpdump: والذي يقوم بجمع ال packets من الشبكة .

السنة الرابعة

نظم تشغيل 1



المحاضرة الثالثة

Process management

د. المثني خضر

كلية الهندسة المعلوماتية

المحاضرة الثانية: المفردات الرئيسية

1. مفهوم الإجرائية Process Concept
2. الإجرائية في الذاكرة Process in memory
3. حالات الإجرائية Process state
4. جدول التحكم بالإجرائية Process Control Block (PCB)
5. أنواع الأرتال
6. المجدول Scheduler
7. تبديل السياق Context Switch
8. إنشاء إجرائية Process Creation
9. إنشاء إجرائية بلغة الـ C
10. إنهاء الإجرائية Process Termination
11. الاتصال بين الإجرائيات Inter-process Communication
12. الاتصال بين الإجرائيات في الـ Windows

مفهوم الإجرائية Process Concept

- الـ **process** هي البرنامج بعد تحميله إلى الـ RAM و أثناء التنفيذ .
- يتم تنفيذ العديد من البرامج ضمن أنظمة التشغيل كـ process يتم تخصيص قسم من الذاكرة لكل process حيث تدعى هذه المساحة بـ process address space.

الفرق بين البرنامج والعملية

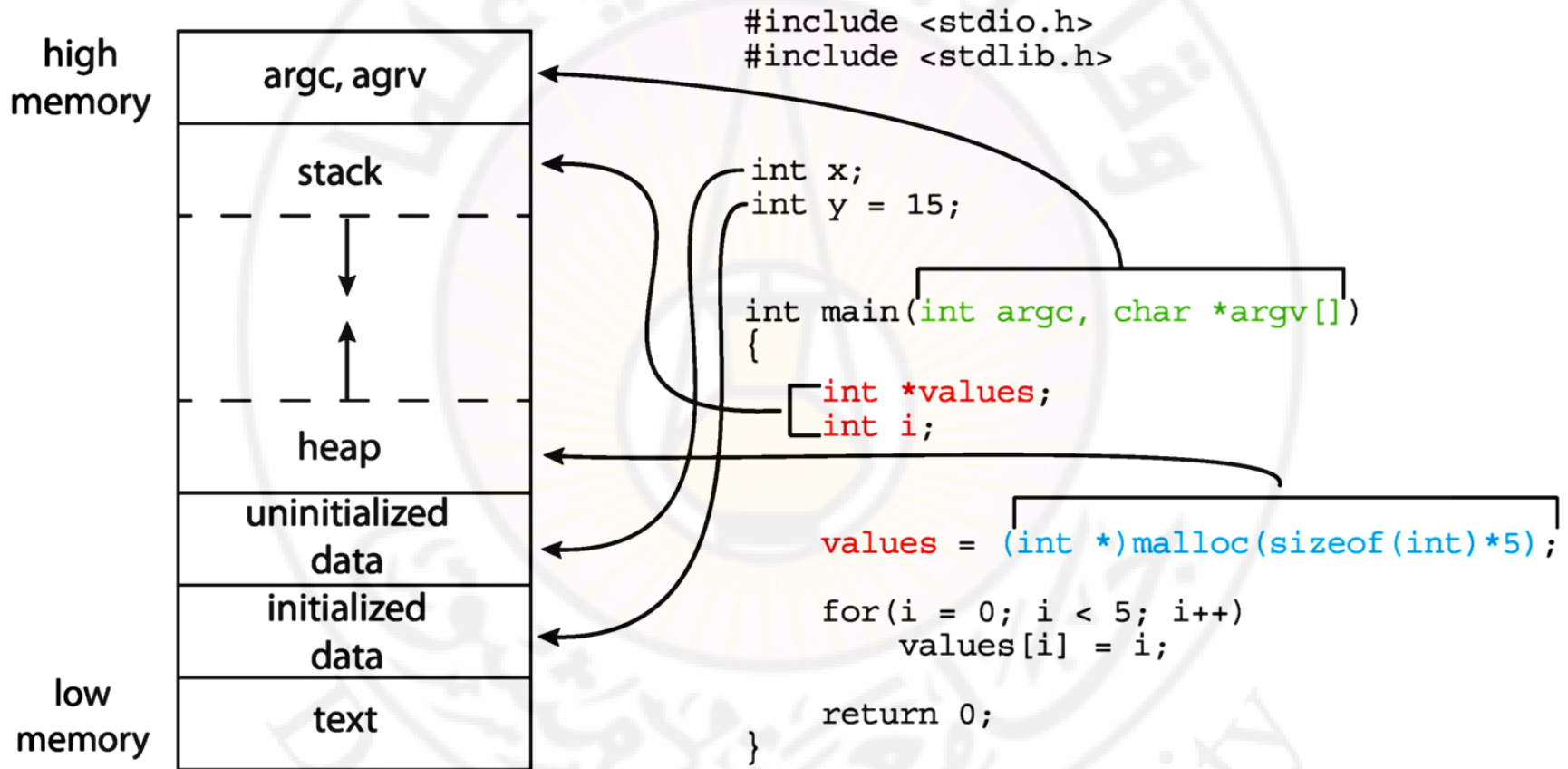
Program : هو passive entity مخزن على hard disk وهو ملف تنفيذي executable file.

Process : هو البرنامج عندما يتم تحميل الملف التنفيذي executable file الخاص به إلى الـ RAM فيصبح active entity.

- تتضمن الـ **Process Address space** الأقسام التالية:
- Text section: يضم الكود البرمجي الخاص بالبرنامج .
- current activity: النشاط الحالي يتضمن (program counter) و محتويات السجلات في المعالج.
- Stack: يحوي الـ data المؤقتة مثل براميتر التابع , return value, local variable
- Data section: يضم , static local variables, global variable
- Heap: تحوي المتحولات التي يتم حجزها بشكل ديناميكي أثناء الـ run time (مثل تعليمة malloc التي تقوم بحجز حجم ديناميكي لمتحول ما) .

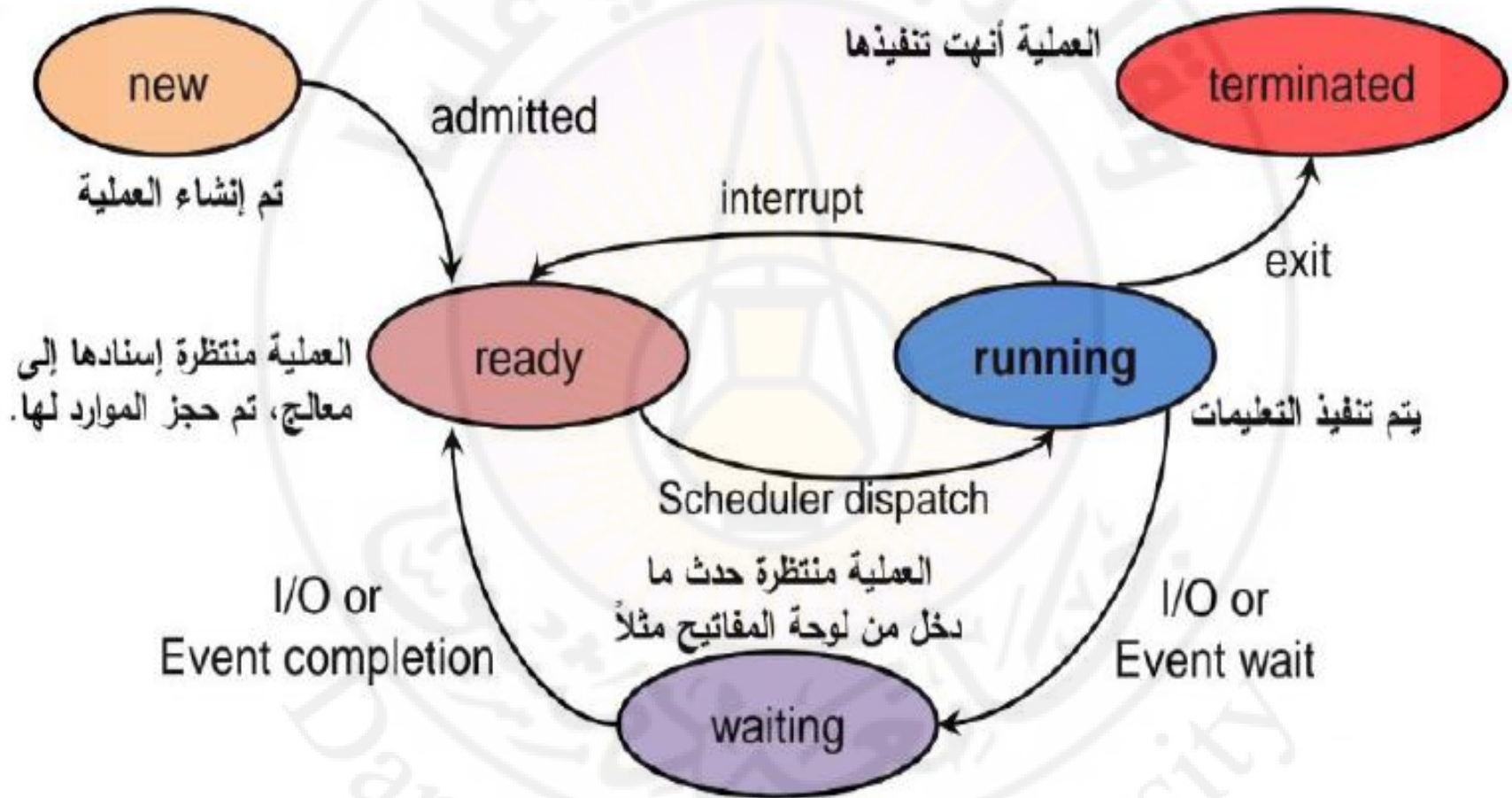
الإجرائية في الذاكرة Process in memory

توزيع أقسام الذاكرة لبرنامج مكتوب اللغة C



نلاحظ أن المؤشر `values` يتم تخزينه ضمن الـ `Stack` لأنه **متحول محلي** ضمن التابع `main` ، لكن ما يقوم بالتأشير عليه يتم تخزينه ضمن الـ `heap` ،

Process state حالات الإجرائية



حالات الإجرائية Process state

1. **New**: مرحلة إنشاء العملية وتحميلها على ال RAM.
 2. **Ready**: العملية في حالة انتظار ضمن ready queue ريثما يتم نقلها للمعالج لينفذها, حيث أن ال ready queue يحتوي على عمليات يقوم الجدول scheduler بانتقاء عملية منها وفق خوارزمية معينة ليتم نقلها للتنفيذ إلى المعالج.
 3. **Running**: مرحلة تنفيذ التعليمات الخاصة العملية .
 4. **Waiting**: المرحلة التي تكون فيها العملية في حالة انتظار حصول حدث ما , مثل عندما تنتظر I/O فيتم نقلها إلى ال waiting queue إلى أن تنتهي عمليات ال I/O التي لا يتدخل بها المعالج وبعد الانتهاء ترجع العملية إلى ال ready queue.
 5. **Terminated**: انتهى تنفيذ العملية.
- أثناء تنفيذ ال process إما يحصل عليها interrupt أو تصبح في حالة انتظار ل i/o events أو تنتهي.

يحصل نوعين من ال interrupts التي تنقل العملية إلى حالة ال ready queue:

- Software interrupts أو ما يسمى events أو trap (مثل ال exceptions).
- Hardware interrupt نبضة كهربائية تدل انتهاء العملية أو وجود مشكلة ما .

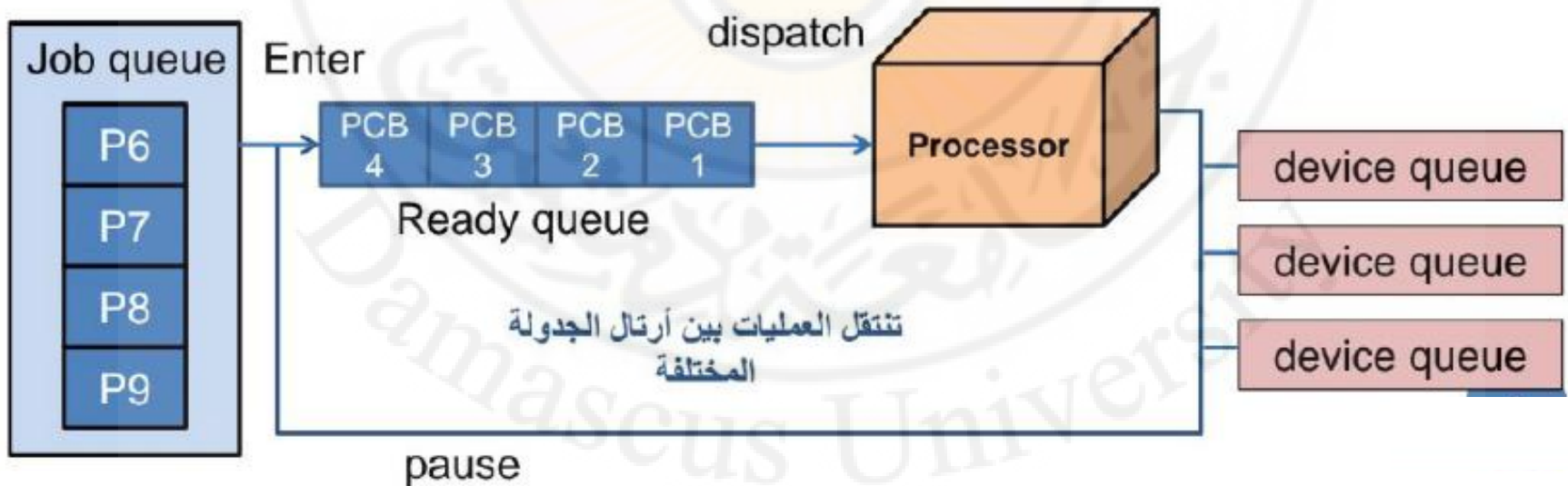
جدول التحكم بالإجرائية (PCB) Process Control Block

Process State
Process number
Program counter
Registers
Memory limits
List of open files
.....

- يتم تمثيل كل عملية ضمن نظام التشغيل بـ PCB.
- حالة العملية: running, waiting ...
- معرف العملية: رقم فريد بكل عملية (مثل الرقم الجامعي)
- معرف المستخدم: المسؤول عن العمل الذي تقوم به العملية. (حساب المستخدم المسجل دخوله).
- معرف العملية الأب التي أنشأت هذه العملية (الأب).
- عدّاد البرنامج: موضع التعليمات التالية ليتم تنفيذها.
- مسجلات CPU: محتويات المسجلات المحددة للعملية ليتم تنفيذها ضمن CPU.
- معلومات الجدولة: تحدّد أولوية تنفيذ هذه العملية ضمن CPU.
- معلومات إدارة الذاكرة: تحدّد الذاكرة المخصصة لهذه العملية. (جداول الصفحة ...)
- معلومات accounting: CPU المستخدمة، الوقت المنقضي من البدء وحتى الانتهاء.
- معلومات حالة I/O: قائمة الأجهزة المخصصة للعملية، قائمة الملفات المفتوحة...

جدولة الإجراءية : أنواع الأرتال

- **الهدف:** زيادة استخدام وحدة المعالجة المركزية، وتحويل العمليات بسرعة إلى CPU بهدف مشاركة الوقت.
- من بين العمليات المتاحة ؛ يختار **المجدول** العملية التالية ليتم تنفيذها على CPU.
- **رتل المهام (job):** مجموعة العمليات في النظام.
- **رتل الجاهزية (ready):** مجموعة العمليات الموجودة ضمن الذاكرة الرئيسية، جاهزة وتنتظر التنفيذ. (يمثل هذا الرتل المنفذ الوحيد لوصول العملية إلى وحدة المعالجة).
- **رتل الجهاز (device):** مجموعة العمليات التي تنتظر النفاذ إلى جهاز دخل/خرج.



المجدول Scheduler

○ المجدول قصير الأمد (مجدول CPU)

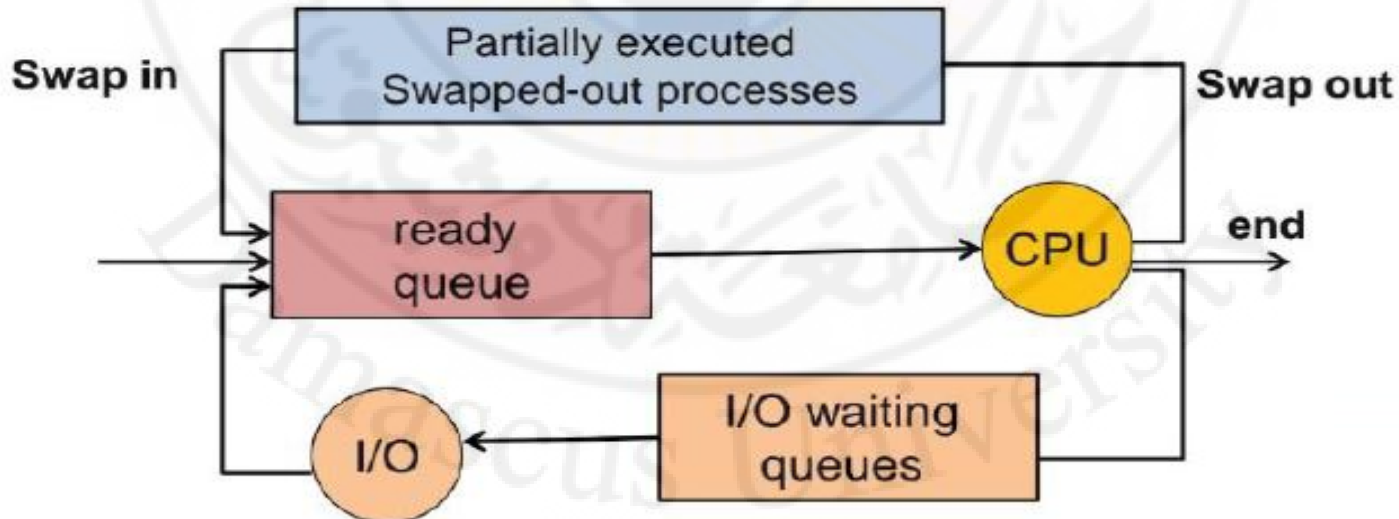
- يختار العملية التالية التي ينبغي أن تنفذ، ويخصّص لها CPU.
- يُستدعى بشكل متكرر (من رتبة ميلي ثانية، 100ms مثلاً)، وبالتالي يجب أن يكون سريعاً.

○ المجدول طويل الأمد (مجدول المهام)

- يختار العمليات التي ينبغي جلبها إلى رتل الجاهزية.
- يُستدعى بشكل غير متكرر (ثواني، دقائق)، وبالتالي يمكن أن يكون بطيئاً.
- يتحكّم بدرجة تعدّد البرمجة.

○ المجدول متوسط الأمد

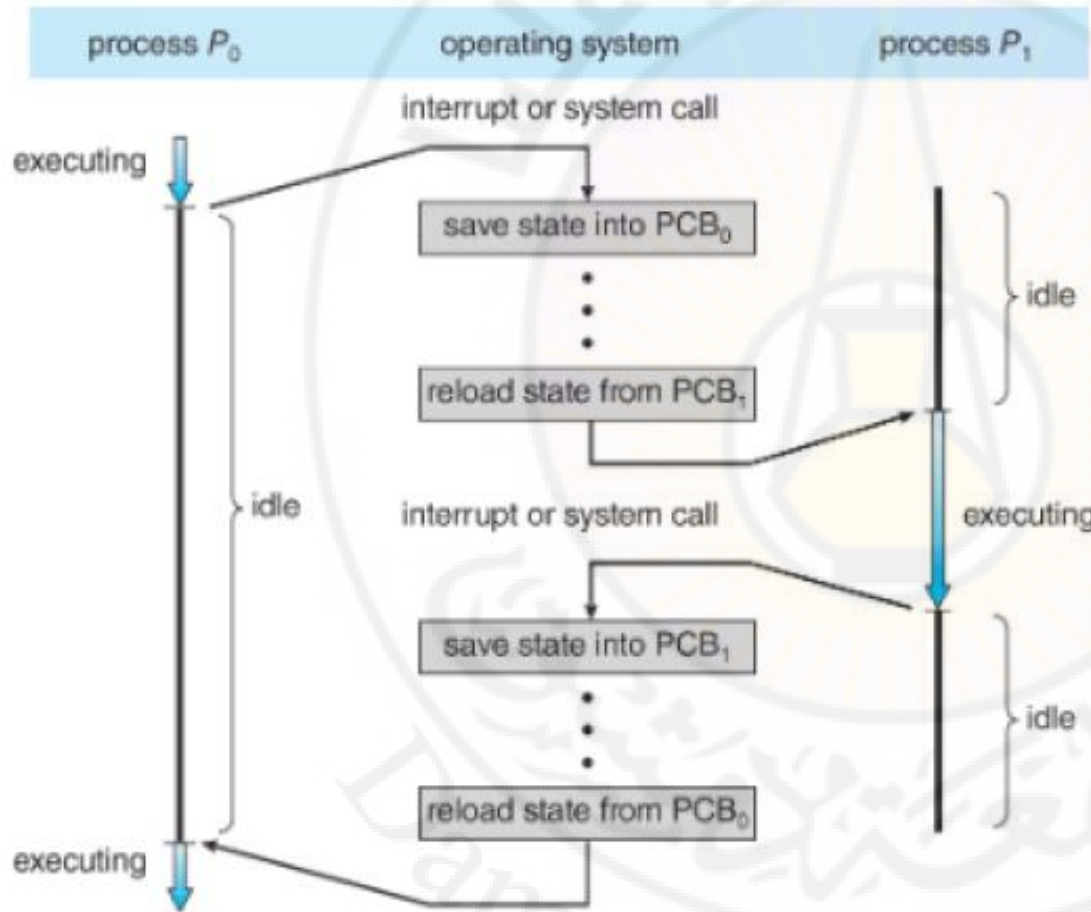
- يحذف عملية من الذاكرة (swap out)، يخزنها على القرص لبضع ثواني، يجلبها ثانية من القرص لتتابع تنفيذها. يمكن أن يضاف عندما يراد تخفيض درجة تعدّد البرمجة.



تبديل السياق Context Switch

- هي التقنية التي سمحت بال multi-programing على CPU وحيد بنواة وحيدة.
- يقوم المعالج بالتبديل بين عملية وأخرى بشكل سريع ليبدو للمستخدم تنفيذ عدة عمليات في الوقت ذاته.
- عملية التبديل بين العمليات تستلزم حفظ الحالة والسياس التي وصلت إليها عملية معينة, لذلك يتم حفظ حالة العملية وسياقها الحالي ضمن البنية PCB .
- إن الوقت الذي يتم فيه التبديل بين العمليات هو وقت مهدور overhead لأن المعالج لا يقوم بتحقيق أي وظيفة
- كلما كانت العملية أكبر complex process و نظام التشغيل معقد أكثر , كلما زاد الوقت اللازم لعملية ال context-switch وزاد الوقت المهدور من ال CPU time.
- في بعض المعالجات يوجد أكثر من مجموعة من السجلات ضمن المعالج وبالتالي يمكن الاحتفاظ بعدة process contexts ضمن المعالج ذاته.

تبديل السياق Context Switch



هذا المخطط يوضح طريقة ال Context Switching : بدايةً كان المعالج في مرحلة تنفيذ العملية P_0 وبعد حدوث مقاطعة

(Interrupt) أو (System-call) توقف تنفيذ العملية P_0 و تم حفظ حالتها ضمن ال PCB_0 , و من ثم تم تحميل الحالة الخاصة بالعملية P_1 من PCB_1 و بدأت بالتنفيذ, ثم بالمثل يتم توقف العملية P_1 بعد حدوث مقاطعة (Interrupt) أو (System-call) فيتم حفظ حالتها ضمن ال PCB_1 و يستأنف بعدها تنفيذ العملية السابقة P_0 .

نلاحظ أن زمن التنفيذ قليل مقارنة بالزمن اللازم لتوقف عملية وحفظ حالتها واسترجاع حالة عملية سابقة ومن ثم استئناف تنفيذها. لذلك يوجد هدر وفي الوقت .

إنشاء أجرائية Process Creation

- عملية إنشاء العمليات processes تكون من خلال هرمية معينة : حيث تقوم عملية ما (تدعى العملية الأب) بخلق عمليات جديدة (تدعى بالعمليات الأبناء) و التي بدورها يمكنها أن تخلق عمليات جديدة
- فيتشكل لدينا بما يدعى بشجرة العمليات حيث يكون ال root الخاص بهذه الشجرة هو ال process الخاص بال Kernel
- بشكل عام يتم تحديد العملية و إدارتها من خلال ما يسمى معرف العملية (process ID pid)

• يوجد عدة أنواع من مشاركة الموارد بين العمليات الآباء والأبناء و منها :

- العمليات الآباء و الأبناء تتشارك نفس الموارد .
- العمليات الأبناء تشارك جزء من موارد عمليات الآباء .
- العمليات الآباء و الأبناء لا تتشارك في الموارد أبدا .

• كما يوجد خيارات للتنفيذ بين العمليات الآباء والأبناء :

- تنفذ عمليات الآباء و الأبناء في وقت واحد (concurrent execution)
- تنتظر عمليات الآباء حتى تنتهي عمليات الأبناء من التنفيذ عن طريق تعليمة (wait)

انشاء أجزائية Process Creation

□ كيف يتم إنشاء عملية جديدة من خلال عملية أب ؟

يتم نسخ ال address space الخاص بالعملية الأب ويتم تخصيصها للعملية الجديدة , ومن ثم يتم تحميل البرنامج الخاص بالعملية الجديدة إلى هذا ال address space.

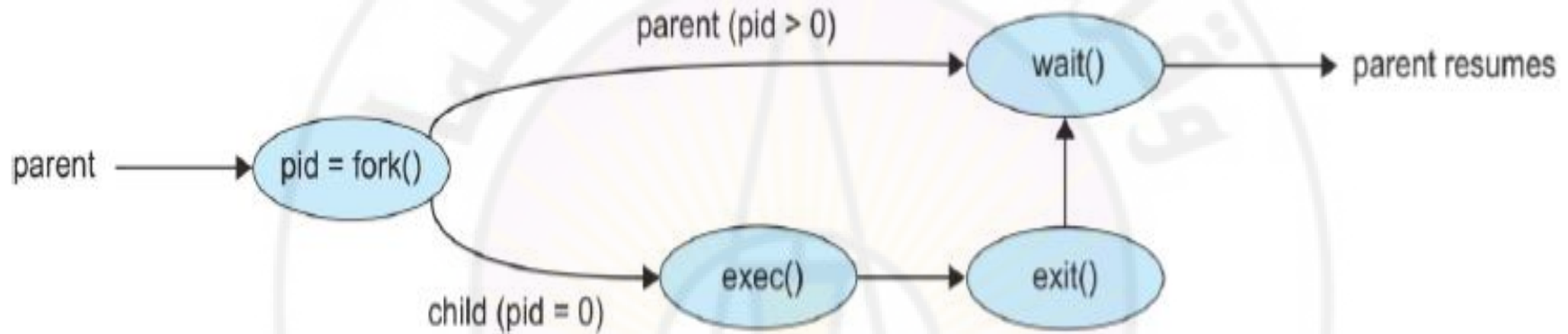
مثال عن عملية إنشاء عملية جديدة في نظام التشغيل (UNIX) :

❖ **fork()** تعليمة هي System Call لإنشاء new process , تنسخ ال address space الخاصة بالعملية الأب للابن , وبعد تطبيق هذه العملية يتفرع عنها عمليتين: العملية الأب والعملية الجديدة الابن.

❖ **exec()** تعليمة هي system call يتم استدعاءها بعد تعليمة fork() , مهمتها تبديل ال address space الذي تم نسخه من العملية الأب ليصبح خاصا بالبرنامج الجديد الخاص بالعملية الجديدة (مما يضمن تبديل الكود "الخاص بالأب في ال address space المنسوخ " بالكود الخاص بالعملية الابن).

❖ **wait()** تعليمة تقوم بتنظيم بين عمل العمليات ال child وال parent , حيث يتم استدعاؤها في العملية الأب وذلك لانتظار تنفيذ العملية الابن.

إنشاء إجرائية Process Creation



ملاحظات:

- `fork()` تقوم بخلق عملية جديدة من العملية الأب وبعدها تكمل العمليتان تنفيذ البرنامج بشكل متوازي ومنفصل , حيث تقوم التعليمات `fork()` بارجاع:
 - 0 : إذا كان التنفيذ في إطار ال Child process .
 - Child pid (عدد موجب) : إذا كان التنفيذ في إطار ال Parent process .
 - عدد سالب : في حال فشل عملية ال `fork()` .
- المقابل للتعليمات `fork()` بالنظام windows هي `CreateProcess()` حيث تنسخ ال adress space من ال process المعرفة فيها لنسخة أخرى .

إنشاء إجرائية بلغة الـ C

□ مثال على إنشاء إجرائية بلغة الـ C

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    pid_t pid;
    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
        printf("LINE J"); // will not be executed Why?
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("parent code");
    }

    return 0;
}
```

إنشاء إجرائية بلغة الـ C

- الـ `pid_t` هي data type في لغة الـ C و تعني الـ Process ID عرفنا من هذا النمط متحول `pid` وأسندنا له `fork()` وهي System call لخلق process جديدة (child process).
- كل التعليمات بعد `fork()` ستنفذ مرتين , مرة مع الـ parent process و مرة مع الـ child process.
- الـ `pid` الآن إما تحتوي على الـ process id الخاص بال (child) إذا نحن في تنفيذ العملية الأب, أو تحتوي القيمة 0 إذا نحن الآن بتنفيذ العملية الابن.
- حالة تنفيذ العملية الابن (حالة `pid=0`) : ستنفذ التعليمات `execlp` وهي من عائلة التعليمات `exec()` و ستقوم باستبدال الكود المتبقي بالكود الخاص بالتعليمات `ls` , لذلك لن يتم طباعة السطر (LINE J) وذلك لأن الكود الخاص بطباعة هذا السطر تم استبداله بالكود الخاص بـ `ls` لذلك ستنفذ التعليمات `ls` والتي تقوم بعرض محتويات المجلد الحالي .
- حالة تنفيذ التعليمات الأب (حالة `pid = child pid > 0`) ستقوم هذه العملية بانتظار العملية الابن حتى تنتهي ثم ستقوم بطباعة `parent code`.
- إذا الخرج الكلي للبرنامج هو خرج التعليمات `ls` أولاً ثم تتم طباعة الجملة `parent code`.

إنهاء الإجرائية Process Termination

عند نهاية تنفيذ process معينة تقوم بطلب حذفها (إنائها) من نظام التشغيل باستخدام ال system call `exit()` فيحدث ما يلي :

- سيتم إرجاع ال status الخاصة بال process ال Child إلى العملية ال Parent (من خلال `wait()` , و ال system call `wait()` تعني أن ينتظر ال parent انتهاء تنفيذ ال Child) .
- يقوم نظام التشغيل بعملية resources deallocation (تحرير الموارد) الخاصة بال process .
- يمكن لل Parent إنهاء تنفيذ العمليات ال Children الخاصة به باستخدام استخدام ال system call `abort()` , وال Parent هو الوحيد الذي يملك هذه الصلاحية, من الممكن أن يقوم بهذه العملية لعدة أسباب منها :
 - العملية ال Child تجاوزت الموارد المخصصة لها.
 - المهمة التي ينفذها ال Child لم يعد لها حاجة .
 - في حال إنتهاء العملية ال Parent وكان نظام التشغيل لا يسمح لل Child بإكمال تنفيذه في انتهاء تنفيذ الأب

بعض أنظمة التشغيل لا تسمح بوجود ال Child في حال أن ال Parent أنهى عمله, أي إذا أنهى ال Parent مهمته فعلى كل ال Children أن تنتهي أيضا ومن هنا يأتي مصطلح ال **cascading termination** : ويعني أنه عندما يتم إنهاء عملية أب ما يتم إنهاء كل العمليات ال Child و ال Grandchildren حتى آخر ابن , حيث أن نظام التشغيل هو من يبدأ بعملية الإنهاء .

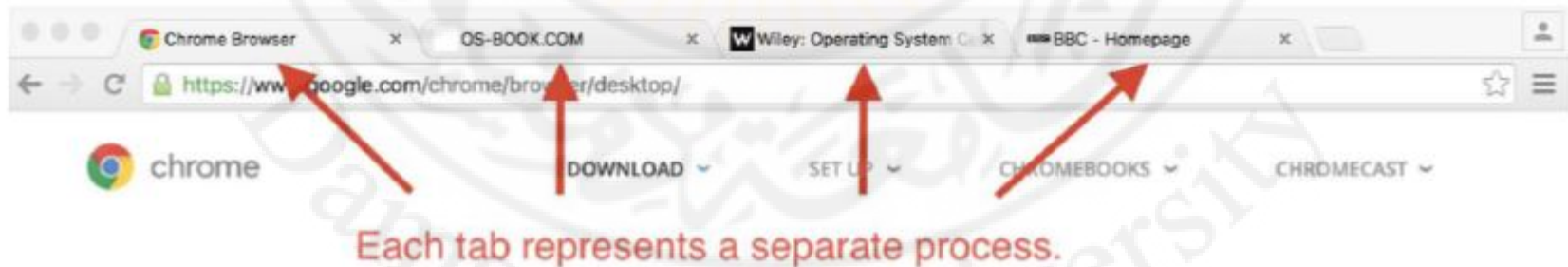
مثال على Multiprocess

□ قديماً كانت المتصفحات web browsers قادرة على تشغيل process واحدة فقط (one tab), و نستطيع أن نفتح من نفس ال web browsers أكثر من نسخة, فإذا تسبب أحد مواقع الويب (أحد نسخ ال web browser) بمشكلة فالمتصفح قد يتوقف عمله بشكل كامل.

□ متصفح Google Chrome له تصميم multiprocess حيث أنه بتشغيله يتم تنفيذ ثلاثة أنواع مختلفة من ال process:

Browser process: تقوم بإدارة واجهة المستخدم و إدارة القرص و عمليات الدخل و الخرج على الشبكة
Renderer process: تقوم بعرض صفحات ال web ومعالجة ال HTML و JavaScript, يتم إنشاء عارض جديد renderer لكل صفحة web.

Plug-in process: تم خلق هذا النوع من العمليات من أجل كل plug-in في المتصفح.



الاتصال بين الإجراءات Inter-process Communication

العمليات في النظام تصنف إلى نوعين :

Independent process : العمليات المستقلة لا تتأثر و لا تؤثر بتنفيذ العمليات الاخرى .

Cooperating process : العمليات المترابطة يمكن ان تتأثر أو تؤثر بتنفيذ غيرها من العمليات , أي يتم تبادل المعلومات بين بعضها البعض .

□ العمليات المترابطة تتطلب وجود تواصل ما بين هذه العمليات inter-process communication (IPC) .

□ من الممكن أن تكون ال Cooperating processes على two systems مثلا المتصفح و ال server الذي يتعامل معه كل منهما يخلق process تتواصل مع الأخرى .

□ وقد تكون ال Cooperating processes على one system .

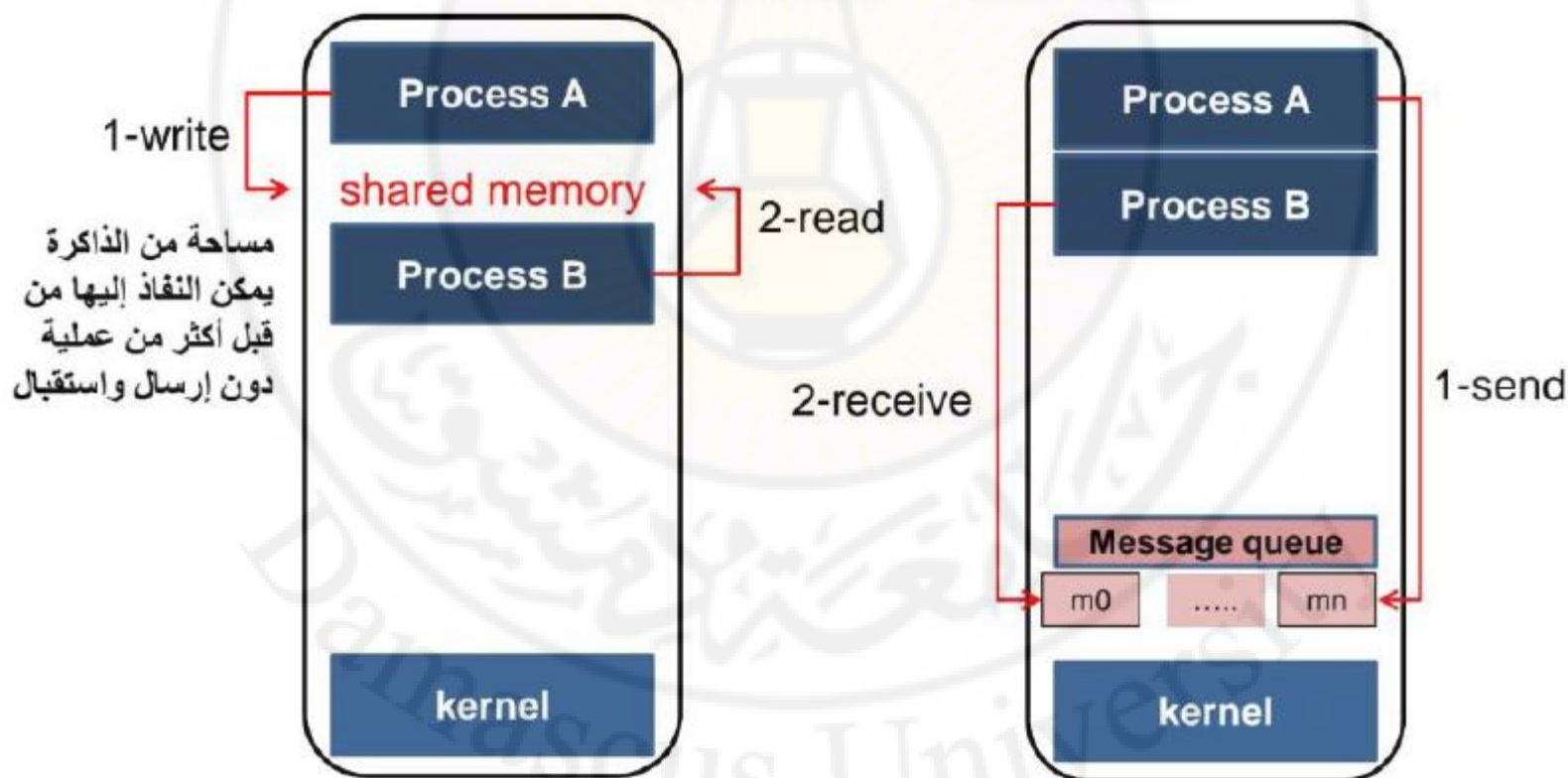
□ هناك نوعين من ال IPC

Shared memory الذاكرة المشتركة

Message passing تبادل الرسائل

الاتصال بين الإجراءات Inter-process Communication

في ال Message passing يكون تبادل الرسائل عن طريق ال kernel, أما في ال shared memory فليس هناك حاجة لل OS بشكل كبير فهناك مساحة مخصصة من ال RAM تستخدم للتواصل بين ال .processes



الاتصال بين الإجراءات Inter-process Communication

الذاكرة المشتركة	تبادل الرسائل	
الأداء	جيد وفعال	جيد للتطبيقات التي لا تحتاج إلى اتصال كثيف بين المعالجات
التطبيق	سهلة التطبيق ولكنها مكلفة ماديا	سهلة التطبيق ومنخفضة التكلفة
وسيلة الاتصال	ذاكرة موحدة لجميع العمليات	عبر خطوط الاتصال في الشبكة
السلبيات	يجب توفير أساليب حماية الذاكرة المشتركة وتنظيم الوصول إليها	تكلفة تمرير الرسائل عالية جدا لأنها تتطلب عمليات دخل خرج كثيرة

الاتصال بين الإجراءات في الـ Windows

- يتمحور الـ Message-Passing حول خاصية الـ advanced local procedure call (LPC).
- تعمل فقط بين العمليات على نفس النظام.
- تستخدم الـ ports (مثل mailboxes) لإنشاء و صيانة قنوات الاتصال.

يعمل الاتصال على النحو التالي:

1. يفتح العميل مقبض لمنفذ اتصال بالنظام الفرعي.
2. يرسل العميل طلب اتصال.
3. ينشئ الخادم منفذ اتصال خاصين ويعيد المؤشر إلى أحدهما إلى العميل.
4. يستخدم العميل والخادم مقبض المنفذ الثاني لإرسال الرسائل أو لعمليات الاستماع إلى الردود.

السنة الرابعة

نظم تشغيل 1



المحاضرة الرابعة

Threads & concurrency

د. المثني خضر

كلية الهندسة المعلوماتية

المحاضرة الثانية: المفردات الرئيسية

1. مفهوم النياسب Threads Concept
2. مكونات النيسب
3. Single and Multithreaded Processes
4. تحديثات Multicore Programming
5. التوازي Parallelism
6. التزامن Concurrency
7. نياسب المستخدم ونياسب النواة
8. نماذج النياسب المتعددة
9. مكتبات النياسب Thread Libraries
10. العمليات على النياسب في مكتبة الـ Pthreads
11. Implicit Threading
12. Thread Pools
13. Scheduler Activations
14. Windows Threads
15. Linux Threads

مفهوم النياسب Threads Concept

ال Thread: هي light weight process تستطيع العمل بسرعة أكبر و لها ميزات خاصة بها. نستطيع القول أن أصغر شيء قابل للتنفيذ هو ال thread, و كل process قد تخلق العديد من ال threads حسب حاجتها.

- معظم التطبيقات الحديثة تكون multithreaded , في البرنامج الواحد تخلق ال process أكثر من thread و تطبقها .
- بشكل عام تكون ال Kernels لأنظمة التشغيل multithreaded .
- ال thread تساعد على تبسيط الكود البرمجي إضافة لزيادة كفاءة efficiency البرنامج أو التطبيق .
- عملية خلق process يعتبر مكلفاً heavy-weight مقارنة بعملية إنشاء thread التي تعتبر أقل كلفة light-weight.
- إن خلق ال Thread في نظام Solaris هو أسرع بـ 5 مرات من خلق Process وتعد ال Context-switching لديه أسرع بـ 30 مرة في حال التعامل مع ال Threads.

مكونات النيسب

□ كل thread لها المكونات التالية :

1. thread ID

2. Program Counter

3. register set

4. Stack

□ إن أي process تقليدية أو ما يدعى بال heavy-weight process تخلق thread واحدة فقط.

□ أما إذا كانت العملية تحتوي على أكثر من thread (multiple thread), فعندها يمكن أن تنفذ أكثر من مهمة في نفس الوقت على التوازي . ويتم التواصل بين ال Threads عن طريق Signals نبضات كهربائية أو عن طريق Interrupts .

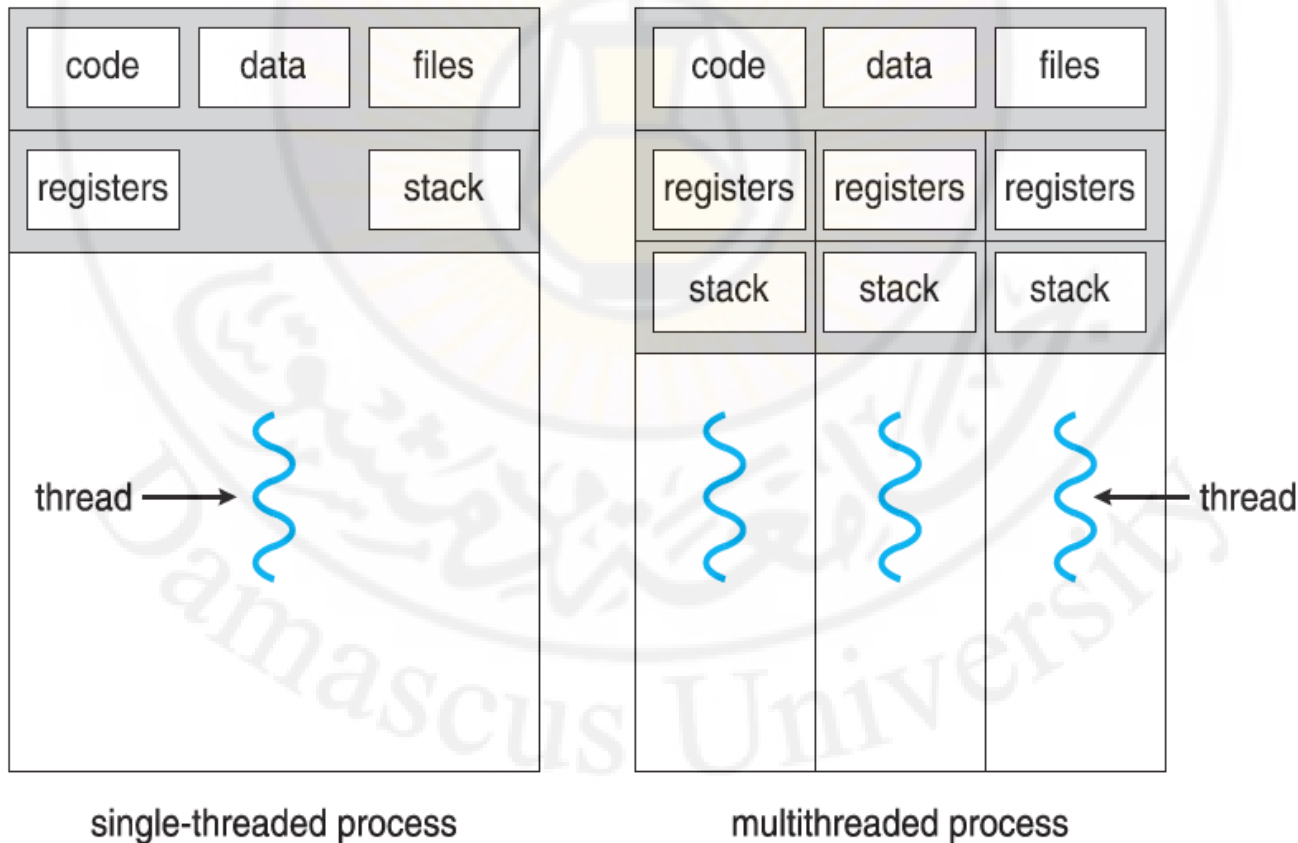
➤ لم يكن نسخ ال address space للعملية الأب لأجل العملية الابن حلاً جيداً, بسبب هدر جزء كبير من الذاكرة في حال كان الأب له مساحة كبيرة لا يحتاج الابن سوى جزء صغير منها .

➤ اتجه المطورون الى فكرة Shared data و التي تقوم على مشاركة الأب و الابن نفس ال address space وبذلك يتم توفير جزء كبير من الموارد.

Single and Multithreaded Processes

□ تشارك كل thread مع ال threads الباقية والتي تنتمي لنفس ال process في كل من قسم ال code البرمجي الخاص بها (Code section) , وقسم البيانات (data section) , إضافة لموارد أخرى في نظام التشغيل كالملفات وال signals (Heap).

□ كل thread على حدى لها ال program counter الذي يشير اليها و registers و stack .



تحديات Multicore Programming

- إن أنظمة ال Multicore او ال Multiprocessor تضع أمام المبرمجين التحديات التالية:
- **تجزئة الأنشطة Dividing activities:** هذا يتضمن عملية فحص التطبيق بحثاً عن المهام التي يمكن عزلها لتكون مستقلة ومتزامنة , مثلاً لدي برنامج يقوم بالقراءة وإجراء عملية حسابية مستقلة , فنجعل ال process تخلق 2 threads كل واحدة تنفذ مهمة .
- **التوازن Balance:** عندما نقسم ال activities يجب خلق التوازن بينها من حيث زمن التنفيذ , حيث أن بعض ال task (process or thread) يكون زمن تنفيذها نفسه إذا قمنا بتقسيمها لعدة مهام , أي كل مهمة ستأخذ وقت نفسه الوقت الذي ستستغرق في حال تنفيذها دون تقسيم , عندها لن يكون لاستخدام core منفصلة لتنفيذ هذه المهمة أي جدوى تستحق تكلفة القيام بذلك.
- **فصل البيانات Data splitting:** تماماً كما تم فصل التطبيق إلى مهام مستقلة , أيضاً البيانات التي يتم التعامل معها من قبل المهام لابد أن يتم تجزئتها لتعمل على cores مستقلة.
- **ترابط المعطيات Data Dependency:** لابد من اختبار إمكانية وجود ترابط بين البيانات التي يتم الوصول إليها من مهام ما مع مهام أخرى , فمثلاً عند اعتماد مهمة ما على بيانات ناتجة من مهمة أخرى , على المبرمجين أن يضمنوا بتمام عمليات المزامنة بين المهام بحيث تتوافق مع ترابط المعطيات بينها للحصول على بيانات صحيحة.
- **Testing and Debugging:** عندما يعمل برنامج ما على التوازي مع الآخر وعلى multiple cores فعندها العديد من مسارات التنفيذ تكون مطروحة وممكنة , ولذلك فإن عمليات الاختبار وال debugging تكون أصعب منها في التطبيقات العاملة ك single-thread.

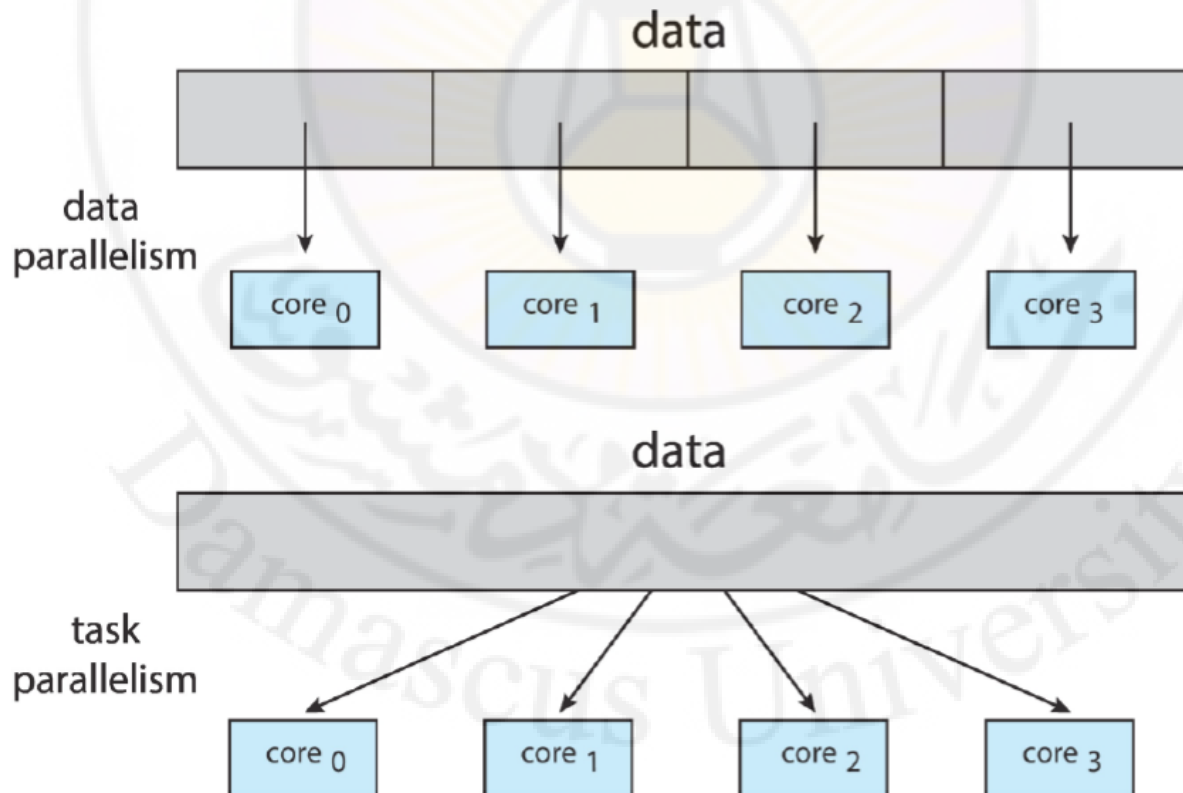
التوازي Parallelism

□ وتعني بأن النظام يمكن أن ينفذ أكثر من مهمة في الوقت ذاته, تنفذ على معالجات Multicore فقط

□ أنواع التوازي Parallelism:

1. **Data Parallelism**: تقوم بتوزيع مجموعات جزئية من البيانات على ال multiple cores والتي تتبع لنفس العملية.

2. **Task Parallelism**: هنا يتم توزيع ال threads على ال cores بحيث يقوم كل نيسب بتنفيذ مهمة منفصلة.

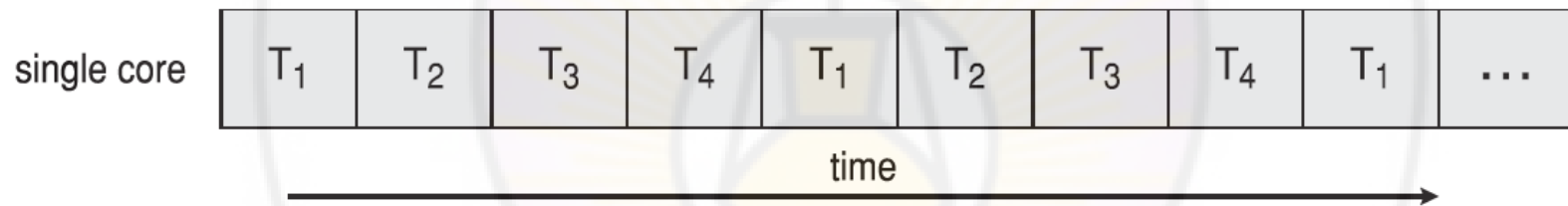


التزامن Concurrency

□ تعني دعم أكثر من مهمة لتكون قيد التنفيذ بشكل متتالي، و هي شبيهة بال-Parallelism.

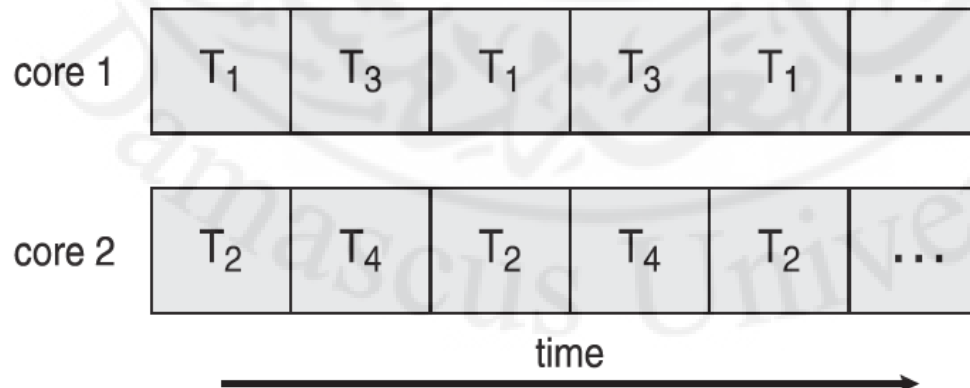
□ التزامن concurrency يقوم باستغلال الوقت و سرعة ال- CPU في **المعالج الواحد** ويحتاج إلى scheduler لاختيار المهمة المختارة للتنفيذ ضمن المعالج ضمن آلية معينة، أما ال-Parallelism يحتاج إلى **أكثر من core**.

□ شكل يوضح تنفيذ التزامن في نظام single-core :



هذه العملية تدعى بال-**overlapping** حيث أن المعالج عاد لتنفيذ T₁ مرة ثانية بعد T₄.

□ شكل يوضح التوازي في نظام multi-core :



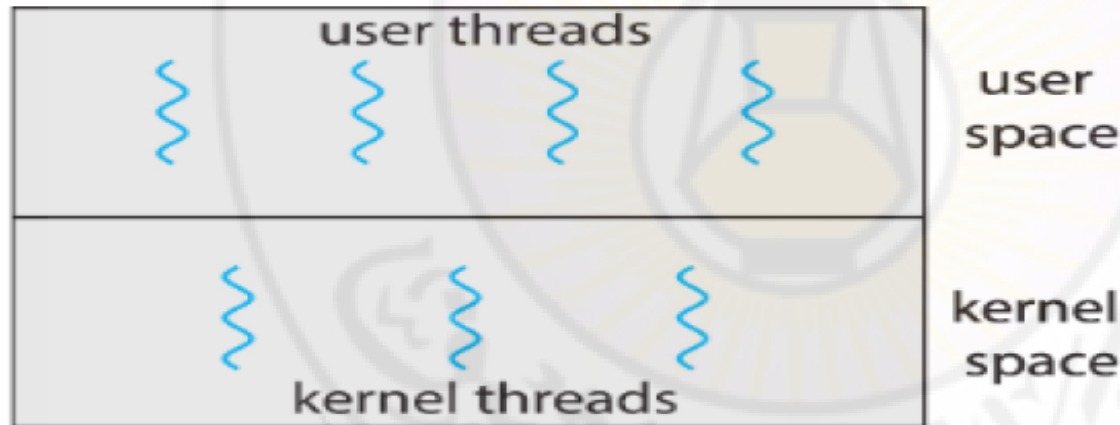
نياسب المستخدم ونياسب النواة

□ **User Thread**: تتم إدارتها من قبل مكتبات ال threads على مستوى المستخدم " user-level threads library " وهي ثلاث مكتبات رئيسية:

▪ **POSIX Pthreads** (POSIX = Portable Operating System Interface) و فيها مكتبة إدارة ال thread تدعى Pthreads.

▪ **Windows threads**.

▪ **Java threads**.



□ **Kernel Threads**: تكون مدعومة من قبل ال Kernel ومثال عليها جميع أنظمة التشغيل ذات التوجه العام general purpose operating systems بما فيها:

Windows – IOS – Linux – Android – Mac OS X.

نماذج النياسب المتعددة Multithreading Models

□ في مستوى ال Kernel لدينا Threads وفي مستوى ال User لدينا Threads, و يجب الوصل بينهما (mapping) ويوجد ثلاث طرق لتحقيق هذا :

Many-to-One ❖

One-to-One ❖

Many-to-Many ❖

Many-to-One

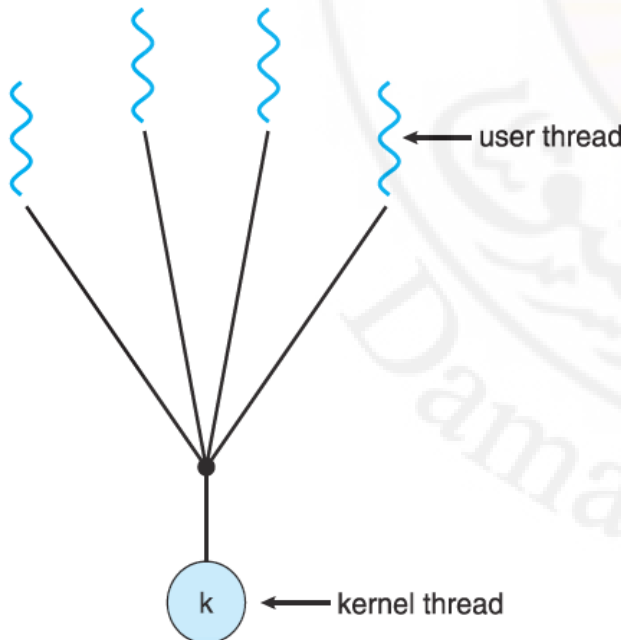
□ أي أكثر من user-level thread يتم ربطها مع kernel thread وحيدة .

□ إن مشكلة هذه الطريقة أن أي process تقوم ب blocking لأحد ال threads يؤدي ذلك إلى block لجميع ال threads الباقية.

□ كما أن ال Multiple threads قد لا تعمل على التوازي في أنظمة ال multicore لأنه يكون لدينا thread وحيدة فقط في ال kernel.

□ الأنظمة التي تعتمد هذا النموذج قليلة منها :

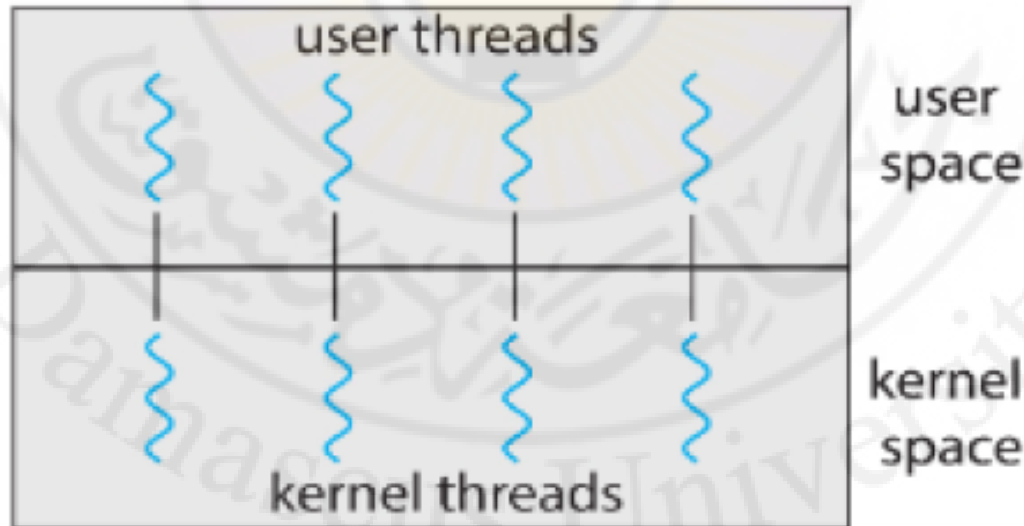
GNU Portable Threads - Solaris Green Threads



نماذج النياسب المتعددة Multithreading Models

One-to-One

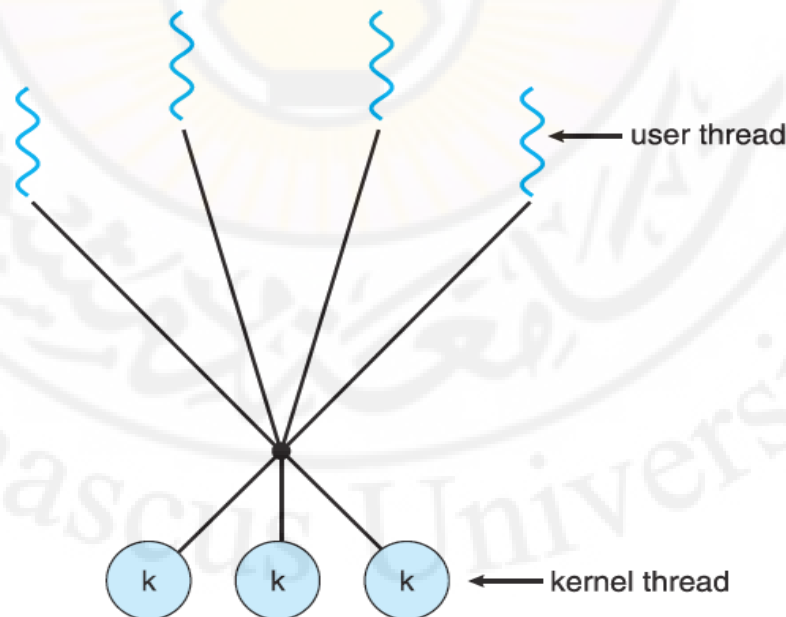
- كل user-level thread يتم ربطها مع kernel thread وحيدة .
- إن إنشاء user-level thread يؤدي حكماً لإنشاء kernel thread.
- هنا يكون الـ concurrency و التوازي أعلى منه في النموذج السابق.
- عدد الـ thread في العملية الواحدة قد يكون محدوداً بسبب الـ overhead.
- من الأنظمة التي تعتمد هذا النموذج: Linux – Windows.



نماذج النياسب المتعددة Multithreading Models

Many-to-Many

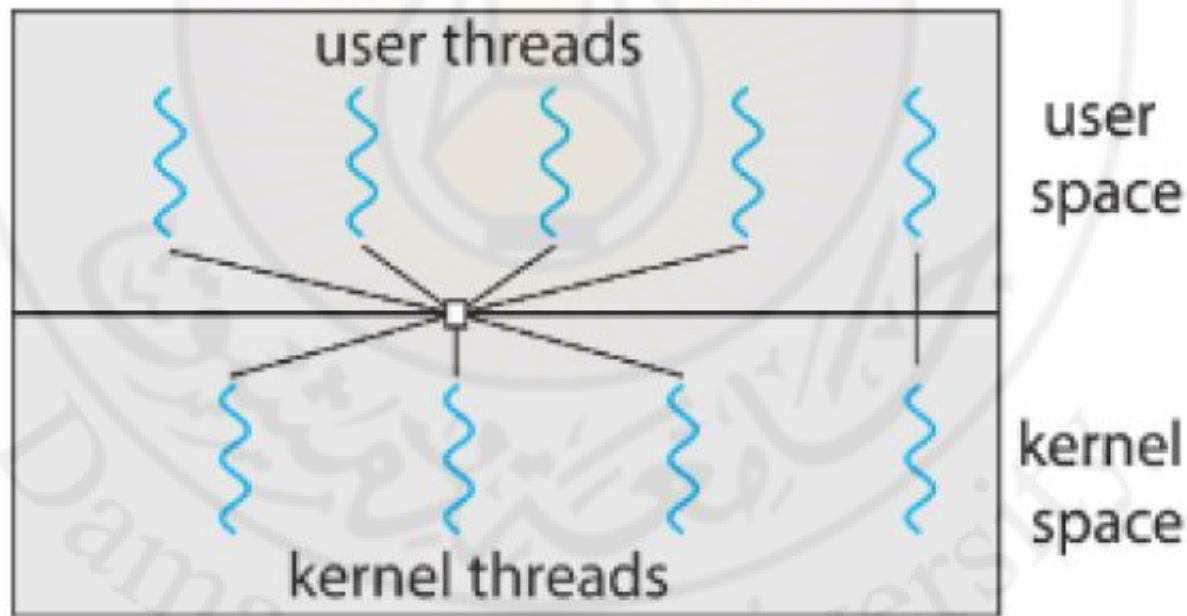
- تسمح بربط أكثر من user-level thread بأكثر من kernel thread.
- تسمح لنظام التشغيل بإنشاء عدد كاف وفعال من الـ kernel thread.
- طريقة الربط ليست بالتسلسل , نظم التشغيل هو من يحدد كل user thread عليها ان ترتبط بأي kernel thread.
- من الأنظمة التي تعتمد هذا النموذج: Windows with the Thread Fiber package .



نماذج النياسب المتعددة Multithreading Models

Two-level Model

- هو الأكثر استخداماً وهو مشابه لنموذج many-to-many .
- ولكن هنا يُسمح لـ user thread بأن يكون ملتزم بـ kernel thread .



مكتبات النياسب Thread Libraries

- المكتبات توفر للمبرمج API من أجل إنشاء وإدارة الـ threads.
- هناك طريقتا رئيسيتان من أجل القيام بـ implementing للمكتبات:
 1. Library entirely in user space: يقوم المستخدم بتضمين المكتبة أثناء بناء البرنامج.
 2. Kernel-level library supported by the OS: مكتبات النواة مدعومة داخل الـ OS.

مكتبة الـ Pthreads هي من الـ POSIX و هي مكتبة خاصة بالـ (LINUX, UNIX) لا يمكن استعمالها بغير نظام) و تحتوي على مجموعة من التعريفات التي يمكن استعمالها, و تسمح للمستخدم بإدارة الـ Threads.

- قد تكون الـ Pthreads إما على شكل user-level أو على شكل Kernel-level.
- الـ Specification : أي أن الـ API تحدد سلوك مكتبة الـ thread.
- أما الـ implementation فهو على عاتق الـ development للمكتبة.

pthread_create()

□ pthread_create() هو system call معرف في POSIX.

□ تقوم pthread_create() بإنشاء thread جديدة وتجعله executable، وهذا التابع يمكن أن نستدعيه عدد غير منته من المرات وفي أي مكان ضمن الكود البرمجي.

□ الشكل العام لهذه التعليمة
pthread_create(threadID, attr, start_routine, arg);
حيث أن :

✓ **threadID**: هو رقم مميز ووحيد يعبر عن الـ thread الجديدة.

✓ **attr**: يستخدم لتحديد attributes خاصة للـ thread، بحيث يمكن أن تحدد attribute object أو تكون NULL.

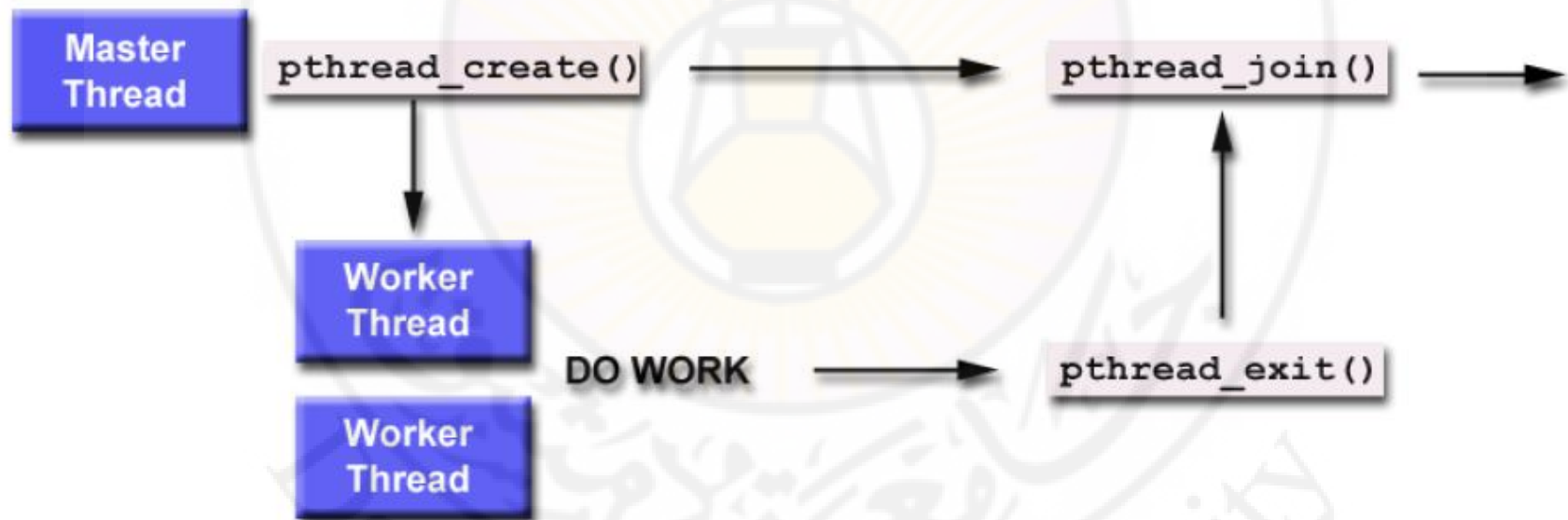
✓ **start_routine**: تعبر عن الـ C routine التي سينفذها الـ thread في لحظة إنشائه.

✓ **arg**: عبارة عن argument يمكن أن نمرره لـ start_routine، لابد أن نمرره by reference أي كمؤشر من النمط void، ويكون NULL في حال لم نرد تمرير أي argument.

pthread_join (threadid,status)

□ تقوم pthread_join() بحظر ال thread المُنشأة وذلك حتى تنتهي thread التي قامت بإنشائها من عملها.

□ ال joining هو أحد الطرق المتبعة لتحقيق ال synchronization بين ال threads.



مثال (1) على إنشاء و ضم النياسب في Pthread

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun(){
    sleep(1);
    printf("Printing from Thread \n");
    int i, upper = 15;
    int sum = 0;
    for (i = 1; i <= upper; i++)
        sum += i;
    printf("from thread sum is:%d:\n",sum);
    return NULL;}
int main()
{pthread_t tid;
    printf("Before Thread\n");
    pthread_create(&tid, NULL, myThreadFun,
        NULL);
    pthread_join(tid, NULL);
    printf("After Thread\n");
    exit(0);}
```

■ قمنا بتضمين مكتبة pthread ,
في الـ main() عرفنا thread
اسمها tid في أول تعليمة و
أنشئناها من خلال
pthread_create() و طلبنا ان
تقوم بـ function معين من
خلال تمريرنا للـ method التي
اسمها myThreadFun , لاحظ
اننا استدعيناها من غير أقواس
لان POSIX عرفتھا بهذا الشكل
, ثم من خلال
pthread_join قام النظام
بانتظار انتهاء تنفيذ هذه الـ
thread .

مثال (2) على إنشاء و ضم النياسب في Pthread

يقوم هذا ال code بضم عشر threads مع بعضها , فبالتالي كل thread تنتظر سابقتها حتى تنتهي .

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

Thread Cancellation

- ❑ هي عملية إنهاء ال thread وذلك قبل أن ينتهي عملها.
- ❑ ال thread المراد إلغائها تدعى target thread.
- ❑ هنالك أسلوبان متبعان لتحقيق ذلك:
- ❖ asynchronous cancellation : تنهي ال target thread بشكل فوري .
- ❖ deferred cancellation : هنا يسمح لل target thread بالتحقق إذا يمكن أن تلغى أو لا.

إنهاء thread في Pthread

□ عملية إنهاء thread تعتمد على حالة هذه الـ thread.

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

□ إذا كان thread cancellation هو disabled عندها يبقى الـ cancellation متوقف حتى تفعله الـ thread, يتم الـ Cancellation عندما تصل الـ thread الى cancellation point فتتحول من disabled الى enabled.

□ النمط الافتراضي للـ thread هو deferred.

□ في أنظمة linux يتم اختبار الـ thread cancellation من قبل الـ signals.

Mode	State	Type
Off	Disabled	—
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

Implicit Threading

- العدد المتزايد من استخدام الـ threads يجعل من المحافظة على صحة البرامج أكثر صعوبة عند استخدام الـ explicit threads وهي threads يتم إدارتها من قبل المبرمج.
- الـ implicit threads هي threads إلا أنها تدار من قبل الـ compiler و الـ run-time libraries.
- وفر هذا الكثير من الوقت والجهد على المبرمج حيث أصبح المطلوب منه فقط تحديد الـ task الذي يريده أن يتوزع على أكثر من core (parallel) أو على أكثر من thread و من ثم تهتم الـ libraries بالتفاصيل, مع بقاء إمكانية للمبرمج أن يحدد توزيع الـ threads إن أراد ذلك بنفسه.
- • يوجد خمسة طرق لتحقيق ذلك وهي :

..

1. Thread Pools
2. Fork-Join
3. OpenMP
4. Grand Central Dispatch
5. Intel Threading Building Blocks

Thread Pools

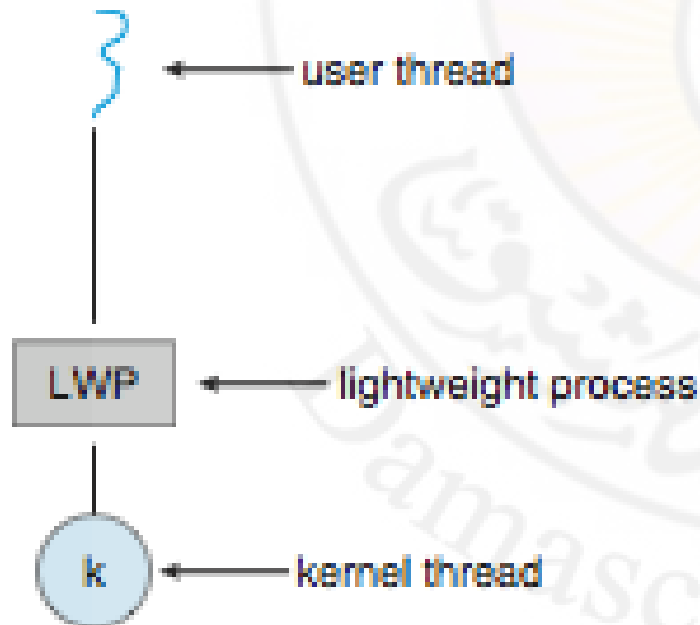
- ❑ فكرة هذه الطريقة بأن يكون هناك مجموعة من الـ Threads منشأة مسبقاً و تنتظر حتى يتم استخدامها, ويتم بذلك توفير وقت إنشاء الـ thread عند الحاجة لتنفيذها.
- ❑ يقوم المبرمج بإنشاء عدد من الـ threads داخل pool بحيث تنتظر أن يحين وقت تنفيذها.
- ❑ فوائدها :
 - ❖ عادة يكون من الأسرع أن يتم تلبية خدمة اعتماداً على thread موجودة مسبقاً بدلاً من إنشاء thread جديدة.
 - ❖ يسمح بأن يكون عدد الـ threads في التطبيق محدوداً بحجم الـ pool .
 - ❖ فصل المهام التي نريد تنفيذها عن طريقة إنشاءها, سمح بتواجد استراتيجيات مختلفة لتشغيل .task
 - ❖ يمكن جدولة المهام لتعمل بشكل دوري .
- ❑ يدعم Windows API الـ Thread pool

مثال على ذلك: الـ Web Server حيث بدلاً من أن يقوم بخلق thread جديدة لكل request فإنه مسبقاً قام بتجهيز عدد من الـ threads داخل pool وعندما يتلقى request جديدة يعطيها من الـ threads الموجودة في الـ pool وبذلك يكون قد وفر زمن خلق thread لكل request و هذا ضروري حيث أن الـ web server يمكن أن يتلقى عدد هائل من الـ requests في اللحظة الواحدة.

Scheduler Activations

- تحتاج ال threads في كل من Many-to-Many model و Two-level model إلى أن تتواصل للحفاظ على العدد المناسب من ال kernel threads المحجوزة للتطبيق .
- عادةً تستخدم بين ال user thread و ال kernel thread بنية معطيات data structure وسيطة تدعى lightweight process LWP , وظيفته القيام بتنظيم العلاقة بين ال user thread و ال kernel thread :

- ❖ تبدو أقرب لمعالج افتراضي بحيث يقوم بجدولة ال user threads ليتم تنفيذها.
- ❖ كل LWP مرتبطة ب kernel thread .



- هذا التواصل يسمح للتطبيق بأن يحافظ على العدد الصحيح والملائم من ال kernel threads .
- توفر Scheduler activations الية التواصل upcalls من ال kernel الى handler في ال thread library .

Windows Threads

Windows API – primary API for Windows applications ☐

كما تحقق one-to-one mapping وعلى مستوى ال kernel.

كل thread تحتوي :

thread id : رقم مميز للنيسب

register : مجموعة سجلات تمثل حالة المعالج

separate user and kernel stacks : من أجل تشغيل ال thread في user mode or kernel mode

private data storage : من أجل ال run-time libraries و DLL

☐ وتعرف الثلاث الأخيرة بما يسمى context of the thread

بنية الداتا الرئيسية لل thread تتضمن 3 مستويات:

▪ **ETHREAD (executive thread block)**: تتضمن مؤشر على ال process التي تنتمي لها ال

thread و مؤشر ثاني لل KTHREAD في ال kernel space

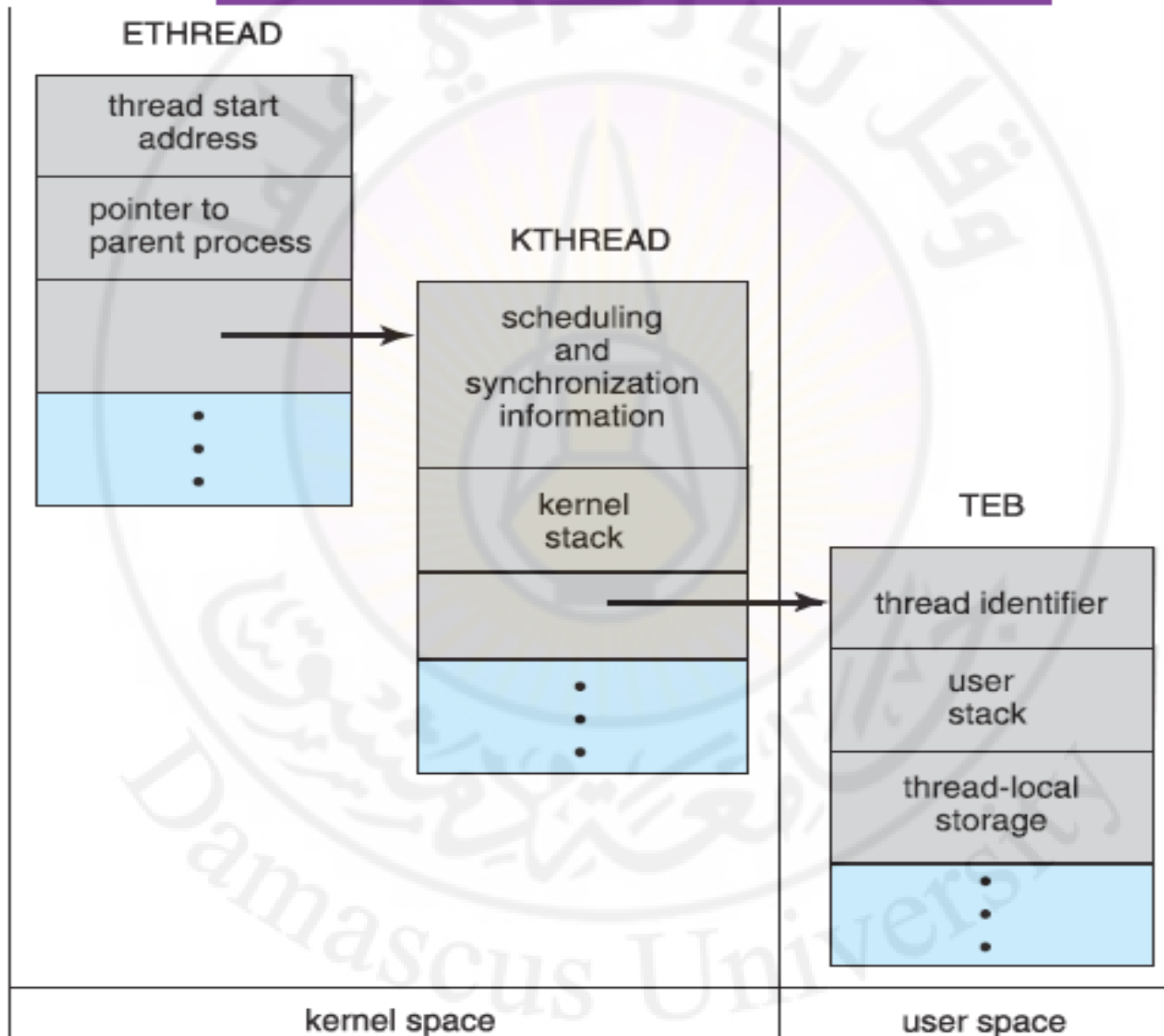
▪ **KTHREAD (kernel thread block)**: يحوي معلومات عملية الجدولة و المزامنة و kernel-

mode stack و مؤشر الى TEB في ال kernel space

▪ **TEB (thread environment block)**: وتحوي thread id, user-mode stack, thread-local

storage, في ال user space

Windows Threads



Linux Threads

- في Linux تسمى الـ threads بـ tasks.
- يتم انشاء thread من خلال system call clone().
- تسمح clone() للـ task الابن بأن يتشارك بفضاء العناوين للـ task الأب والتي تمثل الـ process.
- هيكل الـ task_struct يشير الى معالجة data structures الوحيدة و المشتركة.

Flags control behavior

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

السنة الرابعة

نظم تشغيل 1



المحاضرة الخامسة

CPU Scheduling

د. المثني خضر

كلية الهندسة المعلوماتية

المحاضرة الخامسة: المفردات الرئيسية

- ❖ مفاهيم أساسية
- ❖ جدول وحدة المعالجة المركزية (CPU scheduler)
- ❖ المرسل (Dispatcher)
- ❖ معايير الجدولة (Scheduling Criteria)
- ❖ خوارزميات الجدولة (Scheduling Algorithms)
- 1 القادم أولاً يخدم أولاً (First-Come, First-served)
- 2 العملية الأقصر أولاً (Shortest-Job-First (SJF)
- 3 الجدولة وفق الأولوية (Priority)
- 4 الجدولة الدورانية (Round Robin RR)
- 5 الجدولة المتعددة المستويات
- ❖ تمارين محلولة

مفاهيم أساسية

- **جدولة CPU:** هي أساس نظام التشغيل المتعدد البرمجة.
- في النظام وحيد المعالج: يمكن لإجرائية واحدة فقط أن تُنفَّذ في الوقت نفسه.
- **الهدف: منع CPU أن تصبح (idle)؛** الاستفادة بشكل أعظمي من استخدام CPU من خلال تعدد البرمجة. (تنفيذ إجرائية أخرى موجودة ضمن الذاكرة عندما تنتقل الإجرائية من حالة تشغيل إلى حالة انتظار ..).
- **CPU Burst:** مقدار الوقت الذي تستغرقه العملية في استخدام المعالج قبل أن تصبح غير جاهزة (ينتظر المعالج I/O ليكمل التنفيذ).
- يبدأ تنفيذ العملية مع CPU burst يتبعها I/O burst وهكذا بالتناوب .. حتى يطلب النظام إنهاء تنفيذ العملية.
- يقوم **المجدول قصير الأمد** باختيار إحدى العمليات الموجودة ضمن رتل الجاهزية ويخصّص المعالج لها؛ ليتم تنفيذها .

مجدول وحدة المعالجة المركزية (CPU SCHEDULER)

- يختار عملية من بين العمليات الجاهزة للتنفيذ في الذاكرة (بالاعتماد على خوارزمية الجدولة)، ويخصّص CPU لهذه العملية. يوجد نوعان: شفعية، وغير شفعية.
- **الجدولة غير الشفعية (non-preemptive):** عندما تُحجز CPU لعملية ما؛ تحتفظ العملية بالمعالج حتى انتهاء تنفيذها. أما **الشفعية**: تُجبر العملية على المغادرة في أي وقت.
- قد تحدث قرارات الجدولة في إحدى حالات العملية التالية:
 - ✓ ١. التبديل من حالة التشغيل إلى حالة الانتظار
 - ✓ ٢. التبديل من حالة التشغيل إلى حالة الجاهزية
 - ✓ ٣. التبديل من الانتظار إلى الجاهزية
 - ✓ ٤. انتهاء تنفيذ عملية



المرسل DISPATCH

- تمنح **وحدة المرسل** التحكم في وحدة المعالجة المركزية للعملية التي تم اختيارها من قبل المجدول قصير الأمد؛ ويجب أن يكون سريعاً قدر الإمكان.
- تتضمن وظيفته:
 - تبديل السياق
 - الانتقال إلى وضع المستخدم
 - القفز إلى الموقع الصحيح في برنامج المستخدم لإعادة تشغيل هذا البرنامج
- تأخير الإرسال (dispatch latency)
 - هو الوقت الذي يستغرقه المرسل لإيقاف عملية أولى وبدء تشغيل عملية أخرى.

مقياس الجدولة CRITERIA

- (١) استخدام وحدة المعالجة المركزية : يجب أن تكون وحدة المعالجة المركزية مشغولة قدر الإمكان (ليست في حالة idle).
 - (٢) الإنتاجية (throughput): عدد العمليات التي تكمل تنفيذها في كل وحدة زمنية.
 - (٣) وقت (Turnaround): مقدار الوقت اللازم لتنفيذ إجرائية. (من لحظة وصولها وحتى انتهاء التنفيذ).
 - (٤) وقت الانتظار (waiting): مقدار الوقت الذي تنتظره العملية في رتل الانتظار (مجموع لحظات انتظار العملية خلال تنفيذها).
 - (٥) وقت الاستجابة (Response): الوقت المستغرق من لحظة تقديم الطلب حتى إنتاج الاستجابة الأولى ، وليس الناتج النهائي (في بيئة مشاركة الوقت).
- معايير أمثلة خوارزمية الجدولة:

Max CPU Utilization ❖

Max Throughput ❖

Min turnaround Time ❖

Min Response Time ❖

Min waiting time ❖

١- جدولۀ FIRST- COME, FIRST-SERVED (FCFS)

Process Burst Time

P_1 24

P_2 3

P_3 3

○ نفترض أنّ العمليات تصل بالترتيب P_1, P_2, P_3 (تصل جميعها عند اللحظة 0).

○ مخطط غانت:



Waiting time		
P1	P2	P3
0	24	27
$(0+24+27)/3 = 17$		

Turnaround time		
P1	P2	P3
24	27	30
$(24+27+30)/3 = 27$		

Response time		
P1	P2	P3
0	24	27
$(0+24+27)/3 = 17$		

على فرض وصول العمليات بالترتيب P_2, P_3, P_1 ، أعد حساب القيم السابقة.

٢- جدولة SHORTEST-JOB-FIRST (SJF)

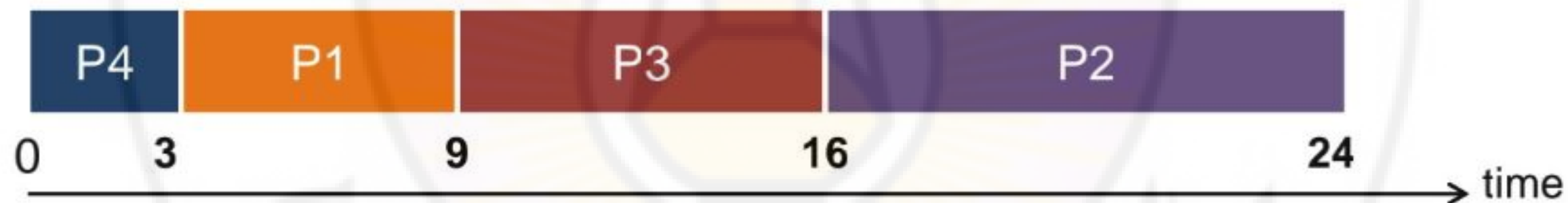
- يتم ربط كل عملية مع طول رشقة CPU التالية لهذه العملية.
- عندما تكون CPU متوفرة، يتم اختيار العملية التي تملك **الرشقة التالية الأقل**.
- عند تساوي طولي رشقتين: تستخدم جدولة FCFS.
- الأصحّ تسميتها: خوارزمية أقصر رشقة CPU تالية.
- يوجد منها نوعان:
 - non-preemptive.
 - SRTF (shortest-remaining-time-first) preemptive.

٢- جدولة SHORTEST-JOB-FIRST (SJF) النوع الأول (NON-PREEMPTIVE) - مثال أول

Process Burst Time

P_1 6
 P_2 8
 P_3 7
 P_4 3

○ نفترض أن العمليات تصل جميعها في اللحظة 0 نفسها.



Waiting time			
P1	P2	P3	P4
3	16	9	0
$(3+16+9+0)/4 = 7$			

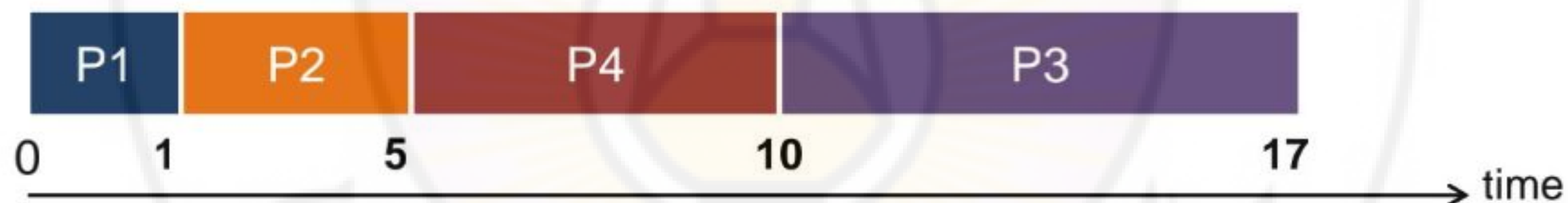
Turnaround time			
P1	P2	P3	P4
9	24	16	3
$(9+24+16+3)/4 = 13$			

٢- جدولة SHORTEST-JOB-FIRST (SJF)

النوع الأول (NON-PREEMPTIVE) مثال ثاني مع أوقات وصول مختلفة

Process	Arrival Time	Burst Time
P ₁	0	1
P ₂	1	4
P ₃	2	7
P ₄	3	5

○ مخطط غانت.



Waiting time			
P1	P2	P3	P4
0	0	8	2
$(0+0+8+2)/4 = 2.5$			

Turnaround time			
P1	P2	P3	P4
1	4	15	7
$(1+4+15+7)/4 = 6.75$			

٢- جدولة SHORTEST-JOB-FIRST (SJF)

النوع الثاني (PREEMPTIVE) SHORTEST-REMAINING-TIME-FIRST

Process	Arrival Time	Burst Time
P ₁	0	8
P ₂	1	4
P ₃	2	9
P ₄	3	5

○ مخطط غانت. (تستمر المقارنة حتى تكون قد وصلت جميع العمليات).



Waiting time			
P1	P2	P3	P4
9	0	15	2
$(9+0+15+2)/4 = 6.5$			

Turnaround time			
P1	P2	P3	P4
17	4	24	7
$(17+4+24+7)/4 = 13$			

Response time			
P1	P2	P3	P4
0	0	15	2
$(0+0+15+2)/4 = 4.25$			

٣- جدولة وفق الأولوية PRIORITY

- يتم إسناد رقم أولوية (عدد صحيح) لكل عملية.
- يتم تخصيص CPU إلى العملية ذات الأولوية الأعلى (تملك الرقم الأقل).
- عند تساوي رقم الأولوية بين عمليتين: تتم الجدولة وفق FCFS.
- يمكن أن تكون شفعية أو لا شفعية.
- SJF هي حالة خاصة من الجدولة وفق الأولوية: حيث يمثل رقم الأولوية مقلوب وقت Burst التالي المتوقع.
- تعاني من مشكلة Starvation: من الممكن ألا تنفذ العمليات ذات الأولوية الأقل.
- الحل من خلال Aging:
 - زيادة بشكل تدريجي لأولوية العمليات التي تنتظر لمدة طويلة.
 - مثال: إذا كانت الأولويات تتراوح بين 0 (عالية) إلى 127 (منخفضة)، يمكن زيادة الأولوية بمقدار 1 كل 15 دقيقة للعمليات المنتظرة

٣- مثال على جدولة PRIORITY

Process	Burst Time	Priority
P ₁	10	3
P ₂	1	1
P ₃	2	4
P ₄	1	5
P ₅	5	2

○ نفترض أن العمليات تصل جميعها في اللحظة 0.



Waiting time				
P1	P2	P3	P4	P5
6	0	16	18	1
$(0+1+6+16+18)/5=8.2$				

Turnaround time				
P1	P2	P3	P4	P5
16	1	18	19	6
$(16+1+18+19+6)/5 = 12$				

٤- الجدولة الدائرية (RR) ROUND-ROBIN

- تحصل كل عملية على حصّة زمنية صغيرة من وقت CPU تسمى **time slice** شريحة أو **حصّة زمنية**، تكون عادةً من ١٠ إلى ١٠٠ مللي ثانية.
- يتم التعامل ضمن رتل الجاهزية وفق النمط **FIFO**، حيث تضاف الإجراءات الجديدة إلى نهاية الرتل.
- يقوم المجدول بالتقاط أول إجرائية في الرتل، ويعدّ المؤقت للمقاطعة بعد حصّة زمنية واحدة، ثمّ ينفذ الإجرائية.
- إذا كانت رشقة CPU للإجرائية أقصر من حصّة زمنية واحدة: تتخلى الإجرائية طوعاً عن CPU، وينتقل المجدول إلى الإجرائية التالية في الرتل.
- إذا كانت رشقة CPU للإجرائية أطول من حصّة واحدة؛ يبدّل السياق مع الإجرائية التالية في الرتل؛ وتوضع الإجرائية التي تمت مقاطعتها في نهاية الرتل. (**الخوارزمية شفعيّة**).
- إذا كان لدينا n إجرائية في الرتل؛ والحصّة الزمنية q ، لن تنتظر كل إجرائية أكثر من:

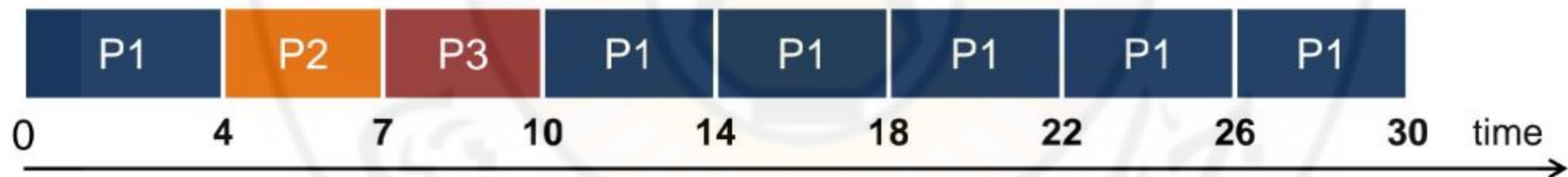
$(n-1)*q$ وحدة زمنية إلى حين الحصّة التالية

٤- مثال على جدولة RR

Process	Burst Time
P ₁	24
P ₂	3
P ₃	3

نفترض أنّ العمليات تصل جميعها في اللحظة 0.

$q = 4 \text{ ms}$



Waiting time		
P1	P2	P3
6	4	7
$(6+4+7)/3 = 5.667$		

Turnaround time		
P1	P2	P3
30	7	10
$(30+7+10)/3 = 15.667$		

Response time		
P1	P2	P3
0	4	7
$(0+4+7)/3 = 3.667$		

٥- الجدولة وفق أرتال متعددة المستويات

- يتم تجزئة رتل الانتظار إلى رتلين منفصلين:
 - أمامي (تفاعلي).
 - خلفي (دفعي).
- يملك كل رتل خوارزمية جدولة خاصة به. مثال:
 - الأمامي: خوارزمية RR.
 - الخلفي: خوارزمية FCFS.
- **يجب تنفيذ الجدولة بين الأرتال:**
- جدولة الأولوية الثابتة (يتم تخديم الجميع من الرتل الأمامي، من ثم تخديم عمليات الرتل الخلفي) .. يمكن أن يسبب مجاعة starvation.
- يحصل كل رتل على قيمة محدّدة من وقت CPU، والتي يمكن أن يستخدمها كل رتل للجدولة بين عملياته. 80% للرتل الأمامي في حالة RR.
- يتم تخصيص 20% للرتل الخلفي في FCFS.

٥- الجدولة وفق أرتال متعدد المستويات



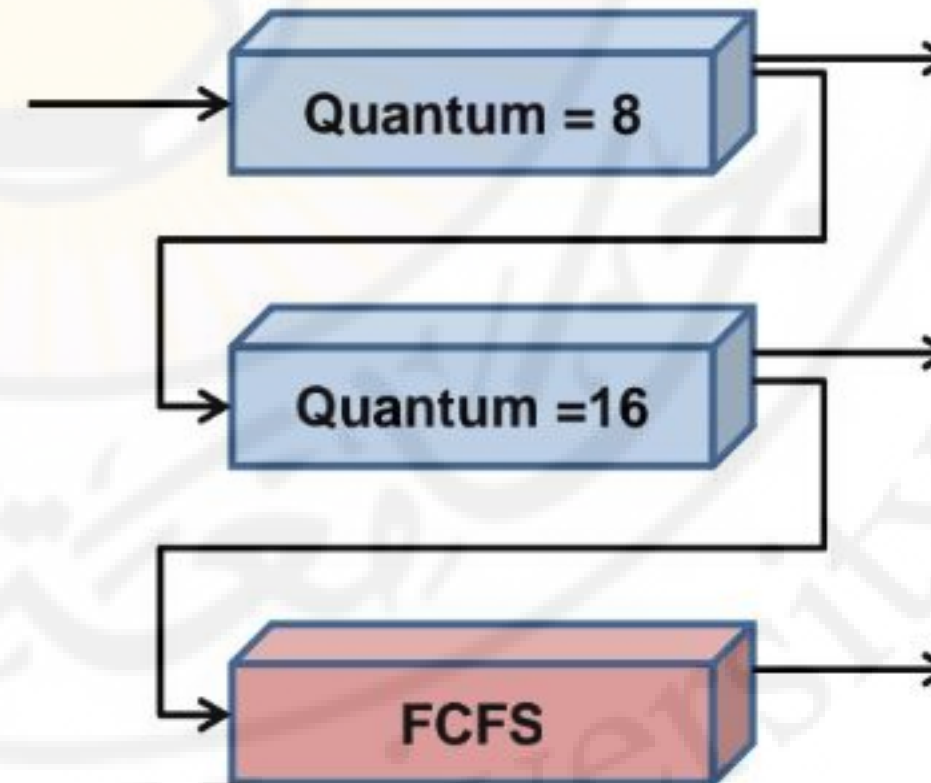
الجدولة وفق أرتال بتغذية راجعة
مع ثلاثة مستويات:

Q0: جدولة RR مع $q=8$.

Q1: جدولة RR مع $q=16$.

Q2: جدولة FCFS.

عندما لا ينتهي العمل في المستوى الأول؛
يتم نقله إلى المستوى الثاني، وإذا لم ينته يتم
نقله إلى الثالث.



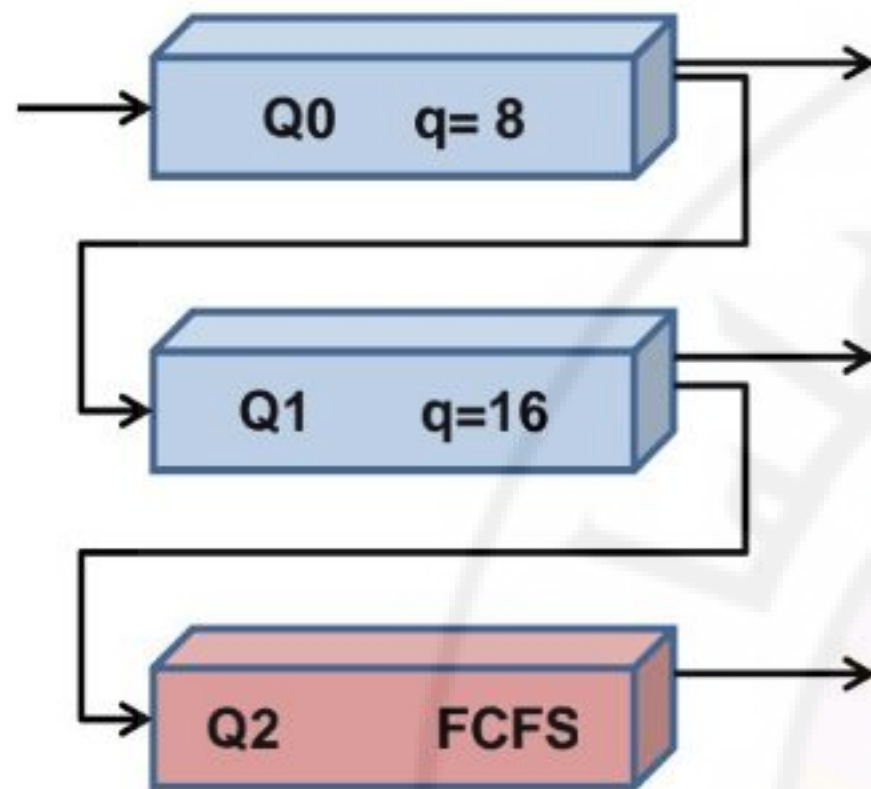
الجدولة وفق أرتال متعددة المستويات بتغذية راجعة
في نظام متعدد المعالجات (أو معالج متعدد النوى)

يتم التنفيذ على التوازي

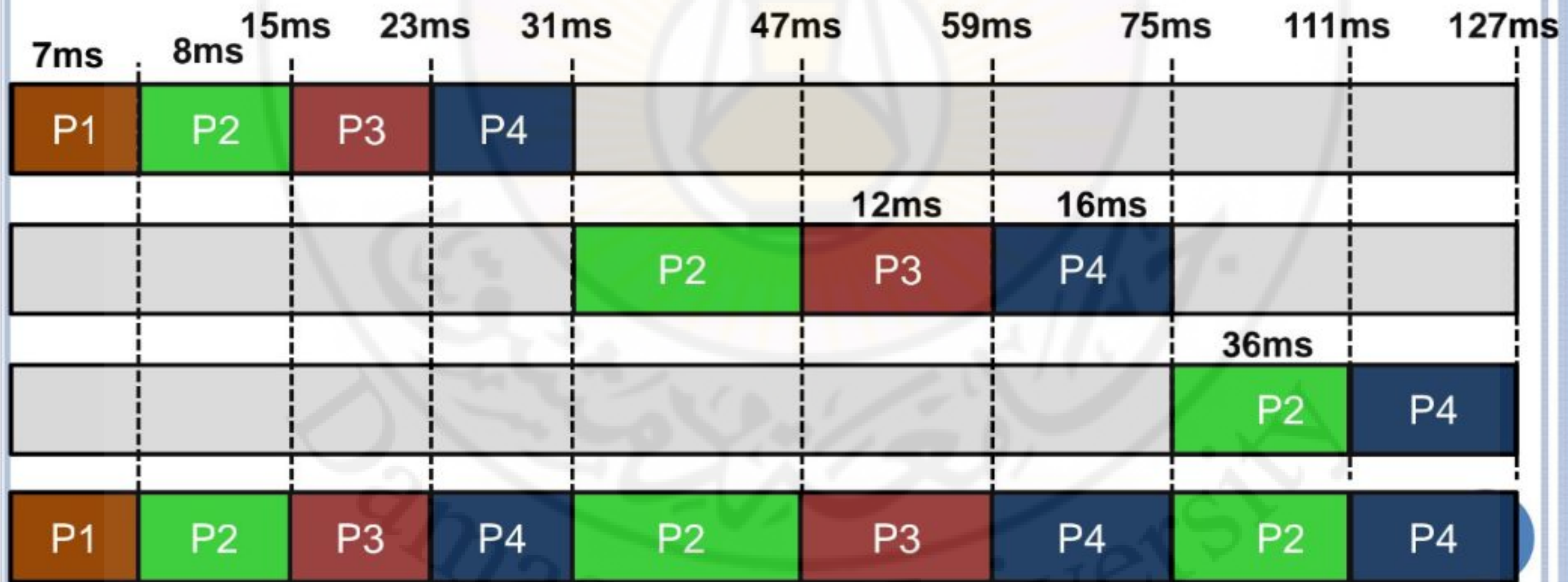
Process	Arrival Time	Burst Time
P_1	0	7
P_2	1	60 52 44 36
P_3	2	20 12
P_4	3	40 32 16



الجدولة وفق أرتال متعددة المستويات بتغذية راجعة
 باستخدام معالج وحيد النواة (لا يوجد توازي في التنفيذ)



Process	Arrival Time	Burst Time
P ₁	0	7
P ₂	1	60 52 44 36
P ₃	2	20 12
P ₄	3	40 32 16



مسألة على خوارزميات الجدولة

- Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5 , all at time 0.

- a.** Draw four **Gantt charts** illustrating the execution of these processes using FCFS, SJF, a non-preemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 1) scheduling.
- b.** What is the **turnaround time** of each process for each of the scheduling algorithms in **part a**?
- c.** What is the **waiting time** of each process for each of the scheduling algorithms in **part a**?
- d.** Which of the schedules in part a results in the minimal average waiting time (over all processes)?

A. GANTT CHART



<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2



	FCFS	RR	SJF	Priority
P_1	10	19	19	16
P_2	11	2	1	1
P_3	13	7	4	18
P_4	14	4	2	19
P_5	19	14	9	6

b. ☐

	FCFS	RR	SJF	Priority
P_1	0	9	9	6
P_2	10	1	0	0
P_3	11	5	2	16
P_4	13	3	1	18
P_5	14	9	4	1

c. ☐

Shortest Job First d. ☐



السنة الرابعة

نظم تشغيل 1



المحاضرة السادسة

Synchronization

د. المثني خضر

كلية الهندسة المعلوماتية

المحاضرة السابعة: المفردات الرئيسية

- ❖ مفهوم التوازي Concurrency
- ❖ الاتصال بين العمليات
- ❖ تنافس العمليات (process competition)
- ❖ حالة السباق (race condition)
- ❖ مشكلة المقطع الحرج
- ❖ نماذج الحلول لمشكلة المقطع الحرج
- ❖ مشكلة المنتج والمستهلك (producer-consumer)
- ❖ حل مشكلة المنتج والمستهلك (برمجياً)
- ❖ الأقفال Mutex (Mutual Exclusion) Locks
- ❖ حل مشكلة المنتج والمستهلك (باستخدام الأقفال)
- ❖ السيمافور Semaphore
- ❖ حل مشكلة المنتج والمستهلك (باستخدام السيمافور)
- ❖ مشكلة القارئ-الكاتب Reading and writing problem
- ❖ مشكلة الفلاسفة الطاعمين Dining philosophers problem

مفهوم التوازي Concurrency

- نقصد بالتوازي تنفيذ أكثر من عملية في وقت واحد.
- قد تكون العمليات المتوازية مستقلة عن بعضها البعض (**independent processes**)
- أو قد تكون متعاونة مع بعضها البعض (**cooperative processes**)

العملية المستقلة هي التي لا تؤثر ولا تتأثر بالعمليات التي تعمل معها في نفس الوقت.
العملية المتعاونة هي العملية التي تؤثر و تتأثر بالعمليات الأخرى التي تعمل معها بالتوازي.

- 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 (concurrent threads)

الاتصال بين العمليات

□ يتطلب عمل عدة عمليات سوية تحقيق اتصالات بين العمليات (inter-process communication IPC): وهي الطريقة التي تسمح للعمليات بتبادل البيانات بالإضافة إلى معلومات مختلفة.

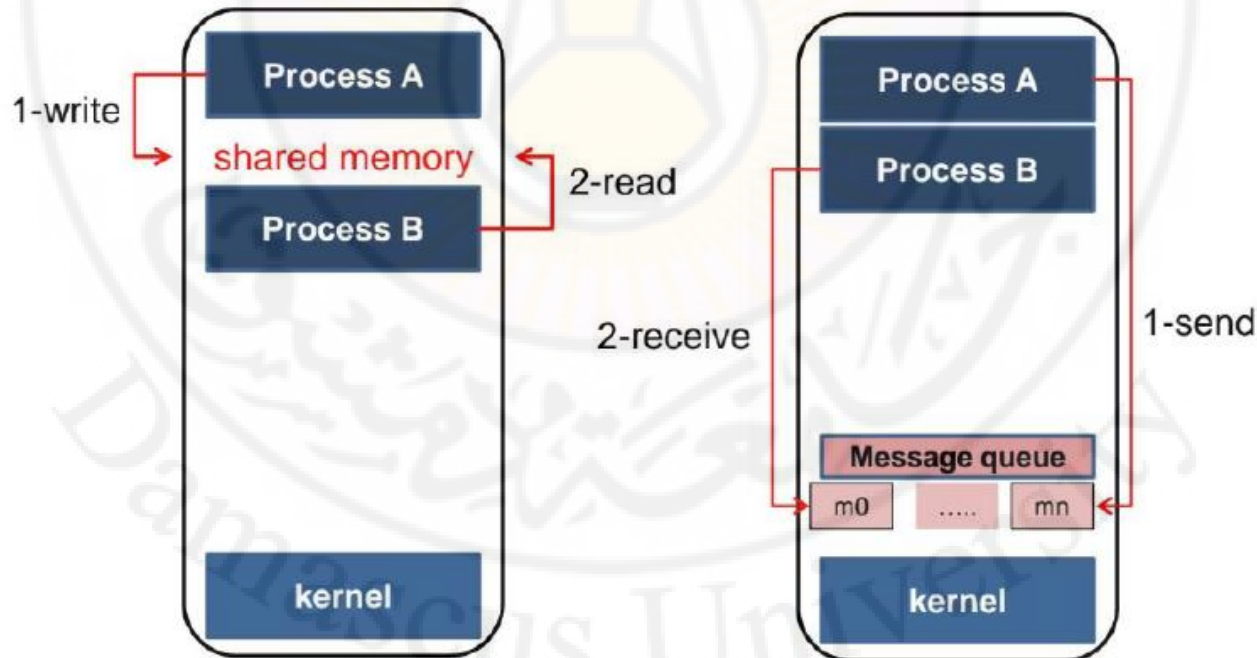
□ يوجد نمطين أساسيين للاتصالات بين العمليات:
✓ **الذاكرة المشتركة (Shared Memory):** يتم إنشاء منطقة الذاكرة التي يتم تشاركها بين العمليات المتعاونة. يمكن للعمليات تبادل المعلومات من خلال قراءة وكتابة البيانات إلى منطقة الذاكرة المشتركة.

✓ **تمرير الرسائل (Message Passing):** يتم الاتصال عن طريق الرسائل المتبادلة بين العمليات المتعاونة. لإرسال الرسالة، `send(msg)` ولإستلام الرسالة، `recv(&buffer)`.

الاتصال بين العمليات

• لماذا تحتاج العمليات المتوازية التعاون فيما بينها ؟

1. طلب عملية لخدمة من عملية أخرى: في هذه الحالة على العملية التي طلبت الخدمة انتظار العملية التي طلب منها الخدمة حتى تفرغ من أدائها.
2. زيادة سرعة التنفيذ: وذلك بتقسيم مهمة واحدة كبيرة على عدة عمليات يتم تنفيذها بالتوازي.
3. قد يكون من الملائم للعمليات أن تعمل مع بعضها لانجاز العمل، مثلا في نظام الوسائط المتعددة



(a) shared memory.

(b) Message passing.

تنافس العمليات (process competition)

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

حالة السباق (race condition)

- `counter++` could be implemented as (producer)

```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

- `counter--` could be implemented as (consumer)

```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

```
register1 = counter (5)  
register1 = register1 + 1 (6)  
register2 = counter (5)  
register2 = register2 - 1 (4)  
counter = register1 (6)  
counter = register2 (4)
```

عدم اتساق البيانات نتيجة التداخل

- Consider this execution interleaving with "**counter = 5**" initially:

```
S0: producer execute register1 = counter {register1 = 5}  
S1: producer execute register1 = register1 + 1 {register1 = 6}  
S2: consumer execute register2 = counter {register2 = 5}  
S3: consumer execute register2 = register2 - 1 {register2 = 4}  
S4: producer execute counter = register1 {counter = 6}  
S5: consumer execute counter = register2 {counter = 4}
```

Critical section المقطع الحرج

جزء من الكود يمكن الوصول إليه من خلال عملية واحدة فقط في الوقت نفسه.
مقطع الكود الذي يتم فيه النفاذ إلى متغيرات مشتركة.

مشكلة المقطع الحرج (critical section)

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

1. قراءة قيمة المتغير x (read x)

2. زيادة قيمة المتغير بواحد ($x=x+1$)

3. 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.

ستكون النتيجة النهائية بعد تنفيذ العمليتان أن المتغير **سيزيد بواحد بدلا من 2**.

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

المنطقة التي تتعامل مع المتغير المشترك في العملية **نسميها المقطع الحرج (critical section)**

ولحل مشكلة حالة السباق يجب أن نتأكد من أن هناك عملية واحدة تنفذ داخل المنطقة الحرجة. ولا يسمح للعملية الثانية بالدخول إلى المنطقة الحرجة إلا بعد خروج العملية الأولى منها.

حلّ مشكلة المقطع الحرج: أي حلّ يجب أن يحقق الشروط التالية

□ **الاستبعاد المتبادل (Mutual Exclusion):** إذا تم تنفيذ العملية P_i في مقطعها الحرج، فلا يمكن تنفيذ أية عمليات أخرى في مقاطعها الحرجة.

□ **التقدّم (Progress):** إذا لم يتم تنفيذ أية عملية في مقطعها الحرج وكانت هناك بعض العمليات التي ترغب في الدخول إلى مقطعها الحرج، فلا يمكن تأجيل اختيار العمليات التي ستدخل المقطع الحرج بعد ذلك إلى أجل غير مسمى

□ **الانتظار المحدود (bounded waiting):** يجب أن يكون هناك ارتباط بعدد المرات التي يُسمح فيها للعمليات الأخرى بدخول مقاطعها الحرجة بعد أن تقدّم العملية طلباً للدخول إلى مقطعها الحرج وقبل منح هذا الطلب.

نماذج الحلول لمشكلة المقطع الحرج

❑ Software solutions حلول برمجية

✓ خوارزميات لا تعتمد صحتها على أية افتراضيات أخرى.

✓ Peterson's solution (بين عمليتين فقط)

❑ Operation System solutions حلول نظام التشغيل

✓ تزود بعض المهام وبنى البيانات إلى المبرمج.

❑ Hardware solutions حلول عتادية

✓ تعتمد على بعض تعليمات الآلة (دقيقة)، مثال عنها: `test_and_set()`

✓ حماية المقاطع الحرجة باستخدام الأقفال locks (إعطاء قفل للعملية التي تدخل المقطع الحرج، وتحريره بعد خروجها منه).

✓ أقفال Mutex.

✓ السيمافورات Semaphores.

سنقوم بدراسة طريقتي الأقفال والسيمافورات بالتفصيل
و سنستخدمهم لحل مشاكل التزامن التقليدية المشهورة

مشكلة المنتج والمستهلك (producer-consumer)

- مشكلة المنتج والمستهلك أحيانا تسمى مشكلة الخازن المحدود (bounded-buffer).
- يكون لدينا عمليتان تستخدمان خازن مشترك، العملية الأولى تضع به بيانات بينما تأخذ العملية الأخرى البيانات منه.
- على 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.



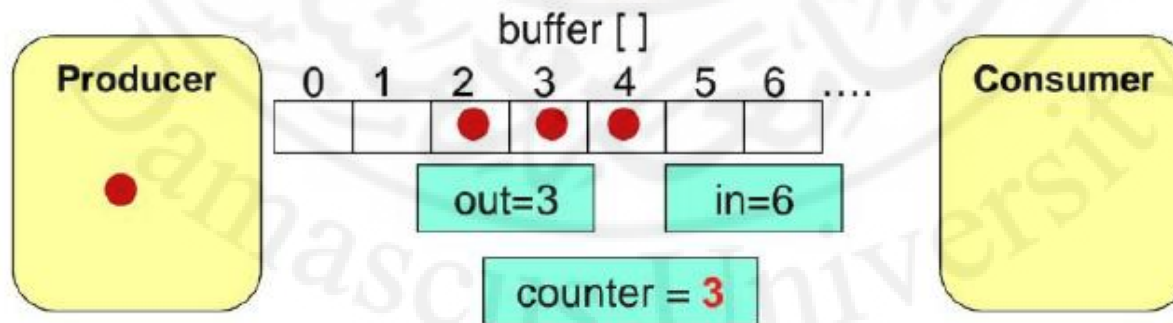
حل مشكلة المنتج والمستهلك (برمجياً)

- لنعرف بداية المعطيات المشتركة للمشكلة
- لدينا مخزن سعته 7 عناصر نضع فيه items من نوع data

```
#define BUFFER_SIZE 7
typedef struct {
    DATA data;
} item;

item buffer[BUFFER_SIZE];

int in = 0;          // Location of next input to buffer
int out = 0;         // Location of next removal from buffer
int counter = 0;     // Number of buffers currently full
```



حل مشكلة المنتج والمستهلك (برمجياً)

- يمكن حل هذه المشكلة بإجبار عملية المنتج على التوقف عن إحضار البيانات للخازن إذا كان ممتلئاً (sleep) (أي حلقة While تدور ولا تفعل أي شيء) وعندما يفرغ مكان في الخازن تقوم عملية المستهلك من إيقاظ المنتج ليبدأ في تعبئة الخازن (أي تنكسر حلقة While و يبدأ المنتج بالإنتاج).
- بنفس الطريقة تتوقف عملية المستهلك sleep إذا كان الخازن فارغاً، وعندما يحضر المنتج بيانات للخازن، يقوم بإيقاظها لتبدأ في أخذ البيانات من الخازن.

إجرائية المنتج

```
item    nextProduced;

PRODUCER

while (TRUE) {
    while (counter == BUFFER_SIZE);
    buffer[in] = nextProduced;
    in = (in + 1)%BUFFER_SIZE;
    counter++;
}
```

إجرائية المستهلك

```
item    nextConsumed;

CONSUMER

while (TRUE) {
    while (counter == 0);
    nextConsumed = buffer[out];
    out = (out + 1)%BUFFER_SIZE;
    counter--;
}
```

حل مشكلة المنتج والمستهلك (برمجياً)

- يمثل counter عدد العناصر الموجودة بالخازن، و المتغير buffer يمثل الخازن.
- ستختبر إجرائية المنتج الخازن، هل هو ممتلئ أم لا ؟ وذلك بالأمر:

while (counter == BUFFER_SIZE);

- إذا كانت المساواة صحيحة سيتوقف المنتج عن تعبئة الخازن ويذهب لينام (حلقة While لا نهائية ولا تفعل شيء) ولن يرجع لعمله ما لم يوقفه المستهلك.
- أما إذا كانت عملية المساواة غير صحيحة، أي أن الخازن غير ممتلئ ، فسيواصل المنتج في إحضار عناصر للخازن بالأمر:

Buffer[in] = nextProduced;

- ثم يزيد المتغير in بمقدار واحد ليحدد مكان المنتج القادم
- in = (in + 1) % BUFFER_SIZE;**
- ثم يزيد counter بواحد ليدل على زيادة عدد العناصر الحالية في المخزن:
- counter ++ ;**

تنفيذ الأمر ++ Counter يعتبر بمثابة عملية إيقاظ للمستهلك في حال كان في حالة sleep , لأنها تكسر حلقة While الموجودة لدى المستهلك .

حل مشكلة المنتج والمستهلك (برمجياً)

- أما بالنسبة لإجرائية المستهلك ستختبر الخازن، هل هو فارغ أم لا ؟ وذلك بالأمر:

while (counter == 0);

- إذا كانت المساواة صحيحة سيتوقف عن أخذ البيانات من الخازن ويذهب لينام ولن يرجع لعمله ما لم يوقظه المنتج (عندما يحضر بيانات للخازن).
- أما إذا كانت عملية المساواة غير صحيحة، أي أن الخازن غير فارغ، فسيواصل المستهلك عمله بأخذ عنصر كل مرة من الخازن بالأمر:

nextConcumed = Buffer[out] ;

- ثم يزيد المتغير out بمقدار واحد ليحدد مكان السحب القادم

out = (out + 1) %BUFFER_SIZE;

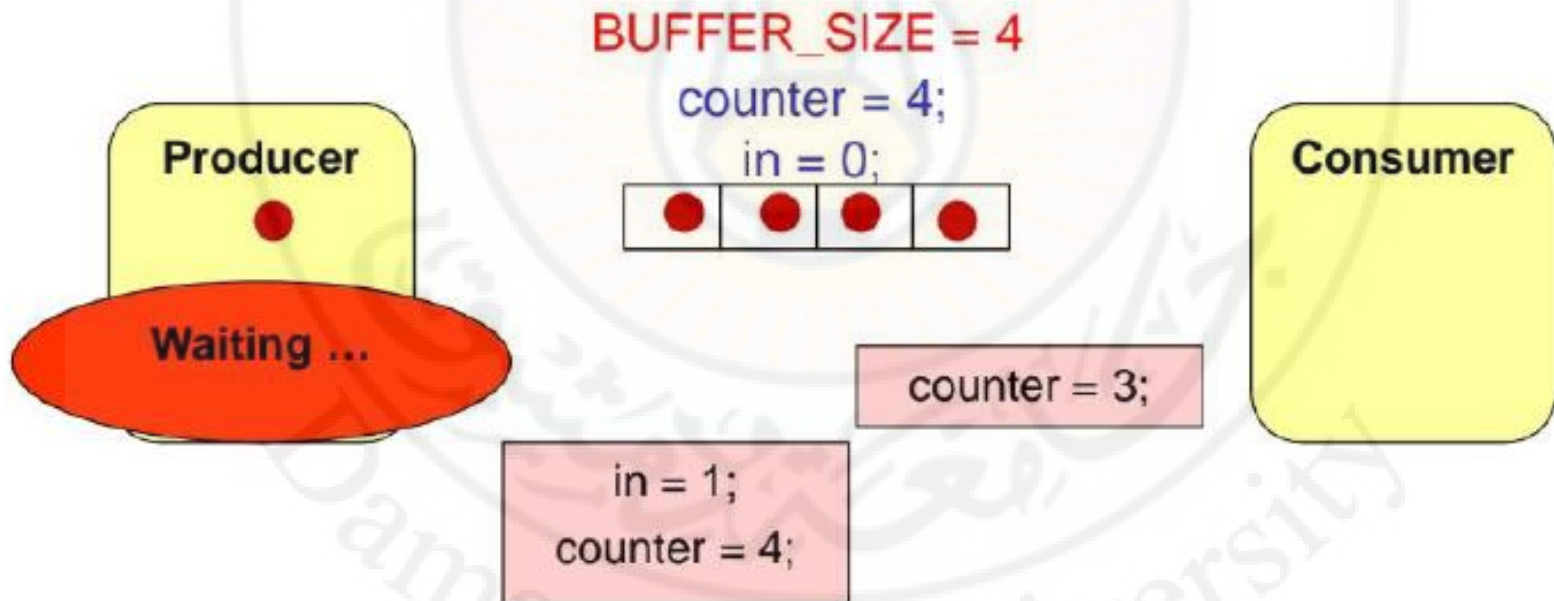
- ثم ينقص counter بواحد ليعدل على نقصان عدد العناصر الحالية في المخزن:

counter -- ;

تنفيذ الأمر Counter -- يعتبر بمثابة عملية إيقاف للمنتج في حال كان في حالة sleep , لأنها تكسر حلقة While الموجودة لدى المنتج.

حل مشكلة المنتج والمستهلك (برمجياً)

- هذا الحل البرمجي لمشكلة المنتج مستهلك يعاني من مشكلة المقطع الحرج, لأن كلاً من إجرائتي المنتج والمستهلك تشتركان بالمتغير `counter`.
- من الممكن حصول حالة السباق (race condition) و بالتالي الوصول إلى نتائج خاطئة



الأقفال Mutex (Mutual Exclusion) Locks

- ❑ تعدّ الحلول السابقة معقّدة وغير قابلة للنفاذ بشكل عام من قبل مبرمجي التطبيق.
- ❑ يبني مصمّمو نظام التشغيل أدوات برمجية لحلّ مشكلة المقطع الحرج, من أبسطها أقفال Mutex.
- ❑ تتم حماية المقطع الحرج مع القفل باستخدام التابع **acquire()** ومن ثم تحريره باستخدام التابع **release()**.
- ✓ يتم استخدام متغيّر بولياني يحدّد فيما إذا كان القفل متوفراً أم لا.
- ✓ يجب أن يكون استدعاء التابعين دقيقاً (atomic). (غير قابلة للمقاطعة)
- ✓ يتم تحقيقه عادةً باستخدام تعليمات عتادية دقيقة.
- ❑ المشكلة: يتطلّب هذا الحلّ الانتظار المشغول (busy waiting)

الأقفال Mutex (Mutual Exclusion) Locks

- القيمة الابتدائية للمتحول available هي true
- عندما available=true يعني أنه يمكننا الدخول إلى المقطع الحرج والتنفيذ
- عندما available=false يعني أنه يوجد قفل وممنوع الدخول أو التنفيذ

```
acquire() {  
    while (!available)  
        ; /* busy wait */  
    available = false;  
}  
release() {  
    available = true;  
}
```

طالما أن available = false فإن نفيها يكون true وتبقى العملية منتظرة عند الحلقة

A process that attempts to acquire an unavailable lock is **blocked** until the lock is released.

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```

busy waiting: While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to `acquire()`.

حل مشكلة المنتج والمستهلك (باستخدام الأقفال)

- القيمة الابتدائية للمتحول **available** هي **true**
- عندما **available=true** يعني أنه يمكننا الدخول إلى المقطع الحرج والتنفيذ
- عندما **available=false** يعني أنه يوجد قفل وممنوع الدخول أو التنفيذ

إجرائية المنتج

```
item nextProduced;

while (TRUE) {

    while (counter == BUFFER_SIZE);
    acquire();
    Buffer[in] = nextProduced;
    in = ( in + 1 ) %BUFFER_SIZE;
    counter ++ ;
    release();

}
```

إجرائية المستهلك

```
item nextConsumed;

while (TRUE) {

    while (counter == 0);
    acquire();
    nextConsumed = Buffer[out] ;
    out = ( out + 1 ) %BUFFER_SIZE;
    counter -- ;
    release();

}
```

السيمافور Semaphore

- السيمافور هو علامة كانت تستخدم قديما لتنظيم مرور القاطرات، حيث سيكون للسيمافور حالتين، إما تحت أو وفوق، عندما تصل قاطرة وتريد الدخول للمحطة سينظر السائق للسيمافور فإذا كانت وضعيته تحت فهذا يعني أن الطريق سالكة (تشبه إشارة المرور الخضراء)، ولأن خط السكة حديد لا يحتمل تلاقي قاطرتين، فسيرفع السيمافور إلى أعلى بعد دخول القاطرة بحيث لا يسمح لقاطرة أخرى أن تدخل (الخط مشغول) ويشبه هذا إشارة المرور الحمراء.
- بنفس المفهوم سنستخدم متغير يمثل السيمافور **(عدد صحيح يحتمل قيمتين)**، حيث تمثل قيمة علامة تحت وقيمة أخرى علامة فوق. ثم إذا بدأت عملية استخدام بيانات مشتركة ستضع القيمة التي تمثل علامة فوق داخل متغير السيمافور ، وإذا أرادت عملية أخرى استخدام البيانات المشتركة ستختبر السيمافور وبما أن علامته فوق فهذا يعني أنه لا يسمح لها باستخدام البيانات المشتركة الآن.
- يمكن استخدام السيمافور لحل مشكلة المقطع الحرج ومشكلة المنتج – المستهلك وغيرها .

السيمافور Semaphore

■ هي أداة تزامن لا تتطلب الانتظار المشغول.

■ أقل تعقيداً.

■ تمثل السيمافور عدداً صحيحاً.

■ Semaphore S : integer variable

■ يمكن النفاذ إلى السيمافور S وتعديل قيمتها باستخدام إحدى العمليتين الدقيقتين:

■ **wait()** and **signal()** Originally called **P()** and **V()**

• تعريف التعليمة **wait()** انتظر

```
wait (S) {  
    while (S == 0)  
        ; // busy wait - wait until s>0  
    S--;  
}
```

لا يمكنها تخفيض القيمة إلا إذا كانت السيمافور أكبر تماماً من صفر.

• تعريف التعليمة **signal()** أرسل

```
signal (S) {  
    S++;  
}
```

استخدام السيمافور Semaphore Usage

■ سيمافور عددية (counting): وهي عدد صحيح يمكن أن يأخذ أية قيمة ضمن مجال غير محدد.

■ سيمافور ثنائية (binary): عدد صحيح يأخذ إحدى القيمتين 0 أو 1، وبالتالي تصبح قفل mutex.

```
do {  
    wait(s);  
    // critical section  
    signal(s);  
    // remainder section  
} while(true);
```

■ نفترض لدينا عمليتين P1 و P2، والتي تطلب أن تحدث S1 قبل S2.

P1:

S₁;

signal(synch);

P2:

wait(synch);

S₂;

• لا يمكن لعمليتين تنفيذ wait() و signal() على السيمافور نفسها في الوقت نفسه.

تحقيق القفل **Mutex** باستخدام السيمافور

```
semaphore mutex = 1;  
int counter = 0 ;    // متغير مشترك
```

P1

P2

```
while(1){  
    wait(mutex);  
    counter++;  
    signal(mutex);  
    /* remainder */  
}
```

```
while(1){  
    wait(mutex);  
    counter--;  
    signal(mutex);  
    /* remainder */  
}
```

Critical section

Entry Section

Exit Section

تحقيق التنبيه Notification باستخدام السيمافور

Semaphore S1 = 1, S2 = 0 ;

Process 1

```
while(1){  
    // do something;  
    wait(S1);  
    cout << "1";  
    signal(S2);  
    // do something;  
}
```

Process 2

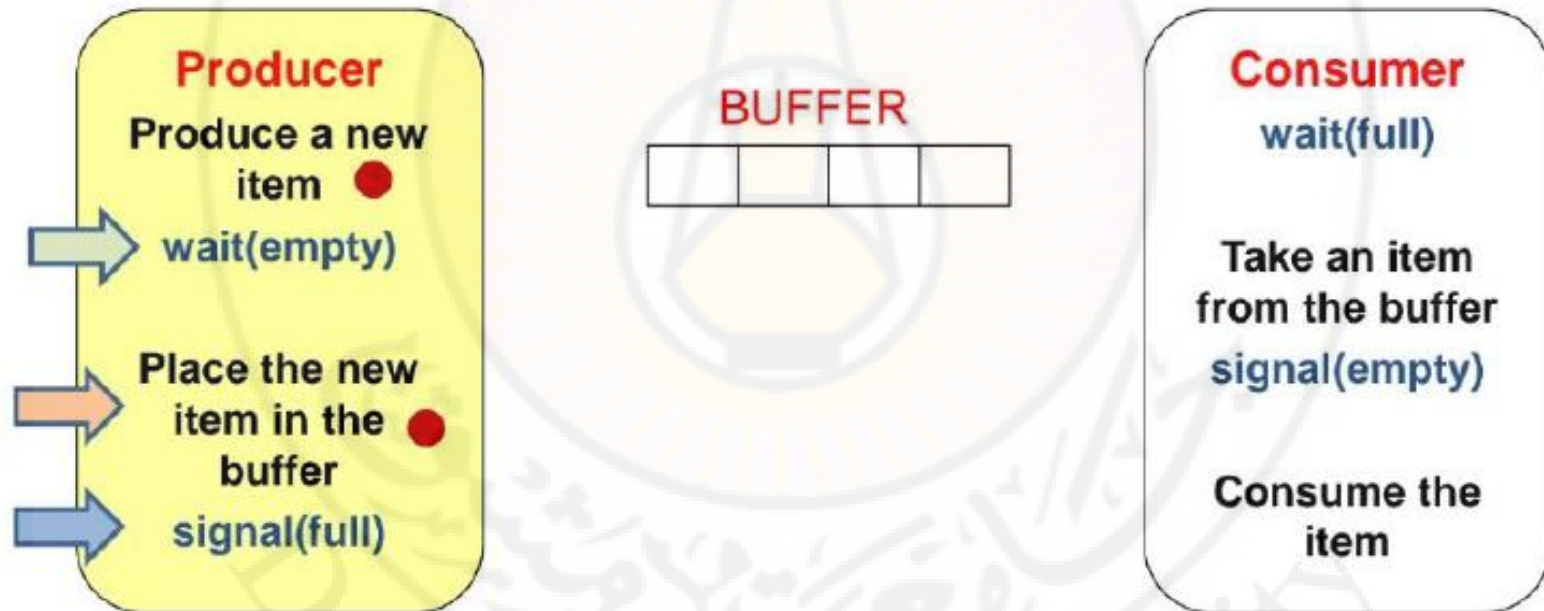
```
while(1){  
    // do something;  
    wait(S2);  
    cout << "2";  
    signal(S1);  
    // do something;  
}
```

- ❑ Process 1 uses signal(S2) to notify Process 2; indicating **"I am done, please go ahead"**.
- ❑ The output is 1 2 1 2 1 2 ..
- ❑ What if S1 = 0 and S2 = 1 ?

حل مشكلة المنتج والمستهلك (السيمافور)

PRODUCER

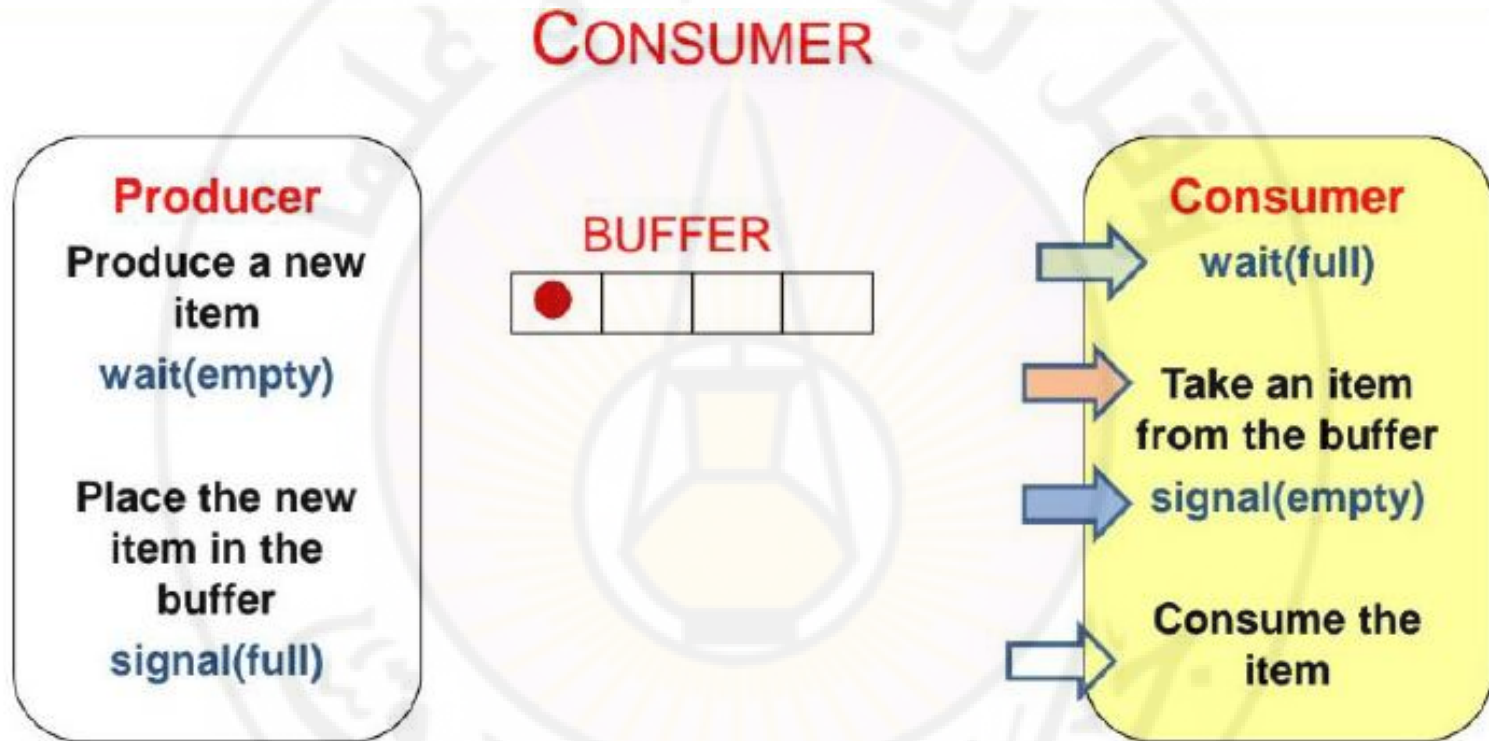
- نحتاج سيمافور لحماية المخزن عند إضافة عنصر وزيادة العداد (**mutex**).
- سيمافور ثانية، لقفل المنتجين عندما يكون المخزن ممتلئاً (**full**).
- سيمافور ثالثة، لقفل المستهلكين عندما يكون المخزن فارغاً (**empty**).



semaphore full 0 0 1

semaphore empty 4 3 3

حل مشكلة المنتج والمستهلك (السيمافور)



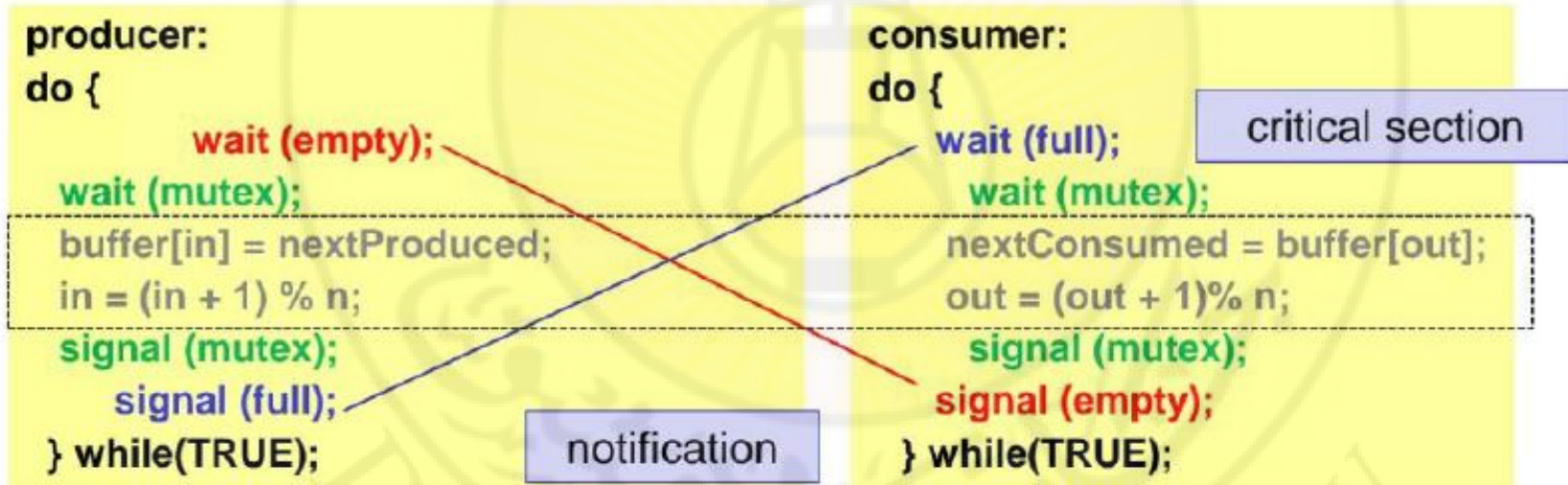
semaphore **full** 1 0 0 تعبر عن عدد العناصر في المخزن

semaphore **empty** 3 3 4 تعبر عن عدد الفراغات في المخزن

حل مشكلة المنتج والمستهلك (السيمافور)

Remember: a **wait decreases** its argument and a **signal increases** its argument.

BINARY_SEMAPHORE	mutex = 1;	// Can only be 0 or 1
COUNTING_SEMAPHORE	empty = n; full = 0;	// Can take on any integer value



مشكلة القارئ-الكاتب Reading and writing problem

- يكون لدينا مجموعتين من العمليات، القارئ والكاتب، تنفذ إلى مورد مشترك (ملف مثلاً) باتباع القواعد التالية:
 - ✓ ١- يمكن للقارئ أن تقرأ جميعها في الوقت نفسه.
 - ✓ ٢- يمكن للكاتب واحد فقط أن يكتب في الوقت نفسه (لا يمكن لقارئ أن يقرأ خلال ذلك).
 - ✓ ٣- عند وجود قارئ يقوم بقراءة المورد؛ يجب على جميع الكاتبات أن تنتظر.وبالتالي تملك القارئ أولوية أعلى.
- نحتاج سيمافور لقفل القارئ عندما يوجد كاتب في الكتابة على المورد المشترك، وقفل الكاتب عند وجود قارئ في حالة قراءة على المورد المشترك. السيمافور `rw_mutex`.
- عند التعديل على عدد القارئ (مقطع حرج): يتم استخدام سيمافور `mutex`.

حل مشكلة القارئ-الكاتب (باستخدام السيمافور)

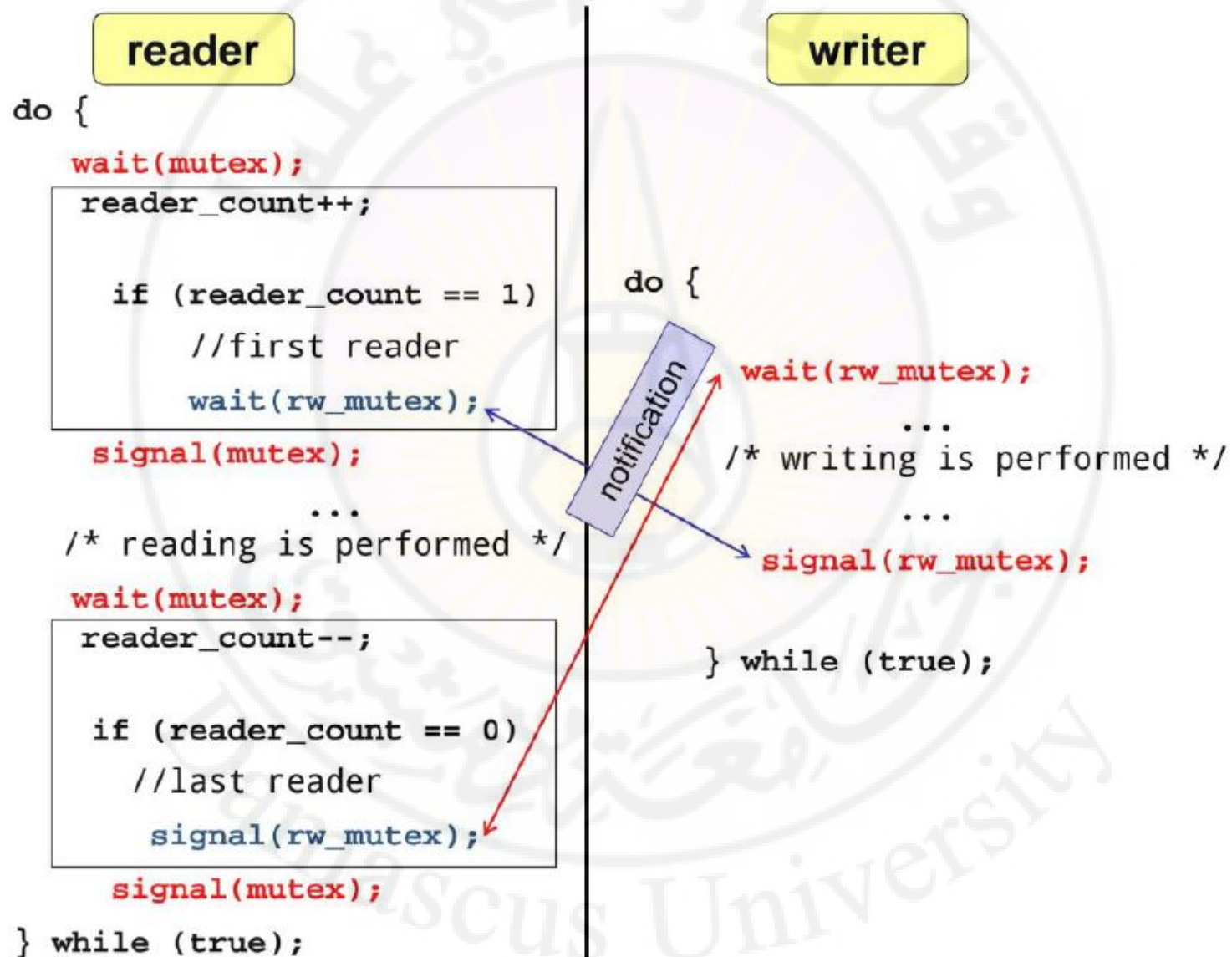
• المعطيات المشتركة للمشكلة :

■ Shared Data

- Data set // shared File
- Integer **reader_count** initialized to 0
- Semaphore **rw_mutex** initialized to 1
- Semaphore **mutex** initialized to 1

//shared between readers to modify reader_count

حل مشكلة القارئ-الكاتب (باستخدام السيمافور)



مشكلة الفلاسفة الطاعمين Dining philosophers problem

- هنالك 5 فلاسفة يجلسون على طاولة مستديرة وعلى كل فيلسوف أن يقوم بأحد أمرين، إما أن يأكل أو يفكر. فإذا أكل لا يفكر، وإذا فكر لا يأكل.
- وهنالك خمسة شوكة، شوكة بين كل فيلسوفين، بحيث يكون على يمين كل فيلسوف شوكة وعلى يساره شوكة. ويحتاج كل فيلسوف أن يستخدم الشوكتين للأكل. ولا بد من أن يستخدم الشوكتين اللتين على يمينه ويساره مباشرة.



حل مشكلة الفلاسفة الطاعمين

- **الحل الأول و مشكلة الاختناق**
- إذا أخذ كل الفلاسفة شوكاتهم اليسرى عندها لن يستطيع أي منهم الحصول على الشوكة اليمنى، وسيظل كل فيلسوف منتظرا الشوكة اليمنى حتى تفرغ، أي أن الفيلسوف ف 1، سيكون في انتظار الفيلسوف ف 2 لترك شوكته، والفيلسوف ف 2 سينتظر الفيلسوف ف 3 لترك شوكته، وهكذا انتظار دائري ، مما يسبب **إختناق (deadlock)**.
- **الحل الثاني و مشكلة الحرمان**
- يمكن حل المشكلة بجعل الفيلسوف يلتقط شوكته اليسرى ثم يختبر هل الشوكة اليمنى متاحة، فإن لم تكن كذلك يضع الفيلسوف شوكته اليسرى وينتظر فترة زمنية محددة ثم يعيد الكرة مرة أخرى.
- هذه الطريقة قد تفشل إذا قام كل الفلاسفة بالتقاط شوكاتهم اليسرى في وقت واحد، فلن يجدوا شوكاتهم اليمنى، فسيضعوا شوكاتهم اليسرى وينتظروا فترة زمنية محددة متساوية لكل الفلاسفة، وهكذا يظلوا يكررون في هذه المحاولة إلى ما لا نهاية وتظل المشكلة قائمة.
- هذا يحرم جميع الفلاسفة من الأكل وتحصل مجاعة أو ما يسمى **الحرمان (starvation)**

حل مشكلة الفلاسفة الطاعمين

- **الحل الثالث و احتمال الفشل**
- إذا يمكن حل هذه المشكلة بجعل فترات انتظار الفلاسفة عشوائية وبالتالي إحتمال أنها تكون متساوية قد يكون ضعيفا جدا، وهذا ما يطبق بالفعل في كثير من الأنظمة، لكن أحيانا نكون بحاجة إلى حل لا يحتمل الفشل بسبب تطابق الفترات العشوائية. ذلك لأن هنالك من الانظمة ما تدير أجهزة حساسة لا تقبل إحتمال أي فشل مثل أنظمة مراقبة المفاعل النووية.
- **الحل الرابع باستخدام السيمافور**
- هنا نقوم باستخدام سيمافور ثنائي (semaphore)، حيث يقوم الفيلسوف قبل الحصول على شوكة بغلاق 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 قادرين على الأكل في اللحظة الواحدة.

حل مشكلة الفلاسفة الطاعمين

- Shared data
 - Bowl of rice (data set)
 - Semaphore **chopstick** [5] initialized to 1

■ Semaphore Solution

■ The structure of Philosopher *i*:

```
while (true){  
    wait (chopstick[i] );  
    wait (chopstick[ (i + 1) % 5] );  
  
    /* eat for awhile */  
  
    signal (chopstick[i] );  
    signal (chopstick[ (i + 1) % 5] );  
  
    /* think for awhile */  
  
}
```