Plan

- Introduction
- Architecture : multi-coeurs, threads
- Les ordonnanceurs
- Création de processus, problèmes de partage d'information entre processus, fonctions élémentaires de gestion des processus
- Communication locale
- Interfaces Posix
- Introduction aux processus sous Python
- Annexe (si le temps le permet) : fonctionnement bas niveau (masques, processus)

Partie I : introduction

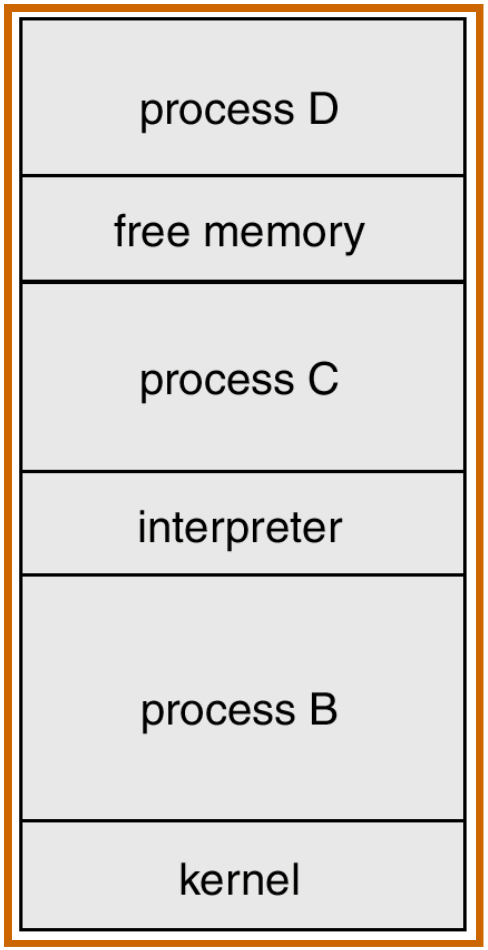
Un **processus** est un programme en exécution
Un processus a besoin de ressources : du temps CPU, de la mémoire, des fichiers, des devices d'I/O… pour accomplir sa tâche
Le système d'exploitation est responsable des activités suivantes en connection avec la gestion des processus :

- Création et suppresions de processus
- Suspension et redémarrage de processus
- Fourniture de mécanismes pour :
    - Synchronisation de processus
    - Communication entre processus

Le concept de processus

- Un système d'exploitation exécute une grande variété de travail :
- Batch system – jobs
- Time-shared systems – user programs or tasks
- Dans les livres, on utilise souvent de manière interchangeable les termes job (travail) et process (processus)
- Un processus c'est concrêtement :
- Program counter (compteur de programme)
- Stack (une pile)
- Data section (une zone de donnée)

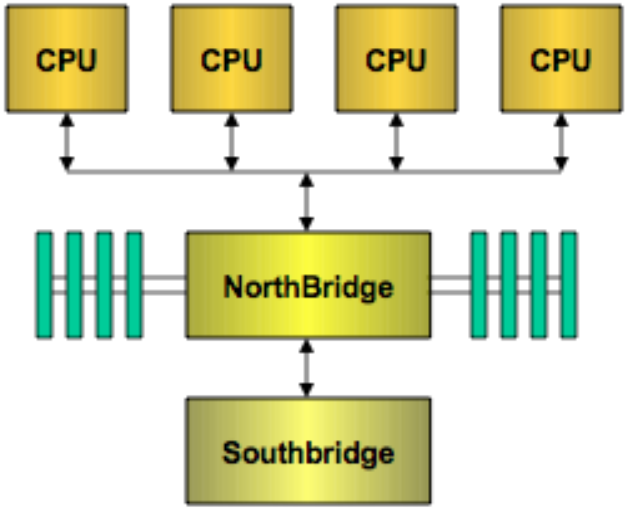
UNIX, la mémoire lorsqu'on tourne plusieurs processus



Pourquoi tourner plusieurs processus sur un seul CPU ?

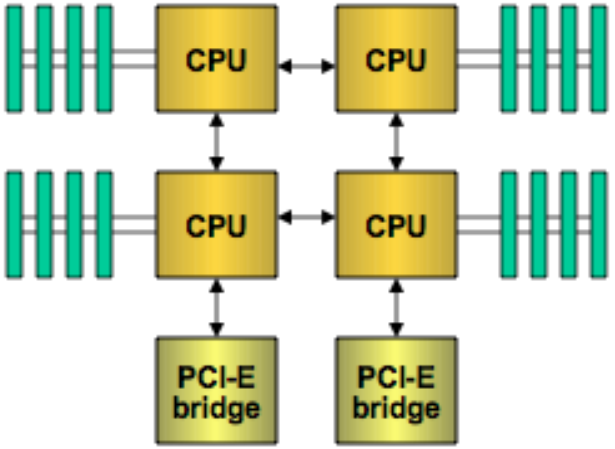
- Plusieurs utilisateurs sur le même processeur : passer équitablement la main à chacun d'eux
- Pour masquer la latence mémoire (on passe la main à un autre processus si l'on fait une opération coûteuse d'accès à la mémoire) - plus généralement, masquer un coût non lié au CPU afin de l'utiliser au mieux (best effort) : recherche du meilleur rendement

Intel Xeon



All CPUs share the FSB
- performance bottleneck
- 110 nsec memory latency from Cache miss
- Memory bandwidth does not scale with more CPUs

AMD Opteron



Each CPU has integrated memory controller
~ 55 nsec memory latency from Cache miss
Memory BW scales with Number of CPUs

Why is Latency so Important?

- Opteron has than half the memory latency of Xeon
- Total CPI = Native CPI + Miss Rate * Miss Cost

	AMD	Intel
– Native CPI:	0.4	0.4
– Miss Rate:	1%	1%
– Miss Cost:	160	360
– Total CPI:	2.0	4.0

Miss Rate and Memory Latency dominate CPU performance for memory intensive applications including floating point. Assuming the same 1% Miss Rate, Intel system performance is half of AMD

En 2011 Intel passe au NUMA

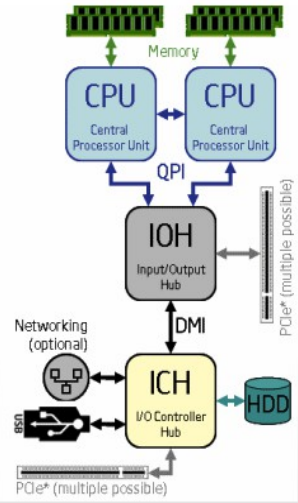
```
top - 16:35:26 up 2 days, 54 min,  1 user,  load average: 0.00, 0.00, 0.00
Tasks: 729 total,   1 running, 728 sleeping,   0 stopped,   0 zombie
Cpu0  :  0.0%us,  0.0%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.3%si,  0.0%st
Cpu1  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu2  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu3  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu4  :  0.3%us,  0.0%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu5  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu6  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu7  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu8  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu9  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu10 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu11 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu12 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu13 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu14 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu15 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu16 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu17 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu18 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu19 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu20 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu21 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu22 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu23 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu24 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu25 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu26 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu27 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu28 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu29 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu30 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu31 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu32 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu33 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu34 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu35 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu36 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu37 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu38 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu39 :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  529292356k total,  2050816k used, 527241540k free,   131580k buffers
Swap:  1023428k total,        0k used,  1023428k free,   554872k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
31840 christop  20   0 17584 1852 1008 R  0.3  0.0   0:00.84 top
    1 root      20   0 21260 1532 1244 S  0.0  0.0   0:10.20 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.01 kthreadd
    3 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
    4 root      20   0     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/0
```

Home ▸ Intel® Processors ▸ Intel® Xeon® Processor E7 Family ▸ Intel® Xeon® Processor E7-4800 Series ▸ E7-4850

Intel® Xeon® Processor E7-4850
(24M Cache, 2.00 GHz, 6.40 GT/s Intel® QPI)

SPECIFICATIONS
COMPATIBLE PRODUCTS
BLOCK DIAGRAMS
ORDERING / SSPECS / STEPPINGS

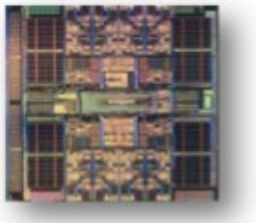
BLOCK DIAGRAMS



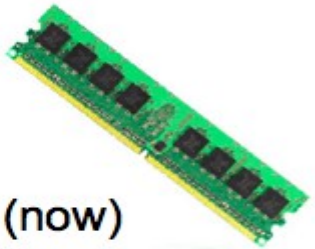
Processeurs multi-coeurs

- http://gamma.cs.unc.edu/SC2007/

Looking Forward …



- **Cores**
  - **more, but simpler/smaller**
    - less out-of-order hardware, reduced power
  - **more heterogeneous**
    - multiple services
- **DRAM**
  - **getting bigger**
    - 64 Mb (1994) to Samsung 2 Gb DDR2 (now)
  - **but probably not enough faster**
    - 70 ns (1996) to Samsung DDR2 40-60ns (now)
  - **and banking has its limits (cost and pins)**

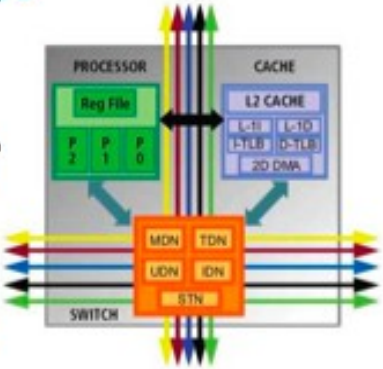


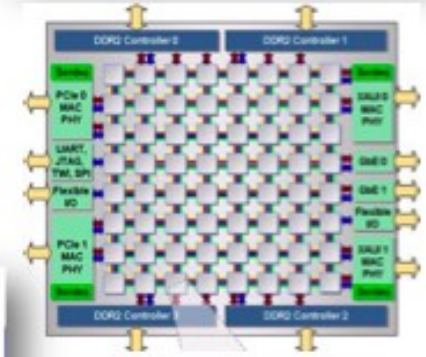
Dan Reed (IBM)

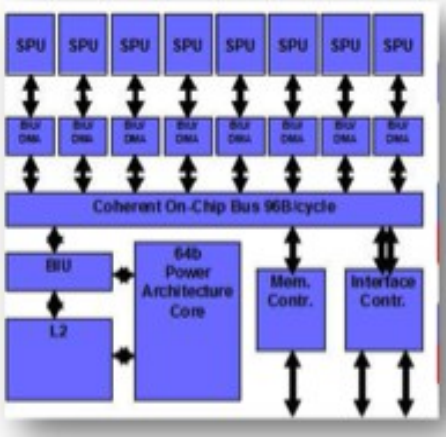
Dan Reed (IBM)

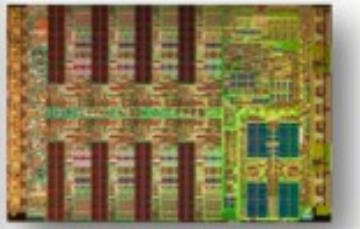
ManyCore Mashups

- Intel's 80 core prototype
  - 2-D mesh interconnect
  - 62 W power
- Tilera 64 core system
  - 8x8 grid of cores
  - 5 MB coherent cache
  - 4 DDR2 controllers
  - 2 10 GbE interfaces
- IBM Cell
  - PowerPC
  - and 8 cores





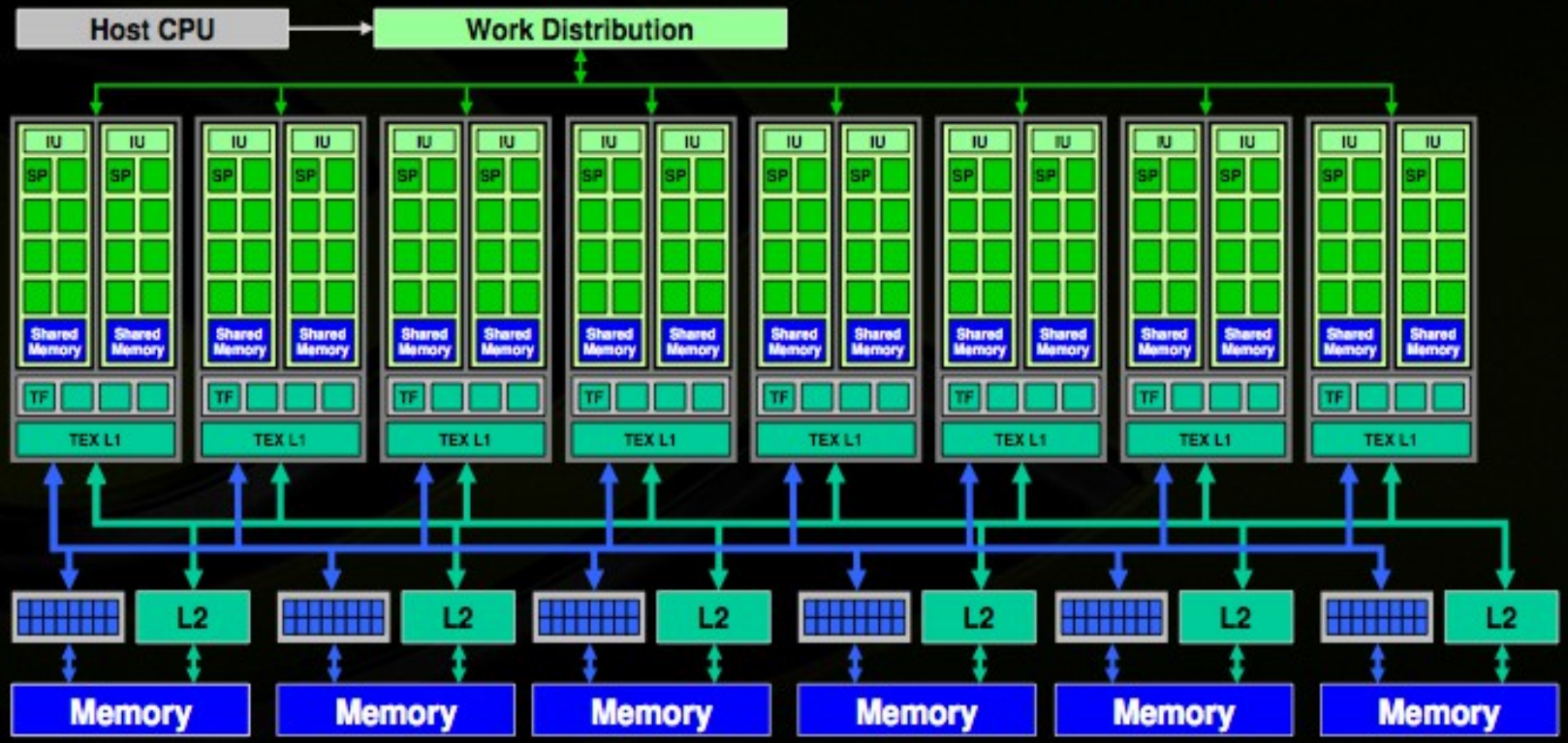




Renci

The Future of Computing is Parallel

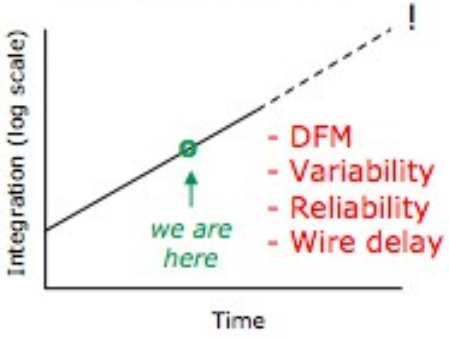
- CPU clock rate growth is slowing, future speed growth will be from parallelism
- GeForce-8 Series is a massively parallel computing platform
  - 12,288 concurrent threads, hardware managed
  - 128 SP Thread Processor cores at 1.35 GHz == 518 GFLOPS peak
  - GPU Computing features enable C on Graphics Processing Unit



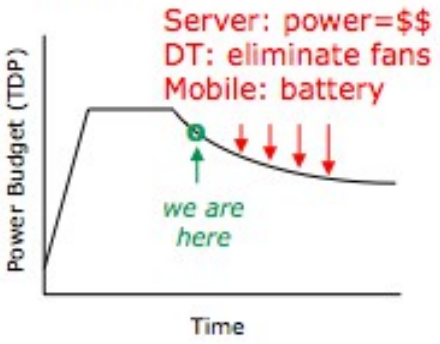
A Few High-level Trends

AMD
Smarter Choice

Moore's Law ☺

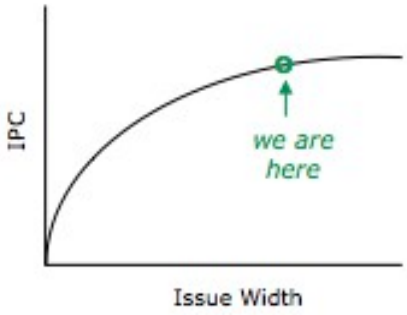


The Power Wall ☹

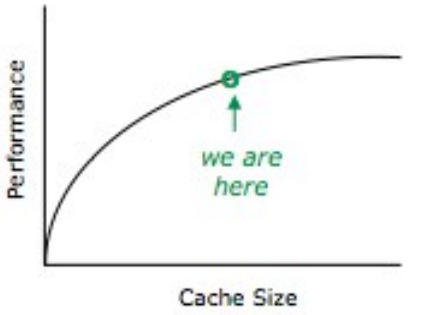


The Frequency Wall ☹

The Complexity Wall ☹



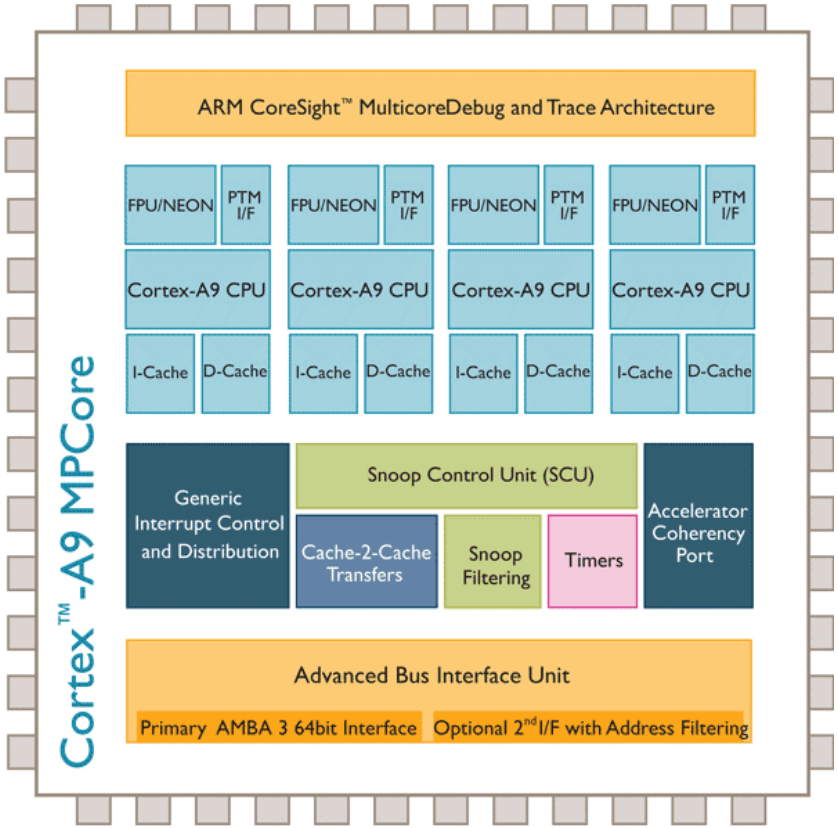
Locality 😐



Single thread Perf (!)

So, how can we add customer value?

Dans les smartphones on trouve aussi plusieurs coeurs



ARM Cortex-A9 Performance Power & Area			
	Cortex-A9 Single Core Soft Macro Trial Implementation	Cortex-A9 Dual Core Hard Macro Implementations	
Process	TSMC 65G	TSMC 40G	
Optimization method	Performance Optimized	Performance Optimized	Power Optimized
Standard Cell Library	ARM SC12	ARM SC12 + High Performance Kit	ARM SC12 + High Performance Kit
Performance (Total DMIPS)	2,075 DMIPS	10,000 DMIPS	4,000 DMIPS
Frequency	830 MHz	2000 MHz (typical)	800 MHz (wc/ss)

NVIDIA TEGRA processor

NVIDIA Tegra 2

As the world's first mobile super chip, NVIDIA® Tegra™ 2 brings extreme multitasking with the first mobile dual-core CPU, the best mobile Web experience with up to two times faster browsing, hardware accelerated Flash, and console-quality gaming with an ultra-low power (ULP) NVIDIA® GeForce® GPU. Get never-before-seen experiences on a mobile device with NVIDIA Tegra.

Key Features

- **Dual-core ARM Cortex-A9 CPU** – The world's first mobile dual-core CPU for faster Web browsing, snappier response time, and overall better performance. The Cortex-A9 is the first mobile CPU with out of order execution for more efficient processing, resulting in a better overall experience.
- **Ultra-low power (ULP) GeForce GPU** – Architected for low-power applications, the ULP GeForce GPU delivers outstanding mobile 3D game playability and a visually engaging, highly-responsive 3D user interface.
- **1080p Video Playback Processor** – Watch 1080p HD movies stored on your mobile device on your HDTV without compromising battery life.

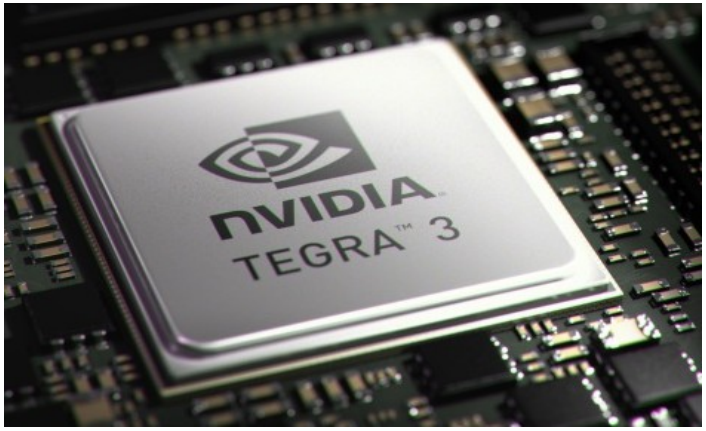
NVIDIA TEGRA 3

Tegra 3 (Kal-El) series [edit]

The Tegra 3 is functionally a SoC with a quad-core CPU, but includes a fifth "companion" core. While all cores are Cortex-A9s, the companion core is manufactured with a special low power silicon process that uses less power at low clock rate but does not scale well to high clock rates; hence it is limited to 500 MHz. There is also special logic to allow running state to be quickly and transparently transferred between the companion core and one of the normal cores. The goal is for a mobile phone or tablet to be able to power down all the normal cores and run on only the companion core, using comparatively little power, during standby mode or when otherwise underutilizing the CPU. According to Nvidia, this includes playing music or even video content.[19] Compared to Tegra 2, the ARM Cortex-A9s in Tegra 3 now supports ARMs SIMD extension, marketed as NEON. It can also output video up to 2560×1600 resolution and supports 1080p MPEG-4 AVC/h.264 40 Mbps High-Profile, VC1-AP, and DivX 5/6 video decode.[20]

The Tegra 3 was officially released on November 9, 2011.[21]

Model number	Semiconductor technology	CPU instruction set	CPU	CPU Cache	GPU	Memory technology	Availability	Utilizing Devices
Tegra 3	40 nm LPG by TSMC	ARMv7	1.5 GHz (in single-core mode) quad-core ARM Cortex-A9	L1: 32 KB instruction + 32 KB data, L2: 1 MB	ULP GeForce (improved over Tegra 2)	32-bit single-channel LPDDR2-1066 or DDR3-L up to 1600 MHz	Q1 2012	Asus Eee Pad Transformer Prime[22], IdeaTab K2 / LePad K2[23], Acer Iconia Tab A510, Acer Iconia Tab A700[24], LG Optimus 4X HD, HTC One X, ZTE Era, ASUS Transformer Pad Infinity 700 Series (WiFi-version), ASUS Transformer Pad 300 Series, ZTE PF 100, ZTE T98, Toshiba AT270



Apple Ax

The Apple A5 SoC

The Apple A5X SoC

List of Apple SoCs

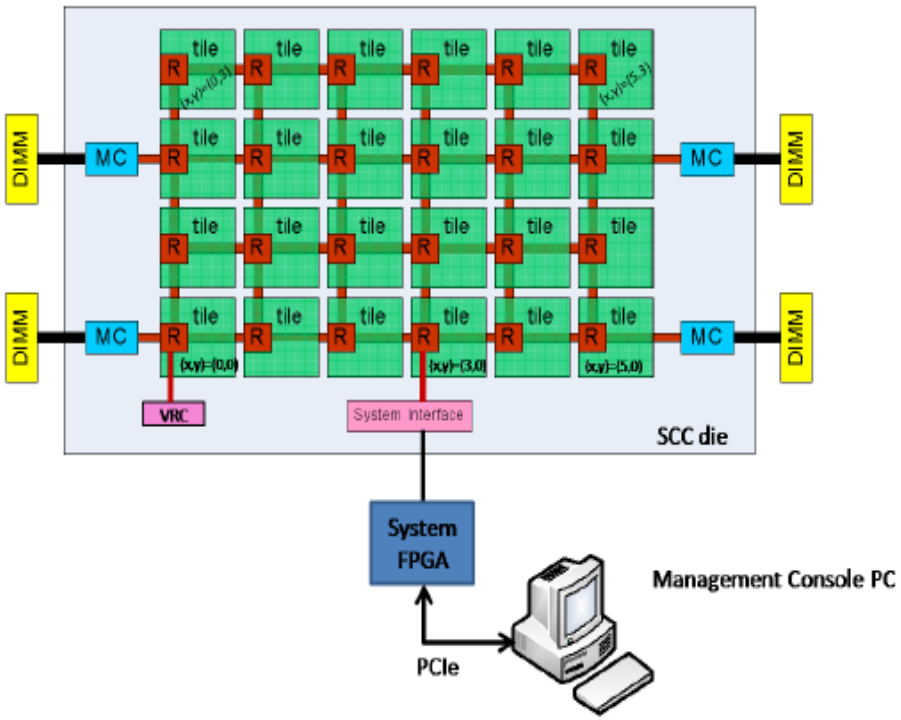
Name	Model no.	Fabrication process	Instruction set	CPU	CPU cache	GPU	Memory technology	Introduced	Devices
A4	S5L8930	45 nm, by Samsung	ARMv7	800–1000 MHz single-core ARM Cortex-A8	L1: 32 kB Instruction + 32 kB Data, L2: 512 kB	PowerVR SGX535	32-bit Dual-channel 200 MHz LPDDR	March 2010	• iPad • iPhone 4 • iPod Touch (4th gen.) • Apple TV (2nd gen.)
A5	S5L8940	45 nm, by Samsung	ARMv7	800–1000 MHz dual-core ARM Cortex-A9	L1: 32 kB instruction + 32 kB data, L2: 1 MB	PowerVR SGX543MP2 (dual-core)	32-bit Dual-channel 266 MHz LPDDR2	March 2011	• iPad 2 • iPhone 4S
	S5L8942	45 nm, by Samsung	ARMv7	Single-core ARM Cortex-A9	L1: 32 kB instruction + 32 kB data, L2: 1 MB	PowerVR SGX543MP2 (dual-core)	32-bit Dual-channel	March 2012	• Apple TV (3rd gen.)
A5X	S5L8945	45 nm, by Samsung[37]	ARMv7	1 GHz Dual-core ARM Cortex-A9	L1: 32 kB instruction + 32 kB data, L2: 1 MB	PowerVR SGX543MP4 (quad-core)	32-bit Quad-channel 400 MHz LPDDR2[38]	March 2012	• iPad (3rd gen.)

INTEL Single Chip Cloud Computing (SCC) Platform Overview

The SCC is the second generation processor design that resulted from Intel's Tera-Scale research. The first was Intel's Teraflops Research Chip; it had 80 non-IA cores. The second is the SCC. The SCC is a 48-core chip; each tile has two cores. The SCC core is a full IA P54C core and hence can support the compilers and OS technology required for full application programming.

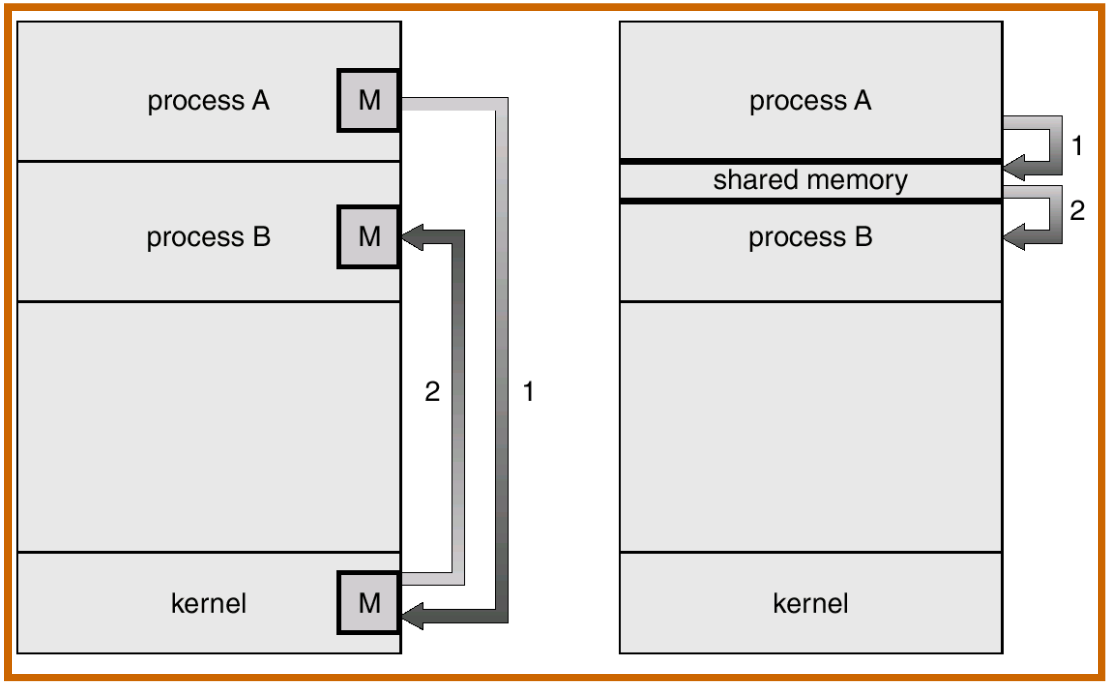
The architecture is based on the 24 tiles arranged in a XxY = 6x4 array. There is a router associated with each tile. Four memory controllers go to off-die (but on-board) DDR3 memory. The SCC board communicates with its Management Console (MCPC) over a PCIe bus. ***Key features of this software are the ability to load a Linux image on each core or a subset of cores, to read and modify SCC configuration registers, and to load programs on the SCC cores.***

INTEL Single Chip Cloud Computing (SCC) Platform Overview



Modèles de communication : vues abstraites

La communication peut avoir lieu en utilisant soit le passage de messages soit la mémoire partagée



Message Passing

Shared Memory

Etat d'un processus

Au fur et à mesure de **son exécution**, un processus change **d'état** :

new: le processus est en cours de création

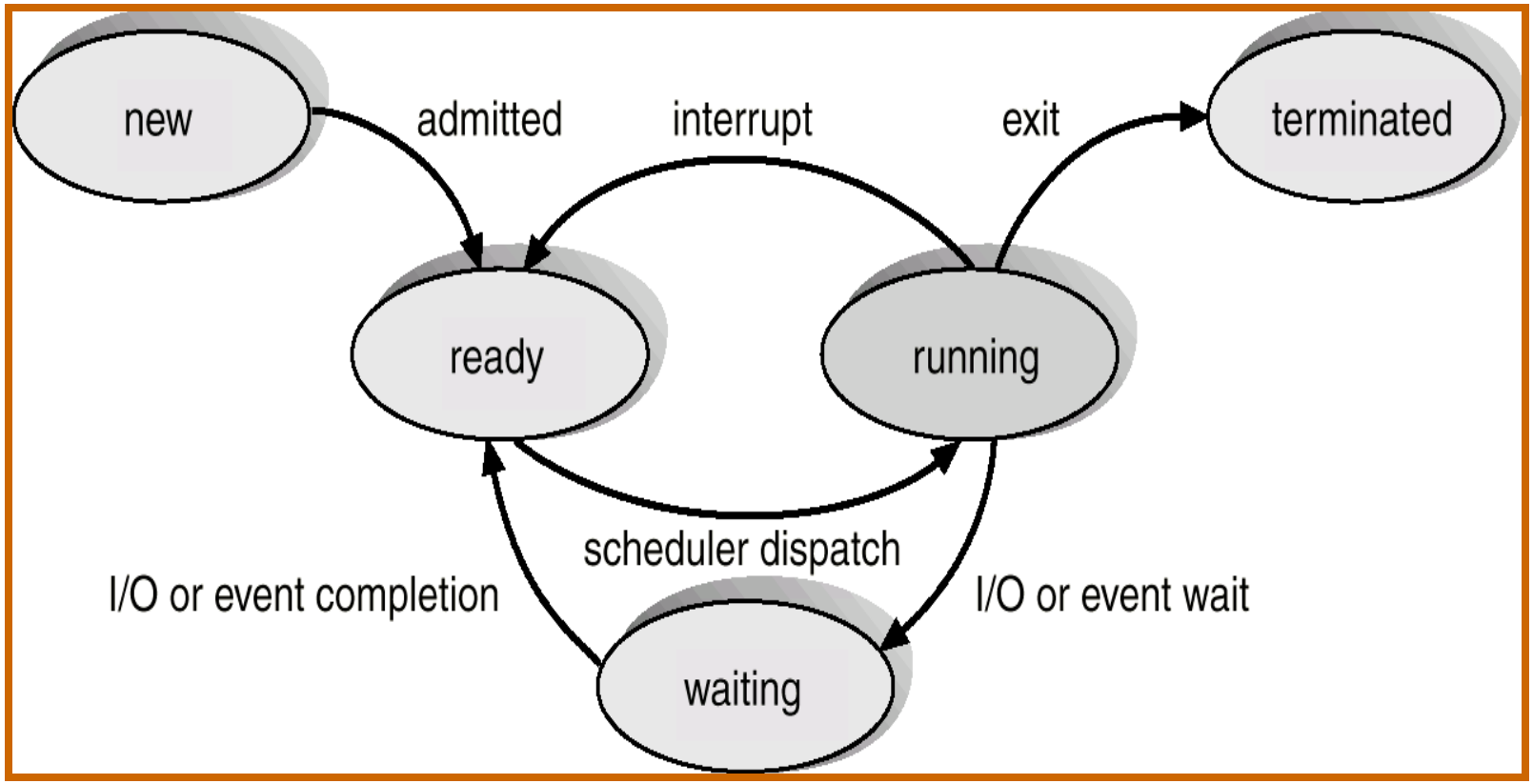
running: des instructions sont en cours d'exécution

waiting: le processus est en attente d'un évènement

ready: le processus est en attente de passer en running (gestion du scheduler)

terminated: le processus a terminé son exécution

Diagramme d'état d'un processus

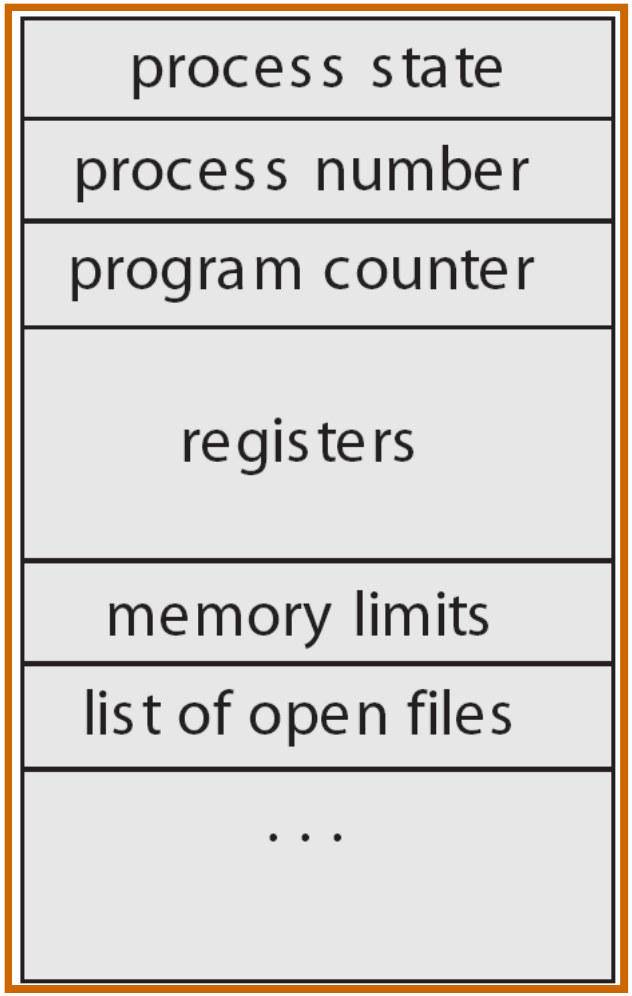


Process Control Block (PCB)

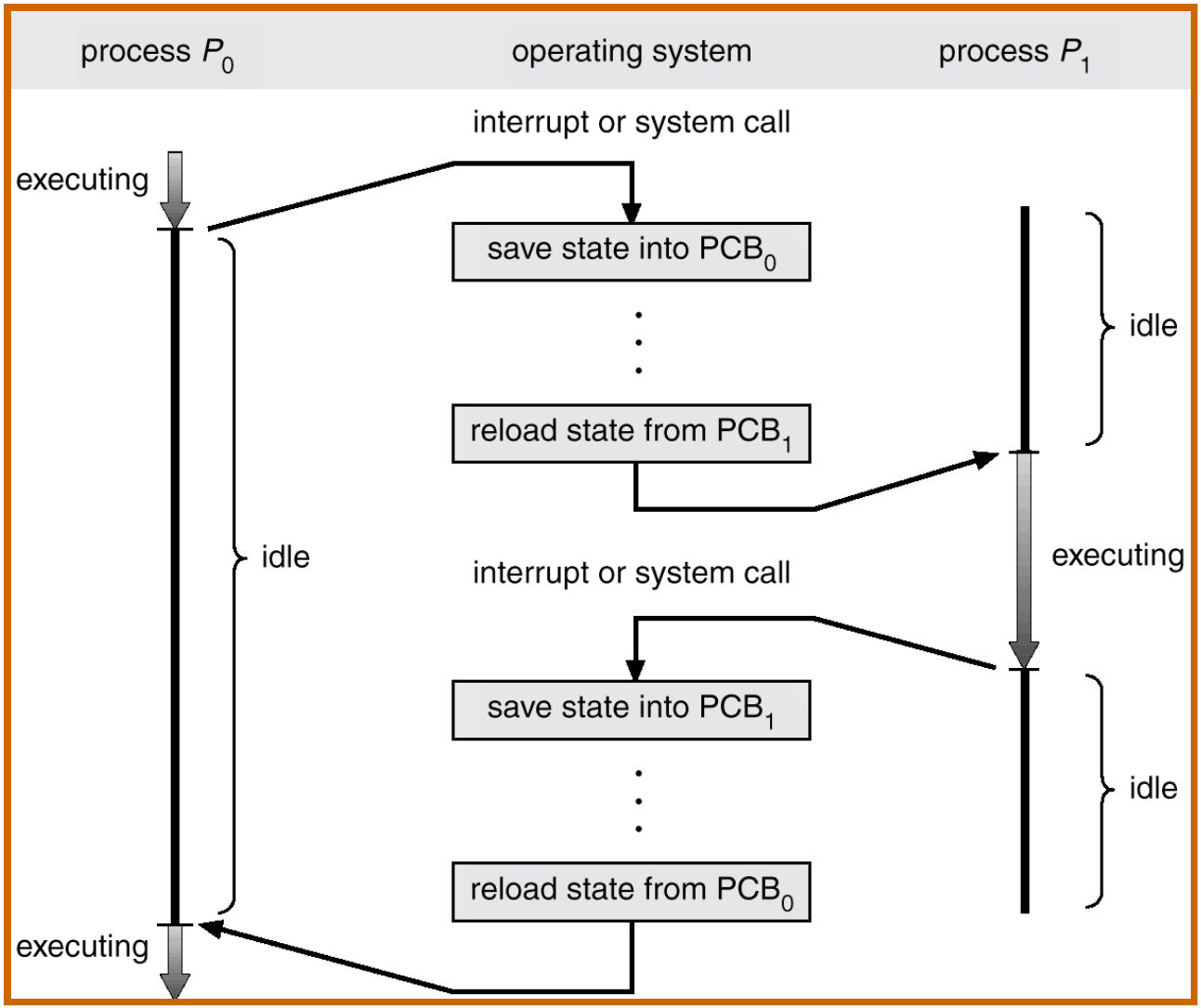
Information associée avec chaque processus :

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

Process Control Block (PCB)



Le CPU change de contexte



Context Switch

- Quand le CPU change de contexte pour un autre processus il faut sauvegarder et charger de nouveaux états
- Le temps du Context-switch est un surcoût dans le sens on cela ne fait pas progresser le programme / calcul
- Le temps du Context-switch dépend du support matériel dont on dispose : des machines ont même été construites pour masquer 128 cycles CPU (machine Tera)

Partie 2 : Schedulers (ordonnanceurs)

Il a la charge d'ordonnancer les processus (travaux/jobs) et d'assurer des propriétés comme l'équité (tous les processus auront un jour la main), d'équilibrage des charges (contexte multi-coeur)

Souvent les objectifs sont orthogonaux : minimiser le temps de réponse et maximiser l'occupation du CPU

Long-term scheduler (or job scheduler) – sélectionne quels processus doivent être mis dans la file "ready"
Short-term scheduler (or CPU scheduler) – sélectionne quel processus devra être exécuté prochainement et réserve le CPU

Schedulers (Cont.)

Short-term scheduler : invoqué très fréquemment (milliseconds) ⇒ (doit être rapide à répondre)
Long-term scheduler : invoqué pas très souvent (seconds, minutes) ⇒ (peut être lent)
Processes peuvent aussi être décrits comme :
- I/O-bound process – il passe son temps à faire des opérations d'I/O, beaucoup de petits bursts du CPU
- CPU-bound process – passe son temps à faire du calcul; petit nombre de très grands bursts du CPU

Les politiques d'ordonnancement

- Très nombreuses : il s'agit d'heuristiques car le problème théorique caché est non-polynomial (il n'existe pas d'algorithme polynomiaux). De plus il est multicritère (minimiser le temps d'exécution des tâches et maximiser l'utilisation des CPU et minimiser les migrations et....)
- Une heuristique n'atteint pas toujours l'optimalité sur chaque instance.

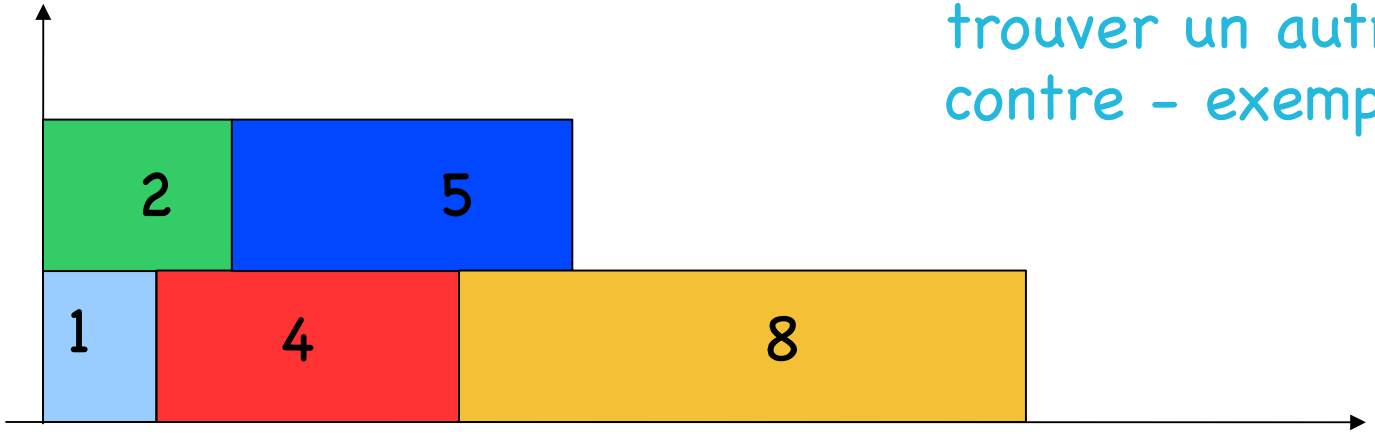
Ordonnancement de liste

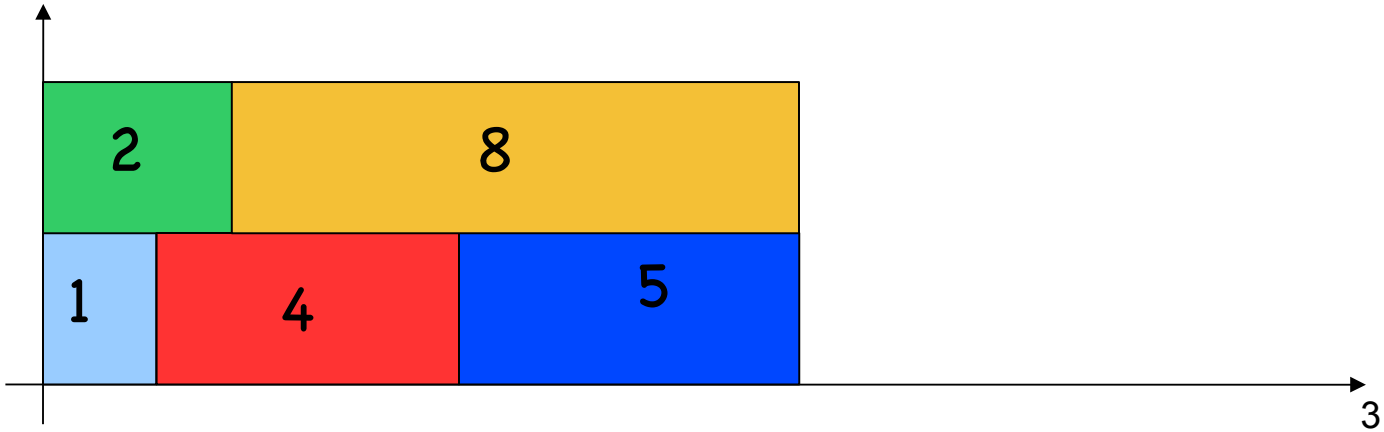
- Ici, vous avez n cours (durées connues) à caser dans m salles
- Algorithme :
  - on place le cours le moins coûteux sur la salle la moins chargée
  - Rq : et si on choisissait de placer le plus coûteux dans la salle la moins chargée ?

```
ListScheduling(m, n, c_1,...,c_n) {
      for i=1 to m {
         cost_i=0;  /* load in room i */
         J(i)=Vide; /* courses assigned to room i*/
      }
      for j=1 to n {
          i = min_{1<= k <= m} cost_k; /* room i has smallest load */
          J(i) = J(i) '+' {j}; /* assign course j to room i */
          cost_i=cost_i + c_i
      }
}
```

Optimalité ?

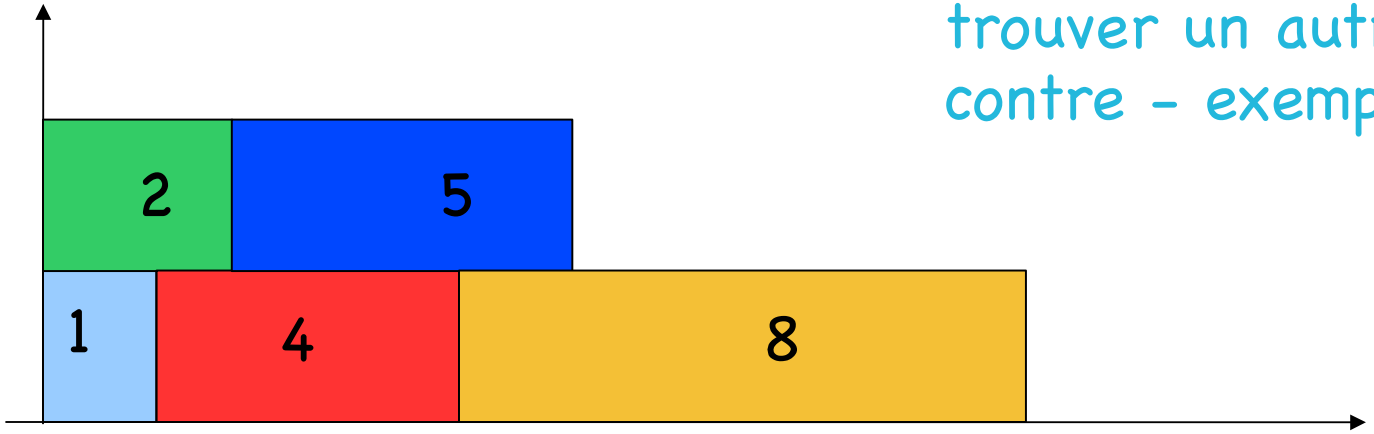
Sur cet exemple, optimalité si on place d'abord les plus gros jobs... mais on peut trouver un autre contre - exemple

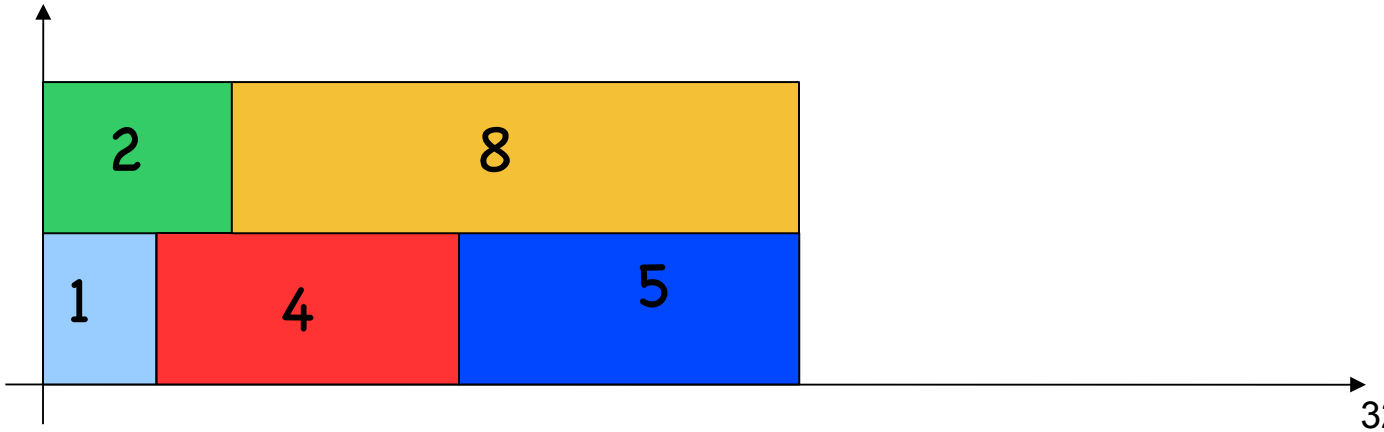




Optimalité ?

Sur cet exemple, optimalité si on place d'abord les plus gros jobs... mais on peut trouver un autre contre - exemple





Théorème & démonstration

Théorème (Graham 1966) : le précédent algorithme donne un solution qui est toujours éloignée de l'optimal d'un facteur pas plus grand que (2-1/m)... ce qui est plutôt bon dans la pratique.

Preuve : on considère m machines et n tâches de temps d'exécution $p_1,...,p_n$. Soit c^*_{max} le temps d'exécution (makespan) de l'ordonnancement optimal. Considérons le plus grand temps d'exécution parmi les n tâches : $p_{max} = max_j\ p_j$ Puisque la tâche correspondante à p_{max} doit s'exécuter on a:

$$c^*_{max} \geq p_{max}$$

On peut aussi faire l'hypothèse que les tâches peuvent être découpées en morceaux (préemptées) de telle sorte que le temps d'exécution de toutes les machines est le même. Nous avons donc une autre borne inférieure :

Théorème & démonstration

On peut aussi faire l'hypothèse que les tâches peuvent être découpées en morceaux (préemptées) de telle sorte que le temps d'exécution de toutes les machines est le même. Nous avons donc une autre borne inférieure :

$$c^*_{\max} \geq \frac{1}{m}\left(\sum_{j=1}^{n} p_j\right)$$

Théorème et démonstration

Soit J_k la dernière tâche à être exécutée. Puisque LS n'autorise pas aucune machine à être inactive, toutes les machines sont occupées jusqu'au temps sk, calculant des tâches autres que k. D'où :

$$s_k \leq \frac{1}{m}\left(\sum_{j \neq k} p_j\right)$$

Théorème et démonstration

Soit J_k la dernière tâche à être exécutée. Puisque LS n'autorise pas aucune machine à être inactive, toutes les machines sont occupées jusqu'au temps sk, calculant des tâches autres que k. D'où :

$$s_k \leq \frac{1}{m}\left(\sum_{j \neq k} p_j\right)$$

Théorème et démonstration

En conséquence :

$$
\begin{aligned}
c_{\max}(LS) &= c_k \\
&= s_k + p_k \\
&\leq \frac{1}{m}\left(\sum_{j \neq k} p_j\right) + p_k \\
&= \frac{1}{m}\left(\sum_{j=1}^{n} p_j\right) + \left(1 - \frac{1}{m}\right) p_k \\
&\leq c^*_{\max} + \left(1 - \frac{1}{m}\right) c^*_{\max} \\
&= \left(2 - \frac{1}{m}\right) c^*_{\max}
\end{aligned}
$$

… on préfère donc des heuristiques, parmi lesquelles :

- First In, first Out (FIFO), que l'on peut traduire par « premier arrivé, premier servi »
- Earliest deadline first scheduling est un algorithme d'ordonnancement préemptif à priorité dynamique utilisé dans les systèmes temps réel. Il attribue une priorité à chaque requête en fonction de l'échéance de cette dernière. Plus l'échéance d'une tâche est proche, plus sa priorité est grande.

… on préfère donc des heuristiques, parmi lesquelles :

- Round-robin (tourniquet) qui attribue des tranches de temps à chaque processus en proportion égale, sans accorder de priorité aux processus. Problème : quelle valeur pour le quantum de temps ?
- SJF est l'acronyme de Shortest Job First (« plus court processus en premier ») :
  - Il existe deux versions de cet algorithme : une version préemptive, et une version non préemptive. Dans cette dernière, un processus ayant pris le contrôle de l'UC ne le quitte qu'une fois la rafale terminée.
  - La version préemptive, aussi appelée SRIF, Shortest Remaining Time First (« plus court temps restant en premier »), est plus souple. Si un processus dont le temps d'exécution est plus court que le reste du temps d'exécution du processus en cours de traitement entre dans la file d'attente, alors il prendra sa place.

Mise en garde

- Dans les transparents précédents on a joué subtilement sur le fait de connaître ou pas dès le départ l'ensemble des tâches, leurs durées d'exécution (dates de début, date de fin)
- Dans la réalité : il faut faire du online (à la volée) et prendre des décisions quand une tâche arrive :
  - online clairvoyant : quand la tâche arrive elle annonce sa durée d'exécution
  - non clairvoyant : elle ne dit rien
- ET DONC TOUT EST ENCORE PLUS DIFFICILE DANS LA « VRAIE VIE »

Support du SMP(multiprocesseur)

- Le noyau Linux 2.6 associe un processus avec un des coeurs (affinity) ;
- Vous avez une runqueue par coeur ;
- Chaque 200ms on vérifie si les charges des coeurs différent et si c'est le cas, on migre des processus depuis un coeur chargé vers un coeur moins chargé :
  - attention : la migration a un coût
  - attention : migration = problème avec les caches
  - statistiques disponibles si CONFIG_SCHEDSTATS

Principales fonctions du noyau

Function name	Function description
schedule	The main scheduler function. Schedules the highest priority task for execution.
load_balance	Checks the CPU to see whether an imbalance exists, and attempts to move tasks if not balanced.
effective_prio	Returns the effective priority of a task (based on the static priority, but includes any rewards or penalties).
recalc_task_prio	Determines a task's bonus or penalty based on its idle time.
source_load	Conservatively calculates the load of the source CPU (from which a task could be migrated).
target_load	Liberally calculates the load of a target CPU (where a task has the potential to be migrated).
migration_thread	High-priority system thread that migrates tasks between CPUs.

```
    pid = atol(argv[1]);
    sscanf(argv[2], "%08lx", &new_mask);

    if (sched_getaffinity(pid, len,&cur_mask) < 0) {
        perror("sched_getaffinity");
        return -1;
    }
    printf("pid %d's old affinity: %08lx\n",
           pid, cur_mask);


  if (sched_setaffinity(pid, len, &new_mask)) {
      perror("sched_setaffinity");
      return -1;
  }

  if (sched_getaffinity(pid, len,&cur_mask) < 0) {
      perror("sched_getaffinity");
      return -1;
  }

  printf(" pid %d's new affinity: %08lx\n",
         pid, cur_mask);

  return 0;
}
```

Partie 3 : Création de processus

Il y a toujours un processus parent pour la création d'un nouveau processus... qui lui même peut créer de nouveaux processes formant ainsi un "arbre de processus"

Partage de ressources :

- Le père et les fils partagent toutes les ressources
- Les fils partagent un sous-ensemble des ressources du père
- Le père et le fils ne partage rien

Exécution

- Le père et le fils s'exécutent concurremment
- Le père attend que les fils terminent

Programme C qui lance deux processus séparés

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
int pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
        exit(0);
    }
}
```

Termination des processus

- Le processus exécute la dernière instruction qui se trouve être une demande au SE de sortir par l'instruction **exit**
- Synchronisation entre père et fils via **wait** (noter qu'après, le père peut éventuellement faire d'autres instructions)
- Les ressources du processus sont désallouées par le SE (pas par le programmeur)
- Un père peut terminer l'exécution de processus fils via l'instruction **abort :**
- Si le fils a "trop" de ressources
- La tâche allouée au fils n'est plus requise
- Si le père sort :
- certains operating systems n'autorisent pas les fils à poursuivre si le père termine :
- All children terminated - cascading termination

Processes coopérants

- Un processus indépendant ne peut pas être affecté ou affecter un autre processus. Autrement dit : dans l'exemple, les 2 processus ne "partagent" rien
- Cooperating process : peut affecter ou être affecté par un autre processus
- Advantages de la coopération de processus
  - Partage d'information
  - Accélération du calcul (échange direct ; pas via un disque)
  - Modularité
  - Commodité

Mise en garde

- Il est très facile d'écrouler un système lorsqu'on écrit un programme avec des instructions fork()
- Exemple : un fork() se retrouve dans une boucle infinie !
- Dans l'exemple qui suit il y a 5 processus : quand on fork(), le processus créé ne reprend pas son exécution « au début du main »... mais à l'instruction qui suit le fork()

Exemple

On reprend la main car le pere termine immédiatement

```
$ ./a.out
$
 Fils a termine son sommeil
   pid fils: 1200

 Fils a termine son sommeil
   pid fils: 1201

 Fils a termine son sommeil
   pid fils: 1202

 Fils a termine son sommeil
   pid fils: 1203

 Fils a termine son sommeil
   pid fils: 1204

$
```

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
  int i, pid;

  for (i = 0; i < 5; i++)
    {
      /* fork another process */
      pid = fork ();
      if (pid == 0)
        {
          sleep (2 * i + 1);
          printf ("\n Fils a termine son sommeil\n ");
          printf ("  pid fils: %d \n", getpid ());
          exit (0);
        }
    }
}
```

Que se passe t'il sur cet exemple ?

- A la création, il y a duplication de l'image du processus : le fils obtient une copie de l'espace d'adressage du père (en particulier une copie des variables) + descripteurs fichiers +...
- Le fils travaille sur les copies des variables - à la terminaison la valeur est oubliée ⇒ on ne peut pas passer le contenu d'une variable du fils au père
- L'exemple qui suit montre ce problème.

```
/*
 * Calcul d'une valeur approchee de PI. John Wallis (1616-1703) decouvrit
 * en 1655 une methode de calcul de PI. Il considere un cercle de rayon
 * unitaire et il cherche l'air d'un quart de cercle. Il propose alors la
 * formule PI/4 = Int_0^1 sqrt{1-x*x} dx  (l'integrale de 0 a 1 de la
 * fonction racine carree de 1-x*x (note : l'equation d'un cercle de
 * rayon R est y^2 + x^2 = R^2))
 *
 */
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
const double pi = 3.14159265358979323 85;

int main( int argc, char *argv[] )
{
  /*  Arguments required for executing Pi:           */
  /*  0                          -> program name     */
  /*  1                          -> numberOfIntervals */

  int numberOfIntervals, interval;
  double intervalWidth, intervalMidPoint;
  double area=0.0, area1 = 0.0, area2 = 0.0;

  pid_t n ;
```

```
int status ;


 if(argc != 2) {
   printf("Please, specify interval number\n"); exit(0);
 }

 /*  The number of intervals is a command line argument. */
 numberOfIntervals = atoi( argv[1] );

 /*  Compute the interval width. */
 intervalWidth = 1.0 / numberOfIntervals;

 /*  Now compute the area. */
 n = fork();
 if (n == 0) { /* fils */
    for ( interval = 0; interval < numberOfIntervals/2; interval++ )
      {
        intervalMidPoint = (interval + 0.5) * intervalWidth;
        area1 += 4.0 / ( 1.0 + intervalMidPoint*intervalMidPoint );
      }
     area1 *= intervalWidth;
```

```
 } else { /* pere */
     for ( interval = numberOfIntervals/2; interval < numberOfIntervals;
           interval++ )
       {
         intervalMidPoint = (interval + 0.5) * intervalWidth;
         area2 += 4.0 / ( 1.0 + intervalMidPoint*intervalMidPoint );
       }
      area2 *= intervalWidth;
  }
  /* Suspend l'execution du processus dans lequel il est appele */
  /* jusqu'a reception d'une valeur status d'un processus enfant */
  /* termine. */
  wait(&status);

  /*  Print the results. */
  area = area1 + area2;
  printf( "The computed value of the integral is %.15f\n", area );
  printf( "The exact value of the integral is    %.15f\n", pi );
  printf( "The error (exact value - estimated) is %.15f\n", pi-area );

  return 0;
}
```

Exécution

```
[christophe@localhost christophe]$ gcc pi.c
[christophe@localhost christophe]$ a.out 100
The computed value of the integral is 1.854601102849095
The exact value of the integral is    3.141592653589793
The error (exact value - estimated) is 1.286991550740698
The computed value of the integral is 1.286999884074031
The exact value of the integral is    3.141592653589793
The error (exact value - estimated) is 1.854592769515762

On remarque :
  - que le résultat n'est pas correct
  - qu'il y a 2 fois l'affichage (et on ne peut pas
décaler l'affichage car il faut produire les résultats)
```

Donc, pas de partage de variables entre les processus

Fonctions utiles

- getpid(): identité du processus appelant : getppid(): identité du pére du processus appelant
- wait() et waitpid()
  - synchronisation père-fils
  - récupération d'info (code retour, Texe)

Fonctions utiles

- pid_t wait(int *ptr_status)
  - Valeur de retour : aucun fils => -1 ; si fils alors pid du fils
  - S'utilise du coté du père
  - Sémantique : si aucun des fils n'est terminé, le père se bloque. La fin du blocage arrive soit par la fin d'un fils soit par une « interruption » du père (réception d'un signal : voir plus loin)

Fonctions utiles

Si le fils mentionné par pid s'est déjà terminé au moment de l'appel il est devenu "zombie"

- pid_t waitpid(pid_t pid, int ptr_status, int options) :
  - valeur de retour :
    - -1 : erreur (pas de fils)
    - 0 : échec
    - >0 : pid d'un fils (zombie) s'il existe
  - argument pid (sélection processus à attendre) :
    - -1 : " fils
    - 0 : " fils appartenant groupe de l'appelant
    - >0 : fils ayant cette identité

Fonctions utiles

- pid_t waitpid(pid_t pid, int ptr_status, int options) :
  - options :
    - 0 : bloquant
    - 1 : non bloquant (cte WNOHANG)
      - s'il n'existe aucun fils terminé il y a échec et retourne '0' sinon retourne le pid d'un fils terminé ou -1

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
#include<sys/types.h>
/*
Une tache comprend 2 sous taches de durée différente.
Le pere décide d'attendre la fin d'exécution du
premier fils et de tuer le second fils s'il n'etait
pas termine.
*/
int
main ()
{
  int i, status;
  pid_t pid[2];

  for (i = 0; i < 2; i++)
    {
      pid[i] = fork ();
      if (pid[i] == 0)
     {
       printf("PID[%d] s'endort pendant %d\n",i,5*i+2);
       sleep (5 * i + 2);
       break; // ou exit(0) pour eviter fork() pour fils
     }
    }

  if (i == 2)            /* seulement le pere */
    {
      printf("Pere commence son attente\n");
      pid[0] = waitpid (pid[0], &status, 0);
      if (pid[0] < 0 || status != 0)
     kill (pid[1], SIGINT);
    }
  //exit (0);

}
```

```
cerin@taipei:~$ ./a.out
PID[0] s'endort pendant 2
PID[1] s'endort pendant 7
Pere commence son attente
cerin@taipei:~$
```

Partie 4 : faire communiquer les processus ; échange d'information

- Objectif : mettre en place le partage d'information, la communication entre processus
- Communication locale uniquement (sur le même processeur) entre processus
  $\Rightarrow$ en 2ème année : communication distante via socket()

Différents mécanismes pour communiquer localement sous UNIX

- Via la mémoire partagée (primitive mmap())
- Via les tuyaux (primitive pipe())
- Via les signaux (primitives signal() et kill())
- Les files de messages (primitive msgget(), msgsnd() ⇒ non abordées

Mémoire partagée

- Idée : se munir de primitives permettant de réserver/supprimer une zone mémoire commune (partagée) entre les processus
- Problème supplémentaire : assurer éventuellement une exclusion mutuelle (accès par au plus un seul processus à tout instant à cette zone sinon gare aux valeurs des objets car accès concurrents en lecture et/ou en écriture !)

Méthodologie de la mémoire partagée en langage C

- Définir une structure qui représentera les objets partagés (par exemple : un entier + un char +...)
- Créer un descripteur via la primitive open()
- Appeler ftruncate() entre le descripteur et la taille de la structure ce qui réserve de la place
- Mettre en correspondance (via mmap()) le descripteur et la structure. mmap() renvoie un pointeur sur la zone partagée ;
- Utiliser le pointeur pour accéder aux champs de la structure et ceci pour chacun des processus
- dé-allouer (via munmap()) et fermer le descripteur ouvert par open avec un close()

Les tubes (tuyaux)

- Ils peuvent être vus comme des files (first in, first out) : un processus écrit à un bout du tube ; un autre lit à l'autre extrémité
- Potentiellement des problèmes d'accès concurrents (si plusieurs processus lisent concurremment un tube, un seul lit la valeur qui est enlevée du tuyau).
- Très simple à gérer si un seul processus écrit, un deuxième lit !

Les tubes

- Un tube est implémenté comme un fichier sauf qu'il ne possède pas de nom sur le disque
- Les communications qui transitent par lui sont uni-directionnelles
- Un tube a une taille limitée : si un processus essaye d'écrire dans un tube plein alors il est suspendu. La capacité d'un tube est fixée par une constante dans le noyau.

Les tubes

- Création : `int pipe(int p[2]);`
- Correspondont à la création de deux descripteurs de fichiers, l'un pour écrire (via `write`), l'autre pour lire via (`read` par exemple)
- p[0] = descripteur par lequel on peut lire ; p[1] = descripteur par lequel on peut écrire
- Si on fait un fork() après le pipe(). Les descripteurs sont connus du père et du fils, donc père/fils peuvent communiquer

Les tubes : contrôle d'accès

- La seule façon de placer une fin de fichier dans un tube est de fermer celui-ci en écriture dans tous les processus qui l'utilise (`close(p[1])`)
- La primitive `read` renvoie 0 lorsque la fin de fichier est atteinte c.à.d si le tube est vide ET si tous les processus qui utilisent ce tube l'ont fermé en écriture

Algorithmes des Read et des Writes

Lecture

```
si tube non vide alors
   nb_lu = min{nb_dem, contenu tube};
sinon
   s'il existe un écrivain alors
       si lecture bloquante alors
          suspension jusqu'a écriture;
       sinon
          nb_lu = -1;
       finsi
   sinon
       fin de fichier => nb_lu = 0
   finsi
finsi
```

Ecriture

```
// Principe : pour écrire on doit avoir
// un récepteur pret à recevoir
s'il n'existe pas de lecteur alors
   envoie de SIGPIPE (terminaison par défaut)
   au processus ecrivain
sinon
   /* il existe au moins un lecteur */
   si pas de place alors
      si ecriture bloquante alors
         bloquage jusqu'a place dispo
      sinon
         -1 ou < nb_dem si nb_dem > PIPE_BUF
      finsi
   sinon
      nb_ecrit = nb_dem
      /* cad ecriture indivisible */
   finsi
finsi
```

Les tubes simples : résumé

- Les opérations de lecture et d'écriture se font avec les mêmes primitives que pour un fichier (read et write) mais le noyau ajoute en interne une **synchronisation**. Ainsi, un tube a la structure d'un modèle de type producteur/consommateur classique avec en particulier :
  - une lecture FIFO
  - une lecture destructrice (l'information lue est retirée du tube)
  - une lecture bloquante lorsque le tube est vide
  - une écriture bloquante lorsque le tube est saturé

Les tubes simples : résumé

La principale restriction liée au tube est qu'il n'est accessible qu'au processus qui l'a créé et à sa descendance c'est-à-dire aux processus fils créés par FORK. De plus, il disparaît à la mort de la famille du processus qui l'a créé et des risques de blocage existent en raison des opérations bloquantes.

```
void main () {
  int df[2]; // descripteur du tube
  char c, d; // caractère de l'alphabet
  int ret;

  pipe(df); // On crée le tube
  if (fork() != 0) { // On est dans le père
     close (df[0]); // Le père ne lit pas
     for (c = 'a'; c <= 'z'; c++) {
       write(df[1], &c, 1); //Père écrit dans tube
     }
    close(df[1]); // On ferme le tube en écriture
    wait(&ret); // Le père attend le fils
  } else {
   close(df[1]); // On ferme le tube en écriture
   while (read(df[0], &d, 1) > 0) {
      // Fils lit tube
      printf("%c\n", d);// Il écrit le résultat
   }
   close (df[0]); // On ferme le tube en lecture
}}
```

Complément : les tubes nommés

- Ce sont des tubes simples sauf qu'ils ont un nom (rémanent)
- Le nom est créé par la primitive `mknod` ou à partir de la ligne de commande Unix avec `mkfifo` : le père et le fils doivent connaître le nom pour pouvoir faire le open ;
- Permmetent à des processus (programmes compilés) sans lien de parenté de communiquer
- Utilisation : faire communiquer 2 exécutables (par exemple, un exécutable écrit dans le tube, l'autre lit le tube)

Tubes nommés (open)

- Par défaut bloquante (rendez-vous :
  - Une demande d'ouverture en lecture est bloquante s'il n'y a aucun écrivain sur le tube ;
  - Une demande d'ouverture en écriture est bloquante s'il n'y a aucun lecteur sur le tube ;
- Il y a aussi une option pour obtenir des ouvertures non bloquantes O_NONBLOCK

Example : écrivain / lecteur

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define S_BUF 100
int n;cchar buffer[S_BUF];
intcmain (int argc, char **argv)
{
  int fd_write;
  if (mkfifo (argv[1], S_IRUSR | S_IWUSR)
    == -1){
      perror ("mkfifo");
      exit (1);}
  if ((fd_write = open (argv[1],
    O_WRONLY)) == -1){
      perror ("open");
      exit (1);}
  if ((n = write (fd_write, "Bonjour", 7))
    == -1){
      perror ("write");
      exit (1);}
  close (fd_write);
  return EXIT_SUCCESS;
}
```

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define S_BUF 100
int n;
char buffer[S_BUF];
int main (int argc, char **argv)
{
  int fd_read;
  if ((fd_read = open (argv[1], O_RDONLY)) == -1) {
      perror ("open");
      exit (1);}
  if ((n = read (fd_read, buffer, S_BUF)) == -1){
      perror ("read");
      exit (1);
    } else {
      buffer[n] = '\0';
      printf ("%s\n", buffer);
    }
  close (fd_read); remove (argv[1]);
  return EXIT_SUCCESS;
}
```

Interlude : programmation Python et gestion de processus

- Il y a plusieurs interfaces de programmation en Python pour gérer les processus... et les interfaces continuent à évoluer (cf Python 3.2 : http://docs.python.org/dev/whatsnew/3.2.html#threading)
- Nous allons regarder que la partie liée aux processus système (les «gros processus» observables par `top` sous Linux) :
  - Exemple de problème à regarder : comment se fait le partage de variables ?

Exemple introductif pour faire du lien avec ce qui a été vu avec C

```
#!/usr/bin/env python
"""A basic fork in action"""
import os
def my_fork():
    child_pid = os.fork()
    if child_pid == 0:
       print "Child Process: PID# %s" % os.getpid()
    else:
       print "Parent Process: PID# %s" % os.getpid()

if __name__ == "__main__":    my_fork()

$ python fork.py
Parent Process: PID# 8357
Child Process: PID# 8358
```

Comme avec C : pas de partage

```
#!/usr/bin/env python
"""A fork that modifies a variable """
import os
from os import environ

def my_fork():
    FOO="baz"
    print "FOO variable set to: %s" % FOO
    child_pid = os.fork()
    if child_pid == 0:
       FOO="bar"
       print "Child Process: PID# %s" % os.getpid()
       print "Child FOO variable == %s" % FOO
    else:
       print "Parent Process: PID# %s" % os.getpid()
       os.wait()
       print "Parent FOO variable == %s" % FOO

if __name__ == "__main__":
   my_fork()

$ python fork1.py
FOO variable set to: baz
Parent Process: PID# 8453
Child Process: PID# 8454
Child FOO variable == bar
Parent FOO variable == baz
```

Du partage avec mmap aussi !

- L'interface mmap s'utilise comme en C c.à.d via des descripteurs et des primitives d'écriture/lecture et ouverture/fermeture sur le disque (http://docs.python.org/library/mmap.html)
- mmap a un paramètre qui spécifie un mode d'accès à la mémoire partagée : ACCESS_READ, ACCESS_WRITE, or ACCESS_COPY to specify read-only, write-through (écriture immédiate sur le disque) or copy-on-write memory (écriture dans le buffer mais pas immédiatement dans le fichier) respectively.

Exemple d'usage de mmap()

```
import mmap
import os
map = mmap.mmap(-1, 13)
map.write("Hello world!")
pid = os.fork()
if pid == 0: # In a child process
    map.seek(0)
    print map.readline()
    map.close()
```

Exemple d'usage de mmap() : écrivain

```
#!/usr/bin/env python
import ctypes
import mmap
import os
import struct


def main():
    # Create new empty file to back memory map on disk
    fd = os.open('/tmp/mmaptest', os.O_CREAT | os.O_TRUNC | os.O_RDWR)

    # Zero out the file to insure it's the right size
    assert os.write(fd, '\x00' * mmap.PAGESIZE) == mmap.PAGESIZE

    # Create the mmap instace with the following params:
    # fd: File descriptor which backs the mapping or -1 for anonymous mapping
    # length: Must in multiples of PAGESIZE (usually 4 KB)
    # flags: MAP_SHARED means other processes can share this mmap
    # prot: PROT_WRITE means this process can write to this mmap
    buf = mmap.mmap(fd, mmap.PAGESIZE, mmap.MAP_SHARED, mmap.PROT_WRITE)

    j = 0
    offset = 0
```

Exemple Mmap Python (écrivain)

```
    while j<10:
        # Now create an int in the memory mapping
        i = ctypes.c_int.from_buffer(buf,offset)

        # Set a value
        i.value = j

        # Before we create a new value, we need to find the offset of the next free
        # memory address within the mmap
        offset = (j+1)*struct.calcsize(i._type_)

        if j+1 != 10:
            # The offset should be uninitialized ('\x00')
            assert buf[offset] == '\x00'

        # increase the offset
        j = j+1

if __name__ == '__main__':
    main()
```

Exemple Mmap Python (lecteur)

```
import mmap
import os
import struct

def main():
    # Open the file for reading
    fd = os.open('/tmp/mmaptest', os.O_RDONLY)

    # Memory map the file
    buf = mmap.mmap(fd, mmap.PAGESIZE, mmap.MAP_SHARED,mmap.PROT_READ)

    i = 0

    while i<10:
        new_i, = struct.unpack('i', buf[i*4:i*4+4])
        print new_i
        i = i+1
    os.remove('/tmp/mmaptest')

if __name__ == '__main__':
    main()
```

Les tubes simples en Python (Linux)

```
import os, time

def child(pipeout):
    zzz = 0
    while 1:
        time.sleep(zzz)                                  # make parent wait
        os.write(pipeout, 'Spam %03d' % zzz)             # send to parent
        zzz = (zzz+1) % 5                                # goto 0 after 4

def parent( ):
    pipein, pipeout = os.pipe( )                 # make 2-ended pipe
    if os.fork( ) == 0:                          # copy this process
        child(pipeout)                           # in copy, run child
    else:                                        # in parent, listen to pipe
        while 1:
            line = os.read(pipein, 32)           # blocks until data sent
            print 'Parent %d got "%s" at %s' %
                   (os.getpid(), line, time.time( ))

parent( )
```

```
$ python pipe.py
Parent 34025 got "Spam 000" at 1333781709.48
Parent 34025 got "Spam 001" at 1333781710.48
Parent 34025 got "Spam 002" at 1333781712.48
Parent 34025 got "Spam 003" at 1333781715.48
Parent 34025 got "Spam 004Spam 000" at 1333781719.48
Parent 34025 got "Spam 001" at 1333781720.48
Parent 34025 got "Spam 002" at 1333781722.48
Parent 34025 got "Spam 003" at 1333781725.49
Parent 34025 got "Spam 004Spam 000" at 1333781729.49
Parent 34025 got "Spam 001" at 1333781730.49
```

Les tubes simples en Python (Linux)

```
import os, time

def child(pipeout):
    zzz = 0
    while 1:
        time.sleep(zzz)                              # make parent wait
        os.write(pipeout, 'Spam %03d' % zzz)         # send to parent
        zzz = (zzz+1) % 5                            # goto 0 after 4

def parent( ):
    pipein, pipeout = os.pipe( )                 # make 2-ended pipe
    if os.fork( ) == 0:                          # copy this process
        child(pipeout)                           # in copy, run child
    else:                                        # in parent, listen to pipe
        while 1:
            line = os.read(pipein, 32)           # blocks until data sent
            print 'Parent %d got "%s" at %s' %
                   (os.getpid(), line, time.time( ))

parent( )
```

```
$ python pipe.py
Parent 34025 got "Spam 000" at 1333781709.48
Parent 34025 got "Spam 001" at 1333781710.48
Parent 34025 got "Spam 002" at 1333781712.48
Parent 34025 got "Spam 003" at 1333781715.48
Parent 34025 got "Spam 004Spam 000" at 1333781719.48
Parent 34025 got "Spam 001" at 1333781720.48
Parent 34025 got "Spam 002" at 1333781722.48
Parent 34025 got "Spam 003" at 1333781725.49
Parent 34025 got "Spam 004Spam 000" at 1333781729.49
Parent 34025 got "Spam 001" at 1333781730.49
```

Les tubes simples en Python (Linux)

```
# same as pipe.py, but wrap pipe input in stdio file object
# to read by line, and close unused pipe fds in both processes

import os, time

def child(pipeout):
    zzz = 0
    while 1:
        time.sleep(zzz)                          # make parent wait
        os.write(pipeout, 'Spam %03d\n' % zzz)   # send to parent
        zzz = (zzz+1) % 5                        # roll to 0 at 5

def parent( ):
    pipein, pipeout = os.pipe( )      # make 2-ended pipe
    if os.fork( ) == 0:               # in child, write to pipe
        os.close(pipein)              # close input side here
        child(pipeout)
    else:                             # in parent, listen to pipe
        os.close(pipeout)             # close output side here
        pipein = os.fdopen(pipein)    # make stdio input object
        while 1:
            line = pipein.readline( )[:-1] # blocks until data sent
            print 'Parent %d got "%s" at %s' % (os.getpid(),line,time.time( ))

parent( )
```

Les tubes nommés en Python

```
#########################################################
# named pipes; os.mkfifo not available on Windows 95/98/XP
# (without Cygwin); no reason to fork here, since fifo file
# pipes are external to processes--shared fds are irrelevent;
#########################################################

import os, time, sys
fifoname = '/tmp/pipefifo'                          # must open same name

def child( ):
    pipeout = os.open(fifoname, os.O_WRONLY)        # open fifo pipe file as fd
    zzz = 0
    while 1:
        time.sleep(zzz)
        os.write(pipeout, 'Spam %03d\n' % zzz)
        zzz = (zzz+1) % 5

def parent( ):
    pipein = open(fifoname, 'r')                    # open fifo as stdio object
    while 1:
        line = pipein.readline( )[:-1]              # blocks until data sent
        print 'Parent %d got "%s" at %s' % (os.getpid(), line, time.time( ))

if __name__ == '__main__':
    if not os.path.exists(fifoname):
        os.mkfifo(fifoname)                          # create a named pipe file
    if len(sys.argv) == 1:
        parent( )                                    # run as parent if no args
    else:                                       # else run as child process
        child( )

# COMMENT LANCER CE PROGRAMME ?
#
#
```

Les signaux

Au moins 2 interfaces !

de gestion

- Les signaux informent un processus d'un événement de façon asynchrone
- Peuvent aussi servir à la synchronisation
- La liste des signaux, ainsi que les différentes fonctions de manipulation des signaux sont définies dans le fichier **/usr/include/signal.h** (Dans les SYSTEM V) et **/usr/include/bits/signum.h** (Dans les versions linux et sont un peu différents).

Liste (partielle) des 32 signaux

Numéro	Nom	Signification
•	SIGHUP	Fin de session
•	SIGINT	Interruption
•	SIGQUIT	Instruction
•	SIGILL	Instruction illégale
•	SIGTRAP	Trace
•	SIGABRT	Instruction IOT / abort
…		
10	SIGUSR1	Signal pour utilisateurs
12	SIGUSR2	Signal pour utilisateurs

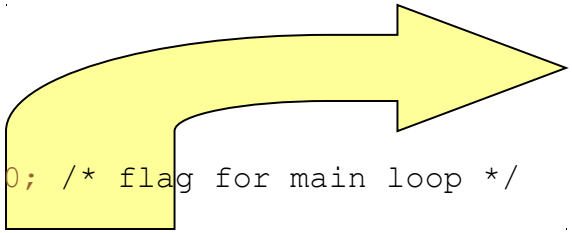
A quoi servent les signaux ?

- à avertir/capter un événement bas niveau MAIS pas vraiment à échanger des données
- `int kill(int pid, int signal)` permet à un processus d'envoyer un signal à un autre processus (... et donc de se synchroniser par un Rendez-vous : non présenté ici)
- kill() rend 0 en cas de succès et -1 en cas d'échec.

Utilisation des signaux

- Un processus peut détourner les signaux reçus et modifier son comportement par l'appel de la fonction : `void *signal(int signal, void (*fonction)(int))`
- Le dernier argument est un pointeur sur une fonction qui implémente le nouveau comportement.

Exemple introductif



Cette fonction ne peut pas renvoyer de valeur, par def

```
#include <signal.h>

static int signal_flag = 0; /* flag for main loop */
void sig_int(int signo)
{
  if (signal(SIGINT, sig_int)== SIG_ERR)/* reestablish  */
    exit(1);                            /* for next time */
  printf("Received signal %d\n", signo);
  signal_flag = 1; /* set flag for main loop */
}

int main(void)
{
  if (signal(SIGINT, sig_int) == SIG_ERR) {
    perror("Cannot catch SIGINT");
    exit(1);
  }
  while (1) {
    while(!signal_flag)/* enter main loop:   */
      pause();         /* sleep until a signal is received */
    printf("Received SIGINT\n");
    signal_flag=0;
  }
 return 0;
}
```

Sous le shell faire un
kill -SIGINT <num_process>

Le problème de l'exclusion mutuelle

- Plusieurs processus veulent accéder à une ressource partagée : comment faire en sorte qu'au plus un processus acquiert la ressource ?
- Exemple : imprimante. On ne veut pas voir sortir une page de monsieur X puis 2 pages de Madame Y puis 2 pages de monsieur X...
- Il y a des solutions matérielles et des solutions logicielles

Exclusion mutuelle

- Dans un des TP en ligne il y a du code qui implémente deux opérations : P() et V() qui permettent de délimiter une section critique (SC)
- Le code dans la section critique est tel qu'au plus un processus l'exécute ;
- Rappel de contexte : plusieurs processus ont été 'forkés' : ils exécutent donc le même code. P() et V() permettent de laisser rentrer dans la SC au plus un processus !

Exclusion mutuelle

- P() et V() implémentent de la lecture / ecriture / test sur une variable partagée
- In computer science, the test-and-set instruction is an instruction used to both test and (conditionally) write to a memory location as part of a single atomic (i.e. non-interruptible) operation. This means setting a value, but first performing some test (such as, the value is equal to another given value). If the test fails, the value is not set. If multiple processes may access the same memory, and if a process is currently performing a test-and-set, no other process may begin another test-and-set until the first process is done. CPUs may use test-and-set instructions offered by other electronic components, such as Dual-Port RAM; CPUs may also offer a test-and-set instruction themselves.(WIKIPEDIA)
- http://en.wikipedia.org/wiki/Test-and-set

Exclusion mutuelle via test & set

```
boolean lock = false
function Critical(){
    while TestAndSet(lock)
        skip //spin until lock is acquired
    //critical section
    //only one process can be in
    // this section at a time
    lock = false//release lock when
                //finished
                 //with the critical section
}
```

ICI on implémente de l'attente active

Exclusion mutuelle et fetch & add

- In computer science, the fetch-and-add CPU instruction is a special instruction that atomically modifies the contents of a memory location. It is used to implement Mutual exclusion and concurrent algorithms in multiprocessor systems.
- http://en.wikipedia.org/wiki/Fetch-and-add

Fetch & add

```
<< atomic >>
function FetchAndAdd(address location) {
    int value := *location
    *location := value + 1
    return value
}
```

With fetch-and-add primitive a mutual exclusion lock can be implemented as:

```
 record locktype {
    int ticketnumber
    int turn
 }
```

Fetch & add

```
procedure LockInit( locktype* lock ) {
    lock.ticketnumber := 0
    lock.turn  := 0
 }
 procedure Lock( locktype* lock ) {
    int myturn :=
         FetchAndAdd( &lock.ticketnumber )↰
    while lock.turn ≠ myturn
        skip //spin until lock is acquired
 }
 procedure UnLock( locktype* lock) {
    FetchAndAdd( &lock.turn )↰
 }
```

X86 synchronization primitives

The simplest is the XCHG instruction, which can be used to atomically exchange two registers or a register and a memory location. This makes it possible to implement multiple exclusion; reserve a particular location in RAM as a mutex, and initially set it to 1. To acquire the mutex, set a register to 0, and XCHG it with the location in RAM. If what you get back is a 1, then you have successfully acquired the mutex; otherwise, someone else has it. You can return the mutex simply by setting the location to a nonzero value.

Notre histoire : Dijkstra

http://www.cs.utexas.edu/users/EWD/welcome.html

- C'est au début des année 60 que l'on a commencé à formaliser la notion de processus coopérants (concurrents)
- Disjkstra gère l'accès exclusif à la section critique au moyen d'une variable commune aux deux processus (variable turn). L'auteur fait de plus l'hypothèse que l'examen de la valeur de cette variable ainsi que son affectation se fait de manière indivisible : si deux processus tentent de modifier la variable on voit deux affectations se déroulant séquentiellement

Dijkstra

```
begin
integer turn; turn := 1;
parbegin
   process 1: begin L1: if turn=2 then goto L1;
                        critical section 1;
                        turn:=2;
                        remainder of cycle 1;
                       goto L1
   end;
   process 2: begin L2: if turn=1 then goto L2;
                        critical section 2;
                        turn:=1;
                        remainder of cycle 2;
                       goto L2
   end;
parend;
end;
```

Dans quel ordre entre les processus s'effectue nécessairement l'exécution de ce programme ?

On veut éviter la séquentialisation. Le code suivant est faux !

```
integer c1,c2;
c1 := 1; c2 := 1;
parbegin
   process 1: begin
                  L1: if c2=0 then goto L1;
                      c1:=0;
                      critical section 1;
                      c1:=1;
                      remainder of cycle 1;
                      goto L1
   end;
   process 2: begin
                   L2: if c1=0 then goto L2;
                       c2:=0;
                       critical section 2;
                       c2:=1;
                       remainder of cycle 2;
                       goto L2
   end;
parend;
```

Justifications

Question : expliquez comment dans ce code on évite d'avoir les deux processus simultanément en section critique.

Les deux premières affectations sur c1 et c2 sont conformes avec le fait qu'on demande à ce que les processus commencent leurs exécutions en dehors de la section critique. Pendant l'exécution de la section critique c1 ou c2 vaut 0 et la boucle sur l'étiquette L1 ou L2 permet la protection d'accès à la section critique.

Question : montrer qu'en fait le code précédent est faux.

Supposons que le processus 1 trouve c2=1. Supposons de plus que le processus 2 examine c1 immédiatement après. Alors il va encore trouver c1=1... ce qui veut dire que les 2 processus peuvent entrer en section critique.

Nouveau raffinement

```
begin
integer c1,c2;
c1 := 1; c2 := 1;
parbegin
    process 1: begin A1: c1:=0;
                         L1: if c2=0 then goto L1;
                         critical section 1;
                         c1:=1;
                         remainder of cycle 1; goto A1
                end;
    process 2: begin A2: c2:=0;
                         L2: if c1=0 then goto L2;
                         critical section 2;
                         c2:=1;
                         remainder of cycle 2; goto A2
                end;
parend;
```

Emanen critique de la solution

Vérifions que cette solution est « safe ». Examinons le moment où le processus 1 trouve c2=1 et donc décide de rentrer en section critique. A ce moment on peut conclure :

1. que la relation c1=0 est aussi vraie à ce moment et le restera jusqu'à ce que le processus 1 ait terminé l'exécution de sa section critique ;
2. que, puisque c2=1 est vraie, le processus 2 est bien en dehors de sa section critique et il ne peut y entrer tant que c1=1 est vérifiée i.e. tant que le processus 1 est engagé dans sa section critique.

Ainsi, l'exclusion mutuelle est garantie.

Cependant cette solution contient un défaut. Monter qu'il y a au moins une situation où l'on est en situation de blocage mutuel.

Examinons la situation juste après (dans le processus 1) ou l'on a fait c1:=0 et juste avant l'examen de la valeur c2 (toujours dans le processus 1). Le processus 2 exécute c2:=0, alors les deux processus arrivent sur la relation c1=0 pour l'un et c2=0 pour l'autre... ce qui conduit à boucle sur les étiquettes L1 et L2 et donc les deux processus attendent indéfiniment !

En fait, c'est au mathématicien hollandais Th. J. Dekker que l'on doit la première version correcte du problème. La voici :

Algorithme de Dekker

```
begin
integer c1,c2,turn;
c1 := 1; c2 := 1; turn:=1;
parbegin
     process 1: begin A1: c1:=0;
                          L1: if c2=0 then begin
                                    if turn=1 then goto L1;
                                    c1:=1
                                    B1: if turn=2 then goto B1;
                                    goto A1;
                          end;
                           critical section 1;
                           turn:=2; c1:=1;
                            remainder of cycle 1; goto A1;
                end;
     process 2: begin A2: c2:=0;
                          L2: if c1=0 then begin
                                    if turn=2 then goto L2;
                                    c2:=1;
                                    B2: if turn=1 then goto B2;
                                     goto A2;
                           end;
                           critical section 2;
                            turn:=1; c2:=1;
                            remainder of cycle 2; goto A2;
                end;
parend;
end;
```

Preuve de l'algorithme de Dekker

A partir de l'obversation que chaque processus ne modifie que son propre c, argumentez sur la correction de cet algorithme.

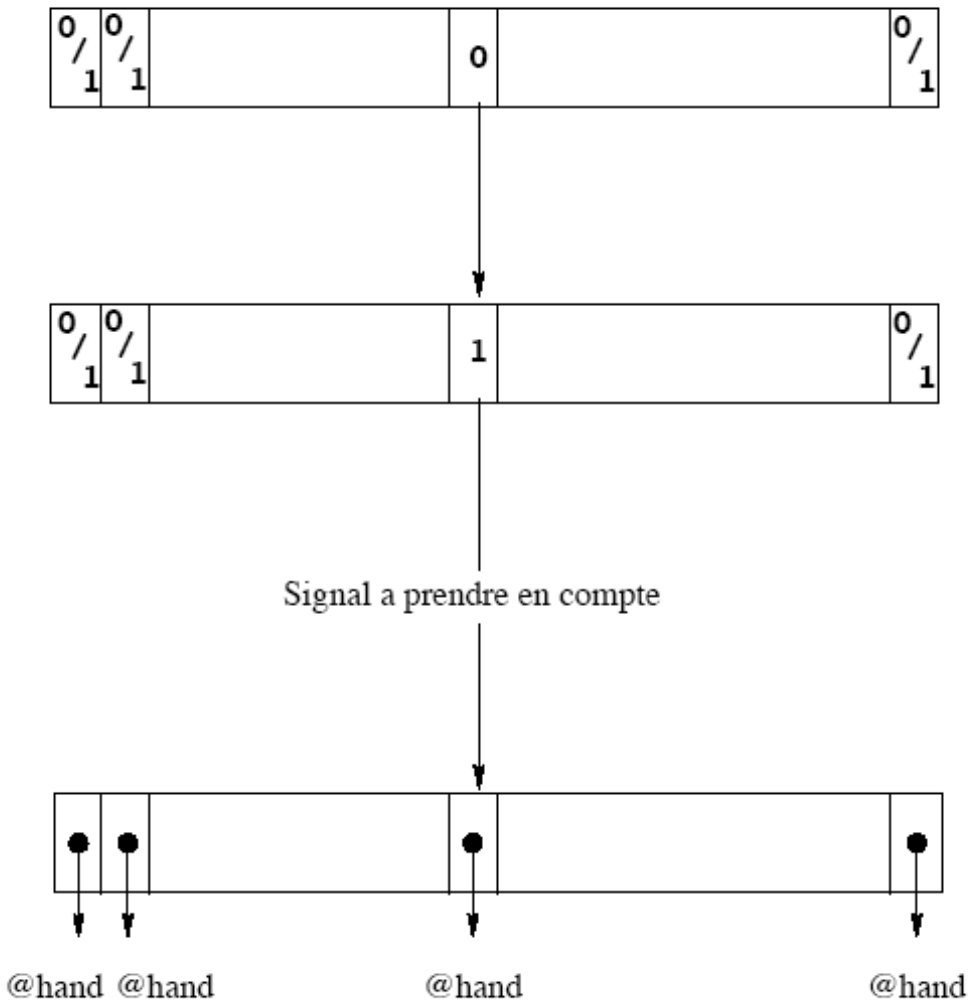
La conséquence de l'observation précédente, le processus 1 inspecte c2 seulement quand c1=0. Il entrera en section critique s'il trouve c2=1. Pour le processus 2, c'est un raisonnement analogue. Concernant la variable turn, on observe qu'elle est modifiée après les sections critiques, c'est à dire après la prise de décision d'entrer en section critique.Supposons que turn=1. Alors le processus 1 ne peut avancer que vers L1, c'est à dire avec c1=0 et ceci tant que c2=0. Mais si turn=1, alors le processus 2 peut avaner dans le cycle B2, mais cet état implique que c2=1 et donc le processus 1 ne peut pas boucler et il passe en secton critique Pour turn=2, un raisonnement analogue s'applique. Pour compléter la preuve, on remarque enfin qu'en exécutant, disons la partie « remainder of cycle 1 », nous ne causons aucun préjudice : la relation c1=1 est alors vraie et le processus 2 peut alors entrer en section crtique. Et ceci de manière indépendante de la valeur de turn.

ANNEXES

Annexe: fonctionnement détaillé et structure interne des processus

- Le mécanisme de prise en compte des signaux se situe dans le bloc de contrôle d'un processus
- Il y a 3 vecteurs de NSIG bits si le système peut disposer de NSIG signaux
- Le bloc de contrôle permet de prendre en compte un signal délivré au processus et d'associer cette réception à une action.

Les vecteurs d'état



Vocabulaire

- **Délivrance d'un signal :** cela correspond à avoir le bit correspondant à ce signal à 1 dans le vecteur des signaux reçus
- **Masquage d'un signal :** cela correspond à positionner un 1 dans le bit correspondant à ce signal dans le vecteur des signaux bloqués (ou masque). Pour prendre en compte un signal on effectue un ET entre le vecteur des signaux reçus et le masque. Si le résultat est 0, on n'exécute pas la fonction de déroutement (handler)

Vocabulaire

- Un signal est pendant : envoyé au processus mais non encore pris en compte ;
- Un signal est délivré : il est pris en compte. Le processus réalise le handler ;
- Un signal est bloqué : la prise en compte du signal est différée jusqu'à ce que le signal ne soit plus bloqué (SIGKILL, SIGSTOP et SIGCONT ne peuvent pas être bloqués)

Vocabulaire

- Un signal est ignoré : le signal est délivré, mais le bit dans le bloc de contrôle est remis à 0. Pas de handler réalisé ;
- Un signal est masqué : même chose que bloqué sauf que le bit correspondant au signal est à 1 dans le masque ;

De bloqué à ignoré à …

- La notion d'ensemble de signaux permet de masquer… les signaux
- Un ensenble de signaux est représenté par un type `sigset_t`
- Les primitives de gestion :

Fonction	Effet
`sigmask(n)`	Donne le masque du signal n
`sigemptyset(S)`	$S = \emptyset$
`sigaddset(S,n)`	$S = S \cup n$
`sigdelset(S,n)`	$S = S - n$
`sigismember(S,n)`	Vrai ssi $n \in S$
`sigorset(S1,S2)`	$S1 = S1 \cup S2$
`sigandset(S)`	$S1 = S1 \cap S2$
`sigdiffset(S)`	$S1 = S1 - S2$

Gestion des masques

Définition :

```
#include <signal.h>

int   sigprocmask( int                 op,
                   const sigset_t    * p_ens,
                   sigset_t          * p_ens_ancien );
```

Paramètres :

- *Données* :
  - `op` : operation à faire sur `M` = masque du processus appelant
    - `SIG_BLOCK` → `M` = `M` + `p_ens`
    - `SIG_UNBLOCK` → `M` = `M` - `p_ens`
    - `SIG_SETMASK` → `M` = `p_ens`
  - `p_ens` : pointeur sur une structure qui contient le masque à prendre en compte. Si ce pointeur est `NULL`, alors la primitive est sans effet.
- *Résultat* :
  - `p_ens_ancien` : pointeur sur une structure dans laquelle la primitive affecte le masque qui existait avant l'appel. Si ce pointeur était à `NULL` avant l'appel, on ne récupère pas la valeur antérieure du masque.

Retour :

- 0 en cas de succès
- -1 en cas d'échec

Gestion des masques (suite)

- `sighold` : bloque un signal donné
- `sigrelse` : débloque un signal donné
- `sigpause` : débloque et met en attente de réception d'un signal
- `sigsuspend` : installe un nouveau masque puis attend l'arrivée d'un signal non masqué. Le masque initial est restauré au retour.
- `sigpending` : donne l'ensemble des signaux pendants

Gestion des masques (suite)

- `sigset` : fonctionne comme `signal` mais en plus elle masque le signal capté ;
- La norme POSIX propose la primitive `sigaction` regroupant l'ensemble des fonctionnalités des différentes primitives de manipulation des handlers ;

Exemple utilisation sigaction

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>

static void hand_int(int sig)
{
  static int n=0;

  n++;
  printf("Dans hand_int n=%d et sig=%d\n",n,sig);
  sleep(2);
  kill(getpid(), SIGUSR1);
  kill(getpid(), SIGUSR2);
  sleep(3);
  printf("Sortie hand_int n=%d\n",n);
}

static void hand_usr1(int sig)
{
  static int n = 0;

  n++;
  printf("Dans hand_usr1 n=%d et sig=%d\n",n,sig);
  sleep(3);
  printf("Sortie hand_usr1 n=%d\n",n);
}
```

Exemple utilisation sigaction

```
static void hand_usr2(int sig)¶
{
  static int n = 0;

  n++;
  printf("Dans hand_usr2 n=%d et sig=%d\n",n,sig);
  sleep(5);
  printf("Sortie hand_usr2 n=%d\n",n);

}

static void hand_quit(int sig)¶
{
  static int n = 0;

  n++;
  printf("Dans hand_quit n=%d et sig=%d\n",n,sig);
  sleep(3);
  printf("Sortie hand_quit n=%d\n",n);

}
```

Exemple utilisation sigaction

```
int main()¶
{
  int n=0;
  char c;
  struct sigaction action;

  /* Capture du signal INT */
  /* L'appel systeme sera repris au retour du handler */
  /* grace a l'option SA_RESTART */
  action.sa_flags = SA_RESTART;
  action.sa_handler = hand_int;
  sigaddset(&action.sa_mask, SIGUSR1);
  sigaction(SIGINT, &action, NULL);

  /* Capture du signal USR1
  // SA_RESETHAND : Restore the signal action to the
  //default state once the signal handler has been called
  //(equivalent a SA_ONESHOT)¶
  // SA_NODEFER : Do not prevent the signal from being
  // received from within its own signal handler
  // (equivalent SA_NOMASK) */
  action.sa_flags = SA_RESETHAND | SA_NODEFER;
  action.sa_handler = hand_usr1;
  sigemptyset(&action.sa_mask);
  sigaction(SIGUSR1, &action, NULL);
```

Exemple utilisation sigaction

```
/* Capture du signal USR2 */
  action.sa_flags = 0 ;
  action.sa_handler = hand_usr2;
  sigemptyset(&action.sa_mask);
  sigaction(SIGUSR2, &action, NULL);

  /* Capture du signal QUIT */
  action.sa_flags = 0 ;
  action.sa_handler = hand_quit;
  sigemptyset(&action.sa_mask);
  sigaction(SIGQUIT, &action, NULL);

  printf("Entrez n\n");
  n = read(0,&c,1);
  printf("Valeur lue pour n=%d\n",n);

  kill(getpid(),SIGUSR1);

  while(1);
  //exit(0);

}
```

Exemple utilisation sigaction

```
/*
Ordinateur-de-Christophe-Cerin:~ cerin$ ./a.out
Entrez n
^CDans hand_int n=1 et sig=2  =>j'ai tape ctrl C pendant le read()↰
Dans hand_usr2 n=1 et sig=31
Sortie hand_usr2 n=1
Sortie hand_int n=1
Dans hand_usr1 n=1 et sig=30
Sortie hand_usr1 n=1
⇨j'ai tape kill -QUIT 523 sur la ligne de commande
apres avoir verifie que le pid de a.out etait 523
Dans hand_quit n=1 et sig=3
Sortie hand_quit n=1 Valeur lue pour n=-1
User defined signal 1
Ordinateur-de-Christophe-Cerin:~
```

Explications :

- Dans le main, on installe les differents signaux SIGINT, SIGUSR1, SIGUSR2 et SIGQUIT

Exemple utilisation sigaction

```
b) au premier ctrl C (le systeme attend son read()), le signal est capte et
l'appel systeme read est interompu ; on renvoie -1... mais apres l'execution de
differents handlers : il est repris au retour du handler grace a l'option
SA_RESTART.

c) Dans le handler hand_int, on envoie les signaux SIGUSR1 et SIGUSR2. Mais SIGUSR1
est masque (a cause de l'instruction sigaddset(&action.sa_mask, SIGUSR1);). On
execute donc uniquement hand_usr2

d) quand hand_usr2 se termine, on reprend la ou on s'etait arrete dans hand_int…
qui se termine : hand_usr1 peut alors s'executer. A la sortie de hang_usr1 le
signal par defaut est reinstalle (option SA_RESETHAND), et donc l'appel systeme
read() peut reprendre.

e) a la reception du signal QUIT (par la commande kill -QUIT), le handler hang_quit
est active mais au retour du handler l'appel systeme read() n'est pas repris car
on n'avait pas mis l'option SA_RESTART et on sort du read() avec la valeur -1
(par defaut)

f) Pour finir, le processus renvoie le signal SIGUSR1 qui active son handler par
defaut (terminaison avec affichage de "User defined signal 1") parce que dans le
main on a fait kill(getpid(),SIGUSR1);
Notez que c'est seulement apres le signal QUIT que le read() se termine
("en echec") et que le kill(getpid(),SIGUSR1); peut s'executer... et faire
sortir du while(1);

*/
```