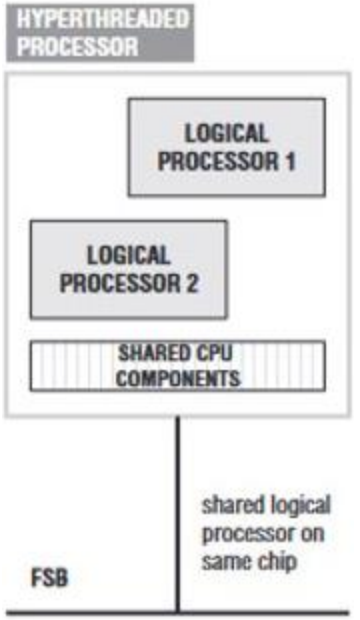
Multi-programmation

Malek.Bengougam@gmail.com

Partie 1 - bases

Architecture matérielle

Simultaneous Multi Threading (SMT)



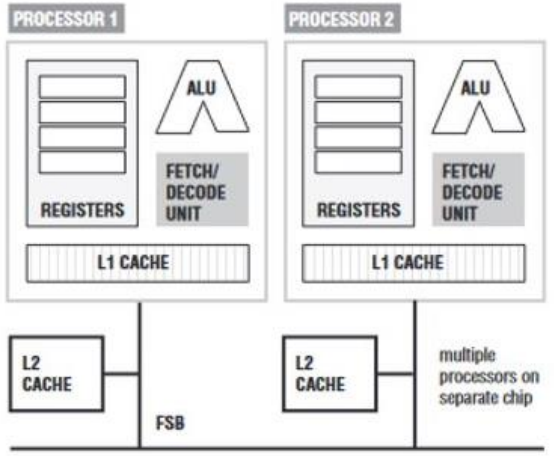
- notion de coeurs logiques
  - seules certaines unités de calcul sont dupliquées
  - les pipelines, caches etc... sont eux partagés

- Le SMT permet d'avoir deux fils d'exécution matériel (hardware thread)
  - Le SMT est appelé "hyper threading" dans les architectures Intel
    - Introduit avec le Pentium4 HT, abandonné avec les core2 puis de nouveau utilisé de nos jours

- L'exécution est temporelle (temporal multithreading) et non pas parallèle
- **fine-grained** : chaque coeur logique est exécuté durant 1 cycle par le processeur
- **coarse-grained** : chaque coeur logique est exécuté durant plusieurs cycles

  - Ne pas confondre ces notions spécifiques à l'implémentation du SMT avec le fonctionnement du multitâche du système d'exploitation.

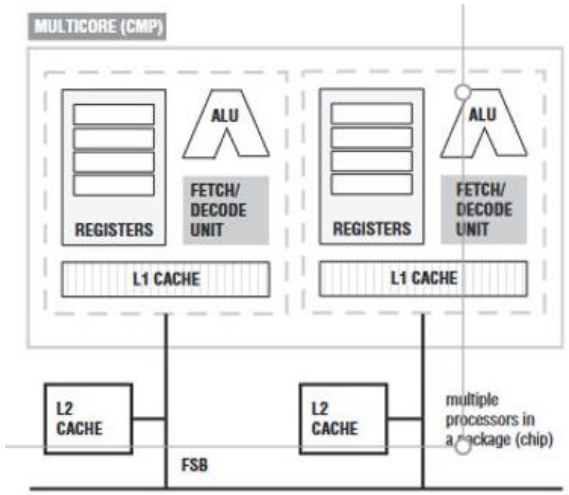
Symmetric Multiple Processors (SMP)



- plusieurs processeurs en parallèle
  - Exemples:
    - Sega Saturn = 2x Hitachi SH-2 (RISC)
- Généralement ces processeurs sont identiques et connectés à la même mémoire via un bus ou un système spécifique (crossbar...)
- Le fonctionnement est dit symétrique
- Les problèmes sont les mêmes que ceux des processeurs single-core à savoir que le gain de performance se fait par augmentation de la fréquence ce qui conduit à des problèmes énergétiques (consommation, dissipation)

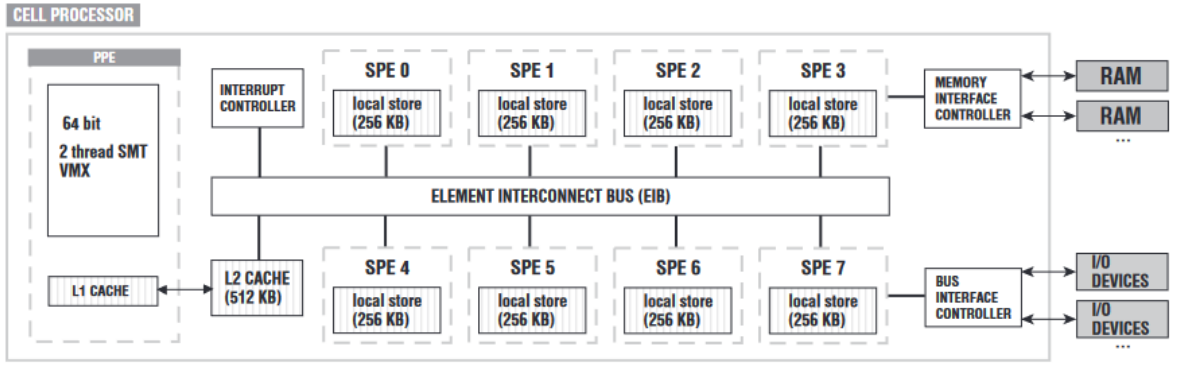
Chip-level Multiple Processors (CMP/CMT)

- Plus simplement appelés processeurs "multicore" multicoeurs
- Contrairement au SMT chaque coeur est un processeur à part entière
- Chaque coeur peut également supporter le SMT
  - Wii U = IBM PowerPC type 750 (celui de la Wii, single-core in-order) 3 coeurs
  - XBox360 = IBM PowerPC (in-order) 3 coeurs avec SMT 2 voies



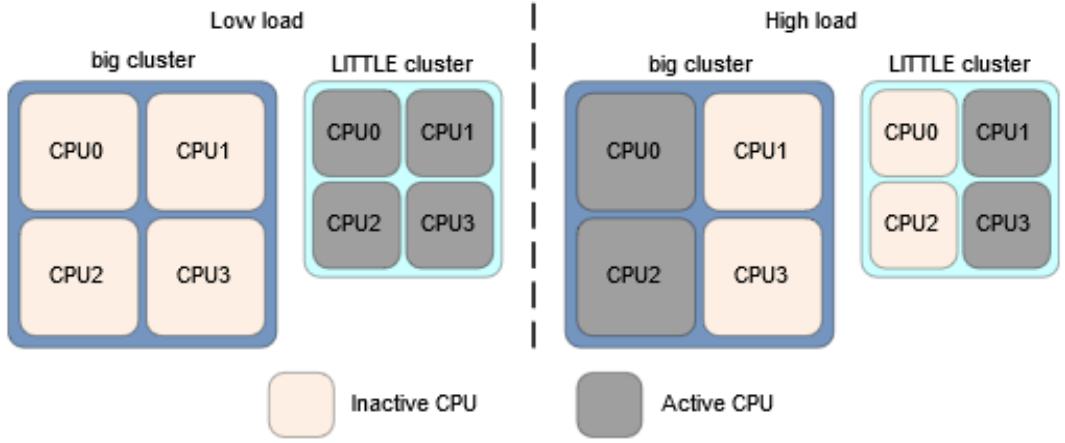
Topologies hybrides (hétérogènes)

- Ces architectures sont dites asymétriques car la répartition des coeurs est inégale:



- IBM Cell (PS3) : 1 coeur IBM PowerPC 970 (in order) avec SMT 2 voies contrôllant 8 co-processeurs SIMD

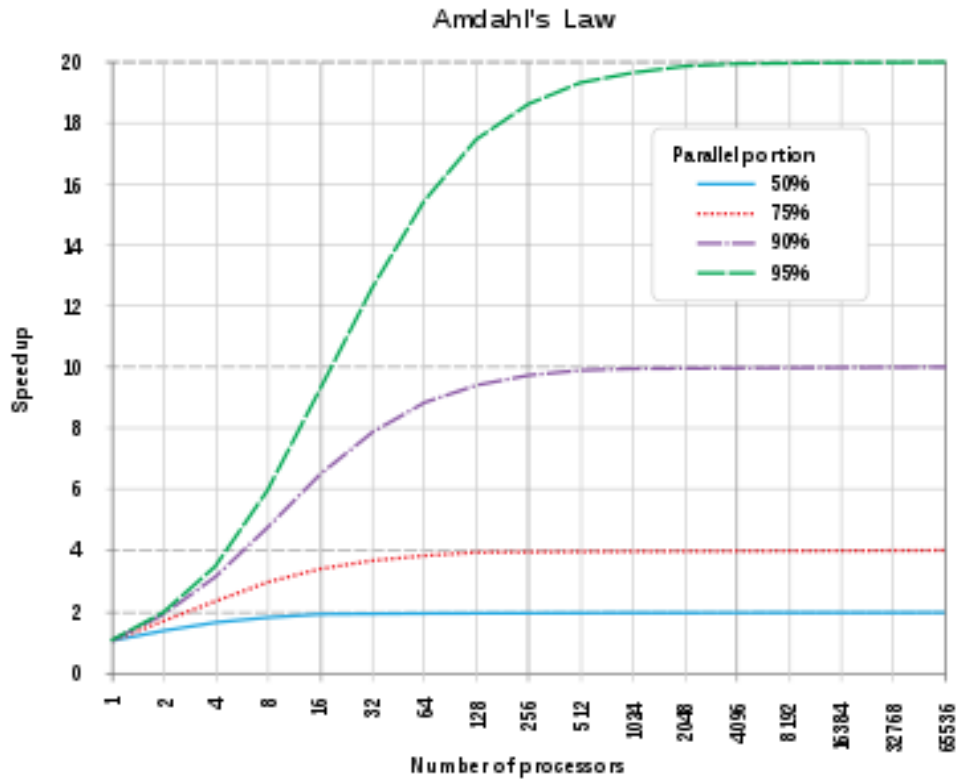
- ARM big.LITTLE : coeurs puissants (big) conjugués à des coeurs plus faible (little) mais plus efficients d'un point de vue énergétique.
  - Nvidia Tegra X1 (Switch) : ARM Cortex A57 (quad-core) + ARM Cortex A53 (quad-core, dédié au système)



Loi d'Amdahl

- La loi établie par Gene Amdahl formule le principe de la limitation de la vitesse d'un programme par sa partie séquentielle, celle qui ne peut être parallélisée.

- Elle est intéressante pour mesurer le gain apporté par la parallélisation d'un programme.
  - En pratique, elle expose qu'au delà d'un certains nombre de threads hardware la parallélisation n'a plus aucun effet (pour une taille fixe de données)



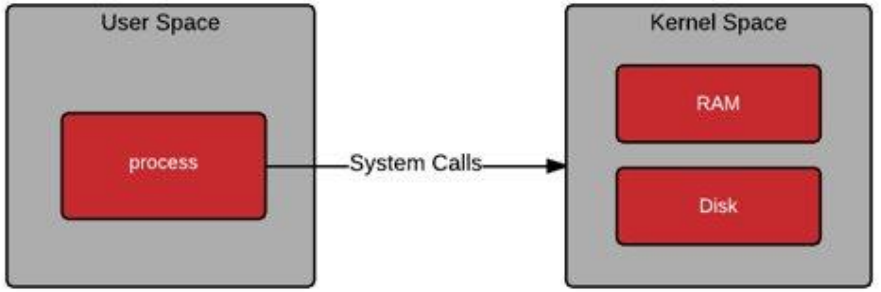
- C'est une loi assez péssimiste, des objections ont été émises notamment pour les architectures modernes
- A contrario la loi de Gustafson est plus optimiste et scalable https://fr.wikipedia.org/wiki/Loi_de_Gustafson
- Cf. https://research.cs.wisc.edu/multifacet/papers/tr1593_amdahl_multicore.pdf et plus récemment https://hal.inria.fr/hal-02404346/file/RR-9311%20%281%29.pdf

"we got fun and games"

- Herb Sutter a developpé dans un long article l'évolution et les différents changements de paradigmes pour le programmeur du fait de l'évolution du hardware
- https://herbsutter.com/welcome-to-the-jungle/

Architecture logicielle

Architectures de l'OS



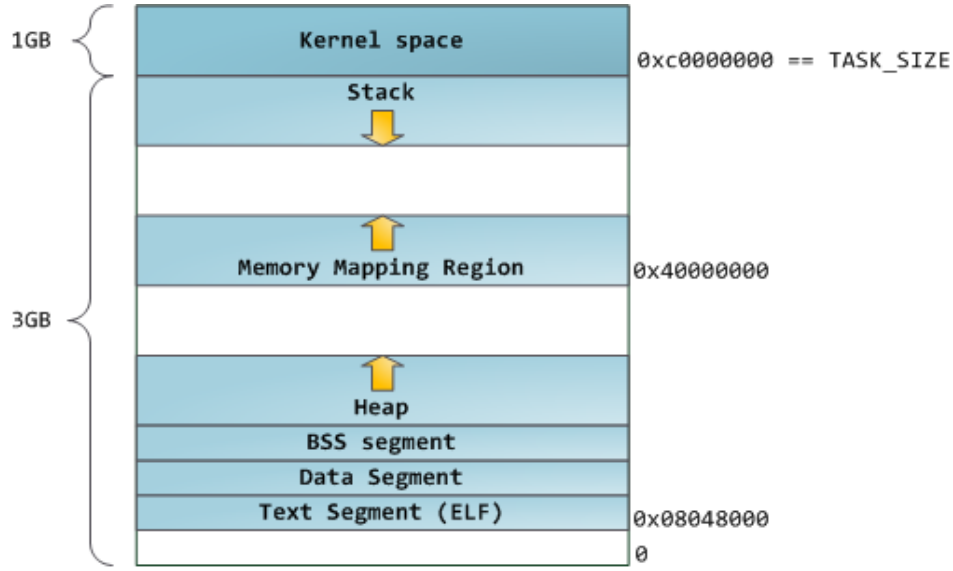
- Un OS est généralement séparé en deux couches
  - Noyau (kernel)
  - Utilisateur (user)

- Chaque couche représente un niveau d'accès aux ressources de la machine, le plus bas niveau étant le noyau.

- Le système d'exploitation gère également la notion de privilège afin de limiter la porter des actions de certains utilisateurs.

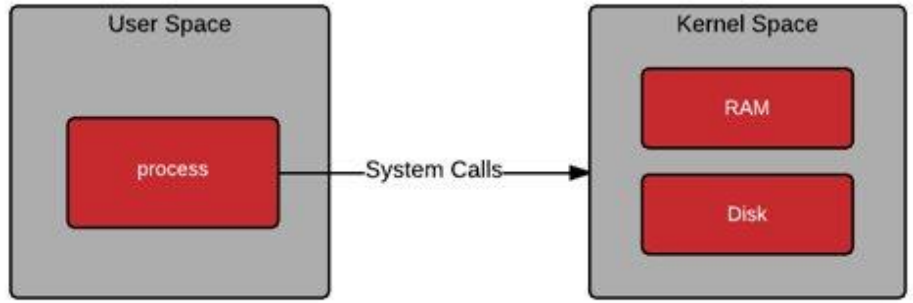
Espace d'adressage

- Une partie de la mémoire est réservée pour le noyau (kernel space)
- En dehors de cette zone mémoire un processus a théoriquement accès à l'ensemble de la mémoire
  - En pratique chaque processus ne voit que sa propre représentation de la mémoire du fait du système de protection/pagination mémoire



  - Détail de l'espace d'adressage Windows https://docs.microsoft.com/fr-fr/windows/win32/memory/memory-limits-for-windows-releases
  - Détail de l'espace d'adressage 64 bit de Linux https://www.kernel.org/doc/html/latest/x86/x86_64/mm.html

Processus



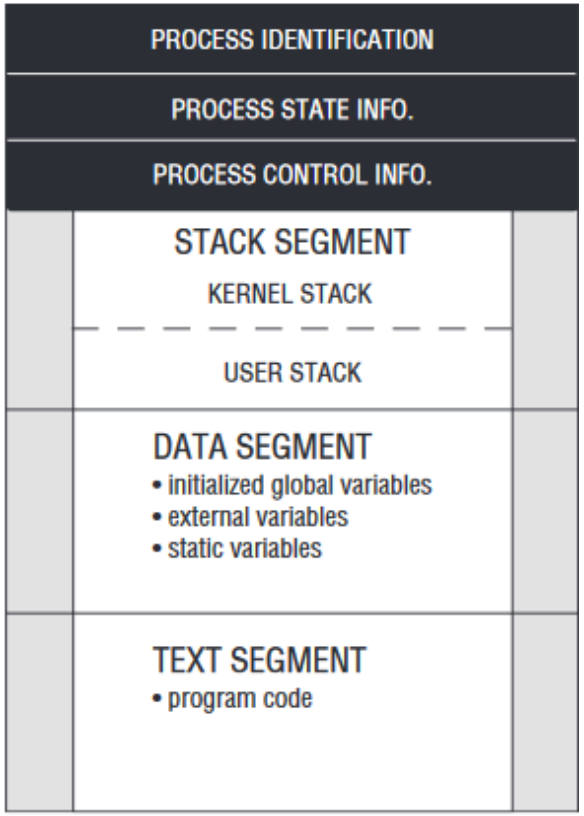
- code exécutable par l'OS issue d'un programme
  - Un exécutable étant précisement le code + les données
  - tandis qu'un processus correspond à une tâche prise en charge par l'OS
    - Un programme peut donc créer plusieurs processus

- Un processus est limité à son espace d'adressage (réel ou virtuel).
  - Cela inclus ses variables locales, les différentes sections (code, data etc...) mais également les évenements de l'OS, états des registres etc... ce que l'on appelle le "contexte d'exécution".

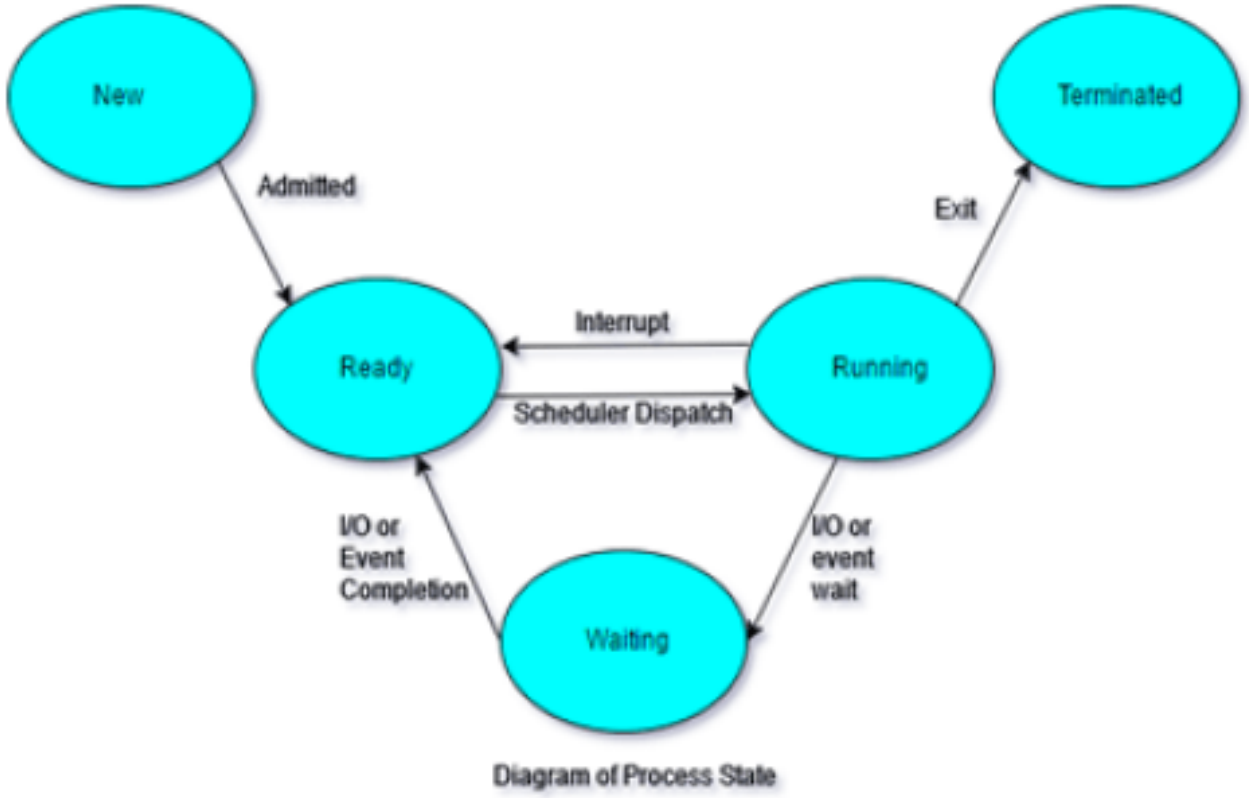
- Ainsi il ne peut pas lire ou écrire directement des données d'un autre processus (le système de protection mémoire de l'OS l'en empêche).
  - Y compris lorsque les processus sont créés depuis un même programme !

Découpage d'un processus

- Le Primary Control Block (PCB) contient les metadata
  - Identificateurs, états, etc...
  - Potentiellement assez lourdes
- Une pile (stack) propre à chaque processus
- Un segment de données (global, static, extern)
- Un segment de code



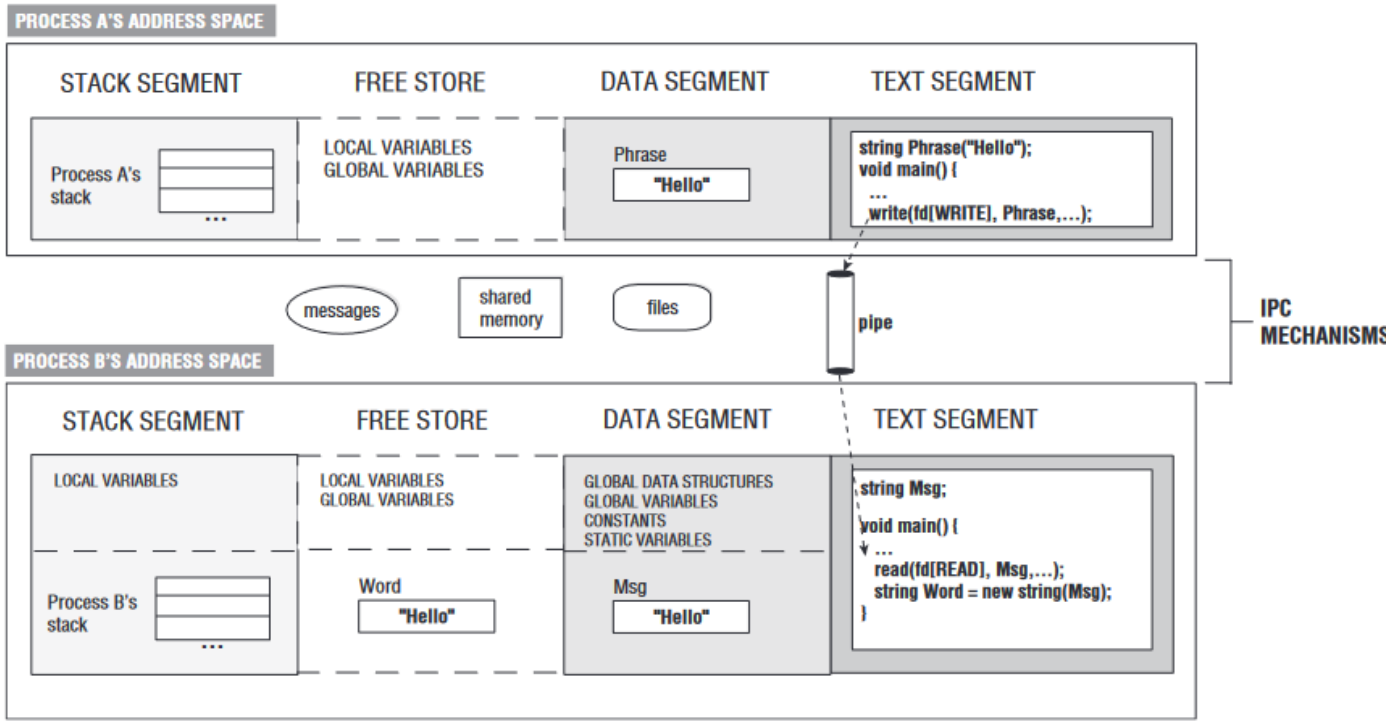
Etats d'un processus



Communication Inter-processus (IPC)

- IPC -pour *Interprocess Communication*- désigne l'établissement d'une communication entre au moins deux processus du système.

- Un processus peut écrire / lire dans un fichier sur disque, ou dans un "**pipe**" FIFO
  - Ou via des sockets systèmes ou réseau
- Un processus peut réserver un espace mémoire partagé (**shared memory**)
- Un processus peut recevoir / émettre des messages (**signaux** nomenclature UNIX, ou **events**, notifications sous Windows) de / vers l'OS ou d'autres processus
  - Nécessite la mise en place d'un gestionnaire (**handler**) pour interpréter le signal



Modèle d'exécution : séquentiel

- Jusqu'à présent nous n'avons parlé que du mode d'**exécution** le plus simple qui est le modèle **séquentiel**.
  - Nous avons déjà vu dans le module consacré à l'architecture des processeurs et au SIMD des améliorations telles le fonctionnement *out-of-order* (réordonnancement d'instructions) ou le parallélisme d'instruction.

- Dans ce modèle, les différentes parties ou tâches d'un programme s'exécutent les unes à la suite des autres en suivant un flux prédéfini.
  - Le flux peut subir quelques modifications de sorte à améliorer le temps d'exécution (sur les processeurs out-of-order par ex.) mais toujours de manière **déterministe**.

Modèle d'exécution : concurrence

- On parle de **concurrence** dès lors que deux événements prennent place dans le même intervalle de temps.
  - On dit qu'ils s'exécutent de manière concurrente même si il ne s'exécutent pas pour autant en parallèle.
  - On peut avoir une forme de "pseudo-parallélisme" qu'il soit hardware (SMT) ou software (multitâche).

- Ce modèle d'exécution est **non-déterministe** car on n'a pas de contrôle sur l'ordre des opérations (dépendent du scheduler à un instant T)

- Le traitement concurrent des tâches ne nécessite pas pour autant un environnement multi-processeurs ou multi-coeurs
  - Le noyau d'Amiga OS (Exec) permettait de faire fonctionner plusieurs tâches en concurrence sur un seul Motorola 68000 - et pourtant dépourvu de MMU (Memory Management Unit) nécessaire à la virtualisation (du coup un crash d'un programme faisait crasher l'OS)

Execution synchrone vs asynchrone

- Un processus est **synchrone** avec un autre processus lorsque l'un des deux doit **attendre la fin** de l'exécution de l'autre processus.

- Un processus **asynchrone** peut devenir localement synchrone si a un moment de son exécution il doit attendre la complétion d'un autre processus.

- Dans les deux cas il peut aussi bien s'agir de processus ayant une relation parent-enfant(s) ou de processus sans relations.
  - Il est de la responsabilité du processus qui "spawn" d'autres processus ou se duplique "fork" d'y mettre fin

types de parallélismes

- Parallelisme des **tâches** :
    - "coarse-grained" : il s'agit de l'exécution d'une tâche bien spécifique
        - Par exemple un serveur va utiliser un thread pour communiquer avec chaque client
    - "fine-grained" : on subdivise alors une tâches en plusieurs sous-tâches indépendantes.
- Parallelisme niveau **données** :
    - la granularité dépend des données à traiter.
    - On découpe tout ou partie des données en autant de portion que l'on a de threads
- Les deux types de parallélisme sont **combinables**

Multitâche

- Le fonctionnement multitâche d'un OS se fait via un "***scheduler***"

- **Coopératif** : une tâche est executée jusqu'à completion sauf si elle cède (*yield*) volontairement son usage du processeur au scheduler.
  - Les versions 16 bits de Windows (3.1) ou les versions de MacOS avant OS X

- **Préemptif** : une tâche peut être interrompue et re-exécutée par le scheduler. Cette préemption implique un changement de contexte.
  - L'interruption peut être volontaire (yield)
  - L'interruption peut suivre une politique temporelle (quantum) et/ou prioritaire
  - Les tâches passent à travers plusieurs états durant leur cycle d'exécution
  - Une particularité de la séparation kernel/user nécessite que les tâches kernel ne peuvent être préemptées (elles doivent toujours être completées)

- **Temps-réel** : (RTOS = Real-Time OS) le scheduler prend également en compte la survenue d'événements externes pour garantir le minimum de délai dans l'exécution / reprise d'une tâche

Context Switch

- Un changement de contexte a lieu lorsqu'un processeur doit interrompre l'exécution d'un processus afin d'en exécuter un autre.
  - A ce moment, l'état des processus est modifié (runnable -> paused, ou blocked, ou zombie etc...)

- Le contenu des registres du processeur est alors sauvegardé
  - Le pointeur ordinal (Program Counter, PC ou IP)
  - de même que l'état de la pile (SP),
  - les informations spécifiques à l'exécution du processus par l'OS (droits, privilèges, états utilisateur et noyau...)

- Sans compter l'état des I/O et des ressources, de la configuration mémoire et du CPU etc...

Ordonnancement

- Les tâches sont classées par priorité, de sorte que les tâches les plus prioritaires sont traitées le plus fréquemment.

- Le scheduler veille à allouer un intervalle de temps (ou tranche) d'exécution que l'on appelle un "**quantum**"
  - La durée est généralement de quelques millisecondes et peut être variable
    - À cela il faut ajouter le temps de commutation (task switching)
  - Certains OS permettent de booster les processus à faible priorité pour éviter qu'ils ne perdent totalement leur quantum

- Le multitâche préemptif est une forme de pseudo-parallélisme qui fonctionne par entrelacement (multiplexage) de l'exécution de chaque processus.

Polices de répartition de charge

- **Round-robin** (tourniquet) : c'est l'algorithme préemptif traditionnel
  - https://fr.wikipedia.org/wiki/Round-robin_(informatique)

- File (**FIFO**) : la particularité est ici que l'on ne privilégie aucune tâche
  - Le scheduler suit simplement l'ordre de soumission
  - Les thread-s ayant une priorité forte vont monopoliser le scheduler jusqu'à complétion (peut-être utile dans certains cas)

- Un autre aspect, qui relève du scheduler, est la gestion de l'**inversion de priorité**. Cela permet de "booster" des tâches à faible priorité lorsque les tâches à fortes priorités sont en attente par exemple
  - Il faut aussi que le scheduler gère correctement le cas où la tâche à forte priorité se réveille
  - https://fr.wikipedia.org/wiki/Inversion_de_priorit%C3%A9

Partie 2 - Multi-threading

Le multi-threading

- Le multi-threading est considéré plus performant que le multi-processing du fait que
  - Les **threads** (fil) d'un même processus partagent le même espace d'adressage
  - Les threads peuvent avoir une zone mémoire dédiée (thread-local storage, tls)
  - Les context-switchs sont plus légers (parfois les threads sont appelés processus légers)
    - Mais nécessitent cependant toujours un passage user <-> kernel
- Les fibres (**fibers**) sont une forme plus légère et rapides car ils s'exécutent dans le domaine utilisateur sans context switch. Ils sont schedulés au sein d'un seul thread.
  - Une fibre peut bloquer et rendre la main au scheduler, ou yield
- Conceptuellement proche des fibres, les **coroutines** sont des thread légers qui ne nécessitent pas de scheduler.
  - Une coroutine rend la main par un yield
  - Cf http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4024.pdf

Multithreading : usages

- Lorsque l'on a besoin d'un fonctionnement asynchrone
  - Gestion événementielle de l'interface utilisateur sans bloquer le thread principal
  - pour décoder des données, ou gérer le réseau, ou encore gérer le son sans perturber le thread principal

- Pour optimiser un programme
  - Utiliser plusieurs thread software pour réduire la latence
  - Utiliser plusieurs thread hardware pour réduire le temps de calcul (throughput)
  - Utiliser plusieurs thread pour traiter plus de données sur le même lapse de temps.

Multithreading : pour l'optimisation

- Déterminer les "**hotspots**" càd les parties coûteuses en performance
  - Nécessite d'expérimenter lorsque l'application en est encore au design

- Répartir les threads afin de maximiser les ressources (**load balancing**)
  - Il faut occuper le plus de thread hardware possible afin de tirer parti au mieux de la machine

- Attention à la **granularité**
  - Trouver le bon découpage entre nombre de thread et tâches à effectuer

- Limiter les **synchronisations**
  - Les tâches complètement indépendantes s'exécutent toujours plus rapidement

- Limiter le **partage** de la **mémoire**
  - Si la mémoire est le goulot d'étranglement, le multi-threading n'a pas d'utilité

Multithreading : les coûts

- La création de thread peut être assez coûteuse du fait de leur fonctionnement
  - Évitez de créer des threads dans les boucles critiques (utilisez un pool par ex.)
  - Si l'OS doit gérer trop de thread les context switch peuvent s'avérer gênant
    - Idéalement autant de thread software que hardware
- Attention à la granularité
  - Le temps de switch/création d'un thread peut être supérieur à la quantité de travail
- Limiter les synchronisations
  - Plus on a de thread à synchroniser plus on risque d'annihiler le parallélisme
- Limiter le partage de la mémoire
  - Plus des threads accèdent à la mémoire plus ils consomment la bande passante CPU<->RAM

Multithreading : API

- Chaque OS propose sa propre API bas niveau pour les threads
  - La lib pthread (portable 'POSIX' thread) est une API standardisée
  - Microsoft dispose de sa propre API qui est différente de pthread avec certains avantages

- Depuis le C++11 le langage intègre une bibliothèque et des idiomes pour la gestion des threads (issues majoritairement de boost.thread)
  - Objectivement la programmation est plus portable
  - Cependant on a moins de contrôle qu'avec les API bas niveau ...
  - … et l'API de base pour les threads laisse un peu à désirer
  - Le traitement en tâche (async, parallel_task, promise...) plutôt qu'en thread est à privilégier

- Les compilateurs sont aussi capables d'auto-parallelisation, soit de manière implicite, ou de manière explicite (en utilisant par exemple OpenMP)

Multithreading : les coûts des caches

- Efficacité des caches : éviter au maximum les conflits ou l'utilisation non nécessaire du bus (données non utilisées d'une structure par ex.)

- Ping-pong des caches : le contenu des caches peut être vidé (flush) notamment lorsque plusieurs threads partagent le même coeur.

- False cacheline sharing : lorsque des threads hardware accèdent à des données mémoires se trouvant sur la même ligne de cache, leur cache respectif doit synchroniser / mettre à jour son contenu (voire mettre à jour la RAM)
  - En général un emplacement mémoire ne peut résider dans deux caches à la fois
  - La meilleure façon d'éviter ce problème est que chaque thread évite de manipuler les mêmes zones mémoires et d'avoir un espacement d'au moins une ligne de cache entre les données traités par chaque thread

Multithreading : design patterns

- On retrouve essentiellement les formes suivantes:
- Delegation (boss-worker ou fork-join)
- Peer-to-peer
- Pipeline
- Producteur-Consommateur

Design pattern : delegation

- Aussi appelé boss-worker ou fork-join, c'est une forme de job system

- 1 thread (le boss) est chargé de :
  - créer d'autres threads (fork)
  - répartir la charge de travail entre plusieurs thread workers
  - Valider la fin du travail (join)

- Le thread boss est bloqué sur

Design pattern : peer-to-peer

- Similaire au pattern précédent à l'exception que tous les threads sont workers, il n'y a pas de boss à proprement parler

- Chaque thread a une responsabilité plus complexe

- Chaque thread peut se synchroniser sur une entrée commune,
- ou bien chacun avoir sa propre entrée

Design pattern : Pipeline

- Les threads s'exécutent en suivant un ordre de dépendance défini
  - Les threads sont connectés un peu comme sur une ligne d'assemblage
- Une fois qu'un thread a terminé de traiter une entrée, le résultat est transféré en entrée du thread suivant
- Chaque thread traite une tâche particulière
  - Ce design permet de traiter plusieurs flux en entrée

Design pattern : producteur -consommateur

- Un thread produit des données qui sont ensuite traitées par un thread consommateur
- Les données intermédiaires sont stockées dans une zone mémoire qui peut être limitée à un certains nombres de données (*bounded*) ou virtuellement illimitée (*unbounded*)

- La difficulté est double:

- Si le producteur produit plus de données que ne peut en traiter le consommateur il y'a un risque d'écrasement ou perte de données
  - Une "solution" consiste à **bloquer** la production tant qu'il n'y a pas de place
  - Alternativement, le producteur peut stocker localement les données produites
- A contrario, un consommateur peut consommer les données plus rapidement que le producteur n'arrive à les générer : on parle alors de famine (***starving***)

- Plusieurs variantes sont possibles : 1 producteur plusieurs consommateurs, plusieurs producteurs, plusieurs consommateurs

Multithreading en C++11

- La classe std::thread expose des fonctions de gestion
- std::thread::**hardware_concurrency**() est une fonction *static,* retourne le nombre de threads hardwares (physiques et logiques)
- std::thread::**get_id**() retourne l'identifiant interne (C++) du thread
- std::thread::**native_handle**() retourne l'identifiant système du thread

Multithreading en C++11 : std::thread

- std::thread représente un objet thread, mais pas forcément un thread système
  - C'est le cas seulement du constructeur par défaut, les autres formes de constructions permettent le passage d'une fonction (au sens large) et de ses arguments à l'objet thread, ce qui démarre le thread système.

- Lorsque la fonction executée par l'objet thread se termine, l'objet est détruit (via std::terminate() qui appel le destructeur).

- L'appel à **join**() permet de bloquer le fil d'exécution appelant jusqu'à ce que le thread référencé par le join() se termine.
  - C'est alors le thread appelant qui libère les ressources du thread joint.

- Il est possible de rendre un thread "asynchrone" par l'appel à **detach**()
  - Les ressources systèmes sont alors libérées à la fin de l'exécution du thread

Multithreading en C++11 : depuis le thread

- Dans le fil d'exécution de la fonction, on peut récupérer les informations suivantes:

- std::this_thread::**get_id**() retourne l'id interne du thread courant

- Std::this_thread::**yield**() demande au scheduler de reporter l'exécution du thread courant au prochain quantum.
  - Le comportement dépend fortement de l'OS et de la police d'ordonnancement du thread (FIFO, RR, autre)

- Std::this_thread::**sleep_for**() et **sleep_until**() reschedule l'exécution du thread courant durant un lapse de temps (_for) ou jusqu'à une période donnée (_until)

Multithreading C++11: utiliser l'API bas niveau

- **native_handle**() retourne un "*handle*" qui permet de manipuler le thread à l'aide des fonctions bas niveau de l'API (pthread ou Win32).

- Le principal intérêt réside dans l'utilisation de fonctionnalités non disponibles en C++ telles que :
  - Changement de priorité
  - Changement de mode de scheduling (FIFO, RR …)
  - Assignation (affinité) du thread à un coeur/processeur en particulier

Multithreading en C++11 : future et promise

- Le concept de future et promise permet de raisonner en tâche à accomplir plutôt qu'en thread.
- La principale différence avec std::thread réside dans la notion de valeur de retour.

- Std::**promise** permet de créer un objet (à usage unique) identifiable de manière asynchrone.
  - Comme son nom le laisse également supposer, une promesse peut-être annulée ou plutôt abandonnée (abandonned)
  - On utilise rarement std::promise directement

- std::**future** permet entre autre de récupérer le résultat d'une promesse à travers la fonction std::promise::**get_future**().
- On peut ensuite tester le succès ou récupérer le résultat via la fonction std::future::get()
- Alternativement on peut bloquer tant que le résultat n'est pas disponible avec la fonction std::future::**wait**()

Multithreading en C++11 : std::async

- Cet objet permet de mettre en place facilement des tâches asynchrone.
  - En pratique cela dépend de l'état de l'OS, des threads softwares et hardwares disponible etc.

- L'implémentation sous-jacente est opaque dans la façon d'utiliser les thread mais c'est ce qui rend std::**async** intéressant pour beaucoup, on a seulement un std::**future** en retour.
  - Ce n'est pas l'objet à utiliser si l'on souhaite avoir un maximum de contrôle sur le fonctionnement des threads
  - Un avantage de std::async est que l'on n'a pas à ce soucier de différentes problématiques (allocation des threads, sur-allocation de threads softwares etc...)
    - Par exemple, si aucun thread software ou hardware n'est disponible la tâche peut s'exécuter sur le thread courant

- En pratique la fonction associée peut être exécutée totalement en asynchrone (police std::**launch**::**async**) ou au moment du get() (police std::**launch**::**deferred**) ou bien les deux (police par défaut)

Multithreading en C++11 : packaged_task

- std::packaged_task est un objet qui encapsule de manière générique le traitement des tâches.
  - L'objet fonctionne de manière similaire à std::function

- La fonction membre **get_future**() permet de récupérer une future sur laquelle on peut atteindre la promesse via **get**()

- Une autre alternative consiste à créer un std::thread avec la packaged_task en paramètre (via std::move) et utiliser join() par ex.

Multithreading : thread-local storage

- Il est souvent utile d'avoir des variables ayant un scope global (globales au sens strict, variables static membre ou dans un scope) mais locales à un thread.

- C'est le concept de thread-local storage (tls) qui amène chaque thread à avoir sa propre vue (et donc une zone mémoire distincte) pour les variables du tls.

- En C++11 il suffit de marquer une variable (ou extern) avec le mot clé **thread_local**.