Introduction à la programmation parallèle et aux processus concurrents

Parallélisme

- **Motivations**
  - **Matérielles**
    - Nécessité sur un calculateur monoprocesseur
      - Mettre à profit les temps de blocage
      - Partager l'utilisation du processeur entre plusieurs activités : timesharing
    - Utilisation de calculateurs multiprocesseurs et/ou multicœurs
      - A mémoire partagée : plusieurs processeurs ont accès à une mémoire principale commune
      - Des calculateurs, reliés entre eux par un réseau, constituant un système réparti
  - **Logicielles**
    - Des parties de programmes sont relativement indépendantes et peuvent être exécutées en même temps
  - **Logiques**
    - Multi-activités mises en évidence dans la conception
- **Utilisation d'outils et de techniques de mise en œuvre de processus concurrents**

Top en Juin 2020

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,299,072	415,530.0	513,854.7	28,335
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
15	**PANGEA III** - IBM Power System AC922, IBM POWER9 18C 3.45GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Volta GV100, IBM Total Exploration Production France	291,024	17,860.0	25,025.8	1,367

Source : https://www.top500.org/lists/

Notion de processus

- **Concept présenté en L2**

Un processus
=

- **Définitions**
  - **N exécutions d'un même programme produisent N processus distincts**
  - **La juxtaposition de plusieurs processus permet de décrire des activités qui ne sont pas séquentielles**
    - ➔ **Application parallèle**

Notion de ressource

Une ressource
=
Un élément de l'environnement utilisé par un ou plusieurs processus

- **Exemples de ressources**
  - Matérielles : imprimante, mémoire…
  - Logicielles : fichier, sémaphore…
  - Environnement : puissance électrique, ventilateur, climatisation…

Taxinomie des processus

- **Processus indépendants**
  - 2 processus sont indépendants si l'exécution de l'un n'interfère en aucune manière avec l'exécution de l'autre
  - Ils produiront les mêmes résultats, quel que soit l'entrelacement de leurs exécutions
- **Processus dépendants**
  - L'entrelacement de leurs exécutions peut interférer sur leurs résultats
  - Processus coopérants
    - Processus contribuant à un objectif commun au sein d'une même application
    - Par exemple, le processus 2 ne peut exécuter une action B que lorsque le processus 1 a fini d'exécuter l'action A
  - Processus concurrents
    - Processus utilisant des ressources partagées, au niveau de leur application (plusieurs processus peuvent vouloir une donnée produite par un autre processus par exemple) ou au niveau du système d'exploitation (l'écran ou une imprimante par exemple)
  - Des processus peuvent être concurrents et coopérants

Taxinomie des processus

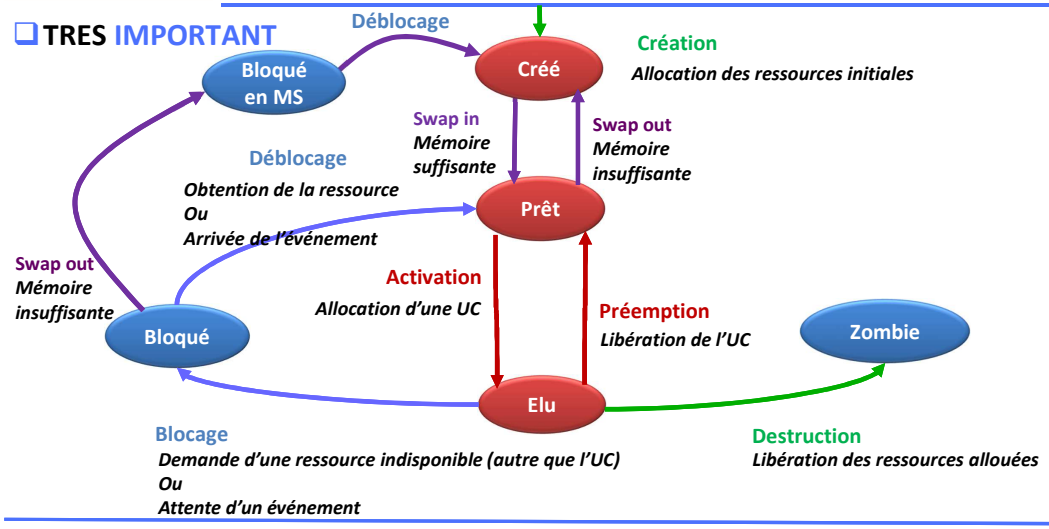
- **Processus dépendants**
  - Pour que leurs exécutions conduisent TOUJOURS aux résultat attendus, ils doivent être synchronisés = ne permettre l'exécution d'instructions que lorsque des conditions nécessaires sont vérifiées :
    - Donnée produite (et non consommée par un autre processus)
    - Imprimante libre
    - …
  - Peuvent communiquer entre eux afin d'obtenir les ressources dont ils ont besoin
    - Par partage de mémoire
    - Par passage de messages

Quelques propriétés des processus

- **Modèle de gestion d'un processus**
  - Statique
    - Le nombre de processus est fixé lors de la compilation et ne varie plus
  - Dynamique
    - Les processus sont créés à n'importe quel moment durant la vie de l'application
    - Ils disparaissent de la même manière
- **Relation parent-enfant**
  - Le processus parent est le processus responsable de la création du processus enfant

Graphe d'états d'un processus

- TRES IMPORTANT



Opérations sur les processus (1/3)

- Création
  - L'opération de création comporte la création d'un PCB (Process Control Block) et l'allocation des ressources initiales
  - Le processus est dans l'état Créé
- Destruction
  - L'opération de destruction entraîne la libération des ressources détenues par le processus (PCB, mémoire…)
  - Selon les systèmes d'exploitation, les descendants du processus peuvent être :
    - détruits en même temps que le processus concerné
    - conservés et recueillis par un processus d'accueil (Unix : processus 1)
  - Le processus passe dans l 'état Zombie, jusque ce que sa terminaison soit prise en compte
- Swap in / Swap out
  - Un processus ne peut candidater à l'obtention d'un processeur pour s'exécuter qu'une fois chargé en mémoire centrale
  - Lorsqu'il est chargé en mémoire, il passe dans l'état Prêt
  - Lorsqu'il est éjecté de la mémoire pour céder la place à un autre, il passe à l'état Créé

Opérations sur les processus (2/3)

- Activation
  - Un processus élu dispose d'une unité centrale (UC) pour s'exécuter, contrairement à un processus prêt
  - L'opération d'activation consiste à choisir un processus prêt pour chaque UC libre
  - Les processus prêts sont mémorisés dans une file d'attente, dont la gestion est adaptée à la politique d'allocation des UC (ordonnancement / scheduling)
    - FIFO
    - ou avec priorité
    - etc.
- Préemption
  - L'opération de préemption consiste à retirer l'UC à un processus élu
  - Le processus passe alors dans l'état prêt
  - L'UC est alors libre et le système doit effectuer une opération d'activation

Opérations sur les processus (3/3)

- Blocage
  - La raison du blocage peut être
    - Une ressource demandée par ce processus n'est pas disponible
    - Un événement particulier est attendu par ce processus
  - L'opération de blocage consiste à rendre non éligible un processus actif
  - Le processus passe de l'état Elu à l'état Bloqué
- Déblocage
  - L'opération de déblocage consiste à replacer le processus dans l'état Prêt puisque la raison de son blocage n'est plus fondée
  - Il devient à nouveau éligible à l'obtention d'une UC
- ATTENTION
  - Un danger de la programmation parallèle est qu'un processus bloqué peut le rester indéfiniment suite à une erreur de programmation

Types d'exécutions

- Une application parallèle peut comporter plusieurs processus

	Mono-Cœur	Multi-Cœur
Monoprocesseur	✓ Un seul processus en exécution (pseudo parallélisme) ✓ Commutation : • gérée par l'ordonnanceur • à la demande du processus (langage Modula-2)	Parallélisme réel
Multiprocesseur avec mémoire partagée (système centralisé) ou sans mémoire partagée (système réparti)	Parallélisme réel	Parallélisme réel

- Parallélisme réel :
  - Plusieurs processus peuvent être en exécution simultanément
  - Commutation gérée par l'ordonnanceur

Ordonnancement des processus

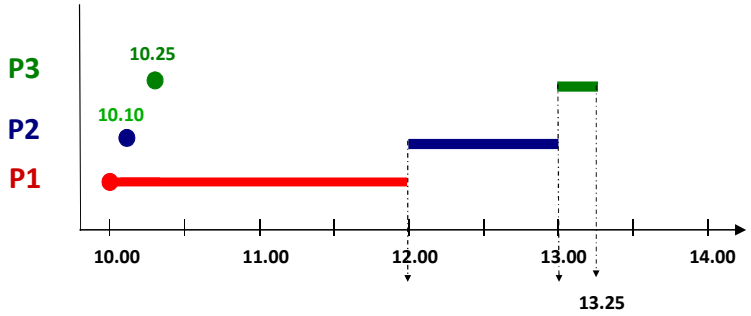
- L'ordonnanceur (scheduler) met en œuvre une politique d'allocation des UC
  - Avec ou sans préemption
  - Selon différentes politiques
    - First Come, First Served (FCFS ou FIFO)
    - Shortest Job First (SJF)
    - Avec priorité
    - Shortest Elapsed Time (SET)
    - Round Robin (RR) ou algorithme du tourniquet

Exemples d'ordonnancement

- 3 processus
  - P1 – Demande à s'exécuter à 10.00 – Durée : 2.00
  - P2 – Demande à s'exécuter à 10.10 – Durée : 1.00
  - P3 – Demande à s'exécuter à 10.25 – Durée : 0.25
- Hypothèse
  - Temps de commutation de contexte supposé nul

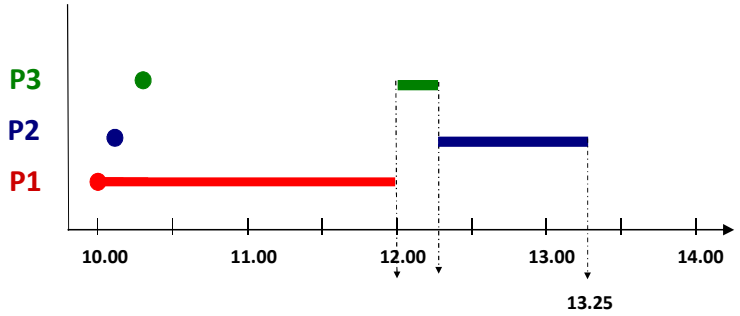
Politique d'ordonnancement FCFS (FIFO)

- Le premier processus qui arrive est élu
- Pas de préemption



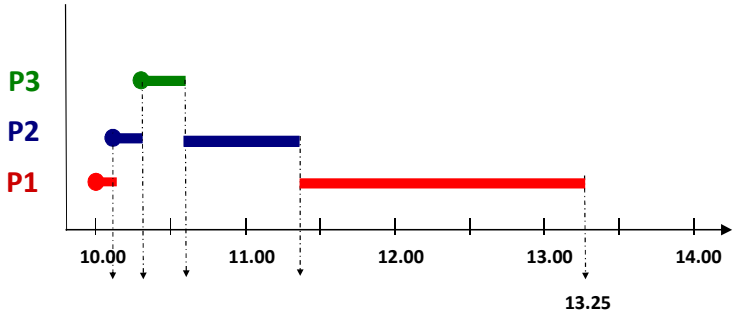
Politique d'ordonnancement SJF (Shortest Job First)

- Processus le plus court servi en premier
- Pas de préemption



Politique d'ordonnancement SRT (Shortest Remaining Time)

- Préemption
- Processus interrompu si une plus courte demande



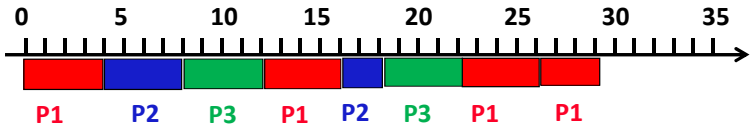
Politique d'ordonnancement Round Robin (1/3)

- Attribution d'un quantum de temps (entre 10 et 100 millisecondes)
- Préemption
- La libération de l'UC peut être due à :
  - la fin du quantum de temps
  - la fin du cycle d'UC

Un exemple d'ordonnancement des processus (2/3)

Processus	Durée de cycle
P1	15
P2	6
P3	8

Quantum de temps = 4 unités



Politique d'ordonnancement Round Robin (3/3)

- Évaluation
  - Algorithme adapté à un système interactif
  - Si le quantum de temps
    - est trop court : nombreuses commutations de contextes donc ralentissement des processus
    - est trop long : équivalent à FCFS
  - Les mesures faites montrent que le quantum devrait être choisi de telle sorte que 80% des cycles d'UC se terminent avant la fin du quantum

Exemple d'application parallèle

- Gestion d'un compte bancaire
  - Environnement : multiprocesseur à mémoire partagée
  - Deux processus qui s'exécutent en parallèle
    - Debiter : qui débite un certain montant du solde du compte géré
    - Crediter : qui crédite un certain montant sur le solde du compte géré
  - Variable partagée (voire fichier si besoin était)
    - montantTotal : solde du compte géré
- Attention !
  - Variable partagée ente activités parallèles = Variable stockée dans une zone de mémoire que peuvent référencer ces activités = mémoire partagée
    - Processus Unix : Partie d'un segment de mémoire partagée
    - Threads Posix : Partie de l'espace d'adressage du processus dans lequel les threads coexistent
  - Variable globale (visibilité dans le code) =/=> Variable partagée !
    - Rappel L2 : De base, un processus père et ses processus fils ne partagent rien !

Gestion d'un compte en banque

```
int montantTotal;    // Environnement à mémoire partagée
                     // Variable supposée partagée

Process Credit (unsigned int montant) {
 montantTotal = montantTotal + montant;        ①
}

Process Debit (unsigned int montant) {
   if (montantTotal < montant)                 ②
     printf("Deficit…");                       ③
    montantTotal = montantTotal – montant;     ④
 }
```

Attention !
« Process »
=
Pseudo-syntaxe

Exemple 1 : Entrelacement des exécutions

- Initialisation : montantTotal = 10
- Exécution : Debit(5) // Debit(9)
- [(2), (3), (4)] entrelacées avec [(2), (3), (4)]
- Exécution : (2), (2), (4), (4)

Process Debit (5)	Process Debit(9)	montantTotal < montant	montantTotal
(2) if (montantTotal < montant)		False	
	(2) if (montantTotal < montant)	False	
(4) montantTotal = montantTotal - montant			5
	(4) montantTotal = montantTotal - montant		-4

Quelle solution envisager ?

```
Process Debit (unsigned int montant) {
  if (montantTotal < montant)                ②
    printf("Deficit…");                      ③
  montantTotal = montantTotal – montant;     ④
}
```

❑ **Problème : Conflit pour accéder à la variable partagée montantTotal**

❑ **Solution**

- Il faut que le processus qui exécute Debit soit le seul à accéder à la variable montantTotal
- Rendre la séquence accessible par un seul processus

➔ Section critique: une section de code qui doit être totalement exécutée par un processus avant qu'un autre ne puisse s'y engager

Exemple 2 : Entrelacement des exécutions

- Initialisation : montantTotal = 10
- Exécution : Credit(5) // Debit(10)
- [(1)] entrelacée avec [(2), (3), (4)]
- Exécution : (1), (2), (4)

Process Credit(5)	Process Debit(10)	montantTotal < montant	montantTotal
(1) montantTotal = montantTotal + montant			15
	(2) if (montantTotal < montant)	False	
	(4) montantTotal = montantTotal - montant		5

- Résultat : montantTotal = 5
- Exécution consistante

Gestion d'un compte en banque - Version 2

❑ **Et si on écrit le processus Credit de cette manière :**

```
int montantTotal; // Envt à mémoire partagée
                  // Var. supposée partagée

Process Credit (unsigned int montant) {
  int tmp;
  tmp = montantTotal;                ①
  montantTotal = tmp + montant;      ②
}
```

❑ **Que se passe-t-il ?**

Exemple 3 : Entrelacement des exécutions

- Initialisation : montantTotal = 10
- Exécution : Credit(5) // Credit(7)
- [(1), (2)] entrelacées avec [(1), (2)]
- Exécution : (1), (1), (2), (2)

Process Credit (5)	Process Credit(7)	tmp		montantTotal
(1) tmp = montantTotal		10		
	(1) tmp = montantTotal		10	
(2) montantTotal = tmp + montant				15
	(2) montantTotal = tmp + montant			17

Exemple 2 : Entrelacement des exécutions

- **Solution**
  - Il faut qu'un processus qui exécute Credit ou Debit soit le seul à accéder à la variable montantTotal
  - Chacune de ces sections de code doit être une section critique
  - Ces sections de code doivent être en plus exécutées en exclusion mutuelle
- ➔ Exclusion mutuelle : l'exécution d'une de ces sections ne peut pas débuter si une autre de ces sections de code est déjà en cours d'exécution

Alors quelle version est la meilleure ?

Version 1

```
Process Credit (unsigned int montant) {
  montantTotal = montantTotal + montant;        ①
}
```

Version 2

```
Process Credit (unsigned int montant) {
  tmp = montantTotal;                           ①
  montantTotal = tmp + montant;                 ②
}
```

- **Aucune !**
  - La V1 peut produire elle aussi un comportement erroné car sa traduction, en instructions machine, est équivalente à la V2 :
    - LOAD montantTotal
    - ADD montant
    - STORE montantTotal
  - Et cette séquence n'est pas atomique…

Exemple : Gestion d'événements

```
int count = 0;     // Envt à mémoire partagée
                   // Variable supposée partagée
Process Sensor () {
  for (;;) {
    WaitEvent();                    ①
    count = count + 1;  }           ②
}
Process Recorder () {
  for (;;) {
    sleep(n);                       ③
    printf("%d events", count);     ④
    count = 0;  }                   ⑤
 }
```


Exemple d'exécution

Sensor()	Recorder()	count
(1) WaitEvent()		0
(2) count++		1
	(3) sleep(n)	1
	(4) print 1	1
(1) WaitEvent()		
(2) count++		2
	(5) count = 0	0

- Des événements sont perdus !
- Pourquoi ?

Synthèse : section critique et exclusion mutuelle

- Des processus parallèles
  - Utilisent des variables partagées pour échanger de l'information
  - Conflit pour accéder à ces ressources
- Résultat d'une exécution parallèle ←→ exécution séquentielle avec des entrelacements
- Une portion de code qui utilise des variables partagées ne doit être exécutée que par un seul processus à la fois. Elle constitue une section critique
- Lorsque plusieurs portions de code utilisent des variables partagées, elles doivent être exécutées en exclusion mutuelle : lorsqu'un processus en exécute une, plus aucune autre ne peut être exécutée par un autre processus

Quelques définitions

- Ressource critique : ressource qui ne peut être accédée (exécutée) que par un et un seul processus à la fois
  - Une imprimante, un écran, un compte en banque, une voie de chemin de fer….
- Section (de code) critique (S. C.) : code devant pouvoir faire l'hypothèse qu'il utilise la ressource de manière exclusive.
  - Si aucune précaution particulière n'est prise, rien n'empêche plusieurs entités d'utiliser simultanément la ressource
  - Empêcher les entités qui sont en compétition (pour une ressource donnée) d'entrer simultanément dans leur section [de code] critique
- Utiliser des techniques de synchronisation pour gérer les problèmes d'entrelacement dans les sections critiques et mettre en œuvre l'exclusion mutuelle (E. M.) entre sections de code ayant des ressources partagées :

```
entréeSectionCritique();        entréeSectionCritique();
// Section de code 1            // Section de code 2
sortieSectionCritique();        sortieSectionCritique();
```


Propriétés vérifiées par une solution au problème de la section critique

- P1 : A tout moment un seul processus exécute la section critique
- P2 : Si plusieurs processus sont bloqués en attente d'entrer en section critique, alors qu'aucun processus n'exécute la section critique, alors un de ces processus entrera en section critique au bout d'un temps fini
- P3 : Si un processus est bloqué à l'extérieur de la section critique, alors ce blocage ne doit pas empêcher un autre processus d'entrer en section critique
- P4 : La solution doit être la même pour tous les processus

S. C. et E. M. : 2 types de solutions

- Des solutions matérielles
  - Basées sur des instructions dédiées
  - Test And Set / Masquage des interruptions
- Des solutions logicielles
  - avec attente active
    - Chaque processus teste de façon répétitive et continue si les conditions lui permettant d'entrer en section critique sont satisfaites
    - Cela suppose que cette attente active est de courte durée
  - avec attente passive
    - Un processus passe dans un état passif si les conditions d'entrée en section critique ne sont momentanément pas satisfaites pour être réveillé ultérieurement lorsqu'elles seront vérifiées
    - Exemples étudiés ultérieurement : sémaphore, condition, etc.
    - Généralement, la mise en œuvre s'appuie sur les deux types de solutions précédents

- **Hypothèse assurée par le matériel : les seules instructions atomiques existantes sont :**
  - La lecture d'une variable
  - L'écriture d'une variable
- **Solutions présentées pour deux processus**

- **Solution 1 : Utiliser un booléen**

```
bool occupe = false; // Envt à mémoire partagée
                     // Variable supposée partagée
void entrer () {
   while (occupe)
     ;               // Attente active
   occupe = true;    // Le processus entre en S.C.
}
void sortir () {
   occupe = false;   // Le processus sort de S.C.
}
```

```
void processus0 () {
  entrer();
 // Processus 0 en S.C.
 sortir();
}
void processus1 () {
  entrer();
 // Processus 1 en S.C.
 sortir();
}
```


- **Solution 2 : Contrôler si l'autre processus demande à entrer en S.C**

Chaque processus vérifie si l'autre a demandé à entrer en S.C avant de s'autoriser à entre lui-même

```
enum NumeroProcessus {0, 1};

bool demandeDe[2] = {false, false};
// Variables supposées partagées (mémoire partagée)
// true = le processus (0 ou 1) demande à entrer en S.C.

void entrer(NumeroProcessus qui) {
   demandeDe[qui] = true;
   while (demandeDe[(qui + 1) % 2])
      ;                  // Attente active
}

void sortir(NumeroProcessus qui) {
   demandeDe[qui] = false;
}
```

```
void processus0 () {
  entrer(0);
 // Processus 0 en S.C.
 sortir(0);
}
void processus1 () {
  entrer(1);
 // Processus 1 en S.C.
 sortir(1);
}
```


- **Solution 3 : Gérer l'identité du processus admis à entrer en section critique**

La valeur d'une variable détermine quel processus peut entrer en section critique. A sa sortie, le tour est donné à l'autre

```
enum NumeroProcessus {0, 1};

NumeroProcessus tour = 0;   // Processus autorisé
                            // à entrer en S.C.
void entrer(NumeroProcessus qui) {
   while (tour != qui)
      ;                     // Attente active
}

void sortir() {
   tour = (tour + 1) % 2;
}
```

```
void processus0 () {
  entrer(0);
 // Processus 0 en S.C.
 sortir();
}
void processus1 () {
  entrer(1);
 // Processus 1 en S.C.
 sortir();
}
```

S.C. et E.M. : Solutions logicielles avec attente active

- **Solution 4 : Solution de Peterson**
  - Combiner les solutions 2 et 3
  - Le tableau demandeDe assure qu'un seul processus entre en section critique
  - La variable tour résout le problème de l'interblocage, c'est-à-dire de conflit et d'élection d'un des deux processus en cas de demandes d'entrée simultanées

```
enum NumeroProcessus {0, 1};
NumeroProcessus tour = 0;              // Envt mémoire partagée
bool demandeDe[2] = {false, false};  // Var. supposées partagées

void entrer(NumeroProcessus qui) {
   NumeroProcessus autre = (qui + 1 )% 2;
   demandeDe[qui] = true;
   tour = autre;
   while (demandeDe[autre] && (tour == autre))
         ;
}
void sortir(NumeroProcessus qui) {
    demandeDe[qui] = false;
}
```

```
void processus0 () {
   entrer(0);
   // Processus 0 en S.C.
   sortir(0);
}

void processus1 () {
   entrer(1);
   // Processus 1 en S.C.
   sortir(1);
}
```


Solution de Peterson avec attente active

- Fonctionne..
- mais…
- consomme de l'UC (et de l'énergie) à ne rien faire !

➔ L'attente active est à

- De plus, solution très difficile à généraliser à N processus
- Il faudra se tourner vers les solutions logicielles sans attente active (sémaphore, condition)

S.C. et E.M. : Solutions matérielles basées sur des instructions dédiées

- Utilisation des interruptions
- Contexte monoprocesseur : un seul processus actif
- Exécuter une séquence d'actions en exclusion avec tout autre processus
  - Lui assurer de garder le processeur en le rendant ininterruptible à l'entrée de la section critique: masquer les interruptions ou le placer sur le niveau d'exécution maximum
  - En sortie de section critique, le replacer à son niveau initial

```
const unsigned int NIVEAU_MAX = xxx;   // Niveau maximum
unsigned int ancienNiveau;

void entrer() {
  ancienNiveau = affecterNiveauIT(NIVEAU_MAX);
}

void sortir() {
  affecterNiveauIT(ancienNiveau);
}
```


S.C. et E.M. : Solutions matérielles basées sur des instructions dédiées

- Instruction indivisible de lecture-écriture
- En un cycle atomique, la valeur d'une variable est lue et affectée à true

```
bool TestAndSet(bool *occupe) {
   bool valActuelle = *occupe;
   *occupe = true;
   return valActuelle;
}

void entrer(bool *occupe) {
        while (TestAndSet(occupe))
          ; // attente active
}

void sortir(bool *occupe) {
        *occupe = false;
}
```

```
bool occupe = false;

void processus0 () {
   entrer(&occupe);
   // Processus 0 en S.C.
  sortir(&occupe);
}

void processus1 () {
   entrer(&occupe);
   // Processus 1 en S.C.
  sortir(&occupe);
}
```

Processus et Threads

Révisions…

- On suppose donné le programme suivant :

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define N 3

int lesPids[N];

void afficherLesPids (int *tab) {
  int i;
  for (i = 0; i < N; i++)
    printf("%3d", tab[i]);
  printf("\n");
}

void maFonction (int monNum) {
  int i;
  printf("Processus fils %d : Contenu de lesPid = ", monNum);
  afficherLesPids(lesPids);
}
```

Contexte : multiprocesseur sans mémoire partagée

```c
int main (void) {
  int i;
  printf("Processus pere debut : \n");
  afficherLesPids(lesPids);

  for (i = 0; i < N; i++)
    switch (lesPids[i] = fork()) {
      case -1 : perror("Echec fork : ");
                exit(1);
      case 0  : maFonction(i);
                exit(0);
      default : break;
    }

  printf("Processus pere fin : \n");
  afficherLesPids(lesPids);

  return(0);
}
```

- Donner le contenu du tableau *lesPids* au début et à la fin de l'exécution de chacun des processus créés par cette application

Rappels sur les processus Unix

- Un processus Unix est un programme en cours d'exécution
- Un processus Unix peut créer un (ou plusieurs) processus fils
  - Il y a copie
    - De la plupart des attributs du père, à l'exception du pid, du pid du père, de la localisation des segments donnée et pile et de quelques autres informations « personnelles »
    - du segment de données
    - du segment pile
  - Il y a partage du segment de code
  - Les segments données et pile sont privés à un processus
    - Un père et un fils ne partagent donc aucune information !
- Échanger des données demande des outils supplémentaires : tubes de communications, segments de mémoire partagée…

Threads

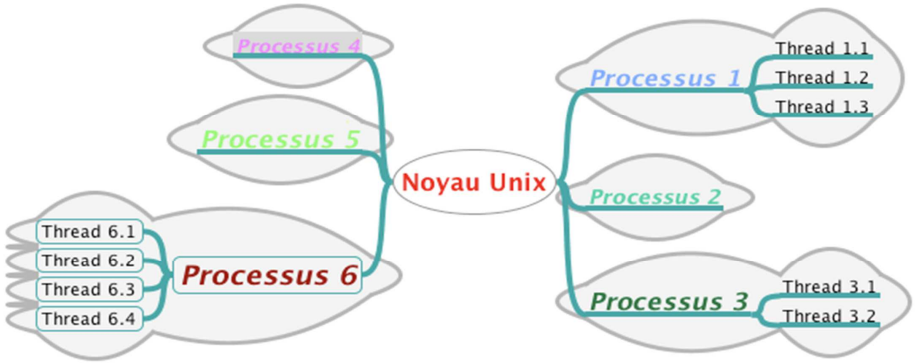
Concept de processus (lourd)

- Un seul flot de contrôle séquentiel par processus
- Un espace d'adressage par processus
  - Il n'existe pas d'espace partagé entre deux processus
- Le processus est l'unité d'allocation de ressources pour le système
- Le processus constitue l'unité d'ordonnancement

Concept de thread (processus léger)

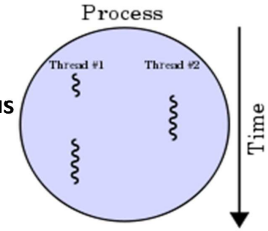
- Plusieurs flots de contrôle séquentiels
  - Les flots de contrôle sont concurrents
- Un seul espace d'adressage
  - Espace partagé entre les threads
- Le processus reste l'unité d'allocation de ressources pour le système
- Le processus n'est pas l'unité d'ordonnancement (selon l'option choisie)
  - Les threads peuvent être directement gérés par l'ordonnanceur du système

Schéma d'activation



Définition d'un thread

- Un thread est simplement un flux d'exécution au sein d'un processus
  - Dans le cas d'un processus Unix traditionnel, un seul thread démarre à la première instruction de la fonction main
  - Le thread suit la logique du programme jusqu'à la terminaison du processus
- Extension avec les threads
  - Une application peut disposer d'un ou de plusieurs flux d'exécution au sein d'un même processus Unix (« threads compagnons)
  - Chaque flux d'exécution constitue un thread
  - Un processus, disposant initialement d'un seul thread, peut créer des threads supplémentaires au sein du même processus
  - Les ressources du processus sont partagées entre les threads internes au processus.
  - Sur une machine multiprocesseur, les threads internes à un même processus Unix peuvent s'exécuter en parallèle sur des processeurs différents



Ressources d'un thread

- Chaque thread dispose de ressources propres
  - d'un morceau de code à exécuter
  - de données sur lesquelles le code travaille
  - d'une pile d'exécution permettant de gérer la dynamique du thread
- Les threads compagnons disposent de ressources communes
  - l'espace d'adressage (données et code)
  - les variables globales
    - Lorsqu'un thread modifie une variable globale, la nouvelle valeur est immédiatement visible des autres threads compagnons
  - les fichiers ouverts, le répertoire courant, le masque de création…
  - …

Attributs d'un thread

- Chaque thread dispose d'attributs spécifiques
  - de données propres définies par l'utilisateur
  - une pile, un compteur ordinal, des registres CPU
  - un état (actif, bloqué ou prêt)
  - de la variable système `errno`
  - d'un masque de signaux bloqués
  - …

Avantages des processus multi-threads

- Amélioration du rendement de l'application
  - Exemple du serveur web
- Meilleure utilisation des multi-processeurs
- Structuration de programme souvent mieux adaptée
  - Souvent, la plupart des threads sont en attente
- Amélioration de la communication de données entre activités

Implantation des threads (1/2)

- Les threads peuvent être implantés
  - en tant qu'abstractions de niveau utilisateur
  - en tant qu'abstractions de niveau noyau
  - comme combinaison des deux
- Abstraction au niveau utilisateur
  - Principes
    - Ils sont gérés directement à l'intérieur de l'espace d'adressage d'un processus
    - N'utilise aucun service système, à l'exception de ceux associés au processus
    - Les threads compagnons sont multiplexés sur le processus et par le processus pour être exécutés
  - Avantages
    - Meilleures performances (pas de commutation noyau)
    - Ensemble de fonctionnalités extensible sans nouvelle version système

Implantation des threads (2/2)

- **Abstraction au niveau noyau**
  - Chaque thread dispose de son propre contexte, d'un espace privé, d'une pile propre
  - **Inconvénients**
    - Accroissement de la taille du noyau
    - Système figé dans ses fonctionnalités
- **Combinaison des deux**
  - **Un thread noyau pour un thread utilisateur**
    - Le thread noyau est qualifié LWP (lightweight process)
    - Le thread utilisateur est le thread lié au LWP
    - Solution intégrant les avantages et les inconvénients des deux solutions précédentes
  - **Multiplexage des processus liés sur un même LWP**

Les threads POSIX

La norme POSIX

- **Normalisation produite par IEEE et standardisée par ANSI et ISO**
  - **POSIX 1003.1 : OS, processus, SGF, API**
  - **POSIX 1003.2 : utilitaires**
  - **POSIX 1003.1b : temps réel**
  - **POSIX 1003.1c : threads → pthreads**
  - **POSIX 1003.1d : extensions TR supplémentaires**
- **La norme POSIX comporte le composant logiciel DCE** (Distributed Computing Environment) **qui offre :**
  - **le type** thread
  - **le type** mutex
  - **le type** condition
  - **le type** exception

Bibliothèque pthread – Les bases

- **Synchronisation de base : mutex et condition**
- **Types (opaques) : <sys/types.h>**
  - **pthread_t, pthread_key_t**
  - **pthread_mutex_t**
  - **pthread_cond_t**
  - **pthread_once_t**
- **Attributs : standard + propres à l'implantation**
  - **pthread_attr_t**
  - **pthread_mutexattr_t**
  - **pthread_condattr_t**
- **Bibliothèque → #include <pthread.h> + compilation : -lpthread**
- **Gestion des erreurs**
- **Primitives « thread-safe »**

Création d'un thread

```
int pthread_create(pthread_t            *thread,
                   const pthread_attr_t *attr,
                   void                 *(*start_routine)(void*),
                   void                 *arg);
```

- start_routine = fonction exécutée par le thread
- arg = argument de cette fonction
- attr = attributs optionnels de création
- thread = identificateur

- Toutes les ressources nécessaires au thread doivent avoir été initialisées
- Erreurs possibles :
  - EINVAL : attributs invalide
  - EAGAIN : ressources insuffisantes

Identité du thread actif

- Retourne l'identification du thread actif
  - Équivalent pour les processus de getpid()

```
#include <pthread.h>
#include <unistd.h>

pthread_t pthread_self (void);
```


Terminaison d'un thread

```
#include <pthread.h>
#include <unistd.h>

void pthread_exit (void *retval);
```

- retval = pointeur sur le résultat retourné par le thread qui se termine

- ATTENTION : Le pointeur doit repérer une zone qui sera encore accessible après la terminaison du thread (cette zone ne peut pas être une variable locale au thread car elle serait allouée dans la pile et donc détruite à la terminaison du thread)
  - Fait passer le thread dans un état « zombie » jusqu'à ce que le résultat rendu par le thread soit effectivement récupéré par un thread compagnon (voir pthread_join)
  - Le thread se termine et pthread_exit ne retourne donc rien

Attente de la terminaison

```
#include <pthread.h>
#include <unistd.h>

int pthread_join ( pthread_t   thread,
                   void        **retval );
```

- thread = identificateur du thread attendu
- retval = est un pointeur `void **` contenant l'adresse d'une variable `void *` devant recevoir le code retour du thread attendu (pour rappel, un pointeur `void *`)
- Le thread attendu se termine réellement après réception de la valeur retournée (il quitte l'état « zombie »)
- Retourne 0 en cas de succès, un code d'erreur sinon
  - EDEADLCK, EINVAL, ESRCH

Exemple – Code d'un thread

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define VAL 100

/* Fonction exécutée par le thread */
void *f_thread (void *p) {
  int *cr = malloc(sizeof(int));
  printf("\tDebut du thread compagnon ");
  printf("\tIci le thread numero: %lu \n", pthread_self());
  *cr = VAL;
  printf("\tValeur retournee: %d \n", *cr);
  pthread_exit((void*)cr);
  /* On pourrait aussi écrire : return ((void *)cr);
  ou return ((void *)&cr); si cr était déclarée en global par : int cr */
}
```


Exemple – Thread principal

```c
int main(){
  pthread_t ptid;
  int *res = NULL;
  pthread_attr_t attribut;

  printf("Debut du thread principal\n");

  /* Creation du thread compagnon */
  if ( pthread_create(&ptid, NULL, f_thread, NULL) != 0){
    perror("Probleme lors de la creation du thread compagnon:");
    exit(99);  /* plutôt pthread_exit() si hors thread principal */
  }

  /* Attente fin d'execution du thread */
  pthread_join(ptid, (void**)&res);

  printf("Test resultat: %d \n", *res);
  free(res);
  printf("Fin du thread principal\n");
}
```


Exemple – Exécution

```
Go %./ex1-0
Debut du thread principal
        Debut du thread compagnon
        Ici le thread numero: 25166848
        Valeur retournee: 100
Test resultat: 100
Fin du thread principal
Go %
```


Processus vs Threads – Exercice 0

- 1. Écrire une application dans laquelle deux activités parallèles a1 et a2 coexistent et affichent un message avant de se terminer.
- 2. Modifier cette application pour que a2 modifie la valeur d'une variable globale qui sera affichée par a1 après la terminaison de a2.
- 3. Modifier cette application pour que a2 renvoie comme compte-rendu d'exécution la valeur de la variable globale qui sera récupérée et affichée par a1.

- Versions 1a, 2a & 3a : a1 et a2 sont des processus
- Versions 1b, 2b & 3b : a1 et a2 sont des threads

- Quelles différences peut-on constater ?

Les attributs d'un thread

- **detachstate**
  - Thread détaché ou non
  - Valeur :
    - PTHREAD_CREATE_JOINABLE
    - PTHREAD_CREATE_DETACHED
- **stacksize**
  - Si _POSIX_THREAD_ATTR_STACKSIZE
  - Taille minimale de la pile du thread > PTHREAD_STACK_MIN
- **stackaddr**
  - Si _POSIX_THREAD_ATTR_STACKADDR
  - Adresse de la pile du thread
- **scope**
  - Portée de la compétition pour l'UC
  - Valeur :
    - PTHREAD_SCOPE_SYSTEM
    - PTHREAD_SCOPE_PROCESS
- **inherit_scheduler**
  - Hériter de son créateur
  - Valeur :
    - PTHREAD_INHERIT_SCHED
    - PTHREAD_EXPLICIT_SCHED
- **schedpolicy**
  - Politique d'ordonnancement
    - SCHED_FIFO, SCHED_RR, SCHED_OTHER

Les attributs d'un thread – Initialisation/destruction

```
int pthread_attr_init (pthread_attr_t *);
```

- **Un attribut doit être initialisé avant de l'utiliser pour créer un thread**
  - Peut servir à créer plusieurs threads

```
int pthread_attr_destroy(const pthread_attr_t *);
```

- **Un attribut non utilisé doit être détruit**
  - Aucun effet sur le thread qui a été créé avec cet attribut

Attributs d'un thread – Politique d'ordonnancement

- **Valeurs : SCHED_FIFO, SCHED_RR, SCHED_OTHER**
- **Politique utilisée**

```
int pthread_attr_getschedpolicy(const pthread_attr_t *,
                                int *);
int pthread_attr_setschedpolicy(pthread_attr_t *, int);
```

- **Paramètres de la politique utilisée <sched.h>**

```
int pthread_attr_setschedparam(pthread_attr_t *attr,
                           const struct sched_param *param);
int pthread_attr_getschedparam(const pthread_attr_t *attr,
                               struct sched_param   *param);
```

- **Héritage**

```
int pthread_attr_getinheritsched(const pthread_attr_t *attr,
                                 int         *inheritedsched);
int pthread_attr_setinheritsched(pthread_attr_t *attr,
                                 int          inheritedsched);
```


Attributs d'un thread – Portée de la compétition

```
int pthread_attr_setscope(pthread_attr_t *, int );

int pthread_attr_getscope(const pthread_attr_t *, int *);
```

- **Portée (scope) de la compétition**
  - Même processus : PTHREAD_SCOPE_PROCESS
  - Processus différents : PTHREAD_SCOPE_SYSTEM

Consulter le man pour plus de détails sur les fonctions gérant les attributs

Attributs d'un thread – Priorité

```
int sched_get_priority_max(int policy) ;

int sched_get_priority_min(int policy) ;
```

- Intervalle de priorité pour la politique utilisée (<sched.h>)
  - Valeur de priorité minimale
  - Valeur de priorité maximale

Exemple – Utilisation d'un attribut

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define VAL 100

/* Fonction exécutée par le thread */
void *f_thread (void *p) {
  int *cr = malloc(sizeof(int));
  printf("\tDebut du thread compagnon");
  printf("\tIci le thread numero: %lu \n", pthread_self());
  *cr = VAL;
  printf("\tValeur retournee: %d \n", *cr);
  pthread_exit((void*)cr);
  /* On pourrait aussi écrire : return ((void *)cr);
  ou return ((void *)&cr); si cr était déclarée en global par : int cr */
}
```


Exemple – Utilisation d'un attribut

```
#include <st
#include <pt
#include <ur
#define VAL

/* Fonction
void *f_thre
  int *cr =
  printf("\t
  printf("\t
  *cr = VAL;
  printf("\t
  pthread_
  /* On pour
  ou return
}
```

```
int main(){
  pthread_t ptid;
  int *res = NULL;
  pthread_attr_t attribut;

  printf("Debut du thread principal\n");
  /* Si le « scope » n'est pas déjà le système par défaut */
  pthread_attr_init(&attribut);
  pthread_attr_setscope(&attribut, PTHREAD_SCOPE_SYSTEM);
  /* Creation du thread compagnon*/
  if ( pthread_create(&ptid, &attribut, f_thread, NULL) != 0){
    perror("Probleme lors de la creation du thread fils:");
    exit(99);  /* plutôt pthread_exit() si hors thread principal */
  }
  pthread_attr_destroy(&attribut);
  /* Attente fin d'execution du thread */
  pthread_join(ptid, (void**)&res);
  printf("Test resultat: %d \n", *res);
  free(res);
  printf("Fin du thread principal\n");
}
```


Exercice 1

On se propose de paralléliser le traitement d'une matrice de réels en confiant le traitement de chaque ligne – calculer la somme des éléments de cette ligne – à un thread.

Le thread initial saisit au clavier le contenu de la matrice, active les threads sous-traitants, puis calcule et affiche la somme des valeurs qu'ils ont calculé

Version 1) Syntaxe d'appel de la commande :

% traiterMatrice

Version 2) Syntaxe d'appel de la commande :

% traiterMatrice NB_LIGNES NB_COLONNES

L'évolution de la version 1 à la version 2 doit entraîner le minimum de modification

On suppose que les fonctions suivantes existent :

void saisirMatrice(float mat[NBLMAX][NBCMAX], int nbL, int nbC);

float sommeLigne(float mat[NBLMAX][NBCMAX], int nbL, int nbC, int numL);

Exercice 2

On se propose de paralléliser le traitement de plusieurs fichiers textes en confiant le traitement de chaque fichier à un thread.

Le traitement effectué par chaque thread consiste à calculer le nombre d'occurrences de chaque caractère alphabétique.

Le thread initial récupère les résultats de ces traitements pour effectuer une synthèse de leurs travaux en affichant :

- le nombre d'occurrences de chaque caractère alphabétique dans l'ensemble des fichiers traités
- pour chaque caractère alphabétique, les noms des fichiers qui n'en contiennent aucune occurrence

Syntaxe d'appel de la commande :

% traiterFichiers nomFichier […]

Réévaluation de l'affectation de l'UC

- Fait passer le thread appelant de l'état actif à l'état prêt, puis élit un nouveau thread
- Erreur
  - Retour : -1 + errno positionné
  - ENOSYS : non supporté

```
#include <pthread.h>
#include <unistd.h>

int pthread_yield (void);
```


Destruction d'un thread (1/2)

- État possible pour une destruction
  - Interdit
  - Autorisé en fonction du type
    - Différé : au prochain point de destruction
    - Asynchrone : n'importe quand ➔ risques
- Points de destruction
  - Tout appel bloquant
  - Certains appels système
  - Précisé par le programmeur
- Demande de destruction d'un thread
  - Nettoyage avant destruction
  - Erreur : ESRCH

```
int pthread_cancel (pthread_t thread);
```


Destruction d'un thread (2/2)

- Positionne l'état de destruction de l'appelant
  - PTHREAD_CANCEL_ENABLE/PTHREAD_CANCEL_DISABLE
  - Erreur : EINVAL

```
int pthread_setcancelstate (int type, int *oldtype);
```

- Positionner le type de destruction de l'appelant
  - PTHREAD_CANCEL_DEFERRED/PTHREAD_CANCEL_ASYNCHRONOUS
  - Erreur : EINVAL

```
void pthread_setcanceltype (int type, int *oldtype);
```

- Positionner un point de « destruction »

```
void pthread_testcancel (void);
```

Threads et signaux (1/2)

- À utiliser avec précaution ➔ 1 seul thread gestionnaire
- Cible :
  - Signaux asynchrones : processus
  - Signaux synchrones (déroutements, matériel) : thread
- Masque des signaux propre, hérité
- État des signaux global : un seul gestionnaire

Threads et signaux (2/2)

- Modification du masque
  - Cf. sigprocmask() mais contexte multi-threadé
  - Erreur : EINVAL

```
int pthread_sigmask(int how, const sigset_t *set,
                                    sigset_t *oset) ;
```

- Envoi restreint au sein du processus
  - thread = récepteur
  - sig = signal (= 0 => contrôle d'erreur seulement)
  - Erreurs : ESRCH, EINVAL

```
int pthread_kill(pthread_t thread, int sig) ;
```