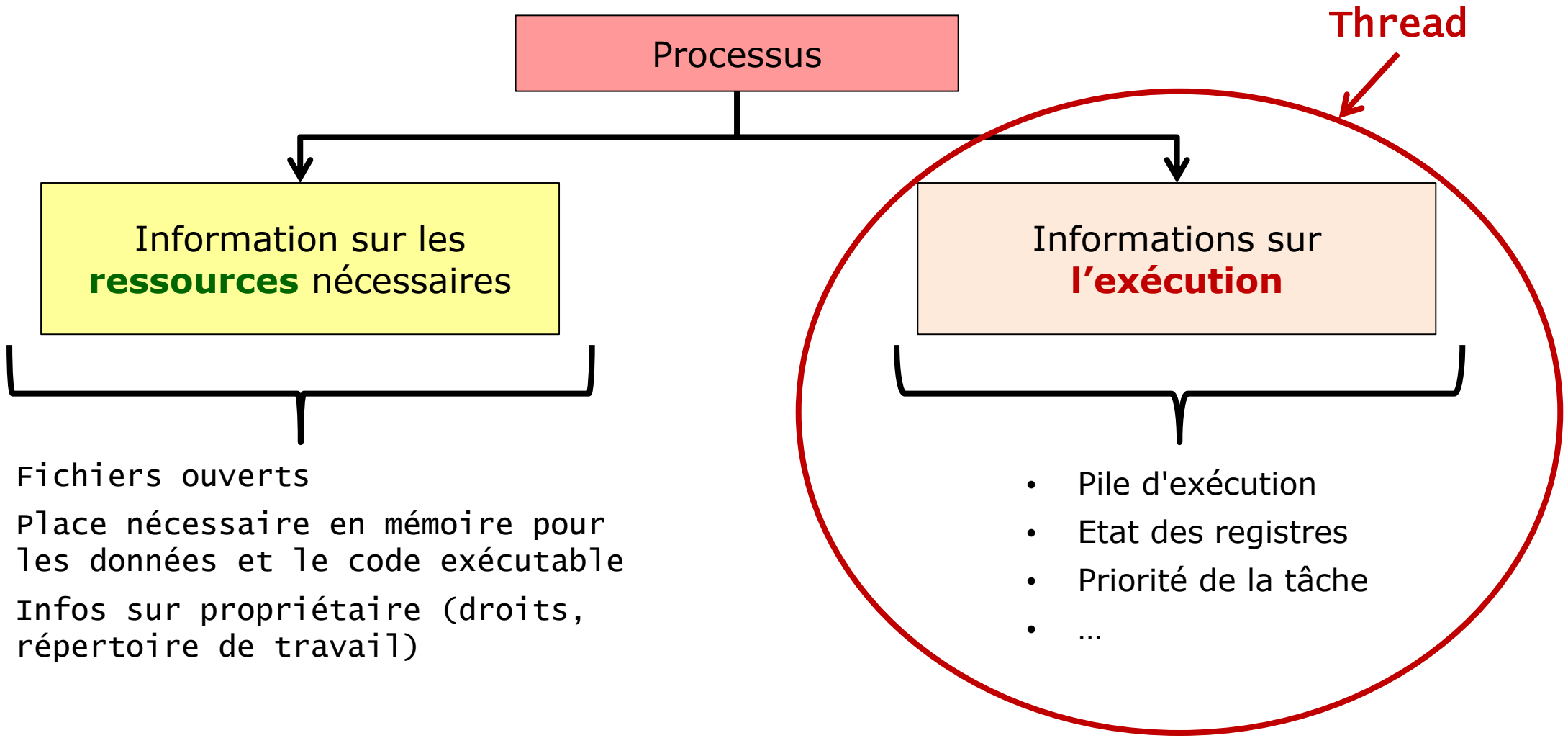
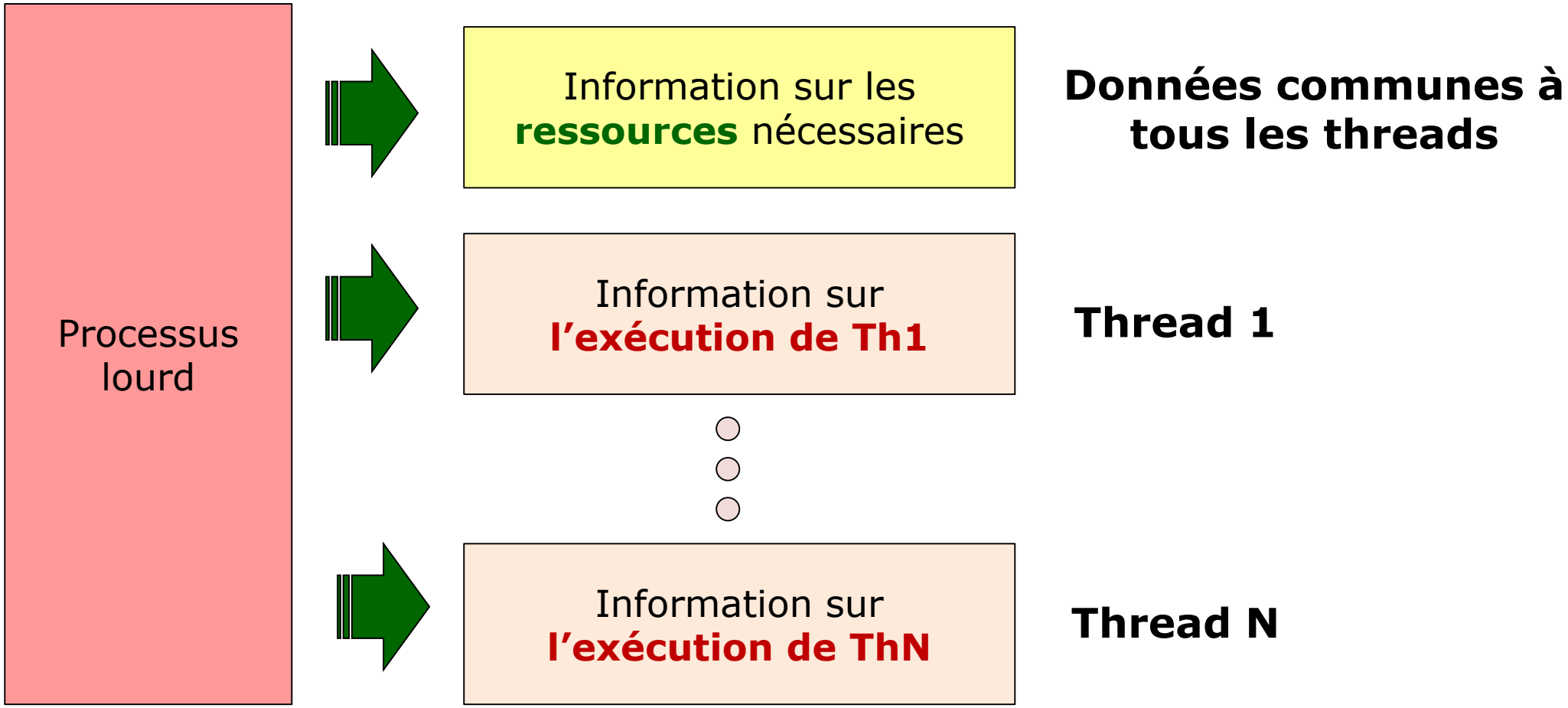
LINUX – LE NOYAU
LES PROCESSUS LEGERS (THREADS)

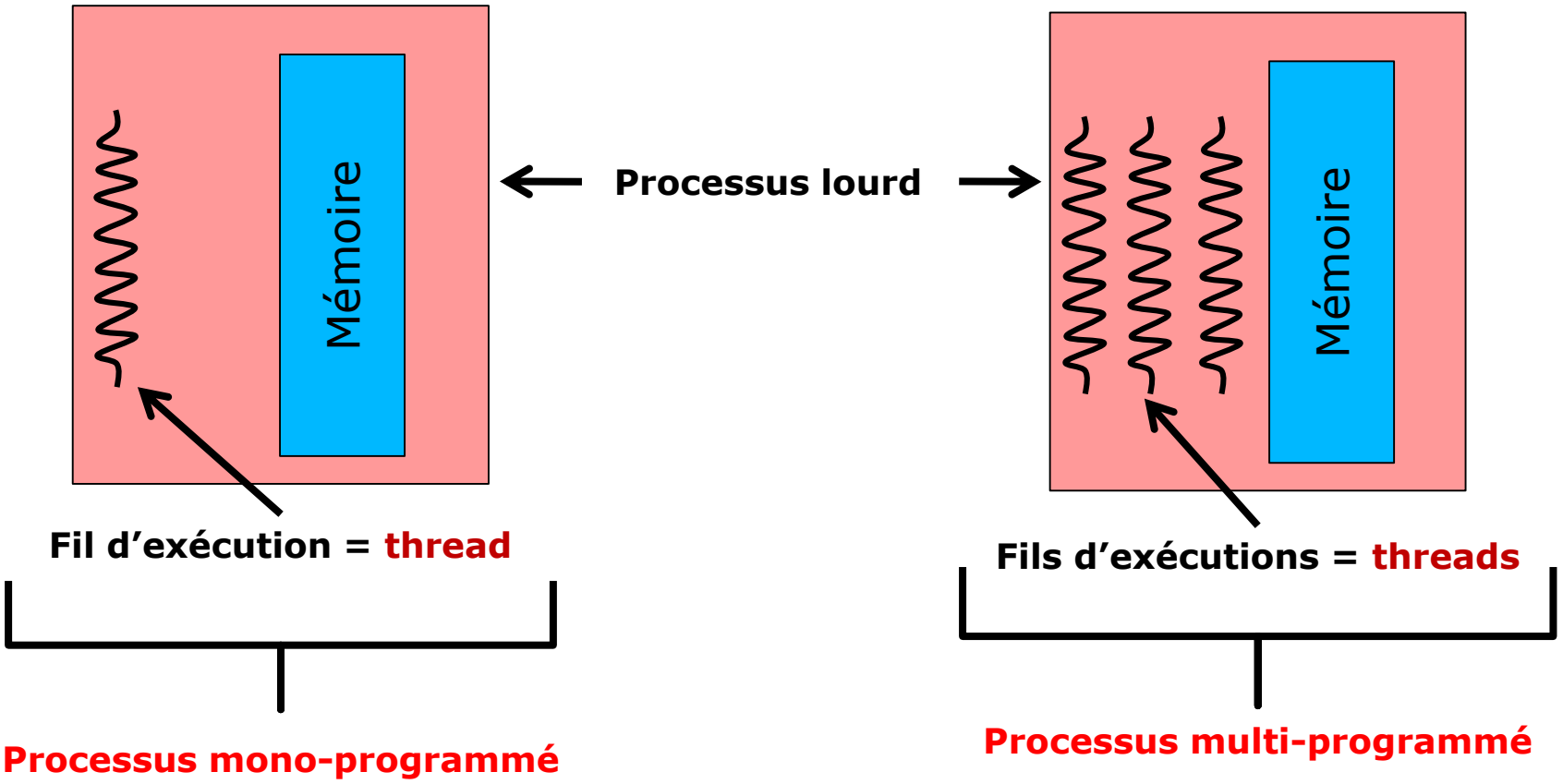
Les threads


Thread
Processus
Information sur les ressources nécessaires
Informations sur l'exécution
• Fichiers ouverts
• Place nécessaire en mémoire pour les données et le code exécutable
• Infos sur propriétaire (droits, répertoire de travail)
• Pile d'exécution
• Etat des registres
• Priorité de la tâche
• …

Les threads = processus légers



Threads et processus



Caractéristiques des threads

- Destinés à la programmation parallèle :
  - Parallélisme réel sur les systèmes multiprocesseurs
  - Parallélisme virtuel (ou concurrence) sur les systèmes monoprocesseurs multitâches

- La création et le changement de contexte d'un processus léger nécessite peu d'opérations
  - 10 à 100 fois meilleurs que dans le cas d'un processus lourd

Intérêts des threads

- Parallélisme
  - Ex multiplication de matrice
- Débit
  - En cas de blocage d'un thread, les autres ne sont pas bloqués
- Temps de réponse
  - Ex : un thread pour chaque commande d'IHM
- Communication :
  - Assurée par la mémoire partagée du processus lourd
- Ressources systèmes
  - Peut créer des milliers de processus légers avec un impact mineur sur les ressources du système

Exemple de threads : multiplication matricielle

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$$

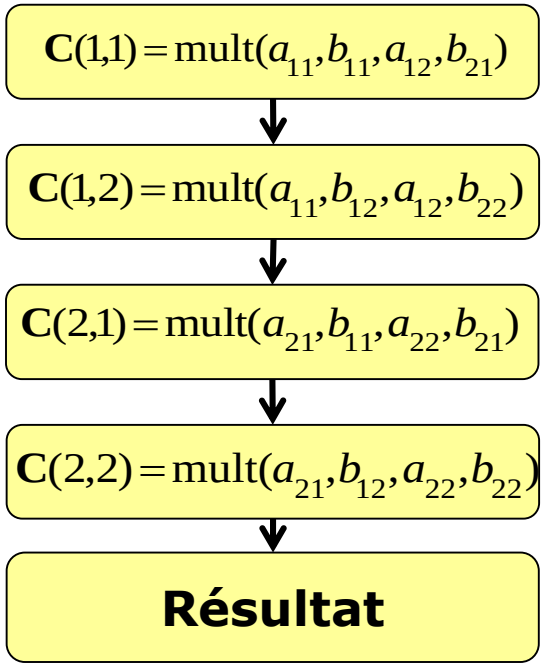
$$\begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

Mise en œuvre avec un seul thread

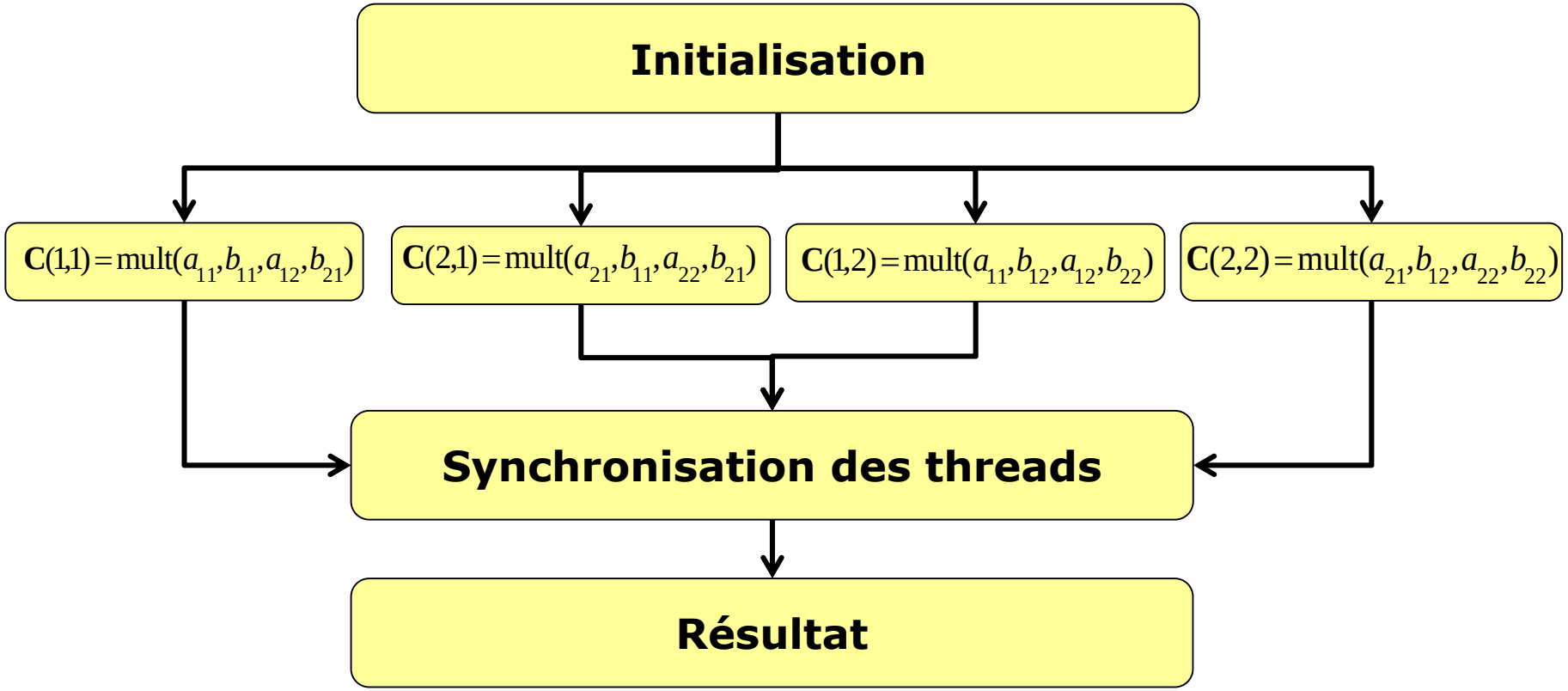
- Fonction élémentaire :

$$\text{mult}(x_1, x_2, x_3, x_4): \quad x_1 \cdot x_2 + x_3 \cdot x_4$$

- Calcul séquentiel :



Mise en œuvre en multithread



Ordonnancement

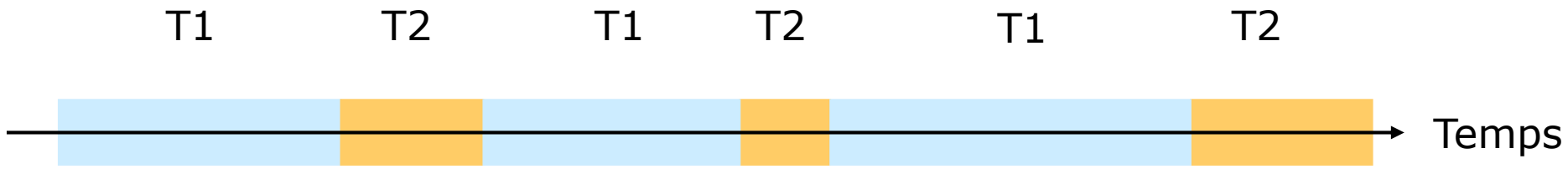
- Problématique
  - A quelle tâche donner le processeur ?
  - Qui gère le processeur pour les tâches ?

- Des critères (parfois contradictoires)
  - Equité : chaque processus doit pouvoir disposer de la ressource processeur
  - Efficacité : utilisation du processeur doit être maximale
  - Temps de réponse : à minimiser pour les utilisateurs
  - Temps d'exécution : à minimiser pour chaque processus
  - Rendement des travaux réalisés par unité de temps

- Différents approches…
  - Ordonnancement non-préemptif (on laisse chaque tâche s'exécuter jusqu'à achèvement)
  - Ordonnancement préemptif (réquisition du processeur)

Ordonnancement non préemptif

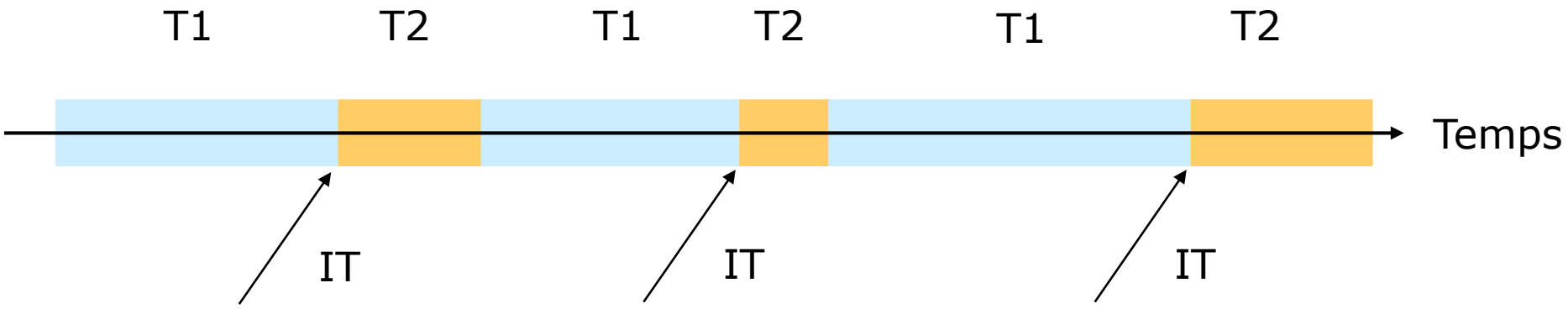
- Un processus quitte le processeur si :
  - **il a terminé son exécution**
  - **il se bloque**



T1 Tâche N°1
T2 Tâche N°2

Ordonnancement préemptif

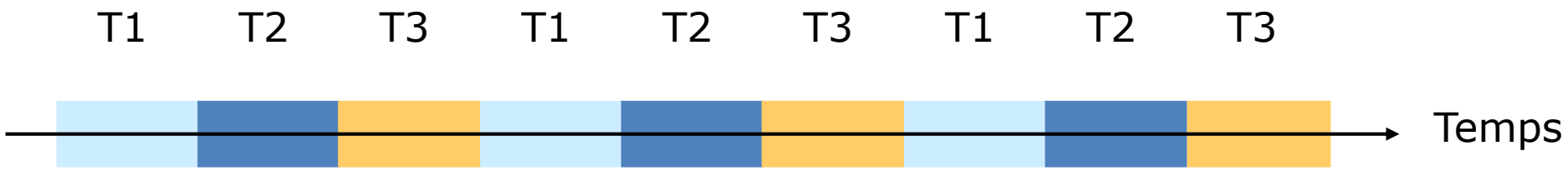
- Un processus quitte le processeur si :
  - **il a terminé son exécution**
  - **il se bloque**
  - **le processeur est réquisitionné** pour une autre tâche (par interruptions)



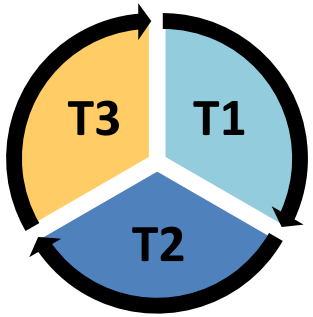
T1 Tâche N°1
T2 Tâche N°2
IT Interruption

Ordonnancement Round Robin (Tourniquet)

- Chaque processus se voit attribuer un quantum de temps
- Si processus non terminé à la fin de son quantum de temps
  - Le processeur est réquisitionné par l'ordonnanceur puis attribué au processus suivant
- Si processus terminé avant la fin du quantum de temps
  - Le processeur est attribué au processus suivant



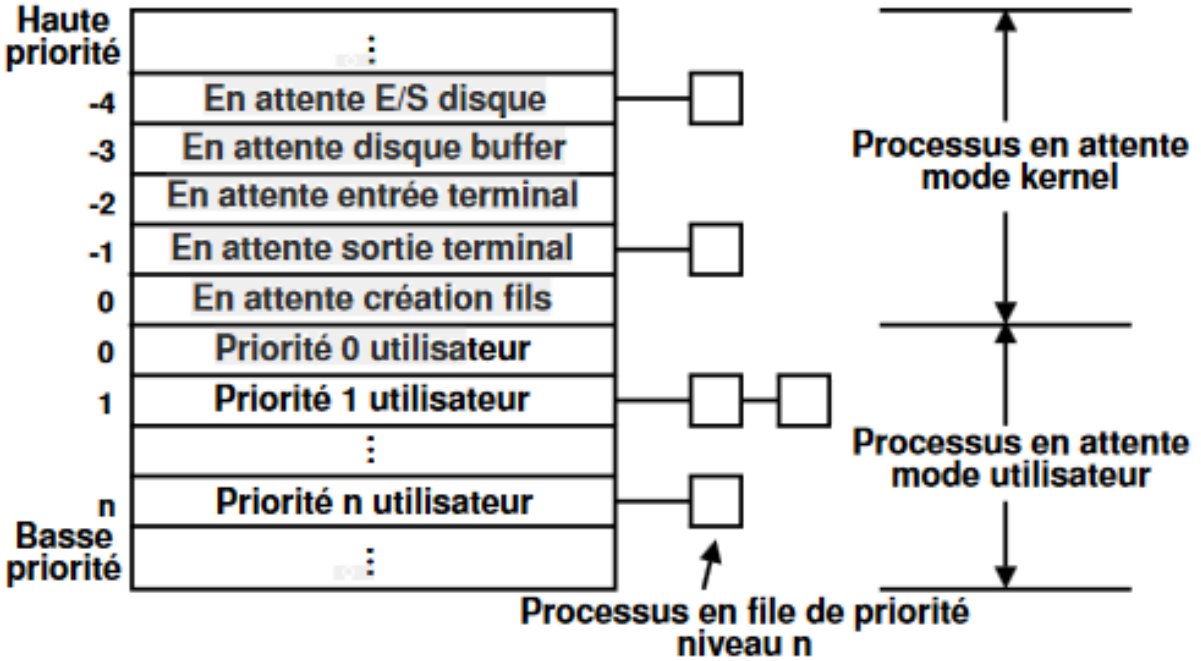
T1 Tâche N°1
T2 Tâche N°2
T3 Tâche N°3



Ordonnancement par priorité

- Mécanisme de priorité afin de favoriser certains processus par rapport à d'autres
  - Dépend par exemple du type de travail et/ou de l'utilisateur
  - Affectation dynamique de la priorité

- Principe
  - Priorité affectée à chaque tâche
  - File d'attente des tâches



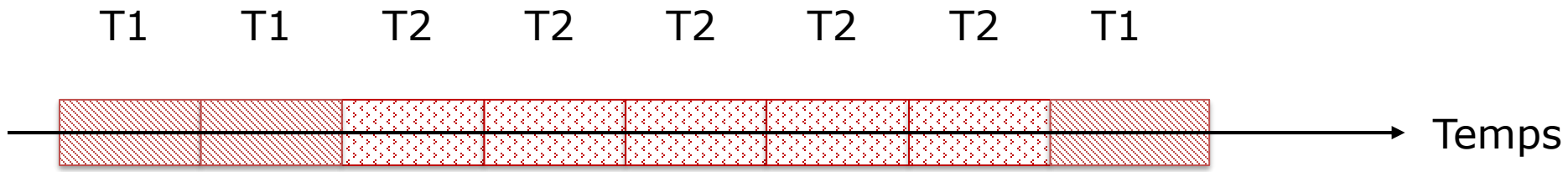
Priorité = Priorité Initiale + Age

- Processeur interrompu régulièrement par une horloge temps réel qui détermine s'il faut commuter de tâche

- Si priorité tâche en cours < priorité tâche en file d'attente
  - Commutation de tâche :
    - Tâche interrompue, réintégrée en file d'attente avec priorité initiale

- Si priorité tâche en cours >= priorité tâche en file d'attente
  - Priorités des tâches de la file d'attente incrémentées

Exemple d'ordonnancement par priorité



Temps	T1	T2	T3	
1	p=60	p=58	p=40	
2	p=60	p=59	p=41	
3	p=55	p=60	p=42	→ Commutation de tâche
4	p=56	p=60	p=43	
5	p=57	p=60	p=44	
6	p=58	p=60	p=45	
7	p=59	p=60	p=46	
8	p=60	p=55	p=47	→ Commutation de tâche

Attention au risque de famine !!!

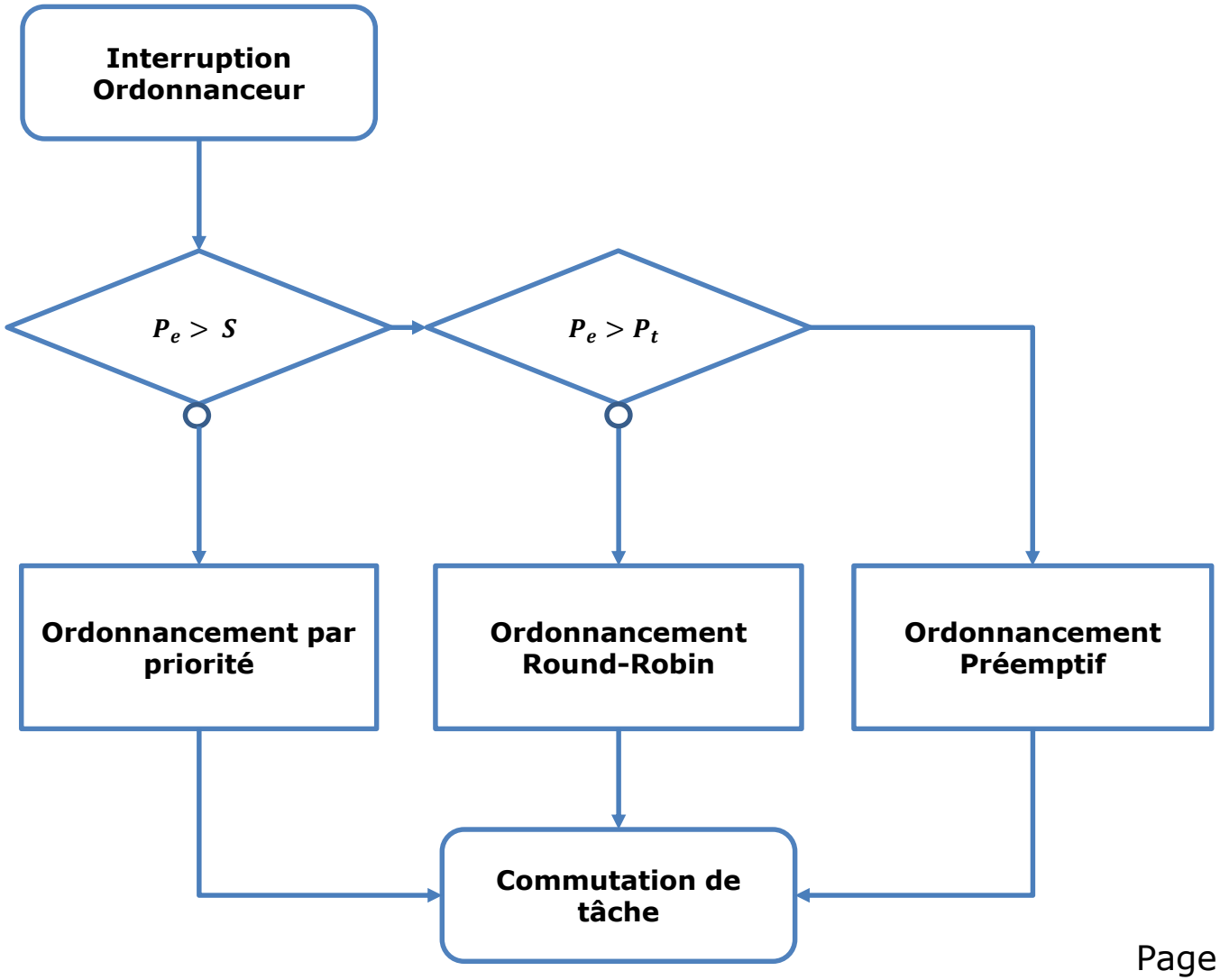
Limitation de l'ordonnancement par priorité

- Le temps de réponse à un événement dépend de la place occupée dans la file d'attente des tâches par la tâche qui doit réagir

- Comment gérer les événements asynchrones, tels que les interruptions ?

Exemple : OS-9

OS-9 : Système d'exploitation
Temps Réel développé par la Société
Microware Systems Corporation



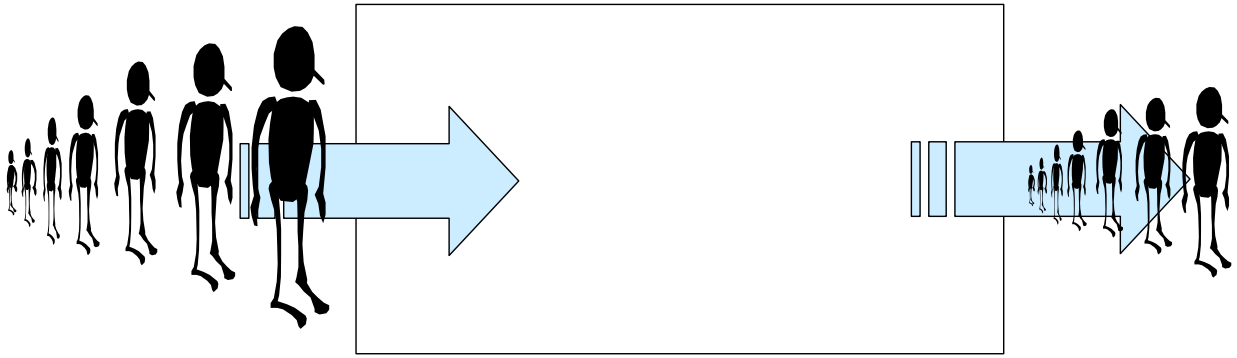
P_e : priorité de l'évènement
S : Seuil de préemption
P_t : priorité de la tâche en cours

Coopération de tâches

- Trois problèmes à résoudre :
  - Partage de ressources
  - Synchronisation avec l'environnement
  - Communication entre tâches

Partages des ressources

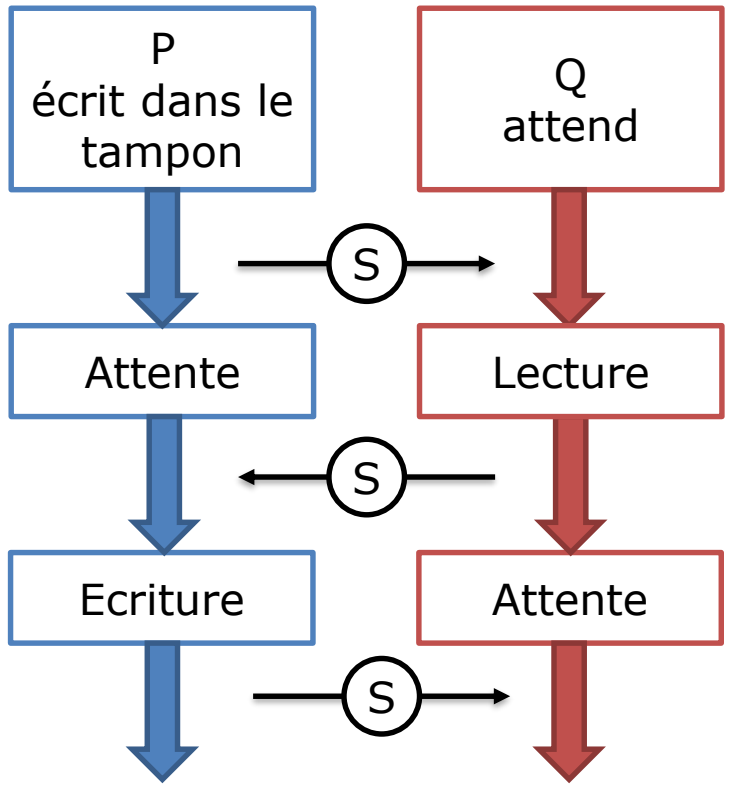
- Exemple :
  - La salle a une taille maximale : il faut compter en continue le nombre de personnes présent dans la salle !



- Solutions :
  - Sémaphores d’exclusion mutuelle
  - Verrous
  - Masquage/Démasquage des interruptions
  - Attente active

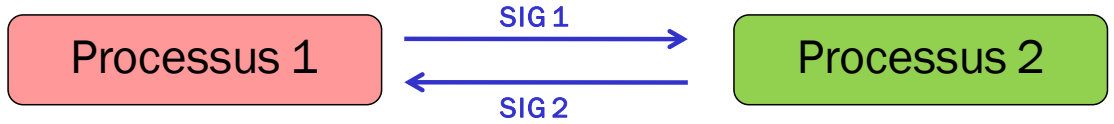
Synchronisation avec l'environnement

- Solutions :
  - Sémaphores
  - Interruptions (Évènements)
  - Rendez-vous
  - Boite aux lettres

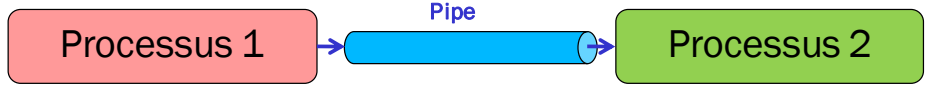


Communications entre tâches

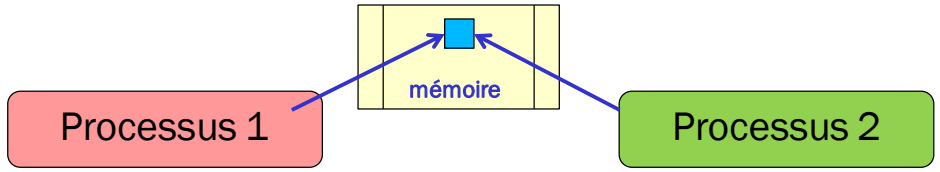
- Signaux
  - "Interruption" signalant un événement système ou autre



- Pipes
  - Canaux de communication pour l'échange de données



- Segments de mémoire partagée



- Messages (boîte aux lettres)

En résumé …

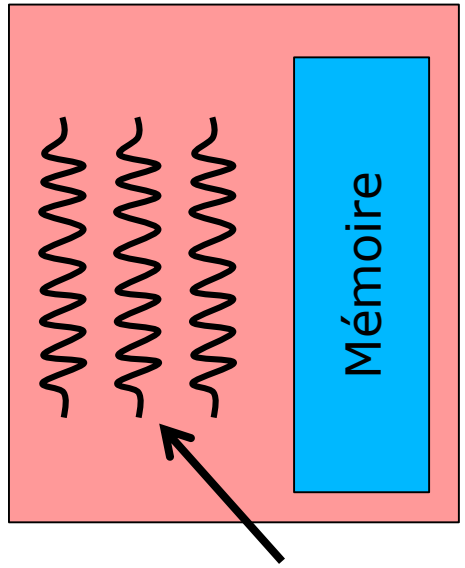
- Le processus est un élément de base du système d'exploitation
  - Permet le multiplexage des ressources matérielles
  - Offre à l'utilisateur une exécution en apparence simultanée des programmes
    - Possibilité de parallélisme au sein d'un processus (Threads)

- Nécessité de partager le ou les processeurs (= section critique)
  - Nécessite un mécanisme d'ordonnancement
    - Pour mettre en œuvre l'ordonnancement choisi, il existe des outils de gestion de tâches

Les threads

- Chaque thread a une copie :
  - Compteur ordinal
  - Registres
  - Pile d'appel
  - Etat ordonnancement

- Les threads se partagent :
  - L'espace d'adressage
  - Les variables globales
  - Les fichiers ouverts
  - Les signaux

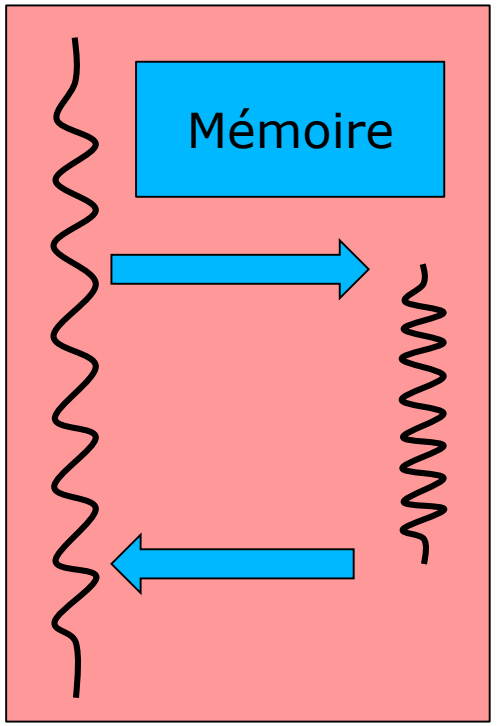


Fils d'exécutions = threads

Implémentation des threads

```
#include <pthread.h>

void routine();
main(){
pthread_t th;
pthread_create(&th,...,
        &routine,...);
 ...
 ...
 ...
pthread_join(th,...);
 ...
}
```



Processus lourd

```
void routine(){
...
 ...
 ...
pthread_exit(...);
}
```

gcc votre_programme.c **-lpthread** -o votre_programme

pthread_create()
Créer un nouveau thread

```
#include <pthread.h>
int pthread_create (      pthread_t *thread,
                          pthread_attr_t *attr,
                          void *(*start_routine)(void *),
                          void *arg
)
```

- **pthread_create()** crée un nouveau thread s'exécutant concurremment avec le processus appelant
  - Le nouveau thread exécute la fonction start_routine en lui passant arg comme premier argument
  - Le nouveau thread s'achève soit explicitement en appelant **pthread_exit()** ou implicitement lorsque la fonction start_routine s'achève.
- L'argument attr indique les attributs du nouveau thread
  - Si **attr=NULL** les attributs par défaut sont utilisés :
    - le thread créé est joignable (non détaché)
    - et utilise la politique d'ordonnancement usuelle (pas de temps-réel)

pthread_join ()
Attendre la mort d'un thread

```
#include <pthread.h>

int pthread_join ( pthread_t th, void **thread_return )
```

- **pthread_join** suspend l'exécution du processus appelant jusqu'à ce que le thread identifié par th achève son exécution soit en appelant **pthread_exit()** soit après avoir été annulé
- Si **thread_return** est différent de NULL, la valeur renvoyée par le thread y sera enregistrée
  - Cette valeur sera :
    - soit l'argument passé à **pthread_exit()**
    - soit PTHREAD_CANCELED si le thread th a été annulé

pthread_exit() fin d'un thread

```
#include <pthread.h>

void pthread_exit ( void *retval )
```

- L'argument retval est la valeur de retour du thread.
- Il peut être consulté par un autre thread en utilisant **pthread_join()**

pthread_kill()
Envoi d'un signal à un thread

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

int pthread_kill ( pthread_t th, int signo )
```

- **pthread_kill()** envoie le signal numéro signo au thread th
- Le signal est reçu et géré tel que décrit dans la fonction **kill()**

Un exemple simple

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

void* func1(){
  printf("Salut, ");
  pthread_exit(NULL);
}

void* func2(){
  printf("ca va ?");
  pthread_exit(NULL);
}
```

```c
int main ( ) {
 pthread_t th1, th2;
 pthread_create(&th1, NULL, func1, NULL);
 pthread_create(&th2, NULL, func2, NULL);
 pthread_join(th1, NULL );
 pthread_join(th2, NULL );
 return 0;
}
```

```
>>./a.out
>>Salut, ca va ?
>>./a.out
>> ca va ? Salut,
```

gcc votre_programme.c **-lpthread** -o votre_programme

Un exemple simple

Un exemple plus complexe

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

struct data{int a;int b;};

Void *add(void *arg){
  int c;
  int* ret = malloc(sizeof(int));
  struct data*  p = (struct data*)arg;
  c = p->a + p->b;
  *ret = c;
  pthread_exit((void*)ret);
}
```

```c
int main ( ) {
  void *ret;
  int *p;
  struct data d;
  pthread_t th;
  d.a = 1;
  d.b = 2;
  pthread_create(&th, NULL, add, (void*)&d);
  pthread_join(th, &ret );
  p = (int*)ret;
  printf("%d+%d=%d\n",d.a,d.b,*p);
  return 0;
}
```

```
>>./a.out
>>1+2=3
```

Un exemple plus complexe

La fonction sched_yield()

- Permet d'abandonner la CPU pour la donner à un autre thread (ou un autre processus)
- Attention : il n'existe pas de préemption de la CPU à l'intérieur des threads d'un même processus
  - En clair, si un thread garde la CPU, les autres threads ne vont pas s'exécuter !
- Cette fonction permet de programmer un partage équitable de la CPU entre threads coopératifs

La programmation multithread

C'est bien !

- Structure modulaire du code reflétée à l'exécution
- Gain conceptuel
  - Système réactif
  - Programmation événementielle
  - Modèle client/serveur
  - Thread spontané
- Gain de temps
  - Appels bloquants
  - Création peu coûteuse
  - Commutation rapide
  - Gestionnaire mémoire non sollicité

La programmation multithread

Mais c'est risqué !

- On travaille en mémoire partagée : attention aux effets de bord !
  - Utiliser des fonctions réentrantes
    - Une fonction est dite réentrante si elle peut être appelée ”simultanément” par plusieurs threads et renvoyer pour chacun le résultat identique à celui attendu en contexte mono-thread.
  - Utiliser des verrous / sémaphores

- Variable modifiée de façon aléatoire par rapport au déroulement général du programme
  - Penser à déclarer les variables comme "volatile"