Programmation Concurrente

Principes et concepts
2024-2025

Chapitre1: Introduction

Evaluation

- Minimum un travail écrit,
- Un ou deux laboratoires par semestre
- Bonus pour la participation en cours

«Descriptif de module **RS430.100.21.2247.3 »**

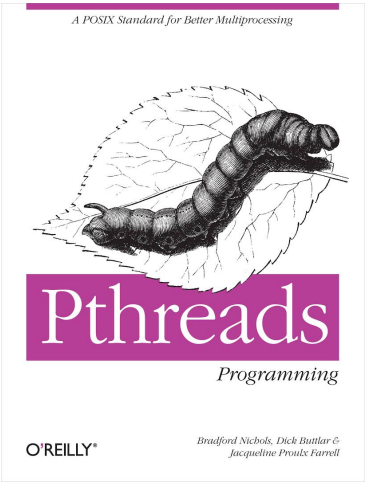
Références

https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html

https://docs.oracle.com/cd/E26502_01/html/E35303/tlib-1.html

Cours du professeur Florent Gluck HES-SO

https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html



Plan de la matière

1. Définition de la programmation concurrente
2. Approche de l'attente Active, algorithme Dekker et Peterson
3. Exclusion mutuelle et section critique
4. Synchronisation
5. Paradigmes
   - Producteurs-Consommateurs
   - Lecteurs-Rédacteurs
   - Philosophes
6. Outils
   - Verrous
   - Sémaphores
   - Moniteurs
7. Langage C, C++

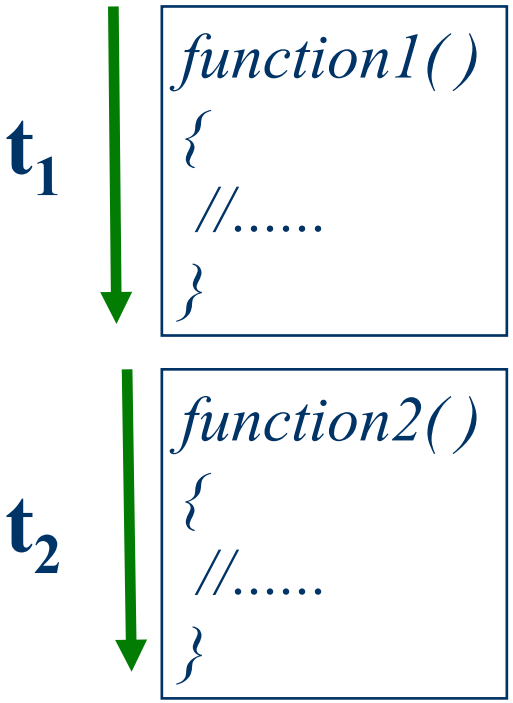
Introduction

- Définition de la programmation Concurrente
- Exemples d'applications
- Fonctionnement et problèmes de la concurrence
- Patterns
- Changement de contexte

Rappel: Paradigmes Programmation

- Séquentielle
  - Le programme comporte un seul thread
- Concurrente
  - Deux ou plusieurs threads qui coopèrent
    - Application multi-contexte (multithread)
    - Application parallèle
    - Application distribuée
- Temps réel
  - La notion de temps est très importante

Programmation séquentielle(1)



C'est d'abord la function1 ( ) qui s'exécute dans le temps et après la function2 ( )

Programmation séquentielle(2)

- Un programme exécute des instructions séquentiellement dans un ordre déterminé
- L'exécution est **prévisible**
- L'exécution est **reproductible**
- Le développeur connait très bien la suite des opérations

Programmation Concurrente(1)

La function1 ( ) et la function2 ( ) s'exécutent en même temps.

Définition: la programmation concurrente

- La programmation concurrente est un paradigme de programmation tenant compte, dans un programme, de plusieurs contextes d'exécutions(thread, processus, tâches) matérialisés par une pile d'exécution(stack) et des données privés. Alors que dans la programmation séquentielle, on trouve la dépendance causale: Une instruction s'exécute après une autre
- La concurrence(indépendance causale): plusieurs instructions s'exécutent en même temps
- Un processus est sous forme de plusieurs threads

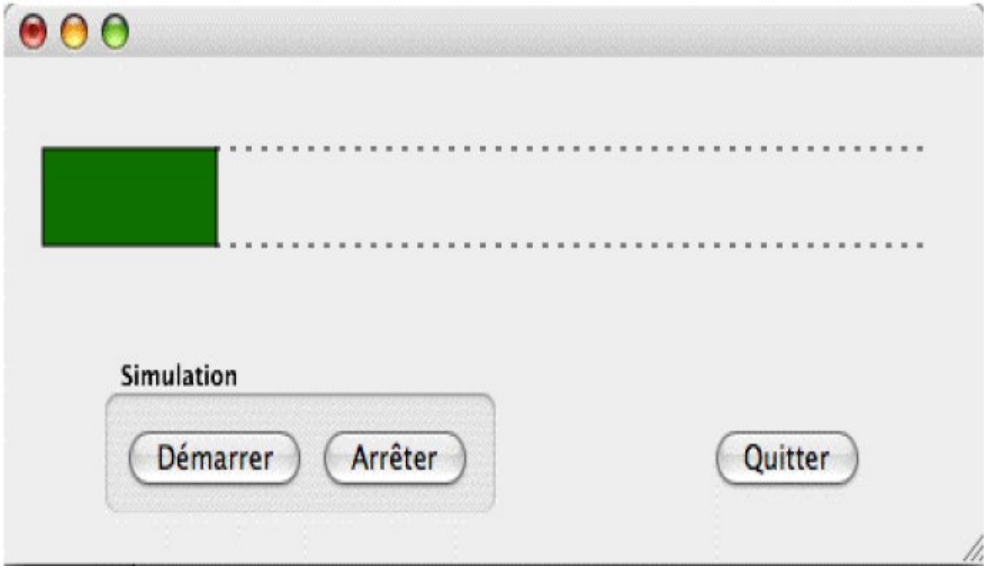
Expressivité : facilité l'écriture d'algorithmes
- **séparation des tâches, explicitation de la communication, . . .**

Exemples

- Jeu vidéo: un thread s'occupe de l'affichage, un autre de l'intelligence, un autre du réseau, un autre du son.etc
- Simulateurs, Calculs mathématiques, etc.
- Serveur web: Chaque client est géré par un thread
- …etc

Exemple : Barre de progression

- Ici la barre de progression s'exécute en même temps que l'affichage de l'interface graphique et le traitement derrière.



Pourquoi de la concurrence ?

- Optimiser l'utilisation du/des processeurs

Pourquoi de la concurrence ?

- Optimiser l'utilisation du/des processeurs
- Eviter de bloquer sur des entrées/sorties

Pourquoi de la concurrence ?

- Optimiser l'utilisation du/des processeurs
- Eviter de bloquer sur des entrées/sorties
- Exploiter l'avantages des multi-cœurs

Pourquoi de la concurrence ?

- Optimiser l'utilisation du/des processeurs
- Eviter de bloquer sur des entrées/sorties
- Exploiter l'avantage des multi-coeurs
- Augmenter le parallélisme(Répartition des calculs)

Pourquoi de la concurrence ?

- Optimiser l'utilisation du/des processeurs
- Eviter de bloquer sur des entrées/sorties
- Exploiter l'avantage des multi-cores
- Augmenter le parallélisme(Répartition des calculs)
- Attendre sur plusieurs entrées

Pourquoi de la concurrence ?

- Optimiser l'utilisation du/des processeurs
- Eviter de bloquer sur des entrées/sorties
- Exploiter l'avantage des multi-cores
- Augmenter le parallélisme(Répartition des calculs)
- Attendre sur plusieurs entrées
- Certains programmes sont naturellement multitâche(Animation)

Pourquoi de la concurrence ?

- Optimiser l'utilisation du/des processeurs
- Eviter de bloquer sur des entrées/sorties
- Exploiter l'avantage des multi-cores
- Augmenter le parallélisme(Répartition des calculs)
- Attendre sur plusieurs entrées
- Certains programmes sont naturellement multitâche(Animation)

Pourquoi de la concurrence ?

- Optimiser l'utilisation du/des processeurs
- Eviter de bloquer sur des entrées/sorties
- Exploiter l'avantage des multi-cores
- Augmenter le parallélisme(Répartition des calculs)
- Attendre sur plusieurs entrées
- Certains programmes sont naturellement multitâche(Animation)
- Simplifier la structure d’un programme

Pourquoi de la concurrence ?

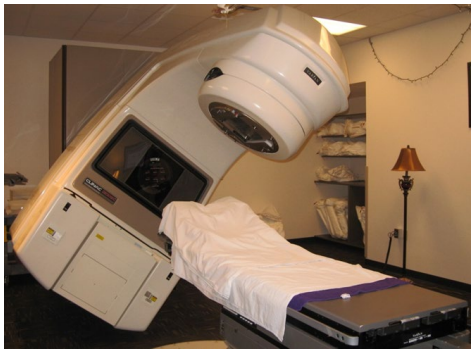
- Optimiser l'utilisation du/des processeurs
- Eviter de bloquer sur des entrées/sorties
- Exploiter l'avantage des multi-cores
- Augmenter le parallélisme(Répartition des calculs)
- Attendre sur plusieurs entrées
- Certains programmes sont naturellement multitâche(Animation)
- Simplifier la structure d'un programme
- Attendre des événements

Difficultés ?

- Partage des ressources et leurs gestions
  - Les ressources peuvent être :Mémoires(octet, mot, bloc…), le temps processeur. les évènements, un fichier….
- Difficulté à synchroniser les tâches
  - Le contrôle de l'ordre d' exécution des tâches est très important
- Prédictibilité
  - C'est difficile de prévoir le moment d'exécution de la tâche
- Reproductibilité
  - C'est difficile de reproduire la même séquence d'exécution des tâches

Accidents(1)

- Accidents dû à des bugs de programmation concurrente :
  - Thérapie par radiation : appareil Therac-25 (1984-87)



«la tâche qui gérait le contrôle du matériel souffrait de problèmes de concurrence avec la tâche qui gérait l'interface destinée à l'opérateur. Une condition de concurrence (*race condition*) apparaissait si l'opérateur changeait les paramètres trop rapidement. Ce problème ne fut pas détecté lors des tests puisqu'il fallait un certain temps aux opérateurs avant qu'ils ne parviennent à utiliser l'interface de manière aisée»

https://fr.wikipedia.org/wiki/Therac-25#Incident_de_juin_1985

Accidents(2)

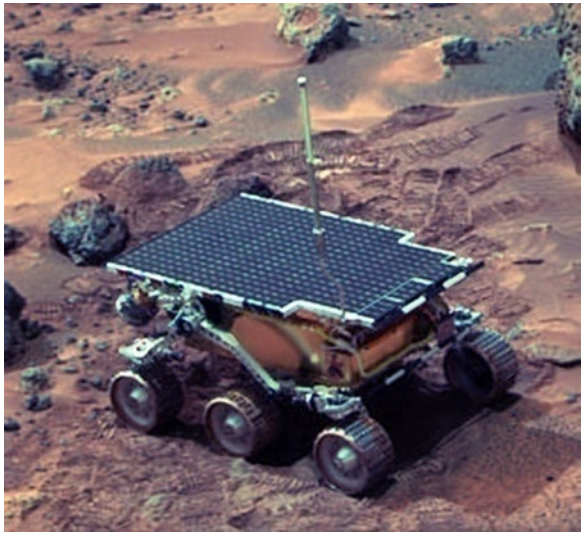
- Blackout dans le nord-est des USA en 2003 (55 mio de personnes touchées)

A software bug known as a race condition existed in General Electric Energy's Unix-based XA/21 energy management system.[13] Once triggered, the bug stalled FirstEnergy's control room alarm system for over an hour. System operators were unaware of the malfunction. The failure deprived them of both audio and visual alerts for important changes in system state

https://en.wikipedia.org/wiki/Northeast_blackout_of_2003

Accidents(3)

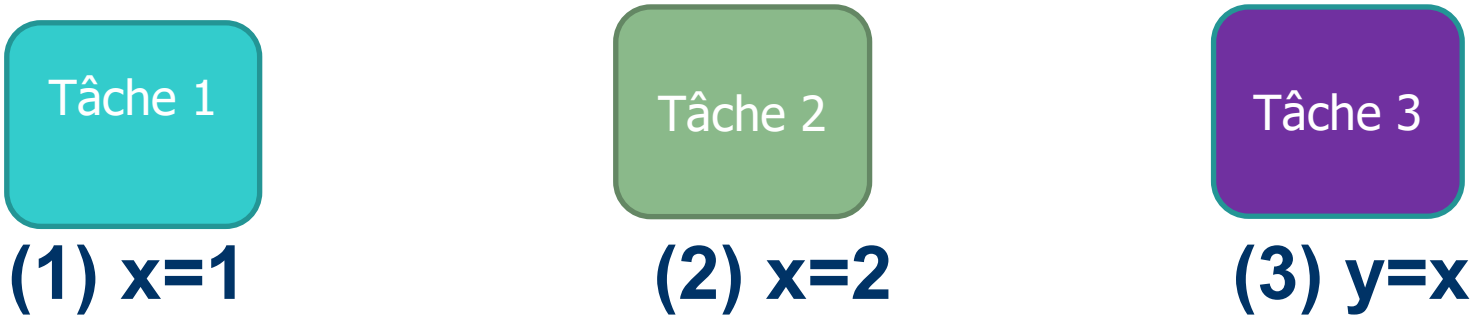
- Mars Pathfinder (1997) : envoyé sur mars pour analyser le climat et la géologie de la planète rouge.
- Progression interrompue par des «resets» fréquents dû à un bug d'inversion de priorité



https://fr.wikipedia.org/wiki/Mars_Pathfinder

Comportement non déterministe(1)

- Supposons que x est initialiser à 0, et que les trois tâches s' exécutent en même temps.



- Quel est la valeur finale de y ?

Comportement non déterministe(2)

- Supposons que x est initialiser à 0, et que les trois tâches s' exécutent en même temps.

- Quel est la valeur finale de y ?

y=0
ou
y=1
ou
y=2

Thread/Processus

- Un processus peut contenir un ou plusieurs threads
- Un thread est une unité d'exécution ou processus léger (*ligthweight process*)
- Les threads s'exécutent en parallèle
  - Sur monoprocesseur: chacun à son tour
  - Sur un multiprocesseur: peuvent être réellement parallèles
- C'est l'ordonnanceur qui est responsable sur l'ordre d'exécution
- Problèmes:
  - Priorité d'exécution
  - Echange des données
  - Cohérence des données
  - Synchronisation et communication entre les threads

Modèles de codages des threads

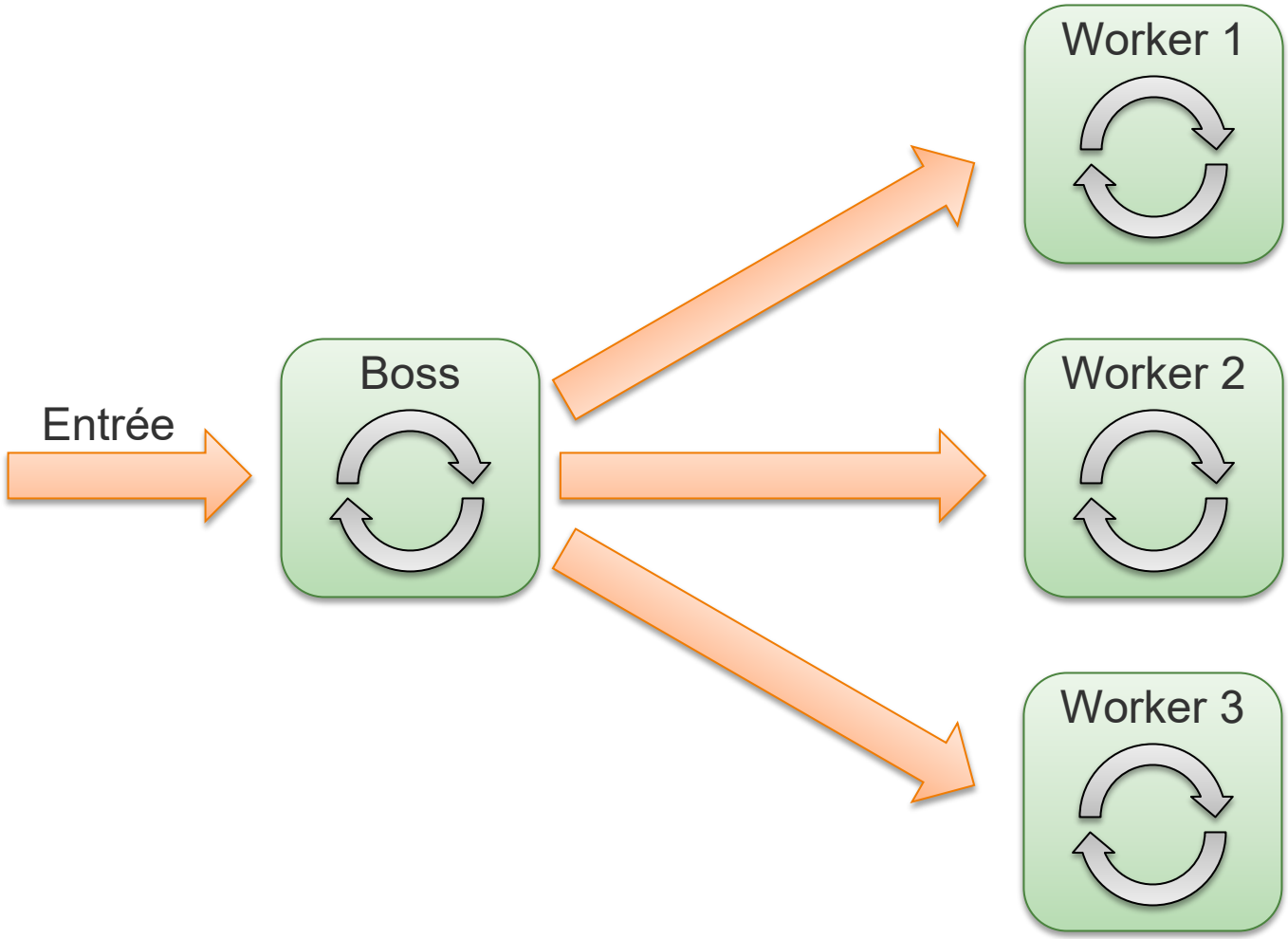
- Comment décomposer un programme en plusieurs Threads ?

- En tous cas 3principaux
  - Le modèle de délégation(boss/worker model)
  - Le modèle pair
  - Le modèle pipeline

Boss/Worker model = Délégation(1)

- Délégation:
  - Un *Thread* principal crée les autres *threads* et leur assigne une tâche.
  - Il peut aussi attendre jusqu'à ce que la tâche soit complétée.

Boss/Worker model = Délégation(2)



Boss/Worker model = Délégation(3)

Approche-1 le boss crée pour chaque requête un nouveau thread

```
void *Boss(void *) {
  boucle infinie {
    attend une requête
    switch (requete) {
      case requeteX: pthread_create( ... workerX);
        break;
      case requeteY: pthread_create( ... workerY);
        break;
      ...
    }
  }
}

void *workerX(void *) {
  exécute le travail demandé, puis se termine
}

void *workerY(void *) {
  exécute le travail demandé, puis se termine
}
```

Boss/Worker model = Délégation(4)

- Approche-2 le boss crée un pool de threads

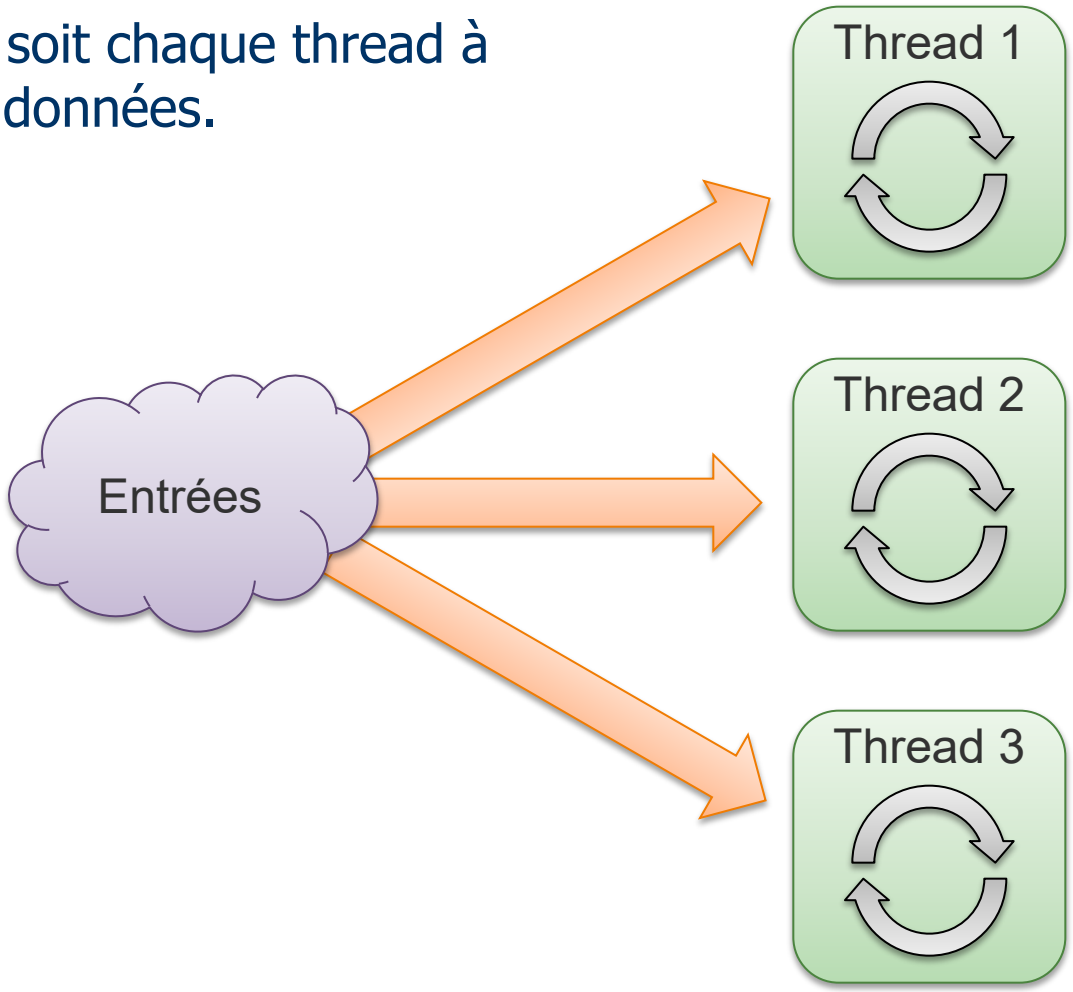
```
void *master(void *) {
  // crée tous les threads
  pthread_create(...);
  boucle infinie {
    attend une requête;
    place la requête dans la file d'attente
    signale aux travailleurs qu'une requête est prête
  }
}
void *worker(void *) {
  boucle infinie {
    bloque jusqu'à être activé par le maître
    récupère la requête de la file d'attente
    switch(requete){
      case requeteX: workerX();
        break;
      case requeteY: workerY();
        break;
      ...
    }
  }
}
```

Programmation séquentielle(1)

- Peer to Peer :
  - Tous les *threads* ont un statut de travail égal
  - Un thread pair crée tous les *threads* nécessaires à une tâche mais il ne délègue pas des responsabilités.
  - Les threads pair peuvent traiter des requêtes d'un simple '*input stream*' partagé par tous les threads ou avoir chacun leur propre '*input stream*'.

Peer to Peer (pair)(2)

- Deux variantes soit avec des données en entrées en commun soit chaque thread à sa propre entrée de données.



Peer to Peer (pair)(3)

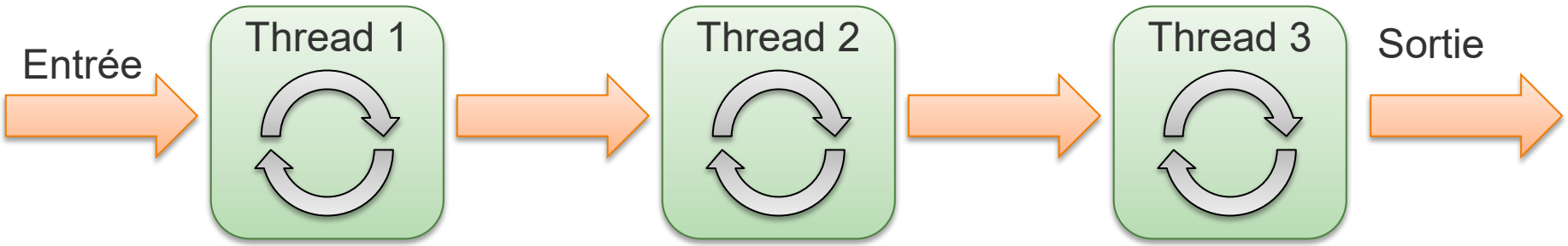
```
main() {
  pthread_create( ... thread1);
  pthread_create( ... thread2);
  ...
  signale aux threads qu'ils peuvent commencer à travailler
}

thread1() {
  attend le signal de commencement
  effectue le traitement, et synchronise
           avec les autres threads si nécessaire
}

thread2() {
  attend le signal de commencement
  effectue le traitement, et synchronise
           avec les autres threads si nécessaire
}
```

Pipeline(1)

- Approche semblable à une ligne de production pour traiter des données par étapes (*stages*).

- Chaque *stage* est un *thread* qui fait du travail sur une unité d'entrée. Lorsque l'unité d'entrée passe par tous les stages, le traitement est complété.

Pipeline(2)

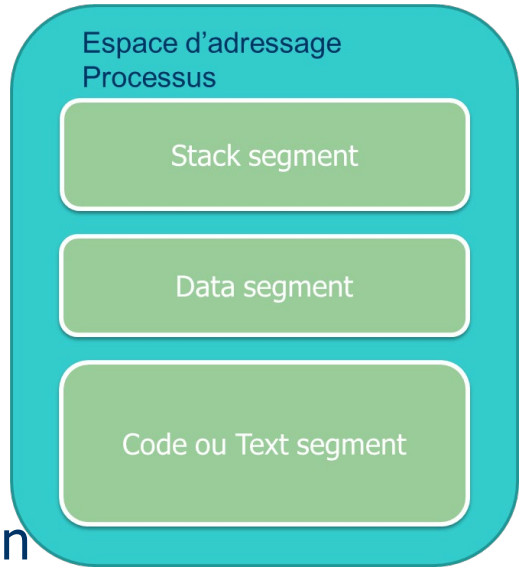


Pipeline(3)

```
stage1() {
  boucle infinie {
    récupérer une entrée du prog.
    traiter cette donnée
    passe résultat à étage suivant
  }
}

stage2() {
  boucle infinie {
    récupérer donnée étage précédent
    traiter cette donnée
    passer résultat à étage suivant
  }
}

stageN() {
  boucle infinie {
    récupérer donnée étage précédent
    traiter cette donnée
    passer résultat en sortie du prog.
  }
}
```

Processus VS thread(1), un processus

- Un processus possède entre autres
  - Un code à exécuter
  - Un espace d'adressage
  - Une priorité
  - Un identifiant
  - Un contexte d'exécution (PC + registres)
- Les processus sont gérés par le système d'exploitation
- Plusieurs processus peuvent s'exécuter en parallèle



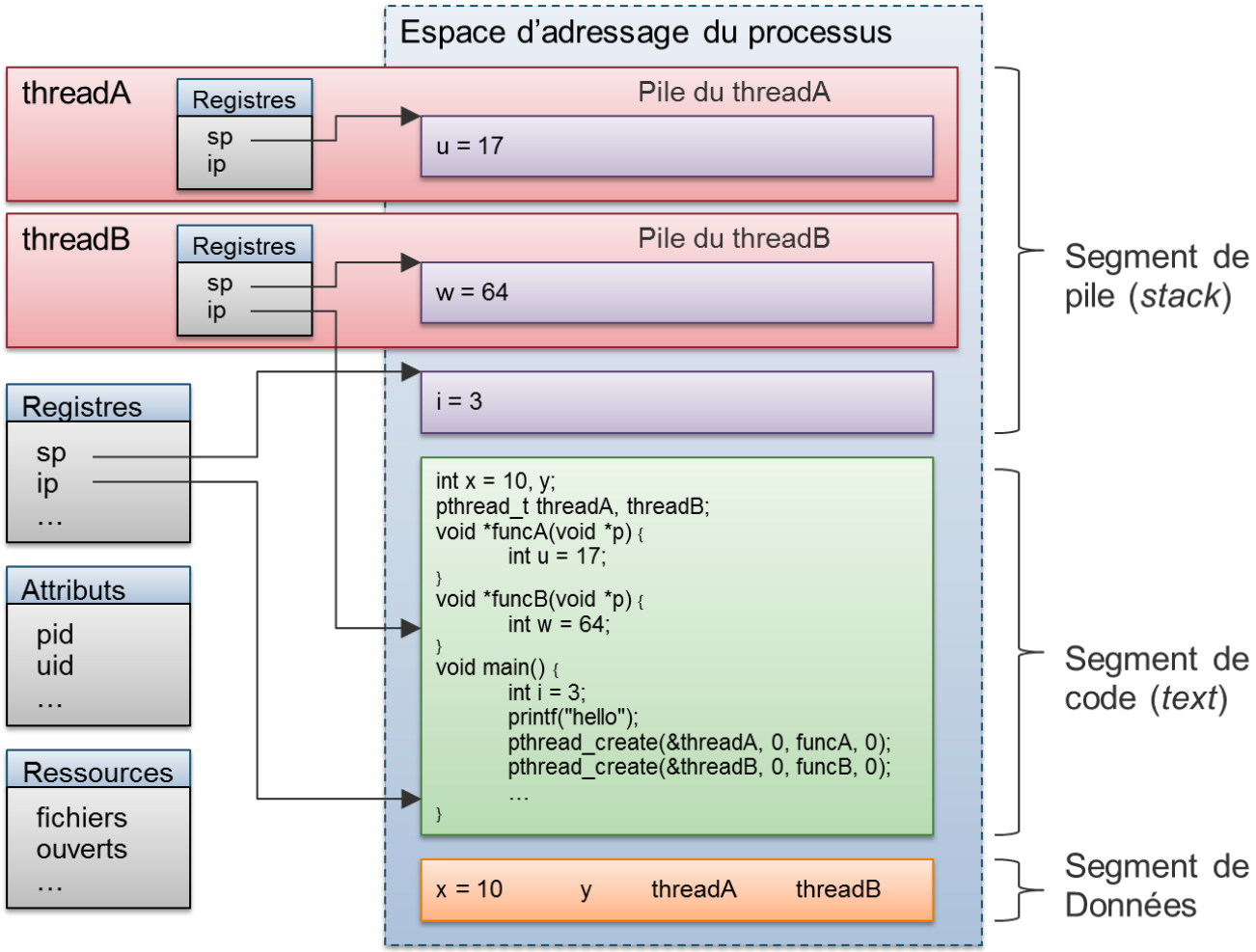
Processus VS thread(2), un thread

- Les threads d'un même processus se partagent l'espace d'adressage du processus
- Ils sont ordonnancés
- Ils possèdent
  - leur propre pile
  - leur propre contexte

  d'exécution (IP(Instruction Pointeur) + registres)
- Ils ont un cycle de vie semblable à celui d'un processus

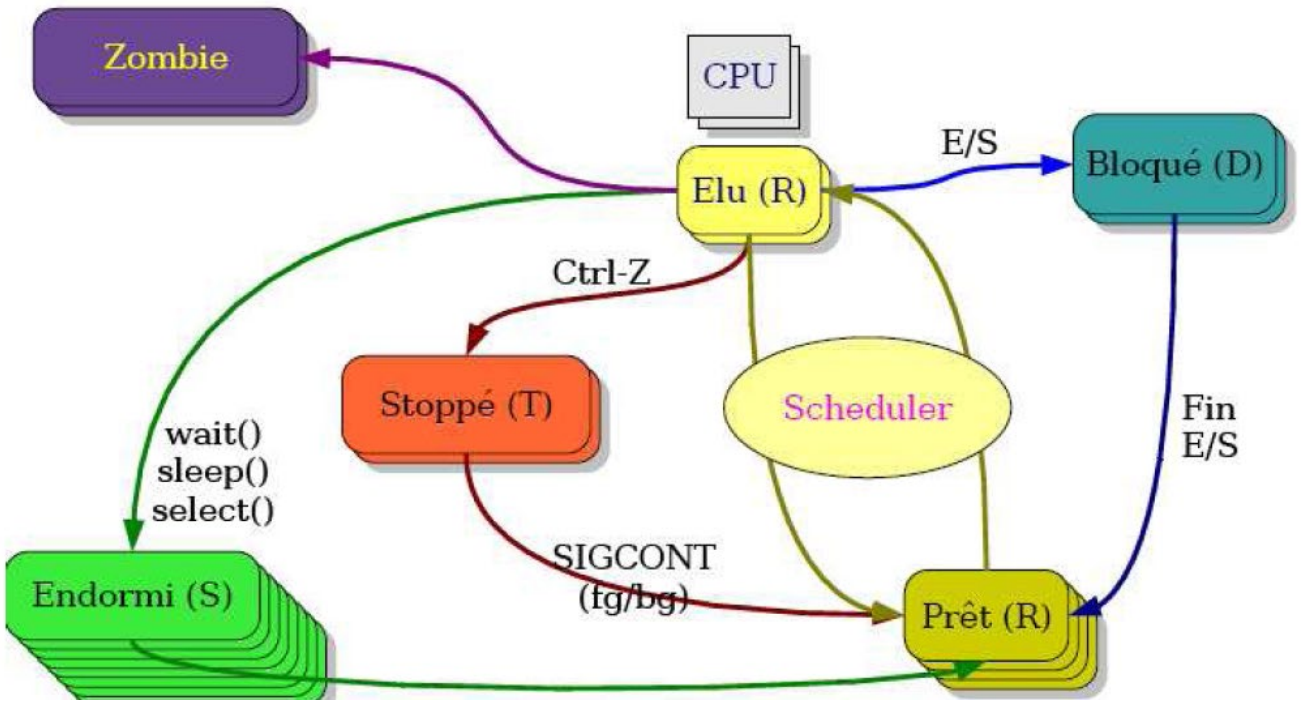
Processus VS thread(3)

- Les threads d'un même processus se partagent l'espace d'adressage du processus
- Ils sont ordonnancés
- Ils possèdent
  - leur propre pile
  - leur propre contexte d'exécution (IP+ registres)
- Ils ont un cycle de vie semblable à celui d'un processus



Processus VS thread(4)

- Cycle de vie d'un processus



Processus VS thread(5), un processus

- **Prêt:**

Le processus est prêt à être exécuté. Cas d'un processus nouvellement créé, débloqué ou, d'un ou plusieurs processus occupant le ou les processeurs disponibles.

- **En exécution:**

Le processus est en cours d'exécution sur un processeur. Plusieurs processus peuvent être en exécution dans le cas d'une machine multiprocesseur.

- **Bloqué:**

Le processus est en attente sur une synchronisation ou sur la fin d'une opération d'entreé/sortie par exemple.

- **Zombie :**

Le processus a terminé son exécution, mais son processus parent doit encore récupérer sa valeur de terminaison.

- **Terminé:**

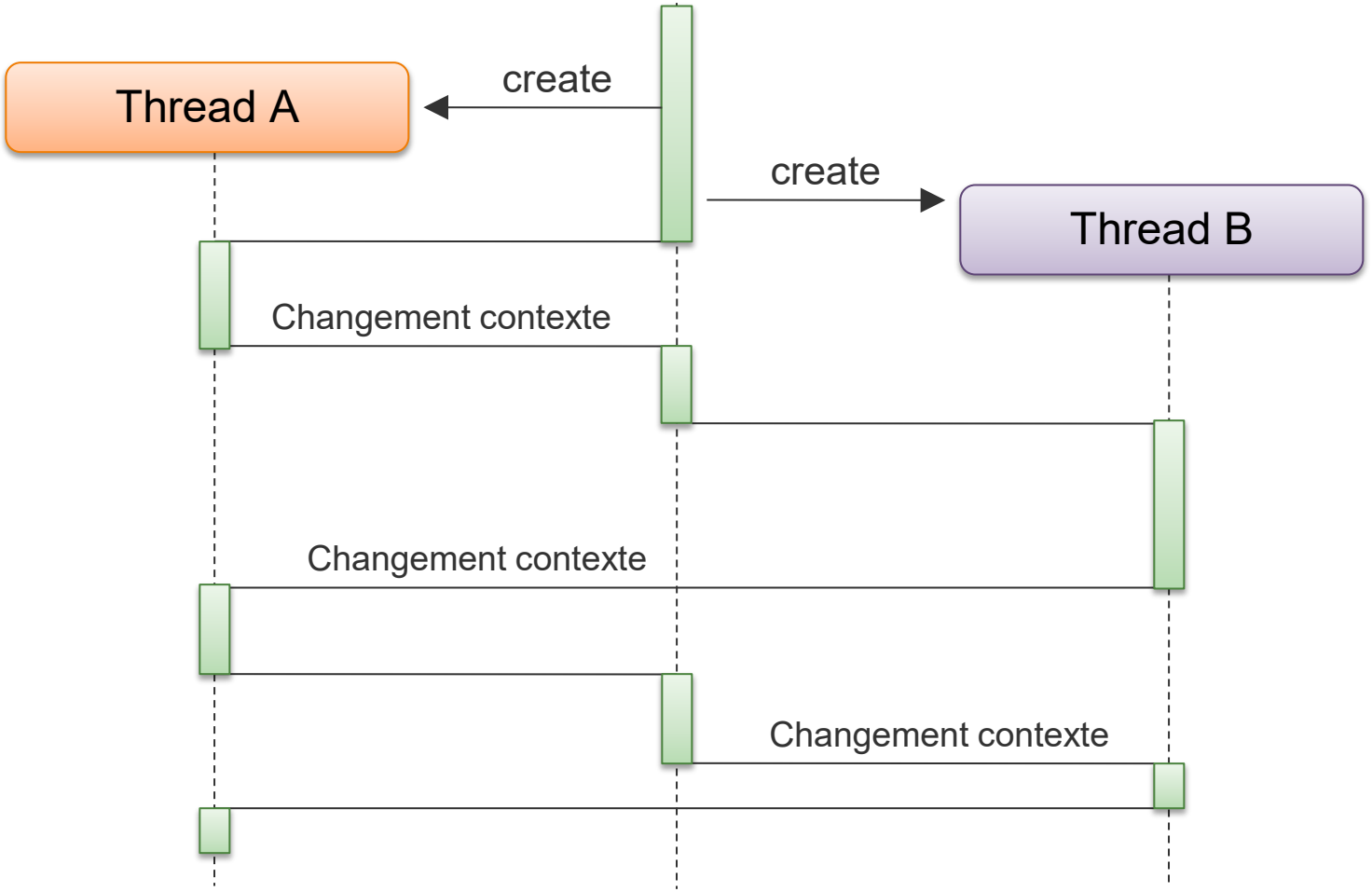
Le processus a terminé son exécution ou a été annulé (cancelled). Les ressources du processus seront libérées et le processus disparaîtra.

Changement de contexte(1)

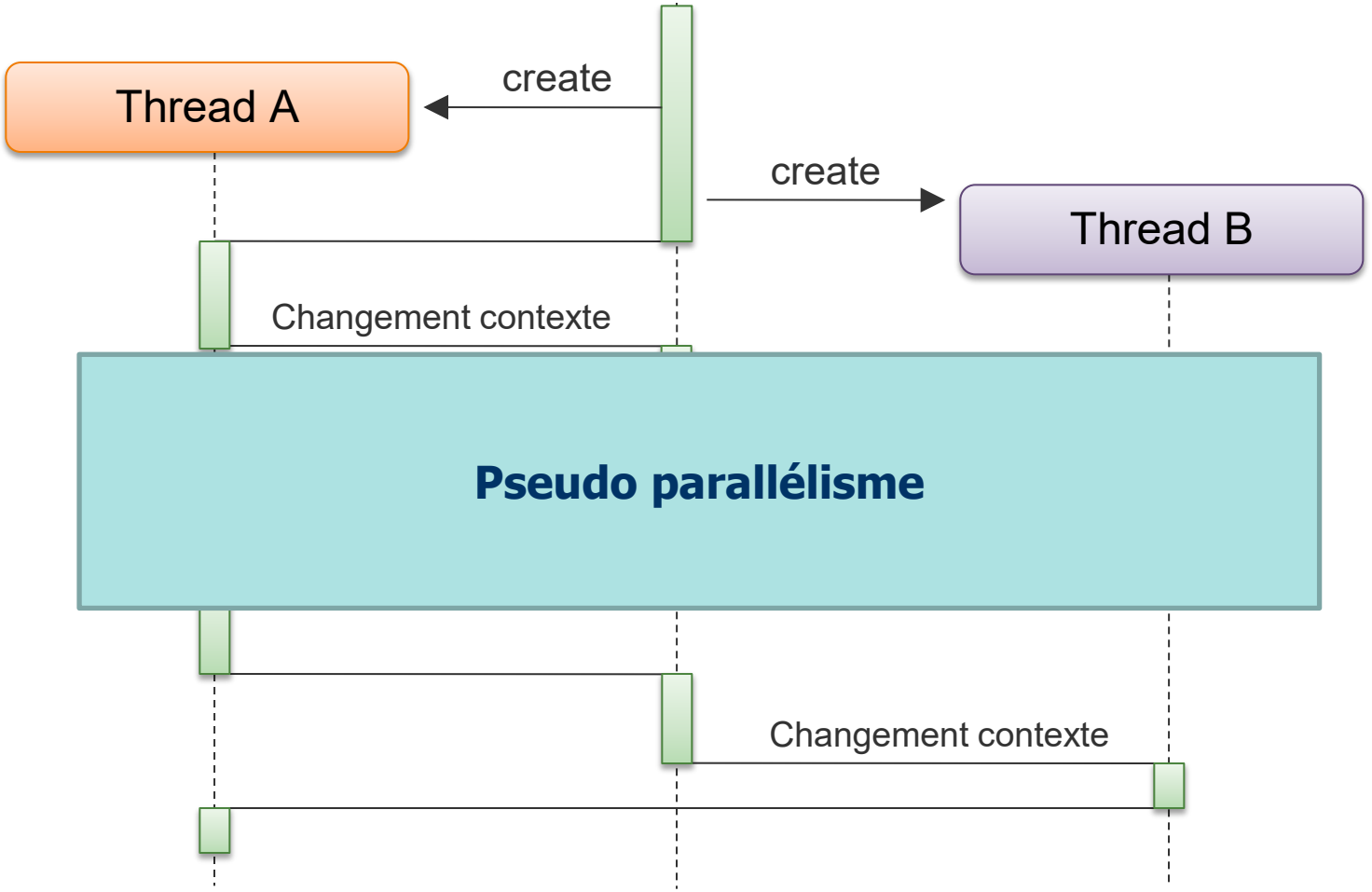
- L'opération de changement de contexte d'un processus (ou thread) comporte les séquences suivantes :
- Mise en attente du processus actif dans la liste des processus bloqués ou prêts (fin du quantum de temps)
- Sauvegarde de son contexte d'exécution
- Recherche du processus éligible ayant la plus haute priorité
- Restauration du contexte d'exécution du processus élu, restauration de la valeurs de ses registres lorsqu'il s'exécutait précédemment
- Activation du processus élu.

Tout se passe comme si le processus préalablement interrompu n'avais pas cessé de s'exécuter

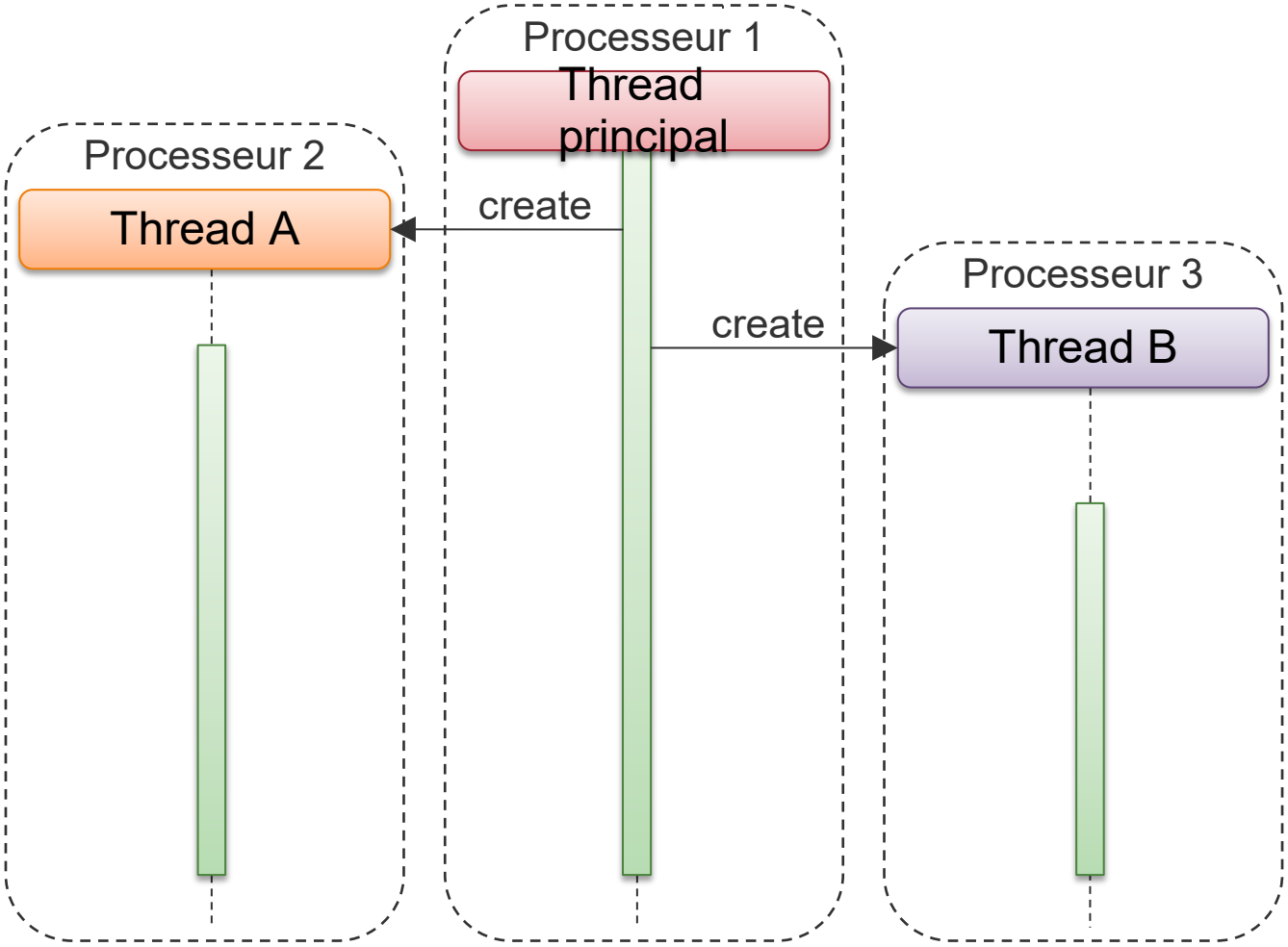
Changement de contexte(2)



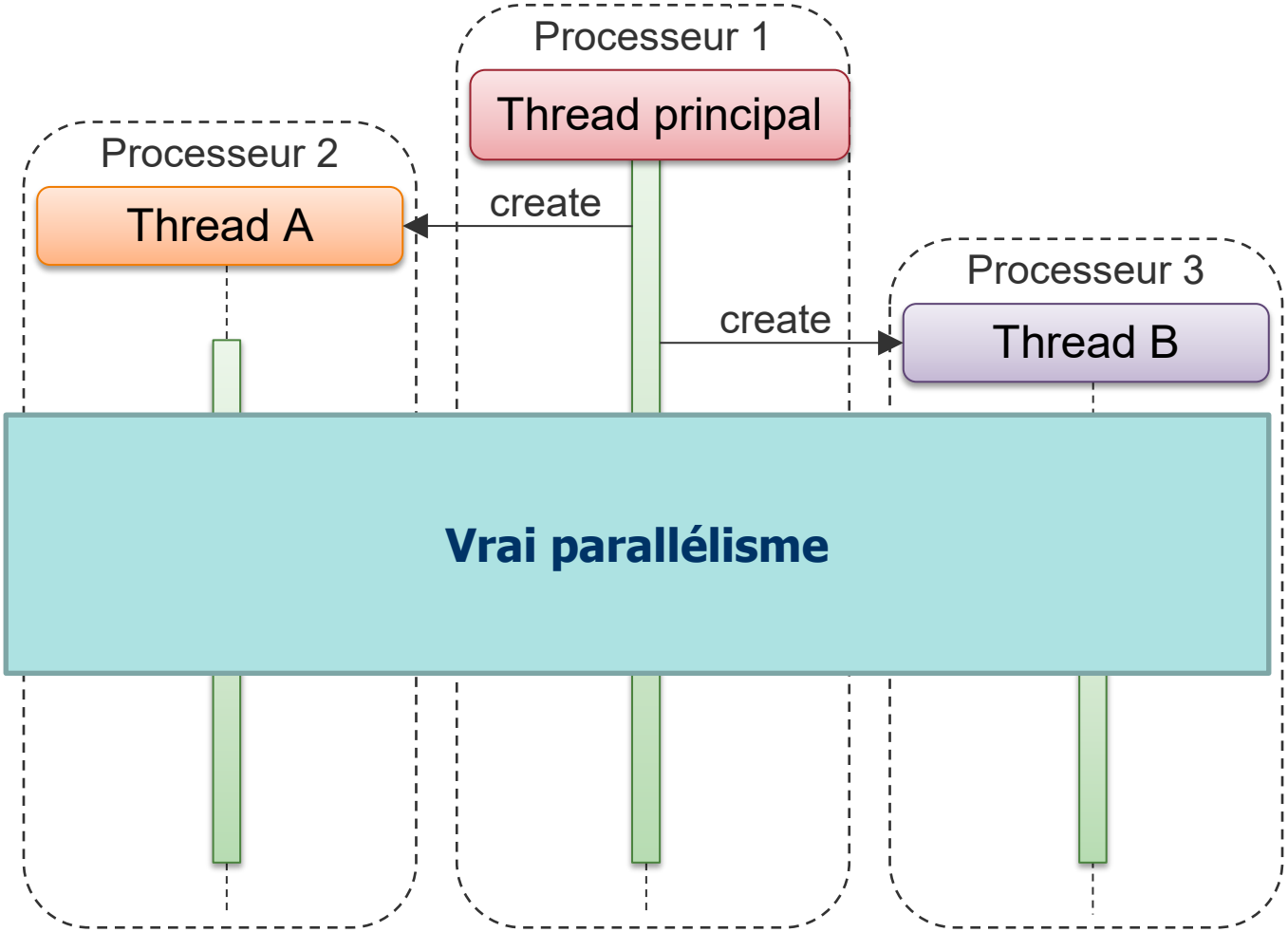
Changement de contexte(3)



Changement de contexte(4)



Changement de contexte(5)



Changement de contexte(Exemple)

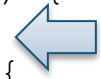
```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

	Prêt	Actif	Bloqué
T1		X	
T2	X		
T3	X		

T1

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```



IP courant

IP sauvegardé

	Prêt	Actif	Bloqué
T1		X	
T2	X		
T3	X		

T1

Programmation séquentielle(1)

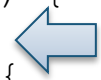
```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

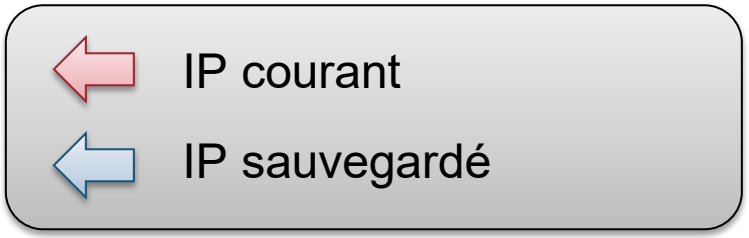
	Prêt	Actif	Bloqué
T1		X	
T2	X		
T3	X		

T1

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```





	Prêt	Actif	Bloqué
T1		X	
T2	X		
T3	X		

T1

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

	Prêt	Actif	Bloqué
T1		X	
T2	X		
T3	X		

Préemption de T1 par T2

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

	Prêt	Actif	Bloqué
T1	X		
T2		X	
T3	X		

T2

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```



	Prêt	Actif	Bloqué
T1	X		
T2		X	
T3	X		

T2

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

	Prêt	Actif	Bloqué
T1	X		
T2		X	
T3	X		

T2

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```



	Prêt	Actif	Bloqué
T1	X		
T2		X	
T3	X		

T2

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```




IP courant


IP sauvegardé

	Prêt	Actif	Bloqué
T1	X		
T2		X	
T3	X		

Préemption de T2 par T3

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```




IP courant


IP sauvegardé

	Prêt	Actif	Bloqué
T1	X		
T2	X		
T3		X	

T3

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

IP courant

IP sauvegardé

	Prêt	Actif	Bloqué
T1	X		
T2	X		
T3		X	

T3

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

IP courant

IP sauvegardé

	Prêt	Actif	Bloqué
T1	X		
T2	X		
T3		X	

T3

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```




IP courant


IP sauvegardé

	Prêt	Actif	Bloqué
T1	X		
T2	X		
T3		X	

T3 passe en attente

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

IP courant

IP sauvegardé

	Prêt	Actif	Bloqué
T1	X		
T2	X		
T3			X

T3 passe en attente

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

IP courant

IP sauvegardé

	Prêt	Actif	Bloqué
T1		X	
T2	X		
T3			X

Ordonnanceur active T1

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

IP courant

IP sauvegardé

	Prêt	Actif	Bloqué
T1		X	
T2	X		
T3			X

T1

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

IP courant

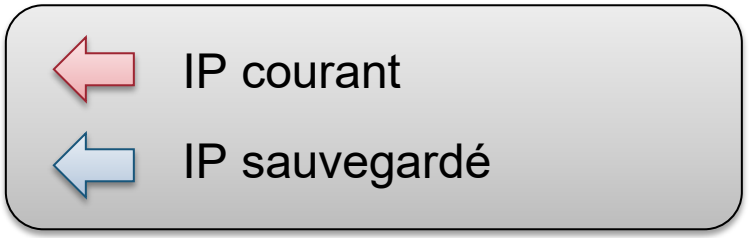
IP sauvegardé

	Prêt	Actif	Bloqué
T1		X	
T2	X		
T3			X

T1

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```



	Prêt	Actif	Bloqué
T1		X	
T2	X		
T3			X

T1

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

	Prêt	Actif	Bloqué
T1		X	
T2	X		
T3			X

T1

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

IP courant

IP sauvegardé

	Prêt	Actif	Bloqué
T1		X	
T2	X		
T3			X

Préemption de T1 par T2

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

IP courant

IP sauvegardé

	Prêt	Actif	Bloqué
T1	X		
T2		X	
T3			X

T2

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

IP courant

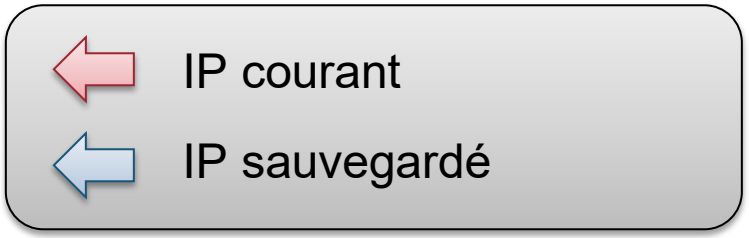
IP sauvegardé

	Prêt	Actif	Bloqué
T1	X		
T2		X	
T3			X

T2

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```



	Prêt	Actif	Bloqué
T1	X		
T2		X	
T3			X

T2

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

IP courant

IP sauvegardé

	Prêt	Actif	Bloqué
T1	X		
T2		X	
T3			X

T2

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

IP courant

IP sauvegardé

	Prêt	Actif	Bloqué
T1	X		
T2		X	
T3			X

T2

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

IP courant

IP sauvegardé

	Prêt	Actif	Bloqué
T1	X		
T2		X	
T3			X

Préemption de T2 par T1

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

IP courant

IP sauvegardé

	Prêt	Actif	Bloqué
T1		X	
T2	X		
T3			X

T1

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

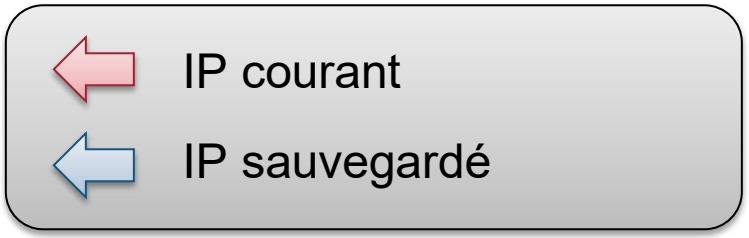


	Prêt	Actif	Bloqué
T1		X	
T2	X		
T3			X

T1

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```



	Prêt	Actif	Bloqué
T1		X	
T2	X		
T3			X

T1

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

IP courant

IP sauvegardé

	Prêt	Actif	Bloqué
T1		X	
T2	X		
T3			X

1 seconde s'est écoulée

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

IP courant

IP sauvegardé

	Prêt	Actif	Bloqué
T1		X	
T2	X		
T3			X

Préemption de T1 par T3

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

IP courant

IP sauvegardé

	Prêt	Actif	Bloqué
T1	X		
T2	X		
T3		X	

T3

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```




IP courant

IP sauvegardé

	Prêt	Actif	Bloqué
T1	X		
T2	X		
T3		X	

T3

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```

IP courant

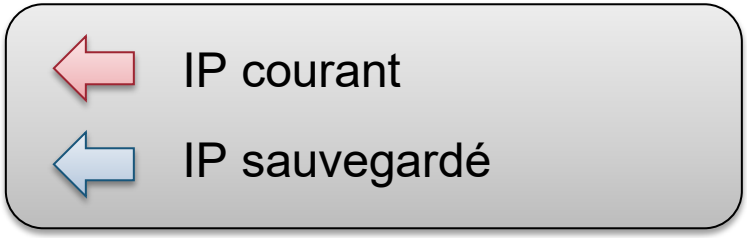
IP sauvegardé

	Prêt	Actif	Bloqué
T1	X		
T2	X		
T3		X	

T3

Changement de contexte(Exemple)

```
void *T1(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
  }
  return NULL;
}

void *T2(vo
  int i = 0
  while (i
    i = i +
  }
  return NU
}

void *T3(void *arg) {
  int i = 0;
  while (i < 100) {
    i = i + 1;
    sleep(1);
  }
  return NULL;
}
```



Et l'histoire continue.....

		oqué	
T3		X	

T3

Pointeurs (1)

- Un pointeur est une variable contenant une adresse comme valeur
- Un pointeur « pointe » sur *quelque chose*
- Placé devant un nom de variable, le symbole ***** signifie « pointe sur »; il défini une variable de type pointeur :

```
int *ptr;                  // pointeur sur un entier

unsigned char *ch;         // pointeur sur un caractère
                           // non signé

struct Complex *cpx;       // pointeur sur une structure
                           // de type Complex

int **p;                   // pointeur sur un pointeur
                           // sur un entier
```

Pointeurs (2)

- L'adresse d'une variable est obtenue avec l'opérateur **&**
- L'adresse renvoyée forme un pointeur sur le type de la variable

```
int i = 8;          // Entier
int *p = &i;        // p pointe sur i
double *d = &i;     // Erreur: pointeur sur un double !
```

Pointeurs (3)

- Un pointeur est déréférencé avec l'opérateur *****
- Renvoi la **valeur** se trouvant à l'adresse pointée par le pointeur
- Utilisé pour lire ou changer la valeur pointée

```
int i = 8;      // Entier
int *p = &i;    // p pointe sur i
int j = *p;     // j vaut maintenant 8
*p = 12;        // signifie ?
```

Pointeurs (3)

- Un pointeur est déréférencé avec l'opérateur *****
- Renvoi la **valeur** se trouvant à l'adresse pointée par le pointeur
- Utilisé pour lire ou changer la valeur pointée

```
int i = 8;      // Entier
int *p = &i;    // p pointe sur i
int j = *p;     // j vaut maintenant 8
*p = 12;        // i vaut maintenant 12!
```

Pointeurs (4)

- On peut donc écrire :

```
int x;
int *p = &x;
```

- Le contenu de `x` peut alors être exprimé de deux manières différentes :

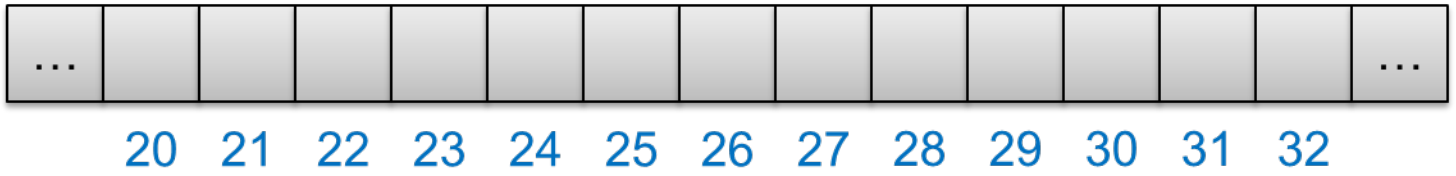
```
x = 7;
*p = 7;
```

  Où `*p` signifie la valeur pointée par `p`, qui n'est rien d'autre que la valeur de `x`.

- A noter que utilisés en cascade, les opérateurs * et & s'annulent:

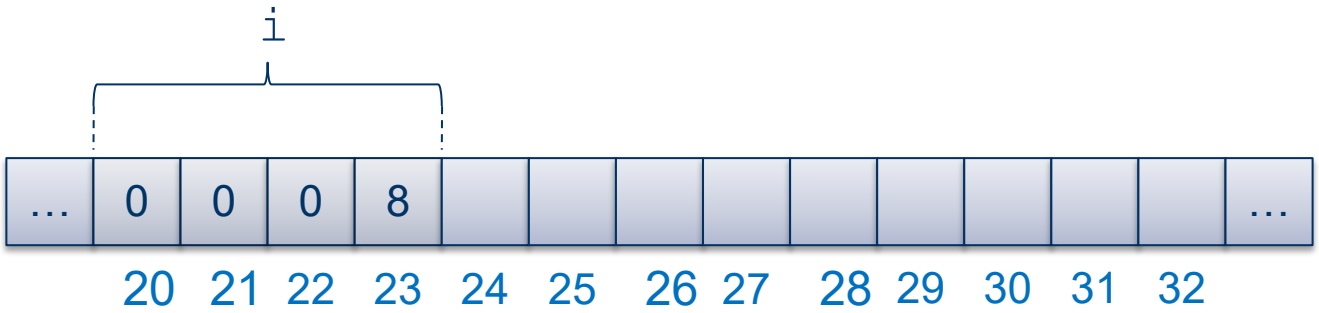
```
*&a = a
```

Pointeurs: exemple



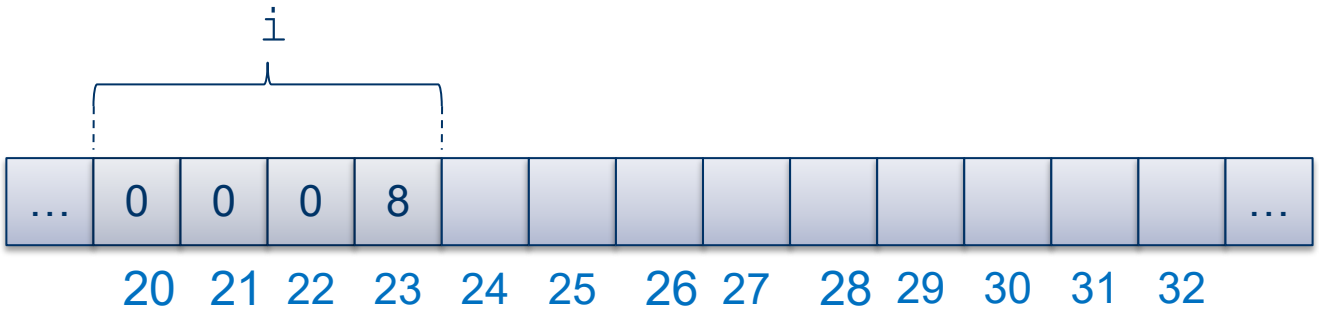

```
int i = 8
```

Pointeurs: exemple




```
int i = 8
&i = 20
```

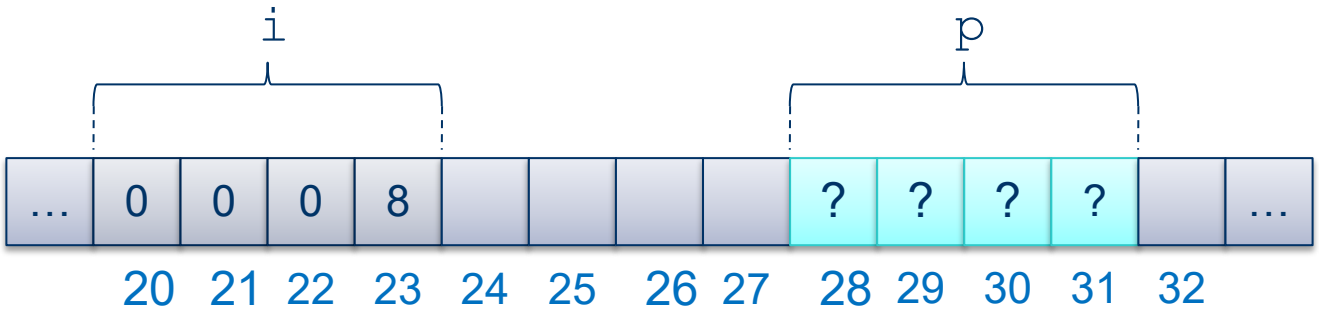
Pointeurs: exemple




```
int i = 8
&i = 20
```

```
int *p
```

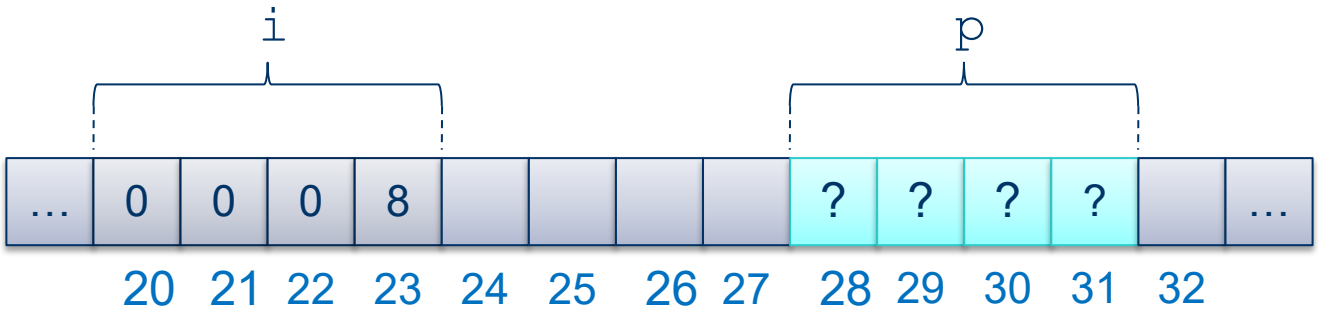
Pointeurs: exemple




```
int i = 8
&i = 20
```

```
int *p
&p = 28
```

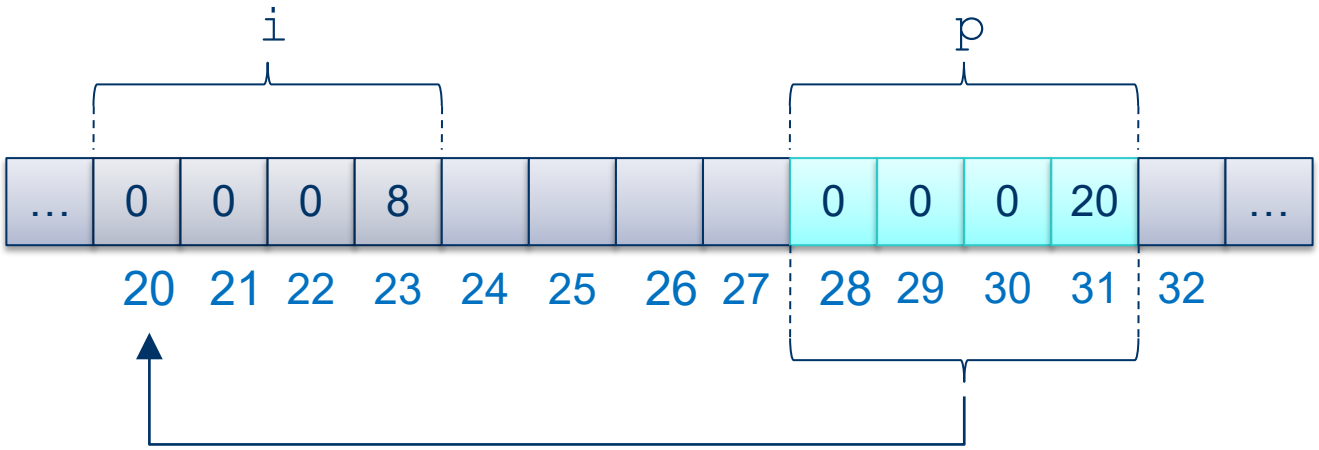
Pointeurs: exemple




```
int i = 8
&i = 20
```

```
int *p = &i
&p = 28
```

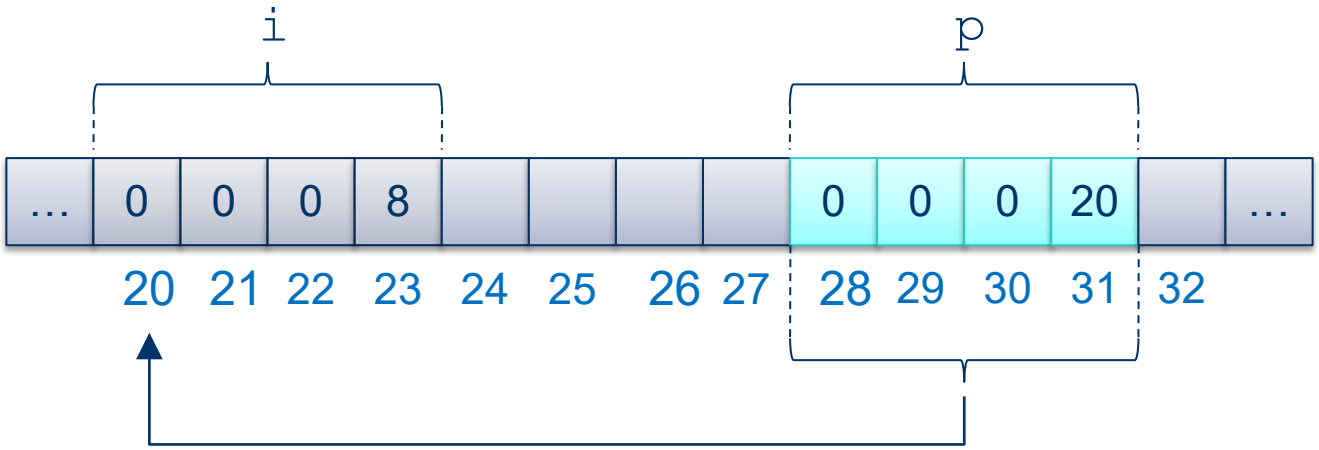
Pointeurs: exemple




```
int i = 8
&i = 20
```

```
int *p = &i (= 20)
&p = 28
```

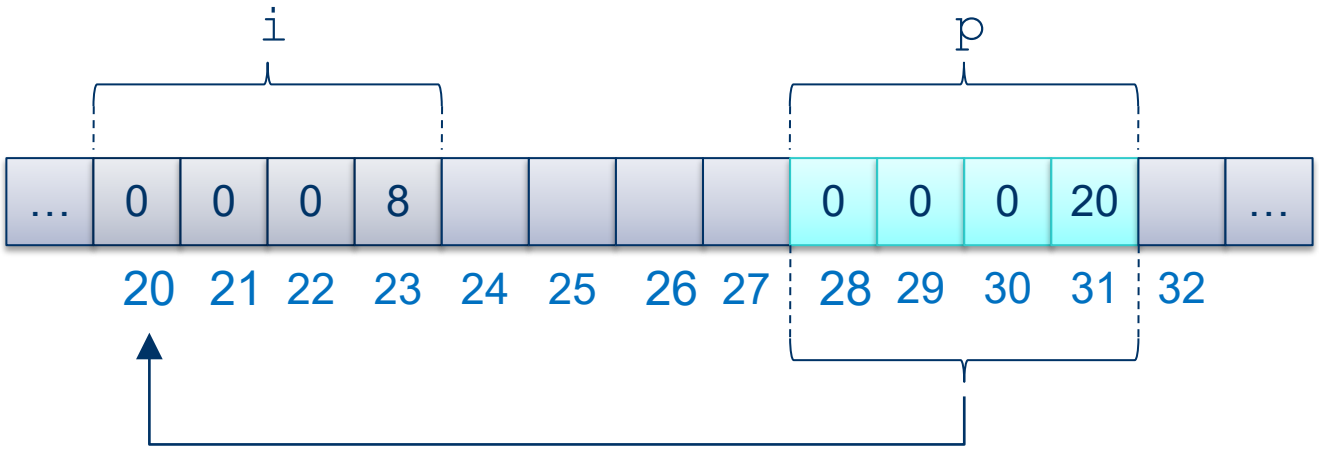
Pointeurs: exemple




```
int i = 8
&i = 20
```

```
int *p = &i (= 20)
&p = 28
int j = *p
```

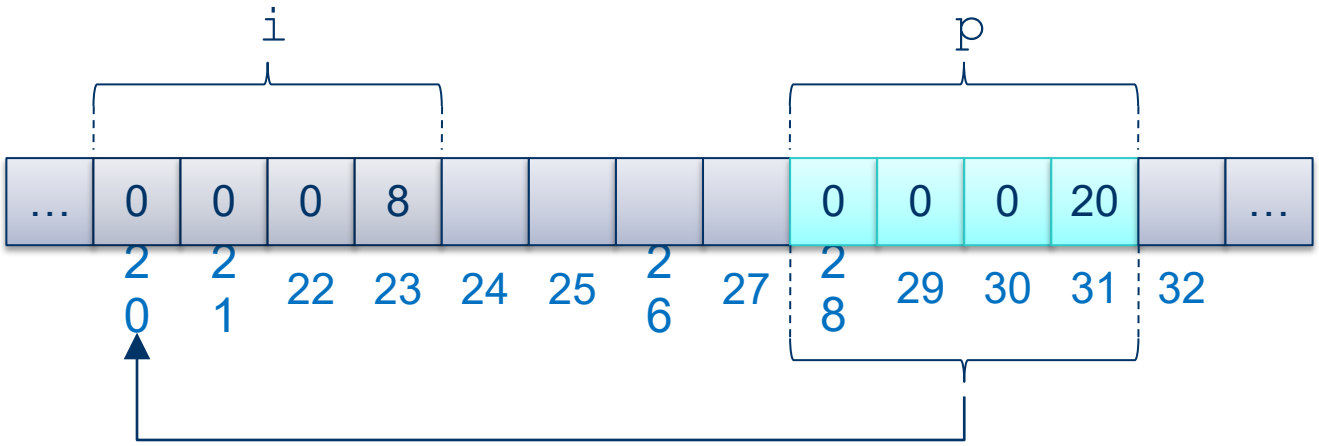
Pointeurs: exemple




```
int i = 8
&i = 20
```

```
int *p = &i (= 20)
&p = 28
int j = *p (= 8)
```

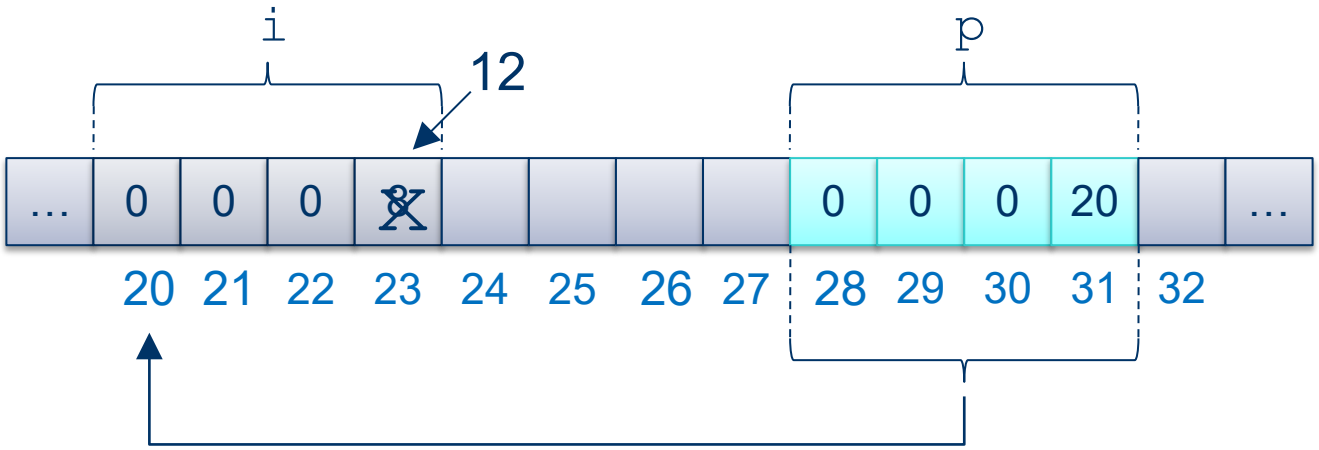
Pointeurs: exemple




```
int i = 8
&i = 20
```

```
int *p = &i (= 20)
&p = 28
int j = *p (= 8)
*p = 12
```

Pointeurs: exemple




```
int i = 8
&i = 20
```

```
int *p = &i (= 20)
&p = 28
int j = *p (= 8)
*p = 12
```

Pointeurs et arguments (1)

- Quels sont les types de passages de paramètres offerts par C ?

Pointeurs et arguments (1)

- Quels sont les types de passages de paramètres offerts par C ?
- C supporte **uniquement** le passage de paramètre par **valeur !**
- Passer l'adresse d'une structure en argument permet d'éviter de recopier toute la structure sur la pile :

```
// Lent (copie complète de la structure)
void slow(struct LargeStructure p) {
        p.x = 10;
        p.y = 20;
}

// Rapide, copie de l'adresse uniquement
void fast(struct LargeStructure *p) {
        p->x = 10;
        p->y = 20;
}
```

Pointeurs et arguments (2)

- Les pointeurs sont nécessaires lorsqu'on désire modifier la valeur des arguments:

```
// Permute les valeurs de a et b
void swap(int *a, int *b) {
        int tmp = *a;
        *a = *b;
        *b = tmp;
}

int main(void) {
        int x = 10, y = 7;
        swap(&x, &y);
}
```

- Sans le passage des adresses en argument, la fonction `swap` n'aurait aucun moyen de modifier les valeurs des arguments!

Pointeurs et arguments (3)

L'utilité des pointeurs dans le cas d'arguments est aussi de permettre à des fonctions d'accéder aux données elles-mêmes et non à des copies.

Dans l'exemple précédent de la fonction permutant la valeur de deux entiers :

```
// Permute les valeurs de a et b
void swap(int *a, int *b) {
        int tmp = *a;
        *a = *b;
        *b = tmp;
}
```

```
int main(void) {
        int x = 10, y = 7;
        swap(&x, &y);
}
```

On passe donc l'adresse des variables `x` et `y` comme paramètres puisque la fonction attend des pointeurs comme paramètres. Il s'agit donc bien d'un passage par valeur, car la valeur du pointeur qui n'est rien d'autre que l'adresse de la variable est simplement copiée sur la pile.