

Introduction à la concurrence

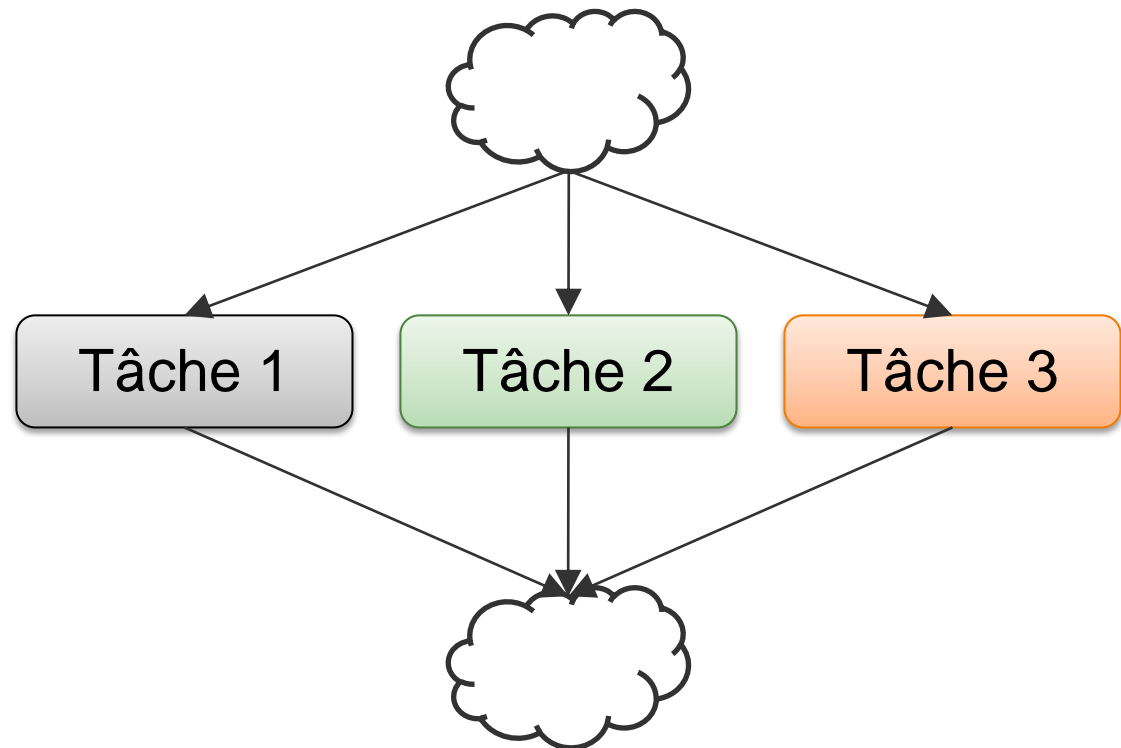
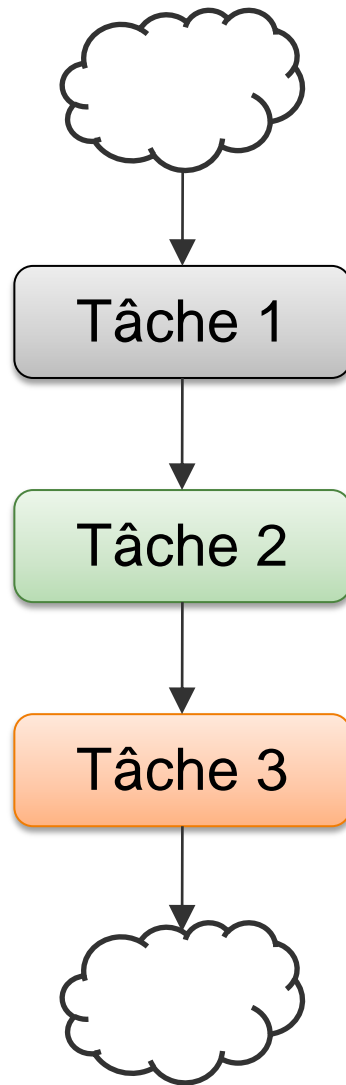
F. Gluck, *V. Pilloux*

Version 0.8

Programmation concurrente ?

- La programmation concurrente est un **paradigme de programmation tenant compte, dans un programme, de plusieurs contextes d'exécution** (tâches, *threads*, processus) matérialisés par une pile d'exécution (*stack*) et des données privées.
- La concurrence est devenue indispensable pour:
 - pouvoir lancer plusieurs tâches qui ont été **spécifiées indépendamment**, mais qui tournent sur une même station, **possédant un nombre de CPU limité**
 - écrire des programmes interagissant avec du matériel
 - tirer parti de multiples processeurs (multi-coeurs, *GPU*, *cloud*, ...).

Programmation séquentielle vs concurrente



Avantages de la programmation concurrente ?

- Simplifier la structure d'un programme, laissant à l'OS la tâche de répartir la puissance de calcul entre les modules
- Tirer avantage des architectures multi-cœurs :
 - Aujourd'hui n'importe quel PC possède *au moins* 2 cœurs.
 - Permet de **maximiser** l'utilisation des processeurs et des ressources, principalement dans les 2 cas suivants:
 - Calculs indépendants
 - Attente sur une ressource matérielle (timer, clavier, GPIO, etc...)
- Permet de faciliter la programmation en temps réel (suivant le type d'ordonnancement de l'OS)

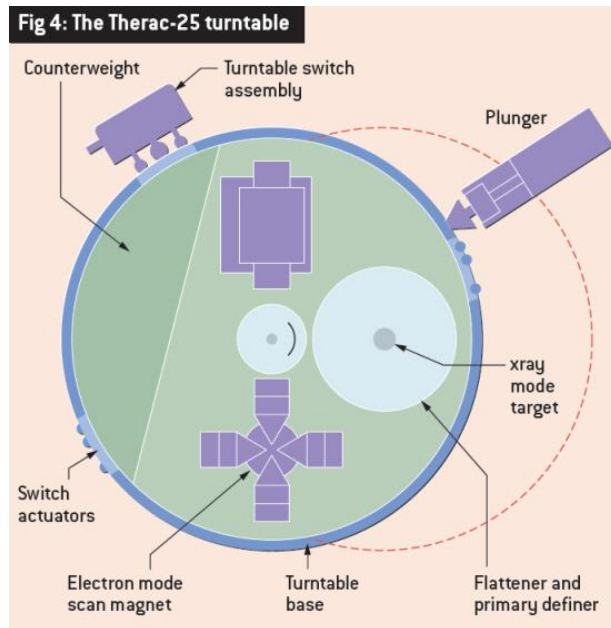
Problèmes de la programmation concurrente ?

- Accès aux ressources partagées
 - Echange de données
 - Séquentialité de l'exécution (d'un CPU)
 - Difficulté à synchroniser des tâches
 - Problème de prédictibilité
 - Problème de reproductibilité
- **Plus difficile à déboguer!**



Accident : 1985-1987

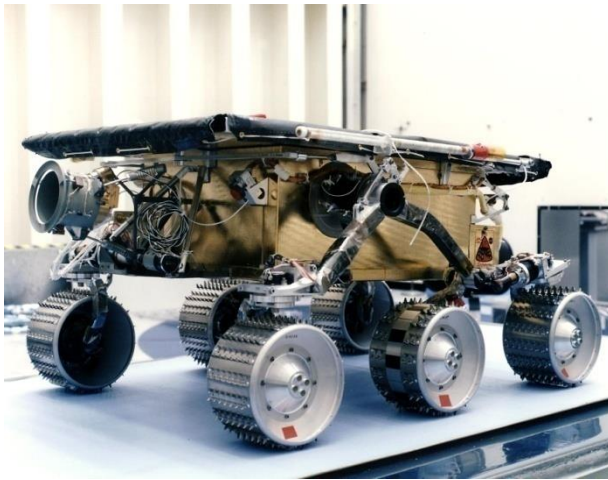
Thérapie par radiation : Therac-25, 6 accidents et 3 morts entre 85 et 87*. Logiciel basé sur un OS spécifique, tests insuffisants.



Leveson N. & Turner C. "An investigation of the Therac-25 accidents", DOI 10.1109/MC.1993.274940

Accident : 1997

- Mars Pathfinder : 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



- “What really happened on Mars?”

http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Mars_Pathfinder.html

Accident : 2003

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



Problème de séquence logicielle (*race condition*)

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

Accident : 2000-2010

Carnegie Mellon

May 25,
2010

Toyota "Unintended Acceleration" Has Killed 89



A 2005 Toyota Prius, which was in an accident, is seen at a police station in Harrison, New York, Wednesday, March 10, 2010. The driver of the Toyota Prius told police that the car accelerated on its own, then lurched down a driveway, across a road and into a stone wall. (AP Photo/Seth Wenig) / **AP PHOTO/SETH WENIG**

Unintended acceleration in Toyota vehicles may have been involved in the deaths of 89 people over the past decade, upgrading the number of deaths possibly linked to the massive recalls, the government said Tuesday.

The National Highway Traffic Safety Administration said that from 2000 to mid-May, it had received more than 6,200 complaints involving sudden acceleration in Toyota vehicles. The reports include 89 deaths and 57 injuries over the same period. Previously, 52 deaths had been suspected of being connected to the problem. <http://www.cbsnews.com/news/toyota-unintended-acceleration-has-killed-89/>

© Copyright 2014, Philip Koopman. CC Attribution 4.0 International license.



5

h e p i a

Haute école du paysage, d'ingénierie
et d'architecture de Genève

(task overflow, memory corruption, dead tasks)
https://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf

9

CPU bound vs I/O bound

- En général, un programme effectue des calculs et attend un changement des entrées/sorties (I/O).
- Un programme faisant principalement des calculs est limité par les performances du CPU : celui-ci est dit **CPU bound**.
- Un programme utilisant principalement des entrées/sorties passera son temps à attendre : celui-ci est dit **I/O bound**.
- Idéalement un programme devrait maximiser l'utilisation du CPU en évitant toute attente.
- La programmation concurrente permet, grâce au mécanisme de **threads**, de continuer à travailler même lorsqu'un thread est bloqué (sur une entrée/sortie par exemple).

Temps partagé

- En exécutant un thread, puis en le stoppant, en en exécutant un autre, etc. → l'OS donne l'illusion que plusieurs CPU existent.
- Le partage dans le temps du CPU (*time sharing*) permet de lancer autant de threads que souhaité.
- Coût en performance.
- L'implémentation nécessite :
 - Mécanismes bas niveau, ex : changement de contexte
 - Mécanismes haut niveau "intelligent", ex : ordonnancement

Anatomie d'un thread

Un thread est une séquence d'instructions s'exécutant séquentiellement, parfois appelé un « fil d'exécution »

- Les threads d'un même processus se partagent l'espace d'adressage d'un seul processus
- Un thread correspond à une tâche qui s'exécute indépendamment des autres
- **L'ordonnanceur** est responsable de l'ordre d'exécution des threads
- Chaque thread possède :
 - Sa propre pile (*stack*)
 - Son propre contexte d'exécution (IP + SP + registres)

Thread vs. processus

- Un processus est composé d'un ou plusieurs threads
- Un thread est aussi appelé **processus léger**
- Un processus est aussi appelé **processus lourd**
- Le changement de contexte d'un processus à un autre est plus coûteux en temps machine qu'un changement de contexte entre threads (d'où les qualificatifs « lourd » et « léger »)

Relation thread-processus

Ce que les threads et processus ont en commun :

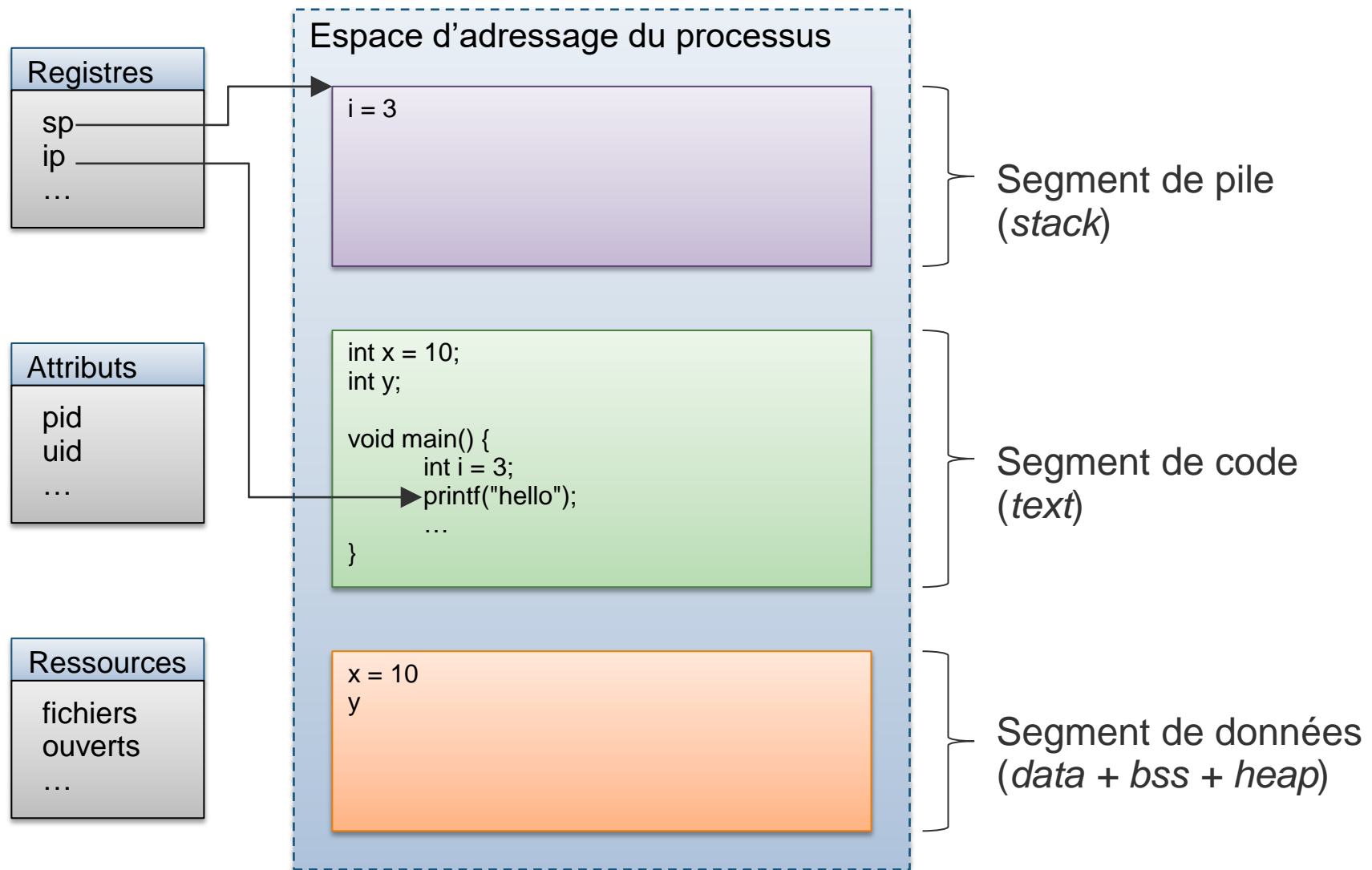
- un ID, un ensemble de registres, un état, une priorité
- un bloc d'information
- partagent des ressources avec les processus parents (descripteurs de fichiers, etc.)
- entités indépendantes une fois créées
- peuvent changer leurs attributs après création et créer de nouvelles ressources

Relation thread-processus

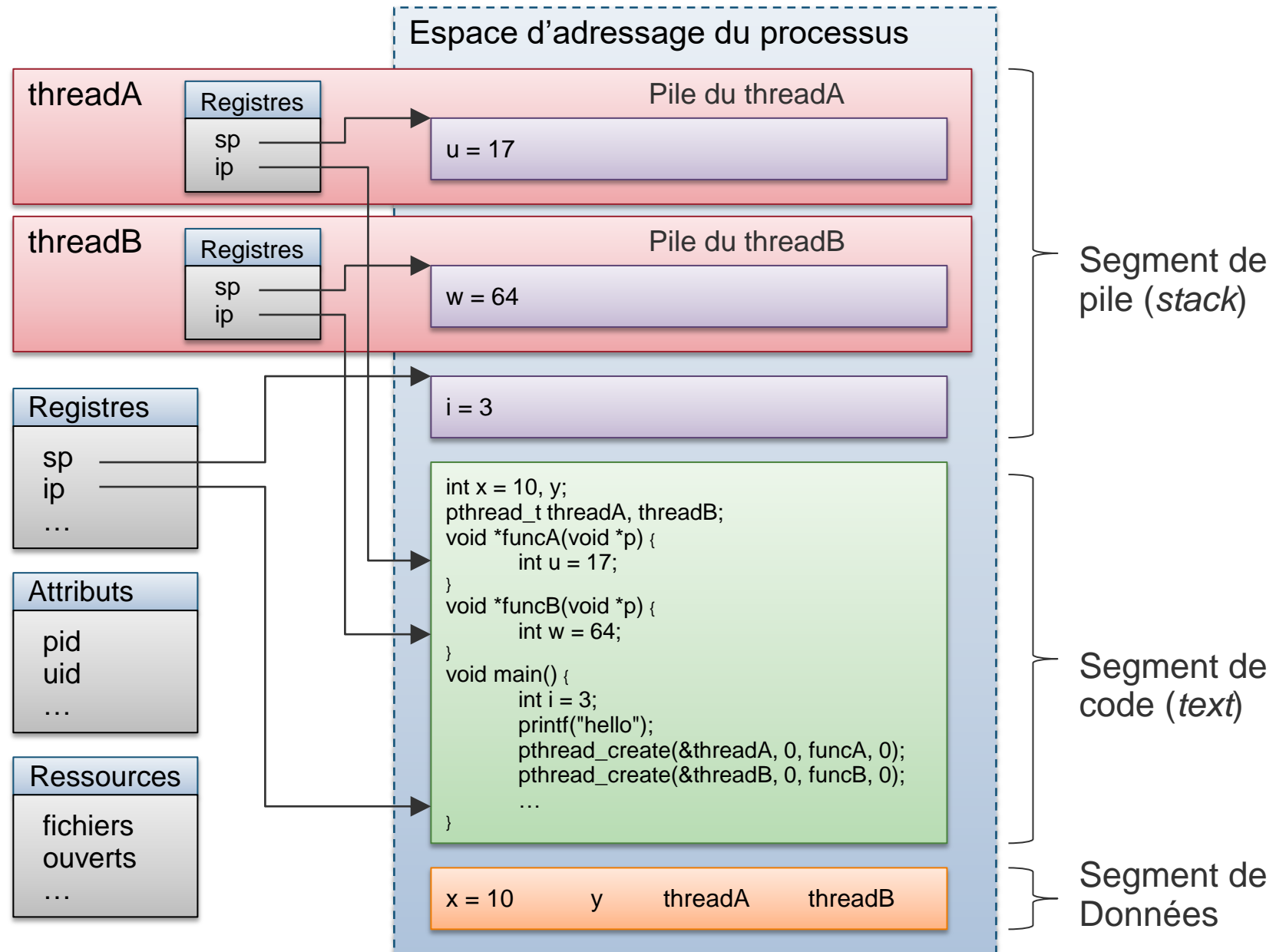
Ce que les threads et processus n'ont **pas** en commun :

- les processus possèdent chacun un espace d'adressage indépendant au contraire des threads
- communication entre processus se fait à l'aide de mécanismes *IPC* (Inter Process Communication) tandis que les threads d'un même processus utilisent **des variables partagées**
- Les processus enfants n'ont aucun contrôle sur les autres processus **enfants ou parents** ; les threads d'un processus sont considérés comme des pairs et **peuvent exercer un contrôle sur les autres threads**

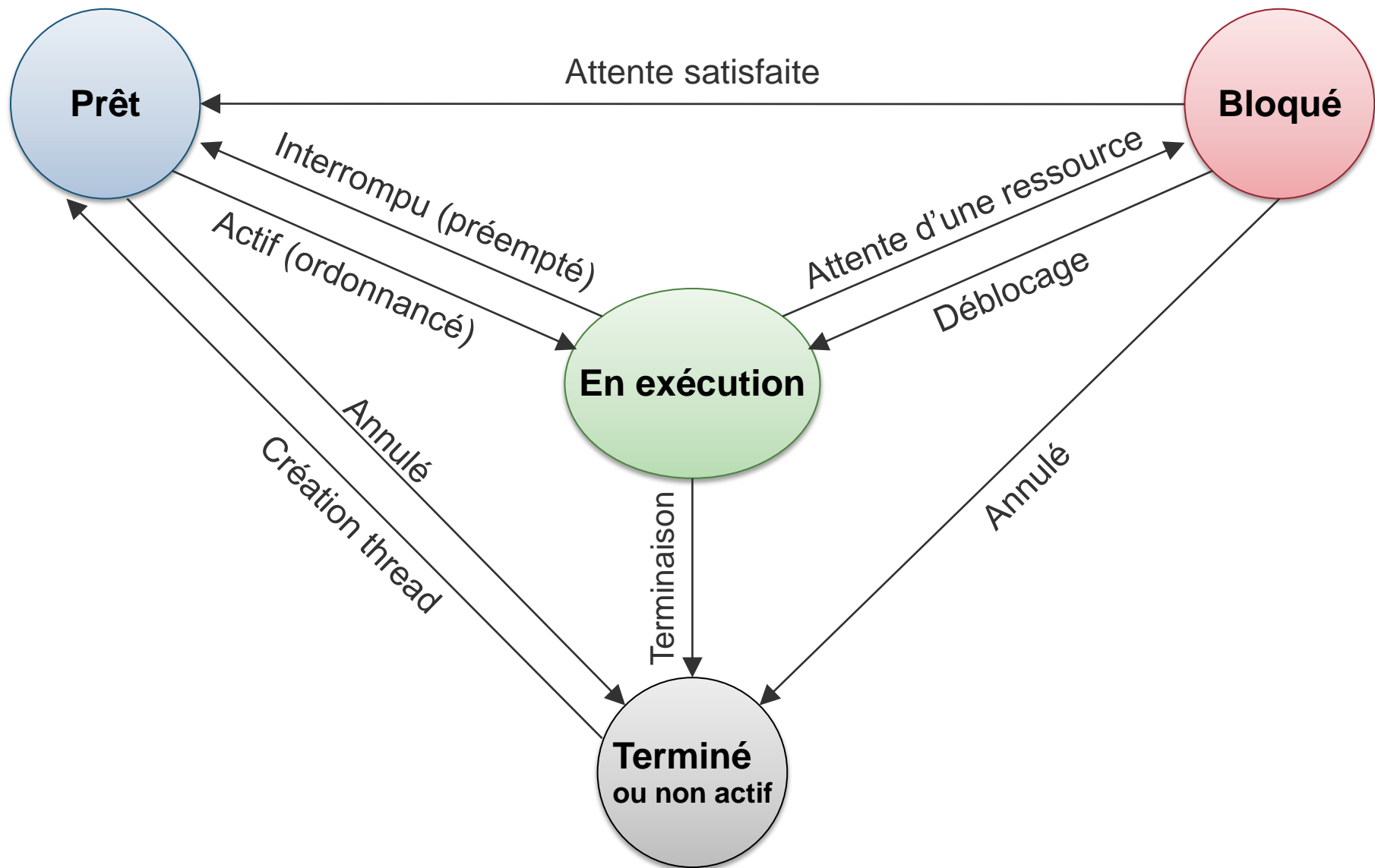
Espace d'adressage d'un processus



Espace d'adressage d'un processus multi-threadé



Etats d'un processus ou d'un thread



Etats d'un processus ou d'un thread

Prêt <i>(ready)</i>	Le processus est prêt à être exécuté. Cas d'un processus nouvellement créé, débloqué ou, d'un ou plusieurs processus occupant le ou les CPU disponibles.
En exécution <i>(running)</i>	Le processus est en cours d'exécution sur un CPU. Plusieurs processus peuvent être en exécution dans le cas d'une machine multiprocesseur.
Bloqué <i>(blocked)</i>	Le processus est en attente sur une synchronisation ou sur la fin d'une opération d'entrée/sortie par exemple.
Terminé <i>(terminated)</i>	Le processus a terminé son exécution ou a été annulé (<i>cancelled</i>). Les ressources du processus seront libérées et le processus disparaîtra.

Exemples d'applications multi-threadées

- Serveur web ou ftp : chaque client est géré par un thread.
- Browser : un thread gère une connexion, un thread fait le rendu de la page, un thread décode une vidéo, un thread gère l'interaction avec l'interface.
- Jeu vidéo : un thread s'occupe du rendu graphique, un autre de l'IA, un autre du son, un autre les joypads, etc.
- Visualisateur d'images : un thread décode et affiche l'image courante alors que n threads chargent les n images suivantes.
- Filtres dans Photoshop : image divisée en blocs où chaque thread applique le filtre à son bloc d'image.
- Systèmes embarqués.
- etc.

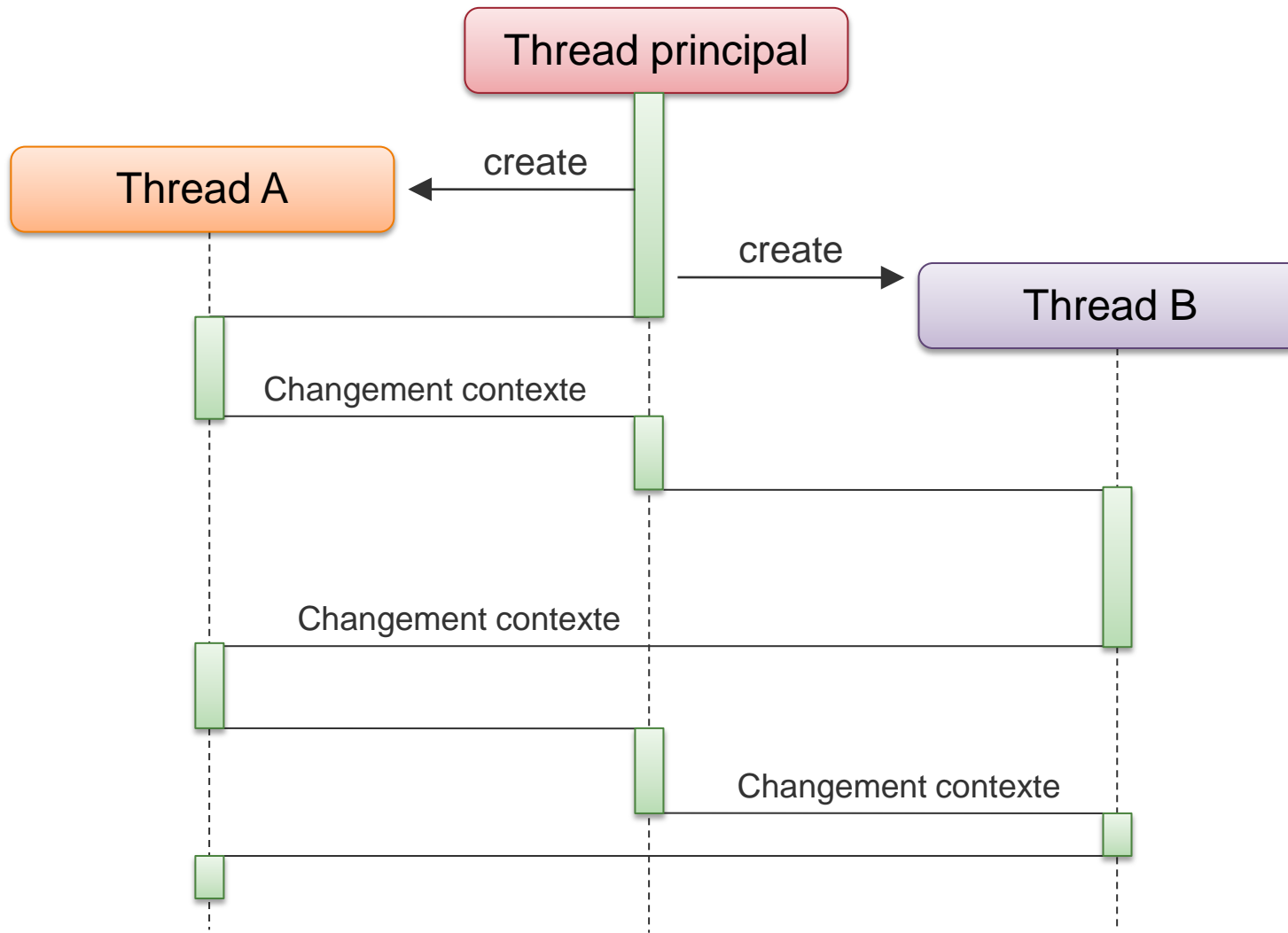
Parallélisme vs concurrence

- Concurrence \neq parallélisme !
- Deux tâches T1 et T2 sont concurrentes si leur ordre d'exécution dans le temps est indéterminé :
 - T1 peut s'exécuter et se terminer avant T2.
 - T2 peut s'exécuter et se terminer avant T1.
 - T1 et T2 peuvent s'exécuter *en même temps* \rightarrow parallélisme.
 - T1 et T2 peuvent s'exécuter en alternance.
- Un système mono-processeur ne peut pas offrir de parallélisme.
- Le parallélisme est uniquement possible sur des systèmes multi-coeurs, multi-processeurs et systèmes distribués.
- Un système **mono-processeur**, peut être **concurrent** mais pas **parallèle**.
- La concurrence est une propriété d'un programme et est un concept plus général que le parallélisme.

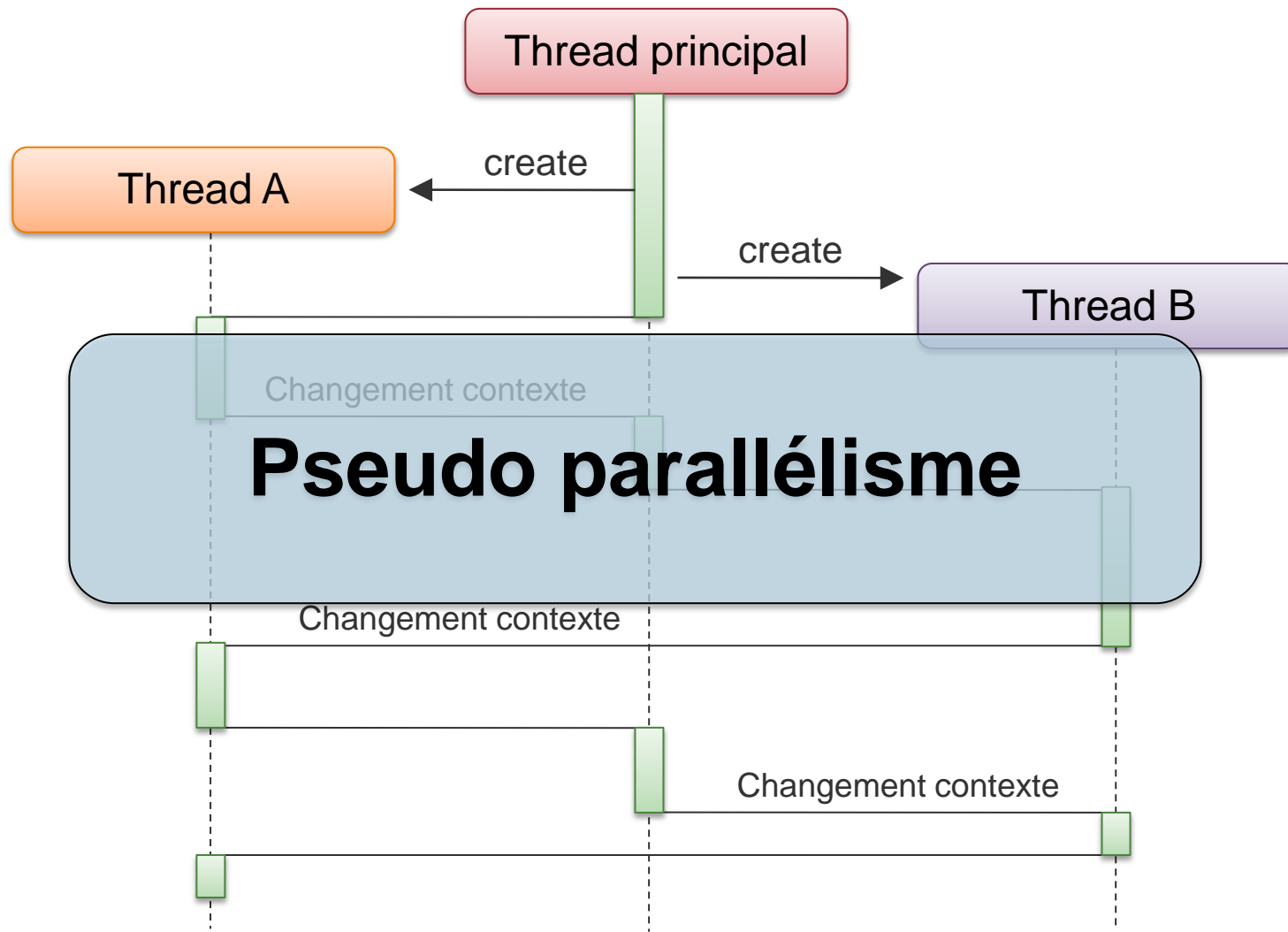
Parallélisme et pseudo-parallélisme

- Soit un programme où plusieurs threads s'exécutent *en même temps* :
 - Sur une architecture mono-processeur, les threads s'exécutent chacun à leur tour **par entrelacement** et sont ordonnancés (*scheduled*) par le noyau (*kernel*).
→ **Pseudo parallélisme**
 - Sur une architecture multi-processeur (ou multi-cœur), les threads (ou une partie d'entre eux) peuvent s'exécuter réellement en parallèle.
→ **Vrai parallélisme**
- Si une machine possède plus que 1 processeur, alors du vrai parallélisme est possible : N processus peuvent alors s'exécuter **simultanément** sur N processeurs.

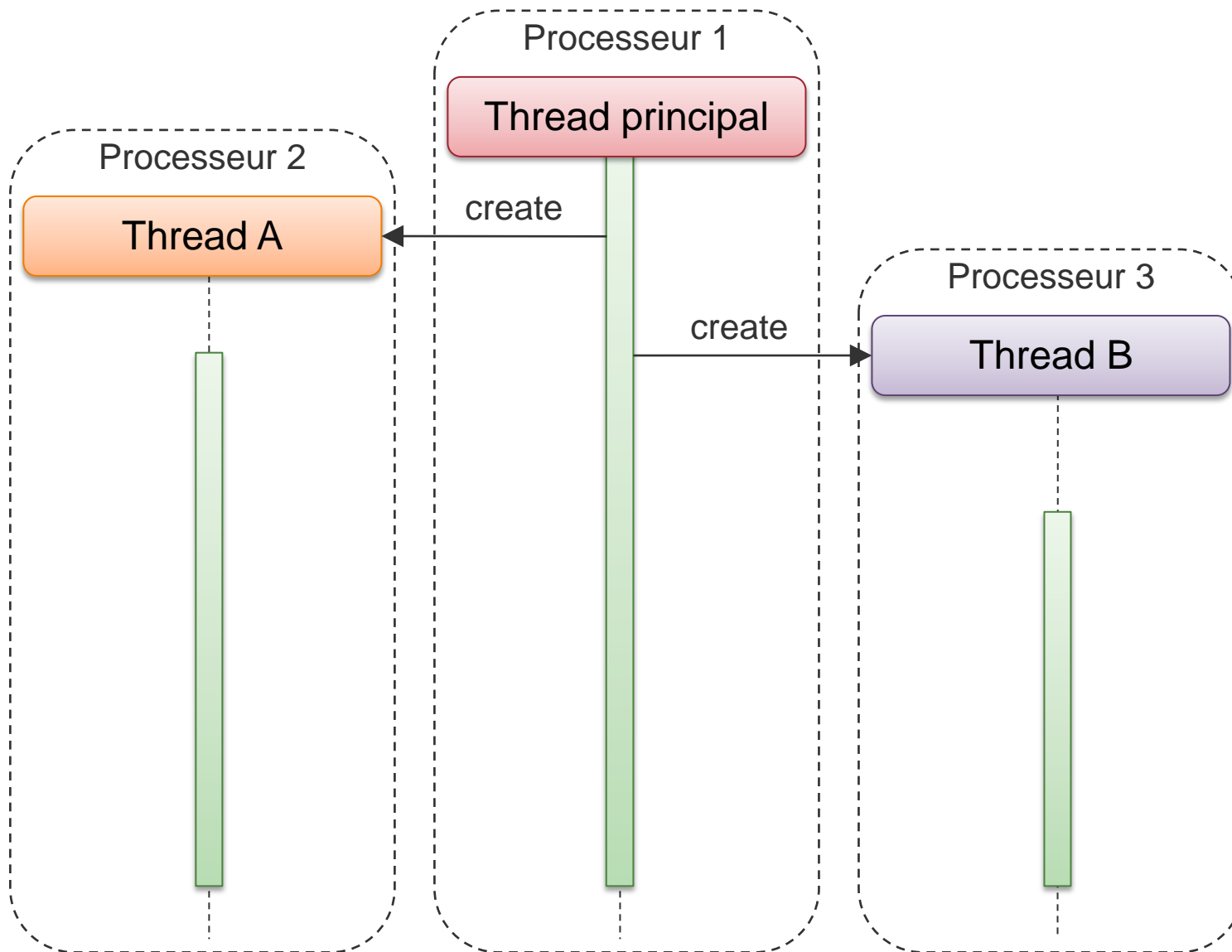
Flot d'exécution d'un processus multi-threadé sur un système **monoprocasseur**



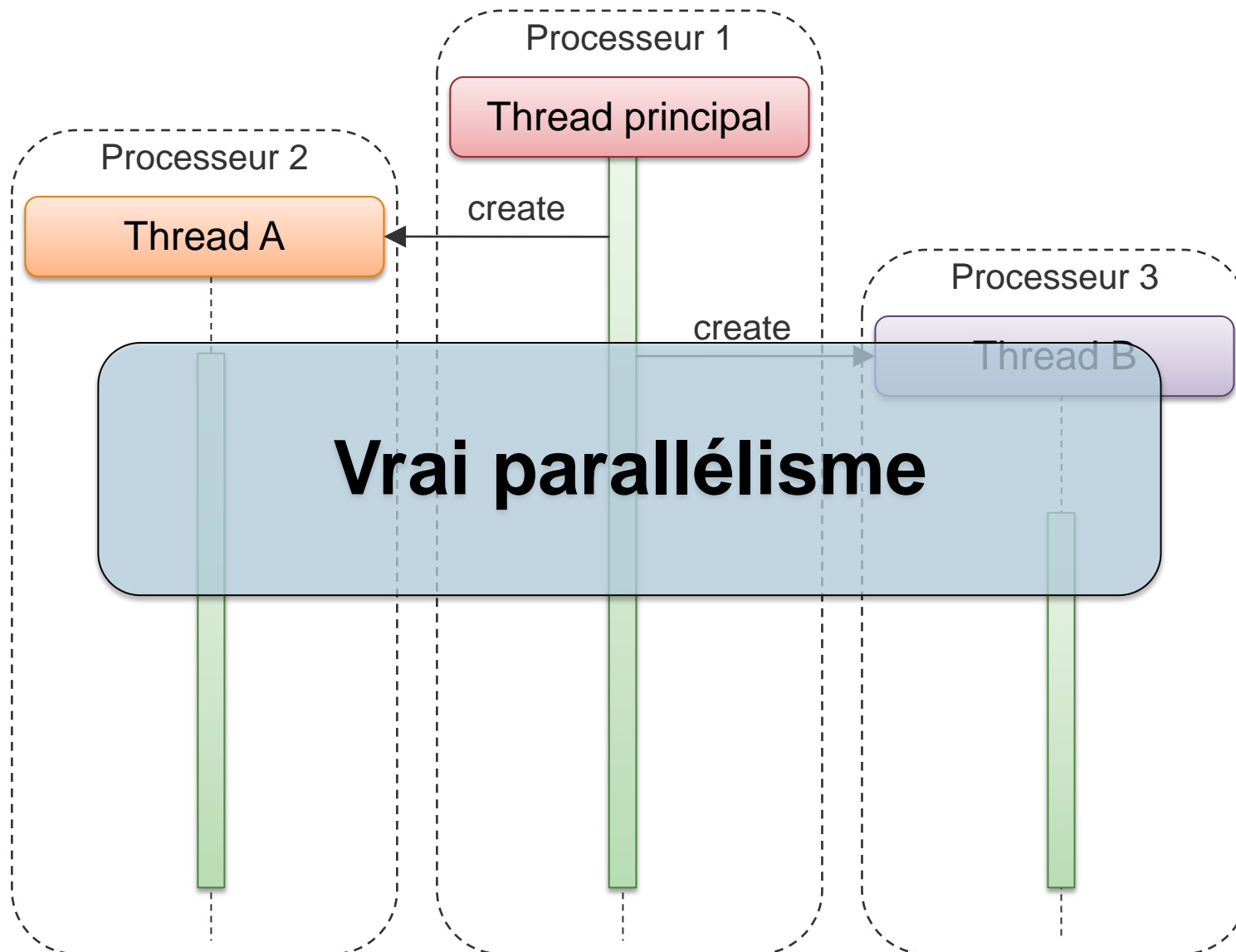
Flot d'exécution d'un processus multi-threadé sur un système **monoprocasseur**



Flot d'exécution d'un processus multi-threadé sur un système **multi-processeurs**



Flot d'exécution d'un processus multi-threadé sur un système **multi-processeurs**

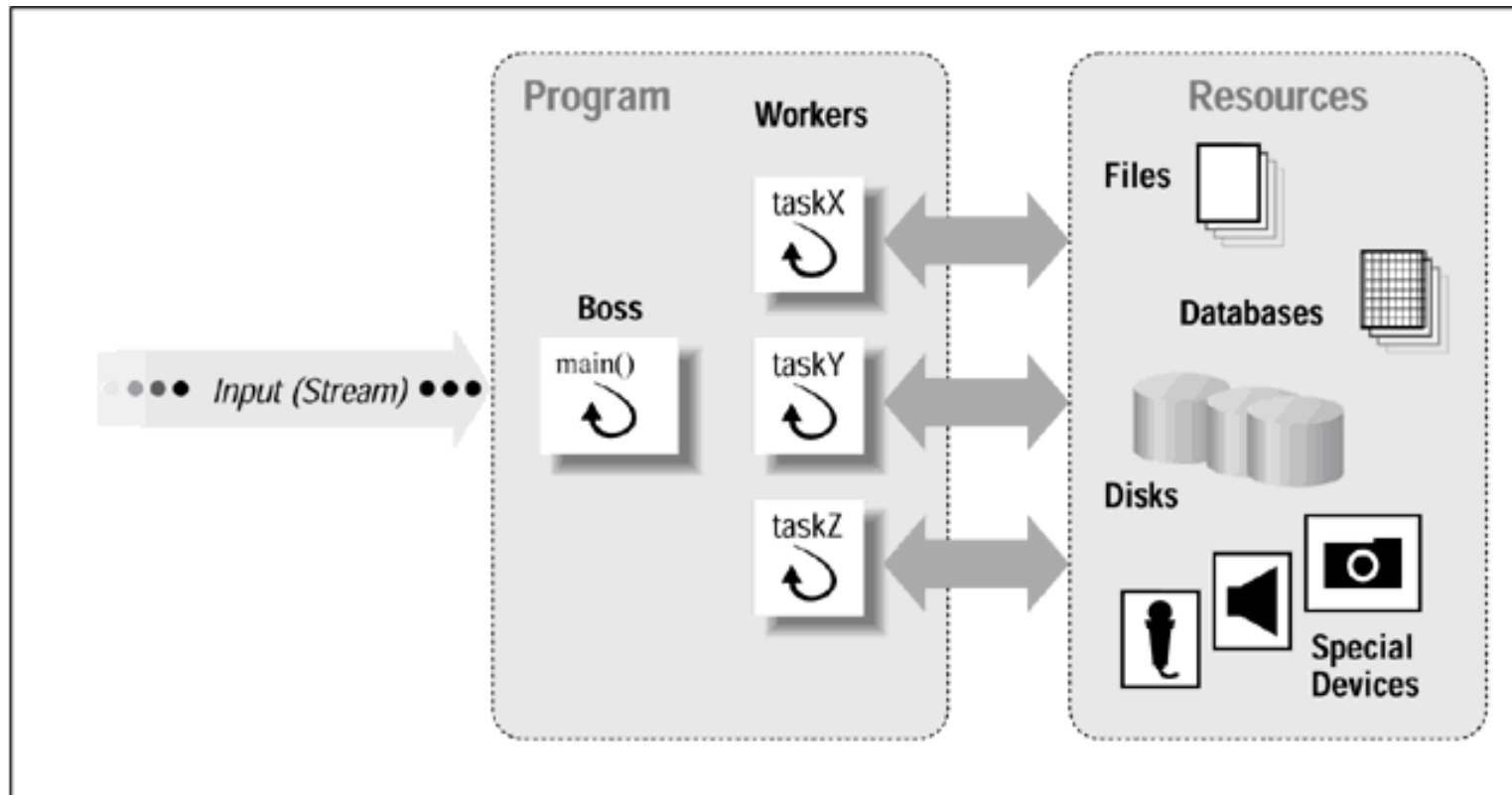


Organisation en threads

- Pour qu'un programme puisse bénéficier d'une implémentation multi-threadée, celui-ci doit être organisé en tâches indépendantes pouvant s'exécuter de manière concurrente.
- Par ex: si deux fonctions f1 et f2 sont indépendantes, alors leur exécution peut se chevaucher dans le temps: celles-ci peuvent être implémentées par des threads.
- Comment décomposer un programme en plusieurs threads?
Plusieurs modèles possibles :
 - Modèle maître/esclave (master/slave ou boss/worker)
 - Modèle pair (*peer model*)
 - Modèle pipeline (*pipeline model*)

Modèle maître/esclaves

- Maître : gère les données d'entrée, crée esclaves et assigne les tâches aux esclaves.
- Chaque esclave : se synchronise avec le maître → gestion des données entrantes et sortantes.



*source: Nichols, Buttlar, Farrell, *Pthreads Programming*, O'Reilly

Modèle maître/esclaves, variante 1

- Le maître crée les esclaves dynamiquement à chaque requête.
 - Autant d'esclaves concurrents qu'il y a de requêtes.
 - Les esclaves se terminent une fois leur tâche effectuée.

```
main() { // The boss
    forever {
        get a request
        switch request
            case X : create_thread(thread1,taskX)
            case Y : create_thread(thread2,taskY)
            ...
    }
}

thread1(taskX) { // Worker processing requests of type X
    perform the task, synchronize if access to shared resources
}

thread2(taskY) { // Worker processing requests of type Y
    perform the task, synchronize if access to shared resources
}
...
```


Modèle maître/esclaves, variante 2

- Le maître crée un *pool* d'esclaves à l'initialisation qui reste actif.
- Chaque esclave attend d'être réveillé par le maître pour réaliser sa tâche.
- Le maître insère les requêtes à effectuer dans une queue.
- Les esclaves récupèrent les tâches à effectuer depuis la queue.

```
main() { // The boss
    for the number of workers
        create_thread(pool_base)
        forever {
            get a request
            place request in work queue
            signal sleeping threads that work is available
        }
}

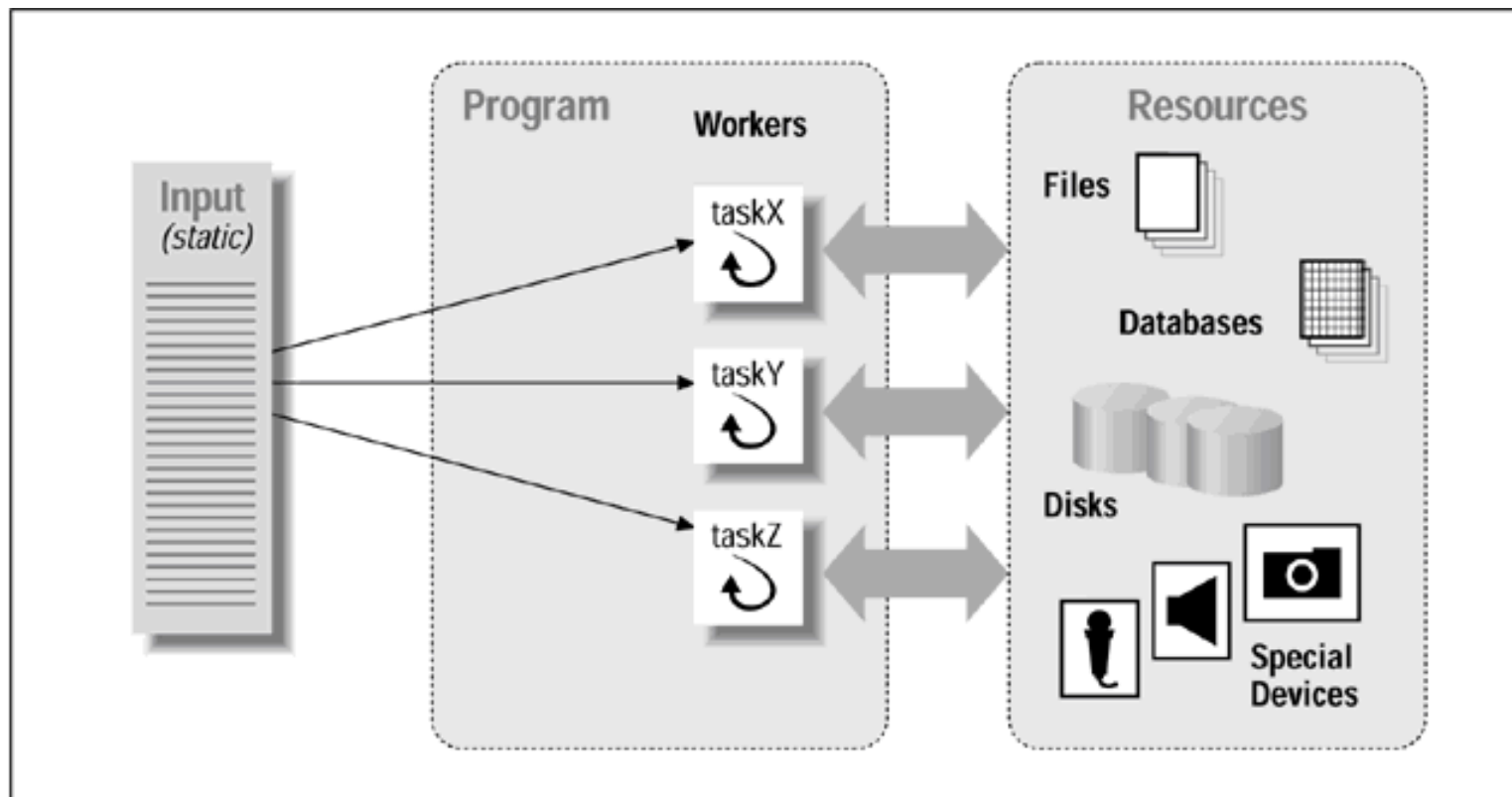
pool_base() { // All workers
    forever {
        sleep until awoken by boss
        dequeue a work request
        switch
            case request X : taskX()
            case request Y : taskY()
            ...
    }
}
```

Modèle maître/esclaves

- Modèle adapté aux serveurs (bases de données, window manager, etc.).
- Complexité des requêtes asynchrones et communication gérée par le maître.
- Le traitement des requêtes et données est **délégué** aux esclaves.
- Important de minimiser la fréquence des communications maître ↔ esclaves.
- Le maître n'est pas bloqué par les esclaves
- Le maître doit éviter au maximum les dépendances entre les esclaves.

Modèle pair

- Pas de thread principal.
- Hiérarchiquement tous égaux.
- Chacun s'arrange avec ses données entrantes/sortantes.



*source: Nichols, Buttlar, Farrell, *Pthreads Programming*, O'Reilly

Modèle pair : pseudo-code

```
main() {  
    thread_create(thread1)  
    thread_create(thread2)  
    ...  
    signal all workers to start  
    wait for all workers to finish  
    do any clean up  
}  
  
thread1() {  
    wait for start  
    perform task, synchronize if access to shared resources  
}  
  
thread2(taskY) {  
    wait for start  
    perform task, synchronize if access to shared resources  
}  
...
```

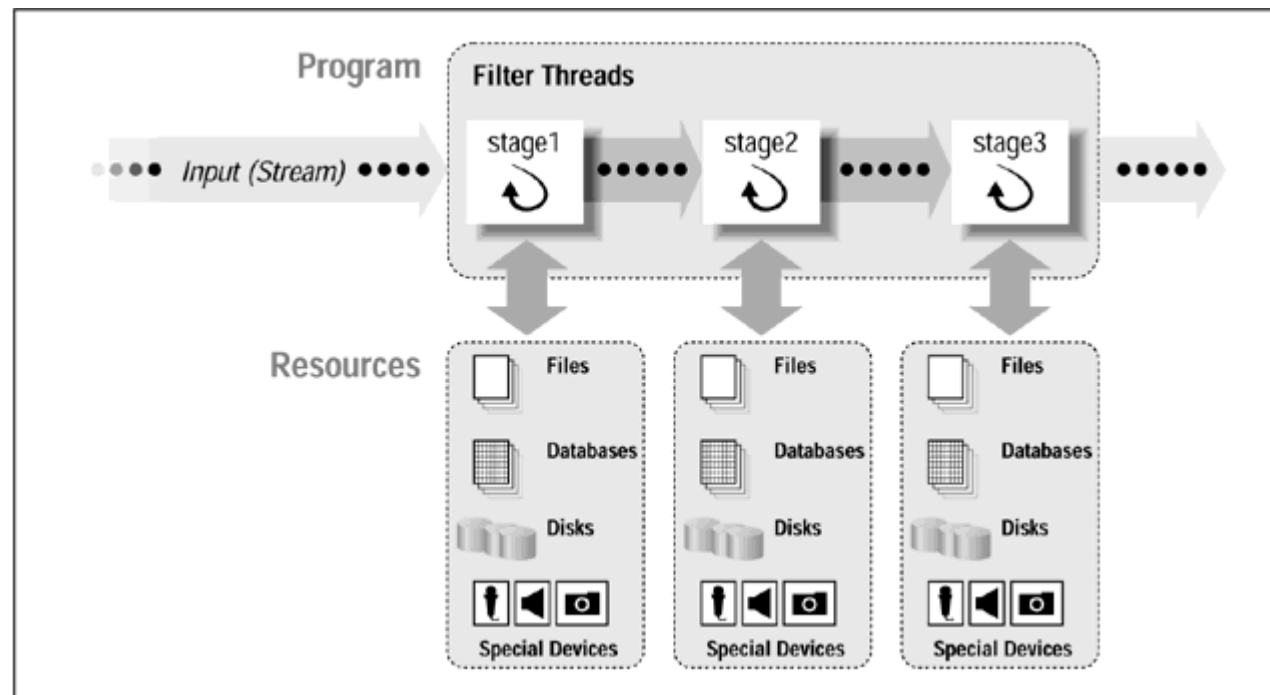
Modèle pair

- Adapté aux applications ayant un ensemble d'entrées fixes ou bien définies et faciles à partager: multiplication matricielle, analyse d'image, etc.
- Entrées fixes permet de faire sans un maître (modèle maître/esclaves sans maître).
- Sans maître, les esclaves doivent se synchroniser pour accéder aux données d'entrée.
- Similairement au modèle maître/esclaves: éviter de nombreuses dépendances entre esclaves (accès ressources partagées).
- Note: les GPU sont particulièrement adaptées à ce modèle

Modèle pipeline

Modèle applicable lorsque :

- L'application traite une longue chaîne d'entrée.
- Le traitement à effectuer sur ces entrées peut être décomposé en sous-threads au travers desquelles chaque donnée d'entrée doit passer.
- Chaque étage peut traiter une donnée différente à chaque instant.
- Un thread attend les données du précédent et les transmet ensuite au suivant.



*source: Nichols, Buttlar, Farrell, *Pthreads Programming*, O'Reilly

Modèle pipeline : pseudo-code

```
main() {
    thread_create(stage1)
    thread_create(stage2)
    ...
    wait for all pipeline threads to finish
    do any clean up
}

stage1() {
    forever {
        get input for the program
        do stage1 processing of the input
        pass result to stage2 in pipeline
    }
}

stage2() {
    forever {
        get input from stage1 in pipeline
        do stage2 processing of the input
        pass result to stage3 in pipeline
    }
}

stageN() {
    forever {
        get input from stageN-1 in pipeline
        do stageN processing of the input
        pass result to program output
    }
}
```


Modèle pipeline

- Modèle inspiré du pipeline d'un processeur :
 - Fetch, decode, execute, write-back
- Exemple classique :
 - Chaîne de montage d'une fabrique de voitures.
- Amélioration du débit (*throughput*) : chaque étage est calculé de manière concurrente.
- **Le débit global est limité par l'étage le plus lent.**
- Performances : important de diviser le travail équitablement entre les différents étages (temps de calcul similaire).

Comment implémenter la concurrence ?

- En travaillant sur un OS
- Grâce à des mécanismes du langage de programmation (ex: ADA, Rust)
- Grâce à une librairie dédiée (construction externe, notre cas)

Grâce au langage de programmation

- **Avantages :**
 - Les notions concurrentes de même que les constructions sont données par le langage.
 - Détection d'une partie des erreurs à la compilation.
 - Méthodologie de programmation imposée par le langage.
- **Désavantages :**
 - Obligation d'utiliser un langage dédié \Rightarrow potentiellement peu répandu.
 - Contraintes liées au langage choisi.
- Exemples : Rust, Java, GO, Ada, Erlang, etc.

Grâce à une librairie

- **Avantages :**
 - Un langage quelconque peut profiter de la librairie
 - Portable sur tout système ayant une implémentation de la librairie
- **Désavantages :**
 - Débogage délicat
 - Pas de méthodologie de programmation imposée, donc plus permissif
- Exemples : POSIX Threads (pthreads), Qt, etc.

Ressources

- **Operating systems concepts** (9^{ème} édition), Silberschatz A., Galvin P., Gagne G.
- **Operating Systems: Three Easy Pieces**, Remzi H. et Andrea C. Arpaci-Dusseau. Arpaci-Dusseau Books.
- **Modern Multithreading**, Richard H. Carver, Kuo-Chung Tai.