Processus et fil d’exécution

INF3173 – Principes des systèmes d'exploitation
Automne 2024

Francis Giraldeau
francis.giraldeau@uqam.ca

Université du Québec à Montréal

Agenda

- Processus
- Cycle de vie : création et terminaison
- Changement d’exécutable
- Espace mémoire
- Appels systèmes et API
- Gestion d’erreur langage C

Résumé

- Création d’un processus : fork()
- Terminer un processus : exit()
- Attendre qu’un processus se termine: wait()
- Changer le programme en cours : exec()

Création d’un processus

- Un processus est un exécutable en mémoire (des instructions machines), avec un état, dont la pile et les registres.
- Chaque processus a un numéro associé (Process ID ou simplement PID).
- Un processus est créé par un processus existant (le parent crée un enfant) par un appel à fork()
- Crée une copie* du programme, puis les deux processus peuvent s’exécuter indépendamment et simultanément
- L’enfant hérite des descripteurs de fichiers ouverts
- *Note : techniquement pas une copie complète systèmatique, mais plutôt une copie paresseuse (Copy-On-Write) pour des raison d’efficacité

Fonction fork()

- Retourne 2 fois : une fois dans le parent, et l'autre dans l'enfant.
- Le reste du programme s'exécute par les deux processus.
- La valeur de retour de fork() identifie le parent et l'enfant.
  - Si 0 alors enfant
  - Si > 0 alors parent (indique le PID de l'enfant)
  - Si < 0 alors erreur

```
printf("AVANT");
fork();
printf("APRES");
```

programme

```
AVANT
APRES
APRES
```

Sortie standard

Terminer un processus

- Appel système exit()
- Code de retour du programme passé en argument
  - Valeur récupérée dans le shell avec $?
- Cette fonction ne retourne pas!
- Le système d’exploitation libère toutes les ressources : mémoire, fichiers ouverts, connexion réseau, etc.
- Appel implicite lorsque main() retourne.
  - Ajouté par la librairie C

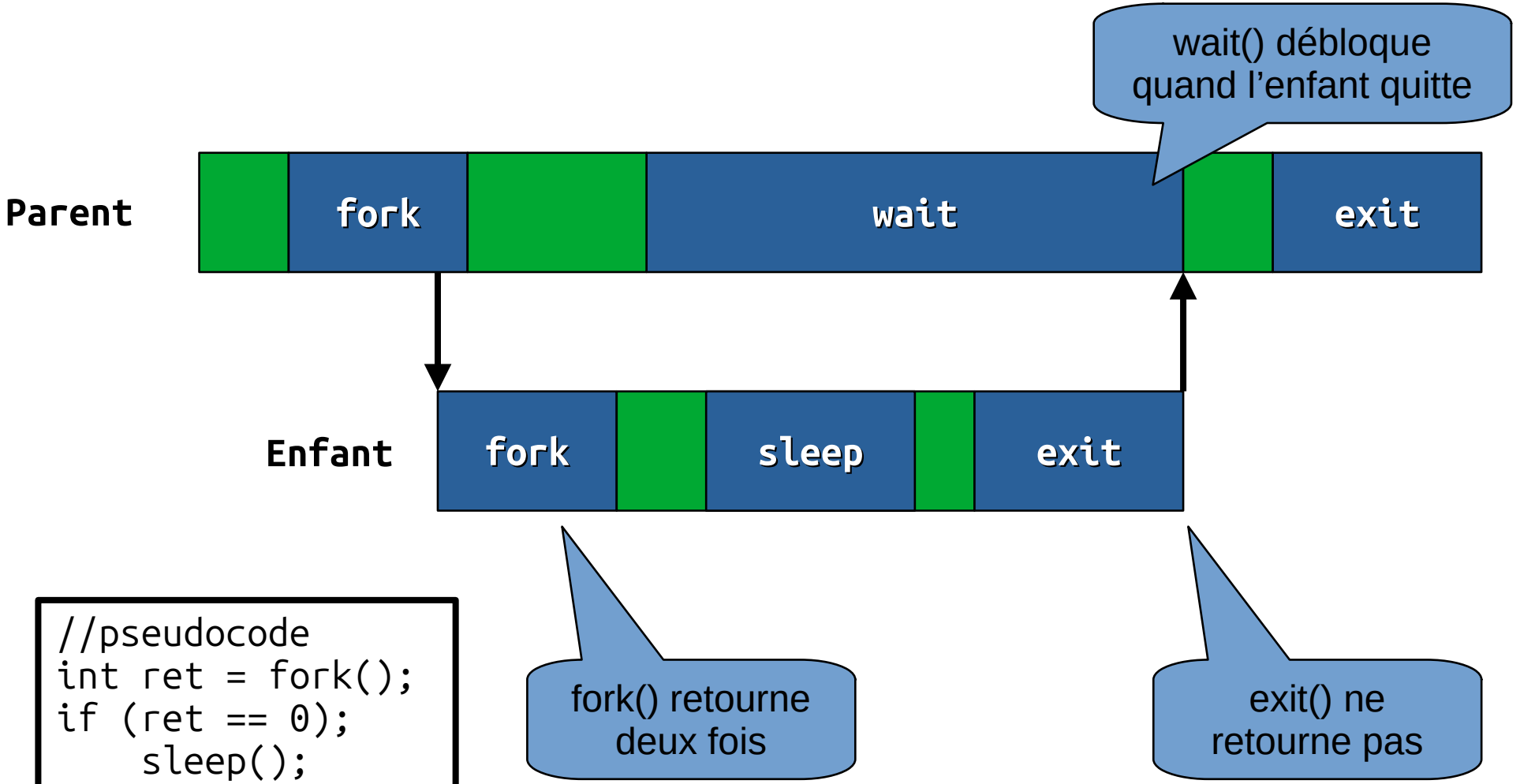
Attendre un processus

- Appel système wait()
- Retourne seulement quand l'enfant s'est terminé (bloque le parent)
- Retourne immédiatement si aucun enfant
  - Sinon, on attendrait un événement qui ne viendrait jamais!
- Typiquement, fork() et wait() s'utilisent ensemble

Remplacer le programme en cours

- Appel système exec()
  - Suffixe v/l : arguments variables ou fixes
  - Suffixe e : Environnement
  - Suffixe p : chercher le programme dans le $PATH
- Réinitialise l’espace mémoire
- Ne retourne pas en cas de succès
- Les descripteurs de fichiers ouverts sont accessibles au nouveau programmes

Représentation temporelle

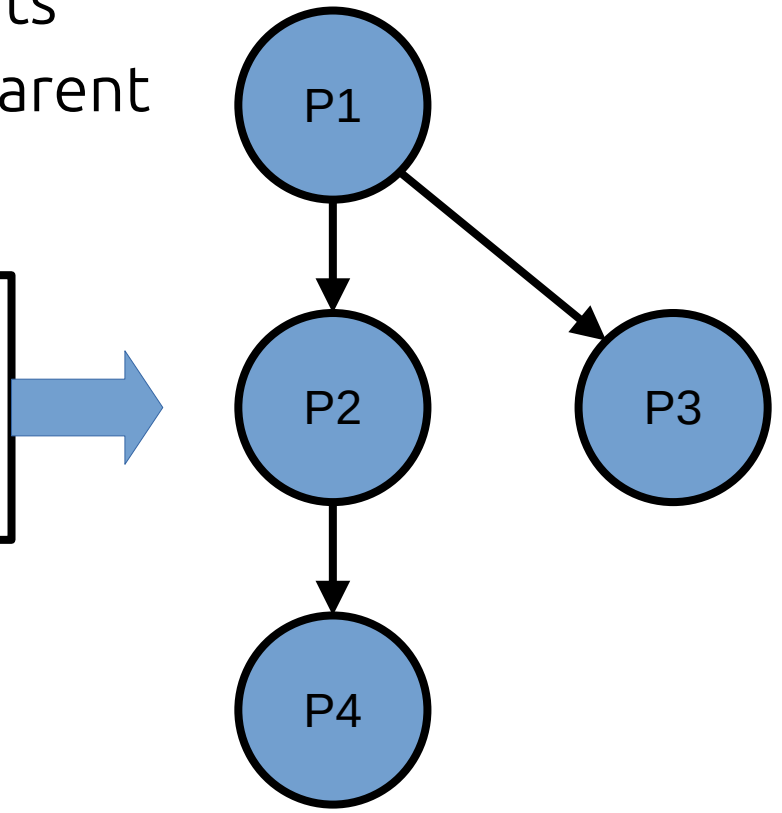



```
//pseudocode
int ret = fork();
if (ret == 0);
    sleep();
if (ret > 0)
    wait();
exit();
```

Représentation en arbre

- Chaque processus est un noeud d'un arbre
- Le parent et son enfant sont reliés par un arc
- Un parent peut avoir plusieurs enfants
- Un enfant ne peut avoir qu'un seul parent
- Exemple: appels successif à fork()

```
fork(); // P1 crée P2
fork(); // P1 crée P3, P2 crée P4
```



Cas spéciaux

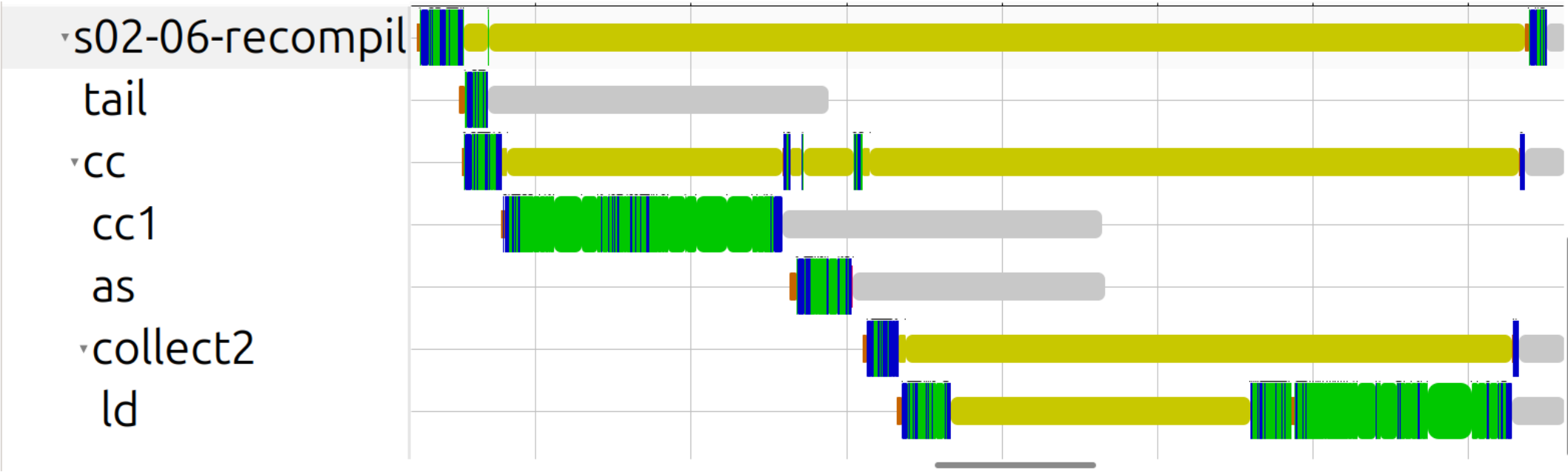
- Si le parent se termine avant l'enfant
  - L'enfant continue de s'exécuter en arrière-plan
  - L'enfant est adopté par un processus racine (ultimement le PID 1)
- Si l'enfant se termine, mais le parent n'a pas lu son code de retour avec wait() ou waitpid()
  - Le système d'exploitation ne peut pas encore libérer toutes les ressources associées à l'enfant terminé, mais il ne peut plus s'exécuter
  - L'enfant est dans l'état spécial "Zombie"

Exemple : compilation en direct

```
#!/usr/bin/env -S sh -c '[ "$0".bin -nt "$0" ] || ↵
tail -n+2 "$0" | cc -x c - -o "$0".bin || ↵
exit 1; exec "$0".bin "$@"'

#include <stdio.h>
int main(void) {
    printf("Hello, world!\n"); return 0;
}
```

test : bin inexistant ou périmé
tail | cc : compilation
exec : lancement



Création d'un fil d'exécution

- Similaire à un processus
- Différence : tous les fils d'exécution d'un processus partagent l'espace mémoire
- Exemple: une liste accessible dans un fil d'exécution est aussi accessible dans un autre
- Identifié par un Thread ID (TID)
- Démarré avec **pthread_create()**
- Attendre la fin avec **pthread_join()**

Lancement d'un fil d'exécution

```c
/*
 * Démarrage d'un seul fil d'exécution
 */
void *travail(void *arg) {
  int *num = (int *)arg;
  printf("num=%d\n", *num);
  return NULL;
}

int demo() {
  // Lancement
  pthread_t thread;
  int num = 42;
  pthread_create(&thread, // identifiant du fil d'exécution type pthread_t
                 NULL,    // attributs (NULL == attributs par défaut)
                 travail,    // routine à exécuter
                 &num     // argument passés à la routine
  );
  // Attendre la fin du fil d'exécution
  pthread_join(thread, NULL);
  return 0;
}
```

f.le d'exec secondaire

exit

f.le d'exec principale

non determine → on sait pas si c'est avant ou apres

Va Bloquer tant que f.le d'exec. n'est pas termine.

Lancement de deux fils d'exécution

```c
/*
 * Démarrage de 2 fils d'exécution. Une variable locale et globale
 * est incrémentée par les deux fils d'exécutions en même temps.
 */
int n = 2;
pthread_t thread[n] = {};
for (long i = 0; i < n; i++) {
  pthread_create(&thread[i], // identifiant du fil d'exécution type pthread_t
                 NULL,       // attributs (NULL == attributs par défaut)
                 werk,       // routine à exécuter
                 (void *)i   // argument passés à la routine
  );

}

/*
 * Attendre la fin des fils d'exécution
 */
for (long i = 0; i < n; i++) {
  pthread_join(thread[i], NULL);
}
```

* volatile long glob = 0

Tâche dans le code noyau

- Représentation d'une tâche: struct task_struct
  - Voir include/linux/sched.h
- Élément central du système d'exploitation
  - PID, TID, numéro de processeur courant, etc.
  - Liste des enfants
  - Liste des fichiers ouverts
  - Liste des pages allouées
  - Statistique du temps d'exécution
  - … et de nombreux autres champs!

Commandes et fonctions

- apropos et man
- fork(), wait(), exit()
- exec()
- pthread_create(), pthread_join()
- ulimit : limiter les ressources d'un processus

man stat
man 2 stat
apropos stat

list

Tip: cmake -G Ninja -S . -B

ninja

git clean -f

gcc -c foo.c -o foo.o
gcc -c bar bar.o
gcc -c main

gcc -o superprog.exe main.o foo.o bar.o