Systèmes d'exploitation et processus

Architecture matérielle, systèmes d'exploitation et réseaux

1 Le système d'exploitation

Un **système d'exploitation** (OS, *Operating System*) est le logiciel fondamental qui fait le lien entre le matériel (processeur, mémoire, disque) et les programmes de l'utilisateur. Parmi ses fonctions principales : la gestion des processus, la gestion des fichiers, la gestion de la mémoire, la gestion des périphériques et la sécurisation du système.

Dans ce chapitre, nous nous intéressons à la **gestion des processus** : comment le système crée, organise et fait cohabiter les programmes en cours d'exécution.

2 Programme et processus

Un **programme** est un fichier stocké sur le disque. C'est une description **statique** : un ensemble d'instructions qui ne fait rien tant qu'il n'est pas lancé.

Un **processus** est ce qui se passe quand on exécute ce programme. C'est une entité **dynamique** : le programme en cours d'exécution, avec ses données, son état d'avancement et ses ressources.

Analogie. Un programme, c'est une recette de cuisine écrite dans un livre. Un processus, c'est une personne en train de réaliser cette recette, avec ses ingrédients devant elle et un état d'avancement propre. Plusieurs personnes peuvent réaliser la même recette en parallèle : chacune constitue un processus distinct.

Cette distinction permet de désigner chaque tâche **individuellement** (même si plusieurs exécutent le même programme) et d'agir sur chacune de façon **indépendante** : en mettre une en pause, en terminer une autre, gérer des priorités.

3 Anatomie d'un processus

3.1 Identifiants

Pour gérer chaque processus, le système lui attribue :

— un **PID** (*Process IDentification*) : numéro unique du processus ;

— un **PPID** (*Parent PID*) : le PID du processus qui l'a créé (son « père ») ;

— un **UID** (*User IDentification*) : l'identifiant de l'utilisateur qui l'a lancé.

Un processus peut en créer d'autres : on parle de **processus fils**. Certains processus fonctionnent en permanence en arrière-plan dès le démarrage de la machine : ce sont les **services** (Windows) ou les **daemons** (Linux).

3.2 Espace mémoire

Le système alloue à chaque processus un espace en **mémoire virtuelle** contenant :

— le **code** du programme à exécuter ;

— les **variables globales** ;

— une **pile** (paramètres de fonctions, variables locales, résultats) ;

— une réserve de mémoire pour les bibliothèques ;

— le **contexte d'exécution** : une copie de l'état de tous les registres du processeur.

3.3 Changement de contexte

Quand un processus est interrompu pour laisser la place à un autre, le système **sauvegarde** son contexte en mémoire. Quand il reprend la main, le contexte est **restauré** dans les registres du processeur et l'exécution reprend exactement là où elle s'était arrêtée.

Ces changements de contexte sont très rapides (de l'ordre de la microseconde), ce qui donne à l'utilisateur l'impression que tous les programmes s'exécutent simultanément. C'est le **pseudo-parallélisme**. Avec un processeur multi-cœurs, certains processus peuvent aussi s'exécuter en vrai parallèle.

④ Cycle de vie d'un processus

Au cours de son existence, un processus passe par plusieurs **états** :

— **nouveau** : vient d'être créé ;

— **prêt** : attend son tour pour accéder au processeur ;

— **élu** : en cours d'exécution sur le processeur ;

— **bloqué** : en attente d'une ressource (saisie clavier, lecture disque...) ;

— **terminé** : exécution achevée.

Les transitions entre ces états suivent une logique précise :

1. Un processus est **créé** → il passe à l'état **prêt**.
2. L'ordonnanceur lui donne accès au processeur → il passe à l'état **élu**.
3. S'il a besoin d'une ressource non disponible → il passe à l'état **bloqué** (le processeur ne doit pas attendre).
4. La ressource devient disponible → il repasse à l'état **prêt** (car entre-temps, un autre processus occupe le processeur).
5. Un processus ne peut **terminer** que depuis l'état **élu**.

L'ordonnanceur peut aussi **préempter** (interrompre) un processus élu pour donner la main à un autre : le processus interrompu repasse alors de **élu** à **prêt**.

⑤ Observer les processus sous Linux

Commande	Rôle
`ps -ef`	Tous les processus avec détails (UID, PID, PPID...)
`ps aux`	Tous les processus avec utilisation mémoire et CPU
`pstree`	Processus sous forme d'arbre (relations père-fils)
`top`	Suivi en temps réel (classé par % CPU)
`pidof <nom>`	PID d'un processus par son nom
`kill <PID>`	Envoyer un signal de terminaison
`killall <nom>`	Terminer tous les processus d'un programme

5.1 Créer un processus

Tout processus est créé par un autre processus, son **père** : le shell crée un processus pour chaque commande tapée, et le système lui attribue un nouveau PID. Depuis un programme Python, le module `subprocess` lance une commande dans un processus fils et attend sa fin ; le module `os` donne les identifiants.

```python
import os
import subprocess

print(f"Je suis le processus {os.getpid()} créé par {os.getppid()}")
resultat = subprocess.run(["ls", "-l"], capture_output=True, text=True)
print(resultat.stdout)
```

Dans un terminal, le caractère `&` lance une commande en arrière-plan et rend la main tout de suite : le shell affiche le PID du processus créé, que l'on retrouve avec `ps` et que l'on arrête avec `kill`.

```
sleep 300 &
[1] 4821
ps -o pid,ppid,stat,comm -p 4821
kill 4821
```

5.2 Lire la sortie de `top`

La commande `top` affiche en continu l'état du système et des processus, classés par consommation de processeur :

```
top - 10:42:07 up 3 days,  2:15,  2 users,  load average: 0.52, 0.38, 0.30
Tasks: 213 total,   1 running, 212 sleeping,   0 stopped,   0 zombie
%Cpu(s):  3.1 us,  1.0 sy,  0.0 ni, 95.6 id,  0.3 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   7852.4 total,   2213.9 free,   3106.0 used,   2532.5 buff/cache

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   4821 eleve     20   0  512340 184216  96520 S   2.7   2.3   1:12.45 firefox
   2210 eleve     20   0   38792  22140  16324 R   0.7   0.3   0:03.10 python3
```

```
    1 root      20   0  168540  13252   8420 S   0.0   0.2   0:04.62 systemd
```

— première ligne : heure, durée depuis le démarrage, nombre d'utilisateurs, **charge moyenne** sur 1, 5 et 15 minutes (nombre moyen de processus prêts ou en exécution) ;

— `Tasks` : nombre de processus par état (en exécution, en attente, arrêtés, zombies) ;

— `%Cpu(s)` : part du temps processeur passée en mode utilisateur (`us`), en mode noyau (`sy`), inactive (`id`) ou en attente d'entrées-sorties (`wa`) ;

— `Mem` : mémoire totale, libre, utilisée et cache ;

— puis, pour chaque processus : `PID`, propriétaire, priorité (`PR`, `NI`), mémoire virtuelle et résidente (`VIRT`, `RES`), **état** `S` (`R` en exécution, `S` en attente, `D` en attente d'entrée-sortie, `T` arrêté, `Z` zombie), pourcentages de processeur et de mémoire, temps processeur cumulé et nom de la commande.

Dans `top`, la touche `k` suivie d'un PID envoie un signal à un processus, `q` quitte.

6 L'ordonnancement

Plusieurs processus peuvent être dans l'état **prêt** simultanément, mais un processeur (mono-cœur) ne peut en exécuter qu'un seul à la fois. Comment choisir lequel ? C'est le rôle de l'**ordonnanceur** (*scheduler*), qui décide de l'ordre de passage.

6.1 Politiques d'ordonnancement

— **FIFO** (*First In, First Out*) : premier arrivé, premier servi. Simple, mais un processus long bloque tous les suivants (effet de convoi).

— **SJF** (*Shortest Job First*) : le plus court d'abord. Optimal en temps d'attente moyen, mais il faut connaître à l'avance la durée de chaque processus.

— **Priorité** : chaque processus reçoit un niveau de priorité. Risque : les processus de faible priorité peuvent ne jamais être élus (famine).

— **Tourniquet** (*Round Robin*) : chaque processus reçoit un **quantum de temps** fixe. S'il n'a pas terminé, il retourne en bout de file d'attente.

6.2 Grandeurs caractéristiques

Grandeur	Définition
Temps d'arrivée	Instant où le processus entre dans la file d'attente
Durée	Nombre de quanta nécessaires à son exécution complète
Temps de terminaison	Instant où le processus achève son exécution
Temps de séjour	Temps de terminaison − temps d'arrivée
Temps d'attente	Temps de séjour − durée du processus

6.3 Exemple avec SJF

Considérons cinq processus :

Processus	P1	P2	P3	P4	P5
Durée (quanta)	3	6	4	2	1
Temps d'arrivée	0	1	4	6	7

Avec SJF, à chaque instant où le processeur se libère, on élit le processus prêt de plus courte durée. Chaque processus s'exécute **entièrement** sans interruption (SJF non préemptif).

Frise chronologique :

	0–3	3–9	9–10	10–12	12–16
Processus	P1 ✓	P2 ✓	P5 ✓	P4 ✓	P3 ✓

Résultats :

Processus	P1	P2	P3	P4	P5
Durée (quanta)	3	6	4	2	1
Temps d'arrivée	0	1	4	6	7
Temps de terminaison	3	9	16	12	10
Temps de séjour	3	8	12	6	3
Temps d'attente	0	2	8	4	2

Temps d'attente moyen : $\frac{0+2+8+4+2}{5} = 3{,}2$ quanta.

6.4 Exemple avec le tourniquet (Round Robin)

Reprenons les mêmes processus, avec un **quantum de temps = 2**.

L'ordonnanceur accorde **2 quanta** à chaque processus. Si le processus n'a pas terminé à l'issue de son quantum, il retourne **en bout de file** d'attente. Lorsqu'un processus est préempté, les nouveaux arrivants sont d'abord ajoutés à la file, puis le processus préempté est placé en bout de file. Un processus qui arrive à l'instant exact où un quantum se termine est lui aussi placé avant le processus préempté.

Frise chronologique :

	0–2	2–4	4–5	5–7	7–9	9–11	11–12	12–14	14–16
Processus	P1	P2	P1 ✓	P3	P2	P4 ✓	P5 ✓	P3 ✓	P2 ✓

Résultats :

Processus	P1	P2	P3	P4	P5
Temps de terminaison	5	16	14	11	12
Temps de séjour	5	15	10	5	5
Temps d'attente	2	9	6	3	4

Temps d'attente moyen : $\frac{2+9+6+3+4}{5} = 4{,}8$ quanta.

6.5 Comparaison et systèmes réels

Avec SJF, le temps d'attente moyen est de 3,2 quanta (meilleur), mais SJF suppose de connaître à l'avance la durée de chaque processus. Le tourniquet est plus **équitable** : chaque processus reçoit régulièrement un quantum de temps, ce qui évite la famine.

En pratique, les systèmes d'exploitation modernes combinent tourniquet et **priorités dynamiques**. L'ordonnanceur de Linux, **EEVDF** (*Earliest Eligible Virtual Deadline First*), en service depuis la version 6.6 du noyau (2023) où il a remplacé CFS (*Completely Fair Scheduler*), attribue à chaque processus une part du processeur proportionnelle à son poids, puis élit celui qui a pris le plus de retard sur cette part. Les processus interactifs, qui dorment souvent, sont ainsi servis rapidement, et aucun processus n'est privé indéfiniment du processeur : il n'y a pas de famine.

⑦ Problèmes de concurrence

7.1 Accès concurrent à une ressource partagée

Considérons un jeu multijoueur avec une variable globale `nb_pions` représentant le nombre de pions disponibles. Il ne reste qu'**un seul pion** et deux processus P1 et P2 veulent en prendre un :

1. P1 est élu, vérifie qu'il reste un pion (oui) et s'apprête à décrémenter `nb_pions`.
2. Mais l'ordonnanceur interrompt P1 **avant** la décrémentation et élit P2.
3. P2 vérifie aussi qu'il reste un pion (oui, car P1 n'a pas eu le temps de décrémenter) et prend le pion.
4. P1 reprend et décrémente à son tour : les deux joueurs ont pris un pion alors qu'il n'y en avait qu'un.

C'est un problème d'**accès concurrent** (*race condition*). La solution consiste à protéger les **sections critiques** du code avec des **verrous** (*mutex*).

7.2 Interblocage

Considérons deux processus P1 et P2, et deux ressources R1 et R2, initialement libres.

1. P1 est **élu**, demande et obtient **R1**. L'ordonnanceur le met en état **prêt**.
2. P2 est **élu**, demande et obtient **R2**. Il poursuit son exécution.
3. P2 demande **R1**, mais R1 est détenue par P1 → P2 passe à l'état **bloqué**.
4. P1 est **élu** à nouveau. Avant de libérer R1, il demande **R2**, mais R2 est détenue par P2 → P1 passe à l'état **bloqué**.

Les deux processus sont **bloqués**, chacun attendant une ressource détenue par l'autre. C'est une situation d'**interblocage** (*deadlock*) : un cycle d'attente sans issue.

7.3 Conditions de Coffman

Edward Coffman a montré en 1971 que l'interblocage ne peut survenir que si **quatre conditions** sont réunies simultanément :

1. **Exclusion mutuelle** : chaque ressource est soit libre, soit attribuée à un seul processus.
2. **Détention et attente** : un processus qui détient déjà une ressource en demande une autre.
3. **Non-préemption** : une ressource ne peut être retirée de force ; seul le processus qui la détient peut la libérer.

4. **Attente circulaire** : il existe un cycle de processus, chacun attendant une ressource détenue par le suivant.

Si l'on supprime **une seule** de ces conditions, l'interblocage devient impossible. C'est le principe de la **prévention**. En pratique, certaines conditions sont plus faciles à supprimer que d'autres :

1. **Supprimer l'exclusion mutuelle** : permettre à plusieurs processus d'utiliser la même ressource simultanément. C'est possible pour une imprimante (on met les documents dans une file d'impression), mais impossible pour un fichier en écriture ou un capteur physique. Cette condition est donc rarement supprimable.
2. **Supprimer la détention et attente** : imposer qu'un processus demande **toutes** ses ressources en une seule fois avant de commencer. S'il ne peut pas toutes les obtenir, il n'en obtient aucune et réessaie plus tard. C'est simple à mettre en œuvre, mais peu efficace : un processus qui a besoin de R1 au début et de R2 beaucoup plus tard bloque R2 inutilement pendant tout ce temps.
3. **Supprimer la non-préemption** : autoriser le système à **retirer de force** une ressource à un processus pour la donner à un autre. C'est faisable pour le processeur (c'est précisément ce que fait la préemption dans l'ordonnancement), mais très difficile pour d'autres ressources : on ne peut pas retirer une imprimante en pleine impression sans corrompre le document.
4. **Supprimer l'attente circulaire** : numéroter toutes les ressources ($R_1 < R_2 < R_3$) et obliger chaque processus à les demander **dans l'ordre croissant** de leur numéro. Exemple : P1 détient R_3 et veut R_1 ; P2 détient R_1 et veut R_3. Avec la règle, P1 aurait dû demander R_1 **avant** R_3 (car $1 < 3$). La situation où P1 détient R_3 en attendant R_1 est donc **interdite par construction** : le cycle ne peut pas apparaître. Contrairement aux autres conditions, celle-ci n'oblige ni à gaspiller des ressources, ni à retirer de force une ressource en cours d'utilisation. C'est la **méthode la plus utilisée** en pratique : il suffit de fixer un ordre et de s'y tenir.

La **détection** est une autre approche : on laisse l'interblocage se produire, puis on l'identifie (en détectant un cycle dans le graphe d'allocation des ressources) et on le résout en terminant un des processus impliqués.

8 Threads

8.1 Pourquoi les threads ?

Nous avons vu que chaque processus possède son propre espace mémoire, isolé des autres. Cette isolation est un atout pour la sécurité, mais elle a un coût : quand deux processus ont besoin de collaborer (par exemple, un navigateur web qui télécharge une page tout en affichant une animation de chargement), la communication entre eux est lente et complexe.

Les **threads** répondent à ce besoin : ils permettent à plusieurs tâches de s'exécuter **au sein d'un même processus**, en partageant directement les mêmes données, sans le surcoût de la communication inter-processus.

8.2 Processus et threads

Un **thread** (ou **processus léger**) est un fil d'exécution à l'intérieur d'un processus. Plusieurs threads d'un même processus **partagent le même espace mémoire** mais possèdent chacun leur propre **pile d'exécution**. Cette légèreté les rend très utiles, mais le partage de mémoire impose une synchronisation rigoureuse pour éviter les accès concurrents.

	Processus	Thread
Mémoire	Isolée	Partagée
Communication	Coûteuse (IPC)	Directe (variables partagées)
Création	Lourde	Légère
Risque	Faible (isolation)	Problèmes de synchronisation

8.3 Les threads en Python

En Python 3, le module `threading` permet de créer des threads :

```python
import threading

def exemple():
    print('Test')

threads = []
for i in range(5):
    t = threading.Thread(target=exemple)
    threads.append(t)
    t.start()
```

La méthode `t.join()` attend que le thread `t` soit terminé avant de continuer.

8.4 Illustration d'un accès concurrent

Remarque 1. Complément hors programme : les threads et les verrous ne sont pas exigibles au baccalauréat ; ils illustrent concrètement la notion d'accès concurrent.

Le programme suivant lance 10 threads qui incrémentent chacun un compteur partagé 10 000 fois. On s'attend à obtenir 100 000 à la fin :

```python
import threading
import time

compteur = 0

def incrementer():
    global compteur
    for _ in range(10_000):
        valeur = compteur
        time.sleep(0)          # rend la main à un autre thread
        compteur = valeur + 1

threads = []
for _ in range(10):
    t = threading.Thread(target=incrementer)
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print(f"Résultat : {compteur}")
```

En exécutant ce programme plusieurs fois, le résultat est **très inférieur** à 100 000 (souvent autour de 10 000). C'est une *race condition* : deux threads lisent la même valeur, l'incrémentent chacun de leur côté, puis écrivent le même résultat, une incrémentation est « perdue ».

Solution avec un verrou (`Lock`) :

```
import threading
import time

compteur = 0
verrou = threading.Lock()

def incrementer():
    global compteur
    for _ in range(10_000):
        with verrou:
            valeur = compteur
            time.sleep(0)
            compteur = valeur + 1
```

Avec le verrou, un seul thread à la fois peut exécuter les trois lignes du bloc `with verrou` (lecture du compteur, pause, écriture). Le résultat est toujours exactement 100 000.