Systèmes d'exploitation

CM5: Gestion des Processus

Ilias Tougui

17 octobre 2025

Plan

Introduction et Motivation

Évolution : De la Monotâche à la Multitâche

Imaginez un ordinateur des années 1960...

- Vous soumettez votre programme via des cartes perforées
- Vous attendez votre tour dans la file (parfois des heures !)
- Le programme s'exécute en **isolation totale**
- Une erreur ? Tout recommencer depuis le début...

Aujourd'hui, sur votre smartphone :

- WhatsApp, Chrome, Spotify, Maps... tous actifs simultanément
- Des dizaines de processus en arrière-plan
- **Comment est-ce possible ?** $\rightarrow$ La gestion de processus !

Pourquoi Avons-nous Besoin du Concept de Processus ?

Le défi : Charger plusieurs programmes en mémoire simultanément

Sans isolation :

- Programme A écrit à l'adresse 0x1000
- Programme B écrit aussi à 0x1000
- → Collision ! Données corrompues
- Crashes système fréquents
- Impossible de garantir la sécurité

Avec le concept de processus :

- Chaque programme dans son espace mémoire protégé
- Adresses virtuelles isolées
- Protection mutuelle garantie
- Stabilité du système

Définition

Un processus est un programme actif qui possède son propre espace mémoire protégé et ses ressources d'exécution.

Processus Utilisateur et Processus Système

Exemple concret : Que se passe-t-il quand vous lancez Firefox ?

Processus utilisateur :

- Firefox s'exécute
- N'accède pas directement au matériel
- Demande des services via des appels système

Processus système :

- Gestionnaire réseau (connexion Internet)
- Gestionnaire de fichiers (favoris, historique)
- Serveur d'affichage (rendu graphique)

La collaboration invisible : Firefox ne gère ni le réseau, ni l'affichage directement. Il communique avec les processus système qui font le travail.

Test de compréhension

Si Firefox plante, votre système s'arrête-t-il ? **Non !**
Pourquoi ? $\rightarrow$ L'isolation protège le noyau. Seul le processus Firefox meurt, vous pouvez le relancer immédiatement.

Partie I: La Concurrence des Processus

Le Défi Suivant : Optimiser l'Utilisation du CPU

Nous avons résolu l'isolation... mais un nouveau problème apparaît !

Observation

Même avec des processus bien isolés, si un seul processus s'exécute à la fois, le CPU reste souvent inutilisé.

Pourquoi ?

- Les processus ne font pas que des calculs
- Ils attendent souvent des données : disque, réseau, clavier...
- Pendant ces attentes, le CPU ne fait rien

→ *Il nous faut une nouvelle innovation : la concurrence des processus*

Le Gaspillage des Ressources Sans Concurrence

Scénario typique : Un processus demande à lire un fichier

1. Le processus lance une lecture disque (opération d'E/S)
2. Le CPU transmet la requête au contrôleur disque
3. Le CPU reste complètement inactif pendant que le disque travaille
4. L'opération dure plusieurs millisecondes (une éternité pour un CPU !)
5. D'autres processus prêts à s'exécuter restent bloqués en attente

Le problème

Le CPU peut rester **inactif 90% du temps** simplement en attendant les opérations d'E/S.

Analogie :
Un chef qui attend devant le four sans rien préparer d'autre !

Gaspillage énorme d'une ressource matérielle coûteuse...

La Solution : Le Multiplexage Temporel du CPU

Principe de la concurrence :

- Lorsque le processus A lance une opération d'E/S (lecture disque, requête réseau), il passe automatiquement en état d'attente
- Le système d'exploitation détecte cette situation et bascule immédiatement le CPU vers le processus B qui est prêt à s'exécuter
- Pendant que A attend la fin de son opération d'E/S, le processus B effectue ses calculs de manière productive
- Dès que l'opération d'E/S de A se termine, celui-ci redevient éligible et pourra reprendre son exécution lors d'un prochain cycle

Résultat

Le CPU peut maintenant être utilisé jusqu'à **90%** de son temps, transformant intelligemment le temps d'attente improductif en temps de calcul utile.

La Concurrence en Action

Exemple concret dans votre quotidien :

1. Votre navigateur Chrome télécharge les images d'une page web depuis Internet (E/S réseau)
2. Pendant ce téléchargement, le processeur fait tourner Spotify pour décoder votre musique en temps réel (calcul intensif)
3. Simultanément, LibreOffice sauvegarde automatiquement votre document en arrière-plan (E/S disque)
4. Le système gère également des dizaines d'autres tâches invisibles : antivirus, notifications, mises à jour, synchronisation cloud

Objectif de la Concurrence

Le Principe du Multiplexage Temporel

Le système d'exploitation alloue le CPU à chaque processus pendant quelques millisecondes, puis bascule vers un autre. Ce changement de contexte se produit des centaines de fois par seconde, créant l'illusion que tous les programmes s'exécutent en même temps.

Objectifs clés : Maximiser l'utilisation du CPU • Améliorer la réactivité • Permettre le multitâche

Quiz de Compréhension

Question 1 : Qu'est-ce qu'un processus ?

a) Un fichier exécutable stocké sur le disque
b) Un programme en cours d'exécution avec son environnement protégé
c) Un service système uniquement
d) Une application installée

Question 2 : Pourquoi le CPU reste-t-il inactif 90% du temps dans un système monotâche ?

a) Parce que le processeur est trop lent
b) Parce que les programmes sont mal écrits
c) Parce qu'il n'y a pas assez de mémoire
d) Parce qu'il attend pendant les opérations d'E/S sans rien faire

Quiz de Compréhension (suite)

Question 3 : Quel est l'objectif principal de la concurrence ?

a) Réduire la consommation électrique

b) Augmenter la taille de la mémoire

c) Transformer le temps d'attente improductif en temps de calcul utile

d) Simplifier la programmation

Réponses

1) b **2)** d **3)** c

Partie II: Concept de processus

Évolution du Vocabulaire : Jobs, Tasks, Processus

Trois époques, trois noms, un seul concept

Époque	Terme	Contexte technologique
Années 1960	Jobs	Systèmes par lots, cartes perforées
Années 1970-80	Tasks	Temps partagé, multi-utilisateurs
Aujourd'hui	Processus	Terme moderne unifié

Exemples d'aujourd'hui : tout est processus !

- Votre navigateur Chrome, Spotify en arrière-plan
- Le gestionnaire de réseau, l'antivirus qui scanne
- Même le pilote de votre imprimante

Définition unifiée

Processus = Toute activité exécutée par le système, visible ou invisible

Pourquoi Conserver Parfois le Terme "Job" ?

Si "processus" est le terme moderne, pourquoi parler encore de "jobs" ?

Raison : L'héritage technique

- La théorie fondamentale des OS a été développée à l'époque des systèmes par lots
- Des expressions techniques universellement reconnues :
  - **Job scheduling** : ordonnancement des travaux
  - **Job queue** : file d'attente des travaux
  - **Long-term scheduler** : appelé aussi "job scheduler"
- Remplacer "job" changerait le sens historique de ces termes établis

Notre convention

Nous utiliserons **processus** par défaut, mais **job** dans les expressions techniques consacrées où ce terme est standard.

Programme vs Processus : Quelle différence ?

Programme

- Fichier statique sur le disque
- Contient des instructions passives
- Peut être copié, déplacé, supprimé
- N'utilise aucune ressource CPU/mémoire

Processus

- Programme en cours d'exécution
- Entité dynamique et active
- Consomme du temps CPU
- Occupe de la mémoire vive

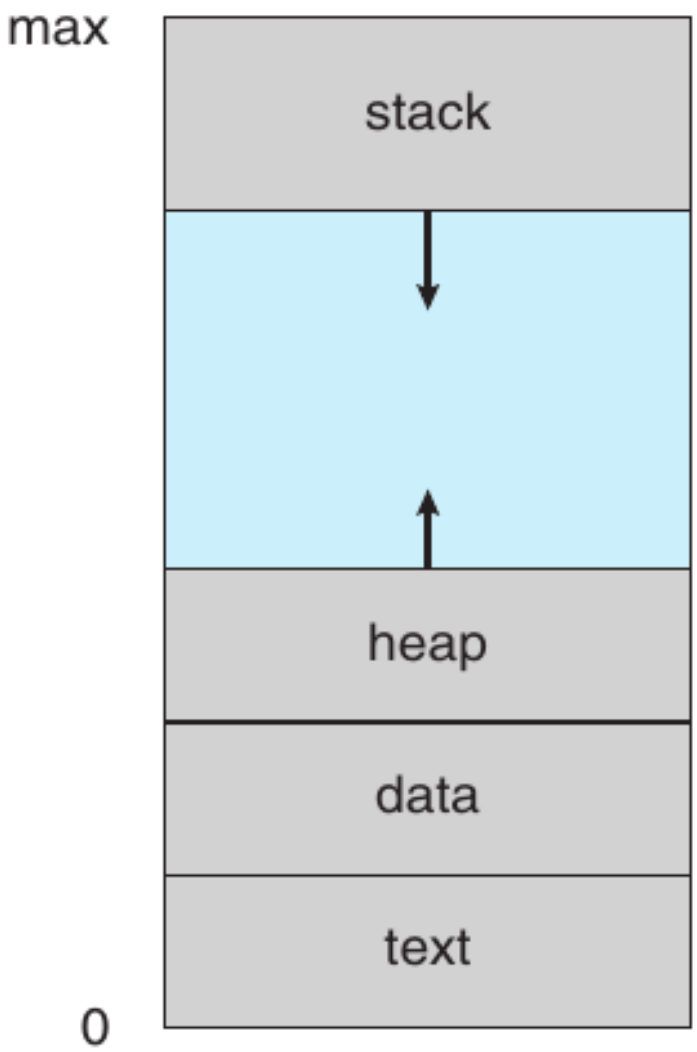
Exemple concret

Le fichier `firefox.exe` (programme) peut générer plusieurs fenêtres Firefox ouvertes simultanément (plusieurs processus distincts).

Organisation de la Mémoire d'un Processus

Comment un processus organise-t-il son espace mémoire ?

1. **Section texte** (*text*)
   - Contient le code exécutable
   - Zone en **lecture seule**
2. **Section de données** (*data*)
   - Stocke les variables globales et statiques
3. **Heap** (tas)
   - Stocke les objets, tableaux, structures dont la taille est déterminée au runtime
4. **Stack** (pile)
   - Stocke les appels de fonctions, variables locales



Sections Mémoire : Statiques vs. Dynamiques

Deux types de sections avec des comportements différents :

Sections statiques

- Text et Data
- Tailles **fixes**
- Déterminées à la compilation
- Ne changent jamais pendant l'exécution

Sections dynamiques

- Stack et Heap
- Tailles **variables**
- Évoluent pendant l'exécution
- Grandissent et rétrécissent selon les besoins

Exemple concret

Un programme de 100 lignes de code aura toujours la même section *text*, mais sa stack variera selon la profondeur des appels de fonctions et son heap selon les allocations dynamiques (`malloc/free`).

Le Mécanisme de la Pile (Stack)

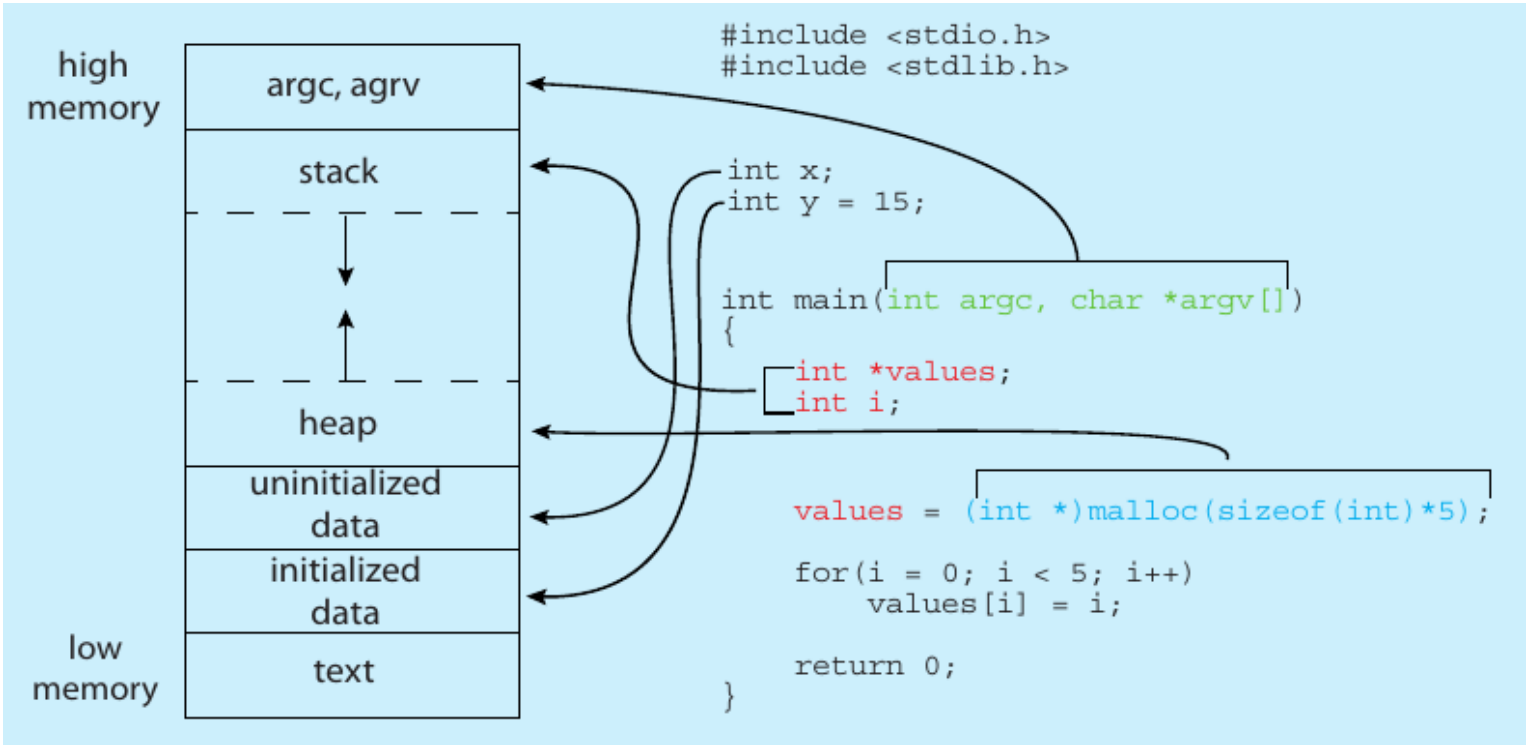
Comment fonctionne la stack lors d'un appel de fonction ?

1. **Appel de fonction** : Un enregistrement d'activation (*activation record* ou *stack frame*) est empilé au sommet de la stack
2. **Contenu de l'enregistrement** :
   - Paramètres de la fonction
   - Variables locales
   - Adresse de retour (où reprendre l'exécution)
3. **Fin de fonction** : L'enregistrement est dépilé, libérant automatiquement la mémoire

Remarque : C'est pourquoi les variables locales "disparaissent" après la fin d'une fonction !

Organisation Mémoire Spécifique à un Programme C

Le schéma de l'organisation mémoire d'un **programme C** est une application concrète du modèle de processus général que nous avons étudié.



Organisation Mémoire Spécifique à un Programme C

Organisation Détaillée de la Section Data en C

La section **data** est subdivisée en deux zones pour optimiser l'espace dans le fichier exécutable :

1. **Segment Data (données initialisées)**
   - Variables globales et statiques avec valeur initiale explicite
   - Stockées directement dans le fichier exécutable
   - Exemple : `int y = 15;` ou `static char nom[] = "Linux";`

2. **Segment BSS (données non initialisées)**
   - Variables globales et statiques sans valeur initiale
   - N'occupent **pas d'espace** dans le fichier exécutable
   - Initialisées automatiquement à 0 au chargement du programme
   - Exemple : `int tableau[1000];` ou `static double resultat;`

Zone Spéciale : Arguments de la Ligne de Commande

Où sont stockés argc et argv ?

- Une zone mémoire séparée est allouée pour les paramètres passés au programme via la ligne de commande
- argc (Argument Count), argv (Argument Vector)
- Ces données sont accessibles via `int main(int argc, char *argv[])`

Exemple concret

Si vous exécutez : `./programme -v fichier.txt`

```
→ argc = 3
→ argv[0] = "./programme"
→ argv[1] = "-v"
→ argv[2] = "fichier.txt"
```

La Commande `size` : Analyser un Exécutable

Syntaxe : `size <nom_du_fichier_executable>`

Interprétation des colonnes :

- `text` Taille du code exécutable (instructions machine)
- `data` Taille des données globales/statiques initialisées
- `bss` Taille des données globales/statiques non initialisées (*Block Started by Symbol* - terme historique)
- `dec` Somme `text + data + bss` en décimal
- `hex` Même somme en format hexadécimal

Question importante

Pourquoi les tailles de la *stack* et du *heap* n'apparaissent-elles pas ?

Réponse : Ces sections sont dynamiques et créées uniquement **à l'exécution**. La commande `size` analyse le fichier exécutable statique, pas le processus en mémoire.

Le Cycle de Vie d'un Processus

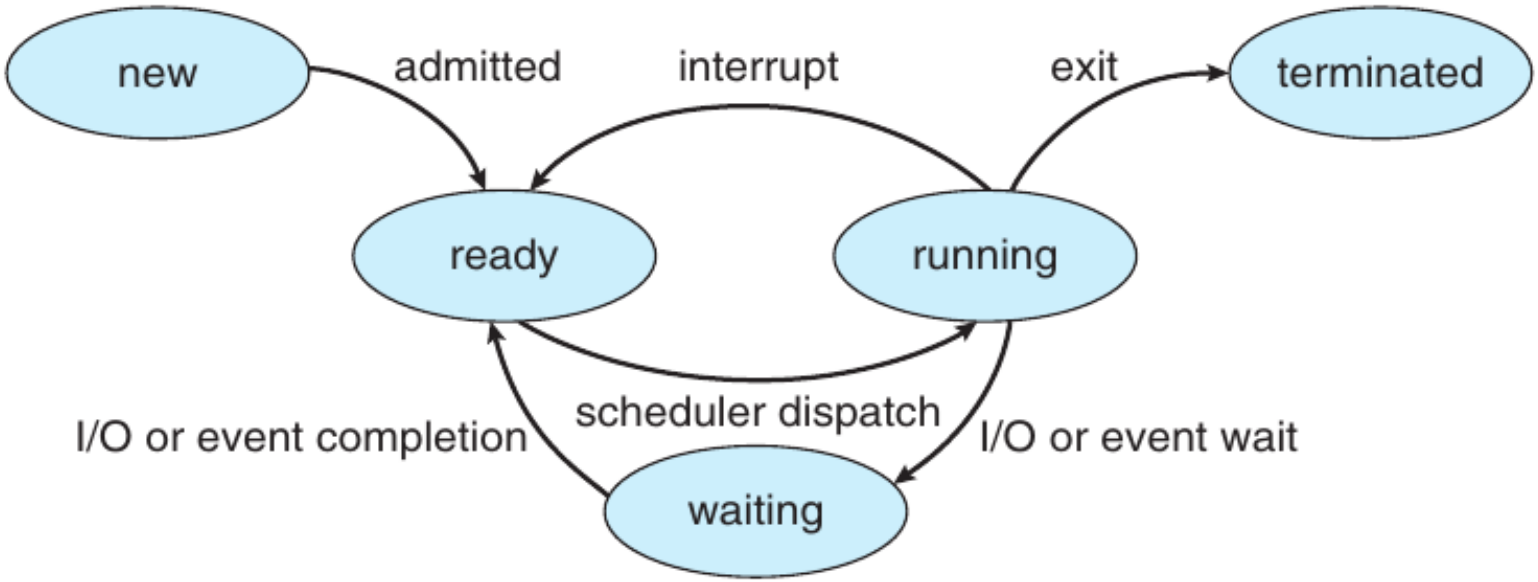
Un processus n'est pas toujours en train de s'exécuter !

Au cours de son existence, un processus passe par différents **états** qui reflètent son activité courante. Comprendre ces états est essentiel pour saisir comment le système d'exploitation gère la concurrence.

Les cinq états fondamentaux :

New	Le processus est en cours de création (allocation mémoire, chargement du code)
Ready	Le processus est prêt à s'exécuter et attend simplement d'être assigné au CPU
Running	Le processus est actuellement en exécution sur un cœur du processeur
Waiting	Le processus attend un événement externe (fin d'I/O, signal, verrou)
Terminated	L'exécution est achevée, les ressources sont en cours de libération

Diagramme de Transitions d'États



Le cycle de vie complet d'un processus

Contrainte physique importante

Sur un cœur CPU, un **seul** processus peut être dans l'état Running à un instant donné, mais **plusieurs** processus peuvent simultanément être dans les états Ready ou Waiting.

Comprendre les Transitions d'États

Quelles actions provoquent les changements d'état ?

1. **New** → **Ready** (*admitted*)
   - Le système a terminé la création du processus
   - Le processus est chargé en mémoire et prêt à démarrer

2. **Ready** → **Running** (*scheduler dispatch*)
   - L'ordonnanceur sélectionne ce processus parmi tous les processus prêts
   - Le processus obtient le CPU et commence/reprend son exécution

3. **Running** → **Waiting** (*I/O or event wait*)
   - Le processus demande une opération d'E/S (lecture fichier, réseau)
   - Il libère volontairement le CPU en attendant la réponse

Transitions d'États (suite)

4. **Waiting** → **Ready** (*I/O or event completion*)
   - L'événement attendu s'est produit (données reçues, verrou libéré)
   - Le processus redevient éligible pour l'exécution
   - Il ne s'exécute pas immédiatement, il attend son tour !

5. **Running** → **Ready** (*interrupt*)
   - Le processus est préempté (interrompu de force)
   - Son temps CPU alloué (*quantum*) est écoulé
   - L'ordonnanceur donne le CPU à un autre processus

6. **Running** → **Terminated** (*exit*)
   - Le processus a terminé son exécution (normalement ou par erreur)
   - Le système libère ses ressources (mémoire, fichiers ouverts)

Exemple Concret : Firefox en Action

Suivons le navigateur Firefox à travers ses états :

1. Vous double-cliquez sur l'icône Firefox
   - État : New → Le système charge Firefox en mémoire
2. Firefox est chargé et prêt
   - État : Ready → Attend que le CPU soit disponible
3. L'ordonnanceur lui donne le CPU
   - État : Running → Firefox s'exécute, affiche sa fenêtre
4. Vous tapez une URL et appuyez sur Entrée
   - État : Waiting → Firefox attend la réponse du serveur web
   - Pendant ce temps, d'autres processus utilisent le CPU !
5. Les données arrivent du réseau
   - État : Ready → Firefox peut reprendre l'exécution
6. Vous fermez Firefox
   - État : Terminated → Libération des ressources

Du Concept à l'Implémentation

Nous venons de voir :

- Qu'un processus = Programme + État d'exécution + Contexte mémoire
- Qu'un processus passe par différents états (New, Ready, Running, Waiting, Terminated)
- Que le système bascule rapidement entre des centaines de processus

Question cruciale :

Comment le système "se souvient-il" de chaque processus ?

Quand un processus passe de Running à Waiting, comment le système sait-il :

- Où reprendre l'exécution ? (quelle instruction ?)
- Quelles étaient les valeurs des registres ?
- Quels fichiers sont ouverts ?
- Quelle est sa priorité ?

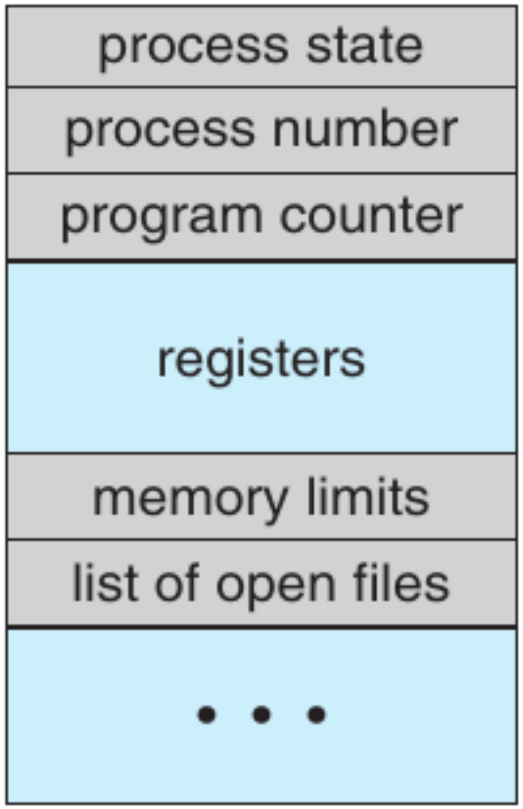
→ **Réponse :** Le Bloc de Contrôle de Processus (PCB)

Le PCB : La Carte d'Identité du Processus

Le Bloc de Contrôle de Processus (*Process Control Block* ou PCB) est une structure de données maintenue par le système d'exploitation pour **chaque processus**. C'est la "fiche d'identité complète" du processus.

Analogie :

- Le PCB est comme un dossier médical : quand vous quittez le cabinet du médecin, votre dossier conserve tout votre historique pour la prochaine visite
- De même, quand un processus quitte le CPU, son PCB sauvegarde tout son état pour qu'il puisse reprendre exactement où il s'était arrêté



Structure d'un PCB

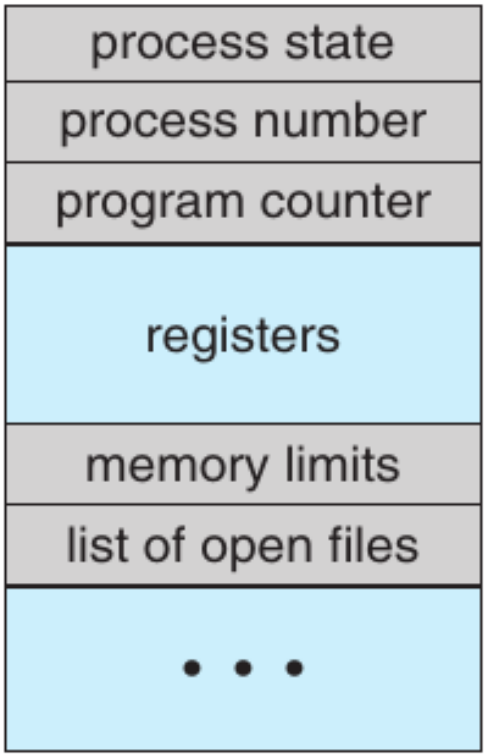
Le Process Control Block (PCB)

Le PCB est la structure de données qui contient toutes les informations nécessaires pour gérer un processus.

Composants principaux :

- Identification du processus
- État d'exécution
- Informations d'ordonnancement
- Gestion de la mémoire
- Gestion des E/S

Chaque composant sera détaillé dans les slides suivantes



Structure complète d'un PCB

PCB : Identification du Processus

Chaque processus possède des identifiants uniques.

- **PID (Process ID)** : Numéro unique du processus
  - Exemple : 1234
  - Permet au système de distinguer chaque processus
- **PPID (Parent Process ID)** : PID du processus parent
  - Établit la hiérarchie des processus
- **UID (User ID)** : Propriétaire du processus
  - Détermine les droits d'accès

Exemple

Firefox : PID=5432, PPID=1, UID=1000 (votre utilisateur)

PCB : État d'Exécution

Quand le système arrête un processus, il doit noter "où il en était" pour pouvoir le reprendre plus tard.

- **État du processus**
  - New, Ready, Running, Waiting, Terminated
- **Position dans le programme**
  - Quelle est la prochaine ligne de code à exécuter ?
  - Le système note cette position pour continuer au bon endroit
- **Valeurs des variables internes**
  - Les calculs en cours dans le processeur
  - Comme une calculatrice qui garde en mémoire ses résultats intermédiaires

Analogie

Comme mettre un marque-page dans un livre : on peut le fermer et le rouvrir exactement à la bonne page

Le PCB en Action : Changement de Contexte

Exemple concret : basculer entre deux processus

Scénario

Processus A : Navigateur Firefox (PID=1234)
Processus B : Lecteur de musique (PID=5678)

1. **Mettre Firefox en pause**
   - Noter où Firefox était dans son code
   - Sauvegarder ses calculs en cours
   - Statut : En exécution $\rightarrow$ Prêt
2. **Choisir le prochain processus**
   - L'ordonnanceur décide de lancer le lecteur de musique
3. **Reprendre le lecteur de musique**
   - Restaurer sa position dans le code
   - Restaurer ses calculs en cours
   - Statut : Prêt $\rightarrow$ En exécution

PCB : Informations d'Ordonnancement

L'ordonnanceur utilise ces informations pour choisir quel processus exécuter.

- **Priorité du processus**
  - Valeur numérique (ex: 0-139 sous Linux)
  - Plus la valeur est basse, plus la priorité est haute
- **Pointeurs vers les files d'attente**
  - Ready queue : processus prêts à s'exécuter
  - Waiting queue : processus en attente d'E/S
- **Statistiques CPU**
  - Temps CPU déjà consommé
  - Temps d'attente

Ces informations permettent des décisions d'ordonnancement équitables

PCB : Gestion de la Mémoire

Le PCB isole et protège l'espace mémoire de chaque processus.

- **Limites de l'espace d'adressage**
  - Adresse de début et de fin
  - Exemple : 0x00000000 à 0xBFFFFFFF
- **Table des pages**
  - Pointeur vers la table de pages du processus
  - Permet la traduction adresses virtuelles $\rightarrow$ physiques
- **Pointeurs vers les sections mémoire**
  - Text segment (code)
  - Data segment (variables globales)
  - Heap (allocation dynamique)
  - Stack (variables locales)

Chaque processus a sa propre vue isolée de la mémoire

PCB : Gestion des Entrées/Sorties

Le système suit toutes les ressources d'E/S utilisées par chaque processus.

- **Table des fichiers ouverts**
  - Liste des descripteurs de fichiers (File Descriptors)
  - 0 = stdin, 1 = stdout, 2 = stderr
  - Autres fichiers : 3, 4, 5, etc.
- **Périphériques alloués**
  - Liste des ressources matérielles utilisées
  - Exemple : imprimante, carte son
- **Opérations d'E/S en cours**
  - Requêtes d'E/S en attente
  - Buffers associés

Exemple

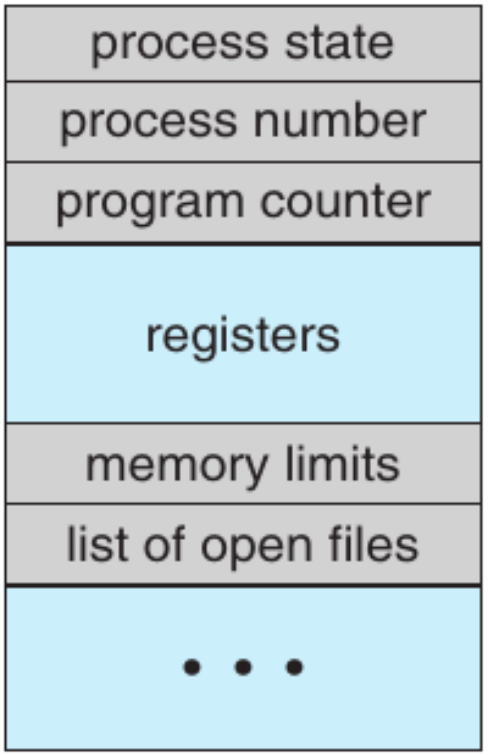
Un processus qui lit un fichier : FD=3 → /home/user/data.txt

Récapitulatif : Le PCB Complet

Le PCB est la "carte d'identité complète" du processus.

- **Identification** : Qui est le processus ?
- **État d'exécution** : Où en est-il ?
- **Ordonnancement** : Quand s'exécutera-t-il ?
- **Mémoire** : Où sont ses données ?
- **E/S** : Quelles ressources utilise-t-il ?

Sans le PCB, le système ne pourrait pas gérer le multitâche ni isoler les processus les uns des autres.



Structure d'un PCB

Partie III: Gestion des Processus sous Linux

Variables d'environnement et terminologie

Variables shell pour les processus :

- `$$` PID du shell actuel (Process ID)
  Identifiant unique du processus bash en cours d'exécution
- `$PPID` PID du processus parent (Parent Process ID)
  Identifiant du processus qui a lancé le shell actuel
- `$!` PID du dernier processus lancé en arrière-plan
  Utile pour récupérer le PID après une commande avec &
- `$?` Code de retour de la dernière commande exécutée
  0 = succès, autre valeur = erreur

Concepts fondamentaux

PID Process ID - Identifiant unique de chaque processus
Nombre entier assigné séquentiellement à la création

PPID Parent Process ID - Identifiant du processus parent
Tous les processus ont un parent, sauf init/systemd (PID 1)

Daemon Processus qui démarre au démarrage et s'exécute indéfiniment
Exemples : `sshd`, `cron`, `rsyslogd`, `NetworkManager`

Zombie Processus terminé qui apparaît encore dans la liste (état Z)
En attente que le parent lise son code de retour

Kill Action d'envoyer un signal à un processus
Ne signifie pas toujours terminer, peut être SIGHUP, SIGSTOP, etc.

Fork Création d'une copie identique d'un processus
Le processus enfant hérite de l'environnement du parent

Exec Remplacement d'un processus par un autre programme
Le nouveau programme garde le même PID

La commande ps

```
Affiche le PID de votre shell actuel
$ echo $$
```

```
Affiche le PID du processus parent
$ echo $PPID
```

```
Affiche les détails de votre shell actuel
$ ps -p $$ -o pid,ppid,comm
```

```
Affiche tous les processus en format détaillé
$ ps aux
```

La commande ps : affichage en arbre

```
Affiche l'arbre hiérarchique des processus
```

```
$ ps axjf
```

```
Affiche les informations sur le processus PID 1 (systemd)
```

```
$ ps -p 1 -o pid,ppid,cmd
```

Rechercher des processus : pidof

```
Trouve le PID du processus bash
```

```
$ pidof bash
```

```
Trouve le PID du processus systemd
```

```
$ pidof systemd
```

```
Trouve le PID du processus sshd (si actif)
```

```
$ pidof sshd
```

Rechercher des processus : pgrep

```
Trouve le PID des processus bash
```

```
$ pgrep bash
```

```
Affiche le PID et le nom du processus
```

```
$ pgrep -l bash
```

```
Affiche tous vos processus
```

```
$ pgrep -u $USER
```

Surveillance temps réel : top

```
Lance top pour surveiller les processus en temps réel
```

```
$ top
```

Touches interactives dans top :

- `M` : Trier par utilisation mémoire
- `P` : Trier par utilisation CPU
- `k` : Tuer un processus (entrez le PID)
- `q` : Quitter top

```
Affiche uniquement vos processus dans top
```

```
$ top -u $USER
```

Créer des processus en arrière-plan

```
Lance un processus sleep de 300 secondes en arrière-plan
$ sleep 300 &
```

```
Affiche le PID du dernier processus lancé en arrière-plan
$ echo $!
```

```
Liste les jobs en cours dans le shell actuel
$ jobs
```

```
Affiche les détails des processus sleep
$ ps aux | grep sleep
```

Gérer plusieurs processus

Lance trois processus sleep différents

```
$ sleep 100 &
sleep 200 &
sleep 300 &
```

Liste tous les jobs actifs

```
$ jobs
```

Trouve tous les PID des processus sleep

```
$ pgrep -l sleep
```

Suspendre un processus avec Ctrl+Z

```
Lance un processus sleep au premier plan
$ sleep 1000
```

Appuyez sur Ctrl+Z pour suspendre le processus

```
Vérifie l'état du processus suspendu
$ jobs
```

```
Affiche l'état STAT du processus (T = stopped)
$ ps aux | grep sleep
```

Reprendre un processus : bg et fg

```
Reprend le job numéro 1 en arrière-plan
$ bg 1
```

```
Vérifie que le job est maintenant Running
$ jobs
```

```
Ramène le job 1 au premier plan
$ fg 1
```

Appuyez sur Ctrl+C pour arrêter le processus

Tuer un processus : kill

```
Lance un processus sleep en arrière-plan
$ sleep 600 &
```

```
Note le PID affiché, puis tue le processus (SIGTERM)
$ kill <PID>
```

```
Vérifie que le processus est terminé
$ jobs
```

```
Lance un nouveau sleep et tue-le avec force (SIGKILL)
$ sleep 700 &
kill -9 <PID>
```

Lister les signaux disponibles

```
Affiche tous les signaux disponibles
$ kill -l
```

Signaux principaux :

- `1 (SIGHUP)` : Rechargement configuration
- `2 (SIGINT)` : Interruption (Ctrl+C)
- `9 (SIGKILL)` : Terminaison forcée
- `15 (SIGTERM)` : Terminaison propre (défaut)
- `19 (SIGSTOP)` : Suspension

Tuer par nom : pkill

Lance trois processus sleep

```
$ sleep 100 &
sleep 200 &
sleep 300 &
```

Vérifie qu'ils sont actifs

```
$ pgrep -l sleep
```

Tue tous les processus sleep

```
$ pkill sleep
```

Vérifie qu'ils sont terminés

```
$ jobs
```

Tuer par nom : killall

Lance deux processus sleep

```
$ sleep 400 &
sleep 500 &
```

Liste les processus sleep

```
$ pgrep -l sleep
```

Tue tous les sleep avec SIGKILL

```
$ killall -9 sleep
```

Vérifie qu'ils sont terminés

```
$ pgrep sleep
```

Commandes nice et renice

Qu'est-ce que le niceness ?

- Valeur numérique associée à chaque processus Linux
- Influence la priorité d'ordonnancement du processeur
- Plage de valeurs : de -20 à +19
- **Plus le niceness est élevé**, plus le processus est "gentil" envers les autres (priorité basse)
- **Plus le niceness est bas** (valeurs négatives), plus le processus obtient de temps CPU (priorité haute)
- Valeur par défaut : 0 (priorité neutre, héritée du processus parent)

Impact sur l'ordonnancement

- L'ordonnanceur utilise le niceness pour calculer la priorité réelle du processus
- Détermine la répartition du temps processeur entre processus concurrents
- Utile quand plusieurs processus demandent plus de CPU que disponible

Commandes nice et renice (suite)

Commande nice

- Lance un nouveau processus avec une valeur de niceness spécifique
- Syntaxe : `nice -n <valeur> <commande>`
- Permet de démarrer un programme avec une priorité modifiée

Commande renice

- Modifie le niceness d'un processus déjà en cours d'exécution
- Syntaxe : `renice -n <valeur> -p <PID>`
- Nécessite le PID (Process ID) du processus à modifier
- Permissions root requises pour diminuer le niceness (augmenter la priorité)

Remarque importante

Seul le super-utilisateur (root) peut attribuer des valeurs de niceness négatives

Priorité des processus : nice

```
Lance un sleep avec priorité normale
$ sleep 1000 &
[1] 5234
```

```
Affiche sa priorité (NI = 0)
$ ps -p 5234 -o pid,ni,comm
```

```
Lance un sleep avec priorité basse (nice = 10)
$ nice -n 10 sleep 2000 &
[2] 5467
```

```
Compare les deux processus
$ ps -o pid,ni,comm
```

```
Nettoie les processus
$ pkill sleep
```

Modifier la priorité : renice

```
Lance un processus sleep
$ sleep 3000 &
[1] 6789
```

```
Vérifie la priorité initiale (NI = 0)
$ ps -p 6789 -o pid,ni,comm
```

```
Modifie la priorité à 15
$ renice +15 6789
```

```
Vérifie le changement de priorité (NI = 15)
$ ps -p 6789 -o pid,ni,comm
```

```
Nettoie
$ kill 6789
```

Arbre des processus

```
Affiche l'arbre complet des processus
$ ps axjf | less
```

```
Affiche uniquement les enfants de systemd (PPID=1)
$ ps -ppid 1 -o pid,comm | head
```

```
Affiche le parent de votre shell
$ ps -p $PPID -o pid,ppid,comm
```

Alternative : pstree

```
Affiche l'arbre depuis votre shell (si pstree installé)
$ pstree -p $$
```

```
Affiche l'arbre complet depuis systemd
$ pstree -p 1 | less
```

```
Affiche l'arbre avec les PID
$ pstree -np
```

Récapitulatif des commandes

ps	Liste les processus
top	Surveillance temps réel interactive
pidof	Trouve le PID par nom de processus
pgrep	Recherche avancée de processus
jobs	Liste les tâches du shell
bg	Reprend un job en arrière-plan
fg	Ramène un job au premier plan
kill	Envoie un signal à un processus
pkill	Tue des processus par nom
killall	Tue tous les processus d'un nom
nice	Lance avec priorité modifiée
renice	Modifie la priorité d'un processus