Cours Synthétique : Systèmes d’Exploitation et Linux

Ce document regroupe et synthétise les notions essentielles issues des chapitres **2, 4, 5, 6** ainsi que du fichier **cours-tp-linux-1A**.

Chapitre 1 : Rappels sur Linux et UNIX

1.1 Historique

- UNIX (1969, Bell Labs), réécrit en C par Dennis Ritchie → portable.
- Deux grandes familles : **System V** (AT&T;) et **BSD** (Berkeley).
- Linux (1991, Linus Torvalds) + GNU → système complet et open-source.

1.2 Caractéristiques de Linux/UNIX

- Multi-utilisateurs, multi-tâches, multi-processeurs.
- Gestion mémoire protégée et segmentation par processus.
- Communication entre processus : IPC (pipes, mémoire partagée, sémaphores, sockets).
- Gestion des périphériques par le principe **« tout est fichier »**.

1.3 Organisation du système de fichiers

- Arborescence : `/` racine.
- Répertoires importants : `/bin`, `/usr/bin`, `/etc`, `/dev`, `/tmp`, `/home`, `/var`.
- Types de fichiers : ordinaires, répertoires, liens (simples et symboliques), fichiers spéciaux (caractère, bloc).

1.4 Processus

- Un **programme** est passif (fichier). Un **processus** est actif (instance en exécution).
- Identifiés par : PID (Process ID), PPID (Parent Process ID).
- Commandes utiles : `ps`, `top`, `kill`, `jobs`, `fg`, `bg`, `nice`, `renice`.
- États possibles : actif (R), endormi (S), stoppé (T), zombie (Z).

Chapitre 2 : Création et Ordonnancement des Processus (ch2)

2.1 Structure mémoire d’un processus

- **Texte** (code du programme).
- **Données** (variables globales, heap).
- **Pile** (appels de fonctions, variables locales).

2.2 Ordonnancement

- Objectif : optimiser l’utilisation CPU (40–95 %).
- Notions : débit, temps de réponse, temps moyen d’exécution.
- États d’un processus : nouveau, prêt, élu, bloqué, suspendu, zombie.
- Algorithme : **Round Robin (tourniquet)** basé sur un quantum de temps.

2.3 Création de processus : `fork()`

- Duplique le processus courant (père $\rightarrow$ fils).
- Valeur de retour : `0` pour le fils, `PID` du fils pour le père, `-1` si erreur.
- Fils hérite du code, données copiées, descripteurs de fichiers.

2.4 Synchronisation : `wait()` et `waitpid()`

- `wait()` : père bloqué jusqu’à terminaison d’un fils, récupère son état.
- `waitpid(pid, …)` : permet d’attendre un fils spécifique, options bloquantes ou non.
- Permet d’éviter les **zombies**.

2.5 Recouvrement : `exec()`

- Charge un nouveau programme à la place du processus courant.
- Famille : `execl`, `execv`, `execvp`, `execve`.
- Pas de retour si succès.

2.6 Visualisation

- `ps` : affiche PID, PPID, UID, état (R/S/T/Z), etc.
- Exemple : un fils terminé mais non récupéré devient `` (zombie).

Chapitre 3 : Communication par Signaux (ch4)

3.1 Définition

- Un signal est un **message court** envoyé à un processus pour notifier un événement.
- Exemples : interruption clavier (Ctrl+C), division par zéro, terminaison.

3.2 Principaux signaux

- `SIGINT (2)` : interruption clavier.
- `SIGQUIT (3)` : interruption avec core dump.
- `SIGKILL (9)` : tue immédiatement, non interceptable.
- `SIGTERM (15)` : terminaison propre.
- `SIGSTOP (19)`, `SIGCONT (18)` : suspension / reprise.
- `SIGCHLD` : mort d'un fils.

3.3 Gestion des signaux

- Par défaut : ignorer, terminer, ou générer un core dump.
- Possibilité de définir un **handler** :

```c
void handler(int sig) {
printf("Signal reçu : %d", sig);
}
signal(SIGUSR1, handler);
```

3.4 Commandes associées

- `kill -9 pid` → envoie SIGKILL au processus.
- `kill -l` → liste les signaux.
- `trap` (shell) → associer traitement à un signal.

Chapitre 4 : Échange de Données – Tubes et Verrous (ch5)

4.1 Les tubes (pipes)

- Un **tube** est un canal unidirectionnel (FIFO).
- Permet communication entre processus apparentés.
- Ex. : `ls | wc -l`.

- Création en C :

```c
int p[2];
pipe(p);
```

- `p[0]` : lecture, `p[1]` : écriture.

4.2 Tubes nommés (FIFO)

- Créés avec `mkfifo` ou `mkfifo()` en C.
- Permettent communication entre processus **sans lien de parenté**.
- Bloquants tant qu'aucune extrémité opposée n'est ouverte.

4.3 Verrous de fichiers

- Utilisés pour contrôler l'accès concurrent à un fichier.
- Trois interfaces : `flock()`, `fcntl()`, `lockf()`.
- Types : **partagé (lecture)**, **exclusif (écriture)**.
- Mode consultatif ou impératif.
- Risque de **deadlock** si mauvaise gestion.

Chapitre 5 : Communication Inter-Processus (IPC) (ch6)

5.1 Mécanismes IPC System V

- **Mémoire partagée** (rapide, pas de recopie).
- **Files de messages** (boîtes aux lettres typées).
- **Sémaphores** (synchronisation).

5.2 Gestion des clés

- Identificateur interne (ID) + clé externe (via `ftok`).
- Commandes utiles : `ipcs`, `ipcrm`.

5.3 Mémoire partagée

- Création : `shmget(key, size, flags)`.
- Attachement : `shmat(shmid, addr, flags)`.

- Détachement : `shmdt(addr)`.
- Suppression : `shmctl(shmid, IPC_RMID, …)`.

5.4 Files de messages

- Création : `msgget(key, flags)`.
- Envoi : `msgsnd(msqid, msg, size, flags)`.
- Réception : `msgrcv(msqid, msg, size, type, flags)`.
- Messages typés (choix de lecture possible).

5.5 Sémaphores

- Variable entière contrôlant accès à ressource.
- Opérations de Dijkstra :
- **P(S)** : décrémente, bloque si `S <= 0`.
- **V(S)** : incrémente, réveille un processus en attente.
- Utilisation : `semget`, `semop`, `semctl`.
- Permet de gérer sections critiques et éviter conflits.

Conclusion

Avec ces 5 chapitres, on couvre les bases de la **programmation système sous UNIX/Linux** :

1. Compréhension du système de fichiers et du fonctionnement multi-tâches.
2. Création, ordonnancement et gestion de processus.
3. Utilisation des signaux pour la communication asynchrone.
4. Échange de données via tubes et verrous.
5. Communication avancée via IPC (mémoire partagée, files de messages, sémaphores).

■ Ce socle est indispensable pour aborder les TP/TD pratiques et écrire des programmes systèmes robustes en C sous Linux.