S4 2019-2020

Nathan TONNELLE

**Système d'exploitation**

INFO0403

Table des matières

[Introduction Système d'exploitation 3](#_Toc29202885)

[Définition d'un système d'exploitation 3](#_Toc29202886)

[Fonctionnement 3](#_Toc29202887)

[Rôle 3](#_Toc29202888)

[Exemple : disque 3](#_Toc29202889)

[Partage de ressource 3](#_Toc29202890)

[Exemple : 3](#_Toc29202891)

[Historique 3](#_Toc29202892)

[Les processus 5](#_Toc29202893)

[Définition 5](#_Toc29202894)

[Plusieurs 5](#_Toc29202895)

[La concurrence : 5](#_Toc29202896)

[Programmation système 6](#_Toc29202897)

[Exemple : 6](#_Toc29202898)

[Quelques appels system 6](#_Toc29202899)

[Les processus 7](#_Toc29202900)

[Programme : 7](#_Toc29202901)

[Processus : 7](#_Toc29202902)

[La création de processus survient lorsque : 7](#_Toc29202903)

[Identification 7](#_Toc29202904)

[Création : 7](#_Toc29202905)

[pid\_t fork(): 8](#_Toc29202906)

[Etats des processus 8](#_Toc29202907)

[Ordonnancement 8](#_Toc29202908)

[Motivations : 8](#_Toc29202909)

[Equité en temps : 8](#_Toc29202910)

[Définition : 8](#_Toc29202911)

[Type d'ordonnanceur 9](#_Toc29202912)

[Non préemptif 9](#_Toc29202913)

[Préemptif 9](#_Toc29202914)

[Non préemptif 9](#_Toc29202915)

[Préemptif 9](#_Toc29202916)

[Exécution d'un programme 9](#_Toc29202917)

[Programmation Système 9](#_Toc29202918)

[Fonction principale 10](#_Toc29202919)

[Schéma mémoire d'un processeur 10](#_Toc29202920)

[La mémoire d'un processus est composée de : 10](#_Toc29202921)

[Allocation dynamique 10](#_Toc29202922)

[Récupérer et modifier les variables d'environnement 11](#_Toc29202923)

[Exemple de variables d'environnement : 11](#_Toc29202924)

[Exercice : 11](#_Toc29202925)

[Limitation ressources 11](#_Toc29202926)

[Exemple : 11](#_Toc29202927)

[Identifier un processus 12](#_Toc29202928)

[Création de processus 12](#_Toc29202929)

[Exemple : 12](#_Toc29202930)

[Terminaison de processus 12](#_Toc29202931)

[Exemple : 13](#_Toc29202932)

[Chargement d'un programme 13](#_Toc29202933)

[La gestion Mémoire 13](#_Toc29202934)

[Quand la mémoire est insuffisante 13](#_Toc29202935)

[Pour maintenir tous les processus actifs 13](#_Toc29202936)

[2 approches 13](#_Toc29202937)

[Le va et vient 13](#_Toc29202938)

[Gestion mémoire (va et vient) 13](#_Toc29202939)

[Gestion par liste chainée 14](#_Toc29202940)

# Introduction Système d'exploitation

## Définition d'un système d'exploitation

* Interface entre un utilisateur et une machine physique
* Dédié à la gestion des périphériques

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SGBD | Appli de calcul scientifique | … | Traitement de texte | **Application** |
| Interpréteur commande | Compilateur | … | Edition de fichier | **Programme Système** |
| Système d'exploitation | | | |
| Langage Machine | | | | **Matériel** |
| Micro Architecture | | | |
| Périphérique physique | | | |

## Fonctionnement

* Portion de code en mode noyau
* Protégé des modifications apportées par un utilisateur

## Rôle

* Gestion de ressources
* Il offre une vision étendue de la machine

### Exemple : disque

* Vision "haut niveau" : Système de fichiers
* Vision "bas niveau" : surface d'enregistrement tête de lecture, moteur, blocs
* Gestion de ressources :
  + Permettre le partage d'accès
  + Solution pour partager sera dépendante des ressources
  + Equitable
  + Récupérer la connaissance des accès concurrents sur une ressource

### Partage de ressource

Peut se faire sur 2 dimensions :

* + Le temps
  + L'espace

#### Exemple :

* Mémoire : Espace
* Processeur : Temps
* Un ensemble de Processeurs : Temps et Espace

## Historique

* Machine de Charle Babbage (~1800)

Non fonctionnel : Mécanique pas suffisamment précise

* 1ère génération 1945-1955
  + Tube à vide et de tableau d'interrupteur
    - 1950 : introduction de la carte perforées
* 2eme génération 1955-1965
  + Introduction du transistor
    - Accroissement de la fiabilité
    - Introduction du traitement par lots
  + Soumission d'un travail en 4 étapes :
    - Ecriture du programme
    - Soumission à l'opération
    - Introduction des cartes perforées
    - Compilation en fortran
  + Traitement par lots (actuellement : script)
* 3eme génération 1965-1980
  + 2 architectures incompatibles en termes de matériel (unité de données : caractère/mot)
  + IBM : introduction System 360 (-> compatibilité ai niveau logiciel)
  + Introduction Multiprogrammation
  + Introduction Spodage
  + Introduction Temps partagé
  + Multitâche (simple/préemptif)
  + Début d'Unix (-> System V, -> BSD)
* 1980-?? (2010) Ordinateurs personnels
  + Intel 8080
  + MSDOS (Bil Gates Tim Paterson)
  + Lisa, Mac Intosh (Steve Jobs (-> IHM))
  + Windows (d'abord comme surcouche à MSDOS puis comme SE à part entière)
  + Développement UNIX (->Linux)
  + Système d'exploitation Distribué
* 2010-today
  + Dématérialisation
  + Big-Data

# Les processus

## Définition

* Programme en cours d'exécution
* Espace d'adressage propre :
  + Code du programme exécutable
  + Données
  + Ensemble de registres
* Table de processus
  + Un processeur est lié à un autre processus par une relation de parenté
    - Arborescence de processus avec un ancêtre commun : 'int'

## Plusieurs

Processus en concurrence pour l'accès à une ressource :

## La concurrence :

Résoudre le problème de concurrence entre plusieurs processus c'est :

* Sureté de l'exécution de la portion de code durant laquelle le processus accède à la ressource s'appelle la section critique. A un moment donné, au plus un processus exécute en section critique
* Vivacité : un processus qui demande à accéder à sa section critique finit par pouvoir l'exécuter
  + Absence d'interblocage

# Programmation système

## Exemple :

cpt =read(df,&tampon, nboctet);



Retour appelant

Déroulement Noyau

5 Place de code de read() en mémoire



11 Incrémentation CO

4 Appel read

3 Empiler nombre octets

2 Empiler tampon

1 Empiler df



Noyau du SE

Colle de l'appel Sys

Branchement



## Quelques appels system

* Gestion de processus fork(), waitpid(), execve(), …)
* Gestion de fichiers open(), close(), read(), write(), stat(),…)
* Gestion du système de fichier mkdir(), rmdir(), chdir(), link(), unlink(),…)

# Les processus

## Programme :

Suite d'instructions

## Processus :

Programme en cours d'exécution.

Sur un processeur, un utilisateur non averti à l'impression que plusieurs processus s'exécutent en même temps.

La simultanéité résulte du fait que des lots de temps suffisamment courts sont alloués à chaque processus.

On a donc cette idée de "continuité".

La simultanéité réelle provient du fait qu'on a plusieurs processeurs.

* Cette "presque simultanéité" s'appelle la multiprogrammation

### La création de processus survient lorsque :

* Requête utilisateur
* Traitement par lot
* Initialisation du système
* Utilisation de l'appel système fork()

### Identification

* pid (cf commande ps)

Identifier l'utilisateur d'un processus

UID (réel/effectif/sauvé)

Identifier le groupe de l'utilisateur du processus

GID (réel/effectif/sauvé)

### Création :

Les processus sont créés par l'appel system fork() et se terminent :

* De manière normale
* Avec une erreur
* Avec une erreur fatale
* A cause d'un autre processus (via des signaux)

Premier processus créé : int

* Création de processus à l'initiative d'autres processus
* Relation de parenté entre processus (-> Arborescence)
* Visualisation des processus (ps, top, pstree)
* Suppression de processus (kill -9)

Appel system fork() :

pid=342

fork();

pid=??

pid=342

a==0

a==0

a==0

pid: 300

pid: 347

int main(int argc, char \*argv[])

A==0

B==0

A==0

B==0

{

int a=0;

int b=0;

fork();

printf("a %d, b %d\n",a,b);

exit(0);

}

## pid\_t fork():

Duplique l'espace mémoire du courant et crée un fil d'exécution dont l'exécution démarre à l'instant où le processus s'est dupliqué.

La valeur retournée (de type pid\_t) est l'identifiant du processus créé dans le processus créant (ou père) et 0 dans le processus créé.

int main(int argc, char \*argv[])

{

int a=0;

int b=0;

if(fork()==0)

{

a=1;

b=432;

exit(0);

}

printf("a %d, b %d \n",a,b);

exit(0);

}

## Etats des processus

Sommeil

fork()

Evénement

Interruption



Création



Zombie

Exécution



FIN



SIGCONT

SIGSTOP



Arrêt

## Ordonnancement

### Motivations :

#### Equité en temps :

* Critère pour un "bon" ordonnancement
* Pourcentage utilisation CPU
* Nombre de processus exécutés en un temps donnée
* Temps nécessaire pour exécuter un processus
* Temps d'attente d'un processus dans la file prêt

### Définition :

L'ordonnanceur est un algorithme qui élit un processus. Ce processus a le privilège d'accéder au CPU.

## Type d'ordonnanceur

### Non préemptif

Sélectionne un processus qui s'exécute jusqu'à ce qu'il bloque ou se termine

### Préemptif

Sélectionne un processus qui s'exécute pendant un délai donné

Si le processus est toujours en cours d'exécution, il est suspendu (Etat sommeil) et un autre processus est alors choisi.

### Non préemptif

Ordonnancement

* FIFO : exécution des processus dans l'ordre de soumission
* SJF : exécution des processus les plus courts d'abord
* LJF : exécution des processus les plus longs d'abord

### Préemptif

Ordonnancement :

* SRTN : au plus court reste d'abord
* Roud Robin : on exécute chaque processus durant un laps de temps donné en suivant l'ordre de soumission
* Priorité : on utilise Round Robin par file de priorité

## Exécution d'un programme

* On utilise la famille d'appels Système exec :

Ces appels système chargent en mémoire du processus le contenu d'un programme stocké sue Disque.

int execv (const char \* app, const char \* argv[])

App : chemin complet du programme

Argv : tableau des différents arguments

int execl (const char \* app, const char \* arg1, const char \* arg2, …)

App : chemin complet de l'application

Arg1 ,…,n : arguments à passer. La liste se termine par NULL

Exercice 1 :

Ecrire un programme qui exécute "ls-al"

int main()

{

char \* app="/bin/ls";

char \* arg[]={"ls","-al"};

execv(app,arg);

exit(0);

}

Exercice 2 :

Ecrire un programme qui exécute "ps" puis "ls-al"

int main(){

{

char \* app="/bin/ls";

char \* arg[]={"ls","-al"};

if(fork()==0){

execv(app,arg);

exit(0);

}

}

## Programmation Système

Langage C

## Fonction principale

Int main(int argc, char \* argv[])

argc : nombre de paramètres à la fonction

argv : tableau de chaine de caractères des paramètres

$ ./programme toto 423 arg1

Int t=512;

Char \* s=malloc(…)

sprintf(s,"%d",t);

argc =4

argv[0]="./programme"

argv[1]="toto"

argv[2]="423"

argv[3]="arg1"

## Schéma mémoire d'un processeur

### La mémoire d'un processus est composée de :

* Le segment texte : ce sont les différentes instructions exécutées par le processeur. Le segment est en lecture seule et peut être éventuellement partagé.
* Segment de données initialisées on y retrouve les données initialisées

Int max=100;

* Segment de données non initialisées, elles sont normalement initialisées à 0 ou à NULL

Int sum[100];

* Pile : sert à la sauvegarde d'informations en particulier lors de l'appel d'une fonction.
* Tas : sert à l'allocation dynamique de mémoire
* Arguments et les variables d'environnement

La commande size fournit la taille des différents segments

# Allocation dynamique

* Void \* malloc (size\_t T)

Reserve un emplacement de taille T dont l'adresse du premier octet est retourné.

Void \* ptr;

ptr = malloc (10);

* Void \* calloc (int n, size\_t T)

Fonctionne comme malloc, reserve de la place dans le tas pour contenir n objets de taille T, soit n x T octets

* Void \* realloc (void \* ptr, size\_t T)

Change la taille de la zone de mémoire précédemment allouée. La nouvelle taille est T.

* Void free(void \* p);

Désalloue la zone mémoire dont la première case est située à l'adresse p

# Récupérer et modifier les variables d'environnement

## Exemple de variables d'environnement :

* HOME : chaine de caractère contenant le répertoire de travail de l'utilisation
* LOGNAME : nom de login d'utilisateur
* PATH : ensemble de chemin désignant les répertoires contenant les exécutables
* TERM : type de terminal

1. Char \* getenv(const char \* name);
2. Int putenv(const char \* name);
3. Int setenv(const char \* name, const char \* value, int valur);
4. Void unsetenv(cast char \* name);
5. Récupérer la valeur de la var. d'env name
6. Déclarer une variable d'env name
7. Positionner la variable d'env name à la valeur value
8. Retirer la var d'env name

## Exercice :

Créer une variable d'environnement TRUC. Positionnez sa valeur à "c'est les vacances". Afficher sa valeur, puis supprimer cette variable d'environnement.

Int main (int argc, char \* argv[])

{

putenv("TRUC");

setenv("TRUC","C'est les vacances");

printf("%s\n",getenv("TRUC"));

unsetenv("TRUC");

exit(0);

}

## Limitation ressources

### Exemple :

RLIMIT\_CORE taille max du fichier Core crée en cas d'erreur fatale

RLIMIT\_CPU maximum temps CPU

RLIMIT\_DATA taille max segment de données (init et non init)

RLIMIT\_FSIZE taille max des fichiers ouverts

RLIMIT\_NOFILE nombre max de fichiers ouvert

RLIMIT\_NPROC nombre max de processus fils

RLIMIT\_STACK taille max de la pile

…

struct rlimit{

rlim\_t rlim\_cur; //limite courante

rlim\_t rlim\_max; //maximum pour le système

}

Int getrlimit(int ressource, struct rlimit \* ptr);

Int setrlimit(int ressource, const struct rlimit \* ptr);

## Identifier un processus

pid\_t getpid(); retourne l'id du processus

pid\_t getppid(); retourne l'id du processus père

pid\_t getuid(); retourne l'id de l'utilisateur

pid\_t geteuid(); retourne l'id de l'utilisateur effectif

pid\_t getgid(); retourne l'id du groupe

pid\_t getegid(); retourne l'id du groupe effectif

## Création de processus

pid\_t fork(); créer un processus fils. La valeur de 0 dans le processus fils

Retour est : le pid du processus crée dans le processus père

### Exemple :

int gbb=6;

int main()

{

int var;

pid\_t pid;

var =88;

printf("avant fork\n");

if((pid=fork())==0)

{

glob++;

var++;

printf("%d,%d\n",glob, var);

exit(0);

}

sleep(2);

printf("%d,%d\n", glob, var);

exit(0);

}

## Terminaison de processus

* Void exit(int s);

Effectue un "ménage" et quitte le processus avec le statut s

* Void \_ext(int s);

Quitte le processus et retourne au système avec le statut s

* Int atexit(void (\*fonc) (void));

Spécifie les opérations à réaliser à la terminaison du processus **(voir ce qu'est qu'un pointeur de fonction)**

Pid\_t wait(int \* stat)

Attend la fin d'un processus fils. La variable stat contient la valeur du statut de terminaison SD1 du processus fils. L'appel à wait est bloquant.

### Exemple :

int gbb=6;

int main()

{

int var;

pid\_t pid;

var =88;

printf("avant fork\n");

if((pid=fork())==0)

{

glob++;

var++;

printf("%d,%d\n",glob, var);

exit(0);

}

Wait(NULL);

printf("%d,%d\n", glob, var);

exit(0);

}

## Chargement d'un programme

* Famille de fonction exec (cf cours plus haut)

## La gestion Mémoire

* Monogrammation

Schéma 1

* Multiprogrammation

Schéma 2

### Quand la mémoire est insuffisante

#### Pour maintenir tous les processus actifs

* Sauvegarde des processus sur disque
* Chargement dynamique

#### 2 approches

Le va et vient :

Chaque processus est considéré dans son intégralité

Exécution et sauvegarde sur disque

La mémoire virtuelle :

Un processus est découpé en page

Il peut donc s'exécuter meme s'il n'est que partiellement en mémoire

#### Le va et vient

* Chargement complet du processus
* Sauvegarde complète éventuelle en cas de manque de mémoire
* Au chargement :

Nécessité de localiser correctement le processus

* Possibilité d'apparition de "trous" mémoire 🡪 Technique de compactage (couteux !)

Schéma 3

Quand la taille des processus est fixe, le SE alloue exactement la place Mémoire nécessaire

Quand un processus doit augmenter sa taille :

* Soit il y a de l'espace libre contiguë à l'espace mémoire du processus (pas de soucis !)
* Il est contigu à d'autres processus : il faut alors déplacer le processus
* S'il n'y a pas suffisamment de place : attente ou terminaison

#### Gestion mémoire (va et vient)

* Tableau de bits
* La mémoire est répartie en unités d'allocation (UA)
* Chaque UA pssède une taille fixe comprise entre quelques octets et plusieurs ko
* A chaque UA correspond un bit dans le tableau

0 : vide 1 : sinon

* Problème : trouver la bonne taille !

Schéma 4

### Gestion par liste chainée

Liste chainée des segments mémoire

Chaque entrée indique :

* L'état du segment (L: libre, 0 : occupé)
* L'adresse à laquelle il débute
* Sa longueur

Schema 5

### Gestion mémoire (va et vient)

Où placer les processeurs ?

Dans quel espace libre ?

Peut on optimiser la taille des espaces libres ?

Optimiser la recherche d'un espace libre ?

#### Plusieurs stratégies

##### First fit :

On parcourt la mémoire depuis le début pour trouver le premier emplacement qui suffit en termes de place

##### Next fit :

On parcourt la mémoire depuis la dernière position pour trouver le premier emplacement qui suffit en termes de place

##### Best fit :

On parcourt l'espace mémoire en cherchant l'espace mémoire dont la taille s'approche le plus de la taille du processus

##### Wost fit :

On parcourt l'espace mémoire pour trouver le plus grand espace libre, si la taille est suffisante, on place le processus "on multiplie "

### La mémoire virtuelle

* La taille d'un programme (instruction, données, pile, …) peut etre supéroeure à la capacité mémoire physique
* Le système d'exploitation conserve en mémoire des parties d'un programme en cours d"xécution le reste est stocké sur disque

### Technique utilisé : la pagination

Schéma 6

Un ensemble de processus est découpé en "pages" mémoire

On doit gerer les échanges de pages entre mémoire rapide et mémoire lente

Pour fonctionner, le système execute les instructions d'un procesus si une partie de ses pages sont en mémoire rapide.

#### Le défaut de page

C'est un accès à une page absente de la mémoire rapide.

Dans ce cas : le SE choisit une page présente en mémoire rapide, l'ecrit sur la mémoire lente, et charge la page qui faisait défaut à sa place.