S4 2019-2020

Nathan TONNELLE

**Système d'exploitation**

INFO0403

Table des matières

[Introduction Système d'exploitation 3](#_Toc27748805)

[Définition d'un système d'exploitation 3](#_Toc27748806)

[Fonctionnement 3](#_Toc27748807)

[Rôle 3](#_Toc27748808)

[Exemple : disque 3](#_Toc27748809)

[Partage de ressource 3](#_Toc27748810)

[Exemple : 3](#_Toc27748811)

[Historique 3](#_Toc27748812)

[Les processus 5](#_Toc27748813)

[Définition 5](#_Toc27748814)

[Plusieurs 5](#_Toc27748815)

[La concurrence : 5](#_Toc27748816)

[Programmation système 6](#_Toc27748817)

[Exemple : 6](#_Toc27748818)

[Quelques appels system 6](#_Toc27748819)

[Les processus 7](#_Toc27748820)

[Programme : 7](#_Toc27748821)

[Processus : 7](#_Toc27748822)

[La création de processus survient lorsque : 7](#_Toc27748823)

[Identification 7](#_Toc27748824)

[Création : 7](#_Toc27748825)

[pid\_t fork(): 8](#_Toc27748826)

[Etats des processus 8](#_Toc27748827)

[Ordonnancement 8](#_Toc27748828)

[Motivations : 8](#_Toc27748829)

[Equité en temps : 8](#_Toc27748830)

[Définition : 8](#_Toc27748831)

[Type d'ordonnanceur 9](#_Toc27748832)

[Non préemptif 9](#_Toc27748833)

[Préemptif 9](#_Toc27748834)

[Non préemptif 9](#_Toc27748835)

[Préemptif 9](#_Toc27748836)

[Exécution d'un programme 9](#_Toc27748837)

[Programmation Système 9](#_Toc27748838)

[Fonction principale 10](#_Toc27748839)

[Schéma mémoire d'un processeur 10](#_Toc27748840)

[La mémoire d'un processus est composée de : 10](#_Toc27748841)

[Allocation dynamique 10](#_Toc27748842)

[Récupérer et modifier les variables d'environnement 11](#_Toc27748843)

[Exemple de variables d'environnement : 11](#_Toc27748844)

[Exercice : 11](#_Toc27748845)

[Limitation ressources 11](#_Toc27748846)

[Exemple : 11](#_Toc27748847)

[Identifier un processus 12](#_Toc27748848)

[Création de processus 12](#_Toc27748849)

[Exemple : 12](#_Toc27748850)

# Introduction Système d'exploitation

## Définition d'un système d'exploitation

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

Schéma 1

## 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);

Schéma 2

## 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

Schéma 5

Sommeil

fork()

Zombie

Exécution

Arret

## 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 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)