

Système d'exploitation

L2 Informatique – UVSQ

Sebastien GOUGEAUD

pro.seb.gougeaud@gmail.com

Avant-propos

Liste des topics abordés

1. Rappels C/Shell
2. Systèmes de fichiers
3. Processus
4. Threads
5. Communication
6. Mémoire
7. Ordonnancement
8. *HPC*

- 60% examen, 40% contrôle continu
- Examen :
 - traditionnel : séries d'exercices portant sur questions de cours, exécution d'algorithmes et écriture de code
- Contrôle continu : 60% TD, 40% CM
 - TD : 3 exercices de TD répartis sur le semestre, commencés durant la séance de TD et à rendre dans la semaine qui suit
 - CM : 1 QCM d'1h, corrigé dans la demi-heure de cours restante

Chaque groupe a sa salle :

- Gr.1 : G207, le mercredi à 9h40
- Gr.2 : D122, le mercredi à 9h40
- Gr.3 : G107, le mardi à 9h40 (+ DL BI)
- Gr.4 : Jungle, le mardi à 9h40

Les TDs notés se font en **binôme** intra-groupe, mais peuvent changer entre chaque TP

Rappels C

Pourquoi ce rappel ?

- Le C est le langage utilisé dans ce cours
- Pour vous rafraîchir la mémoire sur ce que vous savez déjà
- Pour vous introduire ce que vous ne connaissez pas encore et ce qui sera utilisé dans ce cours
- Le premier TD vous permet de vous refaire la main sur les mécanismes de base du C, il vous permettra de voir où sont vos lacunes pour pouvoir les combler

Liste des thèmes abordés

1. Variables
2. Blocs de contrôle
3. Fonctions
4. Structures
5. Pointeurs
6. Pointeurs de fonction
7. Tableaux
8. Directives de pré-compilation
9. 'Bien coder ?'

Variables

```
1 char *s = "Hello world!";  
2 double f = 42.01;  
3 int a = 7;  
4  
5 printf("%d - %f - '%s'\n",  
6       a, f, s);
```

- Déclaration sous la forme
type nom;
- Types basiques : int, float, double, char
- Modificateurs : unsigned, short, long

```
1  const int N = 100;
2  int tot = 0;
3  int i = 0;
4
5  if (N < 0) {
6      printf("Total = 0\n");
7  } else {
8      while (i != N) {
9          tot += i;
10         ++i;
11     }
12
13     printf("Total = %d\n", tot);
14 }
```

Structures conditionnelles :

- if .. else if .. else
- opérateur ternaire
test ? true : false
- switch

Structures itératives :

- while
- for
- do .. while

```
1 int _topic_3(const char *s,  
2             const char delim) {  
3     int res = 0;  
4     int i;  
5  
6     for (i = 0;  
7          i < strlen(s) &&  
8          s[i] != delim;  
9          ++i)  
10        ++res;  
11  
12     return res;  
13 }
```

- Déclaration sous la forme
type nom (parametres)
- Définition avec un bloc
d'instructions (entre
accolades)
- Utilisation de **return** pour
sortir de la fonction

Structures de données

```
1 struct cplx {  
2     double reel;  
3     double imgn;  
4 };  
5  
6 struct cplx r;  
7  
8 r.reel = a.reel + b.reel;  
9 r.imgn = a.imgn + b.imgn;  
10  
11 printf("(%.2f,%.2f)\n", r.reel, r.imgn);
```

- Structures, énumérations et unions
- Utilisation de **typedef** pour créer un nouveau type à partir d'un autre (renommage)
- Initialisation de la forme `c = {.5, 2};`

Variable contenant l'adresse d'une autre variable

```
1 void _topic_5(struct cplx a,  
2             struct cplx b,  
3             struct cplx *r) {  
4     if (r == NULL)  
5         return;  
6  
7     r->reel = a.reel + b.reel;  
8     r->imgn = a.imgn + b.imgn;  
9 }
```

- Si donné en paramètre de fonction, permet de modifier le contenu de la cible
- Adresse d'une variable
 $p = \&v;$
- Valeur de la cible d'un pointeur
 $v = *p;$
- Structure :
 $p \rightarrow f \leftrightarrow (*p).f$

Pointeurs de fonction

```
1 void dire_oui(const int id)
2     { printf("Oui %d\n", id); }
3 void dire_non(const int id)
4     { printf("Non %d\n", id); }
5
6 void topic_6() {
7     void (*f_pair)(const int) =
8         dire_oui;
9     void (*f_impr)(const int) =
10        dire_non;
11    int i;
12
13    for (i = 0; i < 10; ++i)
14        if (i % 2)
15            f_impr(i);
16        else
17            f_pair(i);
18 }
```

- Déclaration sous la forme
`type (*nom)(param);`
- Manipulation de la fonction en tant que variable (en paramètre de fonction, dans une structure ou dans un tableau)
- Utilisation pour faire de la spécification (entre autres)

```
1 void (*func[2])(const int) =  
2     { dire_oui, dire_non };  
3 int *tab;  
4 int i;  
5  
6 tab = malloc(10 * sizeof(*tab));  
7 for (i = 0; i < 10; ++i)  
8     tab[i] = i%2;  
9  
10 for (i = 0; i < 10; ++i)  
11     func[tab[i]](i);  
12  
13 free(tab);
```

- Tableaux fixes (*pile/stack*)
- Taille connue à la **compilation**
- Tableaux dynamiques (*tas/heap*)
- Taille définie à **l'exécution**
- Utilisation de `malloc()` et `free()`
- $T[i] \leftrightarrow *(T+i)$

Directives de pré-compilation

```
1  #define N 30
2      int tab[N];
3
4  #ifndef N
5      printf("N n'est pas défini\n");
6  #elif N < 1
7      printf("Pas d'elements\n");
8  #else
9      int i;
10
11     for (i = 0; i < N; ++i)
12         if (i)
13             tab[i] = i + tab[i - 1];
14         else
15             tab[i] = i;
16
17     printf("Allocation effectuee\n");
18 #endif
```

- Résolution des directives avant la compilation
- Création d'alias
- Ajout conditionnel d'instructions
- Autres...

Quelques bonnes pratiques de développeur

- Indenter
- Donner des noms de variables/fonctions clairs
- Éviter les fonctions trop longues (>50 lignes)
- Éviter les duplications de code
- **Commenter** le code, à minima les structures de données, les fonctions et les parties de code complexes

Rappels Shell

Qu'est-ce que le Shell ?

Interface utilisateur de base avec l'ordinateur – basée sur l'exécution de commandes données au clavier

```
$ ls ~
```

Deux types d'acteur utilisant le Shell :

- Utilisateur (**user**) – utilisation des programmes contenus dans `/*/bin`, manipulation des dossiers/fichiers utilisateur (`~`), etc.
- Administrateur (**sudoer**) – utilisation des programmes contenus dans `/*/sbin`, manipulation des dossiers/fichiers système, manipulation directe des périphériques, etc.

Redirection de flux – concept

- Redirection de l'entrée ou de la sortie standard par défaut vers un fichier ou un périphérique

```
$ ls ~ >fichier
```

```
$ sort <fichier1 >fichier2
```

- Utilisation de la sortie standard d'un processus comme entrée standard d'un autre processus

```
$ ls ~ | grep .txt
```

Redirection de flux – exemple

Fusion des deux types de redirection

```
$ cat fichier1 fichier2 | sort >/dev/lpa
```

^afichier vers un Kernel Printer Device

- Regroupement de commandes au sein d'un fichier
- Exécution du script comme s'il s'agissait d'un programme :

```
$ ./script.sh
```

- Affichage de la page de documentation correspondant à la commande entrée en argument

```
$ man ls
```

- Plusieurs catégories de documentation :
 1. Commandes Unix
 2. Appels systèmes
 3. Fonctions de la bibliothèque standard

- Compilation d'un (ou plusieurs) fichier(s) écrit(s) en langage C

```
$ gcc -o prog main.c
```

- Plusieurs options de compilation et *linkage* disponibles

Commande gdb

- Exécution en mode *debug* d'un programme

```
$ gdb ./prog
```

- Utilisation de commandes pour naviguer dans le programme durant l'exécution : *break*, *run*, *backtrace*, *up/down*, *print*, *next*, *continue*
- Informations de *debug* injectées dans le programme à l'aide d'une option de compilation :

```
$ gcc -g -o prog main.c
```

Introduction au système d'exploitation

Qu'est-ce qu'un système d'exploitation ?

→ Couche logicielle faisant le pont entre les applications et le matériel

Deux rôles :

- Masquer la complexité du matériel géré
- Gérer les ressources disponibles et les faire fonctionner ensemble de manière sûre et équitable

Exemples de ressources

- Processeur (CPU) et registres
- Mémoire (RAM, cache)
- Entrée/sortie (disque, imprimante, clavier, écran)
- Processus
- Autres

Systeme de fichiers

Pourquoi ?

→ Pouvoir stocker des informations de façon **pérenne**, de manière **organisée** et **abstraite**

Besoins :

- Stockage de grande quantité
- Conservation
- Accès simultané

Qu'est ce qu'un système de fichiers ?

→ Partie du système d'exploitation gérant les **fichiers**

Dans un système UNIX, tout est fichier :

- Fichiers
- Répertoires
- Liens symboliques
- Autres (FIFO, etc.)

Qu'est-ce qu'un fichier ?

→ Suite d'octets (binaire, ASCII, etc.) caractérisée par :

- Nom de fichier – label et extension (optionnel sous UNIX)

```
$ cat fichier-test.txt
```

- Chemin – emplacement dans la hiérarchie (absolu ou relatif)

```
$ ls /home/user1/doc
```

- Inode – noeud d'informations entre le système de fichiers et le périphérique
- Méta-données – attributs, ex : créateur, permissions d'accès, etc.

Appels systèmes – fichiers

```
1  int open(const char *pathname,  
2          int flags);  
3  int open(const char *pathname,  
4          int flags, mode_t mode);  
5  int close(int fd);  
6  int unlink(const char *pathname);  
7  
8  ssize_t read(int fd, void *buf,  
9             size_t count);  
10 ssize_t write(int fd, void *buf,  
11              size_t count);  
12  
13 off_t lseek(int fd, off_t offset,  
14             int whence);
```

- Création, ouverture, fermeture et suppression
- Lecture et écriture
- Positionnement

Fonction createFile()

```
1 void createFile(const char *fname, const int size, const char *data) {
2     int fd = -1;
3     ssize_t sz;
4     int rc;
5
6     fd = open(fname, O_CREAT | O_WRONLY, 0666);
7     if (fd == -1) {
8         fprintf(stderr, "ERR on file creation: %s\n", strerror(errno));
9         return;
10    }
11
12    if (size) {
13        sz = write(fd, data, size);
14        if (sz != size)
15            fprintf(stderr, "ERR on file writing: %s\n", strerror(errno));
16    }
17
18    rc = close(fd);
19    if (rc)
20        fprintf(stderr, "ERR on file closure: %s\n", strerror(errno));
21 }
```

Appels systèmes – répertoires

```
1 int mkdir(const char *pathname,  
2          mode_t mode);  
3 DIR *opendir(const char* name);  
4 int closedir(DIR *dirp);  
5 int rmdir(const char *pathname);  
6  
7 struct dirent *readdir(DIR *dirp);  
8  
9 void rewinddir(DIR *dirp);  
10 long telldir(DIR *dirp);  
11 void seekdir(DIR *dirp, long loc);
```

- Création, ouverture, fermeture et suppression
- Lecture
- Positionnement

Fonction createDir() 1/2

```
1  int createDir(const char *dname, const int nbEmptyFiles,
2              char ***dirFiles) {
3      int rc;
4      int i;
5
6      rc = mkdir(dname, 0700);
7      if (rc) {
8          fprintf(stderr, "ERR on dir creation: %s\n", strerror(errno));
9          return 1;
10     }
11
12     *dirFiles = malloc(nbEmptyFiles * sizeof(char **));
13     if (nbEmptyFiles && !*dirFiles) {
14         fprintf(stderr, "ERR on dir file names allocation: %s\n",
15                 strerror(errno));
16         return 0;
17     }
```

Fonction createDir() 2/2

```
1  for (i = 0; i < nbEmptyFiles; ++i) {
2      char *fname;
3      int fd;
4
5      fname = malloc(16 + strlen(dname));
6      if (!fname) {
7          fprintf(stderr, "ERR on file name allocation");
8          return 1;
9      }
10     snprintf(fname, 16 + strlen(dname), "%s/empty_XXXXXX", dname);
11     fd = mkstemp(fname);
12     if (fd == -1) {
13         fprintf(stderr, "ERR on file name creation (%s): %s\n",
14             fname, strerror(errno));
15         continue;
16     }
17     close(fd);
18     (*dirFiles)[i] = fname;
19 }
20 return 0;
21 }
```

Gestion d'erreur incomplète en l.8 et l.15, discutée en cours

Fonction deleteDir()

```
1 void deleteDir(const char *dname, const int nbEmptyFiles,  
2               char **dirFiles) {  
3     int i;  
4  
5     for (i = 0; i < nbEmptyFiles; ++i) {  
6         unlink(dirFiles[i]);  
7         free(dirFiles[i]);  
8     }  
9  
10    free(dirFiles);  
11    rmdir(dname);  
12 }
```

Mapping du fichier en mémoire

```
1 void *mmap(void *addr, size_t length, int prot, int flags, int fd,  
2         off_t offset);  
3 int munmap(void *addr, size_t length);
```

Permet de représenter le contenu d'un fichier dans un tableau

- Programmation facilitée
- Performances améliorées
- Cohérence à maintenir plus lourde (multi-accès)
- Taille du fichier généré, multiple de la taille d'une page

→ Une méta-donnée est un **attribut** d'un fichier

Liste non exhaustive :

- inode
- permissions
- UID
- taille
- date dernier accès
- périphérique
- nb liens matériels
- GID
- taille en blocs
- date dernière modification

Récupération des méta-données

```
1 int stat(const char *path, struct stat *buf);  
2 int fstat(int fd, struct stat *buf);  
3 int lstat(const char *path, struct stat *buf);
```

La structure `stat` est remplie lors de l'appel de la fonction, et contient diverses méta-données, dont :

- les permissions – `st_mode`
- le nombre de liens matériels pointant sur l'inode – `st_nlink`
- l'ID de l'utilisateur – `st_uid`
- la taille – `st_size`
- la date de dernier accès – `st_atime`

Permissions

- En Shell, modifiable avec la commande **chmod**
- En C, modifiable avec les appels systèmes **chmod()** et **fchmod()**

	U	G	O
read [4]	X	X	
write [2]	X	X	
exec [1]	X		
mode	7	6	0



U – user/utilisateur

G – groups/groupes

O – others/autres

- Exemple : **chmod(path, 0760)** → premier 0 indique que la valeur est écrite en base 8 (octale).

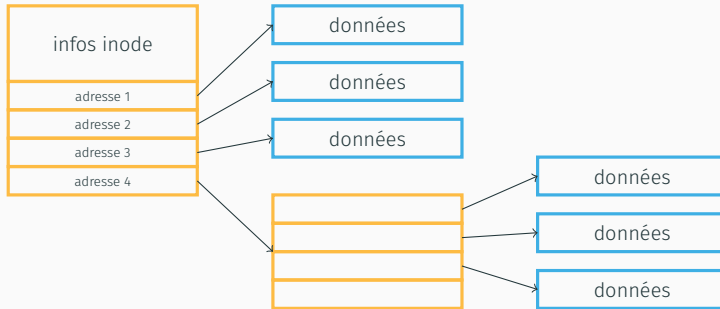
Liens

lien symbolique	lien matériel
alias/raccourci	nouveau nom à un fichier
<code>ln -s</code> <code>symlink()</code>	<code>ln</code> <code>link()</code>
	
invalide si <i>file</i> supprimé	valide si <i>file</i> supprimé

Inode

→ Bloc d'octets comportant :

- le numéro de l'inode
- les méta-données du fichier
- les liens vers les blocs de données du fichier



Les fichiers sont stockés dans des blocs de **taille fixe**

Quelle taille de bloc est la plus intéressante ?

Allocation de l'espace aux blocs – contiguë

État initial



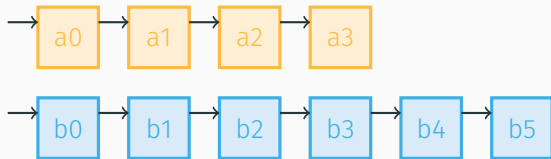
Suppression de 'a'



Avantages : simple à implémenter, bonnes performances

Inconvénients : fragmentation de l'espace, besoin de connaître la taille d'un fichier avant de le stocker

Allocation de l'espace aux blocs – liste chaînée



Avantage : pas de perte de place

Inconvénient : pour accéder au bloc n , il faut lire les blocs précédents.

Allocation de l'espace aux blocs – FAT

indice	suivant
0	-1
1	4
2	-1
3	-1
4	8
5	2
6	9
7	-1
8	6
9	-1

→ File Allocation Table, liste chaînée via table en mémoire

Avantage : liste chaînée sans son inconvénient

Inconvénient : la taille de la table augmente avec le nombre de blocs sur le disque

→ Il reste à stocker la liste des blocs libres :

- Liste chaînée ?
- Table de bits ?

Le système de fichiers peut présenter des incohérences si il tombe en panne lors d'une écriture de fichier :

- blocs libres présents plusieurs fois dans la liste
- blocs manquants (ni utilisés, ni libres)
- blocs utilisés dans plusieurs fichiers

La commande **fsck** permet de vérifier la cohérence du système de fichiers, mais ne garantit pas la préservation des données.

En utilisant les appels systèmes pour la gestion du système de fichiers, excepté pour les appels touchant à la lecture des répertoires, écrivez un programme qui dénombre récursivement les fichiers d'un répertoire.

Faites de même avec la taille qu'occupent ces fichiers.

Processus

Qu'est-ce qu'un processus

→ Abstraction de l'exécution d'un fil d'instructions

Chaque processus a son propre contexte :

- Identifiant unique
- Compteur ordinal
- Registres
- Mémoire (pile, tas)
- Fichiers ouverts

L'ordonnanceur du système possède une table des processus

- Pour l'utilisateur : tous les processus s'exécutent en même temps
- Pour le système : les processus s'exécutent à "tour de rôle"

→ Bascule de l'état d'un processus : actif, bloqué, prêt

- Notion de **parallélisme** : programme partagé en plusieurs processus

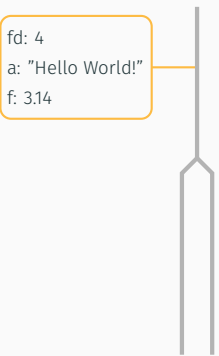
Création de processus

```
1 pid_t fork(void);  
2  
3 pid_t p;  
4  
5 p = fork();  
6 if (p) { // partie du proc. père  
7     ...  
8     ...  
9 } else { // partie du proc. fils  
10     ...  
11     ...  
12 }
```

- Invocation du nouveau processus avec **fork()**
- Contexte du fils copié à partir de celui du père
- La valeur de retour de **fork()** permet de différencier les deux processus :
 - >0 : processus père
 - 0 : processus fils
 - <0 : erreur

Fil d'exécution

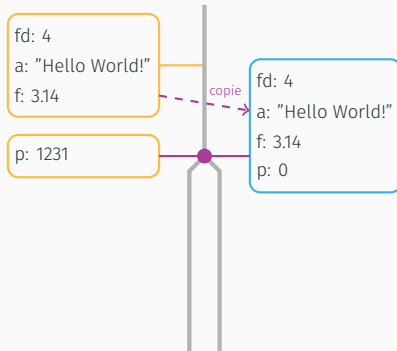
```
1 int fd = open("toto", O_RDWR);
2 char a[16] = "Hello World!";
3 double f = 3.14;
4 pid_t p;
5
6 p = fork();
7
8 if (p)
9     f = 1.62;
10 else
11     close(fd);
12
13 a[p%2] = 'C';
```



fd: 4
a: "Hello World!"
f: 3.14

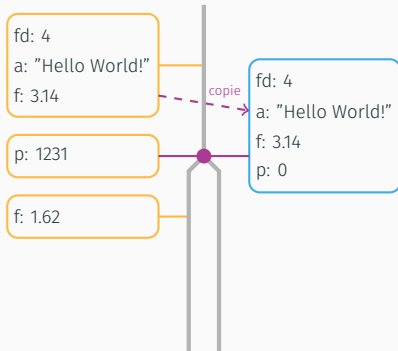
Fil d'exécution

```
1 int fd = open("toto", O_RDWR);
2 char a[16] = "Hello World!";
3 double f = 3.14;
4 pid_t p;
5
6 p = fork();
7
8 if (p)
9     f = 1.62;
10 else
11     close(fd);
12
13 a[p%2] = 'C';
```



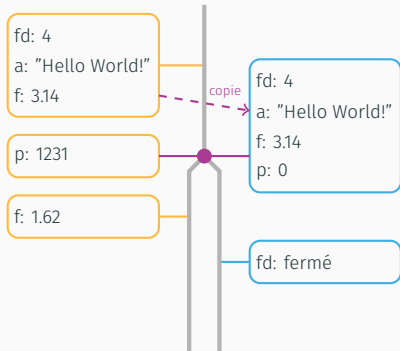
Fil d'exécution

```
1 int fd = open("toto", O_RDWR);
2 char a[16] = "Hello World!";
3 double f = 3.14;
4 pid_t p;
5
6 p = fork();
7
8 if (p)
9     f = 1.62;
10 else
11     close(fd);
12
13 a[p%2] = 'C';
```



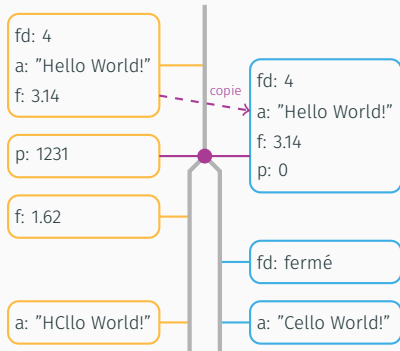
Fil d'exécution

```
1 int fd = open("toto", O_RDWR);
2 char a[16] = "Hello World!";
3 double f = 3.14;
4 pid_t p;
5
6 p = fork();
7
8 if (p)
9     f = 1.62;
10 else
11     close(fd);
12
13 a[p%2] = 'C';
```



Fil d'exécution

```
1 int fd = open("toto", O_RDWR);
2 char a[16] = "Hello World!";
3 double f = 3.14;
4 pid_t p;
5
6 p = fork();
7
8 if (p)
9     f = 1.62;
10 else
11     close(fd);
12
13 a[p%2] = 'C';
```



Fin d'un processus fils

```
1 void exit(int status);
```

- Un des moyens d'arrêter un processus

```
1 pid_t wait(int *status);  
2 pid_t waitpid(pid_t pid, int *status, int options);
```

- Fonction à appeler par le père pour s'assurer que son fils a terminé son exécution
- Possibilité de cibler quel processus on souhaite attendre

Qui suis-je ?

```
1 pid_t getpid();  
2 pid_t getppid();
```

- Récupérer son PID ou celui de son processus père

Qui ? \ PID de ?	père(A)	A	fls(A)
père(A)			
A			
fls(A)			

Qui suis-je ?

```
1 pid_t getpid();  
2 pid_t getppid();
```

- Récupérer son PID ou celui de son processus père

Qui ? \ PID de ?	père(A)	A	fls(A)
père(A)	<code>getpid()</code>		
A		<code>getpid()</code>	
fls(A)			<code>getpid()</code>

Qui suis-je ?

```
1 pid_t getpid();  
2 pid_t getppid();
```

- Récupérer son PID ou celui de son processus père

Qui ? \ PID de ?	père(A)	A	fls(A)
père(A)	<code>getpid()</code>		
A	<code>getppid()</code>	<code>getpid()</code>	
fls(A)		<code>getppid()</code>	<code>getpid()</code>

Qui suis-je ?

```
1 pid_t getpid();  
2 pid_t getppid();
```

- Récupérer son PID ou celui de son processus père

Qui ? \ PID de ?	père(A)	A	fils(A)
père(A)	<code>getpid()</code>	<code>fork()</code>	
A	<code>getppid()</code>	<code>getpid()</code>	<code>fork()</code>
fils(A)		<code>getppid()</code>	<code>getpid()</code>

Qui suis-je ?

```
1 pid_t getpid();  
2 pid_t getppid();
```

- Récupérer son PID ou celui de son processus père

Qui ? \ PID de ?	père(A)	A	fils(A)
père(A)	<code>getpid()</code>	<code>fork()</code>	<code>?</code>
A	<code>getppid()</code>	<code>getpid()</code>	<code>fork()</code>
fils(A)	<code>?</code>	<code>getppid()</code>	<code>getpid()</code>

→ Moyen de communication entre processus, pour notifier d'un événement

```
1 int kill(pid_t pid, int sig);
```

- Envoi d'un signal à un processus donné
- Plusieurs signaux possibles : **SIGINT**, **SIGTERM**, **SIGKILL**, **SIGSTOP**, **SIGCONT**, **SIGSEGV**, etc.

```
1 typedef void (*sighandler_t)(int);  
2 sighandler_t signal(int signum, sighandler_t handler);
```

- Définition d'une fonction de traitement de signal

Autres fonctions/commandes

```
1 unsigned int sleep(unsigned int seconds);
```

- Endormissement d'un processus

```
1 int execl(const char *path, const char *arg, ..., (char *) NULL);
```

- Remplacement du programme courant par celui ciblé, dans le même processus

```
$ ps  
$ top
```

- Affichage des processus courants

Ordonnement des processus

```
1  for (i = 0; i < 10; ++i) {  
2      if (!fork())  
3          break;  
4  }  
5  
6  printf("%d\n", getpid());
```

Dans quel ordre sont fait les affichages ?

Ordonnement des processus

```
1 for (i = 0; i < 10; ++i) {  
2     if (!fork())  
3         break;  
4 }  
5  
6 printf("%d\n", getpid());
```

Dans quel ordre sont fait les affichages ?

→ indéterminé, dépend de l'ordonnanceur

- Gère l'accès au processeur, et sélectionne les processus à exécuter à l'aide d'un algorithme
- Du choix de l'algorithme va dépendre l'efficacité de l'ordonnanceur à utiliser le processeur au maximum
- Par exemple, si des processus actifs attendent des entrées/sorties, il faut les mettre en attente et rendre d'autres processus actifs

But : Assurer l'attribution et l'utilisation équitable du processeur

Quand ordonnancer ?

Lorsqu'un ou des processus changent d'état :

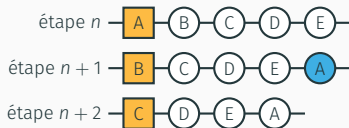
- Lors de la création de nouveaux processus
- Lors de la terminaison de processus
- Lors de l'attente de ressources
- Lors de l'acquisition des ressources
- Préemption

- First In First Out - Premier arrivé, premier exécuté
- Simple à implémenter

- Shortest Job First - Processus le plus rapide en premier
- Demande de connaître la durée des processus en amont

Algorithme round-robin

- → Prémption : chaque processus s'exécute pendant un quantum de temps donné, puis est mis en pause et l'exécution passe à un autre processus, etc.
- L'ordonnanceur maintient une file d'attente de processus, et sélectionne le premier de la liste pour l'exécuter
- Si à la fin du quantum de temps, le processus n'est pas terminé, il est remis dans la file d'attente



- Chaque processus possède un numéro/ticket
- L'ordonnanceur choisit aléatoire un numéro/ticket, et fournit au sélectionné un quantum de temps

Ajout de la priorité

- Certains processus sont plus importants que d'autres
- Sélection du prochain processus à exécuter en fonction de cette importance → **priorité**
- Affinement de la priorité par l'ordonnanceur pour éviter les situations de famine



Communication entre processus

- Excepté le code de retour, pas de communication 'native' possible entre deux processus
- Mais, plusieurs mécanismes peuvent être utilisés à cet effet :
 - Signaux
 - Fichiers (`mmap()` par exemple)
 - Tubes
 - Mémoire partagée

- Un tube est un *fichier* où un expéditeur/écrivain peut transmettre des informations à un destinataire/lecteur
- Deux types de tube :
 - Tubes nommés : informations stockées dans un *fichier* de type **FIFO**
 - Tubes anonymes : informations stockées en mémoire, ne fonctionne que si le créateur du tube est **un ancêtre commun** des interlocuteurs


```
1 int mkfifo(const char *pathname, mode_t mode);
```

- Ouverture bloquante, selon les systèmes, tant que les deux extrémités n'ont pas été ouvertes
- Suppression du tube avec la fonction `unlink()`

```
1 int pipe(int fildes[2]);
```

- Lecture sur `fildes[0]`, écriture sur `fildes[1]`

Opérations sur les tubes

→ Identiques aux opérations sur fichiers

```
1 void main() {  
2     const char *str =  
3         "Hello world!";  
4     int fd;  
5  
6     mkfifo("/tmp/tube-test",  
7         0666);  
8  
9     fd = open("/tmp/tube-test",  
10        O_WRONLY);  
11    write(fd, str, strlen(str));  
12    close(fd);  
13 }
```

```
1 void main() {  
2     char str[64];  
3     int fd;  
4  
5     fd = open("/tmp/tube-test",  
6        O_RDONLY);  
7     read(fd, str, 64);  
8     close(fd);  
9  
10    printf(  
11        "Just received: '%s'\n",  
12        str);  
13  
14    unlink("/tmp/tube-test");  
15 }
```

Mémoire partagée

```
1 int shmget(key_t key, size_t size, int shmflg);
```

Allocation d'un segment de mémoire partagée dont l'identifiant est **key**

```
1 void *shmat(int shmid, const void *shmaddr, int shmflg);  
2 int shmdt(const void *shmaddr);
```

Attachement/détachement d'un segment de mémoire à l'espace d'adressage partagée

```
1 int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Désallocation du segment à l'aide de la commande **IPC_RMID**

Threads

Qu'est-ce qu'un thread ?

Un **thread** peut être vu comme un processus léger qui possède :

- Compteur ordinal
- Registres
- Pile

- Mémoire **distribuée** : chaque entité possède une mémoire qui lui est propre
→ Processus
- Mémoire **partagée** : chaque entité partage sa mémoire avec les autres
→ Threads issus d'un même processus

→ Les allocations faites dans un thread sont accessibles dans les autres

Création de thread

Fonction permettant de créer un thread, qui va utiliser `start_routine()` comme fonction `main()`

```
1 int pthread_create(pthread_t *thread, const pthread_attr *attr,  
2 void *(*start_routine) (void *), void *arg);
```

- Equivalent à `fork()`
- Le thread fils termine son exécution une fois `start_routine()` terminée
- Les attributs `attr` servent à paramétrer la création du thread, `NULL` indique l'utilisation des attributs par défaut
- La fonction `start_function()` est un pointeur de fonction prenant un pointeur en entrée, et retournant un pointeur
- L'argument `arg` est le pointeur d'entrée de `start_function()`

Fonction à appeler par le thread père, pour attendre la fin de l'exécution du fils

```
1 int pthread_join(pthread_t thread, void **retval);
```

- Equivalent à `wait()` pour les processus
- Attente ciblée, en utilisant l'ID `thread`
- Valeur de retour du thread fils récupérée dans `*retval`

Exemple d'utilisation de pthread_create()

```
1 void * print_string(void *arg) {  
2     char *str = (char *) arg;  
3  
4     printf("Child says: '%s'\n", str);  
5     return NULL;  
6 }  
7  
8 void main() {  
9     pthread_t tid;  
10  
11     pthread_create(&tid, NULL, print_string, "Hello world!");  
12     pthread_join(tid, NULL);  
13 }
```

Fonction de sortie d'un thread, avec **retval**, la variable de retour de la fonction de thread

```
1 void pthread_exit(void *retval);
```

Abandon du CPU par le thread, basculant alors dans la file d'attente de l'ordonnanceur

```
1 int pthread_yield();
```

Problème en mémoire partagée

Soit un programme parallèle calculant la somme des éléments d'un tableau

```
1  int global_sum = 0;
2
3  void *compute_array_sum(void *arg) {
4      int *tab = (int *)arg;
5      for (i = start; i < end; ++i) {
6          global_sum += tab[i];
7      }
8  }
```

- Les threads accèdent au même tableau **tab**, sur des indices différents
- Ils mettent à jour la même variable **global_sum**
- Si deux threads mettent à jour la variable au même moment
→ comportement indéterminé

Exclusion mutuelle 1/3

Section critique → section de code où une variable est accédée en écriture par plusieurs threads

Exclusion mutuelle → mécanisme assurant qu'un **seul** thread exécute une section critique à un instant donné

```
1 pthread_mutex_t mutex;  
2  
3 int pthread_mutex_init(pthread_mutex_t *mutex,  
4                        const pthread_mutexattr_t *mutexattr);  
5 int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Création et destruction de la structure de données utilisée pour l'exclusion mutuelle
- Comportement par défaut : `mutexattr` vaut `NULL`

```
1 int pthread_mutex_lock(pthread_mutex_t *mutex);  
2 int pthread_mutex_trylock(pthread_mutex_t *mutex);  
3 int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Entrée dans une section critique en verrouillant le mutex, se met en attente tant qu'il n'est pas entré
- Sortie de la section en déverrouillant le mutex
- `mutex_trylock()` ne se met pas en attente, et place `errno` à `EBUSY` s'il n'a pas pu entrer dans la section

Reprise de l'exemple avec utilisation du mutex, pour assurer qu'un seul thread à la fois modifie la variable partagée

```
1 pthread_mutex_t mut;
2 int global_sum = 0;
3
4 void *compute_array_sum(void *arg) {
5     int *tab = (int *)arg;
6     for (i = start; i < end; ++i) {
7         pthread_mutex_lock(&mut);
8         global_sum += tab[i];
9         pthread_mutex_unlock(&mut);
10    }
11 }
```

Condition → mécanisme assurant l'attente d'un thread tant qu'une "condition n'est pas vérifiée"

```
1 pthread_cond_t cond;  
2  
3 int pthread_cond_init(pthread_cond_t *cond,  
4                       const pthread_condattr_t *cond_attr);  
5 int pthread_cond_destroy(pthread_cond_t *cond);
```

- Création et destruction identiques à l'exclusion mutuelle (`cond_attr` à `NULL` par défaut)


```
1 int pthread_cond_wait(pthread_cond_t *cond,  
2                          pthread_mutex_t *mutex);  
3 int pthread_cond_signal(pthread_cond_t *cond);  
4 int pthread_cond_broadcast(pthread_cond_t *cond);
```

- Le thread fait appel à `cond_wait()` pour attendre d'être libéré par un autre
- Cet appel doit être fait dans une **section critique** (gérée par `mutex`)
- `cond_signal()` permet de libérer un thread qui attend
- `cond_broadcast()` permet de libérer tous les threads qui attendent

Barrière

Barrière → mécanisme assurant qu'un nombre donné de threads s'attendent mutuellement à un point donné de leur exécution

```
1 pthread_barrier_t barrier;  
2  
3 int pthread_barrier_init(pthread_barrier_t *barrier,  
4                          const pthread_barrierattr_t *attr,  
5                          unsigned count);  
6 int pthread_barrier_destroy(pthread_barrier_t *barrier);  
7 int pthread_barrier_wait(pthread_barrier_t *barrier);
```

- La variable **count** indique combien de threads on attend avant de tous les relâcher
- La barrière doit être ré-initialisée entre chaque utilisation
- Cette fonctionnalité n'est pas disponible sur tous les systèmes d'exploitation, car elle est **optionnelle**

Organisation de la mémoire

- Il existe encore un rapport proportionnel entre la rapidité d'un composant mémoire, et son coût
- On établit alors une hiérarchie mémoire (du plus rapide au plus lent) :
 1. Registre
 2. Cache
 3. Mémoire vive (RAM)
 4. Stockage de masse (Flash, Disques)

- Conserve l'état de la mémoire en cours d'utilisation
- Alloue la mémoire aux processus, et la libère une fois leur exécution terminée
- Gère le *swapping* entre la mémoire principale (RAM) et le disque

- Mécanisme stockant sur le disque les processus en attente, afin de libérer une partie de la mémoire principale
- Un processus prêt à être exécuté peut être à nouveau chargé en mémoire

Virtualisation de la mémoire 1/2

Chaque processus dispose de son propre espace mémoire :

- programme : binaire exécutable
- données : variables globales et statiques
- tas : mémoire allouée dynamiquement
- pile : variables locales, arguments et adresses de retour de fonction



- La taille de l'espace mémoire alloué à un processus peut être augmentée, et dépasser la capacité disponible de la mémoire physique
- Elle présente un adressage virtuel : besoin d'une traduction entre les adresses virtuelles/logiques (entre 0x000000 et 0xFFFFFFFF par exemple) et les adresses physiques (sur l'ensemble de la RAM)
→ C'est également le rôle du gestionnaire de mémoire

Traduction des adresses

La traduction est gérée à l'aide de deux registres, pour des meilleures performances :

- Registre de translation : adresse du début de l'espace alloué au programme, dans la RAM
- Registre de limite : limite des adresses logiques réservées à un processus
- Equation de traduction :

$$adr_{phys} = adr_{log} + reg_{translation}$$

$$adr_{log} \geq reg_{limite} \Rightarrow \text{SIGSEGV}$$

- L'espace d'adressage est divisé en blocs de taille fixe (de nos jours, 4ko ou 16ko)
- Une adresse physique est constituée d'un numéro de page, et d'un déplacement dans cette page
- Par exemple, sur une taille de 4ko :

$$adr_{phys} = 19,5ko \Leftrightarrow (\text{page n}^\circ 4, \text{offset de } 3,5ko)$$

- Besoin d'une table des pages en mémoire, décrivant chaque page de l'espace virtuel avec : numéro de page, bit de validité, bit d'accès en écriture, etc.
- Où stocker cette table ?
 - Besoin d'espace, car potentiellement beaucoup de pages chargées → disque ?
 - Besoin d'une faible latence d'accès, pour de meilleures performances → registre ?
- *Translation Lookaside Buffer* : solution hybride gardant en mémoire rapide les pages les plus utilisées

- Séparation de la mémoire en plusieurs espaces d'adressage linéaire, appelés segments
- Chaque segment peut posséder une taille différente des autres, pouvant augmenter dans le temps et contenant des objets de même type (juste le code, juste la pile d'exécution, etc.)
- Ceci permet d'ajuster les caractéristiques du segment aux données qu'il contient
- Une adresse physique est constituée d'un numéro de segment, et d'une adresse dans ce segment

- Combinaison des deux stratégies
- Permet d'éviter de charger des segments trop grands, tout en ayant les avantages de la segmentation
- Une adresse physique est alors constituée d'un numéro de segment, puis d'un numéro de page dans le segment et d'un déplacement dans cette page

Il existe plusieurs algorithmes d'ordonnement de pages mémoire :

- But → si l'espace mémoire est plein, quelle ancienne page enlevée pour accueillir la nouvelle ?
- Métrique → minimiser le nombre de défauts de page
- **Défaut de page** : page mémoire non directement accessible dans la mémoire primaire, devant être récupérée dans la mémoire secondaire
- Succès de page : page mémoire directement accessible dans la mémoire primaire

- Principe : la page enlevée est celle qui ne sera plus utilisée ou le plus tard possible
- **Avantage** : optimal, utilisé comme référentiel
- **Inconvénient** : théorique uniquement, nécessite de connaître l'ordre des pages accédées

- Principe : la page enlevée est celle qui a été mise en mémoire il y a le plus de temps
- **Avantage** : simple à mettre en place
- **Inconvénient** : des 'bonnes' pages peuvent être enlevées

Algorithme de seconde chance

- Principe : FIFO, chaque page possède une seconde chance avant d'être enlevée
 - La seconde chance est représentée par un bit initialisé à 1
 - Si le bit est à 1 lorsque la page doit être enlevée, elle est remise à la fin de la FIFO avec le bit à 0
 - Si le bit est à 0 lorsque la page doit être enlevée, elle est enlevée
 - Si la page est accédée, son bit est remis à 1
- **Avantage** : FIFO améliorée
- **Inconvénient** : manipulation un peu longue, peut être améliorée avec une liste doublement chaînée (horloge)

- Least Recently Used
- Principe : la page enlevée est celle qui a été accédée il y a le plus de temps
- **Avantage** : efficace
- **Inconvénient** : mise à jour des statistiques après chaque accès

- Not Frequently Used
- Principe : la page enlevée est l'une des moins fréquemment utilisées
 - La fréquence est calculée pour chaque page dans un compteur logiciel
- **Avantage** : mises à jour moins lourdes que LRU
- **Inconvénient** : approximation grossière de LRU

- Principe : on considère un ensemble de page (généralement un ensemble de pages par thread), et on applique un LRU ou NFU dessus
- **Avantage** : simple et efficace