



Programmation des systèmes

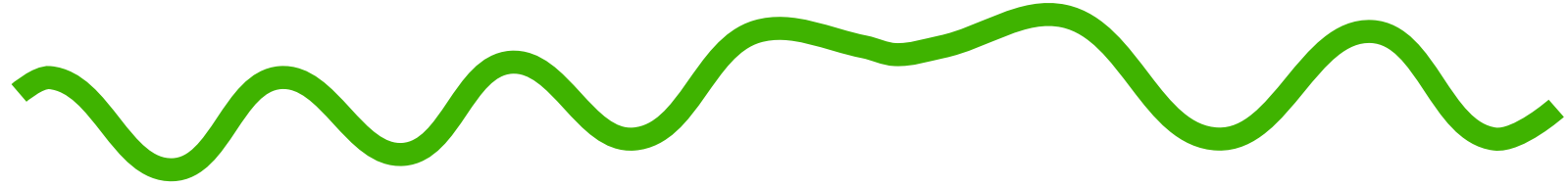
Gestion des processus

Philippe MARQUET

`Philippe.Marquet@lifl.fr`

Laboratoire d'informatique fondamentale de Lille
Université des sciences et technologies de Lille

Licence d'informatique de Lille
février 2005
révision de février 2006



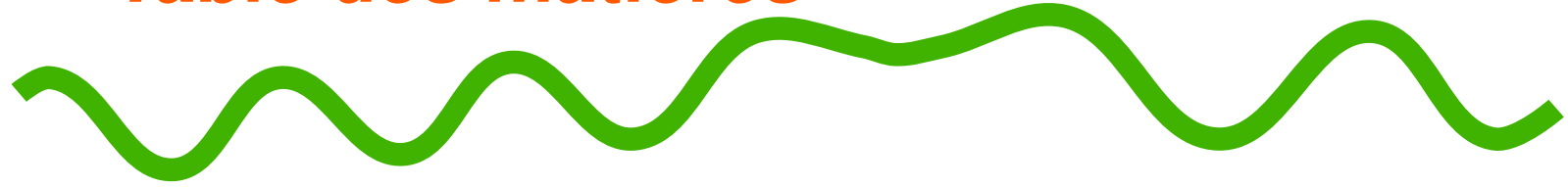
- ~ Ce cours est diffusé sous la GNU Free Documentation License,
`www.gnu.org/copyleft/fdl.html`
- ~ La dernière version de ce cours est accessible à
`www.lifl.fr/~marquet/cnl/pds/`
- ~ `$Id: ps.tex,v 1.13 2007/02/13 14:58:22 marquet Exp $`

Références & remerciements

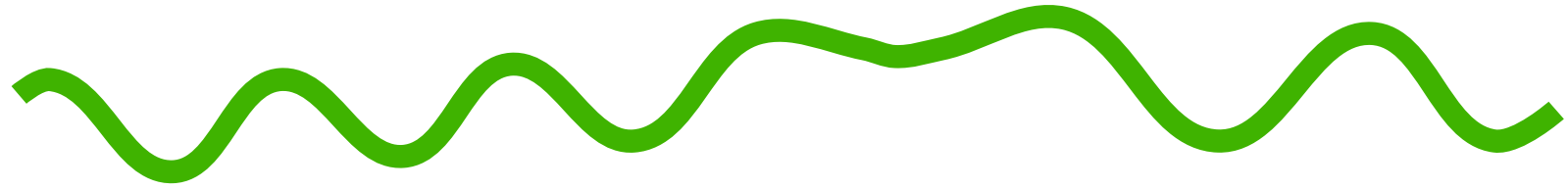


- ✧ *Unix, programmation et communication*
Jean-Marie Rifflet et Jean-Baptiste Yunès
Dunod, 2003
- ✧ *Introduction aux Systèmes et aux Réseaux*
Sacha Krakowiak
Cours de Licence informatique, Université Joseph Fourier, Grenoble
- ✧ *The Single Unix Specification*
The Open Group
www.unix.org/single_unix_specification/
- ✧ man **section 2**

Table des matières



- ~ Notion de processus
- ~ Éléments d'ordonnancement des processus



Notion de processus

Processus & programme



~ Programme

- ~ description statique
- ~ code, suite d'instructions

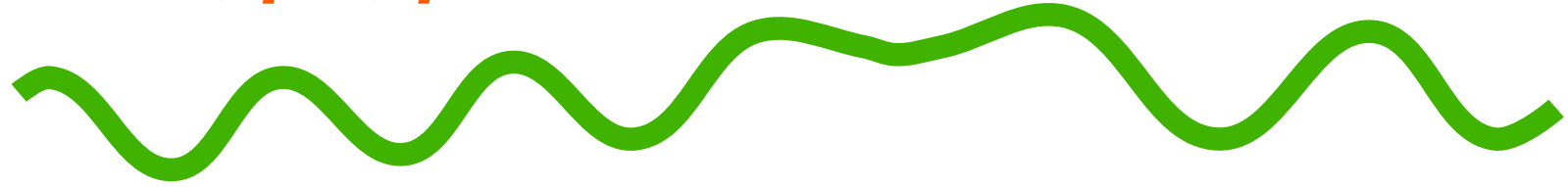
~ Processus

- ~ activité dynamique, temporelle
- ~ vie d'un processus : création d'un processus, exécution, fin d'un processus

~ *Un processus est une instance d'exécution d'un programme*

- ~ plusieurs exécutions de programmes
- ~ plusieurs exécutions d'un même programme
- ~ plusieurs exécutions « simultanées » de programmes différents
- ~ plusieurs exécutions « simultanées » d'un même programme

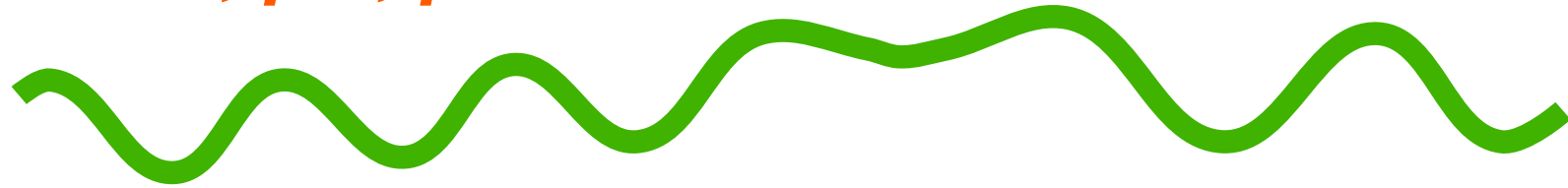
P..., p..., processeur



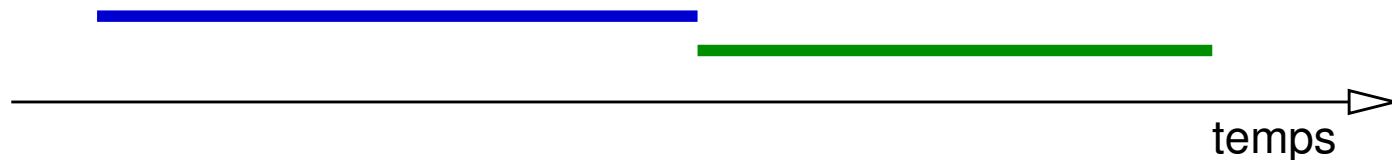
- ~ Programme, processus... processeur
 - ~ entité matérielle
 - ~ désigne l'utilisation du processeur
- ~ Affectation du processeur à un processus
 - ~ pour un temps donné
 - ~ permet de faire progresser le processus
- ~ Choix de cette affectation = ordonnancement
 - ~ système multiprocessus
 - ~ choix à la charge du système d'exploitation (...à suivre)
- ~ P..., p..., p..., parallélisme, pseudo-parallélisme
 - ~ plusieurs processus, un processeur
 - ~ entrelacement des processus
 - ~ deux processus



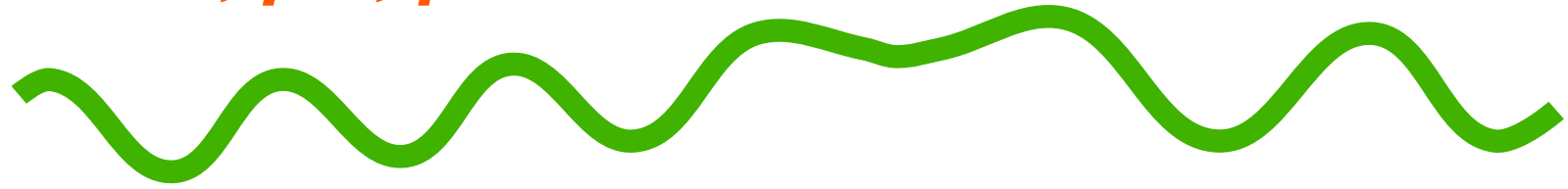
P..., p..., processeur



- ~ Programme, processus... processeur
 - ~ entité matérielle
 - ~ désigne l'utilisation du processeur
- ~ Affectation du processeur à un processus
 - ~ pour un temps donné
 - ~ permet de faire progresser le processus
- ~ Choix de cette affectation = ordonnancement
 - ~ système multiprocessus
 - ~ choix à la charge du système d'exploitation (...à suivre)
- ~ P..., p..., p..., parallélisme, pseudo-parallélisme
 - ~ plusieurs processus, un processeur
 - ~ entrelacement des processus
 - ~ exécutions séquentielles



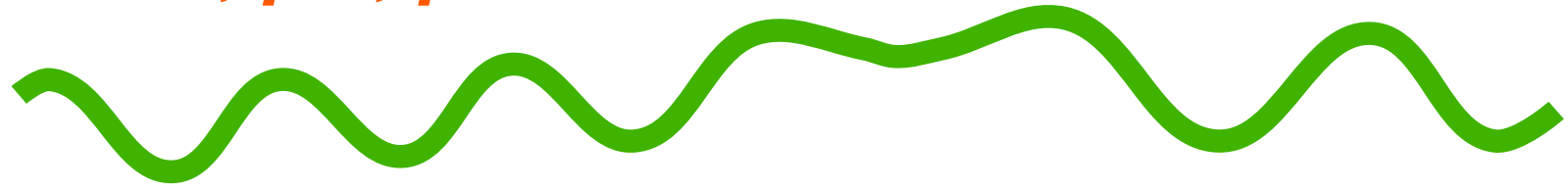
P..., p..., processeur



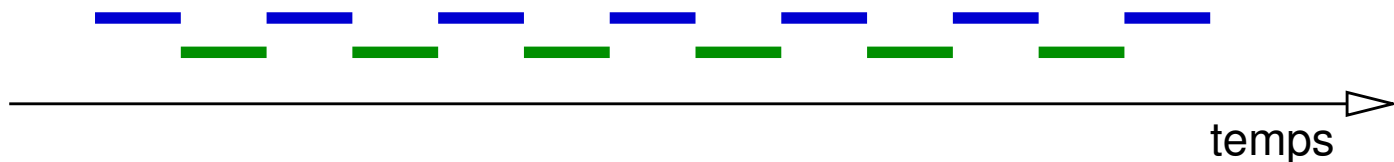
- ~ Programme, processus... processeur
 - ~ entité matérielle
 - ~ désigne l'utilisation du processeur
- ~ Affectation du processeur à un processus
 - ~ pour un temps donné
 - ~ permet de faire progresser le processus
- ~ Choix de cette affectation = ordonnancement
 - ~ système multiprocessus
 - ~ choix à la charge du système d'exploitation (...à suivre)
- ~ P..., p..., p..., parallélisme, pseudo-parallélisme
 - ~ plusieurs processus, un processeur
 - ~ entrelacement des processus
 - ~ exécutions parallèles (deux processeurs)



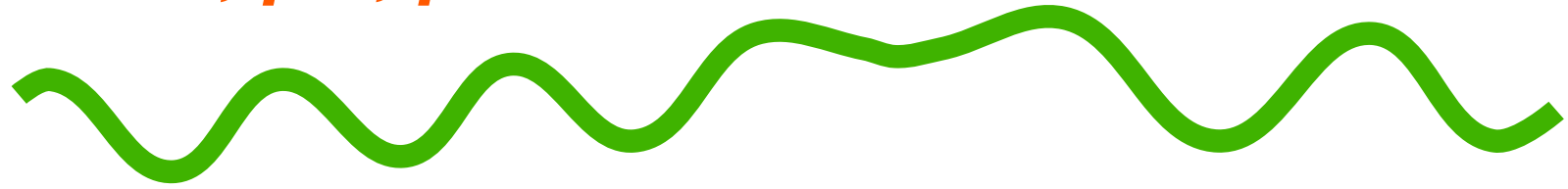
P..., p..., processeur



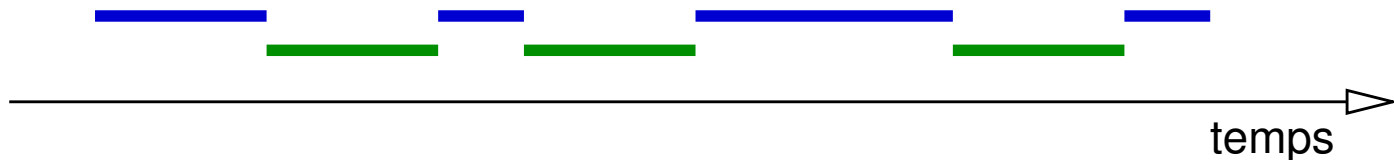
- ~ Programme, processus... processeur
 - ~ entité matérielle
 - ~ désigne l'utilisation du processeur
- ~ Affectation du processeur à un processus
 - ~ pour un temps donné
 - ~ permet de faire progresser le processus
- ~ Choix de cette affectation = ordonnancement
 - ~ système multiprocessus
 - ~ choix à la charge du système d'exploitation (...à suivre)
- ~ P..., p..., p..., parallélisme, pseudo-parallélisme
 - ~ plusieurs processus, un processeur
 - ~ entrelacement des processus
 - ~ exécutions entrelacées



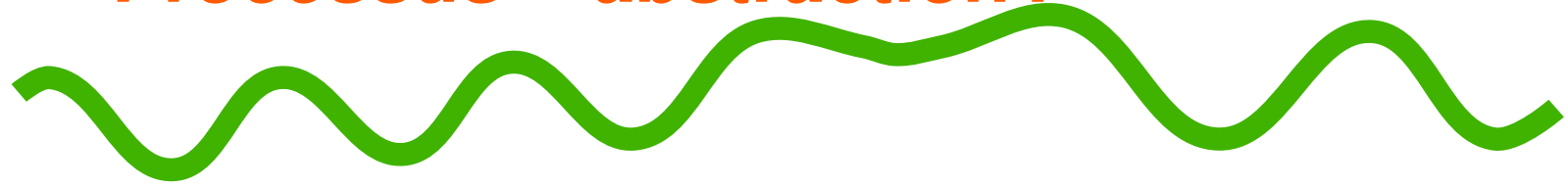
P..., p..., processeur



- ~ Programme, processus... processeur
 - ~ entité matérielle
 - ~ désigne l'utilisation du processeur
- ~ Affectation du processeur à un processus
 - ~ pour un temps donné
 - ~ permet de faire progresser le processus
- ~ Choix de cette affectation = ordonnancement
 - ~ système multiprocessus
 - ~ choix à la charge du système d'exploitation (...à suivre)
- ~ P..., p..., p..., parallélisme, pseudo-parallélisme
 - ~ plusieurs processus, un processeur
 - ~ entrelacement des processus
 - ~ autre entrelacement



Processus = abstraction !



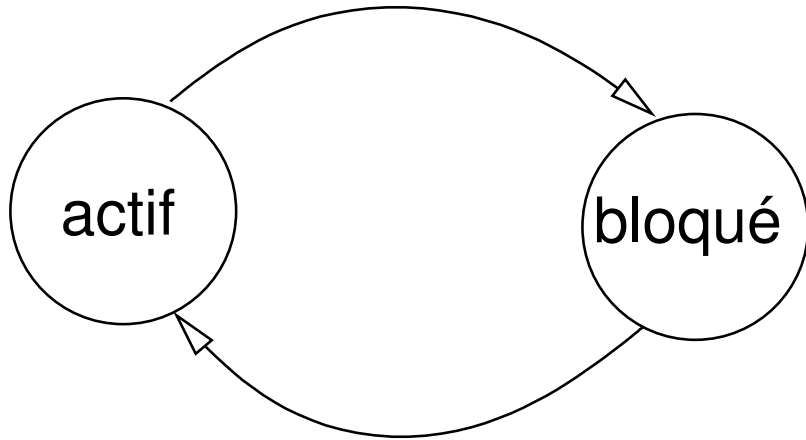
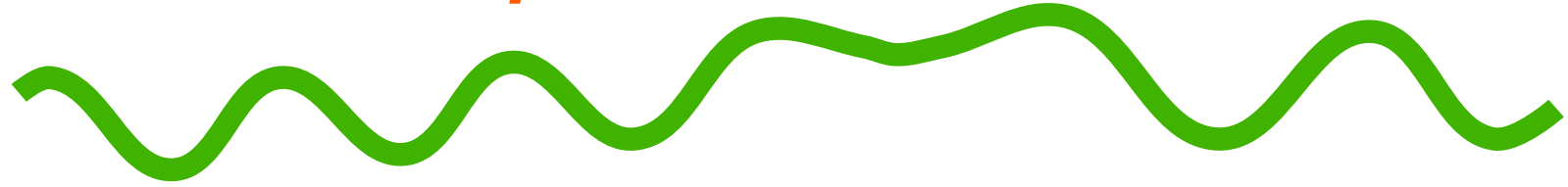
- ~ Processus = exécution abstraite d'un programme
 - ~ indépendante de l'avancement réel de l'exécution
- ~ Exécution d'un programme = réunion des instants d'exécution réelle du programme
 - ~ dépend de la disponibilité du processeur
- ~ Processus = abstraction
 - ~ désigne une entité identifiable
 - ~ par exemple : priorité d'un processus
 - ~ parallélisme, simultanéité, interaction... de deux processus
- ~ Compétition (*race condition*)
 - ~ résultats de deux processus dépend de cet entrelacement
 - ~ par exemple à cause d'accès partagés à un fichier...
 - ~ à éviter

Processus & ressources



- ~ Processus = exécution d'un programme
 - ~ requiert des ressources
- ~ Ressource
 - ~ entité nécessaire à l'exécution d'un processus
 - ~ ressources matérielles : processeur, périphérique...
 - ~ ressources logicielles : variable...
- ~ Caractéristiques d'une ressource
 - ~ état : libre, occupée
 - ~ nombre de possibles utilisations concurrentes (ressource à accès multiples)
- ~ Ressources indispensables à un processus
 - ~ mémoire propre (mémoire virtuelle)
 - ~ contexte d'exécution (état instantané du processus)
 - ~ pile (en mémoire)
 - ~ registres du processeur

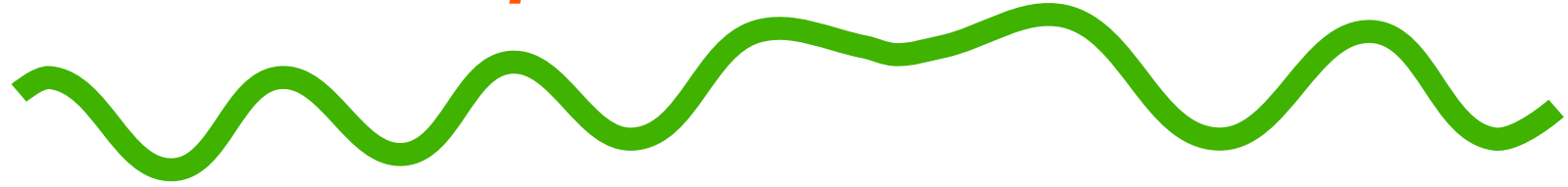
États d'un processus



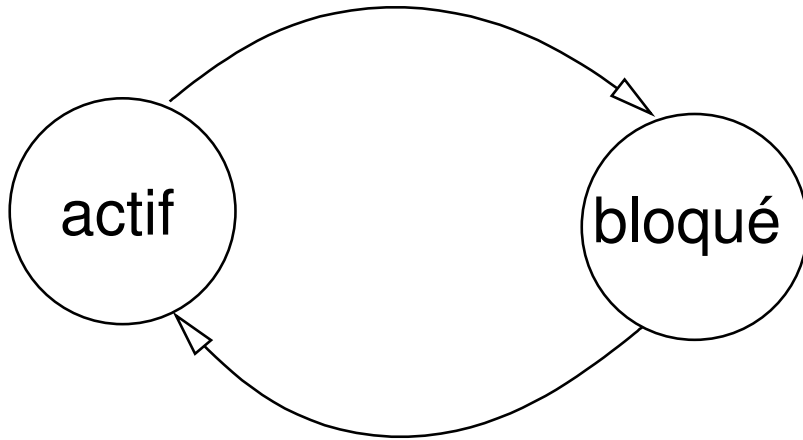
États logiques

indisponibilité
d'une ressource

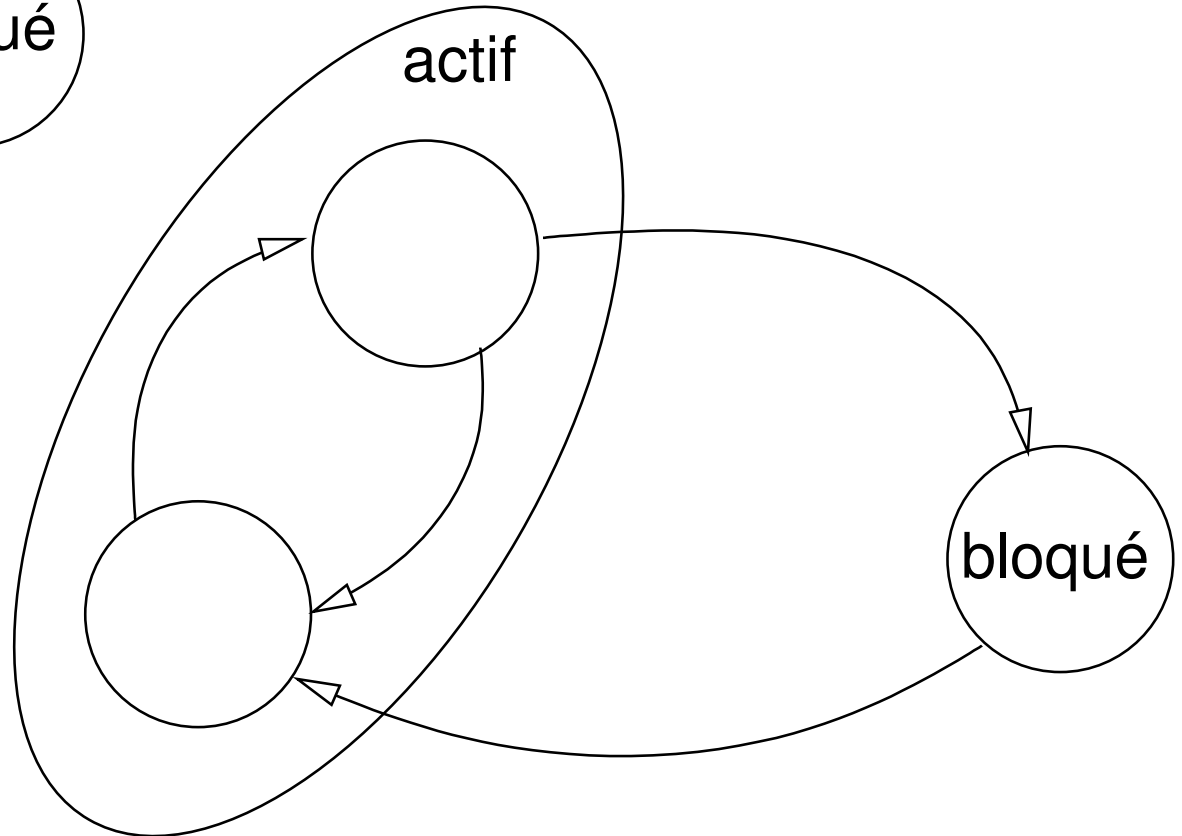
États d'un processus



États effectifs



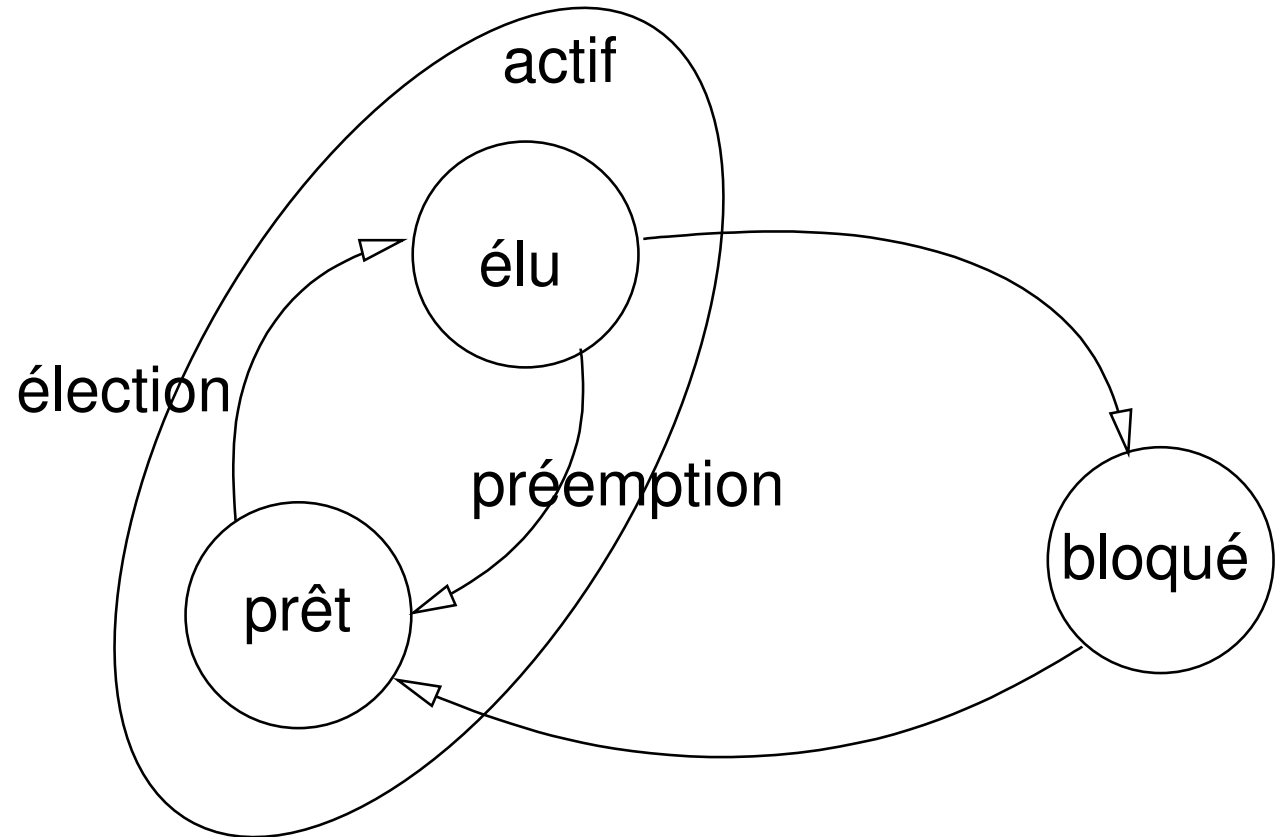
États logiques



particularisation
de la ressource
processeur

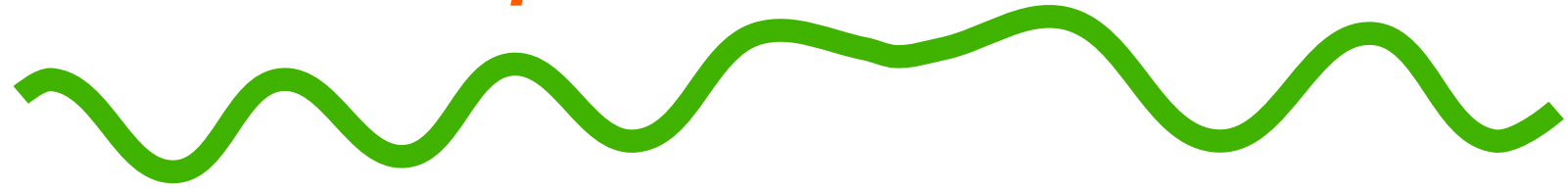
États d'un processus

États effectifs

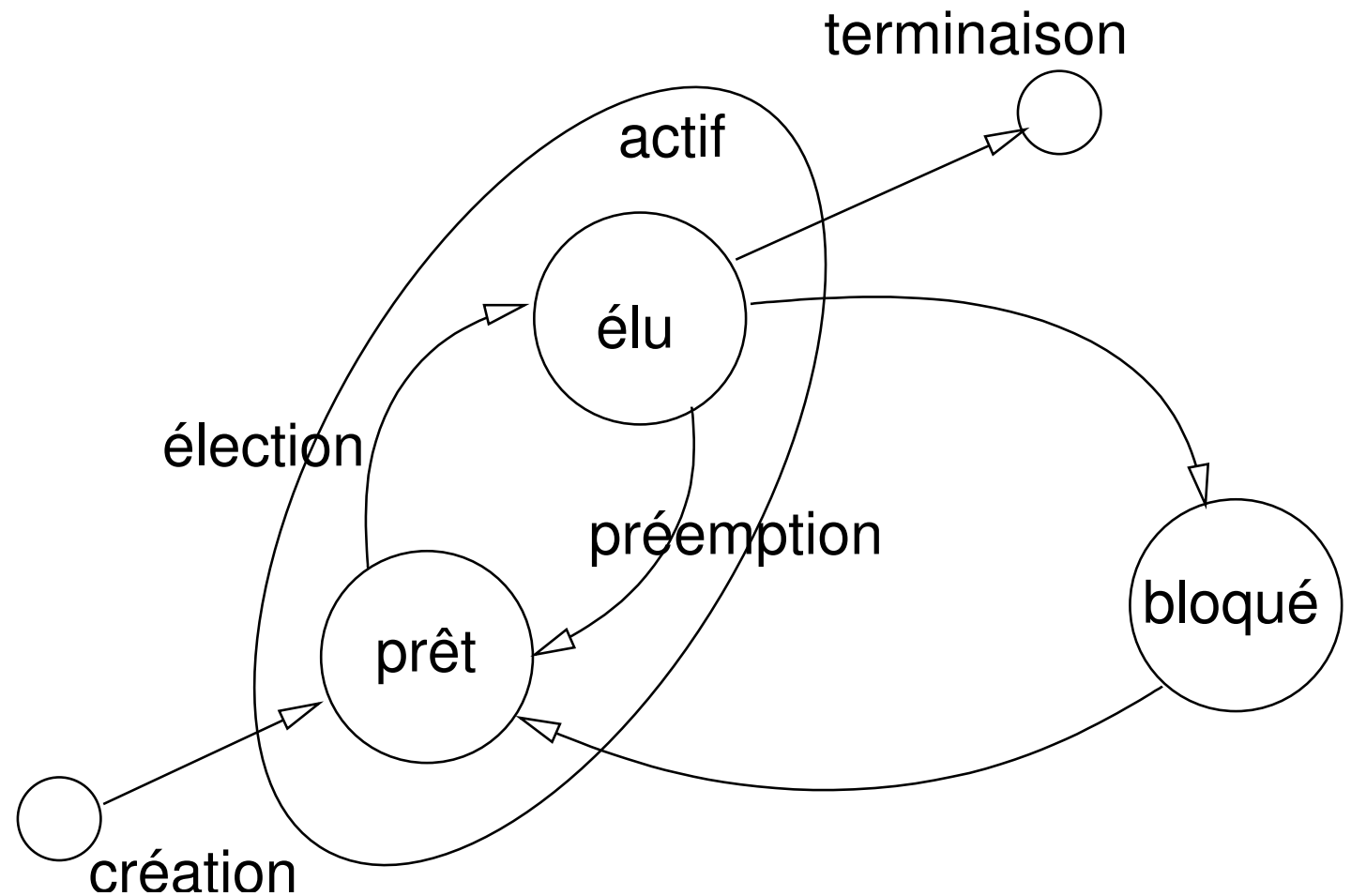


particularisation
de la ressource
processeur

États d'un processus

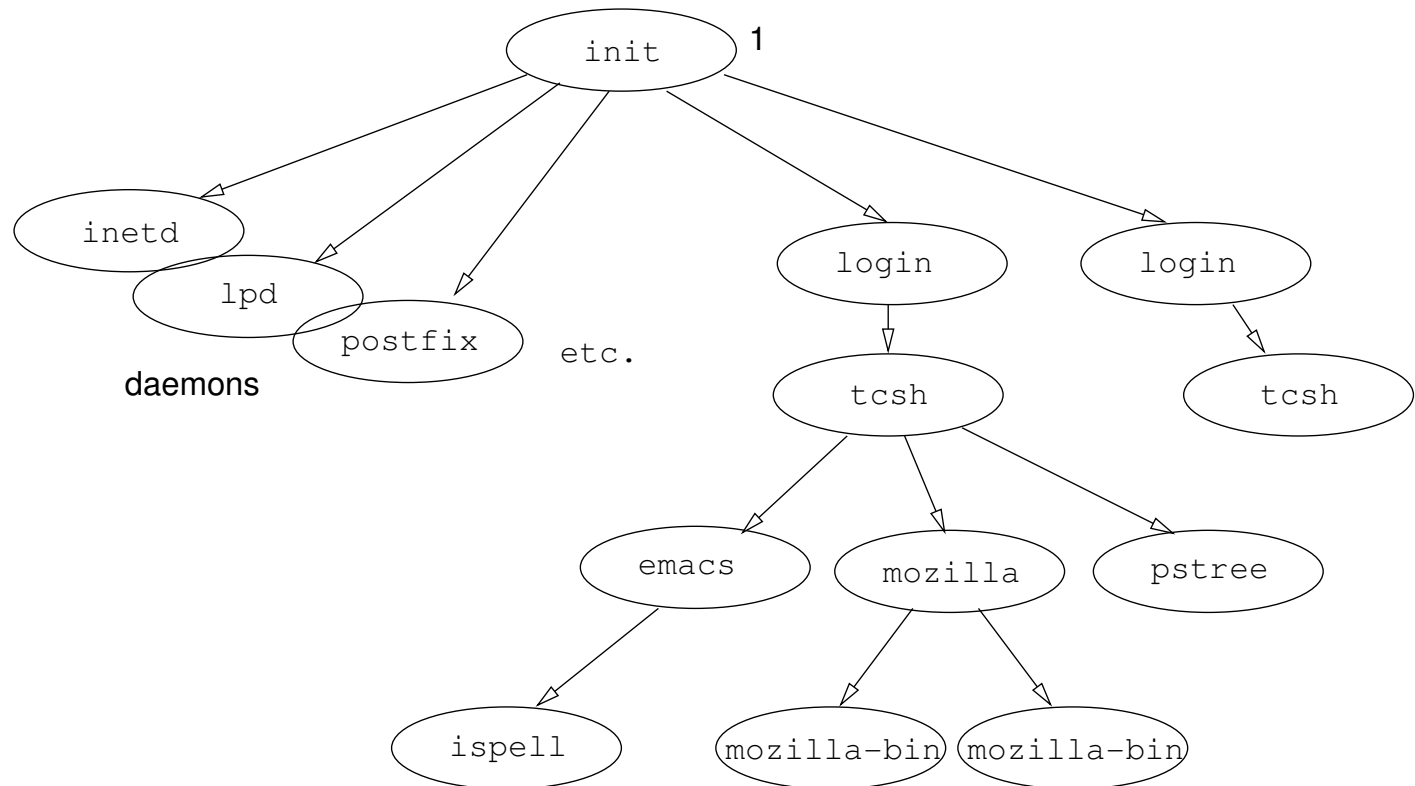


États effectifs



Hiérarchie des processus

- Création d'un processus par un processus père
- ⇒ hiérarchie
- processus ancêtre `init`



Attributs d'un processus



~ Identification univoque

- ~ *process ID*
- ~ numéro entier `pid_t`
- ~ numéro du processus père

~ Propriétaire

- ~ propriétaire réel
- ~ utilisateur qui a lancé le processus, son groupe

~ Propriétaire effectif, et son groupe

- ~ détermine les droits du processus
- ~ peut être modifié / propriétaire réel

~ `#include <unistd.h>`

```
pid_t getpid();  
pid_t getppid();
```

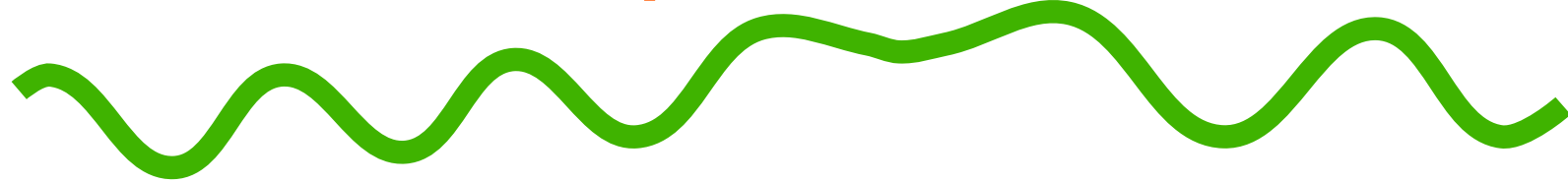
~ `#include <unistd.h>`

```
uid_t getuid();  
gid_t getgid();
```

```
uid_t geteuid();  
gid_t getegid();
```

```
int setuid(uid_t uid)  
int setgid(gid_t gid)
```

Attributs d'un processus (cont'd)



~ Répertoire de travail

- ~ origine de l'interprétation des chemins relatifs

```
#include <unistd.h>
```

```
char *getcwd(char *buffer, size_t bufsz);
```

retourne NULL en cas d'échec

- ~ peut être changé

```
#include <unistd.h>
```

```
int chdir(const char path);
```

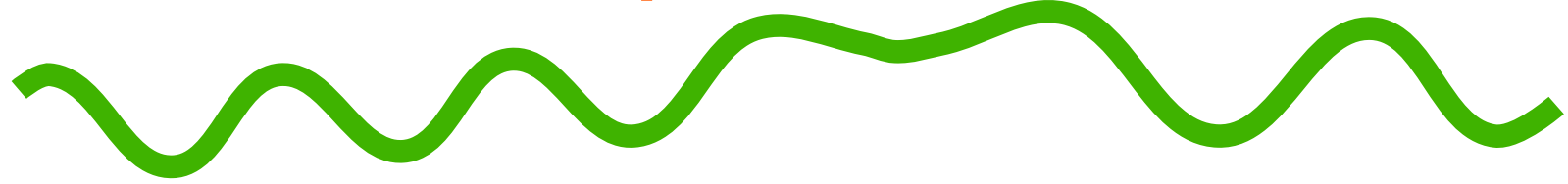
~ Date de création du processus

- ~ en secondes depuis l'*epoch*, 1^{er} janvier 1970

~ Temps CPU consommés

- ~ par le processus / par ses processus fils terminés
- ~ en mode utilisateur / en mode noyau
- ~ structure `struct tms`

Attributs d'un processus (cont'd)



~ Temps CPU consommés (cont'd)

~ `#include <sys/times.h>`

`clock_t times(struct tms *buffer);`

~ retourne le temps en tics horloge depuis un temps fixe dans le passé (démarrage du système par exemple)

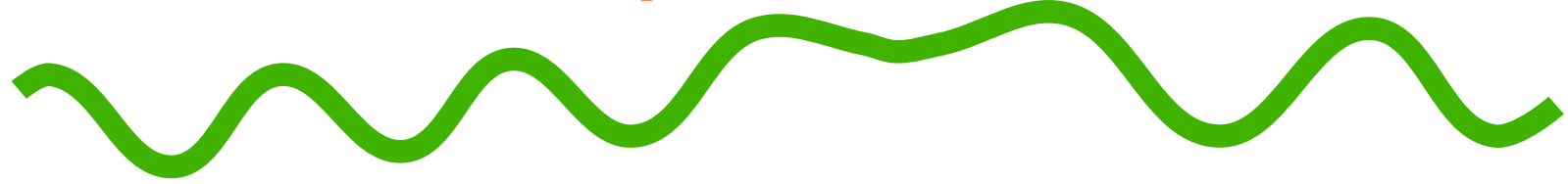
~ retourne 4 champs dans la structure `struct tms`

```
struct tms {  
    clock_t tms_utime;    /* User CPU time. */  
    clock_t tms_stime;    /* System CPU time. */  
    clock_t tms_cutime;   /* User CPU time of terminated  
                           child processes. */  
    clock_t tms_cstime;   /* System CPU time of  
                           terminated child processes. */  
};
```

~ tics horloge survenus alors que le processus était actif

~ conversion en secondes par division par la constante de configuration `_SC_CLK_TCK`

Attributs d'un processus (cont'd)



mtime.c

```
static clock_t st_time;
static clock_t en_time;
static struct tms st_cpu;
static struct tms en_cpu;

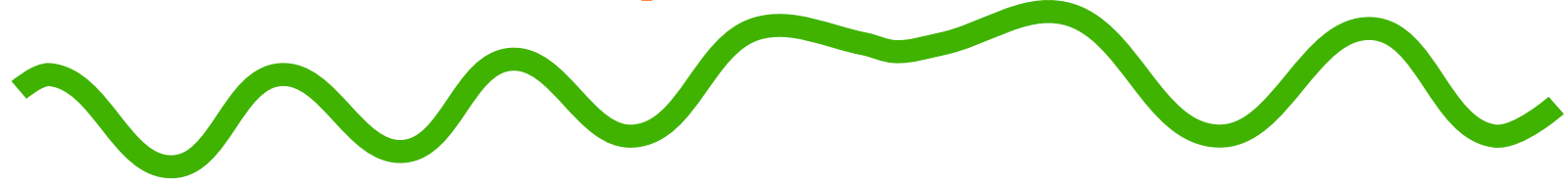
static float
tics_to_seconds(clock_t tics)
{
    return tics/(float)sysconf(_SC_CLK_TCK);
}

void
start_clock()
{
    st_time = times(&st_cpu);
}

void
end_clock()
{
    en_time = times(&en_cpu);

    printf("Real Time: %.2f, User Time %.2f, System Time %.2f\n",
        tics_to_seconds(en_time - st_time),
        tics_to_seconds(en_cpu.tms_utime - st_cpu.tms_utime),
        tics_to_seconds(en_cpu.tms_stime - st_cpu.tms_stime));
}
```

Attributs d'un processus (cont'd)



~ Masque de création des fichiers

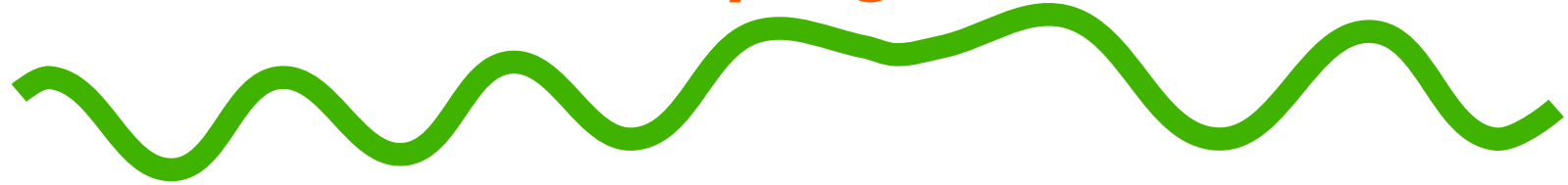
- ~ droits ne pouvant être positionnés lors de la création d'un fichier par `open()/create()`
- ~ voir le cours système de fichiers...

~ Table des descripteurs de fichiers

- ~ voir le cours système de fichiers...

~ Variables d'environnement

Lancement d'un programme



- ~ Lancement d'un programme réalisé en deux étapes
 - ~ création d'un nouveau processus par clonage de son père
 - ~ copie du processus père
 - ~ sauf l'identité, `pid`
 - ~ mutation pour exécuter un nouveau programme
- ~ Seul mécanisme possible
- ~ Clonage
 - ~ appel système `fork()`
- ~ Mutation
 - ~ appel système `execve()`
 - ~ famille de fonctions de bibliothèque `exec*()`

Clonage de processus



~ Appel système `fork()`

~ `#include <unistd.h>`

`pid_t fork(void);`

~ duplique le processus courant

~ retourne le `pid` du processus fils créé

Clonage de processus



~ Appel système `fork()`

~ `#include <unistd.h>`

`pid_t fork(void);`

- ~ duplique le processus courant
- ~ retourne le `pid` du processus fils créé
- ~ quid du processus fils ?

Clonage de processus



~ Appel système `fork()`

~ `#include <unistd.h>`

`pid_t fork(void);`

- ~ duplique le processus courant
- ~ retourne le `pid` du processus fils créé dans le processus père
- ~ retourne 0 dans le processus fils
- ~ retourne -1 en cas d'erreur

Clonage de processus



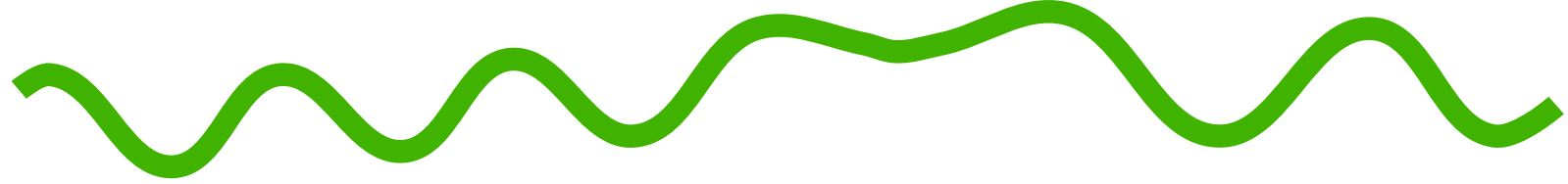
~ Appel système `fork()`

~ `#include <unistd.h>`

`pid_t fork(void);`

- ~ duplique le processus courant
- ~ retourne le `pid` du processus fils créé dans le processus père
- ~ retourne 0 dans le processus fils
- ~ retourne -1 en cas d'erreur

Clonage de processus – exemple



fork.c

```
int
main()
{
    pid_t status;

    printf("[%d] Je vais engendrer\n", getpid());

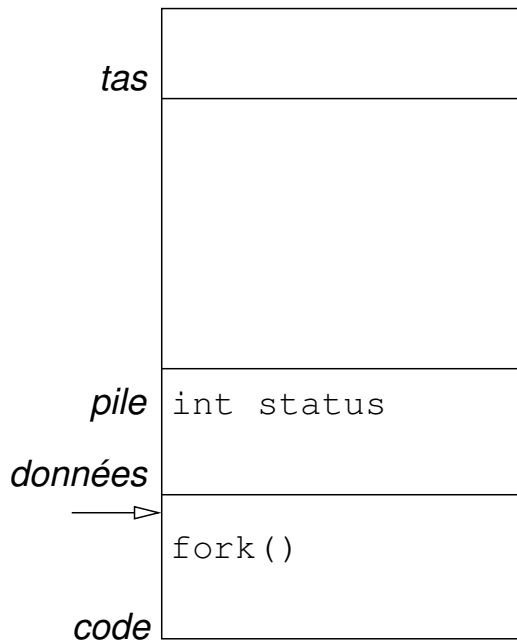
    status = fork();
    switch (status) {
        case -1 :
            perror("Creation processus");
            exit(EXIT_FAILURE);
        case 0 :
            printf("[%d] Je viens de naitre\n", getpid());
            printf("[%d] Mon pere est %d\n", getpid(), getppid());
            break;
        default:
            printf("[%d] J'ai engendre\n", getpid());
            printf("[%d] Mon fils est %d\n", getpid(), status);
    }

    printf("[%d] Je termine\n", getpid());

    exit(EXIT_SUCCESS);
}
```

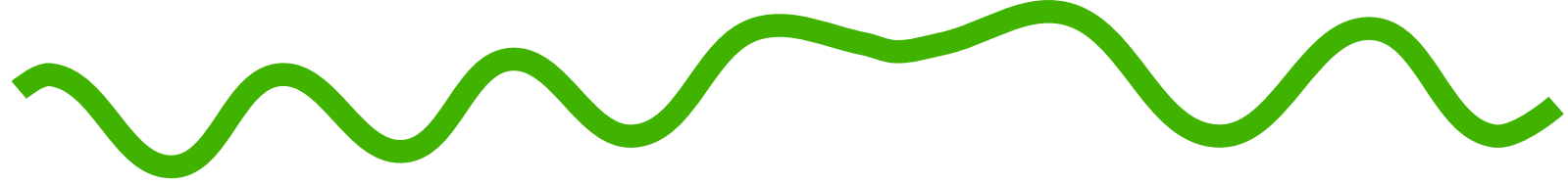
Clonage de processus – exécution

père
processus 2309

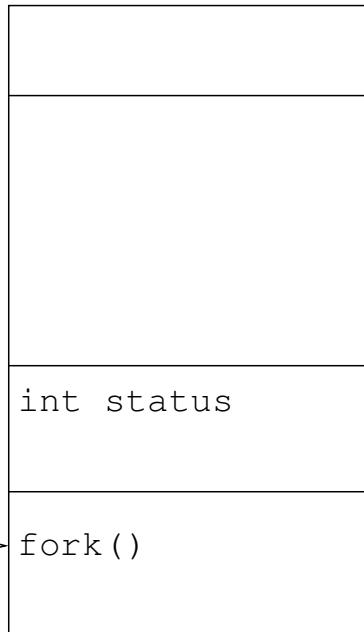


```
% ./fork
[2309] Je vais engendrer
[2310] Je viens de naitre
[2310] Mon pere est 2309
[2310] Je termine
[2309] J'ai engendre
[2309] Mon fils est 2310
[2309] Je termine
```

Clonage de processus – exécution

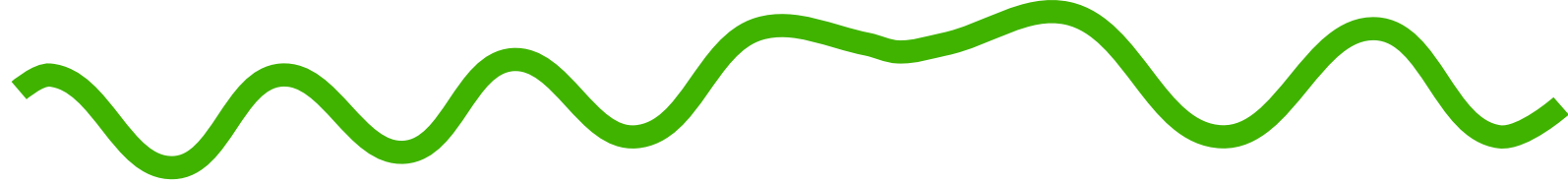


père
processus 2309

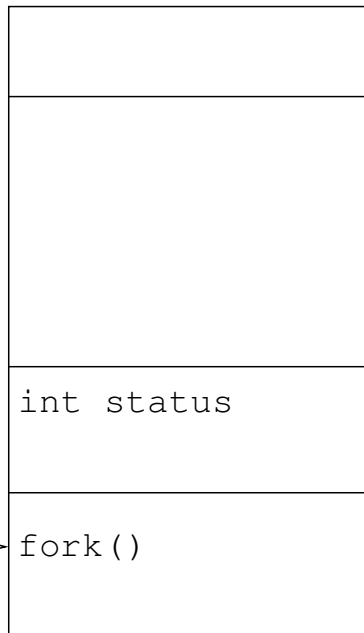


```
% ./fork
[2309] Je vais engendrer
[2310] Je viens de naitre
[2310] Mon pere est 2309
[2310] Je termine
[2309] J'ai engendre
[2309] Mon fils est 2310
[2309] Je termine
```

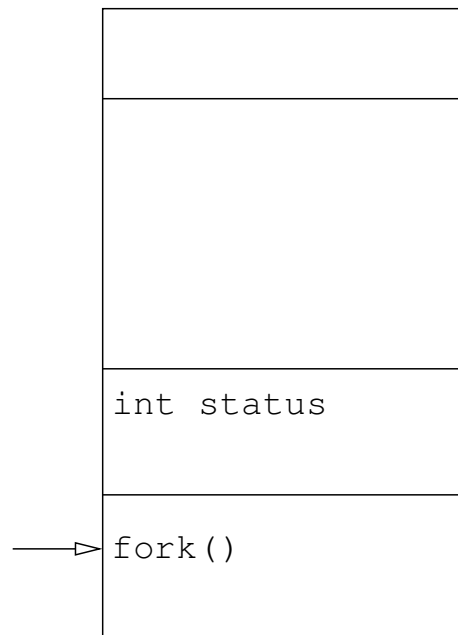
Clonage de processus – exécution



père
processus 2309

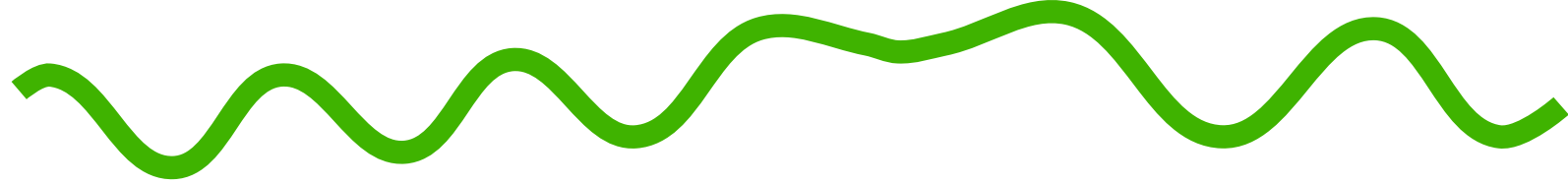


fils
processus 2310

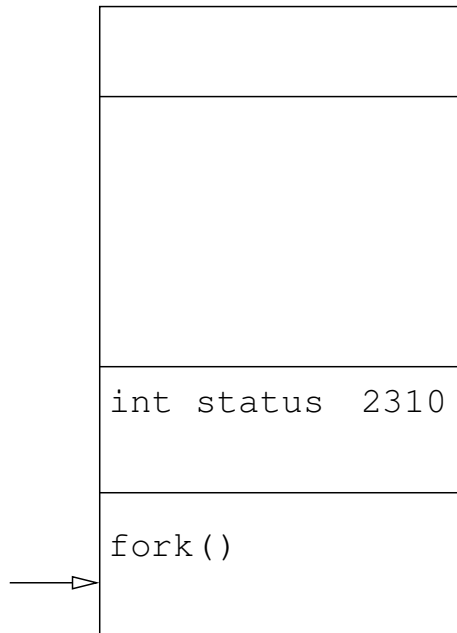


```
% ./fork
[2309] Je vais engendrer
[2310] Je viens de naitre
[2310] Mon pere est 2309
[2310] Je termine
[2309] J'ai engendre
[2309] Mon fils est 2310
[2309] Je termine
```

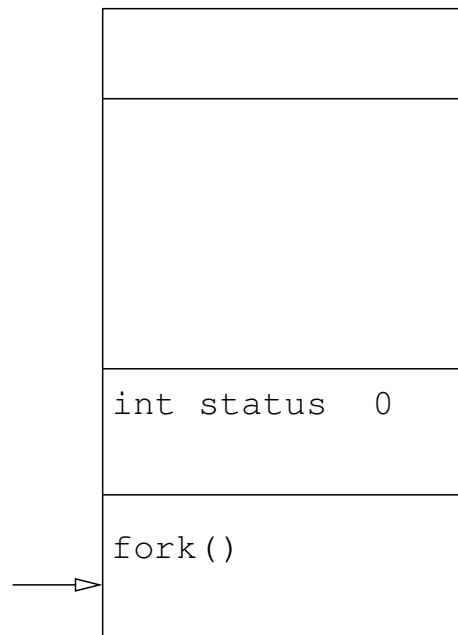

Clonage de processus – exécution



père
processus 2309

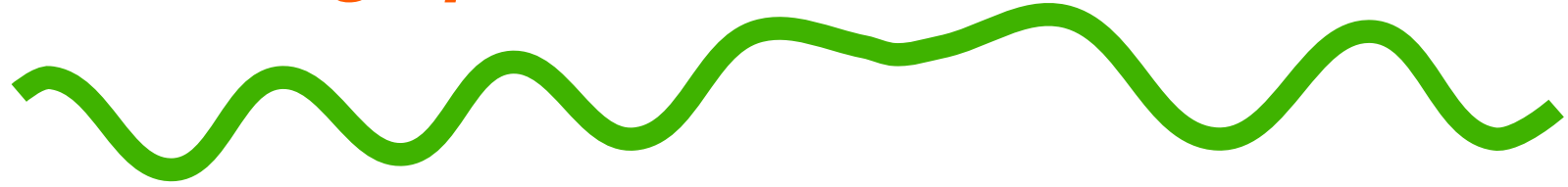


fils
processus 2310



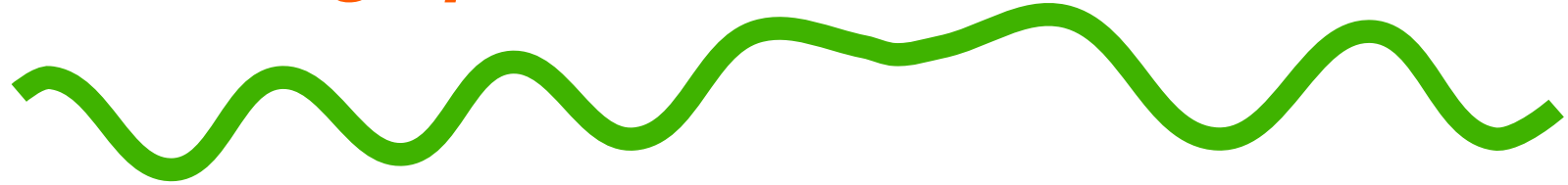
```
% ./fork
[2309] Je vais engendrer
[2310] Je viens de naitre
[2310] Mon pere est 2309
[2310] Je termine
[2309] J'ai engendre
[2309] Mon fils est 2310
[2309] Je termine
```

Héritage père/fils



- ~ Mémoire du processus fils = copie de la mémoire du processus père
- ~ Copie de références = partage
 - ~ la mémoire du père est copiée pour créer la mémoire du fils
 - ~ la mémoire du processus référence des structures systèmes
 - ~ des structures systèmes restent partagées entre père et fils
- ~ Conséquences de la copie de la mémoire
 - ~ tampons d'écriture de la bibliothèque standard d'entrées/sorties dupliqués
 - ~ vider ce tampon avant `fork()` (par un appel à `fflush()`)
 - ~ problème des tampons de lecture

Héritage père/fils (cont'd)



noflush.c

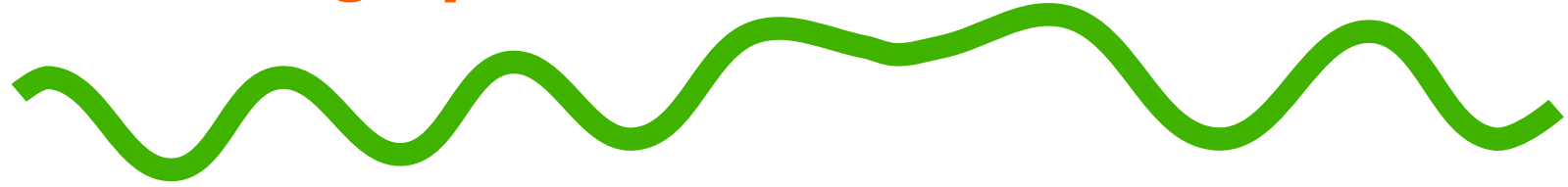
```
int
main()
{
    pid_t status;

    printf("debut ");
    status = fork();
    switch (status) {
        case -1 :
            perror("Creation processus");
            exit(EXIT_FAILURE);
        case 0 :
            printf("[%d] fils\n", getpid());
            break;
        default:
            printf("[%d] pere\n", getpid());
    }

    printf("[%d] fin\n", getpid());
    exit(EXIT_SUCCESS);
}
```

% ./noflush
debut [2983] pere
[2983] fin
debut [2984] fils
[2984] fin

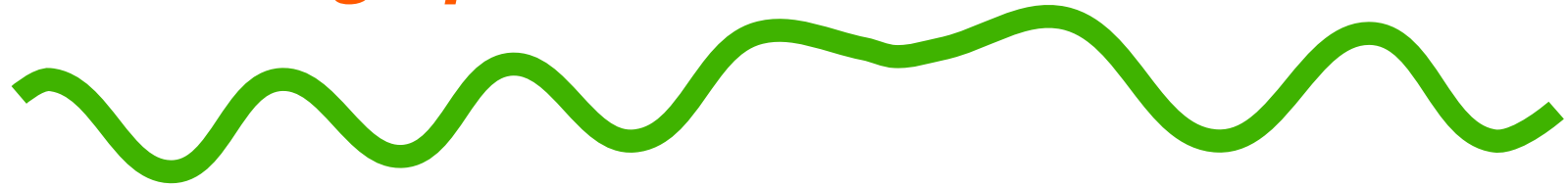
Héritage père/fils (cont'd)



~ Conséquences du partage de structures systèmes

- ~ descripteurs de fichiers dupliqués
- ~ mais entrées dans la table des fichiers ouverts partagées (`f_pos...`)

Héritage père/fils (cont'd)



```
int
main(int argc, char *argv[])
{
    int fd;
    char buf[10];

    fd = open(argv[1], O_RDWR);
    assert(fd != -1);

    read(fd, buf, 2);

    switch (fork()) {
        case -1 :
            perror("Creation processus");
            exit(EXIT_FAILURE);
        case 0 :
            write(fd, "foo", 3);
            sleep(2);
            read(fd, buf, 3); buf[3]='\0';
            printf("[%d] fils a lu '%s'\n", getpid(), buf);
            break;
        default:
            write(fd, "bar", 3);
            sleep(1);
            read(fd, buf, 3); buf[3]='\0';
            printf("[%d] pere a lu '%s'\n", getpid(), buf);
    }

    close(fd);
    exit(EXIT_SUCCESS);
}
```

fdshare.c

```
% echo _123456789_12345_ > xampl
% ./fdshare xampl
[3101] pere a lu '89_'
[3102] fils a lu '123'
% cat xampl
_1barfoo89_12345_
%
```

Héritage père/fils (cont'd)



~ Attributs non copiés

- ~ explicitement gérés par le système
- ~ numéro de processus !
- ~ numéro de processus du père !!
- ~ temps d'exécution remis à zéro
- ~ verrous sur les fichiers détenus par le père
- ~ signaux (...à suivre)

~ Coût de la copie mémoire ?

- ~ sémantique = copie
- ~ performance = pas de copie systématique
- ~ *copy on write* des pages mémoires

Terminaison des fils



~ Processus termine

- ~ appel `_exit()`
- ~ appel `exit()` : ferme les fichiers ouverts
- ~ positionne une *valeur de retour*
- ~ succès \equiv `exit(0)`

~ Processus père peut consulter cette valeur de retour

- ~ indication de la terminaison du fils : succès ou échec...

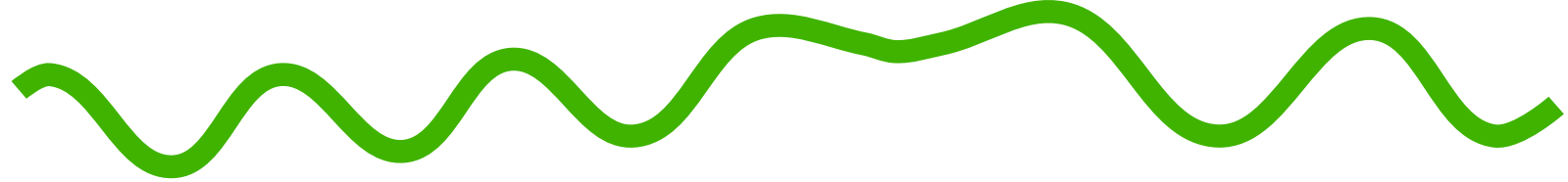
~ Attente de la terminaison d'un fils

- ~ `#include <sys/wait.h>`

```
pid_t wait(int *pstatus);
```

- ~ retourne le PID du fils
- ~ -1 en cas d'erreur (n'a pas de fils...)
- ~ bloquant si aucun fils n'a terminé
- ~ `*pstatus` renseigne sur la terminaison du fils

Terminaison des fils (cont'd)



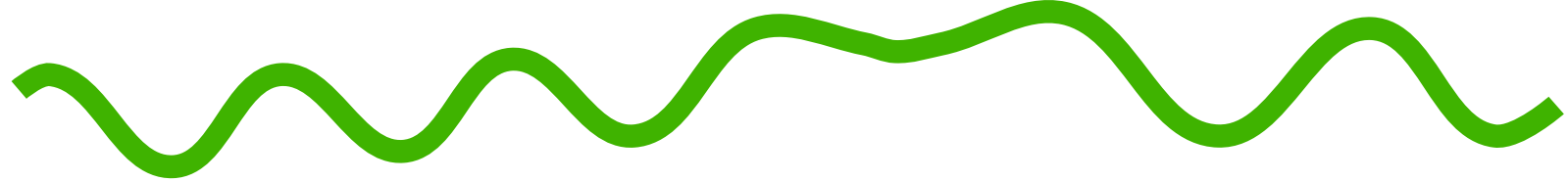
~ Renseignements concernant la terminaison d'un fils

- ~ rangées dans l'entier `status` pointé par `pstatus`
- ~ raison de la terminaison

macro	signification
<code>WIFEXITED(status)</code>	le processus a fait un <code>exit()</code>
<code>WIFSIGNALED(status)</code>	le processus a reçu un signal (...à suivre)
<code>WIFSTOPPED(status)</code>	le processus a été stoppé (...à suivre)

- ~ valeur de retour
 - ~ si `WIFEXITED(status) !`
 - ~ 8 bits de poids faible seulement
 - ~ accessible par la macro `WEXITSTATUS(status)`
- ~ numéro de signal ayant provoqué la terminaison / l'arrêt
 - ~ si `WIFSIGNALED(status) / WIFSTOPPED(status) !`
 - ~ accessible par la macro `WTERMSIG(status) /`
`WSTOPSIG(status)`

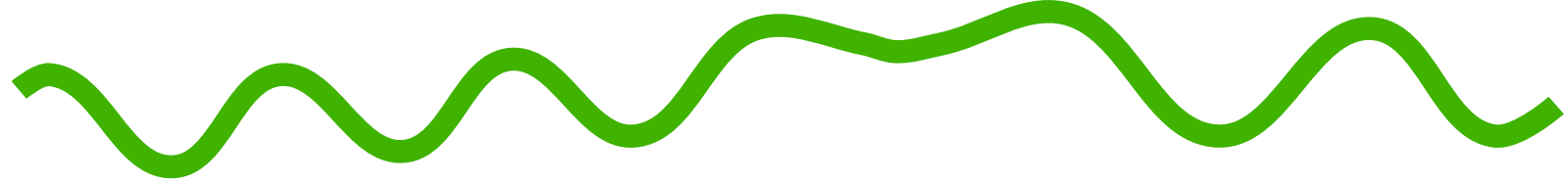
Terminaison des fils (cont'd)



```
int                                     waitexit.c
main(int argc, char *argv[])
{
    int status;                        % ./waitexit
    pid_t pid, pidz;                   [3774] pere a cree 3775
    switch (pid = fork()) {            [3775] fils eclair
        case -1 :                      [3774] mon fils 3775 a termine normlement,
            perror("Creation processus"); [3774] code de retour: 2
            exit(EXIT_FAILURE);
        case 0 :
            printf("[%d] fils eclair\n", getpid());
            exit(2);
            break;
        default:
            printf("[%d] pere a cree %d\n", getpid(), pid);
            pidz = wait(&status);
            if (WIFEXITED(status))
                printf("[%d] mon fils %d a termine normlement,\n"
                    "[%d] code de retour: %d\n",
                    getpid(), pidz,
                    getpid(), WEXITSTATUS(status));
            else
                printf("[%d] mon fils a termine anormlement\n", getpid());
    }

    exit(EXIT_SUCCESS);
}
```

Terminaison des fils (cont'd)



~ Attente d'un fils désigné

~ primitive

```
#include <sys/wait.h>
```

```
pid_t waitpid(pid_t pid, int *pstatus, int options);
```

~ pid désigne le fils à attendre

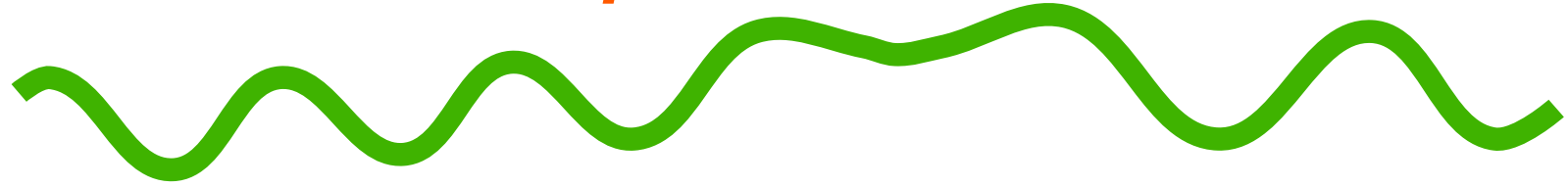
~ pid à 0 \equiv fils quelconque

~ options combinaison binaire

WNOHANG appel non bloquant

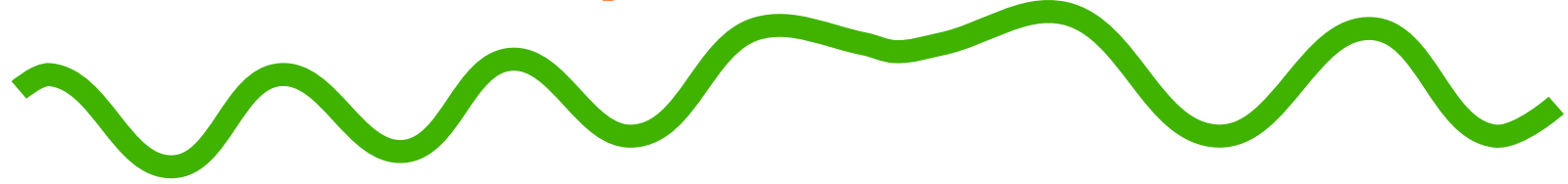
WUNTRACED processus stoppé

Processus orphelins



- ~ Terminaison d'un processus parent ne termine pas ses processus fils
 - ~ les processus fils sont orphelins
- ~ Processus initial `init` (PID 1) récupère les processus orphelins
 - ~ autre exécution de notre programme `fork.c`

Processus orphelins (cont'd)



fork.c

```
int
main()
{
    pid_t status;

    printf("[%d] Je vais engendrer\n", getpid());

    status = fork();
    switch (status) {
        case -1 :
            perror("Creation processus");
            exit(EXIT_FAILURE);
        case 0 :
            printf("[%d] Je viens de naitre\n", getpid());
            printf("[%d] Mon pere est %d\n", getpid(), getppid());
            break;
        default:
            printf("[%d] J'ai engendre\n", getpid());
            printf("[%d] Mon fils est %d\n", getpid(), status);
    }

    printf("[%d] Je termine\n", getpid());

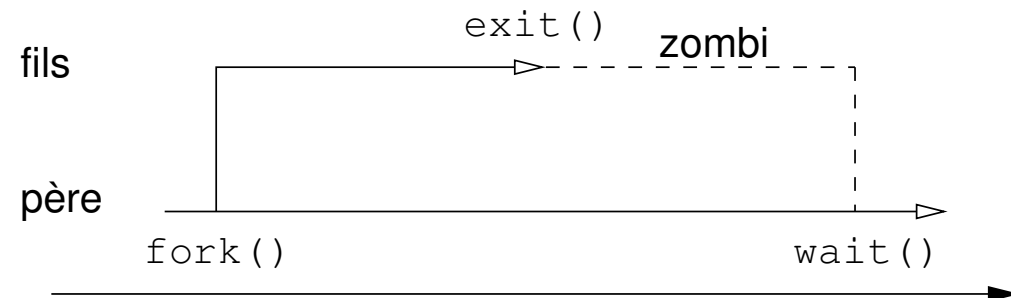
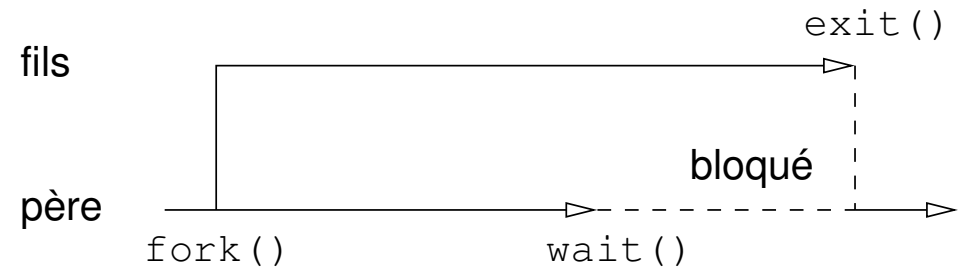
    exit(EXIT_SUCCESS);
}
```

```
% ./fork
[3852] Je vais engendrer
[3852] J'ai engendre
[3852] Mon fils est 3854
[3852] Je termine
% [3854] Je viens de naitre
[3854] Mon pere est 1
[3854] Je termine
```

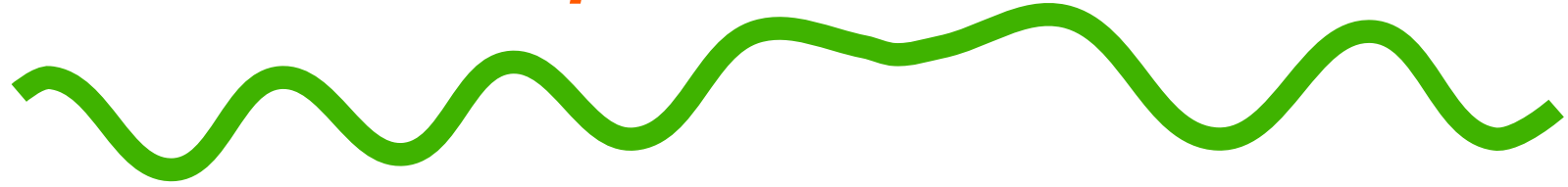
Processus zombies



- ~ Zombi = état d'un processus
 - ~ ayant terminé
 - ~ non encore réclamé par son père
- ~ Éviter les processus zombies
 - ~ système garde des informations relatives au processus
 - ~ pour les retourner à son père
 - ~ encombre la mémoire
- ~ Technique du double `fork()`
 - ~ (voir TD/TP)



Mutation de processus



~ Rappel : lancement d'un programme

- ~ 1- clonage
- ~ 2- mutation

~ Mutation = remplacement

- ~ remplacement du code à exécuter
- ~ c'est le **même** processus
- ~ on parle de recouvrement du processus

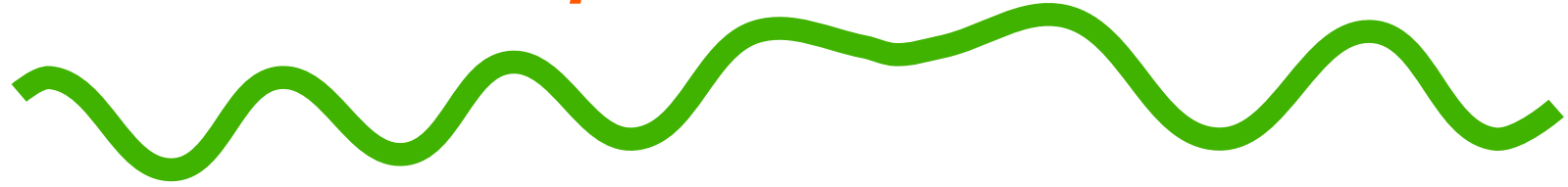
~ Nouveau programme hérite de l'environnement système

- ~ même numéro de processus PID, PPID
- ~ héritage des descripteurs de fichiers ouverts (sauf ...à suivre)
- ~ pas de remise à zéro des temps d'exécution
- ~ héritage du masque des signaux (...à suivre)
- ~ etc.

~ Famille de primitives

- ~ appel système `execve()`
- ~ fonctions de bibliothèque `exec*()`

Mutation de processus (cont'd)



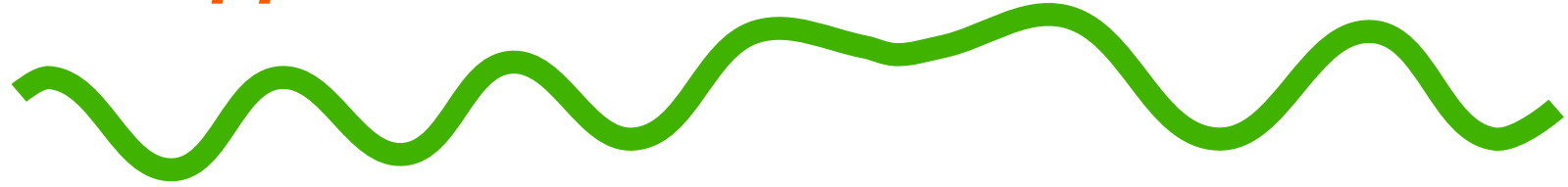
~ Appel système

~ `#include <unistd.h>`

```
int execve(const char *filename,  
           char *const argv[],  
           char *const envp[]);
```

- ~ va exécuter le programme `filename`
- ~ ne retourne pas, sauf erreur de recouvrement

Rappel sur **main()**



~ Prototype général de la fonction

~ `int main (int argc, char *argv[], char **envp);`

<code>argc</code>	nombre d'arguments +1
<code>argv[0]</code>	nom de la commande
<code>argv[1] à argv[argc-1]</code>	arguments
<code>envp</code>	variables d'environnement

Appel système `execve()`



~ Appel système

~ `int execve(const char *filename,
 char *const argv[],
 char *const envp[]);`

~ va exécuter le programme `filename`

~ qui, s'il correspond à un programme C, invoquera

```
int    main(int argc,  
         char *const argv[],  
         char *const envp[]);
```

~ De manière générale

~ le fichier désigné par `filename` est exécuté

~ avec les arguments `argv[]`

~ et les variables d'environnement `envp`

Appel système `execve()` (cont'd)

```
int
main(int argc, char *argv[], char **arge)
{
    execve(argv[1], argv+1, arge);
    perror ("recouvrement");
    exit(EXIT_SUCCESS);
}
```

`execve.c`

```
% ls r*.c
race.c  rdtty.c  readdir.c
% which ls
/bin/ls
% /bin/ls r*.c
race.c          rdtty.c          readdir.c

% ./execve /bin/ls r*.c
race.c          rdtty.c          readdir.c

% ./execve ls r*.c
recouvrement: No such file or directory

% ./execve /bin/ls y*.c
tcsh: ./execve: No match.

% ./execve /bin/ls y.c
ls: y.c: No such file or directory
```

La famille `exec*` ()



~ Plusieurs fonctions de bibliothèque

- ~ écrites au dessus de `execve()`
- ~ différents moyens de passer les paramètres : tableau, liste
- ~ spécification ou non des variables d'environnement

~ Deux spécifications de la commande à exécuter

- ~ chemin complet, `filename`
 - ~ absolu ou relatif
- ~ nom de commande, `command`
 - ~ recherchée dans les chemins de recherche, `$PATH`

```
#include <unistd.h>
```

```
int execl(const char *filename, const char *arg0, ... /*, (char *)0 */
int execv(const char *filename, char *const argv[]);
int execl(const char *filename, const char *arg0, ...
          /*, (char *)0, char *const envp[] */);
int execlp(const char *command, const char *arg0, ... /*, (char *)0 */
int execvp(const char *command, char *const argv[]);
```

La famille **exec* ()** *(cont'd)*



Illustration

exécution du programme `ls`

```
% ls race.c rdtty.c
race.c          rdtty.c
% which ls
/bin/ls
```

par différents appels d'`exec* ()` :

```
char *argv[]={ "ls", "race.c", "rdtty.c", (char *)0};

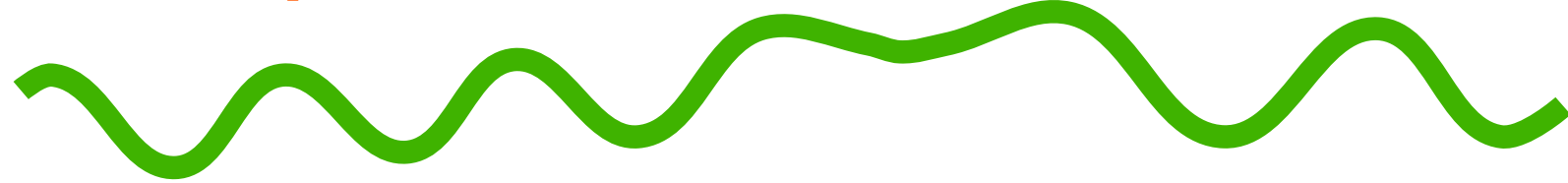
execv("/bin/ls", argv);

execvp("ls", argv);

execl("/bin/ls", "ls", "race.c", "rdtty.c", (char *)0);

execlp("ls", "ls", "race.c", "rdtty.c", (char *)0);
```

Complémentarité *fork()* et *exec*()*



~ Faire exécuter un programme par un nouveau processus

- ~ nouveau processus pour chaque commande exécutée
- ~ ce que fait le shell

```
int
main()
{
    printf("Je suis %5d de pere %5d\n",
           getpid(), getppid());

    exit(EXIT_SUCCESS);
}
```

```
pid.c  % ./pid
Je suis  7851 de pere  717
% ./pid
Je suis  7852 de pere  717
% echo $$
717
```

- ~ erreur dans le programme n'affecte pas le shell
- ~ changement d'attributs dans le programme n'affectent pas le shell
- ~ (sauf commandes internes : par exemple `cd...`)

~ Réalisation en deux étapes

- ~ 1- clonage `fork()`
- ~ 2- mutation `exec*()`

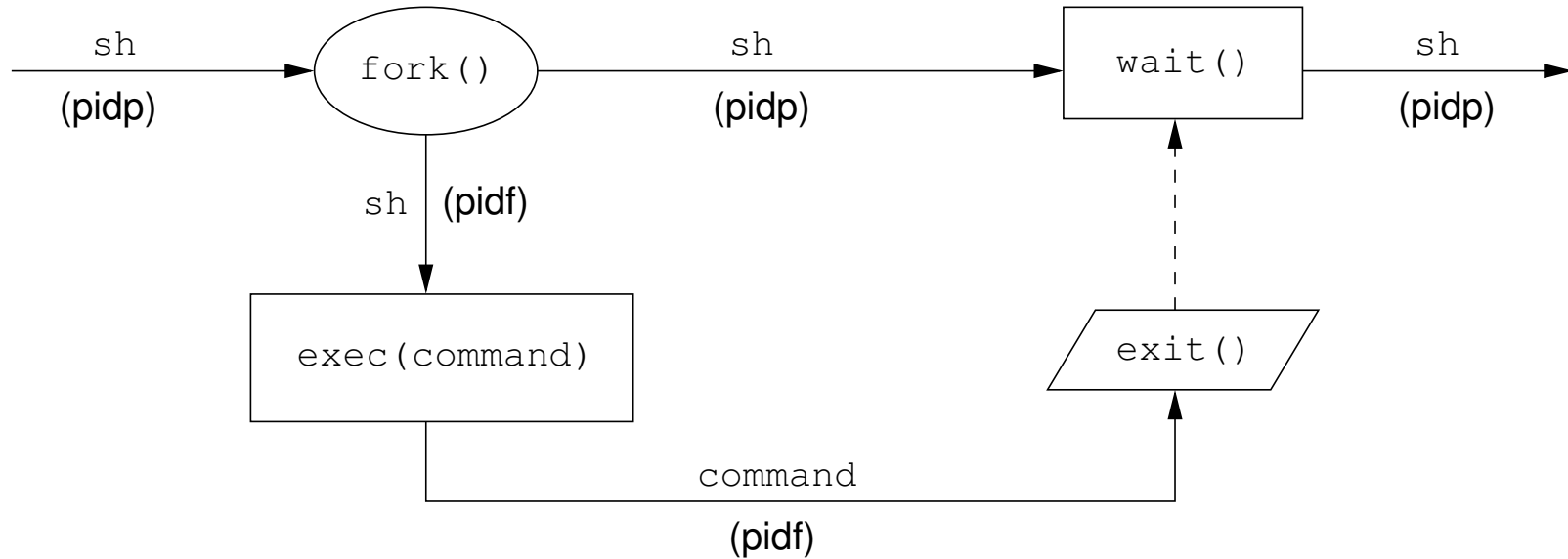
~ Souplesse de par la combinaison deux primitives

- ~ réalisation d'opérations entre le `fork()` et l'`exec*()`
- ~ exemple : redirection des entrées/sorties standard (...à suivre)

L'exemple du shell

Exécution d'une commande `command` par le shell `sh`

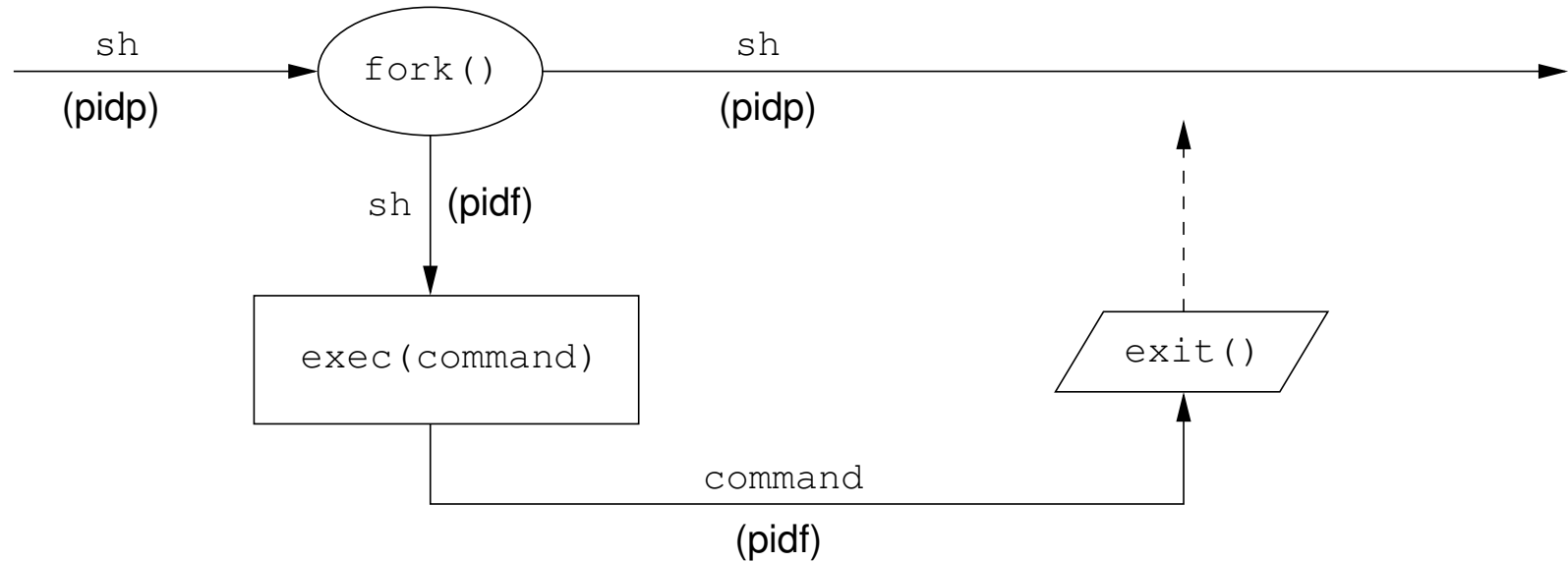
% `command`
%



4 primitives `fork()`, `exec()`, `exit()`, et `wait()`

L'exemple du shell (cont'd)

Exécution en arrière plan d'une commande `command` par le shell `sh`
`% command &`

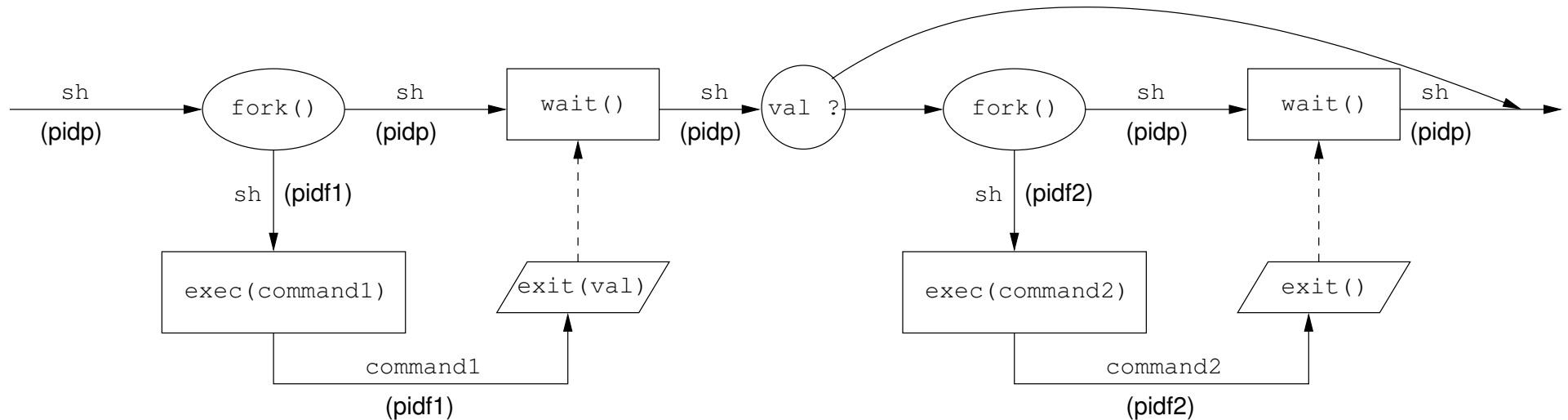


Le shell n'attend plus la terminaison de son fils

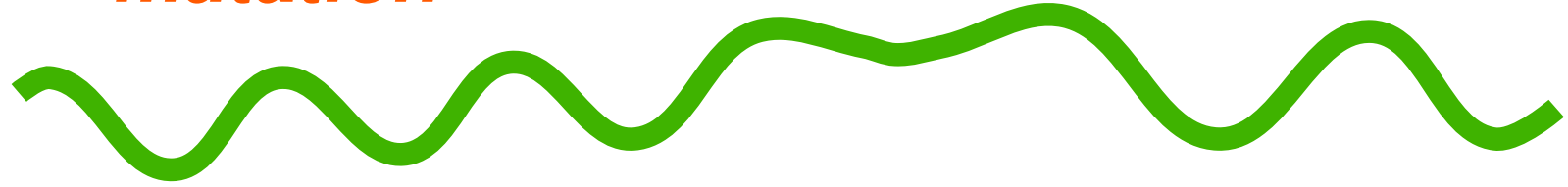
L'exemple du shell (cont'd)

Exécution conditionnelle

- ~ d'une commande `command2`
- ~ suivant le résultat d'une commande `command1`
 - % `command1 && command2`
 - %
 - ou
 - % `command1 || command2`
 - %

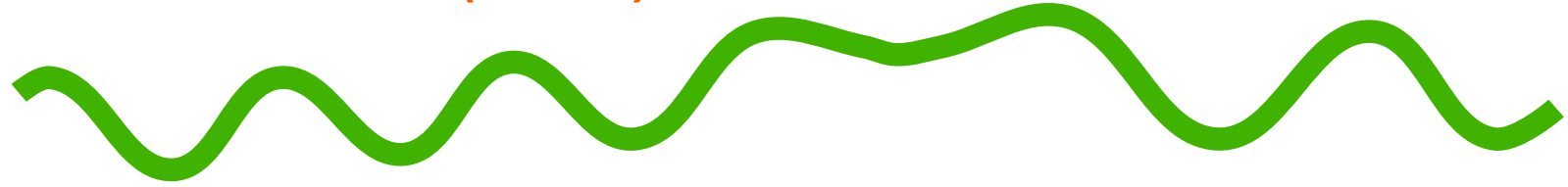


Héritage des descripteurs lors de mutation



- ~ Descripteurs de fichiers sont hérités par le nouveau programme
 - ~ largement utilisé par le shell pour assurer les redirections
 - ~ exemple :
`% cmd > out`
 - ~ le shell associe le descripteur 1, `STDOUT_FILENO` à un fichier `out` (voir `dup ()` ...à suivre)
 - ~ `exec ("cmd", ...)`
 - ~ le programme `cmd` écrit sur `STDOUT_FILENO`
 - ~ ... qui référence le fichier `out`

Héritage des descripteurs lors de mutation (cont'd)



- ~ Sauf si positionnement de l'option *close on exec* sur le descripteur
- ~ positionnement de `FD_CLOEXEC` par un appel à `fcntl()`

```
int                                     cloexec.c
main(int argc, char *argv[])
{
    fcntl(STDOUT_FILENO, F_SETFD,
          fcntl(STDOUT_FILENO, F_GETFD, 0) | FD_CLOEXEC);

    execvp(argv[1], argv+1);
    perror ("execvp");

    exit(EXIT_SUCCESS);
}
```

% ./cloexec true

% ./cloexec ls

ls: write error: Bad file descriptor

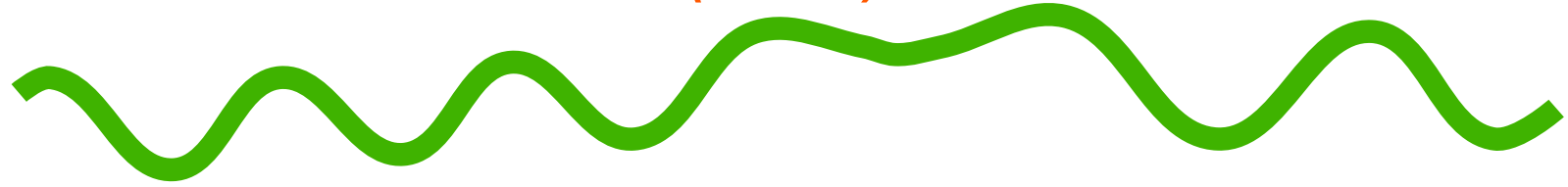
Redirections des entrées/sorties



- ~ Associer un descripteur de fichier donné à un fichier
 - ~ exemple associer le descripteur de fichier donné 1, `STDOUT_FILENO` au fichier `out`
 - ~ les écritures avec le descripteur `STDOUT_FILENO` se feront dans le fichier `out`
- ~ Primitive POSIX
 - ~ `#include <unistd.h>`

`int dup(int fildes);`
 - ~ recherche le **plus petit descripteur** disponible
 - ~ en fait un synonyme du descripteur `fildes`
 - ~ les deux descripteurs partagent la même entrée dans la table des fichiers ouverts du système
 - ~ écrire (/lire) dans ce *plus petit descripteur* revient à écrire (/lire) dans le fichier référencé par `fildes`
- ~ Ce *plus petit descripteur* correspond à un fichier qui vient d'être fermé...

Redirections des entrées/sorties (cont'd)



~ Primitive POSIX `dup2()`

- ~ explicitation du descripteur synonyme créé !

- ~ `#include <unistd.h>`

```
int dup2(int fildes, int fildes2);
```

- ~ force `fildes2` à devenir un synonyme de `fildes`

- ~ ferme le descripteur `fildes2` préalablement si nécessaire

~ Utilisations ultérieures de `fildes2`

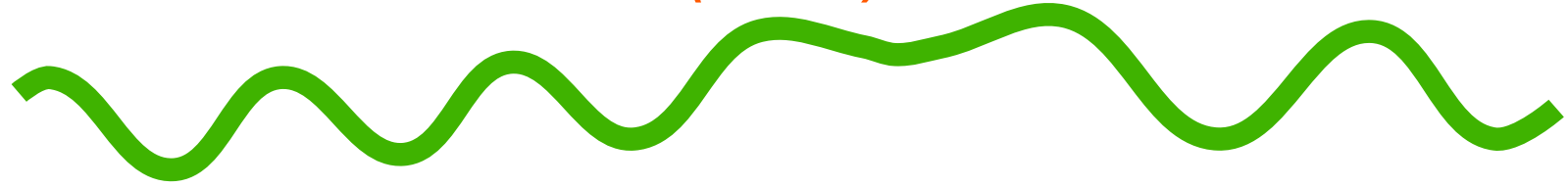
- ~ référencent le fichier identifié par `fildes`

~ Exemples

```
static void                                dup2pr.c (début)
print_inode(const char *str, int fd)
{
    struct stat st;

    fstat(fd, &st);
    fprintf(stderr, "\t%s : inode %d\n", str, st.st_ino);
}
```

Redirections des entrées/sorties (cont'd)



```
int                                     dup2pr.c (suite et fin)
main(int argc, char *argv[])
{
    int fdin, fdout;

    fdin  = open(argv[1], O_RDONLY);
    fdout = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC, S_666);

    fprintf(stderr, "Avant dup2() :\n");
    print_inode("fdin", fdin);
    print_inode("fdout", fdout);
    print_inode("stdin", STDIN_FILENO);
    print_inode("stdout", STDOUT_FILENO);

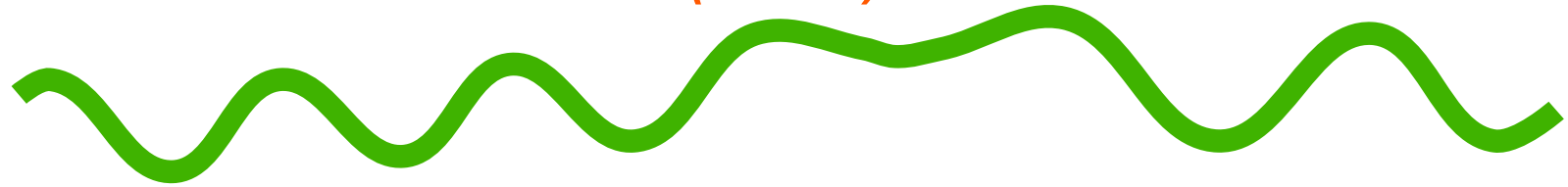
    dup2(fdin,  STDIN_FILENO);
    dup2(fdout, STDOUT_FILENO);

    fprintf(stderr, "Après dup2() :\n");
    print_inode("fdin", fdin);
    print_inode("fdout", fdout);
    print_inode("stdin", STDIN_FILENO);
    print_inode("stdout", STDOUT_FILENO);

    exit(EXIT_SUCCESS);
}
```

```
% ls -li foo bar
ls: bar: No such file or directory
1083956 foo
% ./dup2pr foo bar
Avant dup2() :
    fdin : inode 1083956
    fdout : inode 1083620
    stdin : inode 51125508
    stdout : inode 51125508
Après dup2() :
    fdin : inode 1083956
    fdout : inode 1083620
    stdin : inode 1083956
    stdout : inode 1083620
```

Redirections des entrées/sorties (cont'd)



```
static void
cat()
{
    unsigned char buffer[BSIZE];
    int nread;

    while((nread=read(STDIN_FILENO, buffer, BSIZE)))
        write(STDOUT_FILENO, buffer, nread);
}

int
main(int argc, char *argv[])
{
    int fdin, fdout;

    fdin  = open(argv[1], O_RDONLY);
    fdout = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC, S_666);

    dup2(fdin,  STDIN_FILENO);
    dup2(fdout, STDOUT_FILENO);

    close(fdin);
    close(fdout);

    cat();

    close(STDIN_FILENO);
    close(STDOUT_FILENO);

    exit(EXIT_SUCCESS);
}
```

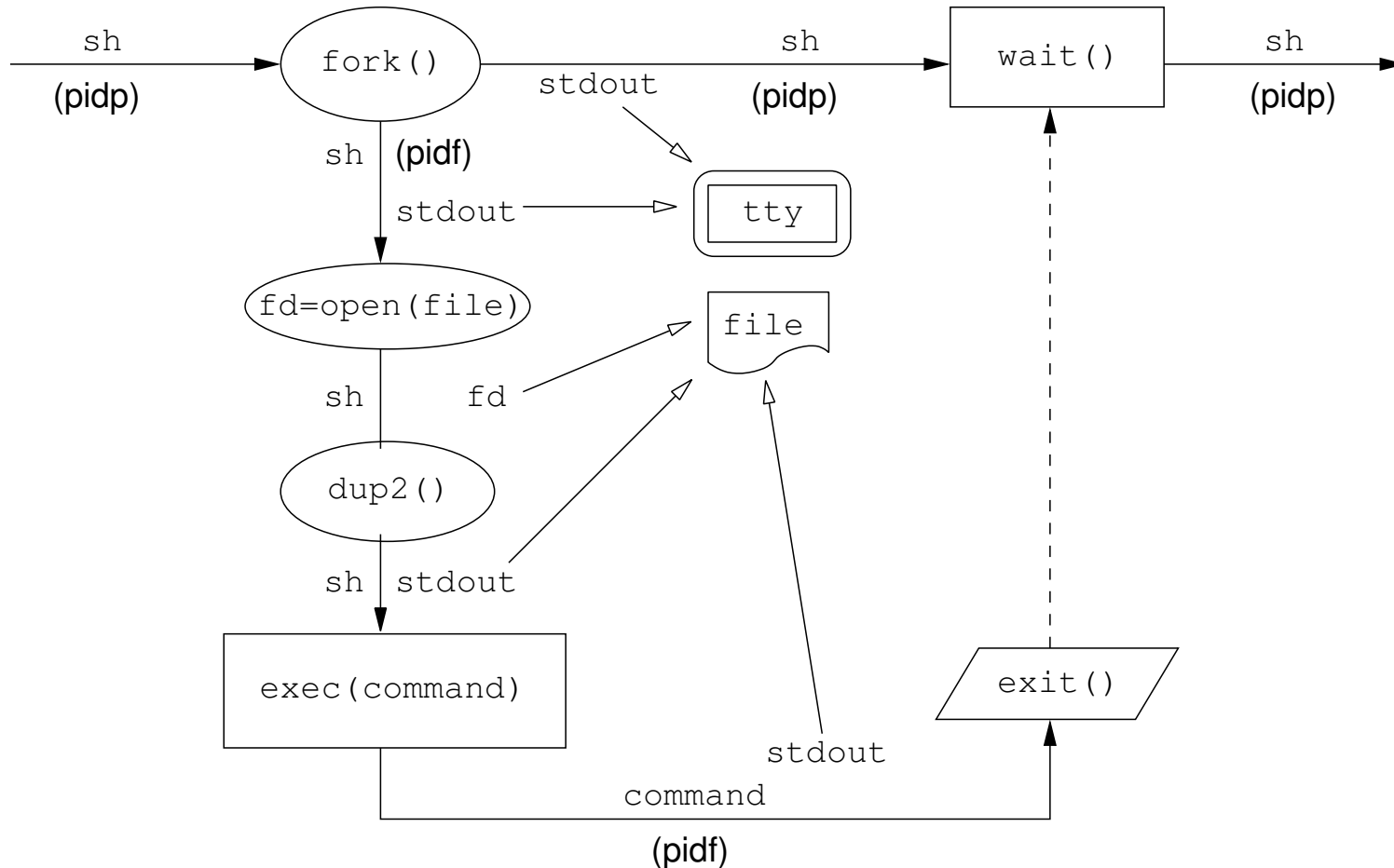
dup2cat.c

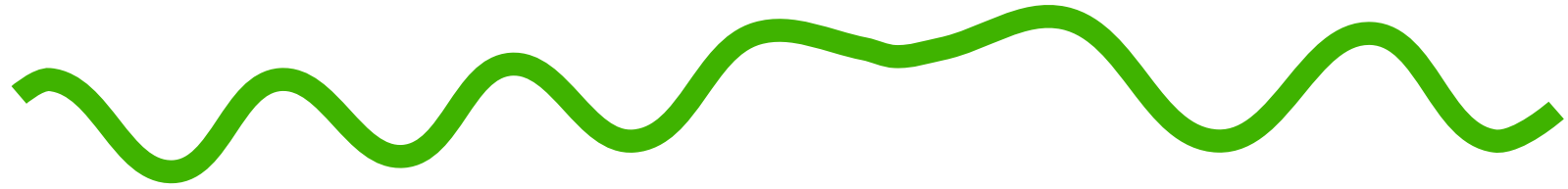
```
% cat bar
% cat foo
azerty
qwerty
dvorak
% ./dup2cat foo bar
% cat bar
azerty
qwerty
dvorak
```

Redirection : l'exemple du shell

Exécution d'une commande `command` redirigée sur un fichier `file`

```
% command > file  
%
```





Éléments d'ordonnancement des processus

Ordonnancement = virtualisation



~ Multiprogrammation

- ~ plusieurs processus *prêts*
- ~ un seul processeur

~ Virtualisation du processeur

- ~ chaque processus détient le processeur tour à tour

~ Ordonnancement

- ~ définition et mise en œuvre de ce « tour à tour »
- ~ choisir le processus *élu*

~ Cœur du système d'exploitation

- ~ bien que tout algorithme d'ordonnancement puisse convenir...
- ~ voir ASE, l'an prochain

~ Ce que nous allons voir ici

- ~ très grandes lignes de quelques algorithmes d'ordonnancement
- ~ norme POSIX, système Unix
- ~ paramétrage par l'utilisateur

Objectif d'un algorithme d'ordonnement



~ Divers et multiples !

- ~ selon les systèmes d'exploitation
- ~ systèmes d'exploitation spécialisés
 - ~ temps-réel, multimédia, traitement par lots, interactifs...
- ~ systèmes d'exploitation généralistes

~ Équité

- ~ attribuer à chaque processus un temps processeur équitable

~ Réactif

- ~ répondre rapidement aux requêtes

~ Débit

- ~ nombre de traitements par unité de temps

~ Performance

- ~ faire en sorte que le processeur soit occupé en permanence
- ~ faire en sorte que toutes les parties du système soient occupées

~ Garanties

- ~ respecter les échéances

Préemptibilité / quantum



~ Préemptibilité des processus

- ~ ordonnancement préemptif
 - ~ peut interrompre l'exécution d'un processus en cours
 - ~ possibilité de réquisition du processeur
- ~ ordonnancement non préemptif
 - ~ changement de contexte à la fin du processus, ou
 - ~ changement de contexte volontaire du processus en cours
(`yield()`)

~ Quantum

- ~ unité de temps processeur attribué au processus élu
- ~ réquisition du processeur à la fin du quantum
- ~ influence de la durée du quantum
 - ~ petit : surcoût des changements de contexte
 - ~ grand : mauvaise réactivité

Priorité des processus



~ Propriété d'un processus

- ~ niveau de priorité
- ~ dénote l'importance relative des processus

~ Numérotation des priorités

- ~ attention, souvent les processus les plus prioritaires ont un numéro de priorité plus faible !

~ Priorité ajustable

- ~ `#include <unistd.h>`

```
int nice(int incr);
```

- ~ augmente le valeur de priorité du processus
- ~ donc diminue sa priorité !
- ~ courtoisie envers les autres utilisateurs
- ~ `incr` négatif autorisé pour le superutilisateur

~ Commande `nice`

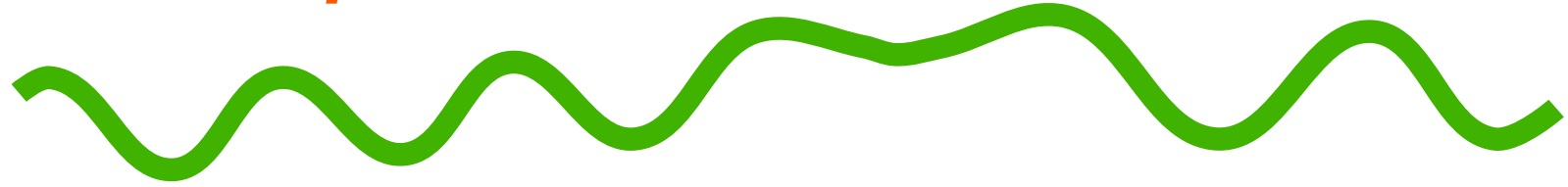
- ~ `nice [-n increment] command [argument...]`

Politiques d'ordonnancement



- ~ Multitudes de politiques possibles
- ~ Trois politiques définies par POSIX
 - ~ premier arrivé, premier servi : *FIFO, first-in, first-out*
 - ~ tourniquet : *RR, round robin*
 - ~ autre : en général l'ordonnancement Unix traditionnel
- ~ **FIFO**
 - ~ non préemptif
 - ~ changement de contexte quand le processus rend la main ou termine
 - ~ POSIX : un nombre fini de niveaux de priorité FIFO
 - ~ processus préempté par un processus plus prioritaire
- ~ **Tourniquet**
 - ~ processeur alloué successivement à chacun des processus
 - ~ expiration du quantum : réquisition du processeur
 - ~ processus placé en queue de la file d'attente
 - ~ POSIX : un nombre fini de niveaux de priorité RR
 - ~ élection des seuls processus de plus forte priorité

Politiques d'ordonnancement (cont'd)



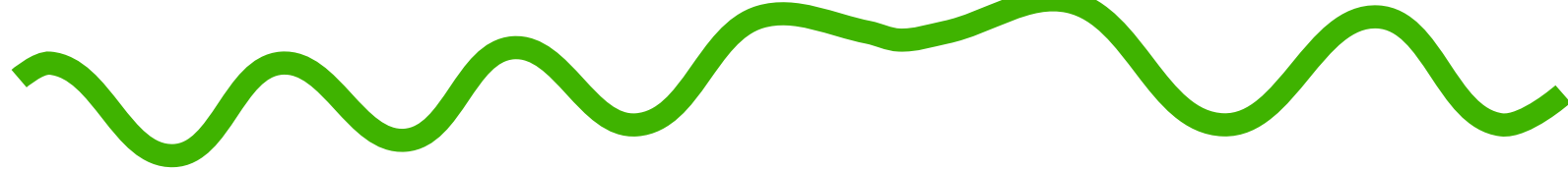
~ Politique Unix classique

- ~ quantum, à priorité dynamique
- ~ priorité de base (dite statique) + priorité dynamique
- ~ principe d'extinction des priorités : évite les famine
- ~ priorité dynamique diminue au fur et à mesure que le processus consomme du temps processeur

~ Ordonnancement POSIX

- ~ des processus gérés par les trois politiques
- ~ si un processus associé à FIFO
- ~ sinon si un processus associé à tourniquet
- ~ sinon les processus associés à l'autre politique

Primitives d'ordonnancement



~ Récupérer les paramètres d'ordonnancement

- ~ politique d'ordonnancement, une valeur parmi SCHED_FIFO, SCHED_RR, et SCHED_OTHER

```
#include <sched.h>
```

```
int sched_getscheduler(pid_t pid);
```

- ~ **structure** struct sched_param

```
struct sched_param {  
    int sched_priority;  
    ..  
};
```

- ~ paramètre de l'ordonnancement

```
#include <sched.h>
```

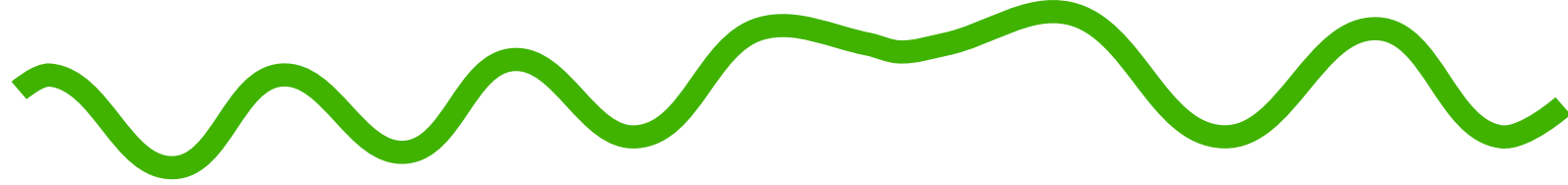
```
int sched_getparam(pid_t pid, struct sched_param *param);
```

~ Positionner les paramètres

```
#include <sched.h>
```

```
int sched_setscheduler(pid_t pid, int policy,  
    const struct sched_param *param)
```

Primitives d'ordonnancement (cont'd)



~ Intervalles de priorité

~ `#include <sched.h>`

```
int sched_get_priority_max(int policy);  
int sched_get_priority_min(int policy);
```

~ Quantum

~ dans le seul cas de la politique tourniquet

`#include <sched.h>`

```
int sched_rr_get_interval(pid_t pid, struct timespec *interval);
```

~ **structure** `struct timespec`

```
struct timespec {  
    time_t tv_sec;           /* Seconds. */  
    long int tv_nsec;        /* Nanoseconds. */  
};
```

~ Utilisation effective

~ faible...

~ sauf FIFO / tourniquet : urgence de processus temps-réel « mou »

~ multimédia sur une système d'exploitation généraliste