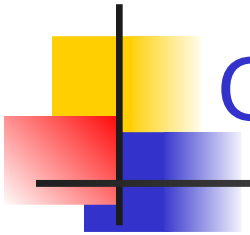
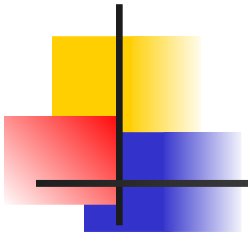


Luc Courtrai

CONCURRENCE

Université de Bretagne SUD
UFR SSI - Departement MIS



Concurrence

Plan

Notion de processus

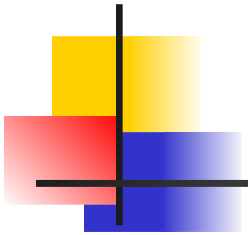
Les appels système UNIX

La synchronisation entre processus

La communication entre processus

Les processus légers : les threads JAVA

Les Posix threads



Concurrence/appels système Unix

Les appels système UNIX: création de processus

- `pid_t fork(void)`

crée un nouveau processus qui s'exécute en parallèle avec son processus père.

Le processus fils est une copie conforme du processus père. Le nouveau processus récupère l'environnement du processus père. Le programmeur doit spécifier après le fork le traitement du père et celui du fils (c 'est le même programme qui s 'exécute 2 fois).

La fonction retourne pour le père le numéro du processus fils et retourne 0 pour le processus fils. C'est cette valeur de retour qui distingue les deux processus. (En cas d'échec , la fonction retourne une valeur négative -1)

Concurrence/appels système Unix

Les appels système UNIX: fork

```
#include<unistd.h>
pid_t   pid;
pid = fork ();
if (pid > 0) {
    /* Processus père       */
} else if (pid == 0) {
    /* Processus fils       */
} else {
    /* Traitement d'erreur */
}
```

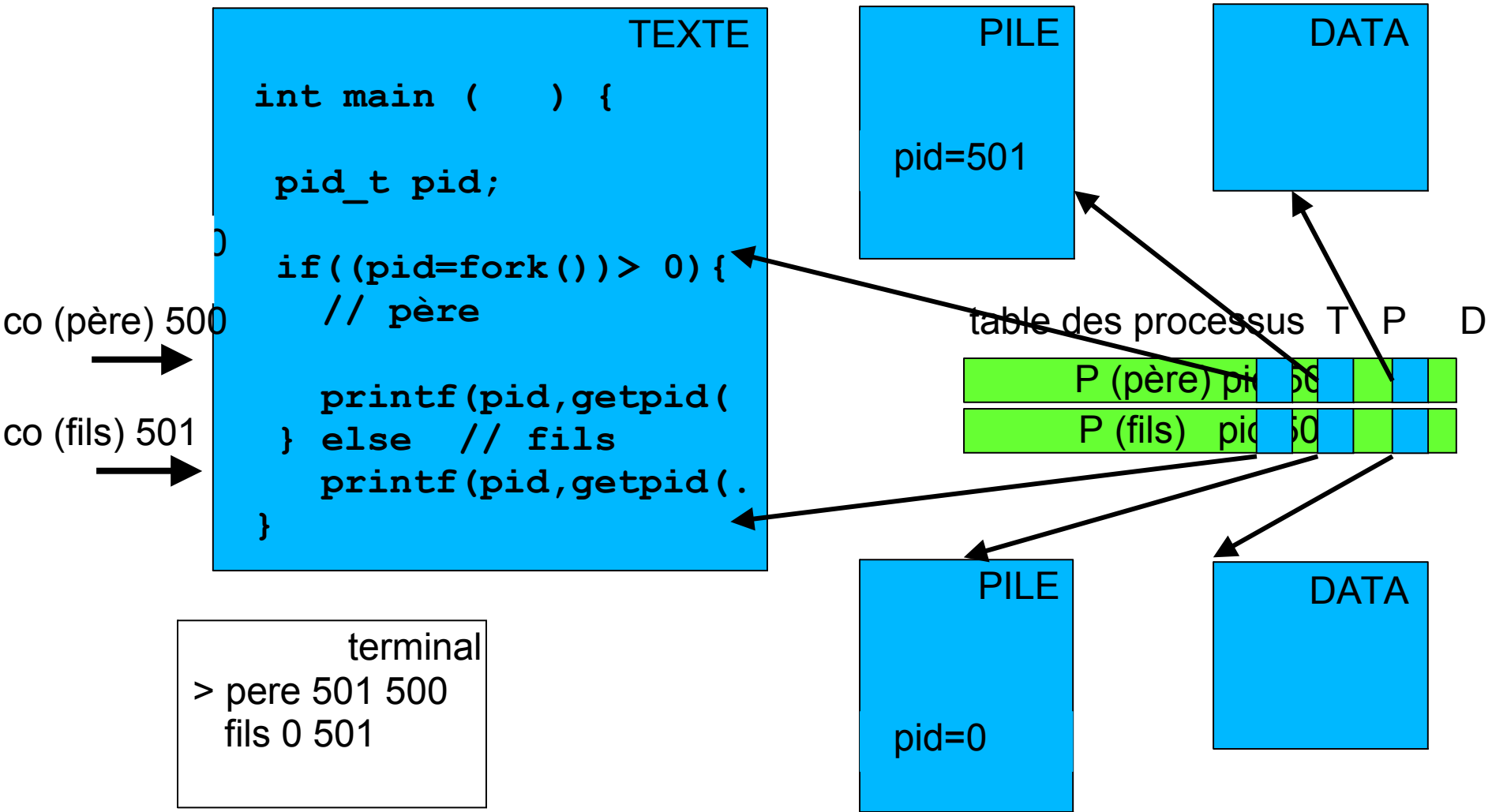
Concurrence/appels système Unix

Les appels système UNIX: fork

```
int main ( int argc , char **argv ) {

 pid_t pid;

 if((pid=fork()) > 0) {// processus pere

   printf("pid %d pere %d\n",pid,getpid ();
 } else  // processus fils si pas d 'echec
   printf("pid %d fils %d\n",pid getpid(); );
}
```

Concurrence/appels système Unix

Les appels système UNIX: fork



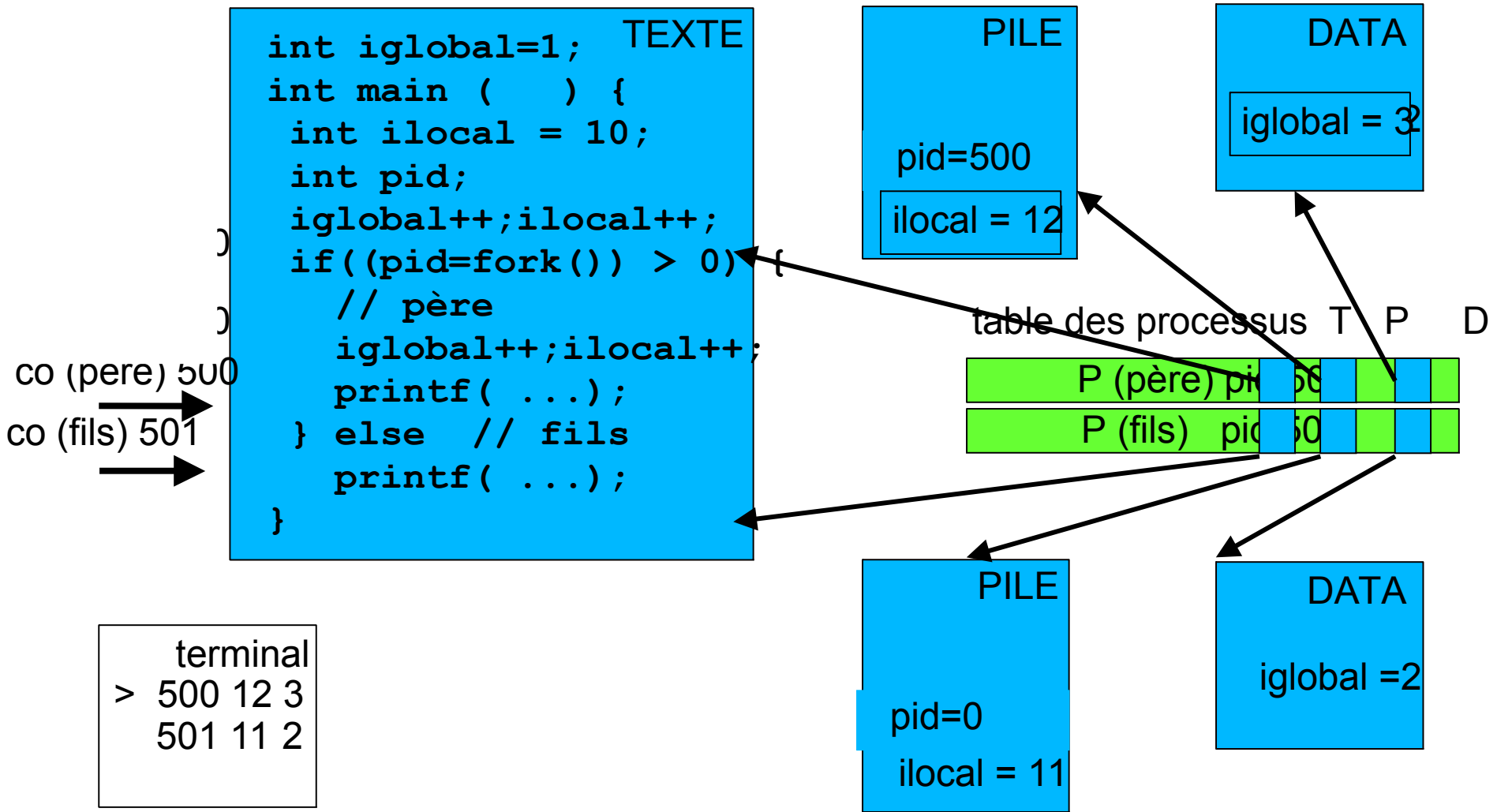
Concurrence/appels système Unix

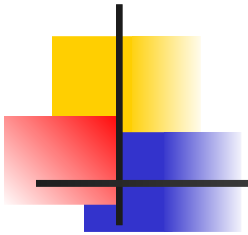
Les appels système UNIX: fork

```
int iglobal=1;
int main ( int argc , char **argv ) {
 int ilocal = 10;
 pid_t pid;
 iglobal++;ilocal++;
 if((pid=fork()) > 0) {// processus pere
   iglobal++;ilocal++;
    printf("pere %d %d %d %\n",
                getpid(),iglobal,ilocal);
 } else  // processus fils si pas d 'echec
    printf("fils %d %d %d  \n",
                 getpid();iglobal,ilocal);
}
```

Concurrence/appels système Unix

Les appels système UNIX: fork





Concurrence/appels système Unix

Les appels système UNIX: wait

```
#include<sys/wait.h>
pid_t wait(int *status)
```

permet au processus père d'attendre la terminaison d ’un processus fils.

La fonction retourne le numéro de processus fils terminé. Le paramètre de la fonction est l'adresse d'un entier qui contiendra l’état de sortie du processus (valeur de exit(Val)).

wait(NULL) pas d'état de fin

Concurrence/appels système Unix

Les appels système UNIX: wait

WIFEXITED(status)

non nul si le fils s'est terminé normalement

WEXITSTATUS(status)

donne le code de retour tel qu'il a été mentionné dans l'appel exit() ou dans le return de la routine main. Cette macro ne peut être évaluée que si WIFEXITED est non nul.

WIFSIGNALED(status)

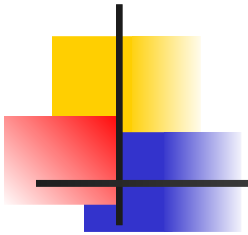
indique que le fils s'est terminé à cause d'un signal non intercepté.

WTERMSIG(status) donne le numéro du signal qui a causé la fin du fils.

Concurrence/appels système Unix

Les appels système UNIX: wait

```
#include<unistd.h>
#include<sys/wait.h>
int main ( ) {
  int pid;int message;
  if((pid=fork())==0) //processus fils
      exit(1);
  // processus pere
  pid=wait(&message);//synchro sur la fin du
  fils
  if (WIFEXITED(message))
    printf("code %u",WEXITSTATUS(message));
}

//    terminal >code 1
```

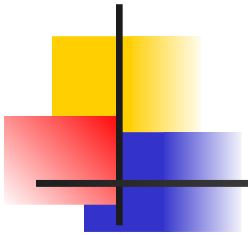


Concurrence/appels système Unix

Les appels système UNIX: `sched_yield`

```
int sched_yield(void);
```

- **Un processus peut volontairement libérer le processeur sans se bloquer en appelant sched_yield.**
- **Note: Si le processus est le seul avec une priorité élevée, il continuera son exécution après un appel à sched_yield.**



Concurrence/appels système Unix

Les appels système UNIX: fork, wait, exit, sched_yield

exemple : multif.c

Affiche en parallèle sur la sortie standard les contenus des fichiers dont les noms sont en argument de la commande. Chaque fichier est traité par un processus différent.

La sortie standard est désynchronisée pour illustrer le parallélisme.

Concurrence/appels système Unix

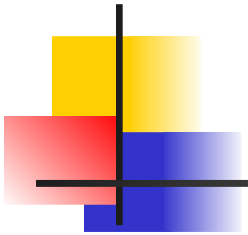
Les appels système UNIX: fork, wait, exit, sched_yield

```
#include <unistd.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <stdlib.h>
int main (int argc,char **argv) {
  int pid, i;
  if (argc <= 1) exit(1);
  fcntl(1,F_SETFL,O_NONBLOCK); // pas de cache
  for (i=1;i<argc;i++) {  // les arguments
    // creation d 'un fils
    if ((pid = fork()) == 0) {
```

Concurrence/appels système Unix

Les appels système UNIX: fork, wait, exit, sched_yield

```
    // processus fils
    int fic,c;
    if ((fic=open(argv[i],O_RDONLY))<0) exit(2);
    while (read(fic,&c,1) > 0) {
      write(1,&c,1);
      sched_yield(); //rend la main
    } // while
    exit(0); // sortie et synchro avec le pere
  } // for
  // pere  (tous les fils font un exit())
  for (i=1;i<argc;i++)  // père
        pid =wait(NULL); // sans status
  } // main
```



Concurrence/appels système Unix

Les appels système UNIX: exec

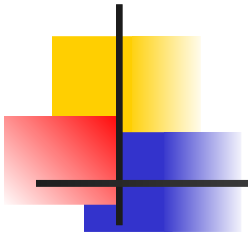
Les appels système de type exec permet à un processus de modifier son programme binaire en exécution. Il exécute alors un autre programme. Ses segments texte, pile et data sont alors modifiés.

```
int execl (const char *path, const char
  *arg, ...);
   //    args … terminée par NULL
```

Concurrence/appels système Unix

Les appels système UNIX: exec

```
#include <unistd.h>

int main (int argc,char **argv) {
  execl("/bin/pwd","pwd",NULL);
  // code normalement jamais exécuté
  fprintf(stderr, "echec au exec\n");
  exit(1);
}
// arg "pwd" -> argv[0]
```



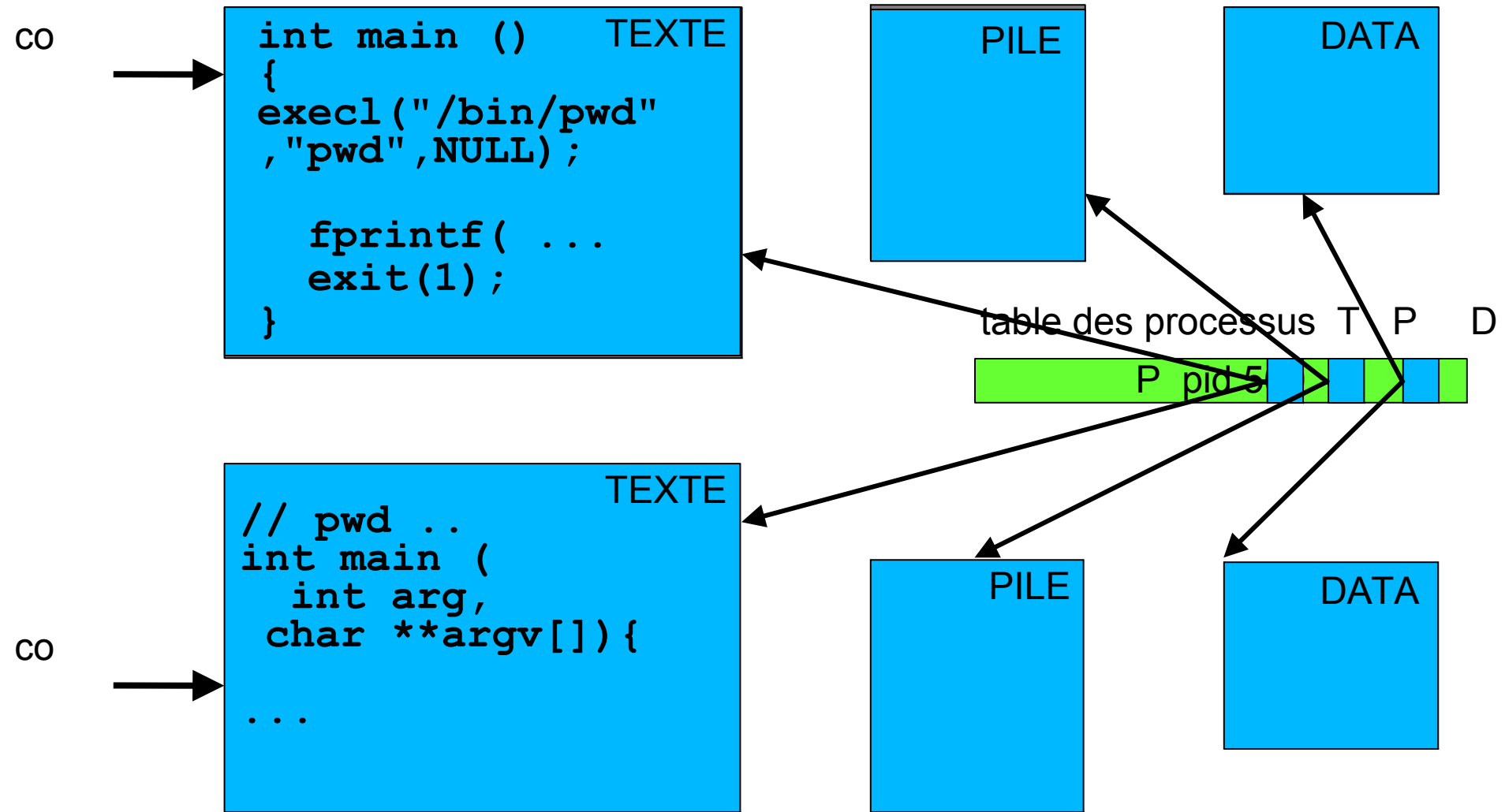
Concurrence/appels système Unix

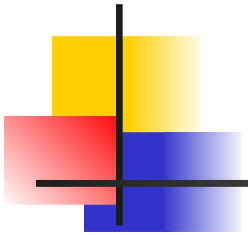
Les appels système UNIX: exec

```
int execl (const char *path, const char
 *arg, ...);
int execle (const char *path, const char
 *arg , ...,  char* const envp[]);  // e
 environment
int execv (const char *path, char *const
 argv[]); // v  argV
int execlp (const char *file, const char
 *arg, ...) // p path  recherche dans les path
int execvp (const char *file, char *const
 argv[]); // v + p
```

Concurrence/appels système Unix

Les appels système UNIX: exec





Concurrence/appels système Unix

Les appels système UNIX: sleep

```
#include <unistd.h>
unsigned int sleep (unsigned int nb_sec);
```

sleep() endort le processus jusqu'à ce que nb_sec secondes se soient écoulées

```
unsigned int usleep(unsigned int

  nb_microsec);
```

usleep() endort le processus jusqu'à ce que nb_microsec micro secondes se soient écoulées

Concurrence/appels système Unix

La commande nice et renice UNIX:

nice [-n ajustement] [-ajustemand] command arg...

lance un processus en spécifiant sa priorité (défaut celle du père)

prio -20 +19 (root priorité négative)

-n val(ajout val la priorité actuelle)

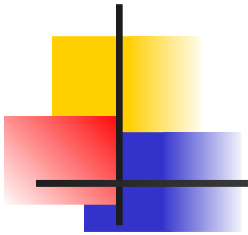
renice priority [pid] [-u user]

modifie la priorité d 'un processus déjà lancé. Un utilisateur ne peut que l 'augmenter.

Concurrence/appels système Unix

La commande nice et renice UNIX:

```
>    ps -l
  F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY
    TIME CMD
000 S   261  5137  5134  0  69   0    -   620 rt_sig pts/1    00:00:00 tcsh
000 T   261  5238  5137  0  68   0    -   268 do_sig pts/1    00:00:00 a.out
000 R   261  5239  5137  0  73   0    -   683 -      pts/1    00:00:00 ps
>  renice +2 5238
>    ps -l
F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY
    TIME CMD
000 S   261  5137  5134  0  76   0    -   620 rt_sig pts/1    00:00:00 tcsh
000 T   261  5238  5137  0  63   2    -   268 do_sig pts/1    00:00:00 a.out
000 R   261  5241  5137  0  70   0    -   683 -      pts/1    00:00:00 ps
```



Concurrence/appels système Unix

Les appels système UNIX: **get/set priority**

int getpriority(int which, int who);

int setpriority(int which, int who, int prio);

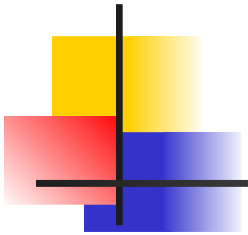
which

PRIO_PROCESS,PRIO_PGRP,PRIO_USER,

who

pid, groupe de processus, user

prio [-20,19]



Concurrence/appels système Unix

Les appels système UNIX: PID PPID USER GRP

pid_t getpid(void);

getpid retourne l'ID du processus actif

pid_t getppid(void)

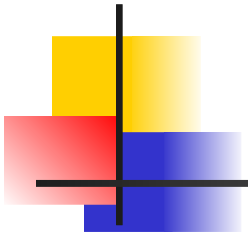
getppid retourne le PID du processus parent de celui en cours

uid_t getuid(void);

getuid retourne l'UID réel du processus en cours.

gid_t getgid(void);

getgid retourne le GID réel du processus en cours.



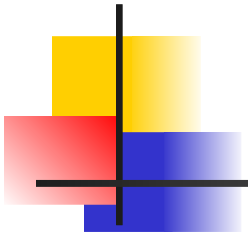
Concurrence/appels système Unix

Les appels système UNIX: session groupe de processus

pid_t setsid(void);

crée une nouvelle session si le processus appelant n'est pas un leader de groupe. Le processus appelant devient le leader du nouveau groupe, et n'a pas de terminal de contrôle. L'ID du groupe de processus et l'ID de session du processus appelant sont fixés à la valeur de PID du processus en cours. Le processus en cours sera le seul dans son groupe et sa session.

Le leader d'un groupe est le processus dont le PID est égal à l'ID du groupe. Pour s'assurer que setsid réussira, il faut effectuer un fork(), suivi d'un exit() pour le père, et le fils appellera setsid() (leader d ’un seul groupe).



Concurrence/appels système Unix

Les appels système UNIX: session groupe de processus

int setpgid(pid_t pid, pid_t pgid);

setpgid fixe à pgid l'ID du groupe de processus auquel appartient le processus mentionné par pid. Si pid vaut zéro, le PID du processus en cours est utilisé. Si pgidvaut zéro, le PID du processus indiqué par pid est utilisé.

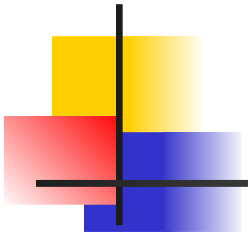
pid_t getpgid(pid_t pid);

getpgid renvoie l'ID du groupe de processus auquel appartient le processus indiqué par pid. Si pid vaut zéro, le PID du processus en cours est utilisé.

Concurrence/appels système Unix

Les appels système UNIX: session groupe de processus

```
#include <sys/types.h>
#include <unistd.h>
int main(){
  printf("%u%u\n", getpid(),getpgrp());// 501 500
  if (fork() ==0) {
    printf("%u%u\n",getpid(),getpgrp());//502 500
    // cree une nouvelle session
    pid_t sid= setsid();
    printf("%u\n", sid); // 502
    printf("%u%u\n",getpid(),getpgrp());//502 502
  }
  exit(0);
}
```



Concurrence/appels système Unix

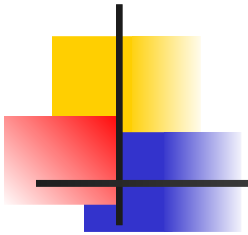
Les appels système UNIX: **kill**

int kill(pid_t pid, int sig);

Si pid est positif, le signal sig est envoyé au processus pid.

Si pid vaut zéro, alors le signal sig est envoyé à tous les processus appartenant au même groupe que le processus appelant.

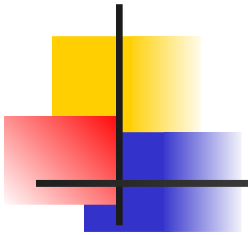
Si pid est négatif, alors le signal sig est envoyé à tous les processus du groupe -pid.



Concurrence/appels système Unix

Les appels système UNIX: kill

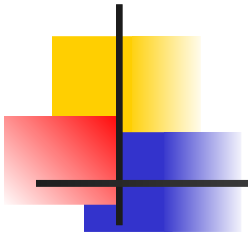
```
#define SIGINT      2    /*  CTRL C
#define SIGQUIT     3    /* (*) quit, CTRL \ */
#define SIGKILL     9    /* kill   kill -9 */
#define SIGUSR1    30    /* user defined signal 1 */
#define SIGUSR2    31    /* user defined signal 2 */
```



Concurrence/appels système Unix

Les appels système UNIX: waitpid

```
pid_t waitpid(pid_t pid,int *status,int
  options);
pid
  -1   attendre la fin de n'importe quel  fils.

       C'est  le même comportement que wait.
  > 0  attendre la fin du processus numéro pid.
  0    attendre la fin de n'importe quel
       processus fils du même groupe que
       l'appelant.
  < -1 attendre  la  fin  de n'importe quel
       processus fils appartenant à un groupe de
       processus d'ID pid.
```



Concurrence/appels système Unix

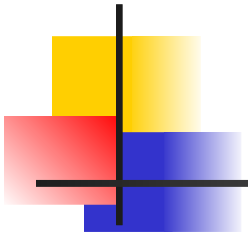
Les appels système UNIX: waitpid

option :

WNOHANG **ne pas bloquer si aucun fils ne s'est terminé (test).**

WUNTRACED

Changement d ’état pour un fils. Pas forcément arrêté recevoir l'information concernant également les fils bloqués si on ne l'a pas encore reçue.



Concurrence/appels système Unix

Les appels système UNIX: waitpid

traitement sur le status

WIFEXITED WEXITSTATUS WIFSIGNALED WTERMSIG

WIFSTOPPED(status)

indique que le fils est actuellement arrêté. Cette macro n'a de sens que si l'on a effectué l'appel avec l'option WUN TRACED

WSTOPSIG(status)

donne le nombre de signaux qui ont causé l'arrêt du fils. Cette macro ne peut être évaluée que si WIFSTOPPED est

non nul.

Concurrence/appels système Unix

Les appels système UNIX: waitpid

```
int main () {
 int pid;int status;
 if((pid=fork())==0) { dcodeFIls(); exit(0)};
 while(1) { //code du père
  waitpid(pid,&status,WUNTRACED); // modif etat

  if (WIFEXITED(status))
    {printf("%u\n",WEXITSTATUS(status));exit(0)};
  if (WIFSIGNALED(status))
    {printf("%u\n ",WTERMSIG(status));exit(1);
  if (WIFSTOPPED(status))
    printf( "%u\n ",WSTOPSIG(status));
 }
}
```

Concurrence/appels système Unix

Les appels système UNIX: EUSER EGRP

Un processus peut changer de propriétaire USER ou de groupe GRP au cours de son exécution (effectif EUSER EGRP) pour avoir de droits particulier

uid_t geteuid(void);

geteuid retourne l'UID effectif du processus en cours.

gid_t getegid(void);

getegid retourne le GID effectif du processus en cours.

int seteuid(uid_t euid);

fixe l'UID effectif du processus

int setegid(gid_t egid);

fixe l'GID effectif du processus

Concurrence/appels système Unix

Les appels système UNIX: EUSER EGRP

```
// su.c
#include <stdio.h>
#include <unistd.h>
#include <pwd.h>
int main(int argc, char *argv[]) {
  struct passwd *ps;
  char * user = argv[1];
  if (argc == 1) {
      fprintf(stderr,"usage su [-] user");
      exit(1);
  }
  if (argc == 3) user = argv[2];
```

Concurrence/appels système Unix

Les appels système UNIX: EUSER EGRP

```
// determine uid a partir du nom symbolique
if(!(ps=getpwnam(user)))    {
    fprintf(stderr,"utilisateur inconnu sur\
          cette machine  %s \n",user);
    exit (1);
}
if((ps->pw_uid)<=100) {// securite
    printf("Acces interdit  %s \n",user);
    exit(1);
}
```

Concurrence/appels système Unix

Les appels système UNIX: EUSER EGRP

```
// change le gid effectif
if (setegid(ps->pw_gid) != 0) {
    fprintf(stderr,"error setgid");
    exit (1);
}
// change l'uid effectif
if (seteuid(ps->pw_uid) != 0) {
    fprintf(stderr,"error setuid");
    exit (1);
}
```

Les appels système UNIX: EUSER EGRP

```
  if (argc == 2)
    execl("/bin/csh","/bin/csh",NULL);
  else
    execl("/bin/csh","/bin/csh","-f",NULL);
  fprintf(stderr ,"echec du execl\n");
}//main
```

Concurrence/appels système Unix

Les appels système UNIX: EUSER EGRP

> **`gcc -o su su.c`**

Il faut que le programme appartient à root et que le seteuid et setegid soient mis.

> **`chmod 6755 su`**

Il faut être root pour exécuter les appels

setuid(ps->pw_uid) et setgid(ps->pw_gid))

> **`dupont>  su durant`**

`durant>`

Après cela un utilisateur peut exécuter le programme su, il est root effectif avec le chmod et peut donc effectuer les setuid