Le System Call di Unix/Linux

11 Febbraio 2005

Dual mode

- **Il S.O. esegue in modo *kernel*.**
- **Ogni programma utente esegue in *user mode***
- **Quando un programma utente tenta l'esecuzione di una istruzione privilegiata, viene generato un *trap*.**

Come puo` un programma di utente richiedere l'esecuzione di istruzioni privilegiate (ad esempio, I/O)?

➔ **chiamata a *system call***

System Call

Per ottenere l'esecuzione di istruzioni privilegiate, un programma di utente deve chiamare una System Call:

1. **invio di un'interruzione software al S.O.**
2. **Salvataggio dello stato (PC, registri, bit di modo, etc.) del programma chiamante e trasferimento del controllo al S.O.**
3. **Il S.O. esegue in modo kernel l'operazione richiesta**
4. **al termine dell'operazione, il controllo ritorna al programma chiamante (ritorno al modo user)**

System Call


Programma utente
system call: read( )
Interrupt SW
(salvataggio dello stato del programma utente)
ripristino dello stato del programma utente
User mode
Kernel mode
Routine di gestione dell'interruzione
Esecuzione dell'operazione read

System Call

Interfaccia di programmazione (API) per:

- **Gestione dei processi/thread**
- **Sincronizzazione**
- **Comunicazione**
- **Gestione del file system**
- **…**

Gestione dei processi Unix

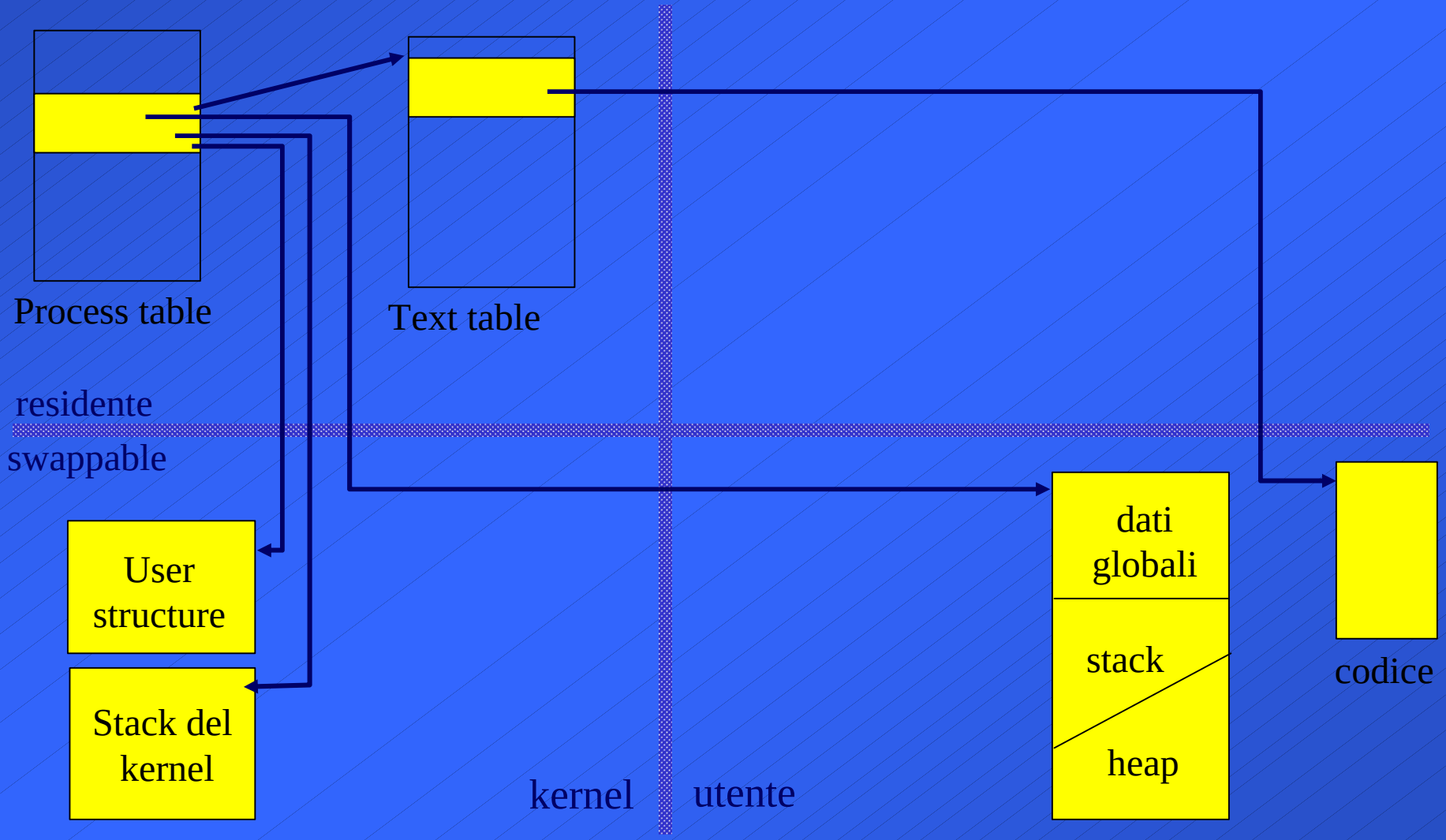
Processi Unix

Caratteristiche del processo Unix:

- **processo pesante con codice *rientrante*:**
  - » **dati non condivisi**
  - » **codice condivisibile con altri processi**
- **funzionamento dual mode:**
  - » **processi di utente (modo *user*)**
  - » **processi di sistema (modo *kernel*)**

☞ **diverse potenzialità e, in particolare, diversa visibilità della memoria.**

Immagine di un processo Unix



System Call per la gestione di Processi

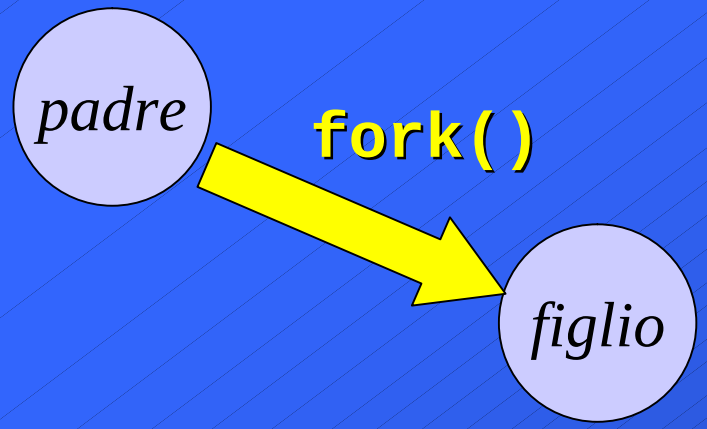
Chiamate di sistema per:

- creazione di processi: **`fork()`**
- sostituzione di codice e dati: **`exec..()`**
- terminazione: **`exit()`**
- sospensione in attesa della terminazione di figli: **`wait()`**

N.B. Le system call di Unix sono attivabili attraverso funzioni di librerie C standard: **`fork()`**, **`exec()`**, etc. sono quindi funzioni di libreria che chiamano le system call corrispondenti.

Creazione di processi: fork()

- La funzione **`fork()`** consente a un processo di generare un processo figlio:
  - padre e figlio **condividono** lo stesso **codice**
  - il figlio *eredita* una **copia dei dati** (di utente e di kernel) del padre



fork()

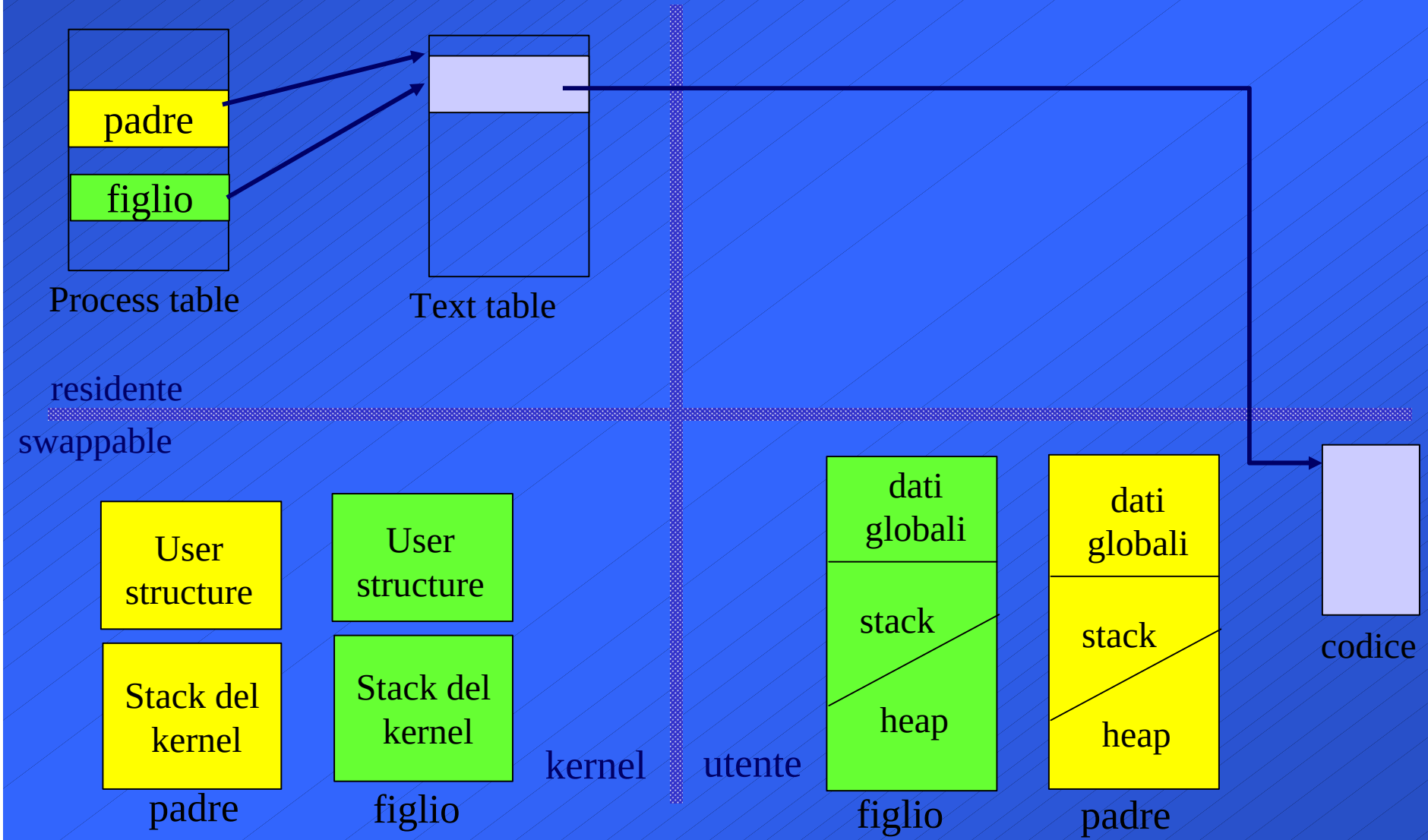
```
int fork(void);
```

- la **`fork`** non richiede parametri
- restituisce un intero che:
  - » per il processo creato vale **0**
  - » per il processo padre è un valore **positivo** che rappresenta il **PID** del processo figlio
  - » è un valore **negativo** in caso di errore (la creazione non è andata a buon fine)

Effetti della fork()

- Allocazione di una **nuova process structure** nella process table associata al processo figlio e sua inizializzazione
- Allocazione di una **nuova user structure** nella quale viene copiata la user structure del padre
- Allocazione dei **segmenti di dati e stack** del figlio nei quali vengono copiati dati e stack del padre
- Aggiornamento della **text structure** del codice eseguito (condiviso col padre): incremento del contatore dei processi, etc.

Effetti della fork()



fork(): esempio

```
#include <stdio.h>
main()
{ int pid;
    pid=fork();
    if (pid==0)
    { /* codice figlio */
   printf("Sono il figlio ! (pid: %d)\n", getpid());
   }
    else if (pid>0)
      { /* codice padre */
     printf("Sono il padre: pid di mio figlio: %d\n", pid);
    ....
      }
   else printf("Creazione fallita!");
 }
```

NB: la system call **`getpid`** ritorna il pid del processo che la chiama.

Relazione Padre-Figlio in Unix

Dopo una `fork()`:

- **concorrenza:**
  - » padre e figlio procedono in parallelo
- **lo spazio degli indirizzi è duplicato :**
  - » ogni variabile del figlio è inizializzata con il valore assegnatole dal padre prima della fork()
- la **user structure** è duplicata **:**
  - » le risorse allocate al padre (ad esempio, i file aperti) prima della generazione sono condivise con i figli
  - » le informazioni per la gestione dei segnali sono le stesse per padre e figlio (associazioni segnali-handler)
  - » il figlio nasce con lo stesso Program Counter del padre: la prima istruzione eseguita dal figlio è quella che segue immediatamente la fork().

Terminazione di processi

Un processo può terminare:

- **involontariamente**:
  - » tentativi di azioni illegali
  - » interruzione mediante segnale
  - ☞ salvataggio dell'immagine nel file **`core`**
- **volontariamente**:
  - » chiamata alla funzione **`exit()`**
  - » esecuzione dell'ultima istruzione

exit( )

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

- la funzione **exit** prevede un parametro (**status**) mediante il quale il processo che termina può comunicare al padre informazioni sul suo stato di terminazione (ad es., l'esito della sua esecuzione).
- è sempre una chiamata senza ritorno

`exit()`

Effetti di una exit():

- chiusura dei file aperti non condivisi
- terminazione del processo:
  - » se il processo che termina ha **figli in esecuzione**, il processo **`init`** adotta i figli dopo la terminazione del padre (nella process structure di ogni figlio al pid del processo padre viene assegnato il valore 1 (pid di init)
  - » se il processo **termina prima che il padre ne rilevi lo stato di terminazione** (con la system call **`wait`**), il processo passa nello stato **zombie**.

NB. Quando termina un processo adottato da **`init`**, il processo init rileva automaticamente il suo stato di terminazione -> i processi figli di **`init`** non diventano mai zombie !

wait

- Lo stato di terminazione può essere rilevato dal processo padre, mediante la system call **`wait()`:**

```
int wait(int *status);
```

  - **il parametro `status` è l'indirizzo della variabile in cui viene memorizzato lo stato di terminazione del figlio**
  - **il risultato prodotto dalla `wait` è il pid del processo terminato, oppure un codice di errore (<0)**

`wait`

Effetti della system call `wait(&status)`:

- il processo che la chiama può avere figli in esecuzione:
  - » se tutti i figli non sono ancora terminati, il processo si sospende in attesa della terminazione del primo di essi
  - » se almeno un figlio è già terminato ed il suo stato non è stato ancora rilevato (cioè è in stato ***zombie***), la **`wait`** ritorna immediatamente con il suo stato di terminazione (nella variabile **`status`**)
  - » se non esiste neanche un figlio, la **`wait`** non è sospensiva e ritorna un codice di errore (valore ritornato < 0).

`wait`

Rilevazione dello stato: in caso di terminazione di un figlio, la variabile status raccoglie il suo stato di terminazione; nell'ipotesi che lo stato sia un intero a 16 bit:

- se il byte meno significativo di status è zero, il più significativo rappresenta lo stato di terminazione (terminazione volontaria, ad esempio con **`exit`**)
- in caso contrario, il byte meno significativo di status descrive il segnale che ha terminato il figlio (terminazione involontaria).

wait & exit: esempio

```
main()
{int pid, status;
pid=fork();
if (pid==0)
   {printf("figlio");
   exit(0);
}
else{ pid=wait(&status);
        printf("terminato processo figlio n.%d", pid);
        if ((char)status==0)
        printf("term. volontaria con stato %d", status>>8);
        else printf("terminazione involontaria  per segnale
                        %d\n", (char)status);
   }

}
```

wait

- **Rilevazione dello stato:** e` necessario conoscere la rappresentazione di **`status:`**
  - lo standard Posix.1 prevede delle macro (definite nell'header file **`<sys/wait.h>`** per l'analisi dello stato di terminazione. In particolare:
    - **`WIFEXITED(status):`** restituisce vero, se il processo figlio è terminato volontariamente: in questo caso la macro **`WEXITSTATUS(status)`** restituisce lo stato di terminazione.
    - **`WIFSIGNALED(status):`** restituisce vero, se il processo figlio è terminato involontariamente
    - in questo caso la macro **`WTERMSIG(status)`** restituisce il numero dell'interruzione SW che ha causato la terminazione.

wait & exit: esempio

```
#include <sys/wait.h>
main()
{int pid, status;
pid=fork();
if (pid==0)
   {printf("figlio");
   exit(0);
}
else{ pid=wait(&status);
        if (WIFEXITED(status))
                printf("Terminazione volontaria di %d con
                        stato %d\n", pid, WEXITSTATUS(status));
        else if (WIFSIGNALED(status))
                printf("terminazione involontaria  per segnale
                        %d\n", WTERMSIG(status));  }

}
```

System call exec

- Mediante la fork i processi padre e figlio condividono il codice e lavorano su aree dati duplicate: in Unix è possibile differenziare il codice dei due processi mediante una system call della famiglia **exec**:

  `execl(), execle(), execlp(), execv(), execve(), execvp()..`

Effetto principale di una exec:

vengono sostituiti **codice** e **dati** del processo che chiama la system call, con codice e dati di un programma specificato come parametro della system call

execl

```
int execl(char *pathname, char *arg0, ..
          char *argN, (char*)0);
```

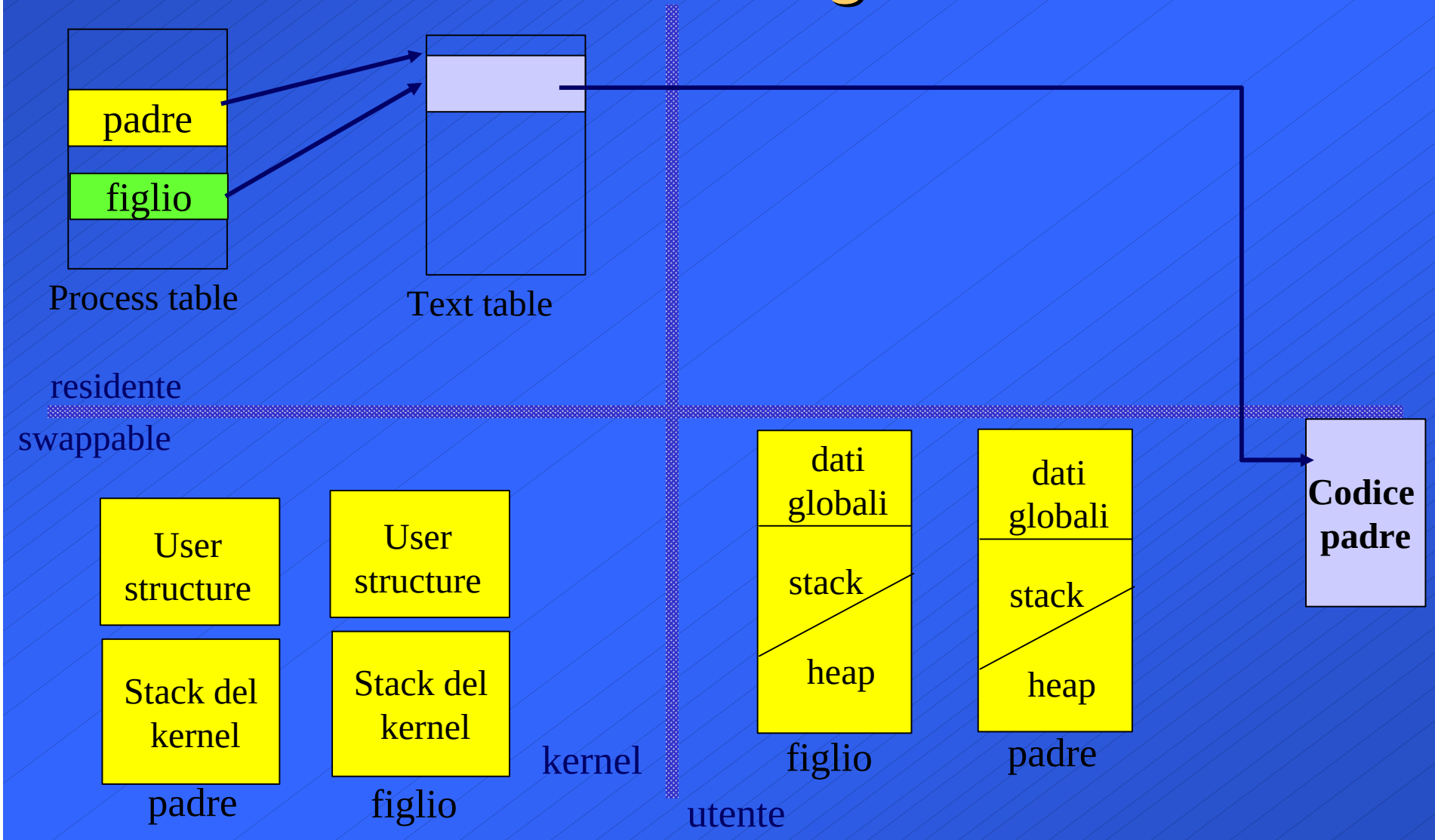
- **pathname** è il nome (assoluto o relativo) dell'eseguibile da caricare
- **arg0** è il nome del programma (argv[0])
- **arg1,..argN** sono gli argomenti da passare al programma
- **(char *)0** è il puntatore nullo che termina la lista.

Esempio: execl

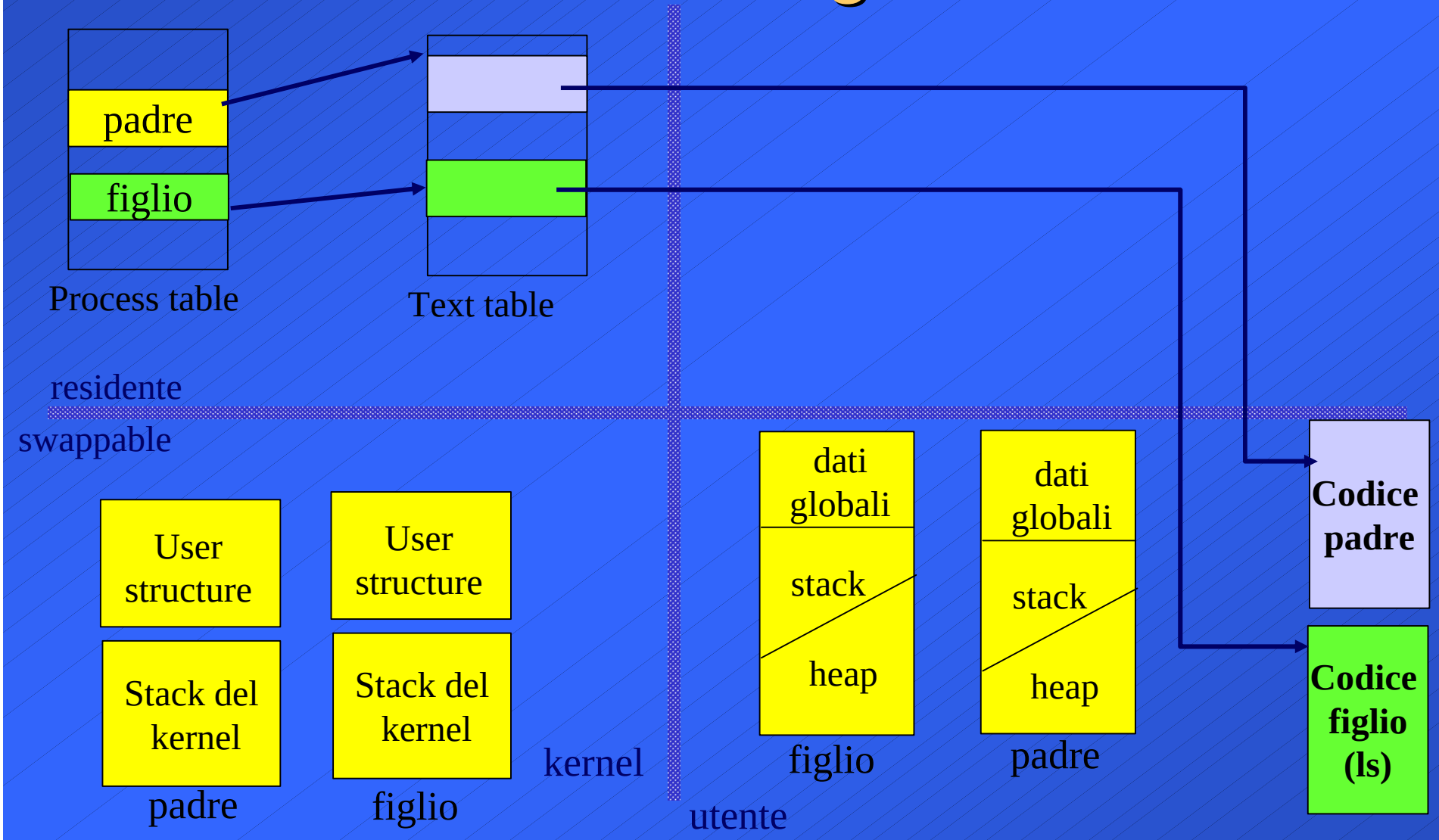
```
main()
{int pid, status;
pid=fork();
if (pid==0)
   {execl(“/bin/ls”, “ls”,”-l”,”pippo”,(char *)0);
    printf(“exec fallita!\n”);
    exit(1);
}
else if (pid >0)
   { pid=wait(&status);
     /* gestione dello stato.. */
   }
else printf(“fork fallita!);
}
```

NB: in caso di successo, l’ exec è una chiamata senza ritorno

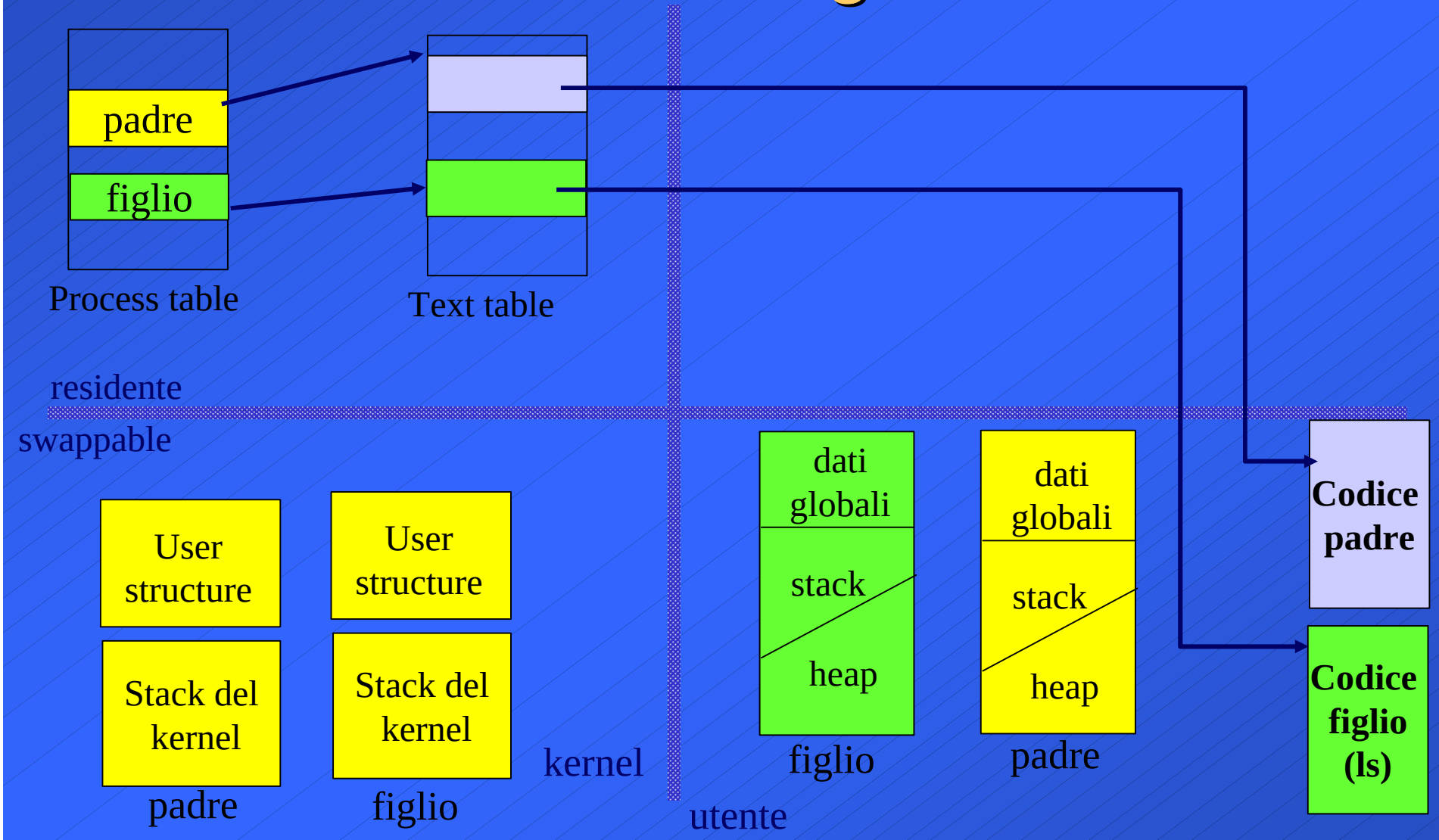
Esempio: effetti della execl() sull'immagine



Esempio: effetti della execl() sull'immagine



Esempio: effetti della execl() sull'immagine



Effetti dell'exec

Il processo dopo l'exec:

- mantiene la stessa **process structure (salvo le informazioni relative al codice)**:
  - » stesso pid
  - » stesso pid del padre
  - » ...
- ha **codice**, **dati globali**, **stack** e **heap** nuovi
- riferisce una nuova **text structure**
- mantiene **user area** (**a parte PC e informazioni legate al codice**) e stack del kernel:
  - » mantiene le stesse risorse (es: file aperti)
  - » mantiene lo stesso *environment* (a meno che non sia `execle` o `execve`)

System call exec . .

Varianti dell'exec: a seconda del suffisso

- **l** : gli argomenti da passare al programma da caricare vengono specificati mediante una LISTA di parametri (terminata da NULL) (es. execl())
- **p** : il nome del file eseguibile specificato come argomento della system call viene ricercato nel PATH contenuto nell'ambiente del processo (es. execlp())
- **v** : gli argomenti da passare al programma da caricare vengono specificati mediante un VETTORE di parametri(es. execv())
- **e**: la system call riceve anche un vettore (envp[]) che rimpiazza l'environment (path, direttorio corrente, etc.) del processo chiamante (es. execle())

Esempio: execve

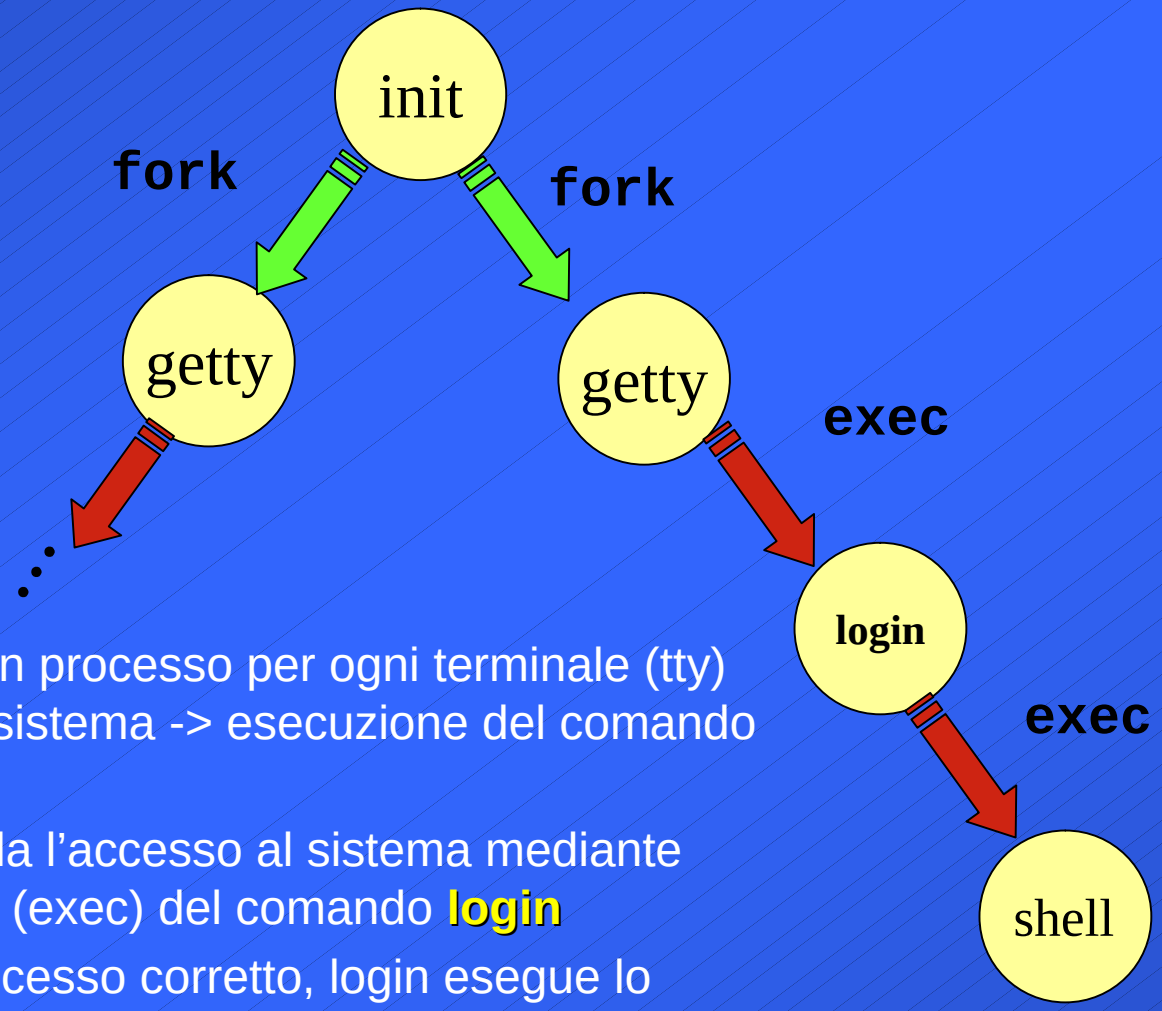
```
int execve(char *pathname, char *argV[], ..
           char * env[]);
```

- pathname è il nome (assoluto o relativo) dell'eseguibile da caricare
- argV è il vettore degli argomenti del programma da eseguire
- env è il vettore delle variabili di ambiente da sostituire all'ambiente del processo (contiene stringhe del tipo "VARIABILE=valore")

Esempio: execve

```
char *env[]={“USER=anna”, “PATH=/home/anna/d1”, (char
   *)0};
char *argv[]={“ls”,”-l”, “pippo”, (char *)0};
main()
{int pid, status;
pid=fork();
if (pid==0)
   {execve(“/bin/ls”, argv, env);
    printf(“exec fallita!\n”);
    exit(1);
}
else if (pid >0)
   { pid=wait(&status);  /* gestione dello stato.. */
   }
else printf(“fork fallita!);
}
```

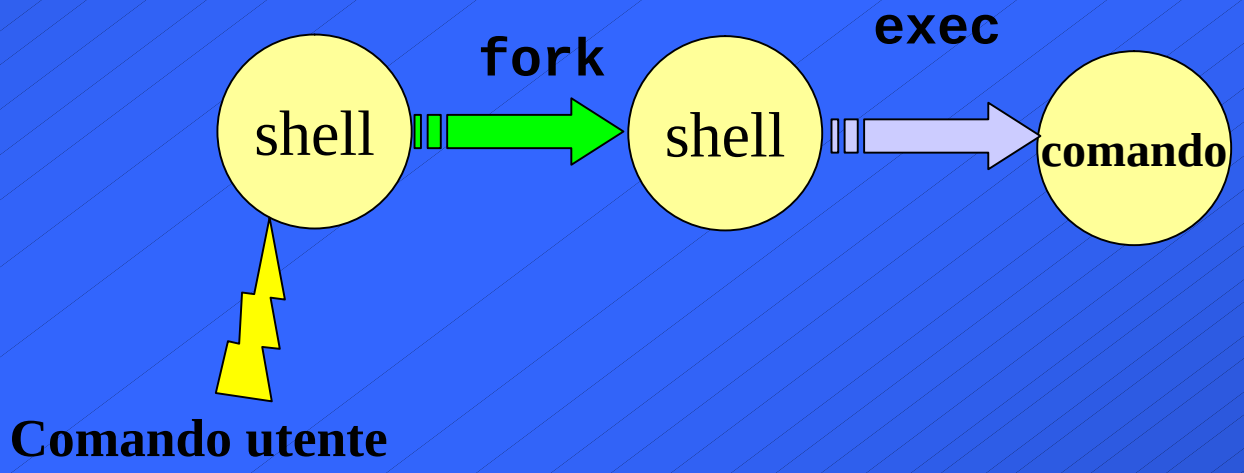
Inizializzazione dei processi Unix



- init genera un processo per ogni terminale (tty) collegato al sistema -> esecuzione del comando **getty**
- getty controlla l'accesso al sistema mediante l'esecuzione (exec) del comando **login**
- in caso di accesso corretto, login esegue lo **shell** (specificato dall'utente in /etc/passwd)

Interazione con l'utente tramite shell

- Ogni utente può interagire con lo shell mediante la specifica di comandi.
- Per ogni comando, lo shell genera un processo figlio dedicato all'esecuzione del comando:



Relazione shell padre-shell figlio

- Per ogni comando, lo shell genera un figlio; possibilità di **due diversi comportamenti**:
  - il padre si pone in attesa della terminazione del figlio (esecuzione in ***foreground***); es:

    ```
    ls - l pippo
    ```

  - il padre continua l'esecuzione concorrentemente con il figlio (esecuzione in ***background***):

    ```
    ls - l pippo &
    ```

Sincronizzazione tra processi in Unix: i segnali

Sincronizzazione tra processi

I processi interagenti possono avere bisogno di meccanismi **sincronizzazione**.

Unix: la sincronizzazione può realizzarsi mediante i segnali

Segnale:

è un'**interruzione software** a un processo, che notifica un evento asincrono.

Per esempio segnali:

- generati da terminale (es. CTRL+C)
- generati da altri processi
- generati da eccezioni HW (violazione dei limiti di memoria, divisioni per 0, etc.)
- generati da condizioni SW (time-out, scrittura su pipe chiusa, etc.)

Segnali Unix

- Un segnale può essere inviato:
  - dal kernel a un processo
  - da un processo utente ad altri processi utente

  (Es: comando **`kill`**)
- Quando un processo riceve un segnale, può comportarsi in **tre modi diversi**:
  - gestire il segnale con una funzione ***handler*** definita dal programmatore
  - eseguire un'azione predefinita dal S.O. (azione di ***default***)
  - ***ignorare*** il segnale
- Nei primi due casi, il processo **reagisce in modo asincrono** al segnale:
  1. interruzione dell'esecuzione
  2. esecuzione dell'azione associata al segnale ricevuto (***handler*** o ***default***)
  3. ritorno alla prossima istruzione del codice del processo interrotto

Segnali Unix/Linux

- Per ogni realizzazione di Unix esistono vari tipi di segnale (per esempio, Linux ne prevede 32 segnali), ognuno identificato da un intero.
- Ogni segnale, è associato a un particolare evento e prevede una specifica **azione di default**.
- È possibile riferire i segnali con identificatori simbolici (**`SIGxxx`**):

  `SIGKILL`, **`SIGSTOP`**, **`SIGUSR1`**, etc.
- L'associazione tra nome simbolico e intero corrispondente (che dipende dalla versione di Unix) è specificata nell'header file **`<signal.h>`**.

Segnali Unix (linux): signal.h

```
#define SIGHUP          1       /* Hangup (POSIX). Action: exit */
#define SIGINT          2       /* Interrupt (ANSI). Action: exit */
#define SIGQUIT         3       /* Quit (POSIX). Action: exit, core dump*/
#define SIGILL          4       /* Illegal instr.(ANSI).Action: exit,core dump */
...
#define SIGKILL         9       /* Kill, unblockable (POSIX). Action: exit*/
#define SIGUSR1         10      /* User-defined signal 1 (POSIX). Action: exit*/
#define SIGSEGV         11      /* Segm. violation (ANSI). Act: exit,core dump */
#define SIGUSR2         12      /* User-defined signal 2 (POSIX).Act: exit  */
#define SIGPIPE         13      /* Broken pipe (POSIX).Act: exit  */
#define SIGALRM         14      /* Alarm clock (POSIX). Act: exit */
#define SIGTERM         15      /* Termination (ANSI).  Act:exit*/
...
#define SIGCHLD         17      /* Child status changed (POSIX).Act: ignore  */
#define SIGCONT         18      /* Continue (POSIX).Act. ignore  */
#define SIGSTOP         19      /* Stop, unblockable (POSIX). Act: stop */
...
```

Gestione dei segnali

- Quando un processo riceve un segnale, può gestirlo in 3 modi diversi:
  - gestire il segnale con una funzione ***handler*** definita dal programmatore
  - eseguire un'azione predefinita dal S.O. (azione di ***default***)
  - ***ignorare*** il segnale

NB. Non tutti i segnali possono essere gestiti esplicitamente dai processi: **`SIGKILL`** e **`SIGSTOP`** non sono nè intercettabili nè ignorabili.

➔ qualunque processo, alla ricezione di SIGKILL o SIGSTOP esegue sempre l'**azione di default**.

System call `signal`

Ogni processo può gestire esplicitamente un segnale utilizzando la system call **`signal`**:

```
void (* signal(int sig, void (*func)()))(int);
```

- **`sig`** è l'intero (o il nome simb.) che individua il segnale da gestire
- il parametro **`func`** è un puntatore a una funzione che indica l'azione da associare al segnale; in particolare **`func`** può:
  - » puntare alla routine di gestione dell'interruzione (***handler***)
  - » valere **`SIG_IGN`** (nel caso di segnale ignorato)
  - » valere **`SIG_DFL`** (nel caso di azione di default)
- ritorna un puntatore a funzione:
  - » al precedente gestore del segnale
  - » SIG_ERR(-1), nel caso di errore

signal

Ad esempio:

```
#include <signal.h>
void gestore(int);
...
main()
{...
signal(SIGUSR1, gestore); /*SIGUSR1 gestito */
...
signal(SIGUSR1, SIG_DFL); /*USR1 torna a default */
signal(SIGKILL, SIG_IGN); /*errore! SIGKILL non è
                            ignorabile */
...
}
```

Routine di gestione del segnale (*handler*):

Caratteristiche:

- l'*handler* prevede sempre **un parametro formale** di tipo **`int`** che rappresenta il numero del segnale effetivamente ricevuto.
- l'*handler* non restituisce alcun risultato

```
void handler(int signum)
{ ....
  ....
  return;
}
```

Routine di gestione del segnale (*handler*):

Struttura del programma:

```
#include <signal.h>
void handler(int signum)
{ <istruzioni per la gestione del segnale>
  return;
}

main()
{ ...
  signal(SIGxxx, handler);
   ...
}
```

Gestione di segnali con handler

- Non sempre l'associazione ***segnale/handler*** è durevole:
  - alcune implementazioni di Unix (BSD, SystemV r.3 e seg, Linux), prevedono che l'azione rimanga installata anche dopo la ricezione del segnale.
  - in alcune realizzazioni (SystemV, prime versioni), invece, dopo l'attivazione dell'handler ripristina automaticamente l'azione di default. In questi casi, per riagganciare il segnale all'handler:

```
main()
{ ..
   signal(SIGUSR1, f);
      ...}
```

```
  void f(int s)
  { signal(SIGUSR1, f);
     ....
  }
```

Esempio: parametro del gestore

```
/* file segnali1.c */
#include <signal.h>
void handler(int);

main()
{ if (signal(SIGUSR1, handler)==SIG_ERR)
       perror("prima signal non riuscita\n");
  if (signal(SIGUSR2, handler)==SIG_ERR)
       perror("seconda signal non riuscita\n");
  for (;;);
}

void handler (int signum)
{ if (signum==SIGUSR1) printf("ricevuto sigusr1\n");
  else if (signum==SIGUSR2) printf("ricevuto sigusr2\n");
}
```

Esempio: gestore del SIGCHLD

- **`SIGCHLD`** è il segnale che il kernel invia a un processo padre quando un figlio termina.
- È possibile svincolare il padre da un'attesa esplicita della terminazione del figlio, mediante un'apposita funzione ***handler*** per la gestione di **`SIGCHLD`**:
  - la funzione ***handler*** verrà attivata in modo **asincrono** alla ricezione del segnale
  - handler chiamerà la **`wait`** con cui il padre potrà raccogliere ed eventualmente gestire lo stato di terminazione del figlio

Esempio: gestore del SIGCHLD

```
#include <signal.h>
void handler(int);

main()
{ int PID, i;
   PID=fork();
   if (PID>0) /* padre */
   {    signal(SIGCHLD,handler);
        for (i=0; i<10000000; i++); /* attività del padre*/
        exit(0); }
   else /* figlio */
   {    for (i=0; i<1000; i++); /* attività del figlio*/
        exit(1); }
}
void handler (int signum)
{ int status;
  wait(&status);
  printf("stato figlio:%d\n", status>>8);}
```

Segnali & `fork`

Le associazioni segnali-azioni vengono registrate nella *User Area* del processo; sappiamo che:

- una **`fork`** copia la ***User Area*** del padre nella ***User Area*** del figlio
- padre e figlio condividono lo stesso codice

quindi

- il figlio eredita dal padre le informazioni relative alla gestione dei segnali:
  - ignora gli stessi segnali ignorati dal padre
  - gestisce con le stesse funzioni gli stessi segnali gestiti dal padre
  - i segnali a default del figlio sono gli stessi del padre

➔ successive **`signal`** del figlio non hanno effetto sulla gestione dei segnali del padre.

Segnali & exec

Sappiamo che:

- una **exec** sostituisce codice e dati del processo che la chiame
- il contenuto della ***User Area*** viene mantenuta, tranne le informazioni legate al codice del processo (ad esempio, le funzioni di gestione dei segnali, che dopo l'**exec** non sono più visibili!)

quindi

- dopo un'**exec**, un processo:
  - ignora gli stessi segnali ignorati prima di **exec**
  - i segnali a default rimangono a default
  - **i segnali che prima erano gestiti, vengono riportati a default**

Esempio

```
#include <signal.h>

main()
{

signal(SIGINT, SIG_IGN);
execl("/bin/sleep","sleep","30", (char *)0);
}
```

N.B. Il comando: `sleep N`

mette nello stato *sleeping* il processo per N secondi

Esempio: esecuzione

```
anna@lab3-linux:~/esercizi$ cc segnali2.c
anna@lab3-linux:~/esercizi$ a.out&
[1] 500
anna@lab3-linux:~/esercizi$ kill -SIGINT
  500
anna@lab3-linux:~/esercizi$ kill -9 500
anna@lab3-linux:~/esercizi$
[1]+  Killed                    a.out
anna@lab3-linux:~/esercizi$
```

System call `kill`

I processi possono inviare segnali ad altri processi con la kill:

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

- **`sig`** è l'intero (o il nome simb.) che individua il segnale da gestire
- il parametro **`pid`** specifica il destinatario del segnale:
  - » pid> 0: l'intero è il pid dell'unico processo destinatario
  - » pid=0: il segnale è spedito a tutti i processi appartenenti al gruppo del mittente
  - » pid <-1: il segnale è spedito a tutti i processi con groupId uguale al valore assoluto di pid
  - » pid== -1: vari comportamenti possibili (Posix non specifica)

kill: esempio:

```
#include <stdio.h>
#include <signal.h>
int cont=0;
void handler(int signo)
{ printf ("Proc. %d: ricevuti n. %d segnali %d\n",
   getpid(),cont++, signo);
}

main ()
{int pid;
 pid = fork();
 if (pid == 0) /* figlio */
 { signal(SIGUSR1, handler);
    for (;;);
 }
 else /* padre */
   for(;;) kill(pid, SIGUSR1);
}
```

Segnali: altre system call

sleep:

```
unsigned int sleep(unsigned int N)
```

- provoca la sospensione del processo per N secondi (al massimo)
- se il processo riceve un segnale durante il periodo di sospensione, viene risvegliato *prematuramente*
- ritorna:
  - » 0, se la sospensione non è stata interrotta da segnali
  - » se il risveglio è stato causato da un segnale al tempo Ns, **`sleep`** restituisce in numero di secondi non utilizzati dell'intervallo di sospensione (N-Ns).

Segnali: altre system call

alarm:

```
unsigned int alarm(unsigned int N)
```

- imposta in timer che dopo N secondi invierà al processo il segnale **SIGALRM**
- ritorna:
  - » 0, se non vi erano time-out impostati in precedenza
  - » il numero di secondi mancante allo scadere del time-out precedente

NB: l'azione di ***default*** associata a SIGALRM è la terminazione.

Segnali: altre system call

pause:

```
int pause(void)
```

- sospende il processo fino alla ricezione di un qualunque segnale
- ritorna -1 (`errno = EINTR`)

Esempio

Due processi (padre e figlio) che si sincronizzano alternativamente mediante il segnale SIGUSR1 (getstito da entrambi con la funzione *handler*):


```
int ntimes = 0;
void handler(int signo)
{printf ("Processo %d ricevuto #%d volte il segnale %d\n",
   getpid(), ++ntimes, signo);
}
```

```
main ()
{int pid, ppid;
   signal(SIGUSR1, handler);
   if ((pid = fork()) < 0) /* fork fallita */
        exit(1);
   else if (pid == 0)  /* figlio*/
   {    ppid = getppid(); /* PID del padre */
        for (;;)
        {       printf("FIGLIO %d\n", getpid());
                sleep(1);
                kill(ppid, SIGUSR1);
                pause();}
   }
   else /* padre */
        for(;;) /* ciclo infinito */
        {       printf("PADRE %d\n", getpid());
                pause();
                sleep(1);
                kill(pid, SIGUSR1); }}
```

Affidabilità dei segnali

Alcune considerazioni:

- il gestore rimane **installato**?
  - Se no: posso reinstallare all'interno dell'handler

```
void handler(int s)
{ signal(SIGUSR1, handler);
  printf("Processo %d: segnale %d\n", getpid(), s);
     ...}
```

Cosa succede se qui arriva un nuovo segnale?

- cosa succede se arriva il segnale durante l'esecuzione dell'handler?
  - **innestamento** delle routine di gestione
  - **perdita** del segnale
  - **accodamento** dei segnali (segnali ***reliable, BSD 4.2***)

Interrompibilità di System Calls

- System Call: possono essere classificate in
  - ***slow*** system call: possono richiedere tempi di esecuzione non trascurabili perchè possono causare periodi di attesa (es: lettura da un dispositivo di I/O lento)
  - system call ***non slow***.
- una ***slow*** system call è interrompibile da un segnale; in caso di interruzione:
  - » ritorna -1
  - » **`errno`** vale EINTR
- possibilità di ri-esecuzione della system call:
  - » automatica (BSD 4.3)
  - » non automatica, ma comandata dal processo (in base al valore di **`errno`** e al valore restituito)

I thread nel sistema operativo LINUX: pthread & *Linuxthread*

LinuxThreads:
Caratteristiche

- Processi leggeri **realizzati a livello kernel**
- System call **`clone`**:

```
int  clone(int (*fn) (void *arg), void *child_stack, int
       flags, void *arg)
```

➔ **E` specifica di Linux: scarsa portabilita`!**

- Libreria *LinuxThreads*: funzioni di gestione dei threads, in conformita` con lo standard POSIX 1003.1c (*pthreads*):
  - *Creazione/terminazione threads*
  - *Sincronizzazione threads:lock, [semafori], variabili condizione*
  - *Etc.*

➢ Portabilita`

LinuxThreads

- Caratteristiche threads:
  - Il thread e` realizzato **a livello kernel** (e` l'unita` di schedulazione)
  - I thread vengono creati all'interno di un processo (task) per eseguire una funzione
  - Ogni thread ha il suo PID (a differenza di POSIX: distinzione tra *task* e *threads*)
  - Gestione dei segnali non conforme a POSIX:
    - Non c'e` la possibilita` di inviare un segnale a un task.
    - SIGUSR1 e SIGUSR2 vengono usati per l'implementazione dei threads e quindi non sono piu` disponibili.
- Sincronizzazione:
  - **Lock**: mutua esclusione (**`pthread_mutex_lock/unlock`**)
  - **Semafori**: esterni alla libreria **`pthread`** <semaphore.h> (POSIX 1003.1b)
  - **Variabili condizione** : (**`pthread_cond_wait,pthread_cond_signal`**)

Rappresentazione dei threads

- Il thread e` l'unita` di scheduling, ed e` univocamente individuato da un indentificatore (intero):

```
pthread_t tid;
```

  - Il tipo **`pthread_t`** **e`dichiarato nell'header file `<pthread.h>`**

LinuxThreads

- Creazione di threads:

```
#include <pthread.h>
int pthread_create(pthread_t *thread,pthread_attr_t *attr,
                   void *(*start_routine)(void *), void * arg);
```

- **Dove:**
  - **`thread:` e` il puntatore alla variabile che raccogliera` il thread_ID (PID)**
  - **`start_routine:` e` il puntatore alla funzione che contiene il codice del nuovo thread**
  - **`arg:` e` il puntatore all'eventuale vettore contenente i parametri della funzione da eseguire**
  - **`attr:` puo` essere usato per specificare eventuali attributi da associare al thread (di solito: NULL):**
    - ad esempio parametri di scheduling: priorita` etc.(solo per superuser!)
    - Legame con gli altri threads (ad esempio: *detached* o no)
- Ritorna : 0 in caso di successo, altrimenti un codice di errore (!=0)

LinuxThreads: creazione di threads

Ad esempio:

```
int A, B;
void * codice(void *){ /*definizione del codice del thread */
   …}
main()
{pthread_t t1, t2;
..
pthread_create(&t1,NULL, codice, NULL);
pthread_create(&t2,NULL, codice, NULL);
..
}
```

- Vengono creati due thread (di tid `t1` e `t2`) che eseguono le istruzioni contenute nella funzione `codice`:
  - I due thread appartengono allo stesso *task* (processo) e condividono le variabili globali del programma che li ha generati (ad esempio A e B).

LinuxThreads

- Un thread puo` terminare chiamando:

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

- Dove:
  - **`retval`: e` il puntatore alla variabile che contiene il valore di ritorno (puo` essere raccolto da altri threads, v. `pthread_join`).**
- E` una chiamata senza ritorno.
- Alternativa: **`return();`**

LinuxThreads

- Un thread puo` sospendersi in attesa della terminazione di un altro thread con:

```
int pthread_join(pthread_t th, void
  **thread_return);
```

- Dove:
  - **`th`: e` il pid del particolare thread da attendere**
  - **`thread_return`: e` il puntatore alla variabile dove verra` memorizzato il valore di ritorno del thread (v. `pthread_exit`)**

Esempio: creazione di thread

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *my_thread_process (void * arg)
{
  int i;

  for (i = 0 ; i < 5 ; i++) {
    printf ("Thread %s: %d\n", (char*)arg, i);
    sleep (1);
  }
  pthread_exit (0);
}
```

```
main ()
{
  pthread_t th1, th2;
  int retcode;
  if (pthread_create(&th1,NULL,my_thread_process,"1") < 0)
  { fprintf (stderr, "pthread_create error for thread 1\n");
    exit (1);
  }
  if (pthread_create(&th2,NULL,my_thread_process,"2") < 0)
  { fprintf (stderr, "pthread_create error for thread 2\n");
    exit (1);
  }
  retcode = pthread_join (th1, NULL);
  if (retcode != 0)
    fprintf (stderr, "join fallito %d\n", retcode);
  else printf("terminato il thread 1\n);

  retcode = pthread_join (th2, NULL);
  if (retcode != 0)
    fprintf (stderr, "join fallito %d\n", retcode);
  else printf("terminato il thread 2\n);
  return 0;
}
```

Compilazione

- Per compilare un programma che usa i linuxthreads:

```
gcc -D_REENTRANT -o prog prog.c   –lpthread
```

```
[aciampolini@ccib48 threads]$ prog
Thread 1: 0
Thread 2: 0
Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
Thread 1: 3
Thread 2: 3
Thread 1: 4
Thread 2: 4
terminato il thread 1
```

Terminazione di threads

- Normalmente e` necessario eseguire la **`pthread_join`** per ogni thread che termina la sua esecuzione, altrimenti rimangono allocate le aree di memoria ad esso assegnate.

- In alternativa si puo` “staccare” il thread dagli altri con:

```
int pthread_detach(pthread_t th);
```

  - il thread rilascia automaticamente le risorse assegnatagli quando termina.

LinuxThreads: MUTEX

- Lo standard POSIX 1003.1c (libreria <**`pthread.h`**>) definisce i **semafori binari** (o lock, mutex, etc.)
  - sono semafori il cui valore puo` essere 0 oppure 1 (*occupato* o *libero*);
  - vengono utilizzati tipicamente per risolvere problemi di **mutua esclusione**
  - **operazioni fondamentali:**
    - **inizializzazione:** **`pthread_mutex_init`**
    - **locking:** **`pthread_mutex_lock`**
    - **unlocking:** **`pthread_mutex_unlock`**
  - **Per operare sui mutex:**

    `pthread_mutex_t` : tipo di dato associato al mutex; esempio:

    ```
    pthread_mutex_t mux;
    ```

MUTEX: inizializzazione

- **L'inizializzazione di un `mutex` si puo`realizzare con:**

```
int pthread_mutex_init(pthread_mutex_t* mutex, const
    pthread_mutexattr_t* attr)
```

attribuisce un valore iniziale all'intero associato al semaforo (default: *libero*):

- **`mutex` : individua il mutex da inizializzare**
- **`attr` : punta a una struttura che contiene gli attributi del mutex; se NULL, il mutex viene inizializzato a *libero* (default).**

- in alternativa , si puo` inizializzare il mutex a default con la macro:

  `PTHREAD_MUTEX_INIZIALIZER`

- **esempio**: `pthread_mutex_t mux= PTHREAD_MUTEX_INIZIALIZER ;`

MUTEX: lock/unlock

- **Locking/unlocking si realizzano con:**

```
int pthread_mutex_lock(pthread_mutex_t* mux)
int pthread_mutex_unlock(pthread_mutex_t* mux)
```

  - **lock: se il mutex mux e` occupato,il thread chiamante si sospende; altrimenti occupa il mutex.**
  - **unlock: se vi sono processi in attesa del mutex, ne risveglia uno; altrimenti libera il mutex.**

Esempio

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define MAX 10
pthread_mutex_t M; /* def.mutex condiviso tra threads */
int DATA=0; /* variabile condivisa */
int accessi1=0; /*num. di accessi del thread 1 alla sez critica */
int accessi2=0; /*num. di accessi del thread 2 alla sez critica */

void *thread1_process (void * arg)
{   int k=1;
    while(k)
    {
        pthread_mutex_lock(&M); /*prologo */
        accessi1++;
         DATA++;
        k=(DATA>=MAX?0:1);
        printf("accessi di T1: %d\n", accessi1);
        pthread_mutex_unlock(&M); /*epilogo */
    }
    pthread_exit (0);
}
```

Esempio

```
void *thread2_process (void * arg)
{   int k=1;
   while(k)
    {
        pthread_mutex_lock(&M); /*prologo sez. critica */
        accessi2++;
        DATA++;
        k=(DATA>=MAX?0:1);
        printf("accessi di T2: %d\n", accessi2);
         pthread_mutex_unlock(&M); /*epilogo sez. critica*/
    }
    pthread_exit (0);
}
```

Esempio:

```
main ()
{ pthread_t th1, th2;
/* il mutex e` inizialmente libero: */
  pthread_mutex_init (&M, NULL);
if (pthread_create(&th1, NULL, thread1_process, NULL) < 0)
        { fprintf (stderr, "create error for thread 1\n");
          exit (1);
        }
  if (pthread_create(&th2, NULL,thread2_process,NULL) < 0)
  { fprintf (stderr, "create error for thread 2\n");
    exit (1);
  }
   pthread_join (th1, NULL);
  pthread_join (th2, NULL);
}
```

Test

```
$
$ gcc -D_REENTRANT -o tlock lock.c -lpthread
$ ./tlock
accessi di T2: 1
accessi di T1: 1
accessi di T2: 2
accessi di T1: 2
accessi di T1: 3
accessi di T1: 4
accessi di T1: 5
accessi di T1: 6
accessi di T1: 7
accessi di T1: 8
accessi di T2: 3
$
```

LinuxThreads: Semafori

- **Memoria condivisa**: uso dei semafori (POSIX.1003.1b)
  - Semafori: libreria <semaphore.h>
    - **`sem_init`: inizializzazione di un semaforo**
    - **`sem_wait`: *wait***
    - **`sem_post`: *signal***
  - **`sem_t`** : tipo di dato associato al semaforo; esempio:

```
static sem_t my_sem;
```

Operazioni sui semafori

- **`sem_init`: inizializzazione di un semaforo**

```
int sem_init(sem_t *sem, int pshared, unsigned int value)
```

attribuisce un valore iniziale all'intero associato al semaforo:

  - **`sem`: individua il semaforo da inizializzare**
  - **`pshared` : 0, se il semaforo non e` condiviso tra task, oppure non zero (sempre zero).**
  - **`value` : e` il valore iniziale da assegnare al semaforo.**

- `sem_t` : tipo di dato associato al semaforo; esempio:

```
static sem_t my_sem;
```

- ritorna sempre 0.

Operazioni sui semafori: sem_wait

- **wait su un semaforo**

```
int sem_wait(sem_t *sem)
```

dove:

- `sem`: **individua il semaforo sul quale operare.**

e` la *wait* di Dijkstra:

- se il valore del semaforo e` uguale a zero, sospende il thread chiamante nella coda associata al semaforo; altrimenti ne decrementa il valore.

Operazioni sui semafori: sem_post

- ***signal*** **su un semaforo:**

```
int sem_post(sem_t *sem)
```

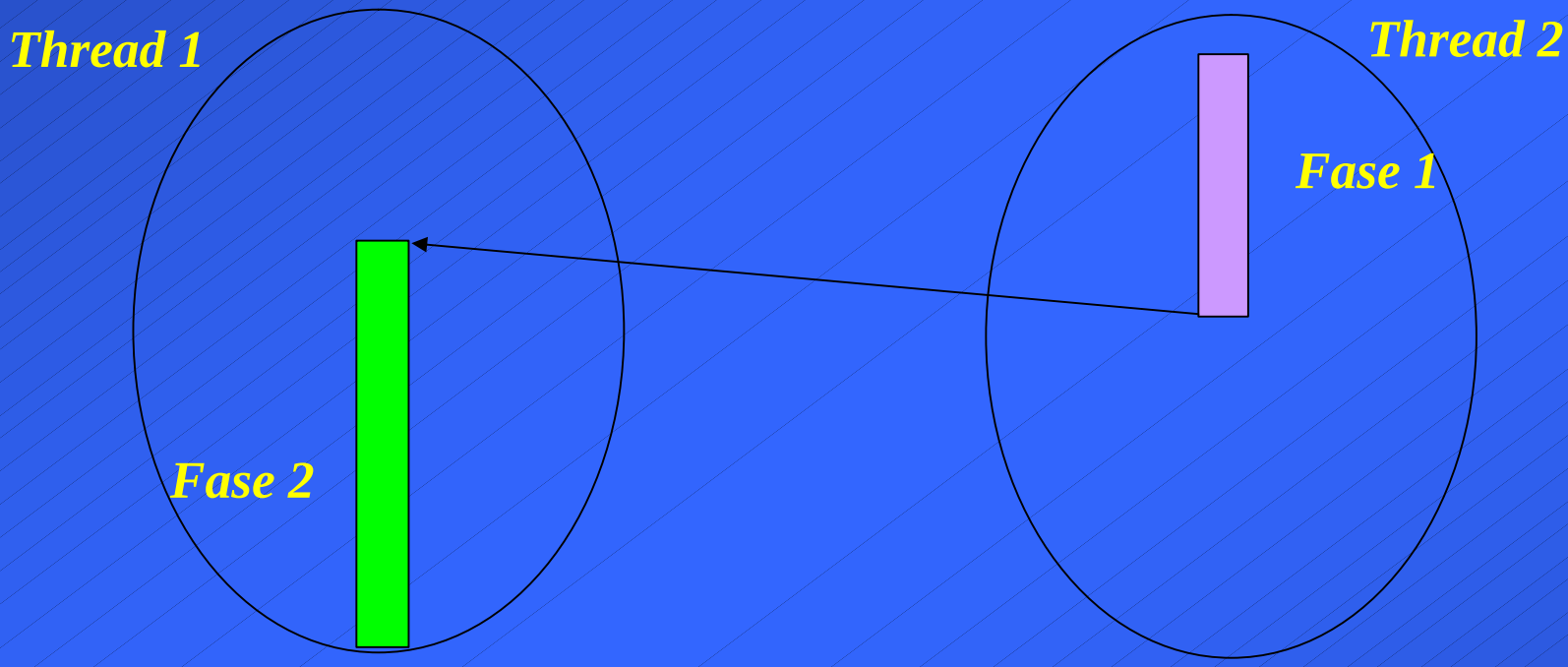
dove:

- `sem`: **individua il semaforo sul quale operare.**

e` la *signal* di Dijkstra:

- se c'e` almeno un thread sospeso nella coda associata al semaforo sem, viene risvegliato; altrimenti il valore del semaforo viene incrementato.

Esempio: sincronizzazione



- **Imposizione di un vincolo temporale: la FASE2 nel thread 1 va eseguita dopo la FASE1 nel thread2.**

Esempio: sincronizzazione

```
/* la FASE2 nel thread 1 va eseguita dopo la FASE1 nel thread
   2*/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

sem_t my_sem;
int V=0;

void *thread1_process (void * arg)
{
  printf ("Thread 1: partito!...\n");
   /* inizio Fase 2: */
  sem_wait (&my_sem);
  printf ("FASE2: Thread 1:  V=%d\n", V);
  pthread_exit (0);
}
```

```
void *thread2_process (void * arg)
{  int i;

     V=99;
     printf ("Thread 2: partito!...\n);
     /* inizio fase 1: */
     printf (“FASE1: Thread 2:  V=%d\n", V);
    /* …
     termine Fase 1: sblocco il thread 1*/
     sem_post (&my_sem);
     sleep (1);
  pthread_exit (0);
}
```

```
main ()
{ pthread_t th1, th2;
  void *ret;
  sem_init (&my_sem, 0, 0); /* semaforo a 0 */

if (pthread_create (&th1, NULL, thread1_process, NULL) < 0)
   { fprintf (stderr, "pthread_create error for thread
   1\n");
     exit (1);
  }

  if (pthread_create(&th2,NULL, thread2_process, NULL) < 0)
   {fprintf (stderr, "pthread_create error for thread \n");
     exit (1);
  }

  pthread_join (th1, &ret);
  pthread_join (th2, &ret);
}
```

Esempio:

- `gcc -D_REENTRANT -o sem sem.c –lpthread`

- **Esecuzione**:

```
[aciampolini@ccib48 threads]$ sem
Thread 1: partito!...
Thread 2: partito!...
FASE1: Thread 2:  V=99
FASE2: Thread 1:  V=99
[aciampolini@ccib48 threads]$
```

Semafori: esempio

```
/* tre processi che, ciclicamente, incrementano a
   turno (in ordine P1,P2,P3) la variabile V*/

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define MAX 13
static sem_t m; /* semaforo per la mutua esclusione
            nell'accesso alla sezione critica */
static sem_t s1,s2,s3; /* semafori per imporre
            l'ordine di accesso (P1,P2,P3) alla
            variabile V */
int V=0,F=0;
```

```
void *thread1_process (void * arg)
{   int k=1;
    while(k)
    {   sem_wait (&s1);
        sem_wait(&m);
        if (V<MAX)
                V++;
        else
        {       k=0;
                printf("T1: %d (V=%d)\n",++F, V);
        }
        sem_post(&m);
        sem_post(&s2);
    }
   pthread_exit (0);
}
```

```
void *thread2_process (void * arg)
{   int k=1;
    while(k)
    {   sem_wait (&s2);
        sem_wait(&m);
        if (V<MAX)
                V++;
         else
        {       k=0;
                printf("T2: %d (V=%d)\n",++F, V);
        }
        sem_post(&m);
        sem_post(&s3);
    }
   pthread_exit (0);
}
```

```
void *thread3_process (void * arg)
{   int k=1;
    while(k)
    {   sem_wait (&s3);
        sem_wait(&m);
        if (V<MAX)
                V++;
         else
        {       k=0;
                printf("T3: %d (V=%d)\n",++F, V);
        }
        sem_post(&m);
        sem_post(&s1);
    }
   pthread_exit (0);
}
```

```
main ()
{ pthread_t th1, th2,th3;

  sem_init (&m, 0, 1);
  sem_init(&s1,0,1);
  sem_init(&s2,0,0);
  sem_init(&s3,0,0);
  if (pthread_create(&th1, NULL, thread1_process, NULL) < 0)
   { fprintf (stderr, "pthread_create error for thread 1\n");
     exit (1);
  }
  if (pthread_create(&th2, NULL,thread2_process,NULL) < 0)
  { fprintf (stderr, "pthread_create error for thread 2\n");
     exit (1);
  }
  if (pthread_create(&th3,NULL,thread3_process, NULL) < 0)
   { fprintf (stderr, "pthread_create error for thread 3\n");
     exit (1);
  }
```

```
    pthread_join (th1, NULL);
    pthread_join (th2, NULL);
    pthread_join (th3, NULL);

}
```

Esecuzione:

```
[aciampolini@ccib48 threads]$ sem1
T2: 1 (V=10)
T3: 2 (V=10)
T1: 3 (V=10)
[aciampolini@ccib48 threads]$
```

LinuxThreads: variabili condizione

- Lo standard POSIX 1003.1c (libreria <**`pthread.h`**>) implementa le **variabili condizione**
  - Le variabili condizione (*condition*) sono uno strumento di sincronizzazione che premette ai threads di sospendere la propria esecuzione in attesa che siano soddisfatte alcune condizioni su dati condivisi.
  - ad ogni *condition* viene associata una coda nella quale i threads possono sospendersi (tipicamente, se la condizione non e` verificata).
  - **operazioni fondamentali:**
    - **inizializzazione:** **`pthread_cond_init`**
    - **sospensione:** **`pthread_cond_wait`**
    - **risveglio:** **`pthread_cond_signal`**
  - **Per operare sulle variabili condizione:**
    `pthread_cond_t`: è il tipo predefinito per le variabili condizione.

Variabili Condizione: inizializzazione

- **L'inizializzazione di una `condition` si puo`realizzare con:**

```
int_pthread_cond_init(pthread_cond_t* cond,
    pthread_cond_attr_t* cond_attr)
```

dove

- `cond` : **individua la condizione da inizializzare**
- `attr` : **punta a una struttura che contiene gli *attributi* della condizione; se NULL, viene inizializzata a default.**

NB: linux non implementa gli attributi !

- in alternativa, una variabile condizione può essere inizializzata staticamente con la costante:

  `PTHREAD_COND_INIZIALIZER`

- **esempio**: `pthread_cond_t C= PTHREAD_COND_INIZIALIZER ;`

Variabili condizione: wait

- **Un thread puo` sospendersi su una variabile condizione, se la condizione non e` verificata:**
  - **ad esempio:**

```
pthread_cond_t C= PTHREAD_COND_INIZIALIZER;
int bufferpieno=0;
...
if (bufferpieno) <sospensione sulla cond. C>
```

  - **La verifica della condizione e` una sezione critica!**
  - **Necessita` di garantire la mutua esclusione:**

e` necessario associare ad ogni variabile condizione un mutex :

```
pthread_cond_t C= PTHREAD_COND_INIZIALIZER;
pthread_mutex_t M=PTHREAD_MUTEX_INITIALIZER;
int bufferpieno=0;
...
pthread_mutex_lock(&M);
if (bufferpieno) <sospensione sulla cond. C>
pthread_mutex_unlock(&M);
```

Variabili condizione: wait

- **La sospensione su una condizione si ottiene mediante:**

```
int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t*
    mux)
```

dove:

- **cond: e` la variabile condizione**
- **mux: e` il mutex associato ad essa**

Effetto:

- **il thread chiamante si sospende sulla coda associata a cond, e il mutex mux viene liberato**

➔ **Al successivo risveglio (provocato da una *signal*), il thread rioccupera` il mutex automaticamente.**

Variabili condizione: signal

- Il risveglio di un thread sospeso su una variabile condizione puo`essere ottenuto mediante la funzione:

```
int pthread_cond_signal(pthread_cond_t* cond);
```

dove:

- cond: e` la variabile condizione.

Effetto:

- se esistono thread sospesi nella coda associata a cond, ne viene risvegliato uno (non viene specificato quale).
- se non vi sono thread sospesi sulla condizione, la signal non ha effetto.

N.B. non e` prevista una funzione per verificare lo stato della coda associata a una condizione.

Esempio: produttore e consumatore

Si vuole risolvere il classico problema del produttore e consumatore.

Progetto della *risorsa* (`prodcons`):

- **buffer circolare di interi, di dimensione data (ad esempio, 16) il cui stato e`dato da:**
  - **numero degli elementi contenuti: `cont`**
  - **puntatore alla prima posizione libera: `writepos`**
  - **puntatore al primo elemento occupato : `readpos`**
- **il buffer e`una risorsa da accedere in modo mutuamente esclusivo:**
  - **predispongo un mutex per il controllo della mutua esclusione nell'accesso al buffer: `lock`**
- **i thread produttori e consumatori necessitano di sincronizzazione in caso di :**
  - **buffer pieno: definisco una condition per la sospensione dei produttori se il buffer e` pieno (`notfull`)**
  - **buffer vuoto: definisco una condition per la sospensione dei produttori se il buffer e` pieno (`notempty`)**

Incapsulo il tutto all'interno di un tipo struct associato al buffer: `prodcons`

Produttori & Consumatori: tipo di dato associato al buffer

```
typedef struct
{
  int buffer[BUFFER_SIZE];
  pthread_mutex_t lock;
  int readpos, writepos;
  int cont;
  pthread_cond_t notempty;
  pthread_cond_t notfull;
}prodcons;
```

Produttore e consumatore

Operazioni sulla *risorsa* `prodcons`:

- **`Init`: inizializzazione del buffer.**
- **`Inserisci`: operazione eseguita dai produttori per l'inserimento di un nuovo elemento.**
- **`Estrai`: operazione eseguita dai consumatori per l'estrazione di un elemento dal buffer.**

Esempio: produttore e consumatore

```
#include <stdio.h>
#include <pthread.h>

#define BUFFER_SIZE 16


typedef struct
{
   int buffer[BUFFER_SIZE];
   pthread_mutex_t lock;
   int readpos, writepos;
   int cont;
   pthread_cond_t notempty;
   pthread_cond_t notfull;
}prodcons;
```

Esempio:Operazioni sul buffer

```
/* Inizializza il buffer */
void init (prodcons *b)
{
  pthread_mutex_init (&b->lock, NULL);
  pthread_cond_init (&b->notempty, NULL);
  pthread_cond_init (&b->notfull, NULL);
  b->cont=0;
  b->readpos = 0;
  b->writepos = 0;
}
```

Operazioni sul buffer

```
/* Inserimento: */
void inserisci (prodcons *b, int data)
{  pthread_mutex_lock (&b->lock);
  /* controlla che il buffer non sia pieno:*/
  while ( b->cont==BUFFER_SIZE)
      pthread_cond_wait (&b->notfull, &b->lock);
  /* scrivi data e aggiorna lo stato del buffer */
  b->buffer[b->writepos] = data;
  b->cont++;
  b->writepos++;
  if (b->writepos >= BUFFER_SIZE)
    b->writepos = 0;
  /* risveglia eventuali thread (consumatori) sospesi */
  pthread_cond_signal (&b->notempty);
  pthread_mutex_unlock (&b->lock);
}
```

Operazioni sul buffer

```
/*ESTRAZIONE:  */
int estrai (prodcons *b)
{  int data;
   pthread_mutex_lock (&b->lock);
   while (b->cont==0) /* il buffer e` vuoto? */
       pthread_cond_wait (&b->notempty, &b->lock);
   /* Leggi l'elemento e aggiorna lo stato del buffer*/
   data = b->buffer[b->readpos];
   b->cont--;
   b->readpos++;
   if (b->readpos >= BUFFER_SIZE)
     b->readpos = 0;
   /* Risveglia eventuali threads (produttori)*/
   pthread_cond_signal (&b->notfull);
   pthread_mutex_unlock (&b->lock);
   return data;
}
```

Produttore/consumatore: programma di test

```
/* Programma di test: 2 thread
   - un thread inserisce sequenzialmente max interi,
   - l'altro thread li estrae sequenzialmente per stamparli */
#define OVER (-1)
#define max 20

prodcons buffer;

void *producer (void *data)
{ int n;
  printf("sono il thread produttore\n\n");
  for (n = 0; n < max; n++)
    { printf ("Thread produttore %d --->\n", n);
      inserisci (&buffer, n);
    }
  inserisci (&buffer, OVER);
  return NULL;
}
```

```
void *consumer (void *data)
{  int d;
  printf("sono il thread consumatore \n\n");

  while (1)
    {
      d = estrai (&buffer);
      if (d == OVER)
        break;
      printf("Thread consumatore: --> %d\n",  d);
    }
  return NULL;
}
```

```
main ()
{
  pthread_t th_a, th_b;
  void *retval;

  init (&buffer);
  /* Creazione threads: */
  pthread_create (&th_a, NULL, producer, 0);
  pthread_create (&th_b, NULL, consumer, 0);
  /* Attesa teminazione threads creati: */
  pthread_join (th_a, &retval);
  pthread_join (th_b, &retval);
  return 0;
}
```

Segnali & Thread

- Per spedire un segnale a un **`thread`**:

```
int pthread_kill(pthread_t thread, int signo);
```

- Per sospendere un thread in attesa di un segnale:

```
int sigwait(const sigset_t *set, int *sig);
```

- Gestione del segnale associata al task:
  - **System call `signal`**
  - **Funzione `sigset` (POSIX)**