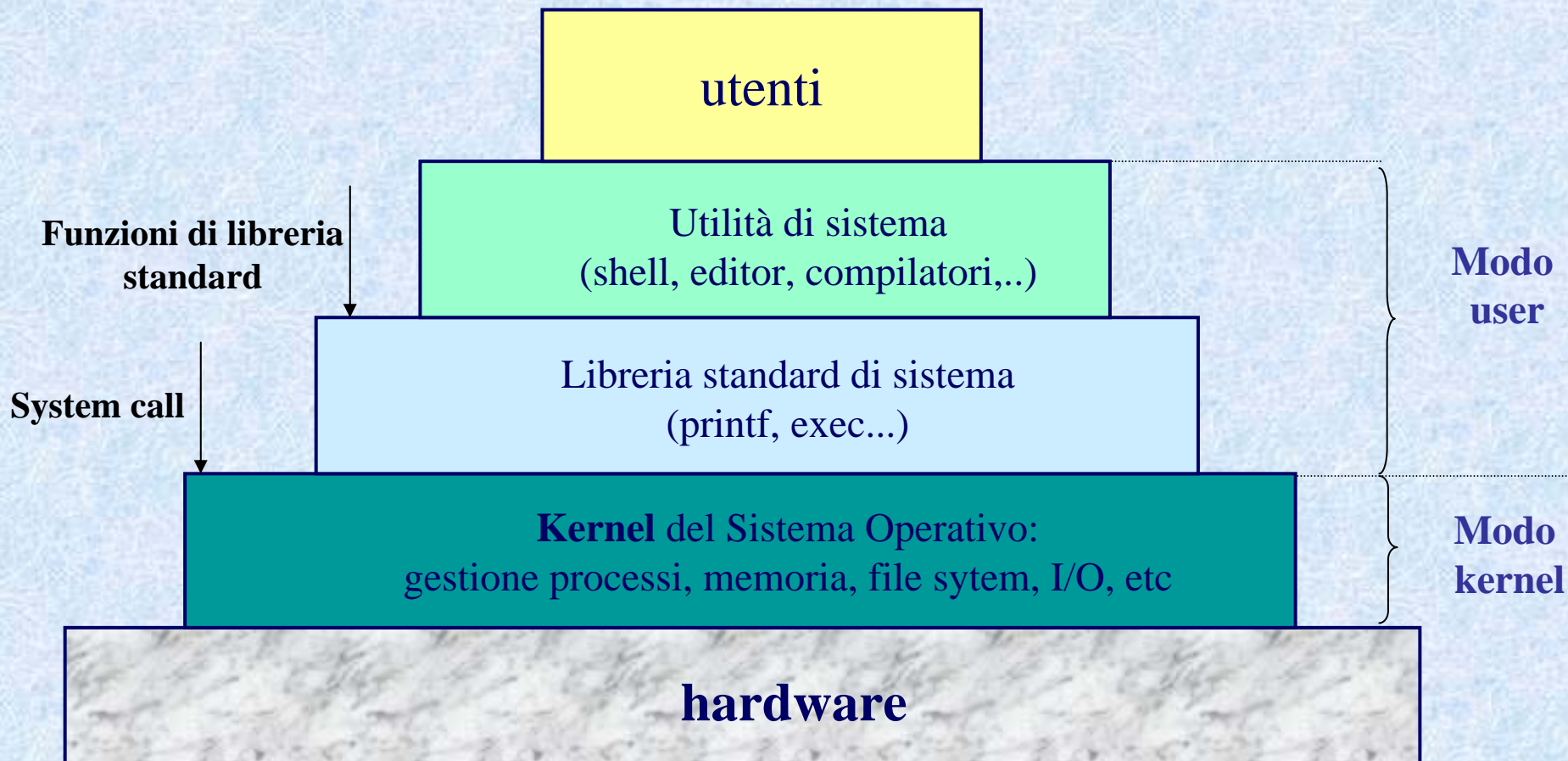


# Processi e thread nei sistemi operativi Unix e Linux

# Architettura di Unix



## 7.4 I processi Unix

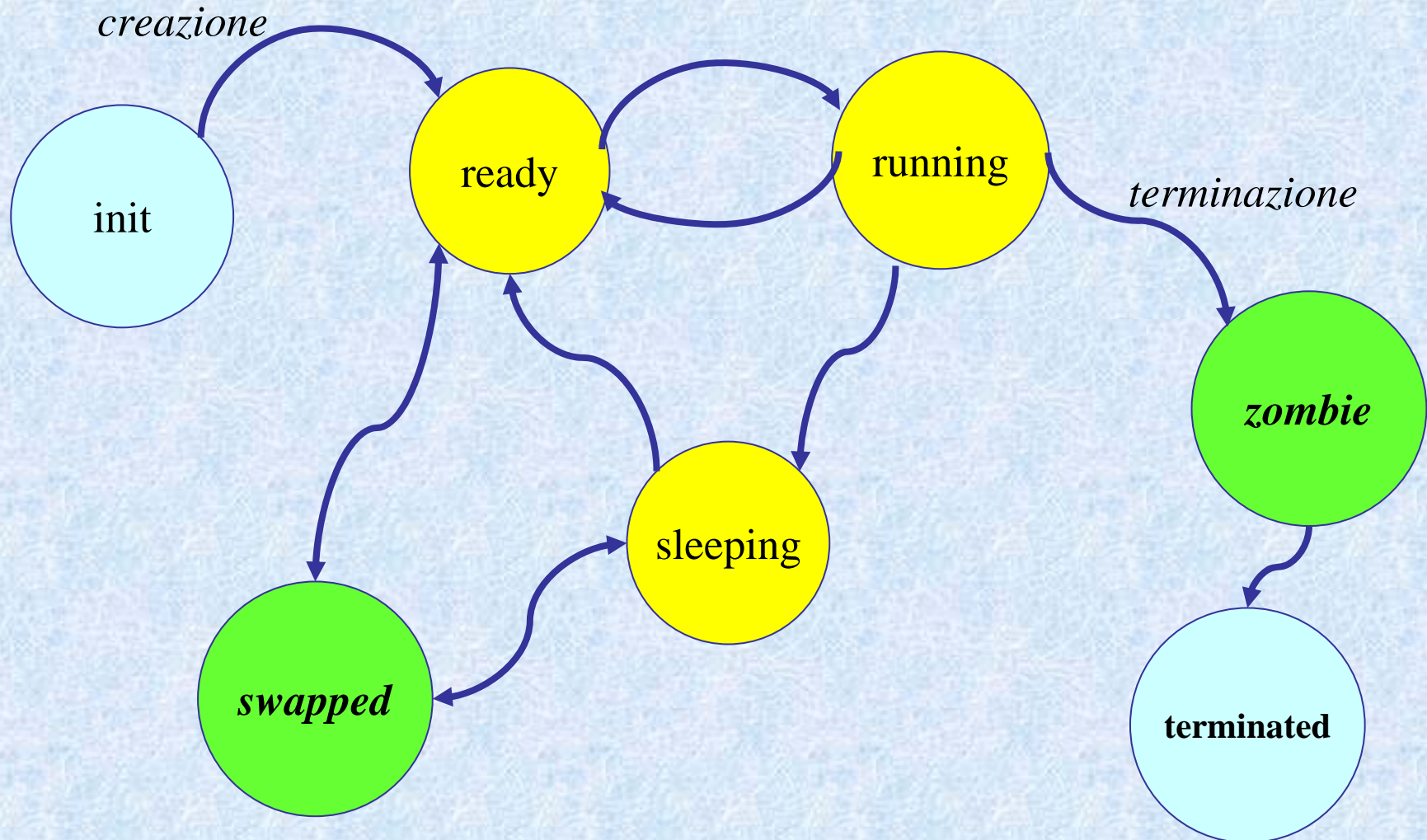
Unix è un sistema operativo *multiprogrammato a divisione di tempo*.

### Caratteristiche del processo Unix:

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

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

# Stati di un processo Unix





# Stati di un processo Unix

- **Init:** caricamento in memoria del processo e inizializzazione delle strutture dati del S.O.
- **Ready:** processo pronto
- **Running:** il processo usa la CPU
- **Sleeping:** il processo è sospeso in attesa di un evento
- **Terminated:** deallocazione del processo dalla memoria.

## In aggiunta:

- **Zombie:** il processo è terminato, ma è in attesa che il padre ne rilevi lo stato di terminazione.
- **Swapped:** il processo (o parte di esso) è temporaneamente trasferito in memoria secondaria.

## 7.4.2 Rappresentazione dei processi Unix

### Process Control Block (PCB)

Suddiviso in 2 strutture:

#### ✓ *Process Structure*

- *Risiede sempre in memoria*
- *appartiene al kernel*

#### ✓ *User Structure*

- *in caso di swapout, scaricato insieme al processo*

## Process Table

Le *Process Structure* di tutti i processi sono contenute nella *Process Table*:

PID <sub>i</sub>	Proc <sub>i</sub>

Process table: 1 elemento per ogni processo

## Process Structure

Contiene le informazioni necessarie al S.O. per la gestione del processo che devono essere sempre conservate in memoria, anche quando il processo **non è residente**:

- PID, PID del padre
- Parametri di scheduling
- stato
- riferimento indiretto al codice
- Riferimento all'area/e di memoria che contiene dati, stack e user area
- informazioni sui segnali pendenti (signal bitmap)
- user e group id
- puntatore alla User Structure



## User Structure

Contiene le informazioni necessarie al S.O. per la gestione del processo, **quando è residente**:

- copia dei **registri** di CPU
- informazioni sulle risorse allocate (ad es. **file aperti**)
- informazioni sulla gestione di **segnali**
  - ✓ **signal handler array** : una tabella che associa i vari segnali con le routines di gestione
- **ambiente** del processo: direttorio corrente, utente, gruppo, argc/argv, path, etc.

## 7.4.2 Rappresentazione dei processi Unix

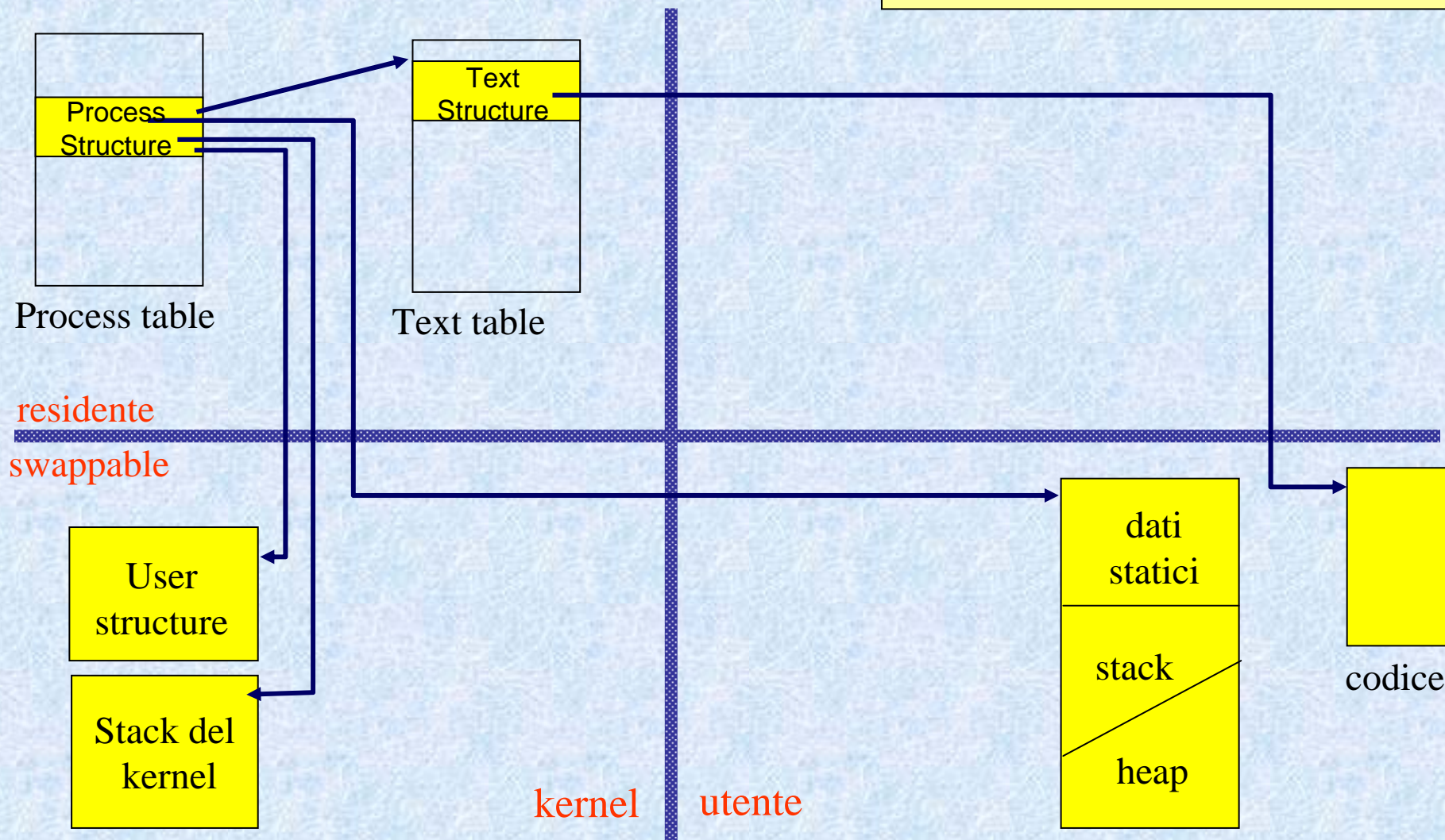
Codice <sub>i</sub>

Il codice dei processi è **rientrante**: più processi possono condividere lo stesso codice (*text*)

**Text table:** 1 elemento  $\forall$  segmento di codice utilizzato

L'immagine di un processo è l'insieme delle aree di memoria e delle strutture dati associate al processo.

## Immagine di un processo Unix



# System Call per la gestione di Processi

## Chiamate di sistema per:

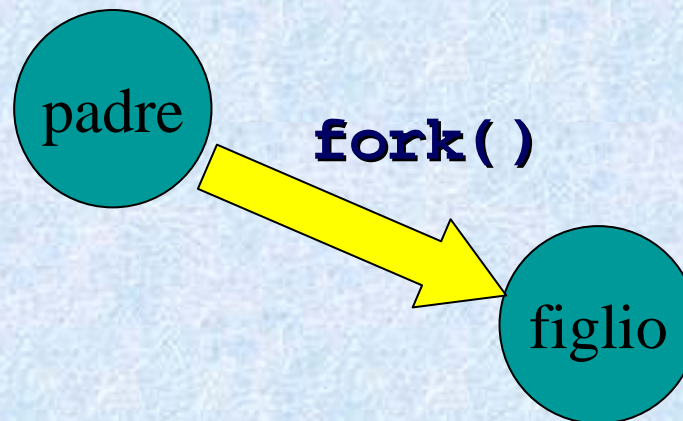
- creazione di processi: `fork( )`
- sostituzione di codice e dati: `exec . . ( )`
- terminazione: `exit( )`
- sospensione in attesa della terminazione di figli:  
`wait( )`



## 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



## Creazione di processi: *fork*

### Effetti principali di una *fork*:

- Si assegna un *PID*, una *process structure* e una *user structure* per il figlio
- Si alloca memoria per dati e stack del figlio
- Nella *process structure* del figlio:
  - si definiscono collegamenti a dati e stack e al segmento codice condiviso con il padre
  - si copiano i rimanenti campi dalla *process structure* del padre (compresa la *pending signal bitmap*)
- Nella *user structure* del figlio:
  - Si definisce valore di PC
  - Si copiano i rimanenti campi dalla *user structure* del padre (registri, tabella dei file aperti, *signal handler array*)

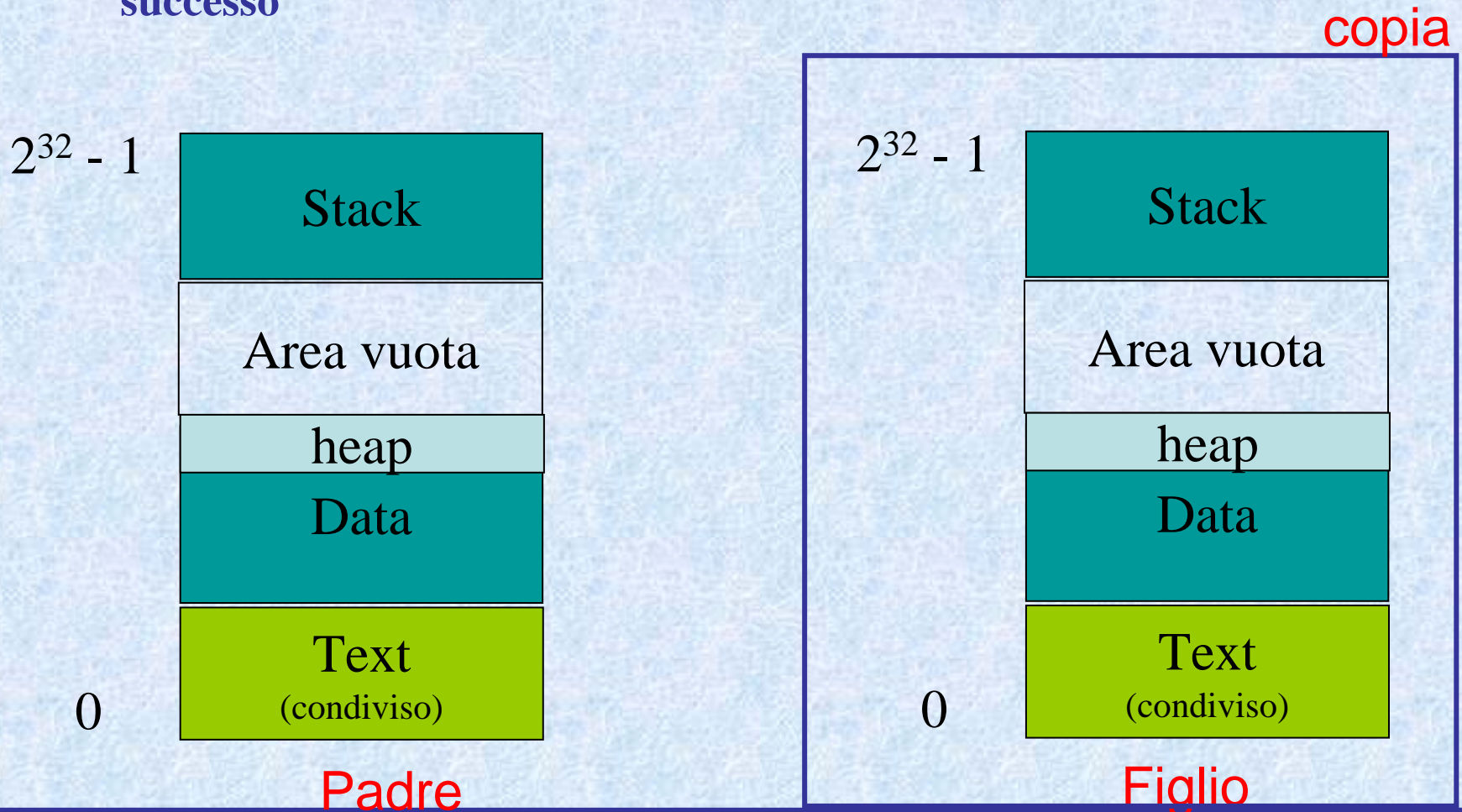
## fork ( )

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

- la fork non richiede parametri
- restituisce un intero che:
  - per il processo creato (figlio) 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

## fork( ) (2)

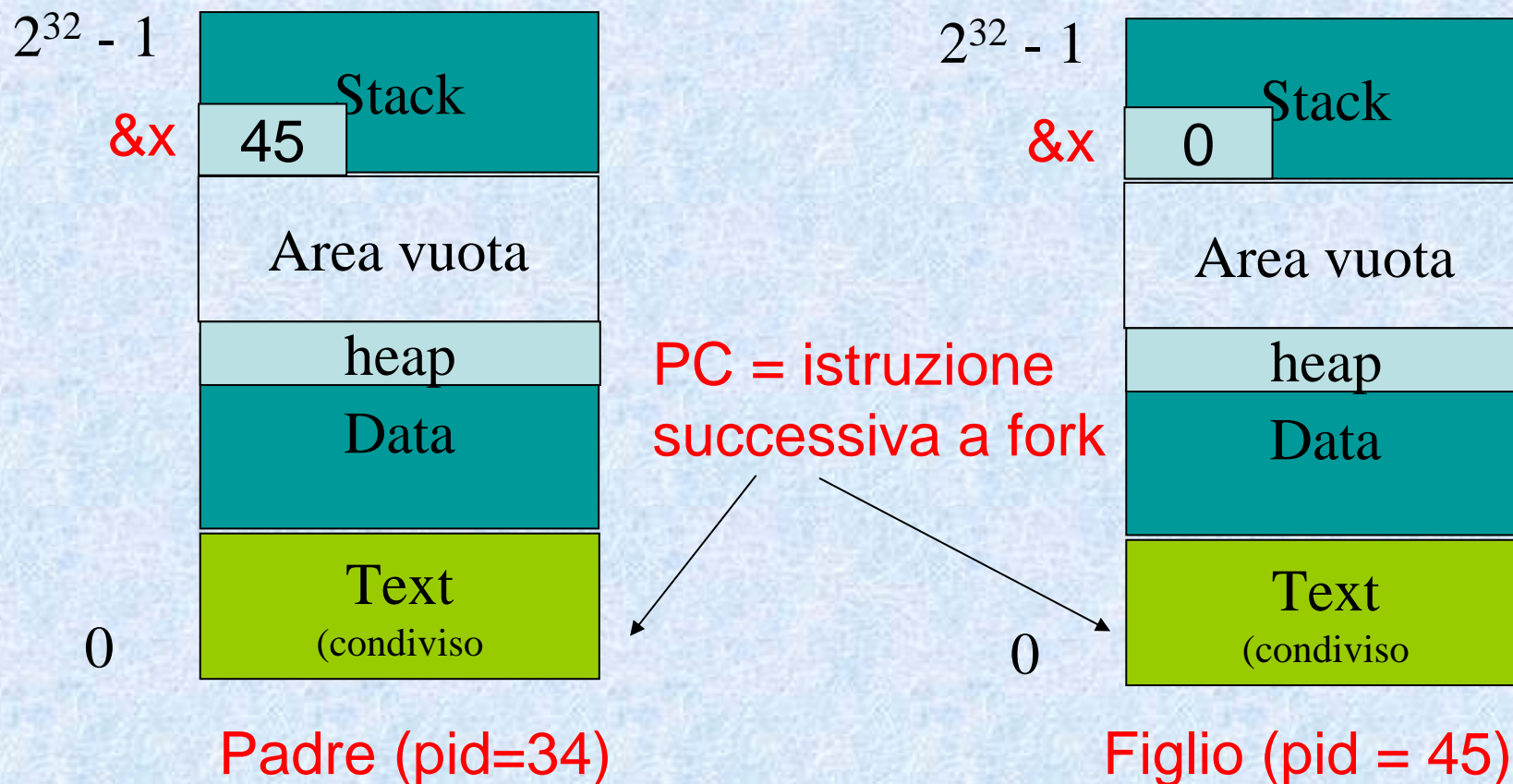
Spazio di indirizzamento di padre e figlio dopo una fork terminata con successo





## fork( ) (3)

Come prosegue l'esecuzione nei processi padre e figlio



## Sostituzione di codice: `exec..`

E' possibile sostituire il codice eseguito da un processo 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
- ✓ non crea un nuovo processo, ma semplicemente modifica lo spazio di memoria del processo che la chiama

## exec1 ( )

```
int exec1(char *pathname, char *arg0, ..  
          char *argN, (char*)0);
```

- **pathname** è il nome del file contenente il nuovo programma
- **arg0** è il nome del programma (argv[0])
- **argN[1],..argN[...]** sono gli argomenti da passare al programma (come quelli passati da riga di comando quando si invoca l'eseguibile)
- **(char \*)0** è il puntatore nullo che termina la lista.

## Effetti della *exec*

==> Se la *exec* ha successo non ritorna!!!

==> In caso di fallimento restituisce un codice di errore

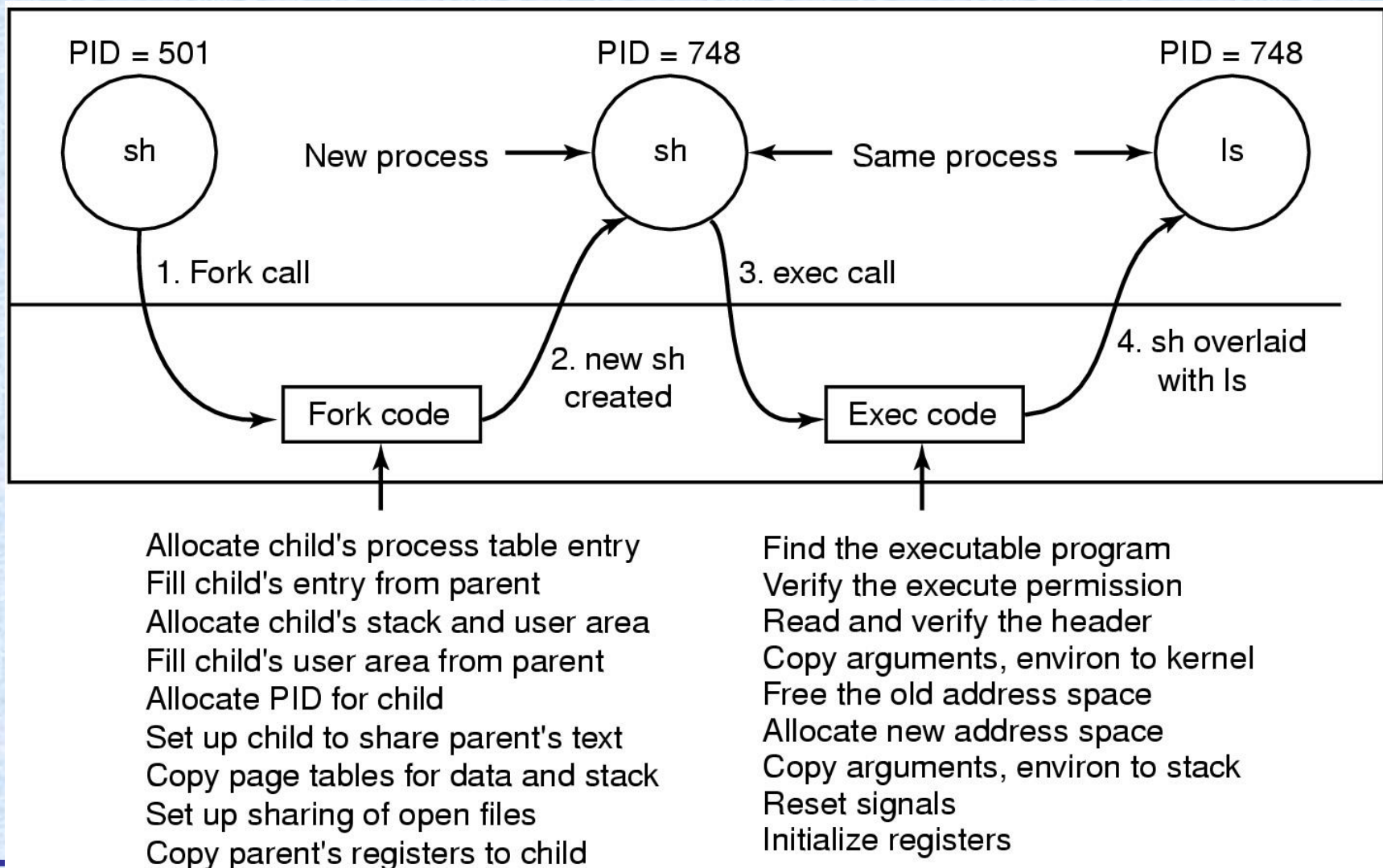
- Ad esempio se non trova il file eseguibile,

Dopo la *exec*, il processo :

- conserva il PID
- conserva la *process structure*, dove però:
  - cambia il riferimento al codice
  - cambia il riferimento a dati globali, stack e heap
- conserva la *user area*
  - con differenti contenuti per PC e altre informazioni del contesto
- conserva lo stack del kernel
- conserva le risorse già assegnate (es. file aperti)



## ESEMPIO: esecuzione del comando *ls* da parte della shell



## Esempio di uso combinato di *fork*, *exec* e *wait*

```
int main(int argc, char *argv[1])
{
    int pid, status;
    pid=fork();
    if(pid==0)
    {
        execl("\bin\ls", "ls","-l", char* 0);
        printf("exec fallita")
    }
    else if (pid>0)
    {
        pid=wait(&status);
        <gestione dello stato>
    }
    else printf("fork fallita")
}
```

# Terminazione di processi

Un processo può terminare:

- ✓ involontariamente (eccezioni dovute ad azioni illegali, ecc.)
- ✓ volontariamente (mediante la system call `exit`)

Terminazione del processo:

- 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*
- se il processo che termina ha figli non terminati, il processo `init` adotta i figli;

## exit()

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

- attraverso il parametro **status** 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

### Effetti di una **exit()**:

- chiusura dei file aperti
- terminazione del processo (eventualmente dopo il passaggio in stato *zombie*)



## wait

Lo stato di terminazione è rilevato dal processo padre, mediante la system call `wait()`:

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

- il parametro **status** è un puntatore ad un intero 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 ( $<0$ )

## Scheduling in Unix (1)

**Obiettivo:** privilegiare i processi interattivi

- ad ogni processo è associato un *livello di priorità*: più grande è il valore, più bassa è la priorità
- Ad ogni livello è associata una coda, gestita con *Round Robin*
- Priorità dinamiche, ricalcolate a intervalli fissi di tempo (1 secondo)

$$\text{priorità} = \text{base} + \text{cpu\_usage} + \text{nice}$$

*cpu\_usage* : aggiornato in base al tempo di permanenza in esecuzione  
(numero di clock tick)

*nice* : valore intero nell'intervallo [-20, +20]

- Dettagli della politica: dipendenti dall'implementazione

# Scheduling in Unix (1)

## Componenti della priorità:

$$\text{priorità} = \text{base} + \text{cpu\_usage} + \text{nice}$$

## Valore di priorità

- positivo per i processi in stato utente;
- negativo per i processi in stato supervisore.

Quando un processo passa in stato supervisore, conserva la sua priorità; alla riattivazione dopo una sospensione in stato supervisore, cambia il suo valore base e il nuovo valore è mantenuto per la permanenza in stato supervisore.

*Esempio:*

<i>Operazione conclusa</i>	<i>base</i>
Uscita su terminale	-1
Ingresso da terminale	-2
Uscita su disco	-3
Ingresso da disco	-4

## Scheduling in Unix

Meccanismo di *aging* (*invecchiamento*) usato per il calcolo di *cpu \_usage* :

- Fissiamo un intervallo di decadimento  $\Delta t$
- I tick ricevuti mentre il processo P è in esecuzione vengono accumulati in una variabile temporanea *tick*
- Ogni  $\Delta t$   
$$cpu\_usage = cpu\_usage / 2 + tick$$
$$tick = 0$$
- Il peso dei tick utilizzati decresce col tempo (*aging*)
- La penalizzazione dei processi che hanno utilizzato intensamente CPU diminuisce nel tempo



## Interazione tra processi e threads

- **Unix (processi):**
  - **eventi asincroni: *segnali***
  - **scambio di messaggi: *pipe***
- **Linux (threads):**
  - **semafori**
  - **variabili condizione**
  - **ecc**

## Segnali

### Segnale: interruzione software

- inviato a uno o più processi,
- notifica un evento asincrono.

### Chi può inviare segnali:

- i processi, mediante la chiamata *kill*
- il kernel, a seguito di:
  - interruzioni (esempio del timer)
  - eccezioni (esempio violazione dei limiti dello spazio di memoria assegnato al processo)

# System call `kill`

Con la chiamata *kill*, i processi possono inviare segnali ad altri processi:

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

- `sig` è l'intero che individua il segnale inviato
- 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)

# System call `signal`

Ogni processo scegliere l'azione da associare a un segnale, mediante la system call `signal`:

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

- `sig` è l'intero (o il nome simbolico) 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)
- Assegna a `func`:
  - ✓ Puntatore al precedente gestore del segnale
  - ✓ `SIG_ERR(-1)`, nel caso di errore



# Funzione di gestione del segnale (*handler*):

## Caratteristiche:

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

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

# Segnali

## Strutture Dati relative ai segnali

Per ogni processo, definiti *signal handler array* e *pending signal bitmap*

- *signal handler array* :
  - risiede nella user structure
  - descrive cosa fare quando arriva un segnale di un certo tipo  
(ignorare, eseguire azione di default o trattare eseguendo la funzione associata)
- *pending signal bitmap (signal mask)*:
  - risiede nella process structure
  - un bit per ogni tipo di segnale  
(il bit  $x = 1$  se è pendente il segnale corrispondente)

*Signal handler array e pending signal bitmap:*

- ereditate dal padre con la chiamata *fork*;
- resettate con la chiamata *exec*

# Segnali

## Come si rileva la presenza di un segnale

- Quando un segnale viene inviato il kernel mette a 1 il corrispondente bit nella *signal bitmap*
  - più segnali dello stesso tipo in rapida sequenza possono essere visti come uno solo
- Segnale riconosciuto immediatamente per il processo in esecuzione; altrimenti *signal bitmap* controllata dal kernel a ogni ritorno da stato supervisore.
  - Per esempio, al ritorno da una SVC, o dalla gestione di una interruzione
- Un processo può sospendersi in attesa di una segnale con la chiamata *pause*
  - riattivato al primo arrivo di un segnale

## Segnali

Cosa accade quando il kernel trova un segnale pendente

- **Esegue l'azione richiesta**
  - Ignorare il segnale, eseguire azione di default o quella del signal handler definito dall'utente
- **Se deve essere invocato un signal handler definito dall'utente:**
  - Il kernel modifica la user stack inserendo il record di attivazione del signal handler e lo manda in esecuzione
  - Se il segnale non provoca la terminazione del processo, al termine dello handler l'esecuzione riprende dal punto di interruzione



# pipe

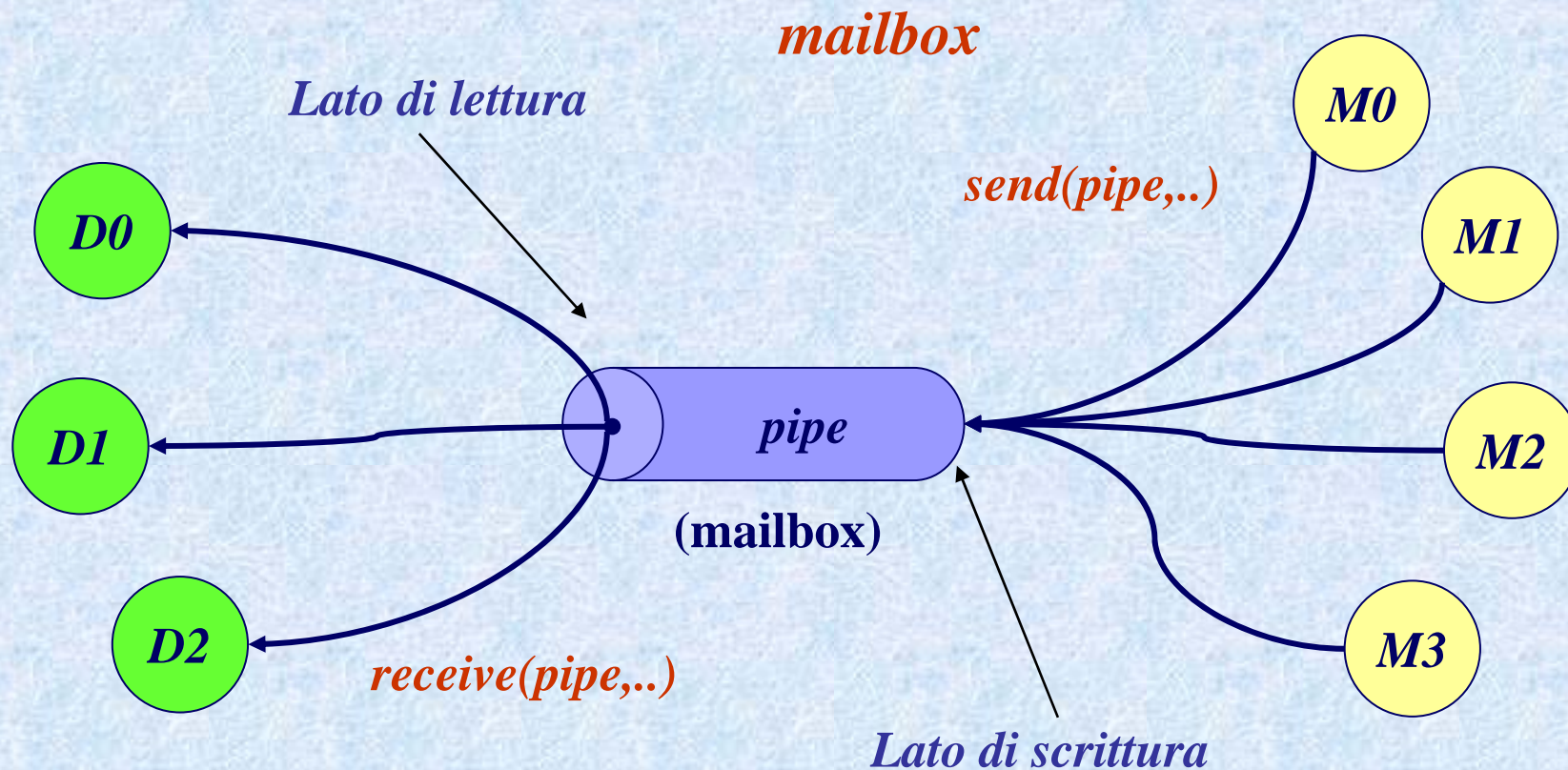
Il pipe è un **canale di comunicazione**:

- *unidirezionale*
- *molti-a-molti*
- *capacità limitata*: è possibile l'accodamento di un numero limitato di messaggi, gestiti in modo FIFO; il limite è stabilito dalla dimensione del pipe (es.4096 bytes).
- *Realizza un meccanismo tipo produttore-consumatore*

## Comunicazione con pipe

Mediante il pipe, la comunicazione tra processi è *indiretta*:

*In linea di principio:*



# System call pipe

Per creare un pipe:

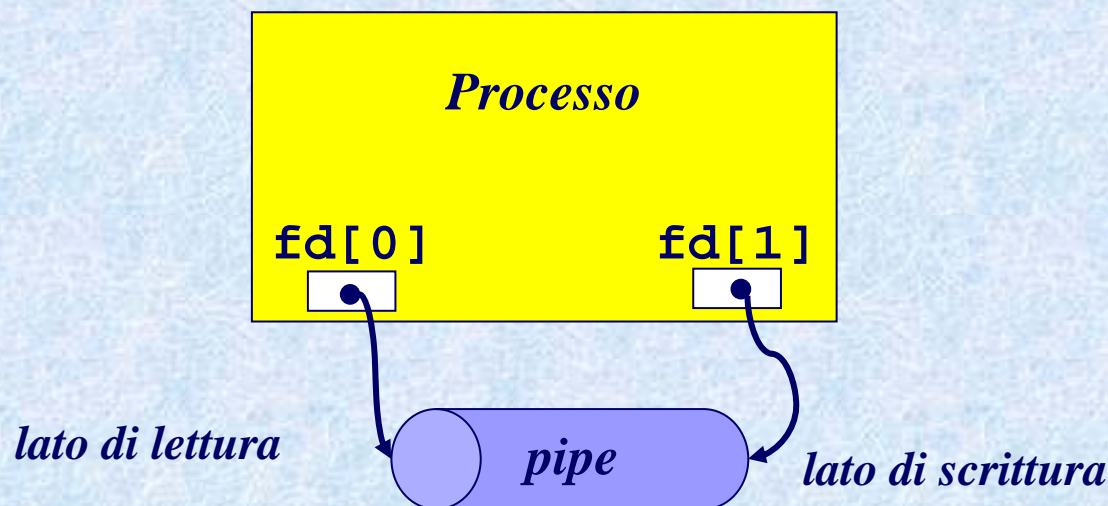
```
int pipe(int fd[2]);
```

- **fd** è il puntatore a un vettore di 2 file descriptor, che verranno inizializzati dalla system call in caso di successo:
  - ✓ **fd[0]** rappresenta il lato di lettura del pipe
  - ✓ **fd[1]** è il lato di scrittura del pipe
- la system call pipe restituisce:
  - un valore negativo, in caso di fallimento
  - 0, se ha successo

## System call `pipe`

Se `pipe(fd)` ha successo:

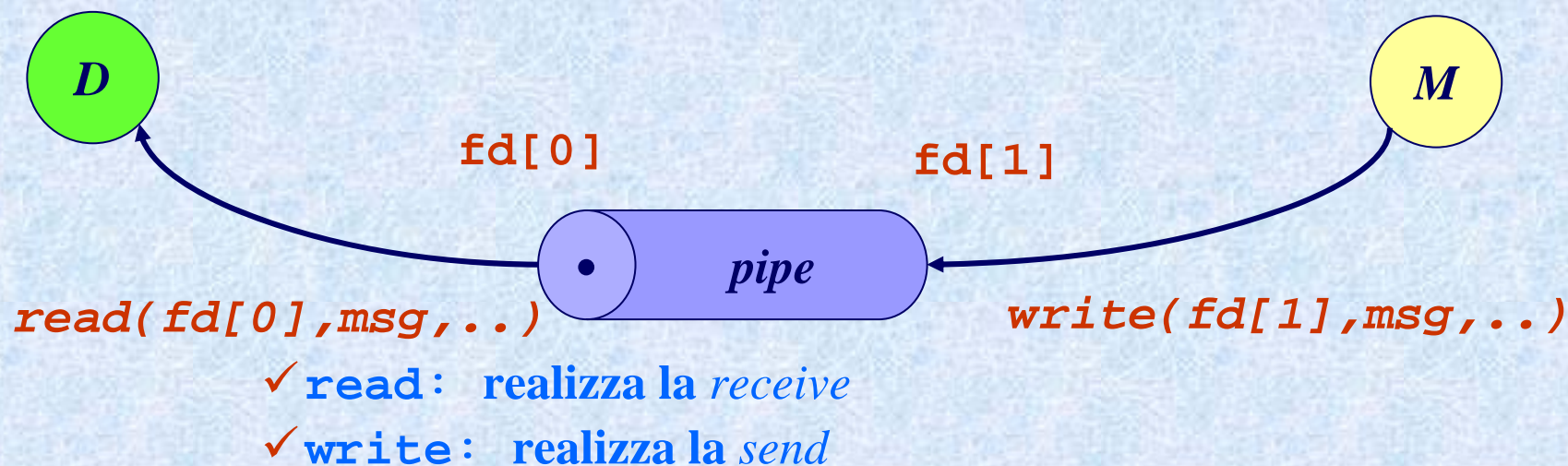
- vengono allocati due nuovi elementi nella tabella dei file aperti del processo e i rispettivi file descriptor vengono assegnati a `fd[0]` e `fd[1]`:
  - ✓ `fd[0]`: lato di lettura (*receive*) del pipe
  - ✓ `fd[1]`: lato di scrittura (*send*) del pipe





## Accesso al pipe

- I processi identificano ogni lato di accesso al pipe con un *file descriptor*
- Per l'accesso al pipe si utilizzano le system call per la lettura e la scrittura nei file: `read`, `write`:



# Sincronizzazione dei processi comunicanti

Il canale (*pipe*) ha capacità limitata: in certe circostanze sincronizza i processi :

- se il *pipe* è vuoto: un processo che legge si blocca
- se il *pipe* è pieno: un processo che scrive si blocca

➔ **read** e **write** da/verso pipe possono essere bloccanti !

## Chiusura di pipe

- Ogni processo può chiudere un estremo del pipe con una `close`.
- La `close` ha effetto solo per il processo che la esegue
- Un estremo del pipe viene chiuso *globalmente* quando tutti i processi che ne avevano visibilità hanno compiuto una `close`.

# Quali processi possono comunicare mediante pipe ?

- Per mittente e destinatario il riferimento al canale di comunicazione è un *file descriptor*
- Per i pipe valgono le stesse regole di ereditarietà dei file

==> soltanto i processi appartenenti a una stessa gerarchia possono scambiarsi messaggi; ad esempio:

- processi **fratelli** (che ereditano il pipe dal processo padre)
- un processo **padre** e processi **figli**;
- **nonno** e **nipote**
- ...



## Comunicazione con pipe: esempio

Realizzazione del comando `sort < f | head` della shell

- La shell esegue *fork* e genera F1;
- F1 esegue *exec* e sostituisce il proprio codice con quello di `sort`;
- F1 crea un pipe
- F1 esegue *fork* e genera F1,1, che eredita il pipe dal padre;
- F1,1 esegue *exec* e sostituisce il proprio codice con quello di `head`, che mantiene l'accesso al pipe;
- F1 prende l'ingresso da `f` e dirige l'uscita sul pipe;
- `head` prende l'ingresso dal pipe.

## Comunicazione con pipe: conclusioni

Il pipe ha due svantaggi:

- consente la comunicazione solo **tra processi in relazione di parentela**
- **non è persistente**: viene distrutto quando terminano tutti i processi che lo usano.

Come realizzare la comunicazione tra processi di gerarchie diverse ?

- *FIFO* (system V)
- *socket*

# Threads in Linux

- **UNIX non dispone di chiamate di sistema per i threads**
  - in particolare: *fork* genera *processi* con spazi di indirizzamento disgiunti
- **LINUX dispone di chiamate di sistemi per i threads**
  - in particolare chiamata *clone*: genera un *thread* figlio che condivide (in parte) lo spazio di indirizzamento del padre
  - se lo spazio condiviso è vuoto, genera un processo
  - ogni processo ha un *thread* iniziale e può generare altri *thread*
- **In LINUX, l'unità di schedulazione è il thread**
- **Programmi che utilizzano la *clone* (e le altre chiamate per i thread) non sono portabili su UNIX**
- **Per garantire la portabilità, POSIX definisce un insieme di funzioni standard per i thread**
  - realizzabili con librerie (*livello delle librerie standard*)
  - differenti realizzazioni per UNIX e LINUX

# LinuxThreads: creazione di thread

## Creazione di thread:

(il tipo `pthread_t` è dichiarato nell'header file `<pthread.h>`)

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

```
..... •
```

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,  
                  void (*start_routine), void *arg);
```

dove:

- **thread**: è il puntatore alla variabile che raccoglierà il thread\_ID (TID)
- **start\_routine**: è il puntatore alla funzione che contiene il codice del nuovo thread
- **arg**: è il puntatore all'eventuale vettore contenente i parametri da passare alla funzione da eseguire
- **attr**: può essere usato per specificare eventuali attributi da associare al thread (di solito: NULL)

==> restituisce 0 in caso di successo, altrimenti un codice di errore



# LinuxThread: terminazione di thread

- Un thread può terminare chiamando:

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

dove:

- **retval**: è il puntatore alla variabile che contiene il valore di ritorno

==> una chiamata senza ritorno.

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

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

dove:

- **th**: è il pid del thread da attendere
- **thread\_return**: è il puntatore alla variabile dove verrà memorizzato il valore di ritorno del thread (vedere **pthread\_exit**)

# LinuxThread: sincronizzazione

***Mutex***: meccanismo per la mutua esclusione nello standard POSIX 1003.1c:

- sono semafori il cui valore può essere 0 oppure 1 (*occupato* o *libero*);
- dati del tipo `pthread_mutex_t`
- operazioni fondamentali:
  - inizializzazione: `pthread_mutex_init`
  - locking: `pthread_mutex_lock`
  - unlocking: `pthread_mutex_unlock`

# Inizializzazione di mutex

L'inizializzazione di un **mutex** si può realizzare con:

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

dove:

- **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*.

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

## Operazioni sui mutex: lock/unlock

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

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



# LinuxThread: variabili condizione

***condition***: meccanismo per la sincronizzazione nello standard POSIX 1003.1c:

- dato del tipo **`pthread_cond_t`**
- analogamente al semaforo ha una coda sulla quale si sospendono i thread
- a differenza dei semafori, non ha un valore che condiziona la sospensione;
- sono associate operazioni per la sospensione e la riattivazione *incondizionata* dei thread
- La sospensione può essere condizionata al valore di un'espressione calcolato esplicitamente dal thread nel proprio ambiente.

## Variabili Condizione: inizializzazione

L'inizializzazione di una *condition* si può 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.

## Variabili Condizione: uso

Per l'utilizzo delle variabili di condizione sono disponibili le operazioni:

**pthread\_cond\_wait(pthread\_cond\_t \*cond)**

- Sospende incondizionatamente il thread sulla variabile di condizione **cond**
- Può essere utilizzata per la sospensione condizionata eseguendola dopo la preventiva verifica di una *condizione di accesso*.

**pthread\_cond\_signal(pthread\_cond\_t \*cond)**

- Se esistono thread sospesi sulla variabile di condizione, riattiva il primo; altrimenti non ha effetto

## Variabili condizione: wait

Il meccanismo `pthread_cond_wait` può essere combinato con la preventiva verifica di una *condizione di accesso* per realizzare la sospensione condizionata

- La verifica dipende generalmente da variabili comuni e pertanto deve essere eseguita in mutua esclusione

==>> necessario associare un *mutex* alla variabile *condition*

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

dove:

- **cond**: è la variabile *condition*
- **mux**: è il *mutex* associato

### Effetto:

- il thread chiamante si sospende sulla coda associata a **cond**, *rilasciando automaticamente il mutex mux*
- alla successiva riattivazione, il thread *riacquisisce automaticamente mux*.



## Variabili condizione: signal

Riattivazione di un thread che si era sospeso sulla *condition* **cond** associata *mutex* **mutex**:

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

### Effetto:

- a) se esistono thread sospesi nella coda associata a **pthread\_cond\_signal**, viene selezionato il primo;
  - Viene identificato il mutex associato **mutex**, che viene *prenotato* per il thread primo;
  - Quando il thread che ha eseguito **pthread\_cond\_signal** esegue (*obbligatoriamente*) **pthread\_cond\_unlock**, primo viene riattivato e **mutex** rimane attivo (--> primo torna in possesso della mutua esclusione);
- b) se non esistono thread sospesi nella coda associata a **pthread\_cond\_signal** l'esecuzione di **pthread\_cond\_unlock** determina il rilascio di **mutex**.

### Esempio: uso combinato di condition e mutex

Problema: risorsa utilizzabile da più thread, fino a un massimo di *max* (1)

*(soluzione conforme alle specifiche e ad (almeno una) implementazione: in rosso sono evidenziate le ambiguità e incongruenze)*

```
(max_utenti: condition, mux: mutex)
/*Fase di Entrata: necessaria la mutua esclusione*/
pthread_mutex_lock(&mux);
while(utilizzatori==max)
    pthread_cond_wait(&max_utenti,&mux);
utilizzatori++;
pthread_mutex_unlock(&mux);
```

```
<uso della risorsa>
/* permesso a non più di max utenti senza mutua esclusione */
```

```
/*Fase di Uscita: necessaria la mutua esclusione*/
pthread_mutex_lock(&mux);
utilizzatori--;
pthread_cond_signal(&max_utenti);
pthread_mutex_unlock(&mux);
```

**Problema:** risorsa utilizzabile da più thread, fino a un massimo di *max* (2)

*Soluzione non conforme alle specifiche (assume rilascio automatico di mux per effetto della **wait**, ma non la riacquisizione automatica per effetto della **signal**): non funziona correttamente con (almeno una) implementazione: in rosso sono evidenziate differenze rispetto alla soluzione precedente)*

```
(max_utenti: condition, mux: mutex)
/*Fase di Entrata: necessaria la mutua esclusione*/
pthread_mutex_lock(&mux);
while(utilizzatori==max)
    pthread_cond_wait(&max_utenti,&mux); pthread_mutex_lock(&mux);
utilizzatori++;
pthread_mutex_unlock(&mux);
```

```
<uso della risorsa>
/* permesso a non più di max utenti senza mutua esclusione */
```

```
/*Fase di Uscita: necessaria la mutua esclusione*/
pthread_mutex_lock(&mux);
utilizzatori--;
pthread_cond_signal(&max_utenti);
pthread_mutex_unlock(&mux);
```

**Problema:** risorsa utilizzabile da più thread, fino a un massimo di *max* (3)

*Soluzione che non contrasta con le specifiche (il thread che esegue signal non rilascia mux se signal ha l'effetto di riattivare un thread). Non funziona correttamente con (almeno una) implementazione: in rosso sono evidenziate differenze rispetto alla soluzione (1)*

```
(max_utenti: condition, mux: mutex)
/*Fase di Entrata: necessaria la mutua esclusione*/
pthread_mutex_lock(&mux);
while(utilizzatori==max) {
    sospesi_max_utenti ++; pthread_cond_wait(&max_utenti,&mux);
}
utilizzatori++;
pthread_mutex_unlock(&mux);
```

<uso della risorsa>

```
/* permesso a non più di max utenti senza mutua esclusione */
```

```
/*Fase di Uscita: necessaria la mutua esclusione*/
pthread_mutex_lock(&mux); utilizzatori--;
if sospesi_max_utenti > 0 {
    sospesi_max_utenti --; pthread_cond_signal(&max_utenti);
    /* riattiva un thread sospeso senza rilasciare la mutua esclusione */
}
else pthread_mutex_unlock(&mux);
```