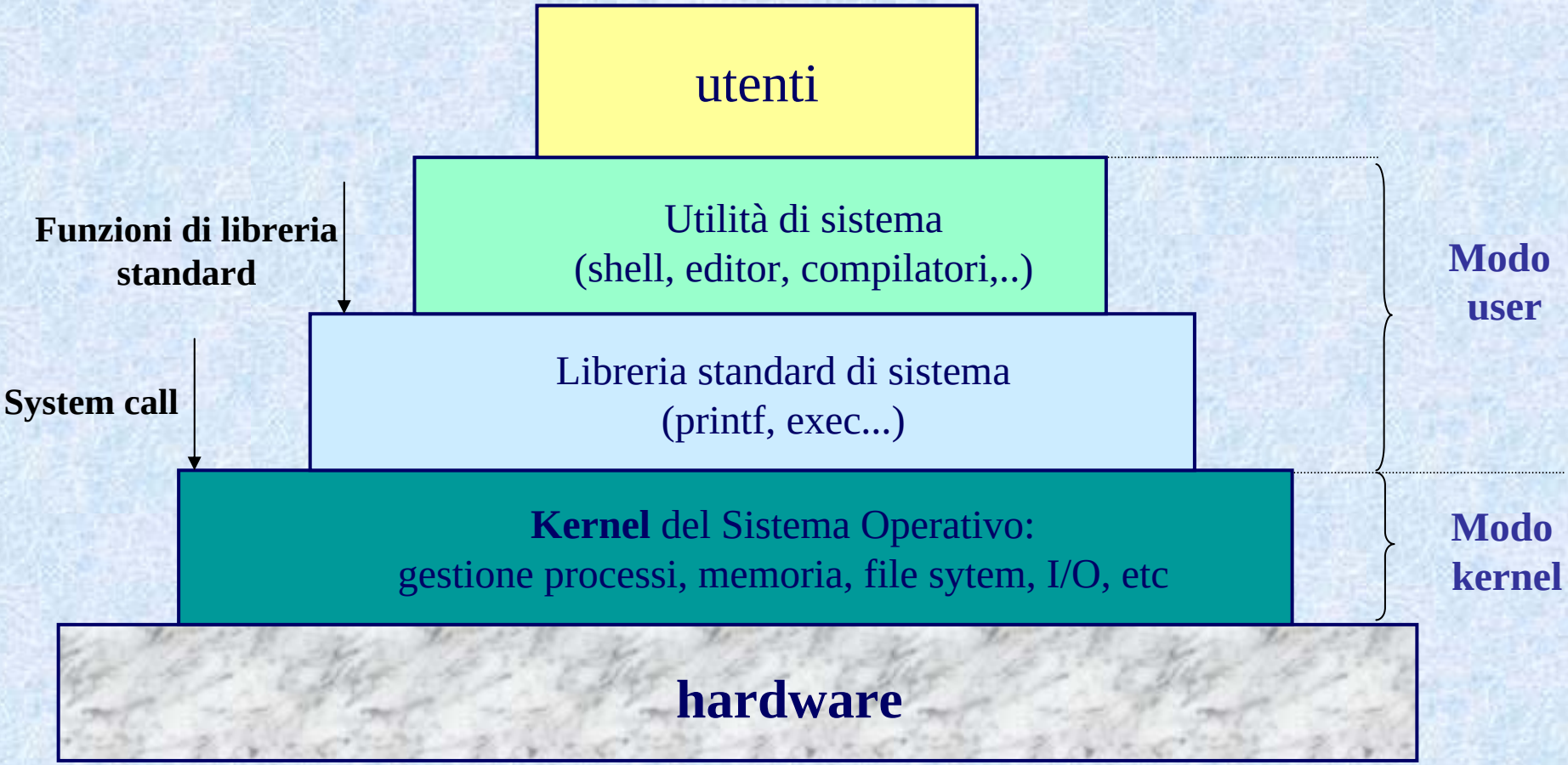
Processi e thread nei sistemi operativi Unix e Linux

Architettura di Unix



7.4 I processi Unix

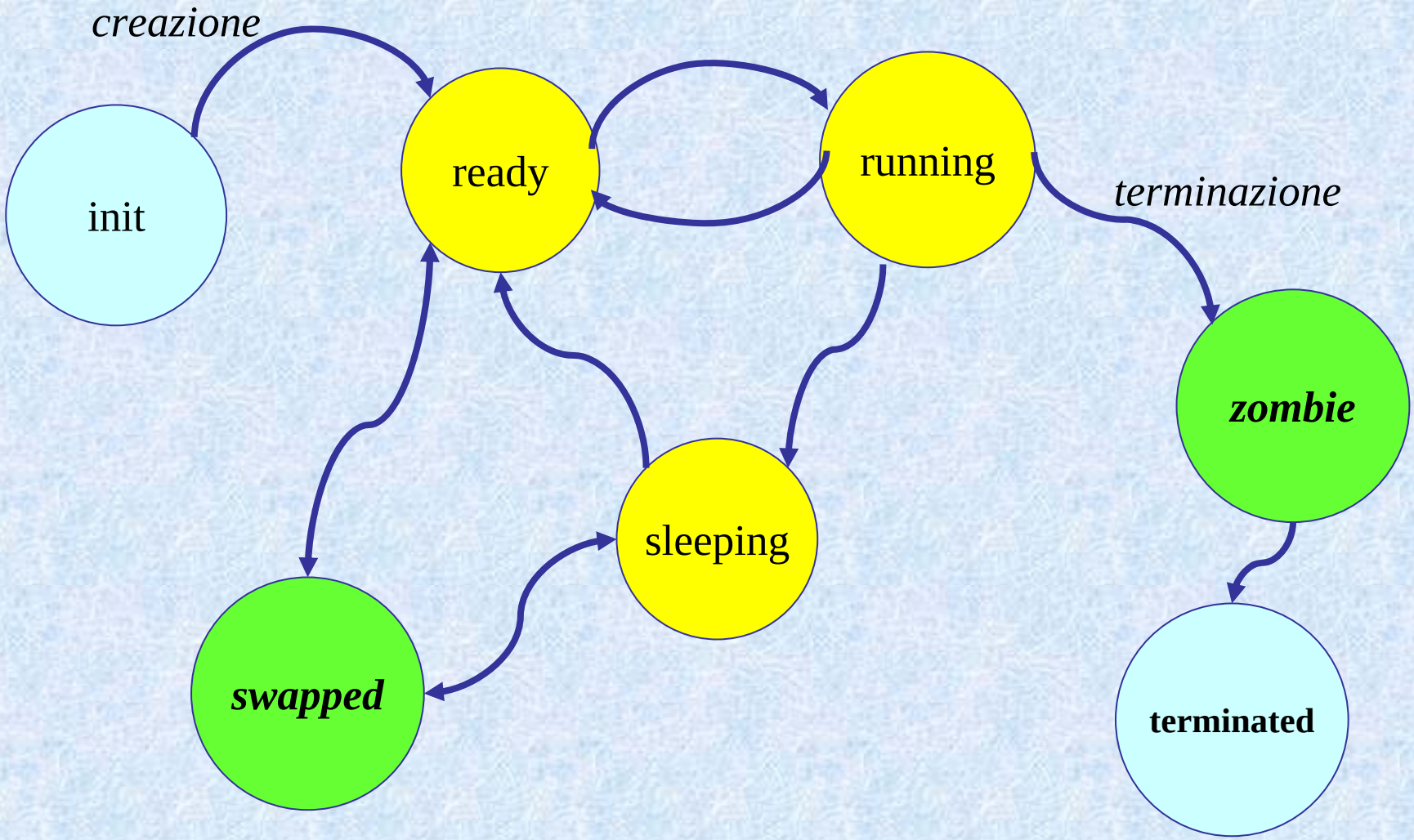
Unix è un sistema operativo *multiprogrammato* a *divisione di tempo*.

Caratteristiche del processo Unix:

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

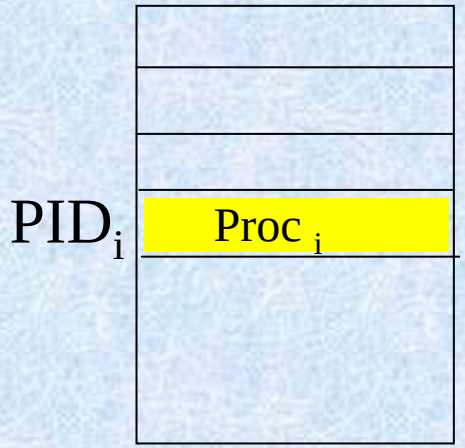
7.4.2 Rappresentazione dei processi Unix

Process Control Block (PCB):

- ✓ *Process Structure*
- ✓ *User Structure*

Process Table

Le ***Process Structure*** **sono contenute nella** ***Process Table:***



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:

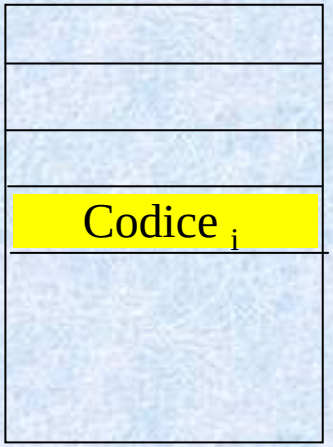
- **Parametri di scheduling**
- **informazioni sull'area/e di memoria che contiene testo, dati, stack e user area**
- **informazioni sui segnali pendenti (signal bitmap)**
- **stato**
- **PID, PID del padre**
- **user e group id**
- **puntatore alla User Structure**
- **riferimento indiretto al codice**

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

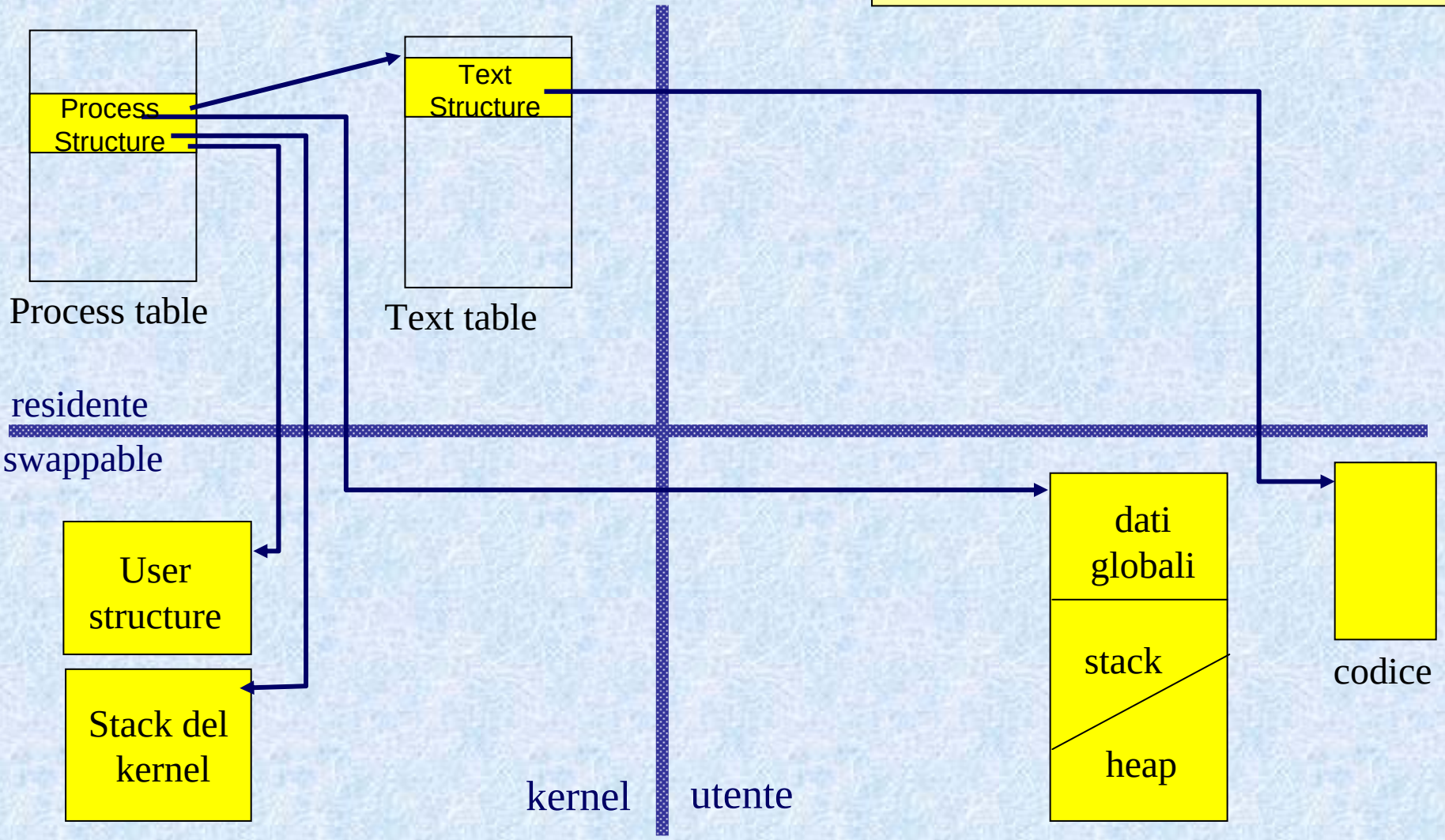


Il codice dei processi è rientrante: più processi possono condividere lo stesso codice (*text*)

Text table: 1 elemento ∀ 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

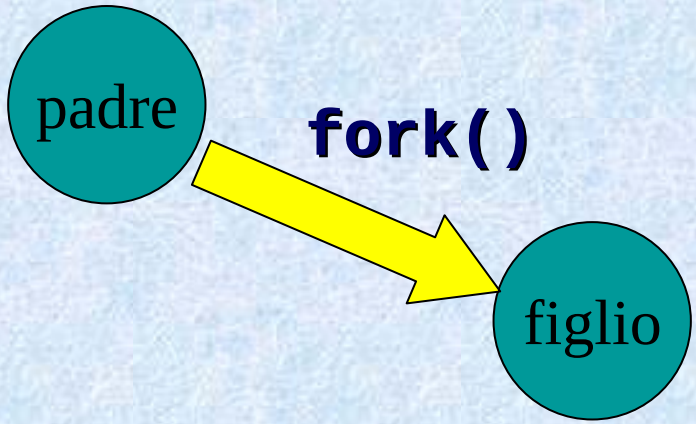
Chiamate di sistema per:

- 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**



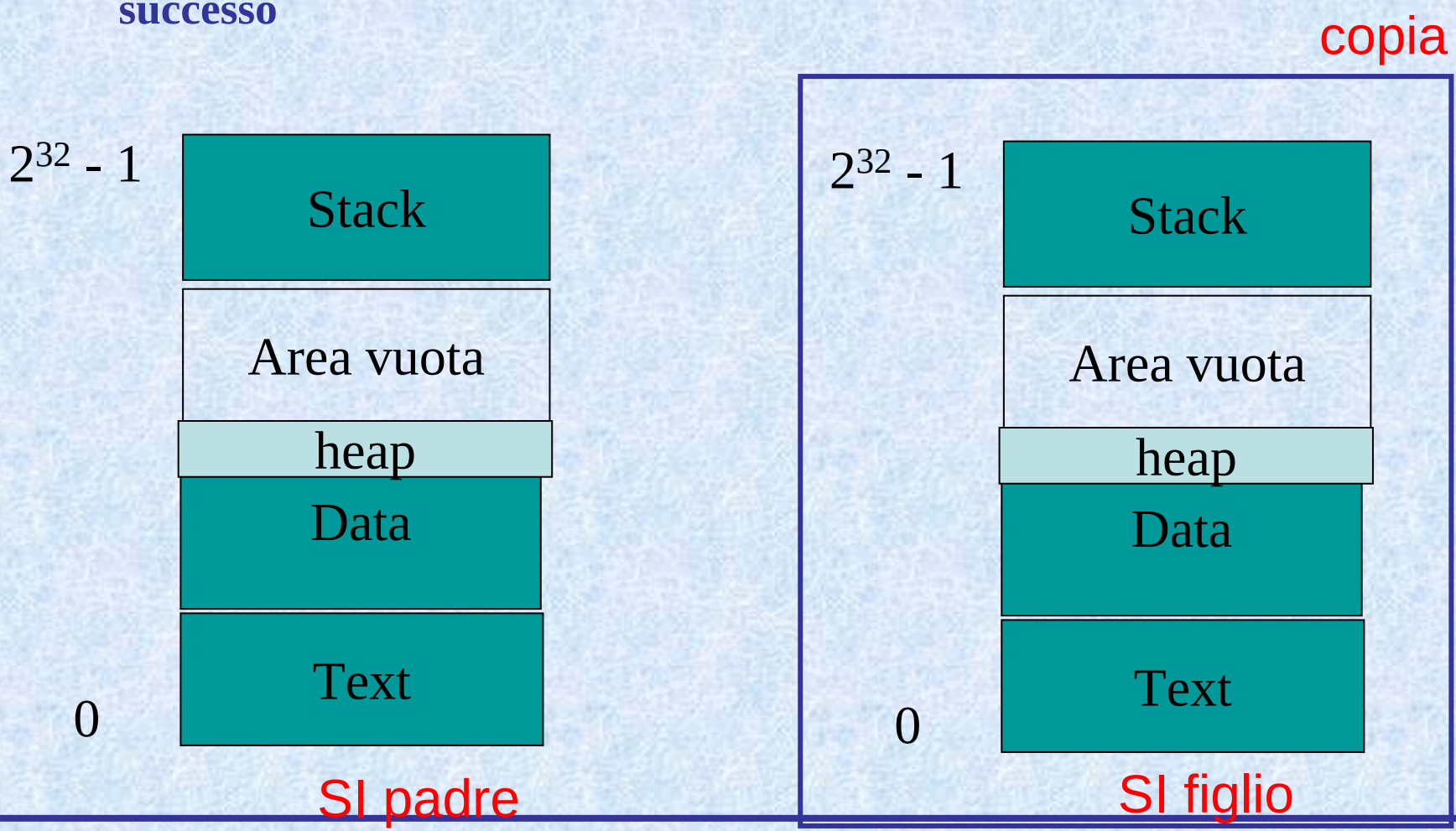
fork()

```
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

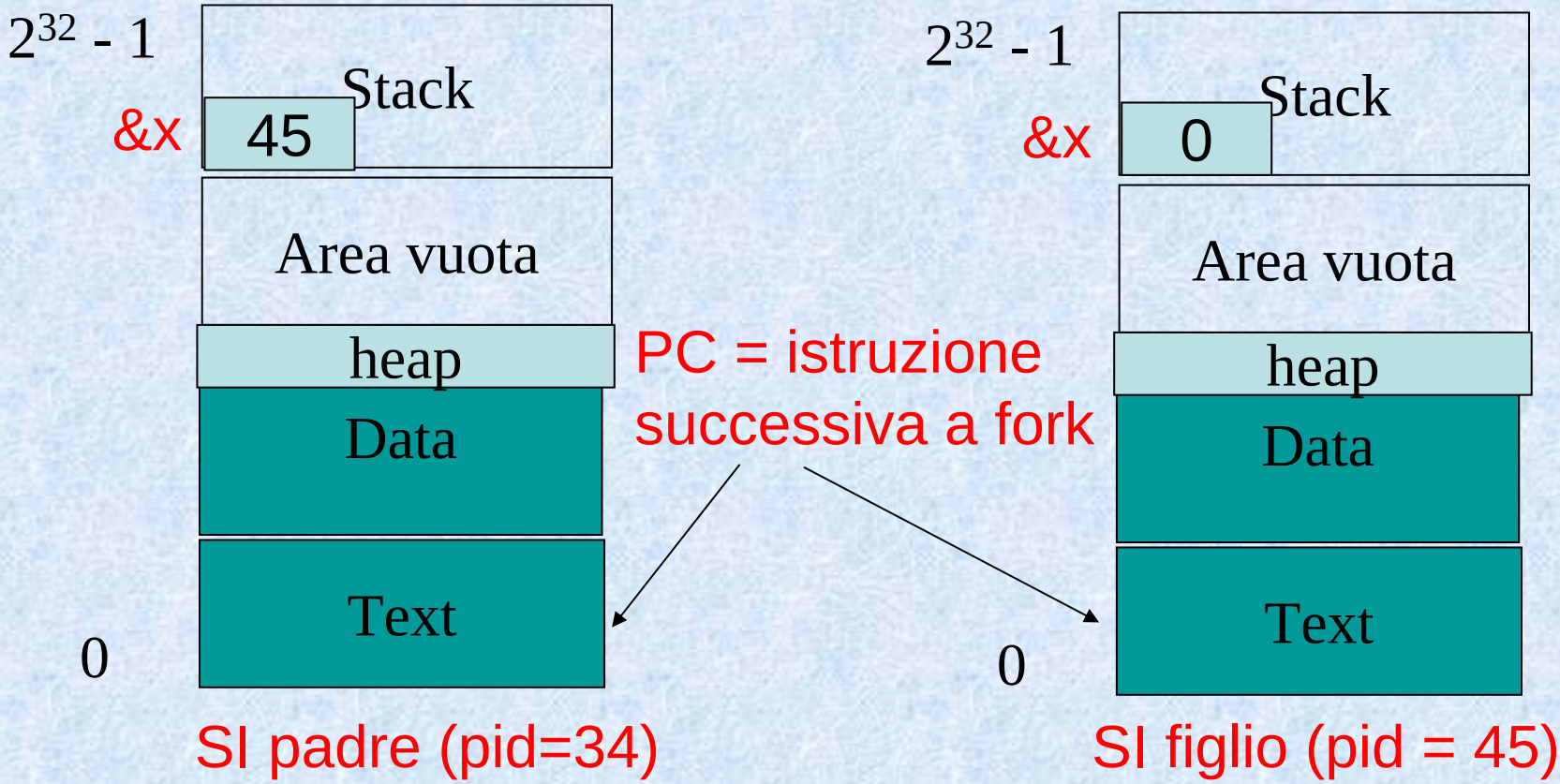
fork() (2)

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

execl()

```
int execl(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 dell' exec

Se l'exec ha successo non ritorna!!!

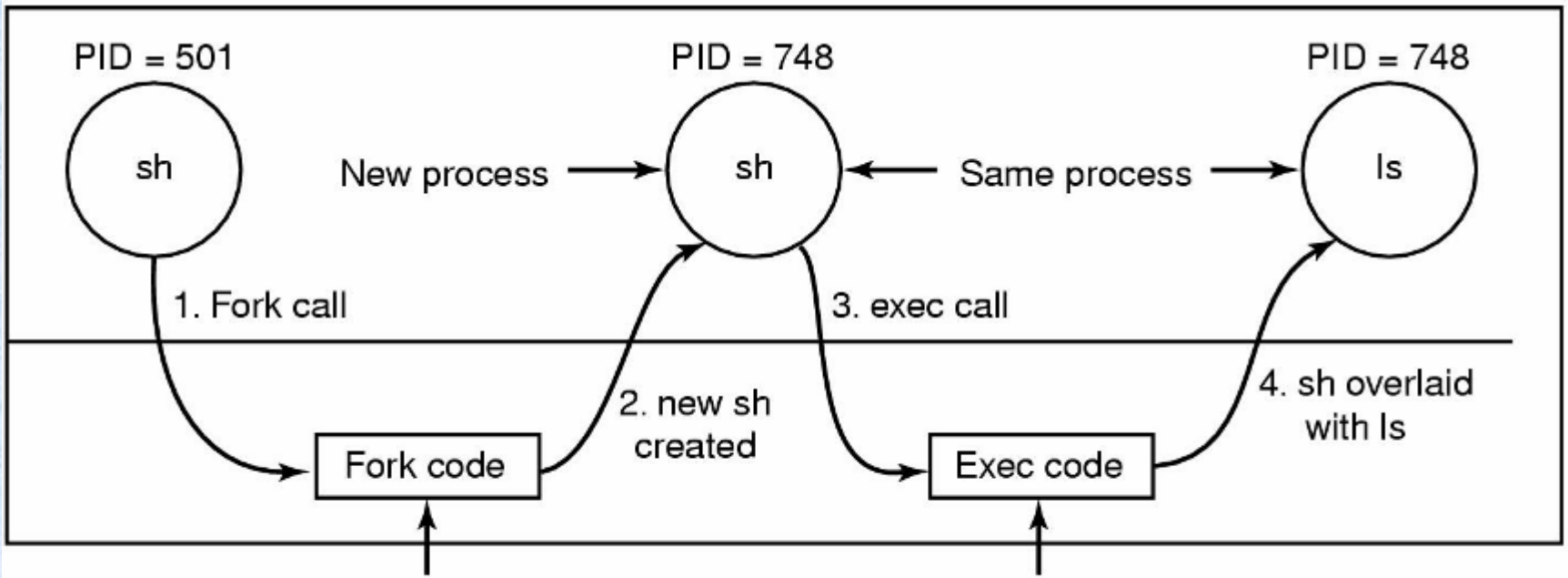
Altrimenti, in caso di fallimento restituisce un codice di errore

- Ad esempio non trova il file eseguibile,

Il processo dopo l'exec:

- **mantiene la stessa** process structure
- **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 lo stesso PID**
- **eventuali risorse allocate (es. file aperti) rimangono ancora assegnate al processo**

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



Allocate child's process table entry
Fill child's entry from parent
Allocate child's stack and user area
Fill child's user area from parent
Allocate PID for child
Set up child to share parent's text
Copy page tables for data and stack
Set up sharing of open files
Copy parent's registers to child

Find the executable program
Verify the execute permission
Read and verify the header
Copy arguments, environ to kernel
Free the old address space
Allocate new address space
Copy arguments, environ to stack
Reset signals
Initialize registers

Terminazione di processi

Un processo può terminare:

- ✓ **involontariamente (tentativi di azioni illegali, interruzioni, ecc.)**
- ✓ **volontariamente (ad esempio, mediante la system call `exit`)**

Terminazione del processo:

- **se il processo che termina ha** figli in esecuzione, **il processo `init` adotta i figli;**
- **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*.**

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

wait

Lo stato di terminazione può essere 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

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 ogni secondo:**

$$priorità = cpu\_usage + nice$$

cpu _usage : numero di clock tick per secondo che il processo ha avuto negli ultimi secondi

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

Scheduling in Unix

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

- **Fissiamo un intervallo di decadimento Δt**
- **I tick ricevuti mentre il processo P è in esecuzione vengono accumulati in una variabile temporanea *tick***
- **Ogni Δt**

  cpu _usage = cpu _usage / 2 + tick

  tick = 0
- **Il peso dei tick utilizzati descresce col tempo**
- **La penalizzazione dei processi che hanno utilizzato molta CPU diminuisce nel tempo**

Interazione tra processi

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

Segnali

Segnale:

interruzione software a un processo, che notifica un evento asincrono.

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 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)
- ritorna un puntatore a funzione:
  - ✓ 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 del segnale effetivamente ricevuto.**
- **l'*handler* non restituisce alcun risultato**

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

System call `kill`

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

```
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)

Segnali

Strutture Dati relative ai segnali

- *signal handler array* : descrive cosa fare quando arriva un segnale di un certo tipo
  - ignorare, trattare + indirizzo del codice della funzione da eseguire (handler)
- *pending signal bitmap* (signal mask): che contiene un bit per ogni tipo di segnale
  - il bit X è a 1 se c'è un segnale pendente di tipo X
- ogni processo ha signal handler array ed una pending signal bitmap

Segnali

Come ci si accorge della presenza di un segnale ?

- Quando un segnale viene inviato il kernel mette a 1 il corrispondente bit nella signal bitmap
  - più segnali sello stesso tipo in rapida sequenza possono essere visti come uno solo
- Il kernel controlla la signal bitmap ogni volta che ritorna da stato kernel a stato utente
  - es al ritorno da una SC, o dalla gestione di una interruzione
- Lo stato sleeping è lo stato nel quale un processo attende l'arrivo di un segnale
  - Appena ne arriva uno, viene risvegliato

Segnali

Cosa accade quando il kernel trova un segnale pendente

- Esegue l'azione richiesta
  - ignora, default o esegue il signal handler definito dall'utente
- Se deve essere invocato un signal handler definito dall'utente:
  - Il kernel modifica la user stack inserendo un frame per il signal handler e lo manda in esecuzione
  - Quando il signal handler è terminato si riprende l'esecuzione interrotta con il frame corretto

pipe

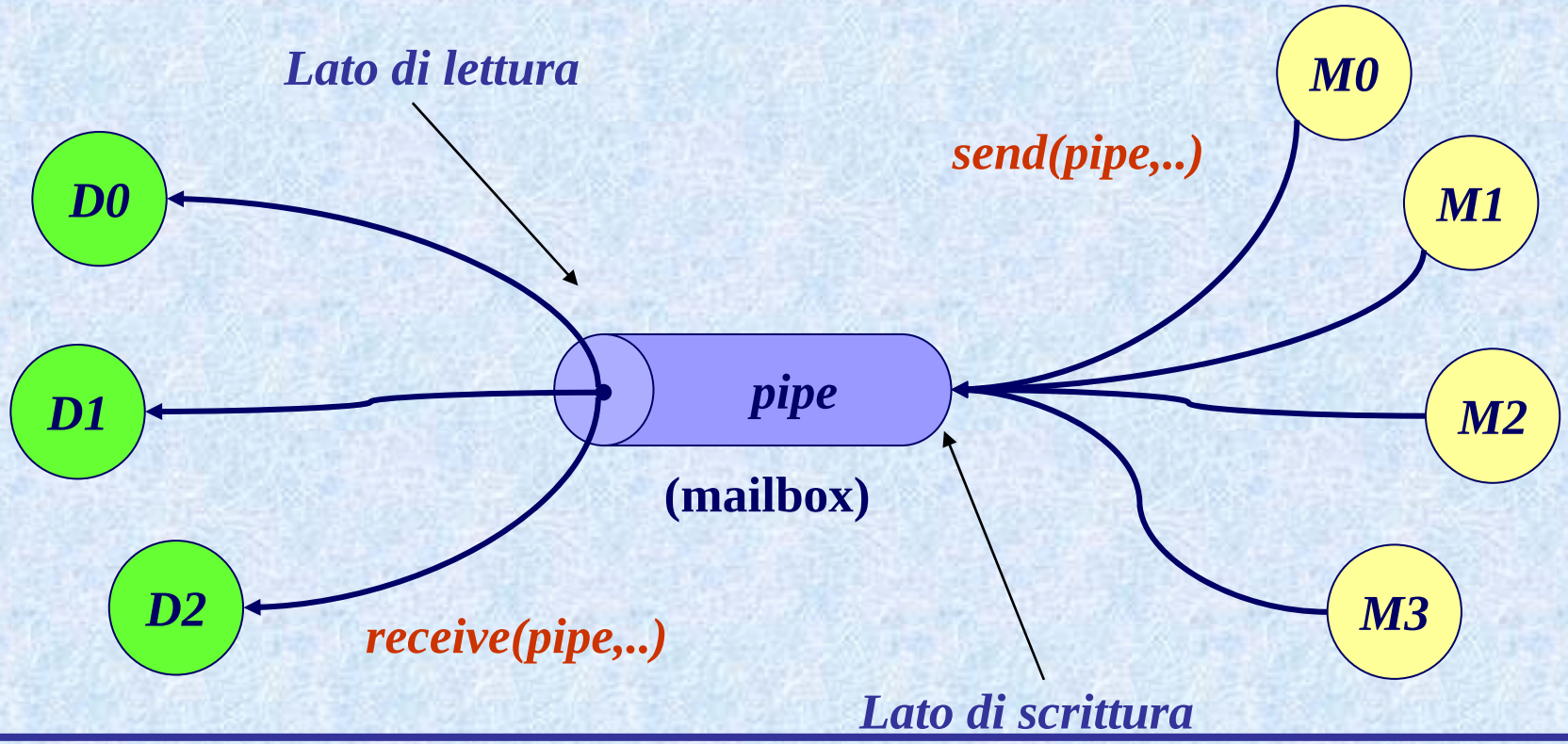
La pipe è un canale di comunicazione:

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

Comunicazione attraverso `pipe`

Mediante la pipe, la comunicazione tra processi è indiretta:

mailbox



System call `pipe`

Per creare una pipe:

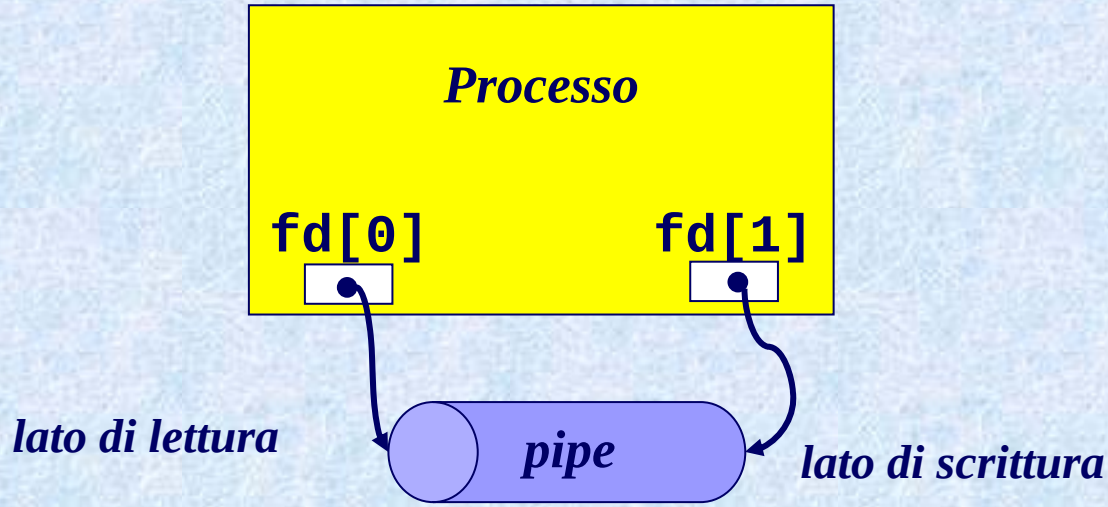
```
int pipe(int fd[2]);
```

- **`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 della pipe
  - ✓ **`fd[1]`** è il lato di scrittura della 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:

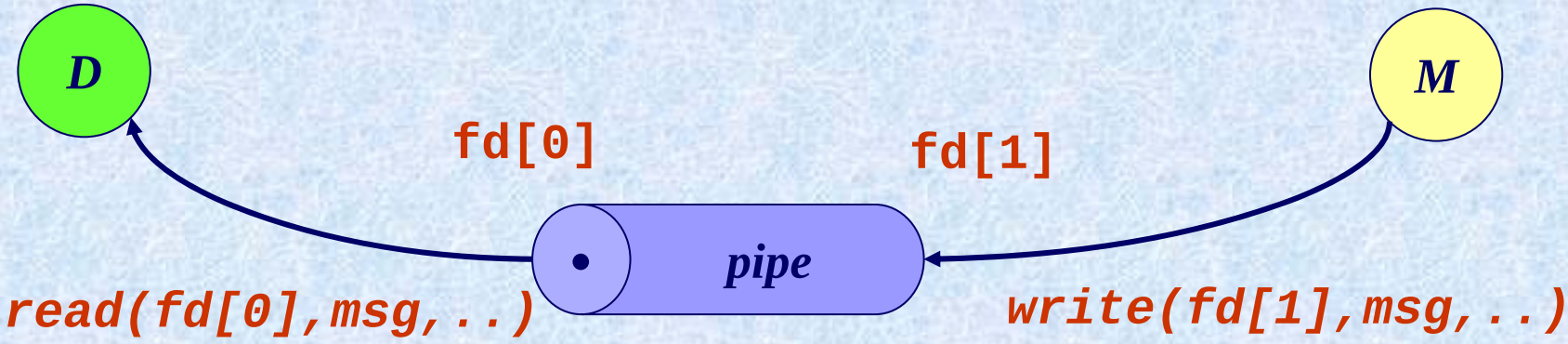
- vengono allocati due nuovi elementi nella tabella dei file aperti del processo e i rispettivi file descriptor vengono assegnati a **`fd[0]`** e **`fd[1]`:**
  - ✓ **`fd[0]`:** lato di lettura (***receive***) della pipe
  - ✓ **`fd[1]`:** lato di scrittura (***send***) della pipe



Accesso alla `pipe`

Ogni lato di accesso alla pipe è visto dal processo in modo omogeneo **al file (file descriptor):**

- **si può accedere alla pipe mediante le system call di accesso a file: `read`, `write`:**



- ✓ **`read`: realizza la** *receive*
- ✓ **`write`: realizza la** *send*

Sincronizzazione dei processi comunicanti

Il canale (la *pipe*) ha capacità limitata: sincronizzazione automatica dei processi:

- se la ***pipe*** è vuota: un processo che legge **si blocca**
- se la ***pipe*** è piena: un processo che scrive **si blocca**

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

Chiusura di `pipe`

- Ogni processo può chiudere un estremo della pipe con una `close`.
- Un estremo della pipe viene *effettivamente* chiuso 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***: **soltanto i processi appartenenti a una stessa gerarchia** possono scambiarsi messaggi; ad esempio:

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

pipe

La pipe ha due svantaggi:

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

Come realizzare la comunicazione tra processi di gerarchie diverse ?

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

LinuxThreads: rappresentazione dei threads

Il thread e` l'unità 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 thread

Creazione di thread:

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

dove:

- **`thread`**: è il puntatore alla variabile che raccogliera` il thread_ID (PID)
- **`start_routine`**: è il puntatore alla funzione che contiene il codice del nuovo thread
- **`arg`**: è 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)

Restituisce 0 in caso di successo, altrimenti un codice di errore (!=0)

LinuxThread: terminazione di thread

Un thread può terminare chiamando:

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

dove:

- `retval`: e` il puntatore alla variabile che contiene il valore di ritorno

È una chiamata senza ritorno.

LinuxThread: terminazione di thread

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 particolare thread da attendere
- `thread_return:` è il puntatore alla variabile dove verrà memorizzato il valore di ritorno del thread (v. `pthread_exit`)

LinuxThread: sincronizzazione

Lo standard POSIX 1003.1c (libreria **<pthread.h>**) definisce i semafori binari (o *mutex*)

- **sono semafori il cui valore può 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**

- **Rappresentazione di mutex: e` disponibile il tipo**
  pthread_mutex_t

Inizializzazione di mutex

L'inizializzazione di un **mutex** si puo`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

Locking/unlocking su un mutex **mux** 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 Thread in attesa del mutex, ne risveglia uno; altrimenti libera il mutex.

LinuxThread: variabili condizione

La variabile condizione è uno strumento di sincronizzazione che premette ai thread di sospendere la propria esecuzione in attesa che sia soddisfatta una condizione su dati condivisi.

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.

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, 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.