

Processi – Concetti di base

Esecuzione parallela e sequenziale

Il concetto di processo

Gestione dei processi

Esecuzione sequenziale

- **I sistemi di calcolo più semplici consentono unicamente l'esecuzione sequenziale**
 - Un programma può essere eseguito se e solo se il programma precedente è terminato
- **Questo schema presenta alcuni vantaggi**
 - La semplicità
 - Il determinismo della sequenza e dei tempi di esecuzione
 - L'assenza di conflitti per l'accesso alle risorse condivise
 - Software: File, Dati
 - Hardware: Memoria, Dispositivi
- **... ma mostra diversi limiti**
 - Accesso di un solo utente alla volta
 - Esecuzione di un solo programma alla volta
 - Mancanza del supporto per applicazioni distribuite
 - Scarso sfruttamento delle risorse hardware
 - Dispositivi
 - Processore/i

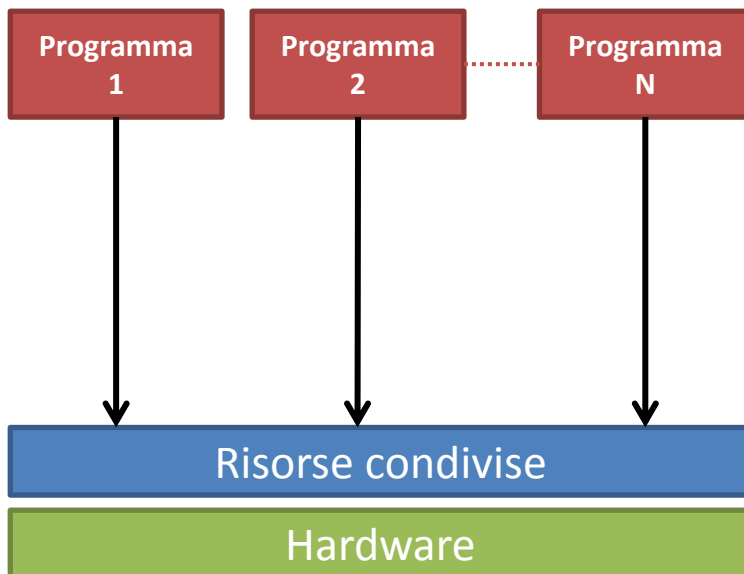
Esecuzione parallela

- **I moderni sistemi di calcolo offrono il supporto all'esecuzione parallela**
 - Più programmi eseguiti contemporaneamente
 - Il parallelismo è "simulato", se la macchina hardware dispone di un solo processore
 - Il parallelismo è "reale", se la macchina dispone di più processori
- **Questo schema supera tutti i limiti dell'esecuzione sequenziale ma introduce alcuni problemi**
 - Il determinismo non è garantito automaticamente
 - L'ordine di esecuzione dei programmi non è fissato
 - L'esecuzione sequenziale di più programmi deve essere gestita esplicitamente dall'utente o dal programmatore
 - I conflitti di accesso alle risorse devono essere gestiti
- **Questi problemi sono risolti grazie al Sistema Operativo**
 - In modo automatico
 - Scheduling, memoria virtuale, periferiche virtuali, ...
 - Offrendo opportuni servizi al programmatore
 - Gestione dei processi, sincronizzazione, ...

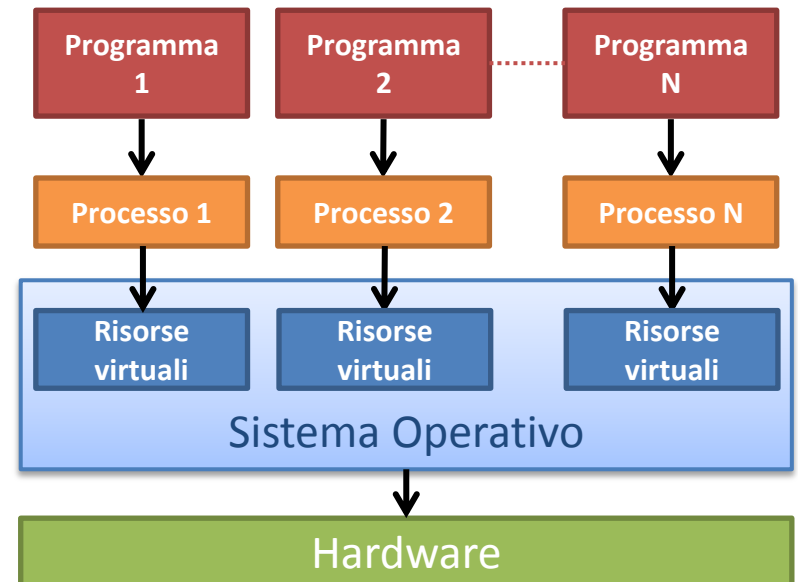
Processo

- A supporto dell'esecuzione parallela i sistemi operativi hanno introdotto il concetto di processo
- Un processo
 - Rappresenta una istanza di esecuzione di un programma
 - Può essere considerato un "esecutore virtuale"
 - Permette di gestire l'accesso concorrente alle risorse

Esecuzione sequenziale senza S.O.



Esecuzione parallela con S.O.



Processo

▪ Esecuzione sequenziale senza S.O.

- Un programma è eseguito solo quando termina il programma precedente

- Un programma accede direttamente alle risorse condivise
 - Non si hanno conflitti per costruzione

- Un programma accede direttamente ai driver delle periferiche e alle risorse condivise

▪ Esecuzione parallela con S.O.

- Tutti i programmi sono eseguiti contemporaneamente
 - Il sistema operativo gestisce l'alternanza dei processi mediante uno "scheduler"

- Un programma non accede direttamente alle risorse condivise
 - Il S.O. si occupa di gestire i conflitti
 - Un programma accede ad un insieme di risorse virtuali private associate al processo

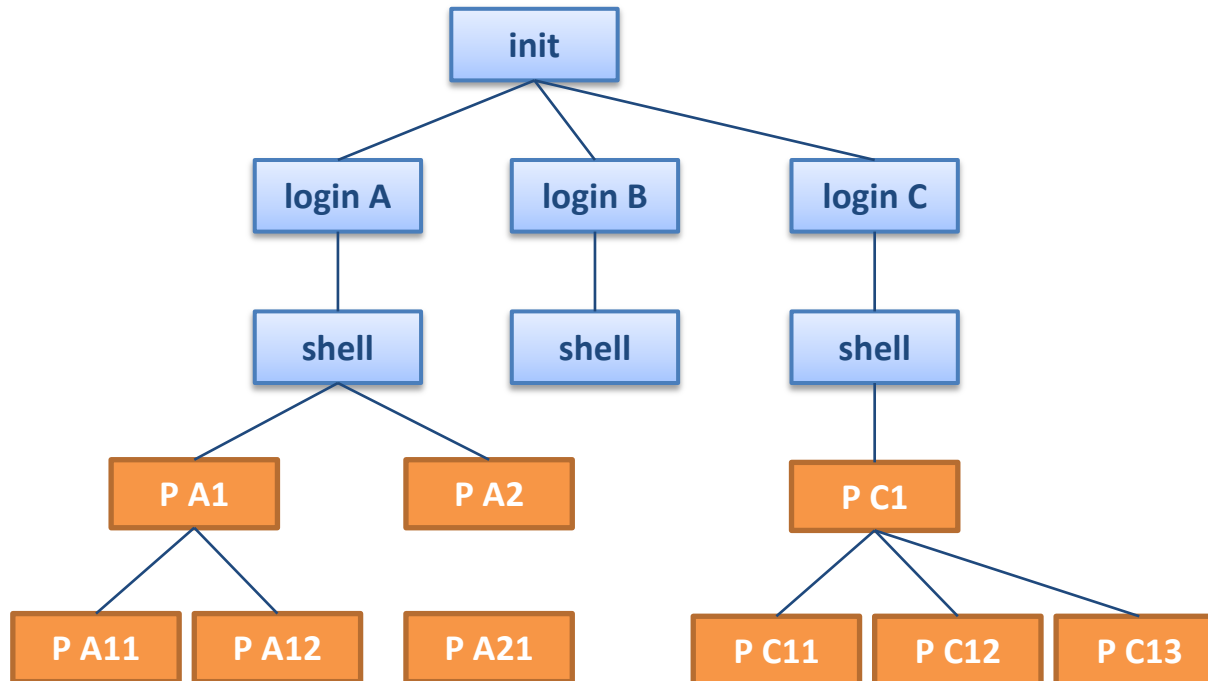
- Un programma delega al S.O. l'accesso alle risorse mediante "system call"

Processo

- **Un processo è identificato da un PID (Process Identifier) univoco**
 - Ogni processo è creato da un altro processo detto "padre"
 - Unica eccezione è il processo "init", il primo ad essere eseguito dal S.O.
- **Il sistema operativo mantiene una tabella che associa ad ogni PID una struttura dati detta Process Descriptor che contiene tutte le informazioni ad esso relative**
- **Il Process Descriptor contiene**
 - PID del processo padre
 - Informazioni sull'utente (uid, gid, permessi)
 - Informazioni sulla memoria
 - Riferimenti alle tabelle di sistema
 - Informazioni sui tempi di esecuzione
 - ...
- **Ogni processo dispone di un insieme di risorse virtuali**
 - Memoria
 - File
 - Periferiche

Gerarchia di processi

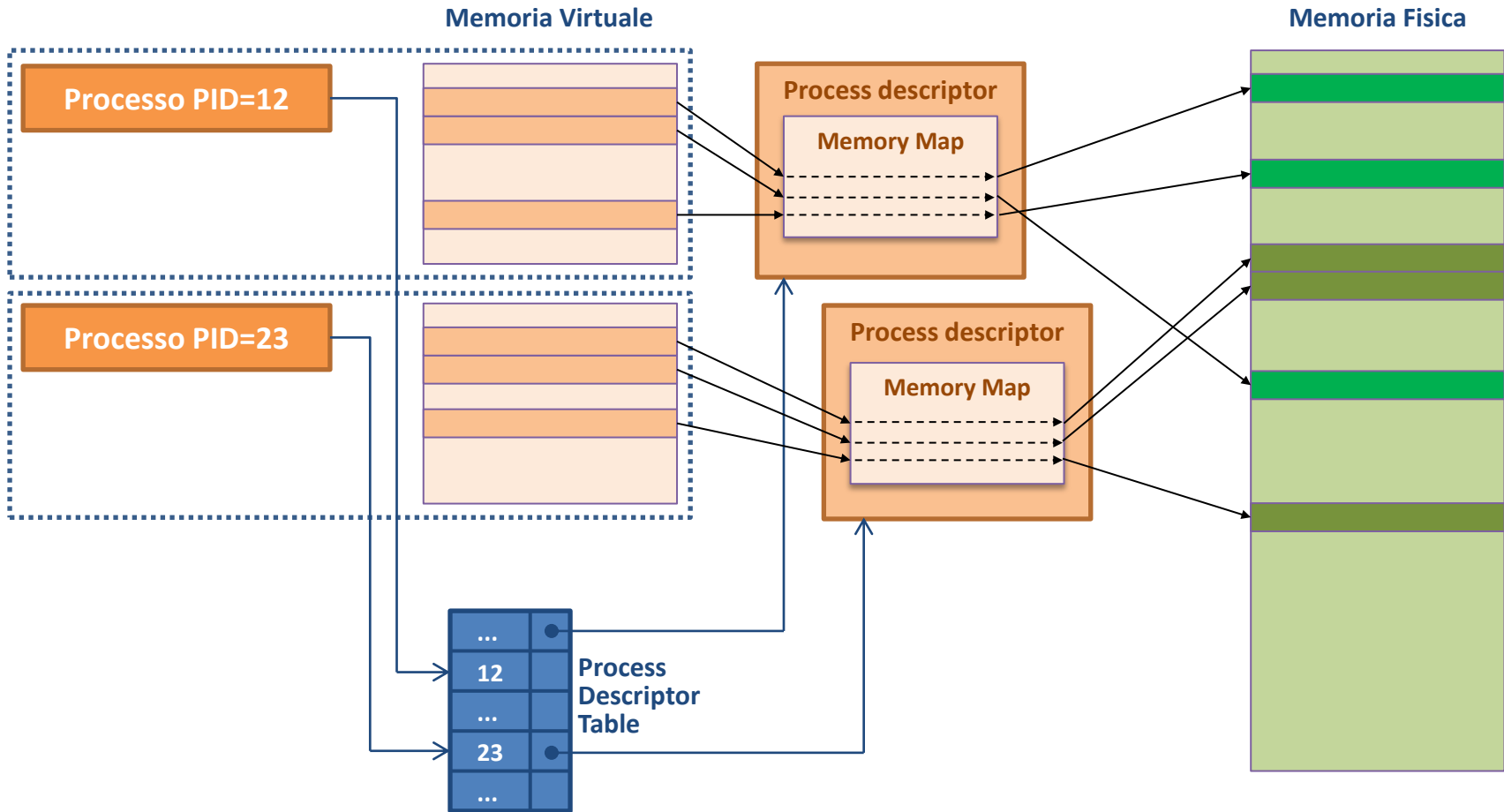
- Quanto segue è vero in generale, ma faremo riferimento specificamente ai sistemi operativi UNIX/Linux
- Tutti i processi sono figli del processo "init"
 - Da questo discende il processo "login" e la relativa "shell"
 - Da una shell discende la gerarchia dei processi utente



Modello di memoria

■ Un processo

- Non accede alla memoria fisica in modo diretto
- Accede ad una "memoria virtuale", mappata sulla memoria fisica mediante il PD



Modello di memoria

- Dal punto di vista logico, la memoria di un processo è organizzata in "segmenti"
- Ogni segmento contiene un tipo specifico di informazioni
 - Dati, codice, informazioni di sistema operativo
- Un semplice modello di memoria segmentata dispone dei segmenti seguenti
- **TEXT**
 - Contiene il codice eseguibile del programma
- **DATA**
 - Contiene i dati del programma. A sua volta è suddiviso in sezioni
 - DATA: Variabili globali esterne e statiche inizializzate
 - BSS: Variabili globali esterne e statiche non inizializzate
 - ROM: Costanti. A volte questa sezione fa parte del segmento TEXT
 - STACK: Stack del programma. Contiene le variabili automatiche
 - HEAP: Memoria dinamica. Contiene i dati allocati dinamicamente
 - SHARED: Memoria condivisa con altri processi
- **SYSTEM DATA**
 - Contiene i dati del programma gestiti dal sistema operativo
 - Descrittori di file, strutture dati dei thread, descrittori dei socket, ...

Operazioni sui processi

- **Creazione di un processo**
 - Il processo corrente crea un nuovo processo
 - Il processo creato (figlio) è identico al padre
- **Terminazione di un processo**
 - Termina un processo
 - Ritorna al padre un codice di terminazione o exit code
- **Attesa della terminazione di un processo**
 - Permette al processo padre di attendere la terminazione di un figlio
 - L'attesa può essere bloccante o non bloccante
- **Sostituzione del codice di un processo**
 - Sostituisce il codice eseguito da un processo con quello di un altro programma
 - Consente di creare processi che eseguono codice diverso da quello del padre
- **Analisi di un processo**
 - Permette di raccogliere informazioni sul processo corrente
- **Segnalazione**
 - Permette di inviare "segnali" ad un processo

Creazione

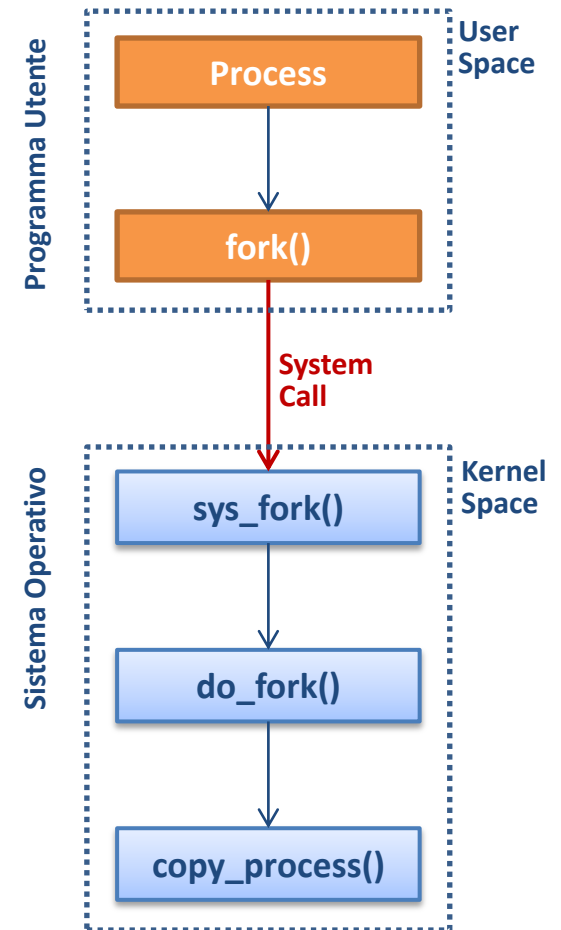
- La creazione di un processo avviene mediante la funzione:

```
#include <unistd.h>
pid_t fork(void);
```

- **L'effetto della chiamata è il seguente**
 - Viene generato un nuovo PID per il figlio
 - Viene creato un nuovo process descriptor
 - I segmenti dati e dati di sistema vengono copiati nella memoria del nuovo processo
 - Immediatamente dopo la creazione le due coppie di segmenti sono identiche
 - A meno del valore restituito dalla `fork()` stessa
 - Il codice del processo padre viene condiviso dal figlio
- **Dato che il codice è duplicato la funzione `fork()` "ritorna due volte", una volta nel processo padre e una volta nel processo figlio**
 - Nel padre restituisce
 - Il PID del processo figlio appena creato
 - Il valore -1 in caso di errore
 - Nel figlio restituisce sempre 0

Creazione

- Vediamo le operazioni effettuate dal programma e quelle delegate al S.O.
- **fork()**
 - Inizia la creazione del processo e prepara le informazioni per la chiamata di sistema
- **sys_fork()**
 - E' la versione della fork() in kernel space
- **do_fork()**
 - Genera il nuovo PID
 - Chiama `copy_process()` con gli argomenti opportuni
 - Al ritorno da `do_fork()` inizia l'esecuzione del nuovo processo
- **copy_process()**
 - Crea un nuovo descrittore di processo e vi copia alcune informazioni del processo padre
 - Copia le varie porzioni (segmenti e sezioni) del processo padre nel processo figlio



Creazione

- **Spesso un processo genera un figlio per eseguire un programma differente**
- **Ciò è possibile**
 - Creando un figlio identico al padre
 - Sostituendo il codice del figlio con quello del programma desiderato (vedi oltre)
- **In questo caso è verosimile che nel processo figlio non avvenga alcuna modifica delle variabili tra il momento in cui il figlio inizia l'esecuzione e il momento in cui il suo codice (e quindi i suoi dati!) vengono sostituiti**
- **Ciò rende superflua la copia dei segmenti dati nello spazio di memoria del figlio**
 - In generale tale copia può essere molto onerosa
- **Esistono due soluzioni per evitare tale inutile copia**
 - La funzione standard `vfork()`
 - Funziona come `fork()`, ma non esegue la copia dei segmenti dati
 - Richiede di sostituire il codice del figlio immediatamente dopo la sua creazione
 - Una versione della funzione `fork()` che implementa il meccanismo "copy-on-write"
 - Non esegue la copia dei dati fintanto che il programma non richiede una scrittura in memoria
 - In tal caso copia tutti i dati del processo padre nella memoria del figlio

Terminazione

- La terminazione di un processo avviene mediante la funzione:

```
#include <stdlib.h>
void exit( int status );
```

- **L'effetto della chiamata è il seguente**
 - Vengono eseguite operazioni di housekeeping
 - I file aperti vengono chiusi, la memoria viene rilasciata, ...
 - Viene salvato il valore dell'exit status nel descrittore del processo
 - In questo modo potrà essere recuperato dal padre mediante le funzione wait() o waitpid()
 - Si noti che il PID non viene rilasciato e il descrittore non viene distrutto
 - Viene segnalata al processo padre la terminazione di un figlio
 - Viene inviato al padre il segnale SIGCHLD (vedi oltre)
 - La funzione exit() non ritorna
- **Se un processo figlio termina prima che il padre abbia invocato la funzione wait(), il processo figlio diventa "defunct" o "zombie"**
 - Il processo è terminato ma il descrittore non può essere rilasciato

Attesa della terminazione

- L'attesa per la terminazione di un processo avviene mediante le funzioni:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait( int* status );
pid_t waitpid( pid_t pid, int* status, int options );
```

- **Con alcune differenze, l'effetto delle due chiamate è il seguente**
 - Si sospende in attesa della terminazione di un processo figlio
 - L'esecuzione riprende nel momento in cui il processo corrente riceve il segnale SIGCHLD
 - Recupera il valore dello stato di uscita specificato dalla funzione exit() nel figlio
 - Rilascia il PID del figlio e rimuove il suo descrittore di processo
 - Restituisce il PID del figlio terminato
- **La funzione wait()**
 - Attende la terminazione di un figlio qualsiasi ed è sempre bloccante
- **La funzione waitpid()**
 - Può attendere un figlio o un insieme di figli specifici e può essere non bloccante

Attesa della terminazione – Exit status

- **La terminazione di un processo può avvenire**
 - In modo naturale
 - Mediante la funzione `exit()` o l'istruzione `return` nella funzione `main()`
 - A causa di un "segnale"
 - Terminazione, ...

- **Il valore salvato nella variabile passata alle chiamate `wait()` e `waitpid()` indica**
 - La ragione della terminazione: terminazione naturale o mediante segnale
 - Tale valore è salvato nel byte meno significativo della variabile
 - Il valore dello stato di uscita specificato nella funzione `exit()` del figlio
 - Tale valore è salvato nel byte più significativo della variabile

- **Ecco alcuni esempi**
 - `status = 0x2000` Terminazione naturale (0x00)
 Exits status 0x20

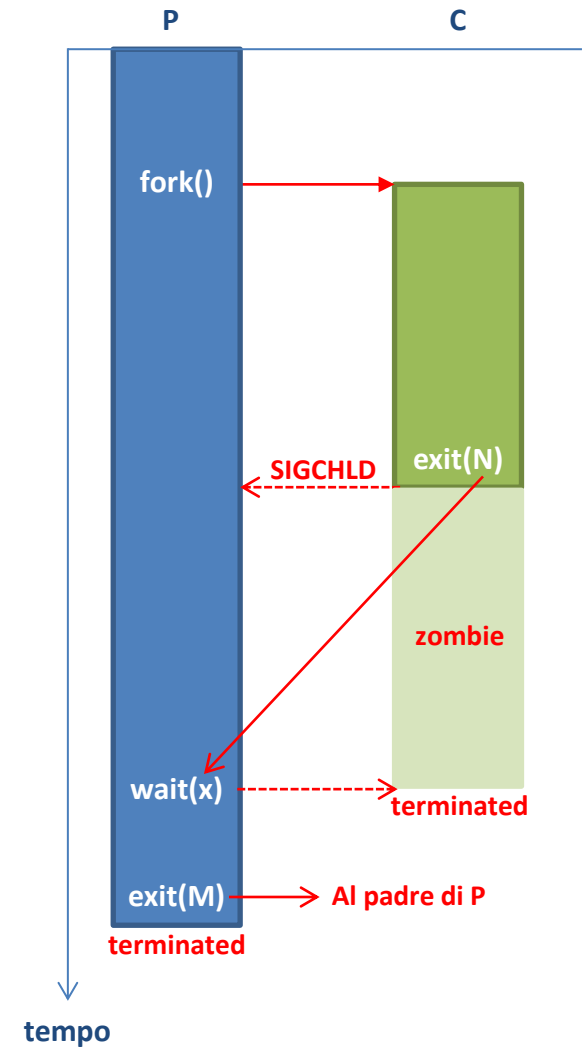
 - `status = 0x0009` Terminazione causata dal segnale SIGKILL (0x09)
 Il valore dello stato di uscita non è significativo

Attesa della terminazione – Exit status

- La libreria standard mette a disposizione alcune macro per analizzare il valore catturato dalle funzioni `wait()` e `waitpid()`
- **WIFEXITED(status)**
 - Ritorna vero se il figlio è terminato in modo naturale
- **WEXITSTATUS(status)**
 - Se il figlio è terminato in modo naturale, restituisce l'exit status passato dal figlio alla funzione `exit()`
- **WIFSIGNALED(status)**
 - Ritorna vero se il figlio è terminato a causa di un segnale
- **WTERMSIG(status)**
 - Se il figlio è terminato a causa di un segnale, restituisce il codice del segnale

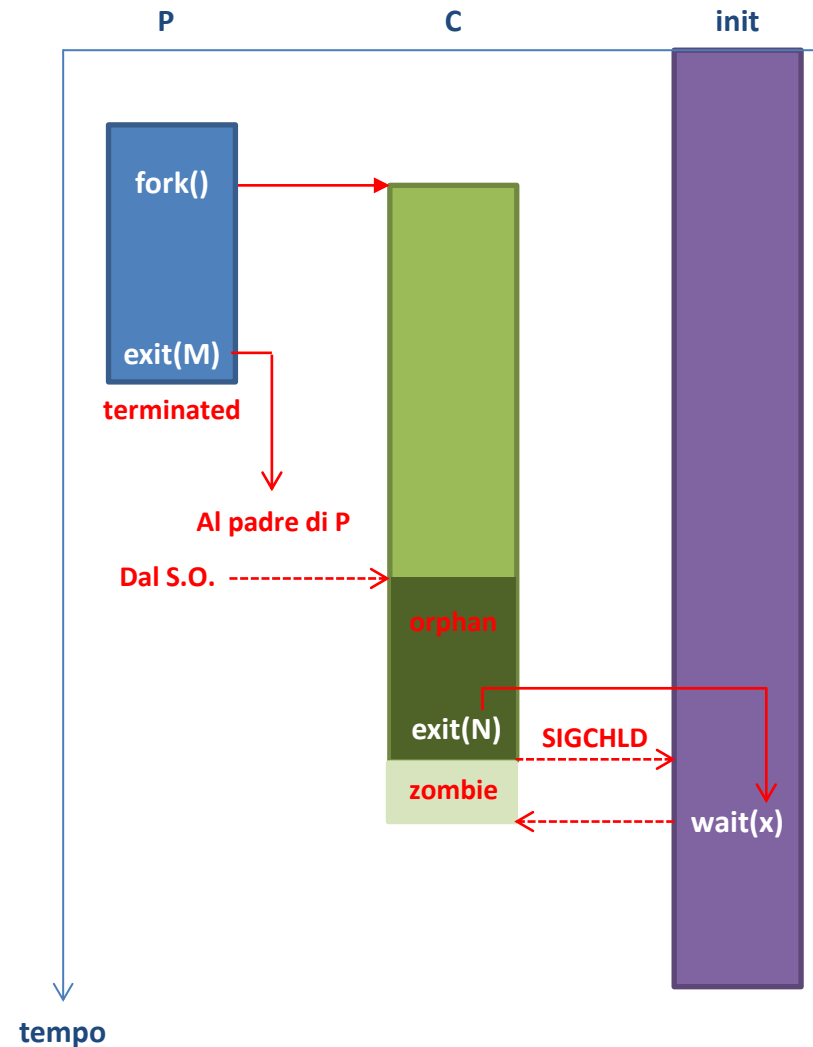
Processi zombie e orphaned

- **Lo stato di un processo dipende da diverse condizioni**
 - Sequenza di chiamata delle funzioni `exit()` e `wait()`
 - Sequenza di terminazione di padre (P) e figlio (C)
- **Un processo C è "zombie" quando**
 - Il processo C termina prima che il padre P si sia messo in attesa
 - Quando C termina
 - Le sue risorse vengono rilasciate
 - Il PID rimane nella tabella dei processi
 - Il descrittore (contenente l'exit status) è conservato
 - Quando il padre P esegue la chiamata alla `wait()`
 - Rileva l'exit status
 - Rilascia il PID e il descrittore del figlio C
 - Il figlio C a questo punto non esiste più
 - Se il padre P non esegue la chiamata alla `wait()`
 - Il figlio C rimane zombie



Processi zombie e orphaned

- Un processo C è "orphaned" quando
 - Il processo padre P termina prima del processo figlio C
 - Il padre non può più invocare la funzione wait() per il figlio
 - Per questo, il processo C viene "adottato" dal processo init, il cui PID è sempre 1
 - Quando il padre P è termina
 - Il sistema operativo deve verificare se esistono processi figli di P, nel nostro caso identificherà il processo figlio C
 - In tal caso modifica il descrittore del figlio C modificando il parent PID da quello di P a quello di "init", cioè 1
 - Quando figlio C termina
 - Il processo "init" riceve il segnale SIGCHLD
 - Esegue una wait() per consentire il rilascio del PID e del descrittore di C e la sua terminazione



Sostituzione del codice

- **La sostituzione del codice avviene mediante la funzione:**

```
#include <unistd.h>
int execl( const char* path, const char* arg, ... );
```

- **L'effetto della chiamata è il seguente**

- Sostituisce i segmenti TEXT e DATA del processo corrente con il codice e i dati del programma specificato mediante il primo argomento
 - Il nome del programma deve essere specificato completo del path
- Il segmento SYSTEM DATA non viene sostituito
 - Ciò significa che le risorse di sistema (file, socket, ...) sono disponibili nel nuovo programma
- Il processo non cambia, perciò descrittore e PID rimangono invariati

- **Al nuovo programma vengono passati gli argomenti specificati**

- Il primo argomento deve essere il path del programma, cioè deve coincidere con path
- Seguono gli argomenti veri e propri, seguiti dal valore NULL che indica la fine della lista

- **Nel programma è possibile accedere agli argomenti mediante i parametri argc e argv della funzione main()**

Sostituzione del codice

- Il passaggio degli argomenti al nuovo programma avviene come mostrato

Prima della sostituzione del codice: Chiamata della funzione `execl()`

```
ret = execl( prgpath, prgpath, arg1, arg2, ..., argN, NULL );
```

Dopo la sostituzione del codice:
Ingresso nella funzione `main()`

```
/*          argv[0]  argv[1]  argv[2]  ...  argv[N]  */  
main( int argc, char** argv ) {  
  
}
```

Sostituzione del codice

- **Esistono diverse varianti della funzione per la sostituzione del codice:**
- **execl(const char* path, const char* arg, ...)**
 - La funzione appena vista
- **execlp(const char* file, const char* arg, ...)**
 - Come execl() ma il nome del programma è specificato senza path e viene cercato dal sistema in tutte le directory specificate dalla variabile d'ambiente PATH
- **execle(const char* path, const char* arg, ..., char* const envp[])**
 - Come execl() ma passa alla funzione main anche l'ambiente
 - Ogni elemento dell'array envp[] è una stringa del tipo "variable=value\0" e la fine dell'array è indicata dal un puntatore NULL
- **execv(const char* path, char* const argv[])**
 - Come execl() ma passa gli argomenti nella stessa forma in cui main() li riceve mediante il parametro argv[], cioè un array di puntatori a stringhe terminato da un puntatore NULL
- **execvp(const char* file, char* const argv[])**
 - Come execv() ma il nome del programma è specificato senza path

Sostituzione del codice

- Tutte le funzioni viste sono front-end della funzione:
- `execve(const char* path, char* const argv[] , char* const envp[])`
 - Richiede il path completo del programma
 - Passa gli argomenti sotto forma di array di puntatori a stringhe terminato da NULL
 - Passa l'ambiente sotto forma di array di puntatori a stringhe terminato da NULL

Example – use of fork() to create 2 processes

- **The parent process creates two new children :**
 - compiler
 - editor
- **with the fork system call**
- **The new processes run the new programs compiler and editor**
- **At the end, the parent process wait until the two new children terminate**

```
void CreateProcess ( void ) {
    int pid1, pid2;
    char *argv[3] = {"compiler", "fileToCompile", 0};
    pid1 = fork();
    if( pid1 == 0 ) { // Child process code begins here
        execv( "compiler", argv ); // execute compiler
        // Child process code ends here.
        // execv does not return
    }
    // Parent executes here because pid1 != 0
    argv[0] = "editor";
    argv[1] = "fileToEdit";
    argv[2] = 0;
    if( (pid2 = fork()) == 0 )
        execv( "editor", argv );
    int reta, retb;
    int pida = wait( &reta );
    int pidb = wait( &retb );
}
```