POLITECNICO DI MILANO

Processi e Sincronizzazione

Laboratorio Software 2008-2009
C. Brandolese M. Grotto

Sommario

Processi

Concetti fondamentali

- È importante sottolineare la differenza tra i concetti di programma e di processo
- Un programma
  - È una sequenza di istruzioni che il calcolatore deve eseguire allo scopo di realizzare una funzione specifica
  - Risiede sui dispositivi di memorizzazione di massa
  - È identificato dal nome del file che lo contiene
  - È un concetto statico
- Un processo
  - È una istanza di un programma in esecuzione
  - Risiede nella memoria centrale
  - È identificato da un process-id (pid) univoco
  - È un concetto dinamico

Processi

Concetti fondamentali

- Nel sistema operativo GNU/Linux tutti i processi discendono dal processo `init`.
- La maggior parte delle primitive di gestione dei processi sono definite in `unistd.h`
- Utilizzare il tipo di dato `pid_t` definito in `sys/types.h`
- Il PID identifica univocamente un processo
  - PID = Process ID
    - Identificativo univoco del processo
  - PPID = Parent Process ID
    - Identificativo univoco del padre
- `pid_t getpid(void);`
  - restituisce il PID del processo che la invoca
- `pid_t getppid(void);`
  - ottiene il PID del processo padre

Esempio
`print-pid.c`

Processi

Stati in GNU/Linux

- `R` In esecuzione o pronto per l'esecuzione
- `S` Attesa interrompibile (in attesa di un evento)
- `D` Attesa non interrompibile (solitamente per operazioni di IO)
- `T` Arrestato (stopped)
- `X` Morto (dead, non dovrebbe mai comparire)
- `Z` Defunct (processo zombie)
- `<` Alta priorità (niceness)
- `N` Bassa priorità (niceness)
- `+` È nel gruppo dei processi in foreground

Creazione

Descrizione

- Un processo è creato dal sistema operativo ogni volta che si chiede l'esecuzione di un dato programma
- All'atto della creazione, il sistema operativo
  - Crea un nuovo descrittore di processo contenente
    - Il nuovo process-id
    - Lo stato del processo
    - Le informazioni di accounting (tempi di esecuzione)
    - I permessi
    - I riferimenti alla memoria allocata la processo
  - Copia il *codice del programma nella memoria centrale*
    - Inizializza il riferimento al codice del processo (programma)
  - Passa il controllo alla prima istruzione nel programma
- Il programma è ora in esecuzione in un nuovo processo

Creazione

Descrizione

- Un processo è sempre creato da un altro processo
  - Il processo originario prende il nome di parent process
  - Il processo creato prende il nome di child process
- Il child process è identico al parent process
  - Fanno riferimento allo stesso programma
    - Condividono la zona di memoria che contiene il programma
  - All'atto della creazione i dati dei due processi sono identici
    - La memoria dati del parent process è copiata nella nuova area di memoria assegnata al child process
  - L'esecuzione del child process inizia dal punto immediatamente successivo all'ultima istruzione eseguita dal parent process
  - A questo punto i due processi evolvono in modo indipendente

Creazione

Descrizione

- Due tecniche differenti
  - Primitiva `system()`
    - Consente l'esecuzione di un comando esterno
    - Molto semplice ma rischiosa in termini di sicurezza
  - Primitiva `fork()`
    - Consente di creare più processi figlio
    - Più complicata
    - Maggior flessibilità, velocità e sicurezza

Creazione

Creazione con `system()`

- `int system(const char *command);`
  - crea un processo che esegue la shell standard (`/bin/sh`) e gestisce il comando passato come parametro
    - `system("ls -l /");`
  - Restituisce il codice d'uscita del programma eseguito
    - 127 se la shell non può essere eseguita
    - -1 per altri errori
  - Soggetta alle limitazioni della shell
  - Preferibile usare la primitiva `fork()`

Esempio
`system.c`

Creazione

Creazione con `fork()`

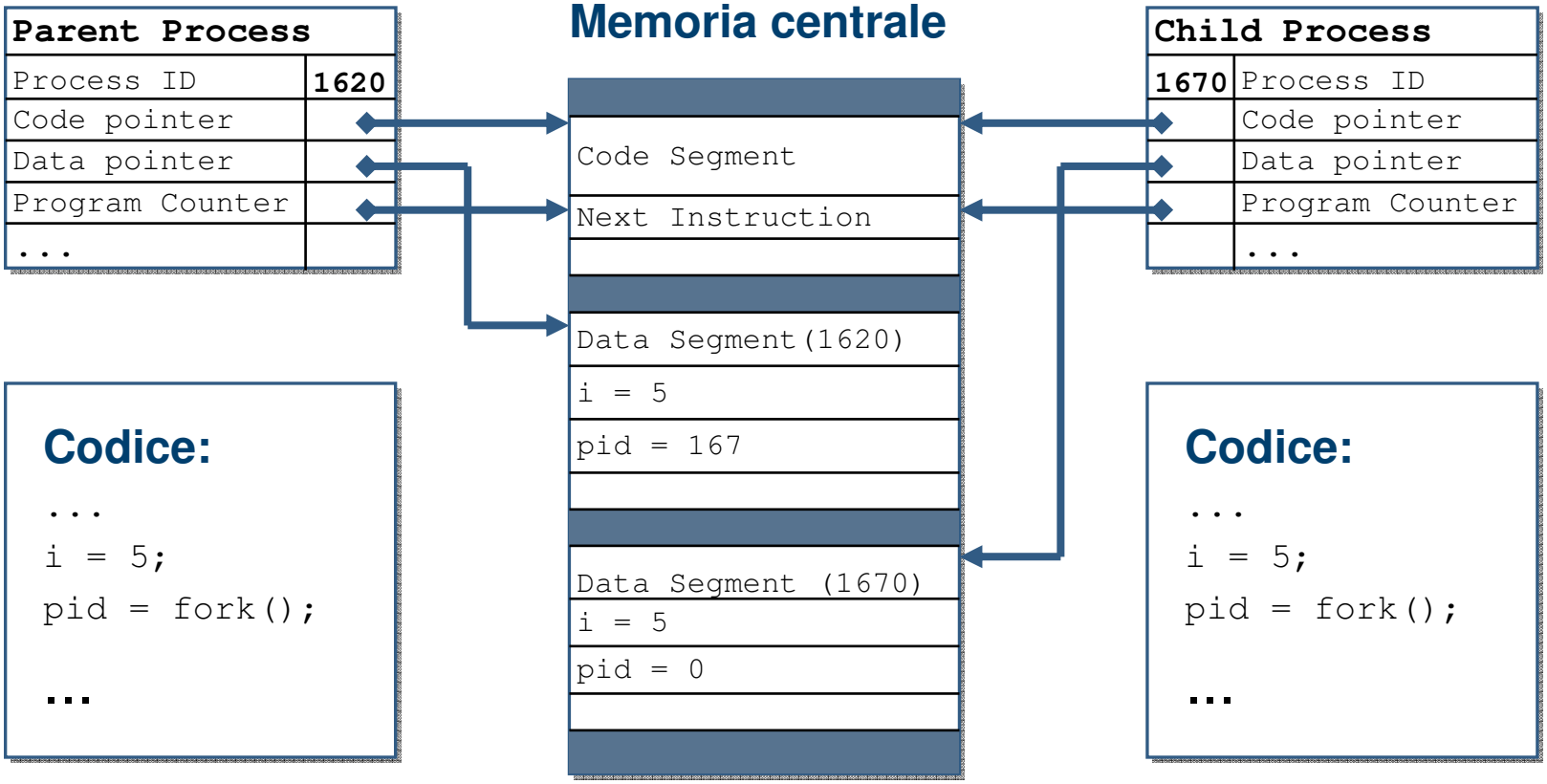
- `pid_t fork( void );`
  - Crea un nuovo processo
    - Il parent process è duplicato e da origine al nuovo child process
    - Inizialmente i due processi sono identici
  - È chaimata solo dal parent process, ma ritorna due volte
    - Nel parent process ritorna il pid del child process
    - Nel child process ritorna sempre il valore 0
  - Se la creazione del processo fallisce
    - Nel parent process ritorna -1
    - Il child process non è creato
  - L'esecuzione continua in entrambi i processi con l'istruzione immediatamente successiva alla chiamata di tale funzione

Esempio
`fork.c`

Creazione

Effetto di `fork()`

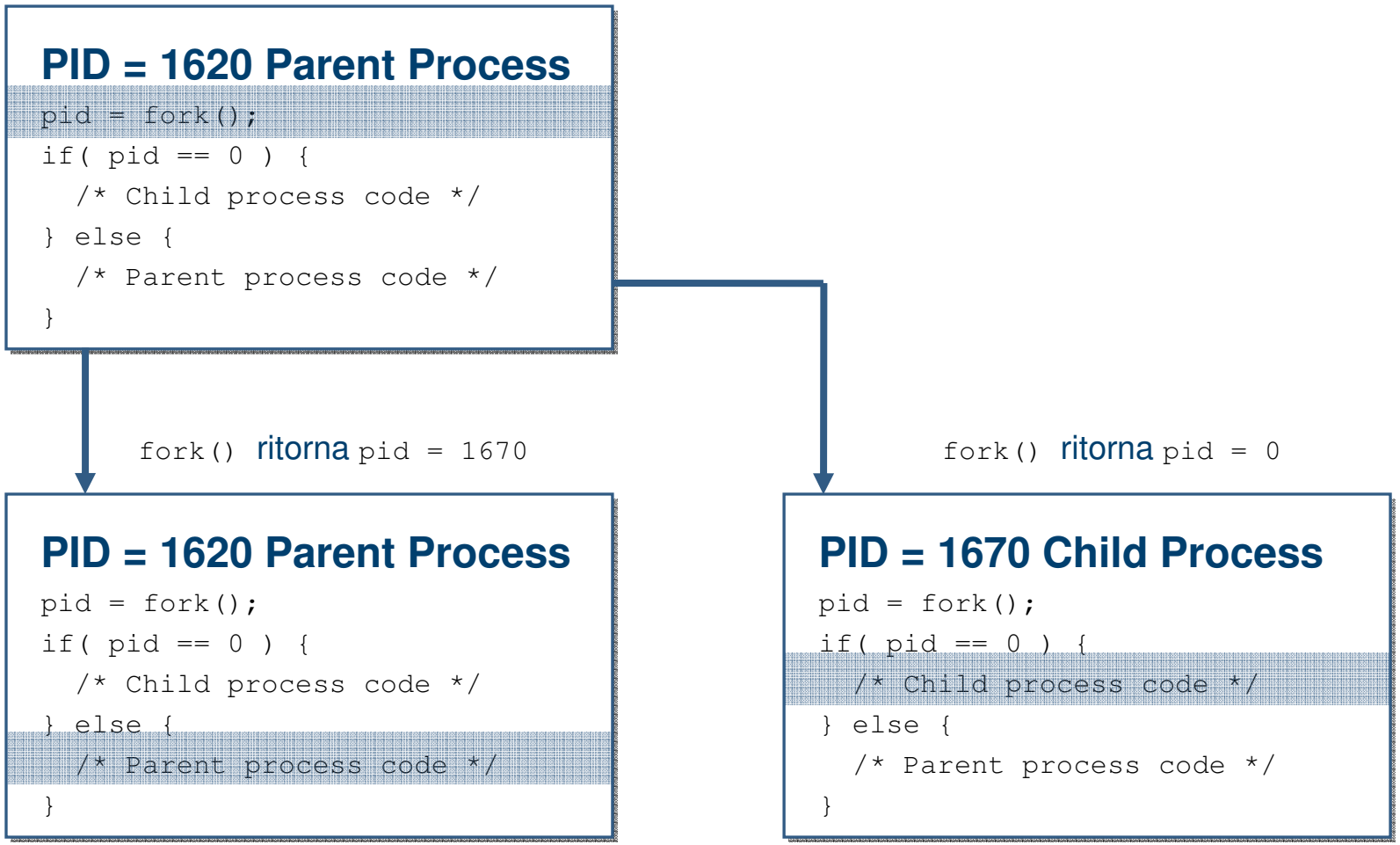
- Schematizzazione dell'effetto della funzione `fork()`



Creazione

Effetto di `fork()`

- In caso di successo si ha la seguente situazione



Sincronizzazione

Descrizione

- Una applicazione tipica:
  - Una applicazione esegue due gruppi di operazioni indipendenti
    - L'applicazione utilizza due processi differenti
    - I due processi devono potersi sincronizzare
- Una possibile soluzione è la seguente:
  - Il parent process si biforca
  - Il parent process effettua il primo gruppo di operazioni
  - Il child process effettua il secondo gruppo di operazioni
  - Il parent process si sospende in attesa della terminazione del child process, cioè si sincronizza
  - Il child process termina
  - Il parent process riprende l'esecuzione

Sincronizzazione

La funzione `wait()`

- `pid_t wait( int *status );`
  - La funzione `wait()` sospende l'esecuzione in attesa della terminazione di uno dei child process del processo chiamante
  - La funzione `wait()` è sempre bloccante
  - Se *`status`* è uguale a `NULL`
    - Ritorna il process id del processo terminato
  - Altrimenti
    - Ritorna il process id del processo terminato
    - Assegna alla variabile *`status`* un valore che indica Il codice di uscita del processo terminato edil segnale che ha causato la terminazione del processo
  - Dopo la crazione di un processo
    - Non è possibile stabilire l'ordine in cui il parent process ed il child process saranno eseguiti
  - La funzione `wait()` fornisce un semplice meccanismo di sincronizzazione

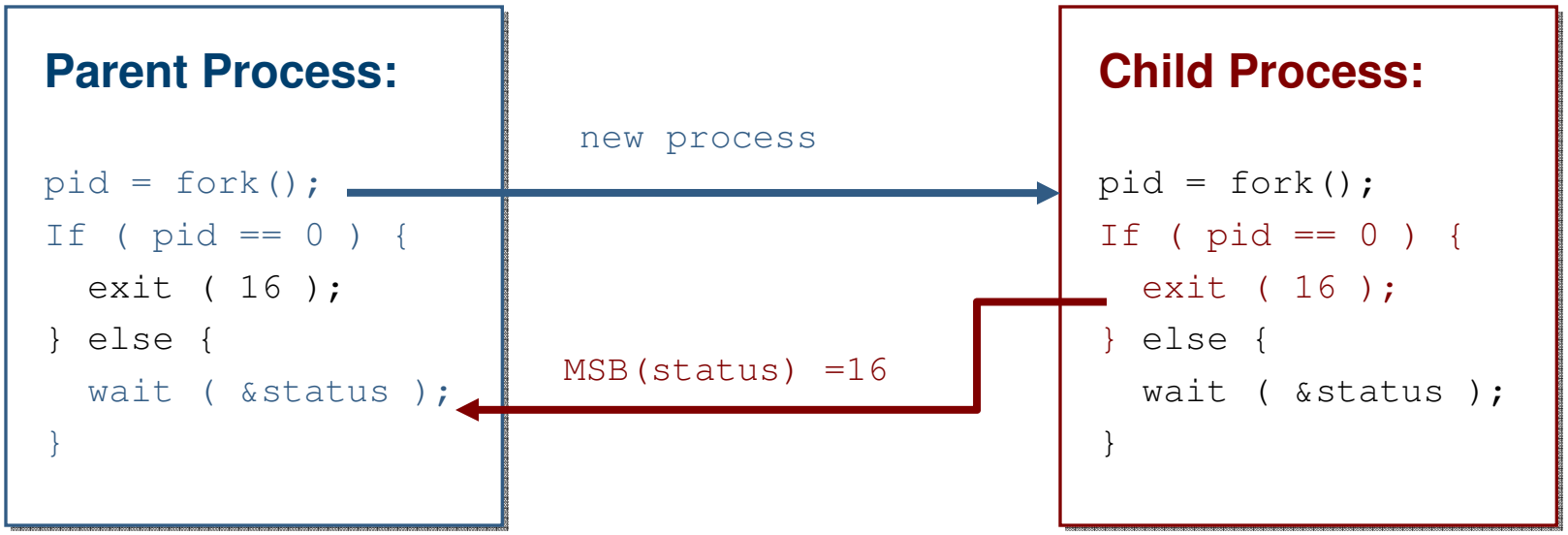
Esempio

`sync1.c sync2.c`

Sincronizzazione

La funzione `exit()`

- `void exit( int status );`
  - Causa la terminazione di un processo
    - Chiude tutti i descrittori di file (file, pipes, socket, ...)
    - Rilascia la memoria allocata
  - Comunica al proprio parent process un codice di uscita
  - Specificato dal byte più significativo della variabile ***status***
  - Il codice è rilevabile nel parent process con la funzione `wait()`



Sincronizzazione

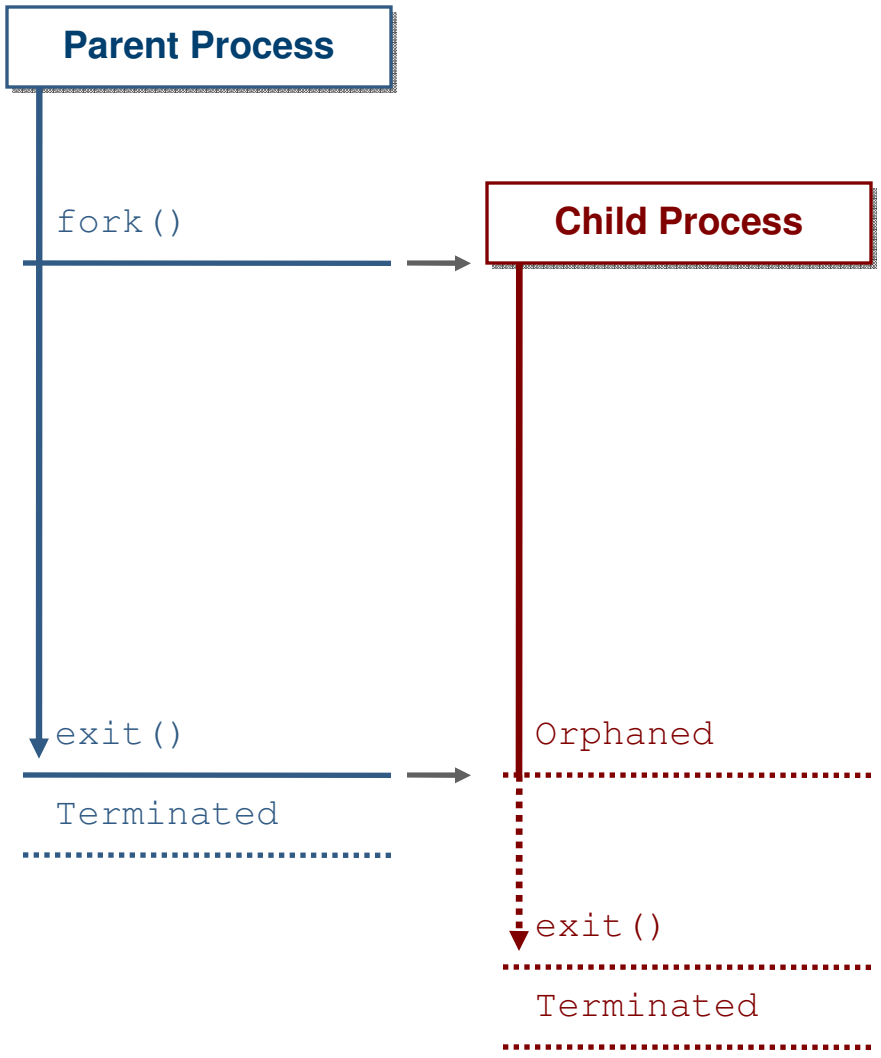
Utilizzo corretto

- La sequenza corretta di eventi dovrebbe essere la seguente
  - Il parent process crea il child process
  - A questo punto sia il parent sia il child process possono essere eseguiti e l'ordine in cui ciò avviene non è noto a prioiri
- Il parent process si mette in attesa della terminazione del child
  - Il parent process è quindi sospeso
- Il child process continua l'esecuzione ed infine termina
  - Chiamando la funzione `exit()`
- Il parent process rileva la terminazione del child ed eventualmente il suo stato di uscita
  - Chiamando la funzione `wait()`
- In alcuni casi questa sequenza non è rispettata
  - Si hanno condizioni anomale, gestite dal sistema operativo

Sincronizzazione

Processi orphaned

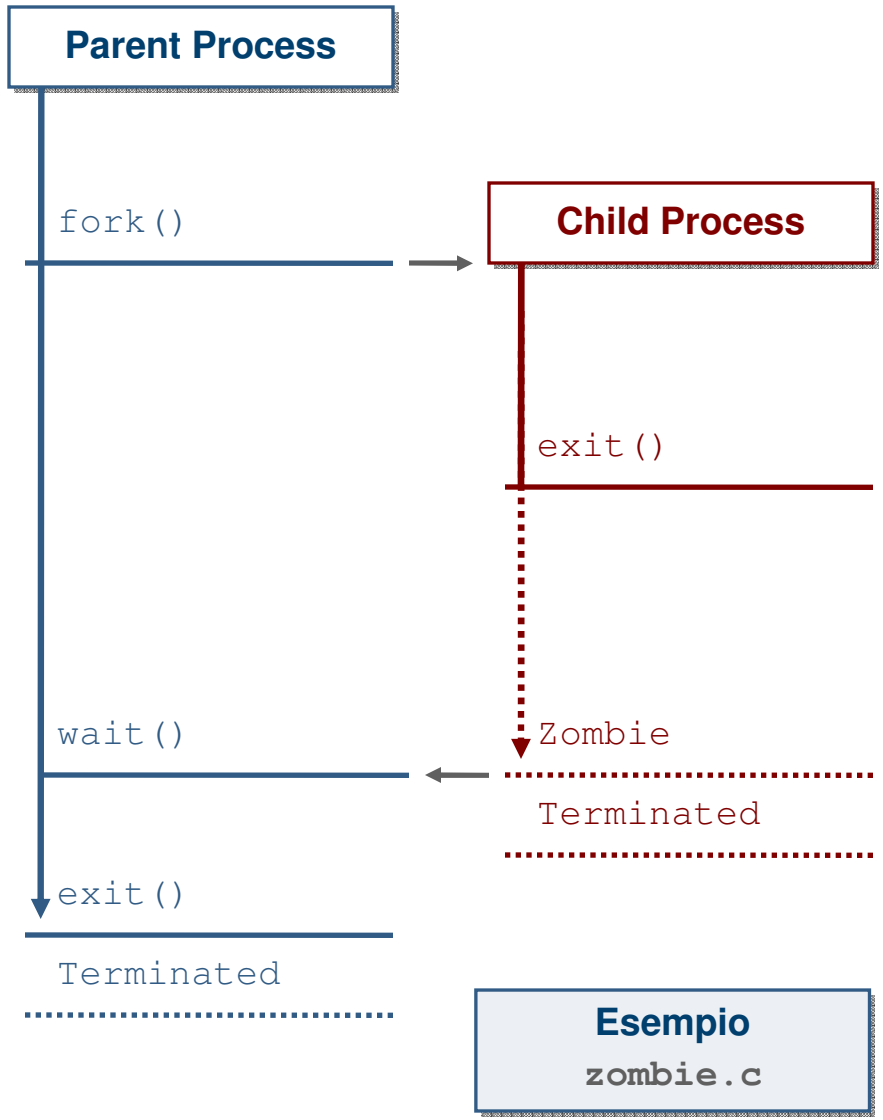
- Il parent process termina
  - Prima di avere effettuato una chiamata alla funzione **`wait()`**
  - Mentre il child process è ancora in esecuzione
- In questo caso il child process
  - Non ha più un parent process (è orfano)
  - È 'adottato' dal processo speciale **`init`**(con pid sempre uguale a 1)



Sincronizzazione

Processi zombie

- I child process
  - Termina prima che il parent abbia effettuato la chiamata alla funzione `wait()`
  - Quando il parent chiama la funzione `wait()`, rileverà lo stato del processo già terminato



Sincronizzazione

Rilevazione dello stato di uscita

- La funzione `wait()` permette di rilevare il codice di uscita di un processo e la causa della sua terminazione
- Si utilizzano alcune macro
  - `WIFEXITED ( status );`
    - Ritorna non-zero se il processo è terminato normalmente
  - `WEXITSTATUS ( status );`
    - Ritorna il codice di uscita del processo
  - `WIFSIGNALED ( status );`
    - Ritorna non-zero se il processo è stato terminato da un segnale
  - `WTERMSIG ( status );`
    - Ritorna il numero del segnale

Esecuzione di programmi

Descrizione

- Spesso è necessario eseguire un programma esterno
  - Il codice del programma risiede in un file
- A questo scopo è necessario
  - Mantenere il processo corrente ed il suo descrittore
  - Sostituire il codice del programma corrente con quello del programma da eseguire
  - Riprendere l'esecuzione del nuovo codice dalla prima istruzione
- Si utilizza una delle funzioni della famiglia `exec*()`
  - Svolgono tutte la stessa funzione
  - Si differenziano per
    - Modalità di passaggio degli argomenti della linea di comando
    - Possibilità di passare l'ambiente
    - Possibilità di ricerca dell'eseguibile secondo la variabile `PATH`

Esecuzione di programmi

Famiglia di funzioni `exec*()`

- `int execl( const char *path, const char *arg, ...);`
  - Esegue il programma *`path`*
    - Il nome dell'eseguibile deve essere completo del path
  - Argomenti sulla linea di comando
    - Sono passati mediante una lista di stringhe
    - Ogni stringa è un argomento
    - La lista deve essere terminata dal valore speciale *`NULL`*

```
int main( int argc, char** argv ) {
  char* cmd = "ls"
  char* dir = "/etc/init.d"
  /* Runs: 'ls -l -a /etc/init.d' */
  execl( "/bin/ls", cmd, "-l", "-a", dir, NULL );
}
```

Esecuzione di programmi

Famiglia di funzioni `exec*()`

- `int execlp( const char *file, const char *arg, ...);`
  - Esegue il programma *`file`*
    - Il nome dell'eseguibile è cercato dal sistema operativo in tutti i percorsi specificati dalla variabile di ambiente `PATH`
  - Argomenti sulla linea di comando
    - Sono passati mediante una lista di stringhe, una per argomento
    - La lista deve essere terminata dal valore speciale `NULL`

```
int main( int argc, char** argv ) {
  char* cmd = "ls"
  char* dir = "/etc/init.d"
  /* Runs: 'ls -l -a /etc/init.d' */
  execlp( "ls", cmd, "-l", "-a", dir, NULL );
}
```

Esecuzione di programmi

Famiglia di funzioni `exec*()`

- `int execle( const char* path, const char* arg , ..., char* const envp[]);`
  - Esegue il programma *`path`*
    - Il nome dell'eseguibile deve essere completo del path
  - Argomenti sulla linea di comando
    - Sono passati mediante una lista di stringhe, una per argomento
    - La lista deve essere terminata dal valore speciale **`NULL`**
  - Ambiente
    - È passato mediante la variabile *`envp`*

```
int main( int argc, char** argv, char** envp ) {
  char* cmd = "echo"
  /* Runs: 'echo $HOME' */
  execle( "/bin/echo", cmd, "$HOME", NULL, envp );
}
```

Esecuzione di programmi

Famiglia di funzioni `exec*()`

- `int execv( const char *path, char *const argv[]);`
  - Esegue il programma *`path`*
    - Il nome dell'eseguibile deve essere completo del path
  - Argomenti sulla linea di comando
    - Sono passati mediante un array di stringhe
    - È la forma con cui gli argomenti sono ricevuti da `main()`
    - Il primo elemento del vettore *`argv[0]`* è ignorato in quanto è il nome del programma eseguibile

```
int main( int argc, char** argv ) {
  /* Runs: 'ls' with the arguments received on the
  command line */
  execv( "/bin/ls", argv );
}
```

Esecuzione di programmi

Famiglia di funzioni `exec*()`

- `int execvp( const char *file, char *const argv[]);`
  - Esegue il programma *`file`*
    - Il nome dell'eseguibile è cercato dal sistema operativo in tutti i percorsi specificati dalla variabile di ambiente `PATH`
  - Argomenti sulla linea di comando
    - Sono passati mediante un array di stringhe
    - È la forma con cui gli argomenti sono ricevuti da `main()`
    - Il primo elemento del vettore *`argv[0]`* è ignorato in quanto è il nome del programma eseguibile

```
int main( int argc, char** argv ) {
  char* arguments[4] = { "aaa", "bbb", "ccc", NULL };
  /* Runs: 'echo aaa bbb ccc' */
  execvp( "echo", arguments );
}
```

Esecuzione di programmi

Famiglia di funzioni `exec*()`

- Le funzoni `exec*()`
  - Sostituiscono il codice del programma originario
    - Non è più possibile continuare l'esecuzione del programma originario
  - Se l'esecuzione ha successo
    - Tali funzioni non ritornano in quanto il codice originario non esiste più nella memoria relativa al processo
  - In caso di fallimento
    - Le funzioni ritornano e segnalano una situazione di errore
  - Non poter ritornare al codice originario è una limitazione
    - Si ricorre ad un processo ausiliario dedicato alla esecuzione del programma esterno
    - Il processo ausiliario è sostituito dal nuovo codice ma il processo originario è ancora in esecuzione

Esecuzione di programmi

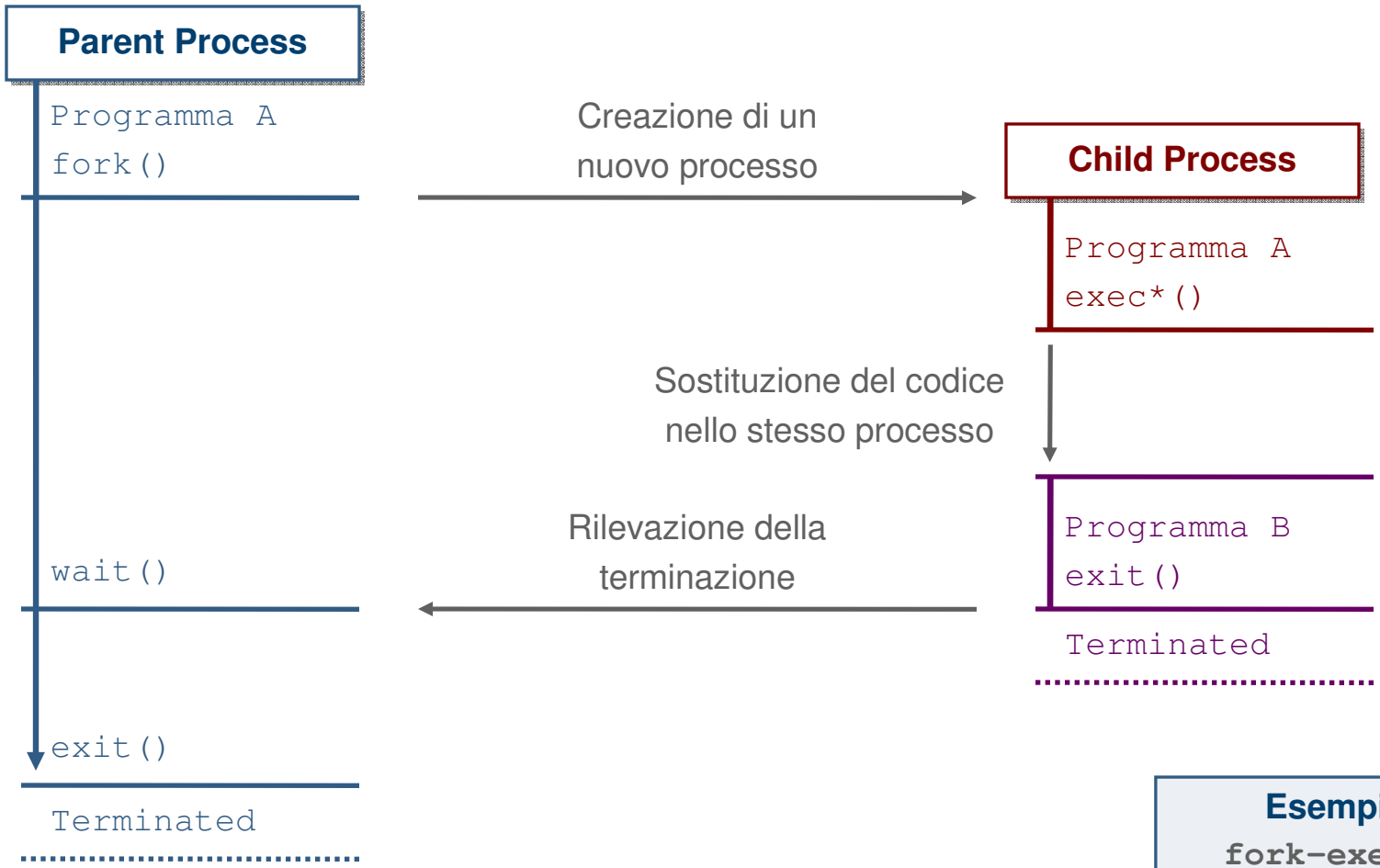
Esecuzione non distruttiva

- Lo schema tipico è il seguente
  - Il processo originario (parent) crea un processo ausiliario (child)
  - Il parent process si sospende in attesa del child process
  - Il child process
    - Sostituisce il proprio codice con quello del programma da eseguire
    - Il descrittore del processo rimane invariato per cui il parent process ha tutte le informazioni necessarie per gestire il child process
    - La terminazione del child process è intercettata dal parent process
    - Lo stato di uscita del child process è rilevabile dal parent process
  - Il parent process si sblocca e continua l'esecuzione
- Questo meccanismo permette di eseguire in modo non distruttivo qualsiasi programma esterno

Esecuzione di programmi

Esecuzione non distruttiva

- Programma A esegue Programma B



Esempio
`fork-exec.c`

Segnali

Descrizione

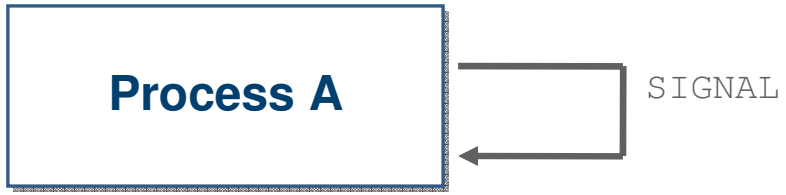
- Due processi possono sincronizzarsi mediante
  - La coppia di funzioni `exit()` e `wait()`
  - Scambio di segnali
- Il meccanismo dei segnali è più flessibile in quanto
  - Non implica la terminazione di uno dei processi
  - Non richiede che i processi siano legati da una 'parentela'
  - Si hanno a disposizione diversi segnali
  - Anche il sistema operativo stesso utilizza i segnali per comunicare con i processi
- Un segnale
  - È un interrupt software
  - È inviato da un processo verso un altro processo qualsiasi

Segnali

Descrizione

- Un segnale può essere diretto
  - Verso un altro processo

  - Verso il processo stesso che lo invia



Segnali

Descrizione

- Lo standard POSIX prevede diversi segnali standard
  - Ad ogni segnale è associato una azione di default
  - Per alcuni segnali tale azione può essere modificata
- Sono usati dal sistema operativo o da un processo per
  - Segnalare ai processi condizioni di errore
    - `SIGHUP, SIGILL, SIGFPE, ...`
  - Segnalare eventi specifici
    - `SIGALRM, SIGTTIN, SIGTTOU, ...`
  - Modificare lo stato di esecuzione dei processi
    - `SIGQUIT, SIGKILL, SIGSTOP, SIGCONT, ...`
  - Segnali generici
    - `SIGUSR1, SIGUSR2`

Segnali

Invio di segnali

- `int kill(pid_t pid, int sig);`
  - Invia un segnale generico ad un processo
    - `pid` il process id del processo cui inviare il segnale
    - `sig` il segnale da inviare

- `int exit( int status );`
  - Causa la terminazione di un processo
  - Invia al parent process il segnale `SIGCHLD`

- `unsigned int alarm( unsigned int seconds );`
  - Allo scadere del numero di secondi specificato, invia il segnale `SIGALRM` a se stesso

Segnali

Handler

- A molti segnali è associato un comportamento di default
  - Si dice che ad un segnale è associato un handler
    - Un handler è una funzione C
    - Gli handler di default sono funzioni che fanno parte del codice del sistema operativo e non del programma utente
  - Quando un processo riceve uno di tali segnali
    - Esegue il corrispondente handler

- Per alcuni segnali è possibile
  - Modificare il comportamento di default
    - Sostituendo l'handler di default con uno definito dal programmatore
  - Mascherarne la ricezione
    - Un processo non è sensibile alla ricezione del segnale

Segnali

Handler

- `int sigaction( int signum, const struct sigaction *act, struct sigaction *oldact );`
  - Installa una nuova azione per il segnale *signum*
  - *act* e *oldact* fanno riferimento alla struttura *sigaction*, tramite la quale si specificano tutte le caratteristiche dell'azione associata ad un segnale
    - *act* fa riferimento alla nuova azione
    - *oldact* fa riferimento all'azione corrente
- Struttura `sigaction`
  - Campo `sa_handler` per indicare la routine di gestione del segnale

```
struct sigaction{
  void (*sa_handler)(int);
  void (*sa_sigaction)(int, siginfo_t *, void *);
  sigset_t sa_mask;
  int sa_flags;
  void (*sa_restorer)(void);
}
```

Segnali

Handler

- Campo `sa_handler`
  - `SIG_DFL`: usa la disposizione di default
  - `SIG_IGN`: ignora il segnale (non con tutti)
  - puntatore a funzione: accetta un parametro numerico (numero del segnale) e ritorna `void`
- Meccanismo asincrono
  - Può portare il processo in uno stato non stabile
    - Mai chiamare primitive di I/O o funzioni di libreria in un handler
  - Interrompibile dall'arrivo di un altro segnale
    - sig_atomic_t per garantire che operazioni di assegnamento siano eseguite con un'unica istruzione

Esempio
`sigusr1.c`