

# **Architettura dei Calcolatori e Sistemi Operativi**

## **Processi**

---

*Chair*

**Prof. C. Brandoles**

e-mail: carlo.brandolese@polimi.it  
phone: +39 02 2399 3492  
web: home.dei.polimi.it/brandole

---

**Politecnico di Milano**

*Teaching Assistant*

**D. Iezzi**

e-mail: domenico.iezzi [at] polimi [dot] it  
material: [github.com/NoMore201/polimi\\_cr\\_acso\\_2019](https://github.com/NoMore201/polimi_cr_acso_2019)

# Outline

---

## ■ Processi

- Monitoraggio dei processi
- *fork*
- *wait*
- *exec*
- Segnali

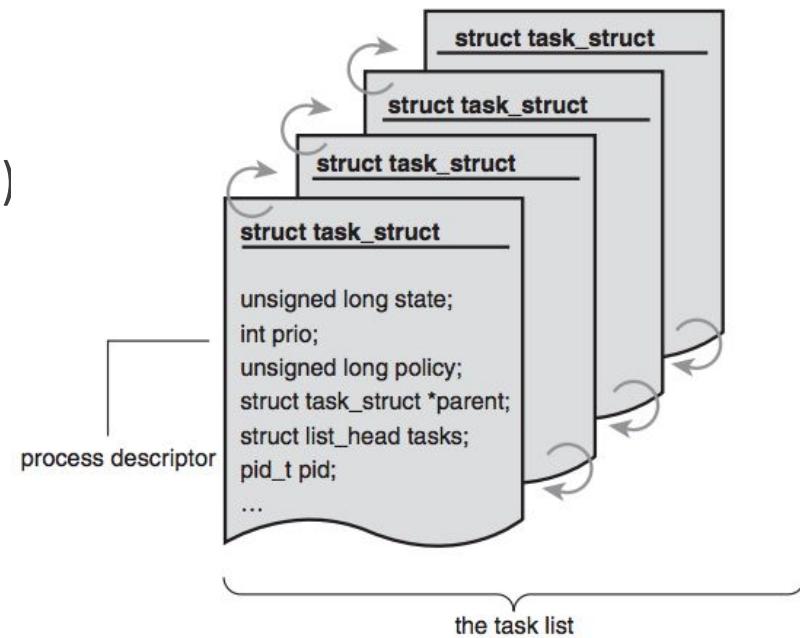
# Processi

## ■ Un processo

- Rappresenta un'istanza di un programma in esecuzione
- Può essere considerato un "esecutore virtuale"
- Permette di gestire l'accesso concorrente alle risorse

## ■ Il Process Descriptor contiene

- PID del processo padre
- Informazioni sull'utente (uid, gid, permessi)
- Informazioni sulla memoria
- Riferimenti alle tavole di sistema
- Informazioni sui tempi di esecuzione
- ...



# Monitoraggio dei processi

---

## Monitoraggio dei processi

- Nei sistemi Unix e GNU/Linux è possibile monitorare i processi attraverso il programma **ps**, che offre una serie di opzioni molto utili. Per conoscerne le opzioni, digitare **ps --help**
- Per monitorare l'evoluzione in tempo reale dei processi, utilizzare **top** o **htop**. Digitare il comando **t**, una volta lanciato **htop**, per visualizzare l'albero di gerarchia dei processi.
- Per cercare il PID di un processo specificando parte del nome o altri parametri si può utilizzare il comando **pgrep**
- È possibile terminare i processi attraverso il comando **kill**, passando come parametro il flag **-KILL** e il PID del processo da terminare

# Fork

---

## Creazione dei processi

- Un processo viene creato tramite la funzione *fork*
- Valgono le considerazioni su *fork* espresse a lezione, di seguito vedremo un esempio basilare

Esercizio

proc-1

Creazione di processo

- Il processo figlio eredita tutte le variabili del padre, ma in forma di “copie”: non è concesso infatti al figlio di modificare i dati del padre, ma solo la propria copia locale di tali dati
- Fanno eccezione a tale regola i descrittori dei file, di cui vedremo di seguito un esempio

Esercizio

proc-2

Processi figli e descrittori dei file

# Wait

---

## Utilizzo della *wait*

- La famiglia *wait* è una famiglia di system call che permettono ad un processo padre di “attendere” la terminazione di un processo figlio e catturare l’*exit status*
- Vi sono due varianti principali:
  - `pid_t wait(int* status);`
  - `pid_t waitpid(pid_t pid, int *status, int options);`

## Funzione *wait(int\* status)*

- La ***wait()*** è bloccante, attende la terminazione di un processo figlio qualunque e ritorna il pid del processo figlio che è terminato. Tale valore può essere:

<code>pid</code>	Il pid del processo figlio che è terminato e che la wait ha raccolto
<code>-1</code>	Segnala un errore nella terminazione e viene settata di conseguenza la variabile <code>errno</code>

`int* status` viene invece settato al valore di ritorno restituito dal figlio all’atto della sua terminazione

## Funzione *waitpid(pid\_t pid, int\* status, int options)*

- La ***waitpid()*** può essere bloccante o non bloccante e tale comportamento è controllato passando un opportuno valore per il terzo argomento, `int options`, specificando `WNOHANG` per ottenere un comportamento non bloccante e `WUNTRACED` che impone di ritornare anche nel caso in cui i figli siano rimasti bloccati, ignorandone di conseguenza l’*exit status*

# Wait

## Funzione *waitpid(pid\_t pid, int\* status, int options)*

- Il primo parametro è il pid del processo da attendere. Vi sono dei valori speciali per tale argomento che determinano comportamenti specifici delle waitpid():

< -1	Attende qualunque processo figlio avente process <b>group</b> id uguale al valore assoluto del pid
-1	Attende qualunque processo figlio, come la wait()
0	Attende qualunque processo figlio avente process <b>group</b> id uguale a quello corrente
> 0	Attende il processo avente il pid specificato

- Così come `wait()`, `waitpid()` restituisce il pid del processo che è terminato, con le stesse regole di `wait()` a cui se ne aggiunge una: restituisce 0 se WNOHANG è settato e nessun figlio è terminato

Esercizio

proc-3

Wait e waitpid

## Tipi speciali: **wait3** e **wait4**

- `wait3` svolge le medesime funzioni di `wait`, ritornando però anche le statistiche di utilizzo della CPU del processo figlio. Inoltre, il comportamento di tale funzione può essere bloccante o non bloccante a seconda delle opzioni passate
- `wait4` permette di specificare ulteriori opzioni circa i processi da attendere. Anche essa può essere sia bloccante che non bloccante

# Wait

## Controllare l'exit status

- Attraverso l'utilizzo di opportune macro, è possibile testare l'exit status registrato dalle wait:

WIFEXITED (status)	Vera se il figlio è terminato "naturalmente"
WIFSIGNALED (status)	Vera se il figlio è terminato a causa di un segnale, che non ha gestito
WTERMSIG (status)	Riporta il segnale che ha causato la terminazione del figlio
WEXITSTATUS (status)	Riporta i less-significant 8 bit dell'exit status ed ha senso solo se WIFEXITED(status) risulta verificata
WIFSTOPPED (status)	Ha valore solo se WUNTRACED è impostata nelle opzioni e risulta vera se il figlio risulta stopped
WSTOPSIG (status)	Ritorna il numero di segnale che ha causato lo stop del figlio

- Secondo quanto visto a lezione, l'utilizzo delle wait e la sequenza di terminazione di padre e figli influisce sullo stato degli stessi, che possono venire dunque a trovarsi in uno stato di Zombie.
- Tale stato si ha nel momento in cui il figlio termina ed il padre non ha ancora effettuato la wait o è terminato a sua volta senza chiamare tale funzione. A seconda dei casi il processo rimarrà zombie fino alla chiamata della wait, oppure, se il padre è terminato, rimarrà zombie finché non verrà adottato e terminato dal processo init.

## Utilizzo della exec

- Spesso all'utilizzo della `fork` è associato l'utilizzo della funzione `exec`, nelle sue varie versioni
- La famiglia di funzioni `exec` non fa altro che interrompere l'esecuzione del programma corrente e passare all'esecuzione del nuovo programma, il tutto all'interno del processo dalla quale è invocata
- Vi sono tre grandi famiglie di `exec`:
  - Quelle che contengono la lettera *p*: queste versioni di `exec` prendono come parametro il nome del programma da eseguire e lo ricercano all'interno della directory corrente. Alle versioni di `exec` senza la *p* occorre passare il path completo del programma da eseguire
  - Quelle che contengono la lettera *v*: queste versioni accettano i parametri da passare al programma in forma di vettori di puntatori a stringhe, terminate dal carattere nullo
  - Quelle che contengono la lettera *l*: a differenza delle versioni con la *v*, accettano il passaggio di parametri secondo il meccanismo *varargs* del C
  - Quelle che contengono la lettera *e*: a tali versioni vengono passate anche le variabili ambientali, sotto forma di vettore di puntatori a stringhe, terminato dal carattere nullo, in cui le stringhe sono nella forma (VARIABILE=valore)
- Le `exec` non ritornano, a meno di errori, in quanto il programma chiamante viene completamente rimpiazzato

# Segnali

---

## Utilizzo dei segnali

- I segnali, come visto a lezione, sono uno strumento estremamente semplice, ma al tempo stesso efficace per la comunicazione tra processi
- Ve ne sono vari tipi, definiti in `/usr/include/bits/signum.h`. Ne vedremo solo alcuni più importanti
- Un processo che riceve un segnale può gestirlo essenzialmente in tre modi:
  - **Default:** ogni segnale ha un handler di default che viene eseguito se nessun handler specifico viene definito e se il segnale non viene ignorato. Tale modalità può essere attivata tramite l'opzione `SIG_DFL` passata alla struttura `sigaction`, che vedremo tra breve.
  - **Ignorandolo:** in tal caso il segnale non viene gestito in alcun modo. Tale modalità può essere attivata tramite l'opzione `SIG_IGN` passata alla struttura `sigaction`.
  - **Gestito:** un segnale può essere gestito attraverso un opportuno handler, ovvero una funzione che specifica le azioni da intraprendere a fronte della ricezione del segnale. Non per tutti i segnali è possibile definire un handler.
- Il sistema operativo (nello specifico Linux) utilizza alcuni particolari segnali per comunicare con i processi, esempi importanti sono:
  - `SIGBUS`: per segnalare un bus error
  - `SIGSEGV`: per segnalare una segment violation
  - `SIGFPE` per le floating point exceptionIn tutti e tre i casi il processo viene terminato e viene generato un opportuno codice di errore.

# Segnali

---

- I processi trasmettono dei segnali tra di loro
  - SIGKILL o SIGTERM per terminare il processo
  - SIGUSR1 e SIGUSR2 per innescare specifiche azioni
  - SIGHUP per risvegliare un processo in idle o costringerlo a leggere nuovamente il file di configurazione
  - SIGALRM, lanciato dalla funzione `unsigned int alarm(unsigned int seconds)` per generare un evento di allarme dopo `tot` secondi
  - SIGCHLD segnale inviato al processo padre quando uno dei suoi figli termina

## Programmare con i segnali

- Per gestire i segnali vengono utilizzate le funzioni `signal()` e `sigaction()`
  - **signal:** `void (*signal(int sig, void (*func)(int)))(int)`
    - prende come parametro un intero e un puntatore a funzione
    - `signal` ritorna un puntatore a funzione `void (*)(int)`
  - **sigaction:** `int sigaction(int sig, const struct sigaction *act, struct sigaction *oact)`  
Il primo parametro specifica il numero del segnale da gestire, il secondo e il terzo sono puntatori a `struct sigaction` che specificano, rispettivamente, le azioni che andranno intraprese al ricevimento del segnale e le azioni disposte in precedenza.  
Il campo più importante delle `struct sigaction` è `sa_handler`, che può essere un puntatore a funzione allo handler definito dal programmatore (che a sua volta prende come parametro di ingresso il numero di segnale), o uno dei due flag `SIG_DFL` / `SIG_IGN`

# Segnali

## Programmare con i segnali

- Per gestire i segnali il programmatore può definire dei signal handler, ovvero delle funzioni che prendono in ingresso il numero del segnale e svolgono il numero di operazioni minimo e indispensabile utili a gestirlo.
- Esempio di signal handler:

```
typedef void (*sighandler_t)(int);

void signal_handler( int sig )
{
    if( sig == SIGUSR1 ) {
        printf( "Handling signal SIGUSR1.\n" );
        exit( 0 );
    } else {
        printf( "Handling signal SIGUSR2.\n" );
    }
}
```

- È possibile che un signal handler possa essere interrotto a sua volta dall'arrivo di un altro segnale (situazione molto difficile da “debuggare”) e per tale motivo è necessario che compia il minor numero di operazioni possibile

# Segnali

## Programmare con i segnali

- Persino l'operazione di assegnamento di una variabile globale è rischiosa in quanto comporta in alcuni casi più di una operazione assembly, poiché il flusso di esecuzione verrebbe interrotto all'arrivo del segnale
- A tale scopo è definito il tipo di variabile `sig_atomic_t` il cui assegnamento è atomico: variabili di questo tipo possono essere utilizzate come variabili globali per tenere traccia, ad esempio, del fatto che il segnale sia stato ricevuto.
  - `sig_atomic_t` è di tipo `int`, ma possono essere utili al caso anche altri tipi di dimensione minore o uguale all'`int`, in quanto per tali variabili l'istruzione di assegnamento è atomica.

Esercizio

Sign-1, Sign-2, Sign-3

SIGUSR1 e 2, SIGALRM, SIGCHLD

# Utilizzo di kill per inviare segnali da shell

---

## Inviare segnali tramite comando *kill*

- La sintassi del comando *kill* segue il seguente schema:

```
kill -SEGNALE pid_processo
```

- SEGNALE è il codice del segnale che vogliamo inviare, ad esempio:
  - KILL
  - SIGUSR1
  - SIGUSR2
  - etc.
- Il terzo argomento, *pid\_processo*, è il pid del processo a cui vogliamo inviare il segnale. Per recuperare il pid di un processo da shell, è possibile utilizzare uno dei seguenti comandi:

```
ps ax | grep nome_processo  
pgrep nome_processo
```

# Terminazione dei processi

---

- Come accennato in precedenza e come spiegato a lezione, un processo termina naturalmente a fronte di un'istruzione `exit()` o della return del `main`
- Un processo può anche terminare in maniera anomala a seguito della ricezione dei segnali:
  - `SIGBUS`, `SIGSEGV`, `SIGFPE` – illustrati in precedenza
  - `SIGINT`, segnale inviato quando l'utente digita il comando `Ctrl+C`
  - `SIGTERM`, tale segnale è inviato dal comando `kill` e come azione di default terminerà un processo.
  - inviando a se stesso un segnale `SIGABRT`, attraverso la funzione `abort`, un processo causa la terminazione di se stesso e la generazione di un *core file*
  - `SIGKILL`, in assoluto il più potente tra i segnali di terminazione, il quale impone la terminazione immediata di un processo che non potrà in alcun modo gestire il segnale
- Un segnale di tipo `SIGKILL` può essere inviato sia da shell, digitando il comando

```
KILL -KILL pid
```

o anche all'interno di un programma attraverso la funzione `kill(pid_t child, int signal)`, utilizzando `SIGTERM` per simulare il comportamento del comando `KILL`:

```
kill(child_pid, SIGTERM);
```