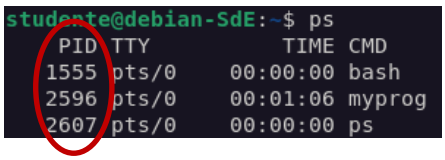
Processi

Creazione e sincronizzazione

Identificatore di processo

- Ogni processo ha un identificatore
  - **P**rocess **ID**entifier → **PID**
- Numero interno *non negativo*
- *Univoco* durante l'esecuzione
- Può essere riutilizzato quando il processo termina
  - Assegnato a un nuovo processo
- PID=1 → processo `init`



man ps

Ogni processo è identificato da un **PID**, che viene assegnato al momento della creazione del processo e rilasciato quando termina, in modo che possa essere utilizzato da un nuovo processo.

Il comando **ps** ci fa vedere i processi in esecuzione, e la prima colonna mostra proprio il PID.
ps riporta i processi mandati in esecuzione dall'utente sulla sessione di shell corrente. Per vedere tutti i processi attivi nel sistema usare il comando ps -e

Anche i processi di sistema (cioè quelli non eseguiti direttamente dall'utente, ma dal kernel) hanno un PID, per esempio il PID 0 è assegnato a un processo speciale del kernel che in alcuni sistemi è detto idle process, cioè il processo eseguito quando non c'è altro da fare. In altri sistemi, il processo 0 è lo shedulatore, che decide quale processo utente deve andare in esecuzione.

Ottenere l'identificatore di processo

```
#include <unistd.h>

pid_t getpid();
```

Restituisce il PID del processo (non può terminare con errore)

- Ogni processo (eccetto `init`) viene creato da un altro processo
  - Tutti i processi hanno un processo genitore → *parent process*

```
pid_t getppid();
```

Restituisce il PID del processo genitore

getpid() è una system call fornita dalla libreria unistd.h, restituisce il PID del processo (pid_t)
Può essere utile, per esempio, se dovete assegnare un nome univoco a qualcosa all'interno del programma, data la sua caratteristica di univocità. Per esempio se vogliamo assegnare un nome a un file generato dal programma e essere sicuri che sia un nome univoco.

Queste funzioni non hanno un valore di ritorno errato (per definizione, i processi hanno sempre un pid, quindi non è possibile terminare con errore)

Come detto, ogni processo è creato da un altro processo, chiamato **parent process**, genitore. È possibile ottenere il PID del genitore con getppid().

Se ogni processo ha un genitore, come ha fatto il genitore a creare il figlio?

Creazione di un processo

```
#include <unistd.h>

pid_t fork();
```

- Il processo crea un nuovo processo figlio
  - *Child process*
- Il processo genitore continua l'esecuzione
  - Esegue le istruzioni successive all'istruzione `fork`
- Il processo figlio va subito in esecuzione
  - Esegue le istruzioni successive all'istruzione `fork`
- Restituisce *due valori diversi* nel processo genitore e nel processo figlio
  - Nel processo genitore, restituisce il PID del processo figlio appena creato
  - Nel processo figlio, restituisce 0
- In caso di errore, non crea il processo figlio e restituisce `-1`

Il processo può invocare la system call **fork()**, generando un nuovo processo. Il processo che ha chiamato la funzione è il **processo genitore**, mentre il nuovo processo è il **processo figlio**.
Subito dopo l'invocazione, il processo genitore prosegue, mentre il figlio comincia a eseguire le sue istruzioni, quali? Il figlio esegue le stesse istruzioni del genitore, quelle che seguono l'istruzione fork.

La fork restituisce un numero, ovvero un PID. Sia il genitore che il figlio possono vedere il valore restituito dalla fork → qui viene la distinzione: **il valore restituito dalla fork è diverso nel genitore e nel figlio**.
Nel genitore, la fork restituisce il PID del processo appena creato, mentre nel figlio restituisce 0. Quindi, è possibile testare il valore restituito dalla fork e eseguire istruzioni diverse nei due casi, per differenziare il comportamento del genitore e del figlio.
Nel genitore, ho il PID del figlio perché posso eseguire la fork più volte e creare più di in figlio, quindi avere il suo PID mi serve a distinguerlo dagli altri. Nel figlio ho 0, tanto un processo figlio può avere solo un genitore, e può sempre recuperare il suo PID con la funzione getppid() vista prima.

In caso di errore nella fork(), il processo figlio non viene creato e la funzione restituisce -1.

Creazione di un processo

```
#include <unistd.h>
...
int main(int argc, char* argv[]) {
    pid_t pid;
    if ((pid = fork()) < 0) {
        fprintf(stderr, "Error while forking. Exit\n");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0) {
        printf("This is the child process!\n");
    }
    else {
        printf("This is the parent process!\n");
    }
    return 0;
}
```

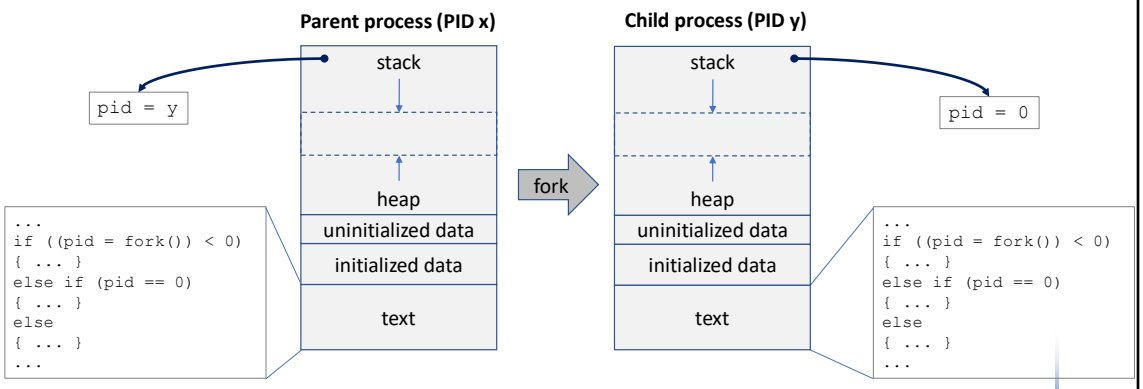
Qui, processo genitore e figlio hanno due valori diversi nella variabile `pid`

Non possiamo sapere quale stringa verrà visualizzata per prima!!

Eseguendo il codice, chi esegue per primo? Probabilmente si vede prima il messaggio del genitore perché il figlio deve essere creato e c'è un tempo non trascurabile che deve passare.

Creazione di un processo

- La memoria del processo figlio è una copia esatta della memoria del processo genitore
- I due processi operano sulle proprie copie in maniera indipendente



Quando si fa la fork, in memoria principale viene creata una copia **esatta** della memoria del processo genitore, con una sola differenza: **il valore della variabile pid**, che è diverso nel processo genitore e nel processo figlio (vedi slide precedente).
Da lì in avanti, i due processi hanno vita indipendente, solo il codice (segmento testo) è uguale. Le variabili hanno stesso nome ma sono oggetti indipendenti.

Creazione di un processo

```
#include <unistd.h>
...
int main(int argc, char* argv[]) {
    int var = 100;
    pid_t pid;
    if ((pid = fork()) < 0) {
        fprintf(stderr, "Error while forking. Exit\n");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0)
        var += 20;
    else
        var += 10;
    printf("PID = %d, var = %d\n", (int)getpid(), var);
    return 0;
}
```

Nel processo genitore, `var = 110`
Nel processo figlio, `var = 120`

Si usa `getpid()` per ottenere il PID del processo in esecuzione, non il contenuto della variabile `pid`

L'ultima istruzione printf() viene eseguita sia dal genitore che dal figlio, ma i valori visualizzati a video del PID e della variabile var sono diversi nei due casi.

Sincronizzazione

- Il processo genitore può voler eseguire delle istruzioni **dopo** che il processo figlio ha terminato la sua esecuzione
- Esempio: si vuole che il processo genitore stampi a video dopo il processo figlio
- Soluzione?

```
...
if ((pid = fork()) == 0) {
    printf("This is the child process!\n");
}
else if (pid > 0) {
    sleep(3);
    printf("This is the parent process!\n");
}
...
```


Non possiamo sapere **a priori** se viene eseguito prima il genitore o il figlio. Come facciamo a imporre un qualche ordinamento?

Usare la system call sleep() può essere una soluzione. La sleep mette in pausa il processo per il numero di secondi specificato come argomento.
Problema:

- Quanti secondi devo indicare? Nell'esempio, chi mi garantisce che il figlio esegua in meno di tre secondi?
- Se il figlio esegue in meno di 3 secondi, il genitore resta in attesa per più tempo del necessario

Sincronizzazione

```
#include <sys/wait.h>

pid_t wait(int* status);
```

- L'esecuzione del processo *si blocca* in attesa che il processo figlio termini
- Quando il processo figlio termina, l'esecuzione del processo riprende
  - La funzione restituisce il PID del figlio terminato
  - La variabile `status` è un puntatore a un intero che rappresenta il codice di terminazione del figlio

- Se il processo figlio era già terminato, il processo non si blocca
- Se il processo non aveva figli, la funzione restituisce errore (`-1`)
- Se il processo ha più di un figlio, l'esecuzione riprende quando uno qualsiasi dei figli termina (o è già terminato)

Il processo genitore che invoca questa system call rimane in attesa che il processo figlio termini.
Quando il processo figlio termina, il processo genitore si sblocca e la funzione restituisce il PID del processo terminato. Inoltre, la variabile **status** contiene un codice di stato che indica se il figlio è terminato correttamente o con errore. Questo sarebbe il valore usato come argomento dalla exit nel processo figlio. Se il processo genitore non è interessato a sapere il codice di stato, è possibile passare il valore NULL a questa funzione.

Se il processo genitore avesse creato due o più processi figlio, la wait si sblocca quando ne termina uno qualsiasi. Per questo è utile che venga restituito il suo PID, così il processo genitore sa quale figlio è terminato.

Sincronizzazione

```
#include <unistd.h>
#include <sys/wait.h>
...
int main(int argc, char* argv[]) {
    pid_t pid;
    if ((pid = fork() < 0) {
        fprintf(stderr, "Error while forking. Exit\n");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0) {
        printf("This is the child process!\n");
    }
    else {
        if (wait(NULL) < 0) {
            fprintf(stderr, "Error in wait. Exit\n");
            exit(EXIT_FAILURE);
        }
        printf("This is the parent process!\n");
    }
    return 0;
}
```


In questo esempio il processo figlio (posto che la fork vada a buon fine) stampa a video il suo messaggio **sempre** prima del processo genitore.
La wait può fallire quando, per esempio, era fallita la fork in precedenza e nessun processo figlio era stato creato.

Sincronizzazione

```
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int* status, int options);
```

- La variante `waitpid` permette di specificare su quale processo figlio mettersi in attesa
  - Argomento `pid`
  - Se `pid == -1`, equivalente a `wait`
- Le opzioni forniscono funzionalità avanzate rispetto a `wait`
  - e.g. versione *non bloccante*, serve per testare lo stato del processo figlio (consultare `man`)
- Tutti i processi figli terminati e su cui non è ancora stata effettuata `wait`/`waitpid` sono detti ***zombie***

Zombie: il processo è terminato, ma ancora non è stato eliminato dalla memoria, perché è in attesa che il genitore faccia una wait e il genitore potrebbe voler ottenere, per esempio, il suo exit status

La famiglia `exec`

- Insieme di **sette** funzioni di libreria
  - `execl`, `execv`, `execle`, `execve`, `execlp`, `execvp`, `execvpe`
  - Le chiameremo genericamente `exec`
- Permettono di **sostituire** il processo in esecuzione con un nuovo programma
- Il nuovo programma inizia ad eseguire a partire dal proprio *entry point*
- Il PID non cambia
  - Non si crea alcun nuovo processo
  - La memoria del processo (segmento testo, dati, stack, heap) viene sostituita dal nuovo programma
- Uso comune
  - Creazione di un nuovo processo figlio tramite `fork`
  - Il processo figlio invoca una delle funzioni `exec` per eseguire il nuovo codice

Mentre con la fork possiamo creare un nuovo processo, ma il suo codice è lo stesso del genitore, la exec ci permette di **rimpiazzare il codice** del programma chiamante.

Attenzione: la exec **non** crea un nuovo processo. Inoltre, tutte le istruzioni che seguono la exec non verranno **mai** eseguite (e meno che l'esecuzione della exec fallisca), perché il codice è stato rimpiazzato.
Per cui, l'utilizzo comune di exec è il seguente:

1) Creare un nuovo processo con la fork
2) Dentro al ramo del codice eseguito dal figlio invocare la exec per iniziare un nuovo programma

Da ricordare: la exec è la funzione che viene invocata dal kernel quando si esegue un programma dalla shell, quella che esegue il programma a partire dal suo entry point.

execl/execv

```
#include <unistd.h>

int execl(const char* pathname, const char* arg0, const char* arg1, ..., (char*)0);

int execv(const char* pathname, const char* argv[]);
```

Lista degli argomenti

Array contenente gli argomenti

```
char* argv[] = {
      "arg0",
      "arg1",
      ...
      (char*)0
};
```

Percorso del programma da eseguire

Restituisce -1 in caso di errore

man 3 exec

Il file eseguibile del programma si deve trovare esattamente nel percorso specficato dal primo argomento.

Il primo argomento, arg0, per convenzione è una ripetizione del nome del programma, alla pari di ciò che accade con gli argomenti della funzione main, argv[0] è il nome del programma.
Il numero di argomenti è flessibile, l'unica cosa che dobbiamo ricordarci è che dopo la lista degli argomenti deve essere uno 0, convertito a char* come richiede la funzione.

execle/execve

```
#include <unistd.h>

int execle(const char* pathname, const char* arg0,..., (char*)0, const char* envp[]);

int execve(const char* pathname, const char* argv[], const char* envp[]);
```

Array contenente le variabili di ambiente

```
char* envp[] = {
     "USER=studente",
     "HOME=/home/studente",
     ...
     (char*)0
};
```

man 3 exec

Le versioni di exec con la ‘e’ finale permettono di specificare esplicitamente le variabile di ambiente

execlp/execvp

```
#include <unistd.h>

int execlp(const char* pathname, const char* arg0, const char* arg1, ..., (char*)0);

int execvp(const char* pathname, const char* argv[]);
```

Se la stringa non è un percorso relativo o assoluto, cerca il programma all'interno delle directory specificate dalla variabile di ambiente `$PATH`

Le versioni con la 'p' vanno a cercare il file eseguibile specificato come primo argomento anche nei percorsi specificati dalla variabile d'ambiente PATH, oltre che nella directory corrente.

La famiglia `exec`: esempio

```
#include <unistd.h>
#include <sys/wait.h>
...
int main(int argc, char* argv[]) {
    int status;
    pid_t pid;
    if ((pid = fork() < 0) {
        /* errore */
    }
    else if (pid == 0) {
        if (execl("./somma", "./somma", "12", "8", (char*)0 ) < 0) {
            /* errore */
        }
    }
    else {
        if (wait(&status) < 0) {
            /* errore */
        }
        printf("Programma somma terminato con stato %d!\n", status);
    }
    return 0;
}
```

Il processo figlio viene sostituito con il programma di nome "./somma"

Per convenzione, il primo argomento del programma è il nome del programma stesso

"12" e "8" sono gli altri argomenti del programma

L'argomento `(char*)0` segnala la fine della lista di parametri

Esempio: implementazione di una shell (1/2)

```
...
int main() {
    const int MAXLEN = 1024;
    char cmd[MAXLEN];
    pid_t pid;

    // mostra ciclicamente un prompt per leggere comandi da tastiera
    printf("> ");
    while (fgets(cmd, MAXLEN, stdin) != NULL) {
        cmd[strlen(cmd) - 1] = '\0';    // sostituisce '\n' con '\0'

        // crea processo figlio
        if( (pid = fork()) < 0) {
            /* errore */
            fprintf(stderr, "fork non riuscita: %s\n", strerror(errno));
            printf("> ");
            continue;
        }
        ...
```

Esempio: implementazione di una shell (2/2)

```
        ...
        else if (pid == 0) {
            /* child process */
            // esegue il comando letto da stdin
            execlp(cmd, cmd, (char*)0);
            fprintf(stderr, "exec non riuscita: %s\n", strerror(errno));
            exit(EXIT_FAILURE);
        }
        else {
            /* parent process */
            // attende la terminazione del processo figlio
            if ((pid=waitpid(pid, NULL, 0)) < 0)
                fprintf(stderr, "waitpid non riuscita: %s\n", strerror(errno));
            printf("> ");
        }
    }
    return 0;
}
```