Processi e IPC

Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma
Luca Iezzi Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Reti di Calcolatori A.A. 2022/23

Prof.ssa Chiara Petrioli Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma
Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma

1. Processi

Esecuzione single-process

L'esecuzione dei programmi visti finora ha considerato solo la modalità single-process:

- Unico processo dall'avvio al termine del programma
- Alcuni blocchi di codice potrebbero essere indipendenti da altri: uno per volta e secondo l'ordine stabilito

Processo Unix

Un processo Unix è un'entità che esegue un dato blocco di codice, mantenendo il proprio stack di esecuzione, insieme di pagine di memoria, la tabella dei file descriptor, ed ha un identificatore di processo (process ID – PID).

Un processo quindi non è equivalente ad un programma: diversi processi possono derivare dall'esecuzione dello stesso programma. Quindi c'è la possibilità che una stessa funzione (o blocco di codice) stia venendo eseguita da più processi e si trovi in diversi stadi contemporaneamente: questo fenomeno si chiama rientranza.

Creazione di un nuovo processo

Librerie: `unistd.h`, `sys/wait.h`

- `pid_t fork(void)`
  Divide il processo corrente in 2: parent e child, entrambi con le stesse pagine di memoria (tabella file descriptor inclusa!); l'esecuzione del child comincia dopo la chiamata di `fork`.
  Il valore di ritorno è duplice:
  - Nel parent process: PID del child process oppure -1 (errori)
  - Nel child process: `0`
- `pid_t wait(int *stat_loc)`
  Attende il termine di un child process, salvandone lo status di uscita nella variabile fornita come argomento.
  Il valore di ritorno è il PID del child process e `stat_loc` contiene lo status dello stesso processo; diverse macro sono definite per interpretare lo status di un processo.

```c
int main(){
  pid_t child_pid = 0;
  int child_status = -1;

  child_pid = fork();
  switch(child_pid){
    case -1:{
      perror("fork() failed");
      return 1;
    }

    case 0:{
      printf("Hello from child process\n");
      return 0;
    }

    default:{
      wait(&child_status);
    }
  }

  printf("Hello from parent process\n");
  return 0;
}
```

Child

Hello world con multi-processing

Termine di un child process

La relazione parent-child tra processi Unix dà luogo ad alcune particolarità:

- Termine di un child process → segnale Unix inviato al parent per la gestione del termine di un child, finché non viene accuratamente gestito dal parent e quindi eliminato, il child process resta in uno stato zombie
- Termine di un qualunque processo → in caso avesse avuto uno o più child process, questi diventano orfani: in Unix, ogni processo orfano viene ereditato come child process del processo `init` (che li gestirà opportunamente)

Se un programma che genera un parent process non tiene conto del primo caso, ogni eventuale child process, raggiunto il suo termine, rimarrà nello stato zombie.

Termine di un child process

`wait` è una syscall adeguata per gestire correttamente la termine di un child process ed evitare che rimanga nello stato zombie.

- Il parent process non prosegue nella sua esecuzione finché il termine di uno dei suoi child process termina (chiamata bloccante)
- Non è molto conveniente usare il multi-processing in questo modo

Gestione asincrona del termine di un child process

Al termine di un child process, il segnale Unix `SIGCHLD` viene inviato al relativo parent process: installando su questo un signal handler che ne tenga conto, si può catturare tale segnale e quindi invocare `wait`.

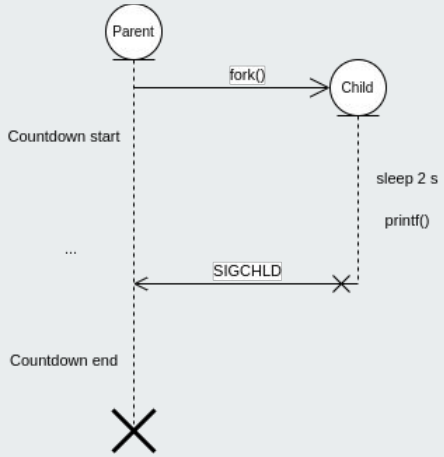
- Il child process è già nello stato zombie
- Il parent process ha proceduto nella sua esecuzione parallelamente
- L'invocazione di `wait` è garantita non essere bloccante
- Il child process viene correttamente eliminato

Signal handler per `SIGCHLD`

Librerie: `unistd.h`, `sys/wait.h`, `signal.h`

```c
int main(){
  // Installa il signal handler per SIGCHLD
  signal(SIGCHLD, handle_sigchld);

  pid_t child_pid = fork();
  switch(child_pid){
    case -1:{
      perror("fork() failed");
      return 1;
    }

    case 0:{
      sleep(2);
      printf("Engine at capacity, ready to go\n");
      return 0;
    }

    default: break;
  }

  printf("Countdown started\n");
  for(int i=10; i>0; i--){
    printf("%d...\n", i);
    sleep(1);
  }
  printf("Parent process exited\n");
  return 0;
}
```

```c
void handle_sigchld(int signal_no){
  int child_status = -1;
  wait(&child_status);
  printf("Child process exited\n");
}
```



Output

```
Countdown started
10...
9...
8...
Engine at capacity, ready to go
Child process exited
7...
6...
5...
4...
3...
2...
1...
Parent process exited
```

Attendere un determinato child process

- `pid_t waitpid(pid_t pid, int *stat_loc, int options)`
  Diversamente da `wait`, il parent process attende il termine di uno specifico child process fornito tramite PID; valore di ritorno e `stat_loc` è equivalente a `wait`.

Durante l'esecuzione di un child process, il parent può aver bisogno di controllare il suo status senza doverne attendere il termine oppure gestirne pausa e ripresa dell'esecuzione.

Argomento `options` di `waitpid`; i seguenti valori possono essere usati in bitwise OR:

- `0`
  Nessuna opzione – analogo ad invocare `wait` su un processo specifico
- `WCONTINUED`
  Ottiene lo status di un child process che ha ripreso l'esecuzione dopo una pausa, e di cui non si sia ancora richiesto lo status
- `WNOHANG`
  Ottiene lo status di un child process in modo non bloccante
- `WUNTRACED`
  Ottiene lo status di un child process che è stato messo in pausa, e di cui non si sia ancora richiesto lo status

Eseguire un programma interamente diverso in un child process

Creare un nuovo processo che esegua lo stesso programma del parent process può essere utile, ma restringe le possibilità di parallelismo a quanto incluso nel programma originale stesso.
Si può sostituire interamente il programma di un nuovo processo tramite le syscall della famiglia `exec`:

- Accetta qualunque eseguibile come argomento – raggiungibile dalla working directory oppure disponibile sul PATH
- Le syscall `exec` vanno interpretate come invocazioni di una funzione `main` di un programma C:
  - Tipo di ritorno `int` → Exit status del programma definito dall'eseguibile
  - Dopo l'argomento che rappresenta l'eseguibile, `int argc` e `char *argv[]`:
    - `argc` → Numero di argomenti da inviare all'eseguibile
    - `argv` → Puntatore ad array degli argomenti per l'eseguibile
- È possibile agire sulle variabili di ambiente visibili all'eseguibile:
  - Il parent process dichiara la variabile `extern char **environ`
  - Il parent process interagisce con le variabili di ambiente tramite `getenv`, `putenv`, `setenv`, `unsetenv`

```c
int main(){
  // Installa il signal handler per SIGCHLD
  signal(SIGCHLD, handle_sigchld);

  pid_t child_pid = fork();
  switch(child_pid){
    case -1:{
      perror("fork() failed");
      return 1;
    }

    case 0:{
      sleep(2);
      char *args_exec[] = {"echo", "Child process here!", NULL};
      execvp(args_exec[0], args_exec);
      return 0;
    }

    default: break;
  }

  printf("Countdown started\n");
  for(int i=10; i>0; i--){
    printf("%d...\n", i);
    sleep(1);
  }
  printf("Parent process exited\n");
  return 0;
}
```

2. Inter-process communication (IPC)

Comunicazione con child process

Dopo l'invocazione di `fork` non è più possibile far comunicare i processi parent e child: ogni modifica a variabili precedentemente impostate rimane locale al processo che la effettua.

Tuttavia, in un contesto multi-processing può essere di estrema utilità:

- Un parent process che ordina l'interruzione arbitraria di un child process (o altri eventi semplici)
- Un parent process che invia nuovi dati al child process
- Un child process che restituisce dati al parent process
- …

Semplici eventi: segnali Unix

Abbiamo visto come il modo per rendere una `wait` asincrona è quello di usare i segnali Unix.

Un segnale Unix in generale serve per indicare ad un processo l'avvenimento di un evento:

- `int sigaction(int sig, const struct sigaction *restrict act, struct sigaction *restrict oact)`
  Gestisce l'arrivo di un segnale (`sig`) tramite l'handler specificato nella struct `act`; durante l'esecuzione di un handler per un dato segnale, lo stesso segnale è bloccato, analogamente a tutti gli altri specificati con la maschera contenuta in `act`. `oact` può essere `NULL`, ma in caso non lo fosse, è una struct che viene usata per salvare la precedente politica di signal handling in uso.
- Il processo non sa esattamente quando il segnale viene inviato
- Durante il blocco di un dato segnale, se questo viene generato più di una volta, tale segnale verrà inviato esattamente una volta: i segnali Unix non sono gestiti tramite coda bensì come flag ON/OFF
- Non esiste un ordine preciso nell'invio dei segnali; in genere però i segnali relativi allo stato corrente del processo (ad es. `SIGSEGV`) sono inviati prima di altri

Segnali Unix: handler

Nella definizione di un signal handler è possibile passare uno dei seguenti valori (sotto forma di macro) al posto del puntatore a funzione che implementa un handler:

- `SIG_IGN` → ignora il segnale
- `SIG_DFL` → applica l'azione di default (quasi sempre *kill* del processo)

Non è possibile installare un handler per tutti i segnali:

- `SIGSTOP`
- `SIGKILL`

questi segnali infatti non possono essere intercettati, senza obbligo di avviso da parte del compilatore.

Segnali Unix: handler

Tra le flag disponibili nella `struct sigaction act`, alcune interessanti sono:

- `SA_RESTART`
  Syscall interrotte da un segnale vengono re-inizializzate automaticamente
- `SA_NODEFER`
  Durante l'esecuzione di un dato signal handler, lo stesso segnale non viene automaticamente bloccato
- `SA_SIGINFO`
  Abilita informazioni aggiuntive ad un signal handler → `struct siginfo`

Segnali Unix: inviare segnali

- `int kill(pid_t pid, int sig)`
  Invia un segnale `sig` ad un processo identificato tramite `pid`
  In realtà, è possibile inviare segnali a più di un processo:
  - `pid > 0` → Processo specifico con il dato PID
  - `pid == 0` → Tutti i processi dello stesso gruppo del processo chiamante
  - `pid == -1` → Tutti i processi eccetto il processo `init`
  - `pid < -1` → Tutti i processi nel gruppo `-pid` (valore assoluto di `pid`)

  Se `sig` ha valore `0`, nessun segnale viene inviato ma il controllo errori viene effettuato.

- `int raise(int sig)`
  Invia un segnale allo stesso processo chiamante della funzione.

Importante: un processo può inviare segnali ad un altro solo se ha privilegi di root oppure real / effective user ID è lo stesso del real / saved user ID del ricevente.

Comunicazione più complessa

I segnali Unix possono indicare l'avvenimento di eventi in contesti multi-processing, inclusi eventi definiti dall'utente. Tuttavia, implementare uno scambio di dati più complessi tramite essi è impossibile.

In C e con le API POSIX, comunicazioni di questo tipo sono supportate da diversi approcci:

- Pipe anonime e named pipes
- Socket di dominio Unix
- Message queues
- *Memoria condivisa*

IPC tramite pipe

Il meccanismo di IPC più elementare è rappresentato dalla serializzazione dello standard output di un processo e re-indirizzarlo come standard input di un altro.

```
ls -l | grep .pdf
```

Questo è chiamato anonymous pipe:

- Half-duplex, ovvero unidirezionale → per ottenere una comunicazione full-duplex c'è bisogno di due pipe
- I processi coinvolti devono definire un protocollo per implementare una comunicazione
- Garanzia di sistema: ordine dei dati ricevuti coerente con quello di invio
- Garanzia di sistema: comunicazione affidabile senza perdita dati (a meno che uno dei processi non termini prematuramente)

Utilizzo di anonymous pipes

- `int pipe(int fildes[2])`

  Creazione di una anonymous pipe per lettura e scrittura; l'argomento `fildes` è un array di due file descriptor, rispettivamente per la lettura e la scrittura → non è specificato se le modalità sono extended!
  - Una pipe creata prima di `fork` verrà ereditata dal child process (per via della tabella dei file descriptor)
  - Piena flessibilità di protocolli di comunicazione tra processi – utilizzo diretto di `read` e `write` POSIX
  - In Unix una pipe è considerata un file speciale; è possibile associare gli estremi della pipe con `stdin` e/o `stdout`
  - Possono essere usate tra processi che hanno un "antenato" in comune
  - Essendo half-duplex, è necessario che i processi chiudano l'estremità della pipe sulla quale non opereranno:
    i. A crea pipe – `fds[2]`, di cui `fds[0]` in lettura e `fds[1]` in scrittura
    ii. A crea B tramite `fork`
    iii. Direzione della comunicazione A → B:
    + A (parent) chiude `fds[0]` – A non legge dalla pipe, scrive solamente
    + B (child) chiude `fds[1]` – B non scrive sulla pipe, legge solamente

Importante: leggere (`read`) da una pipe la cui estremità di scrittura è chiusa restituisce `0`, mentre scrivere su una pipe la cui estremità di lettura è chiusa invia il segnale `SIGPIPE` al processo che scrive.

Utilizzo di anonymous pipes


```c
int fds[2];
int outcome = pipe(fds);
if(outcome == -1){
  perror("pipe() failed");
  return 1;
}

pid_t child_pid = fork();
switch(child_pid){
  case -1:{
    perror("fork() failed");
    return 1;
  }

  case 0:{
    child_process_routine(fds);
    return 0;
  }

  default:{
    parent_process_routine(fds);
    break;
  }
}

sleep(2);
printf("Parent process exited\n");
```

```c
void child_process_routine(int fildes[2]){
  int bytes, count;
  close(fildes[1]);

  while((bytes = read(fildes[0], &count, 1)) > 0){
    printf("Child: %d received\n", count);
  }
}
```

Il child process legge dalla pipe finché non ci sono più byte disponibili (`fildes[1]` chiusa dal parent).

Il parent process scrive sulla pipe finché non ha terminato la propria esecuzione; tuttavia, controlla che la scrittura sia di successo, altrimenti è possibile che il child process sia terminato prematuramente.

```c
void parent_process_routine(int fildes[2]){
  int bytes;
  close(fildes[0]);

  printf("Countdown started\n");
  for(int i=5; i>0; i--){
    printf("%d...\n", i);

    bytes = write(fildes[1], &i, 1);
    if(bytes == -1){
      perror("write() failed");
      close(fildes[1]);
      return;
    }

    sleep(1);
  }

  close(fildes[1]);
}
```

Insidie nell'utilizzo delle pipe

Per supportare una comunicazione full-duplex tra due processi A e B è necessario creare due pipe distinte: una da A verso B e l'altra da B verso A.

Considerando che `read` su una pipe vuota sia bloccante:

1. Entrambe le pipe sono vuote e sia A che B sono in `read` sulle rispettive estremità di input
2. Entrambe le pipe hanno un buffer di dimensione limitata associato; ogni `write` scrive nel buffer su una pipe, il cui contenuto viene mantenuto finché non avviene una `read` sulla pipe; se il buffer è pieno, `write` diventa bloccante finché non c'è dello spazio libero; sia A che B stanno scrivendo sulle rispettive estremità di output e raggiungono la dimensione del buffer

Entrambe queste situazioni portano ad una condizione di deadlock → entrambi i processi attendono la stessa risorsa, aspettando che l'altro la liberi.

Una volta raggiunto il deadlock, non è possibile uscirne se non tramite la terminazione forzata dei processi.

Anonymous pipe per programmi terzi

Le anonymous pipes viste finora possono essere utilizzate solo da processi generati tramite `fork` e che abbiano accesso diretto agli stessi file descriptor del parent.

Tuttavia, le anonymous pipe possono essere usate anche con processi che eseguano un programma terzo:

- `FILE* popen(const char *command, const char *mode)`
  Equivale ad aprire una anonymous pipe e dopodiché invocare `fork` ed `execl` per eseguire `command` in una shell `sh` e con l'opzione `-c` attiva (espande caratteri speciali ed environment variables). Il valore di `mode` determina come il processo invocante ed il processo con programma terzo siano collegati:
  - "`r`" → il file stream si riferisce allo `stdout` di `command`
  - "`w`" → il file stream si riferisce allo `stdin` di `command`
  - Altri valori → non specificato
- `int pclose(FILE *stream)`
  Chiude un file stream restituito da `popen` ed attende il termine del processo che ha eseguito `command`

Entrambe queste funzioni sono disponibili nella libreria standard I/O di Unix (`stdlib.h`), sebbene non siano effettivamente parte della libreria standard C. Letture e scritture su queste pipe avvengono tramite `stdio.h`.

Named pipes

popen comunque non supera la limitazione delle anonymous pipes che restringe la comunicazione solo a processi che abbiano un "antenato" in comune.

Named pipes (Libreria: `sys/stat.h`)

In Unix, le named pipes sono dei file speciali denominati FIFO e la trasmissione di dati avviene a tutti gli effetti come per i file. Aprendo una FIFO in lettura permette ad un processo di connettersi all'estremità di output della pipe, mentre aprendola in scrittura è possibile accedere all'estremità di input.

- `int mkfifo(const char *path, mode_t mode)`
  Crea una FIFO nel percorso `path`, mentre i valori accettati da `mode` sono uguali a quelli per `open` (ad es. `O_WRONLY`); restituisce `0` se l'operazione va a buon fine, altrimenti `-1` e viene impostato `errno`
  Dopo la creazione → POSIX I/O tramite `open`, `close`, `read`, `write`, `unlink` (eliminazione file)

Importante: una named pipe non può essere aperta sia per lettura che scrittura (ad es. `O_RDWR` non ammesso), e le operazioni `read` / `write` sono bloccanti per default: se la FIFO è vuota, una `read` non ottiene `EOF` ed il processo rimane in attesa; una `write` è bloccante finché non c'è alcun processo in lettura sulla stessa FIFO.

Utilizzi di named pipes

Superando la limitazione di un antenato in comune, è possibile implementare il trasferimento dati da una pipeline ad un'altra senza l'uso di file temporanei.

Ad esempio:

- Output di un processo che serve l'input di altri due processi
- Applicazione client-server in cui tutti i client comunicano con il server tramite un'unica FIFO
  La comunicazione server → client dovrà però basarsi su una pipe apposita per ogni client

IPC tramite socket

- Anonymous pipes possono essere usate solo all'interno dello stesso albero di processi
- Named pipes risolvono la limitazione dell'ereditarietà ma sono comunque unidirezionali
- Entrambi si basano sulla definizione di protocolli interni ai processi per ottenere una comunicazione efficace

Perchè usare socket per IPC

- Interfaccia immediata e di facile integrazione con la libreria standard I/O
- Full-duplex: tramite un solo socket è possibile garantire la comunicazione bidirezionale tra processi
- Le operazioni principali esistono sia in modalità bloccante che non → più flessibilità
- È possibile distinguere facilmente tra diversi messaggi (ad es. UDP socket)
- Socket di dominio Unix sono riconosciuti come tali dal kernel → più veloci rispetto a socket di dominio Internet
- Prestazioni paragonabili a named pipes
  Leggermente inferiori a pipes, ma distinguibili solo tramite benchmark sintetici
- Possibilità di evitare deadlock tramite l'imposizione di timeout sulle operazioni
  Funzione `setsockopt` con flag `SO_RCVTIMEO`

Utilizzo di socket per IPC

Partendo dallo stesso esempio visto con le pipes:

```c
int main(){
  signal(SIGCHLD, handle_sigchld);

  pid_t parent_pid = getpid();
  pid_t child_pid = fork();
  switch(child_pid){
    case -1:{
      perror("fork() failed");
      return 1;
    }

    case 0:{
      child_process_routine(parent_pid);
      return 0;
    }

    default:{
      sleep(1);
      parent_process_routine(child_pid);
      break;
    }
  }

  sleep(1);
  unlink(ADDRESS);
  return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>

#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <string.h>

#define BUFSIZE 256
#define ADDRESS "socket_unix_IPC_example"
```

Le librerie necessarie per i socket sono le stesse

sys/un.h viene usata per socket Unix

La macro ADDRESS verrà usata come indirizzo per socket Unix che, a differenza dei socket Internet, hanno indirizzi basati su nomi file

```c
void handle_sigchld(int signal_no){
  int child_status = -1;
  wait(&child_status);
  printf("Child process exited\n");
}
```

Utilizzo di socket per IPC

Il child process agisce da server in questo caso, accettando dati e rispondendo con il risultato del processing.

```c
void child_process_routine(pid_t parent_pid){
  // Child agisce da server
  int n, sock_fd, new_sock_fd, cli_len;
  char buffer[BUFSIZE];
  struct sockaddr_un srv_addr, cli_addr;

  // Apertura di un socket
  sock_fd = socket(AF_UNIX, SOCK_STREAM, 0);
  if(0 > sock_fd){
    perror("Child (Server): socket()");
    exit(1);
  }

  memset((char*) &srv_addr, 0, sizeof(srv_addr));
  srv_addr.sun_family = AF_UNIX;
  memcpy(srv_addr.sun_path, ADDRESS, sizeof(ADDRESS));

  // Eliminazione di ogni eventuale named socket per evitare bind fail
  unlink(ADDRESS);
  if(0 > bind(sock_fd, (struct sockaddr *) &srv_addr, sizeof(srv_addr))){
    perror("Child (Server): bind()");
    exit(1);
  }

  listen(sock_fd, 5);
  cli_len = sizeof(cli_addr);
  new_sock_fd = accept(sock_fd, (struct sockaddr *) &cli_addr, &cli_len);
  if(0 > new_sock_fd){
    perror("Child (Server): accept()");
    exit(1);
  }
```

Unix domain

```c
  // Inizia comunicazione
  while(0 != (n = read(new_sock_fd, buffer, BUFSIZE-1))){
    if(0 > n){
      perror("Child (Server): read()");
      exit(1);
    }

    printf("Child: %s received\n", buffer);
    int rcv = atoi(buffer);
    rcv *= 2;
    printf("Child: sending back %d\n", rcv);

    memset(buffer, '\0', strlen(buffer));
    snprintf(buffer, BUFSIZE-1, "%d", rcv);
    n = write(new_sock_fd, buffer, strlen(buffer));
    if(0 > n){
      perror("Child (Server): write()");
      exit(1);
    }

    memset(buffer, '\0', strlen(buffer));
  }

  close(new_sock_fd);
  close(sock_fd);
  exit(0);
}
```

Aperto per lettura, ora viene usato anche in scrittura

Utilizzo di socket per IPC

Il parent process agisce da client, inviando dati da processare ed attendendo il risultato.

```c
void parent_process_routine(pid_t child_pid){
  // Parent agisce da client
  int n, sock_fd;
  char buffer[BUFSIZE];
  struct sockaddr_un srv_addr;

  sock_fd = socket(AF_UNIX, SOCK_STREAM, 0);
  if(0 > sock_fd){
    perror("Parent (Client): socket()");
    kill(child_pid, SIGKILL);
    exit(1);
  }

  memset((char*) &srv_addr, 0, sizeof(srv_addr));
  srv_addr.sun_family = AF_UNIX;
  memcpy(srv_addr.sun_path, ADDRESS, sizeof(ADDRESS));

  if(0 > connect(sock_fd, (struct sockaddr *) &srv_addr, sizeof(srv_addr))){
    perror("Parent (Client): connect()");
    kill(child_pid, SIGKILL);
    exit(1);
  }

  printf("Countdown started\n");
  for(int i=5; i>0; i--){
    printf("%d...\n", i);

    memset(buffer, '\0', strlen(buffer));
    snprintf(buffer, BUFSIZE-1, "%d", i);
    n = write(sock_fd, buffer, strlen(buffer));
    if(0 > n){
      perror("Parent (Client): write()");
      kill(child_pid, SIGKILL);
      exit(1);
    }

    sleep(1);

    n = read(sock_fd, buffer, BUFSIZE-1);
    if(0 > n){
      perror("Parent (Client): read()");
      kill(child_pid, SIGKILL);
      exit(1);
    }

    printf("Parent: %s received\n", buffer);
  }

  close(sock_fd);

  printf("Parent process exited\n");
  exit(0);
}
```

Ogni errore sul parent process dovrebbe terminare il child

Unix domain

Aperto per scrittura, ora viene usato anche per lettura

IPC tramite message queues

L'utilizzo dei dati inviati tramite una connessione tra processi implementata con pipe o socket generalmente avviene in modo ordinato (FIFO).

Nel caso dei socket questo è garantito dall'uso di TCP socket ma non da quelli UDP, mentre con le pipe non c'è alcuna possibilità di scelta.

Message queues

- Una linked list di messaggi mantenuta nel kernel → accesso in qualsiasi ordine, non solo FIFO
- Ogni messaggio consiste da un tipo (sotto forma di numero) e dati associati al messaggio
- Possono essere pubbliche o private
  - Message queue pubblica → accesso da ogni processo che conosce la chiave della coda
  - Message queue privata → accesso dal processo creatore e suoi child processes
- Uno o più processi possono scrivere e leggere messaggi su una coda

Librerie: `sys/types.h`, `sys/ipc.h`, `sys/msg.h`

Utilizzo di message queue

- `int msgget(key_t key, int msgflg)`
  Crea una nuova message queue, restituendo l'identificativo della coda oppure `-1` se la chiamata fallisce.
  Valori ammessi per `key` sono:
  - `IPC_PRIVATE` → coda privata
  - Intero non negativo → coda pubblica con la chiave specificata

  L'argomento `msgflag` invece servono per indicare flag di gestione (ad es. `IPC_CREAT`) ma gli ultimi 9 bit sono usati per i permessi: stesso approccio dei permessi file Unix.

- `int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg)`
  Scrittura di un messaggio `msgp` sulla coda identificata da `msqid`, con flag opzionali rappresentate da `msgflg`.
  I messaggi sono puntatori a variabili di una `struct` definita dall'utente, ma vincolata con dei requisiti minimi; in particolare `msgsz` non rappresenta la dimensione dell'intera variabile puntata da `msgp`, bensì la dimensione del payload del messaggio.
  Restituisce `0` se ha successo, altrimenti `-1`.
- `ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg)`
  Lettura di un messaggio `msgp` di tipo `msgtyp` dalla coda identificata da `msqid`, con flag opzionali rappresentate da `msgflg`.
  Analogamente, `msgsz` rappresenta la dimensione massima del payload e non quella della `struct` del messaggio.

Utilizzo di message queue

Stesso esempio visto con socket: comunicazione bidirezionale

- Message queue si basa sulla definizione di una struct che contenga
  - Primo membro di tipo `long`
  - Il resto dei membri rappresenta i dati associati al messaggio

```
1 struct msgbuf {
2   long mtype;
3   char mtext[1];
4 };
```

Message payload di 1 byte?

Nel main è necessario che la coda sia creata e che la chiave (o ID) della coda sia resa nota ai processi che vogliono utilizzarla allo scopo di IPC:

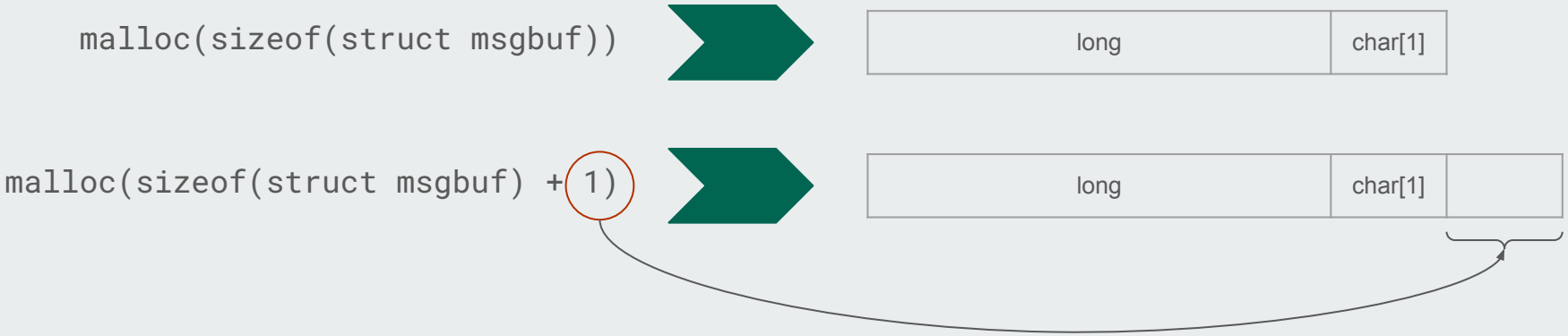
```
1 int queue_id = msgget(IPC_PRIVATE, 0600);
2 if(0 > queue_id){
3   perror("Msg queue get");
4   exit(1);
5 }
```

I permessi Unix `0600` permettono solo ai processi eseguiti dall'utente owner di accedere a questa message queue privata

Utilizzo di message queue


```c
struct msgbuf {
  long mtype;
  char mtext[1];
};
```

Per permettere l'uso di payload di dimensioni arbitrarie si può semplicemente allocare dinamicamente più spazio di `sizeof(struct msgbuf)` per una variabile puntatore:



Utilizzo di message queue


```c
struct msgbuf {
  long mtype;
  char mtext[1];
};
```

La dimensione minima è generalmente fissata ad 1, ma alcuni compilatori ammettono anche dichiarazioni di tipo `char mtext[0]`.

- È consigliabile tenere quindi 1, sia per motivi di portabilità tra compilatori che per ricordare di allocare spazio per il `'\0'`, nel caso in cui il payload del messaggio sia una stringa
  Inoltre, funzioni di utilità per stringhe (ad es. `strlen` e `memset`) possono facilitare l'interazione con i messaggi

Utilizzo di message queue

Il child process riceve la chiave della coda ed i tipi dei messaggi da usare per l'input e l'output (non obbligatorio).

```c
void child_process_routine(int queue_id, long incoming_type, long outgoing_type){
  int n;
  char buffer[BUFSIZE];

  // Allocazione memoria per il messaggio più grande
  struct msgbuf *msg = malloc(sizeof(struct msgbuf) + BUFSIZE - 1);
  if(NULL == msg){
    perror("Child: malloc()");
    exit(1);
  }

  while(-1 != (n = msgrcv(queue_id, msg, BUFSIZE, incoming_type, 0))){
    memcpy(buffer, msg->mtext, strlen(msg->mtext));
    printf("Child: %s received\n", buffer);
    free(msg);

    int rcv = atoi(buffer);
    rcv *= 2;
    printf("Child: sending back %d\n", rcv);

    memset(buffer, '\0', strlen(buffer));
    snprintf(buffer, BUFSIZE-1, "%d", rcv);

    // Crea messaggio di risposta
    struct msgbuf *new_msg = malloc(sizeof(struct msgbuf) + strlen(buffer));
    if(NULL == new_msg){
      perror("Child: malloc()");
      exit(1);
    }

    new_msg->mtype = outgoing_type;
    memcpy(new_msg->mtext, buffer, strlen(buffer));

    n = msgsnd(queue_id, new_msg, strlen(buffer) + 1, 0);
    if(-1 == n){
      perror("Child: msgsnd()");
      free(new_msg);
      exit(1);
    }

    free(new_msg);
    memset(buffer, '\0', strlen(buffer));
  }

  exit(0);
}
```

Dopo aver copiato il payload dal messaggio ricevuto, si può liberare la memoria; analogamente dopo l'invio.

Utilizzo di message queue

Il parent process analogamente al child gestisce i messaggi di input ed output con tipi diversi.

```c
void parent_process_routine(int queue_id, pid_t child_pid, long incoming_type, long outgoing_type){
  int n;
  char buffer[BUFSIZE];
  printf("Countdown started\n");
  for(int i=5; i>0; i--){
    printf("%d...\n", i);

    // Creazione del messaggio
    memset(buffer, '\0', strlen(buffer));
    snprintf(buffer, BUFSIZE-1, "%d", i);
    struct msgbuf *msg = malloc(sizeof(struct msgbuf) + strlen(buffer));
    if(NULL == msg){
      perror("Parent: malloc()");
      kill(child_pid, SIGKILL);
      exit(1);
    }

    msg->mtype = outgoing_type;
    memcpy(msg->mtext, buffer, strlen(buffer));

    // Invio del messaggio
    n = msgsnd(queue_id, msg, strlen(buffer) + 1, 0);
    if(-1 == n){
      perror("Parent: msgsnd()");
      free(msg);
      kill(child_pid, SIGKILL);
      exit(1);
    }
```

```c

    free(msg);
    sleep(1);

    // Attendi messaggio di risposta
    struct msgbuf *new_msg = malloc(sizeof(struct msgbuf) + BUFSIZE - 1);
    if(NULL == new_msg){
      perror("Parent: malloc()");
      kill(child_pid, SIGKILL);
      exit(1);
    }

    n = msgrcv(queue_id, new_msg, BUFSIZE, incoming_type, 0);
    if(-1 == n){
      perror("Parent: msgrcv()");
      free(new_msg);
      kill(child_pid, SIGKILL);
      exit(1);
    }

    memcpy(buffer, new_msg->mtext, strlen(new_msg->mtext));
    printf("Parent: %s received\n", buffer);

    free(new_msg);
  }

  printf("Parent process exited\n");
  exit(0);
}
```

Non solo: va liberata la memoria anche nel caso di errori.

3. Esercizi

Esercizi

1. Scrivere i seguenti tre programmi:
   a. Azione da server, comunica con un processo client tramite socket Unix; riceve l'indirizzo su cui ascoltare connessioni in ingresso tramite riga di comando; trasmette un file testuale al processo connesso una riga per volta
   Hint: argomenti della funzione `main`
   b. Azione da client, comunica con un processo server tramite socket Unix; riceve indirizzo per socket Unix e chiave di una message queue tramite riga di comando; processa ogni riga del file letto calcolandone il numero di caratteri non blankspace, scrivendo un messaggio sulla coda contenente: numero righe lette, media del numero caratteri non blankspace calcolata sulle righe finora processate, e parola più lunga in una riga
   Hint: usare `struct msgbuf` *come vista negli esempi (solo due membri), e va dichiarata in un apposito file* `msgqstruct.h`*; può essere utile interpretare il membro* `mtext` *come byte e non come stringa C*
   c. Azione da wrapper e punto di avvio del programma multi-process: crea una message queue, lancia i due programmi dei punti a e b in nuovi processi tramite `fork` ed `execXX` fornendo i parametri necessari al loro funzionamento; legge i messaggi dalla coda, stampando a video i risultati, ed infine termina l'esecuzione non appena il numero di righe lette eguaglia il termine del file
   Hint: si assume che questo processo conosca a priori il numero di righe del file e che utilizzi la stessa dichiarazione per la `struct` *dei messaggi presente nel file* `msgqstruct.h`*; i programmi dei punti a e b devono essere compilati separatamente da quello del punto c*

Riferimenti

- http://pelusa.fis.cinvestav.mx/tmatos/LaSumA/LaSumA2_archivos/Paralelizacion/Unix%20Multi-Process%20Programming%20and%20Inter-Process%20Communications%20(IPC).htm
- `fork`, `wait`, `signal`, `exec`
- `sigaction`, `kill`, `raise`
- `pipe`, `popen`, `pclose`, `mkfifo`,
- `setsockopt`
- IPC benchmark
- `msgget`, `msgsnd`, `msgrcv`