Introduzione alla programmazione multipla: i **processi** ed i **thread**

Appunti per lezione T.P.S.I.T – A.S. 2016/17

Ver. 0.6 (ultimo aggiornamento 27/03/2017)

@2016 - Prof. Dott. Ing. DI GIUSTO Sandro Ph.D. - sandro.digiusto@malignani.ud.it - www.digiusto.eu

Introduzione alla programmazione multipla

- Approfondimento sui processi
- La duplicazione dei processi – `fork`
- L'esecuzione di nuovi programmi – `exec*`
- I thread POSIX - `pthread`
- Lo scambio dati (IPC)
- Analisi di alcune problematiche
- Bibliografia / sitografia

Introduzione alla programmazione multipla

Cos'è?

- La **programmazione multipla** è quel genere di programmazione atta a permettere la suddivisione efficiente di uno o più compiti da parte di un unico programma (**processo**)
- Può essere implementata almeno attraverso due grosse modalità
  - La programmazione **multi-task** (multi-processo)
  - La programmazione **multi-thread**
- Tale tipo di programmazione ha una gran mole di vantaggi e di campi di utilizzo, ma si porta dietro anche un gran numero di problematiche (alcune delle quali davvero notevoli)
- In un sistema multi-programmato l'**elaborazione** è detta **parallela**
  - In realtà se il sistema è mono-processore è solo *pseudo-parallela* (*concurrency illusion*)

Introduzione alla programmazione multipla

Terminologia: Programma

- Per comprendere in modo chiaro e semplice alcuni degli aspetti più importanti di questa tecnica, occorre avere ben chiari alcuni concetti fondanti che ora ripasseremo
  - **Programma**
    - È una sequenza di istruzioni in un opportuno linguaggio per un determinato OS e per una determinata architettura (linguaggio macchina)
    - E' memorizzato su una memoria secondaria tramite un FS (non in RAM)
    - Non è in esecuzione (è un *entità passiva* gestita dall'OS)
    - La sua esistenza è garantita anche dopo un reboot del sistema
    - La sua esecuzione da parte dell'OS da luogo ad un processo ospitato generalmente in RAM (ma non sempre e necessariamente)

Introduzione alla programmazione multipla

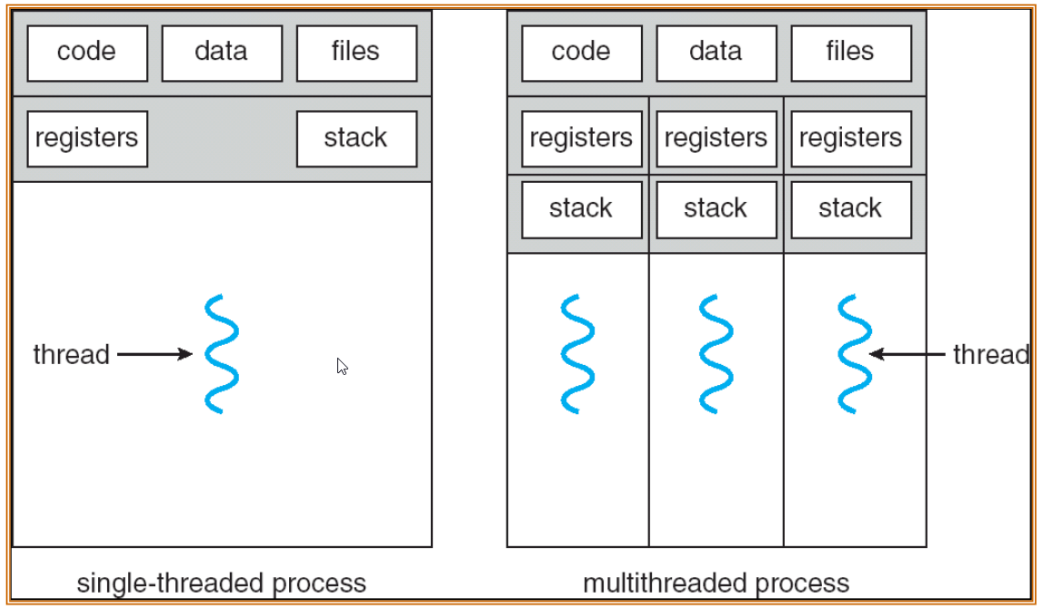
Terminologia: Processo

- **Processo (o Task o processo pesante)**
  - È un *programma* in *esecuzione (è* un *entità attiva* gestita dall'OS)
  - E' *l'unità fondamentale di esecuzione* in un OS e risiede in memoria principale (RAM)
  - Per un unico programma possono essere gestiti contemporaneamente anche N processi distinti (su *OS multiprogrammati*)
  - Può essere rappresentato sinteticamente dai seguenti sotto-elementi:
    - *Codice* : elenco delle istruzioni che dovranno essere eseguite (in formato binario)
    - *Dati* : le variabili globali
    - *Lo stato* : alcuni registri della CPU tra cui il PC (Program Counter, lo Stack-Pointer, …)
    - *Lo stack* : una parte di memoria in cui si allocano le variabili locali, i parametri, gli stack di invocazione di funzioni/procedure
  - Può attingere anche ad altre risorse dedicate o condivise (file, socket, I/O, …)
  - Può trovarsi in più stati:
    - Init, Ready, Running, Waiting, Terminated
    - La gestione degli stati e delle transizioni di stato è compito fondamentale dell'OS (sistemi multi-processo)

Introduzione alla programmazione multipla

Terminologia: Thread

- **Thread (o processo leggero)**
  - È un *programma* in *esecuzione (è* un *entità attiva* gestita dall'OS)
  - E' *un'unità di esecuzione* in un OS che condivide necessariamente codice e dati con altri thread associati (all'interno del processo ospitante)
  - L'intero gruppo di thread associati che condividono codice e dati prende il nome di *Task* (un processo quindi è essenzialmente un task in un sistema che non permette thread ovvero per il quale in un task c'è un solo thread)

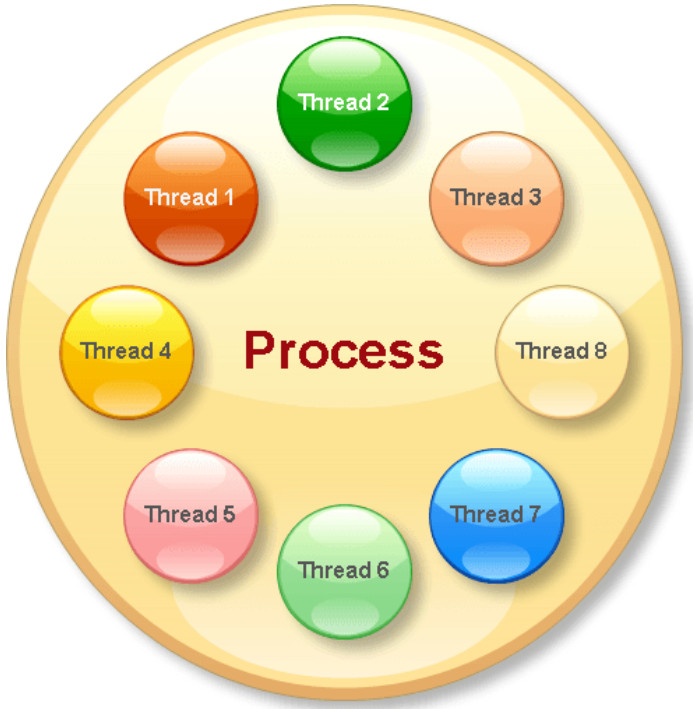


https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

Introduzione alla programmazione multipla

Numerosità relativa a programmi, processi e threads

- In generale si può ritenere che:
  - 1 programma → N processi
  - 1 processo → M threads



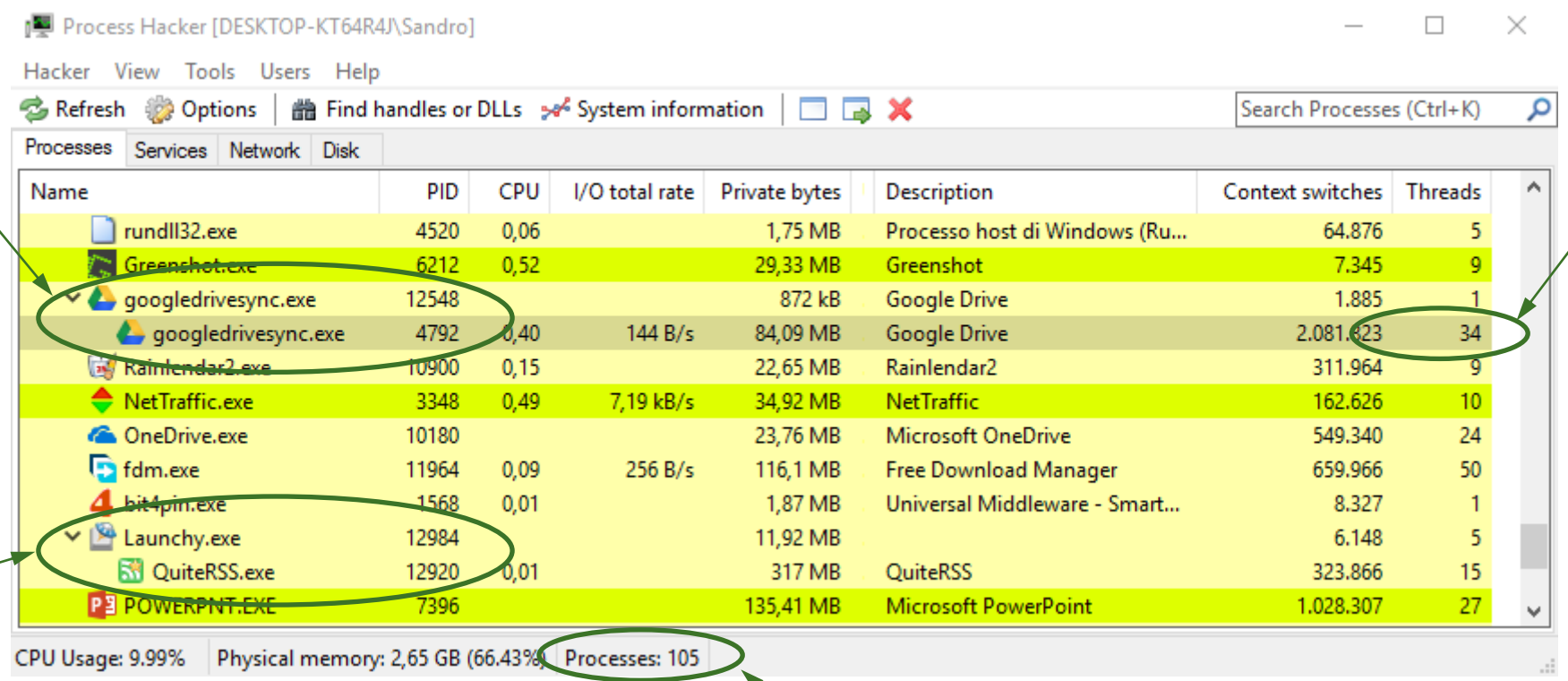
https://ksearch.wordpress.com/2010/09/12/program-vs-process-vs-thread

Introduzione alla programmazione multipla

Numerosità relativa a programmi, processi e threads

Esempio di fork: stesso programma, due processi distinti padre-figlio (due PID)

Un unico processo, molti thread (37!)



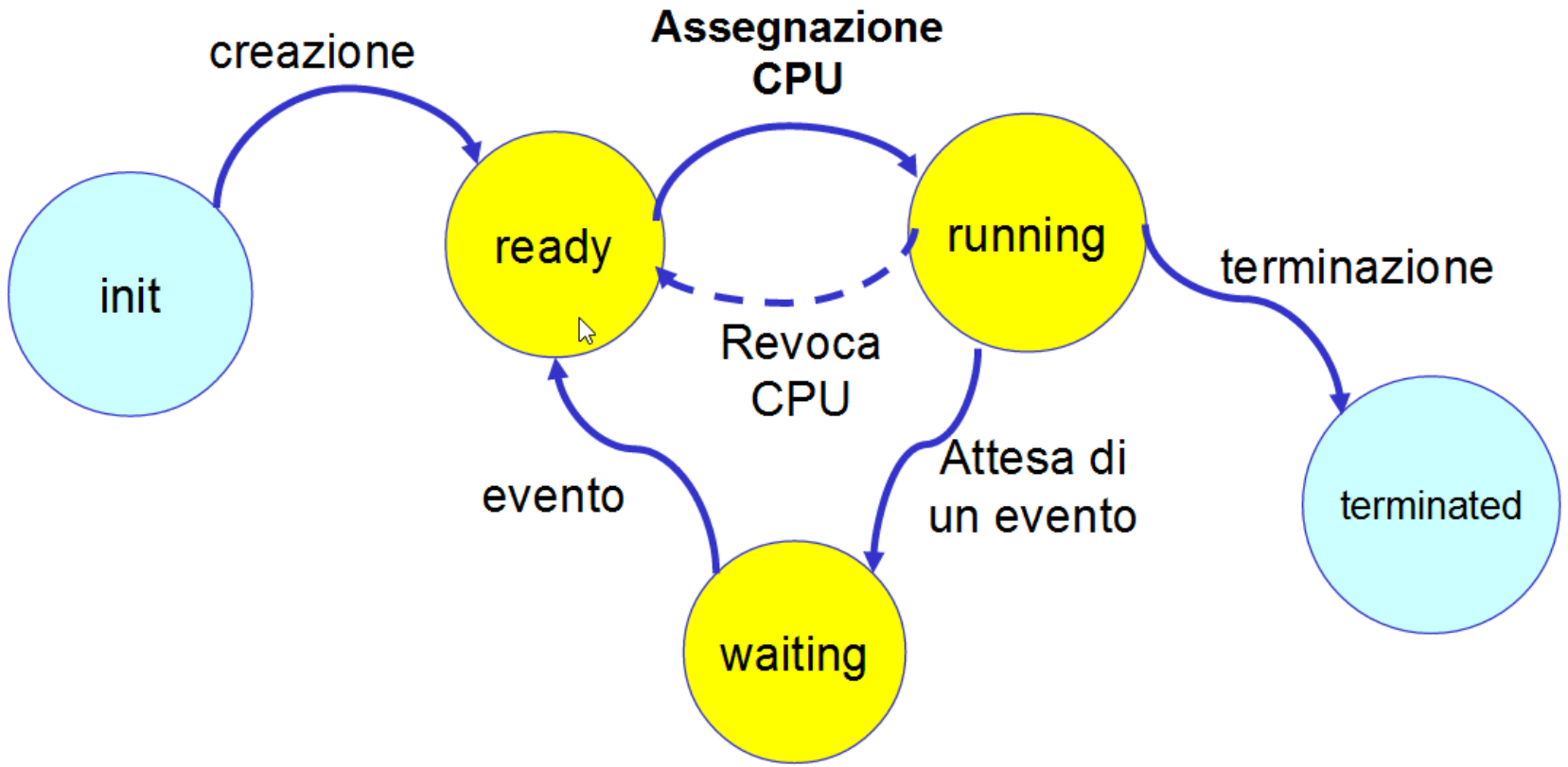
Esempio di exec: i programmi associati ai processi padre-figlio sono diversi

Numero complessivo di processi attualmente in esecuzione (105!)

Approfondimento sui processi

Le transizioni di stato dei processi

› La seguente è la tipica mappa delle transizioni di stato per un processo in un OS multiprogrammato



Approfondimento sui processi

La gestione dei processi da parte dell'OS

- Si ricordi che in un OS multiprogrammato accade sempre che:
  - Un solo processo può contemporaneamente essere in fase di **running**
  - Molti processi possono essere contemporaneamente in fase **ready** o **waiting** (anche nati dallo stesso programma)
- Le cose possono essere diverse (e più complesse) in sistemi multiprogrammati e anche **multiprocessore** (non trattati in questa dispensa)
- La gestione delle informazioni vitali di ogni processo è compito dell'OS ed è molto complessa e delicata
  - Deve tenere sempre distinte ed aggiornate tutte le informazioni (tante)
  - Deve gestire la coda delle transizioni di stato, le politiche di accesso al processore (acquisizione CPU) e lo storico degli eventi I/O

Approfondimento sui processi

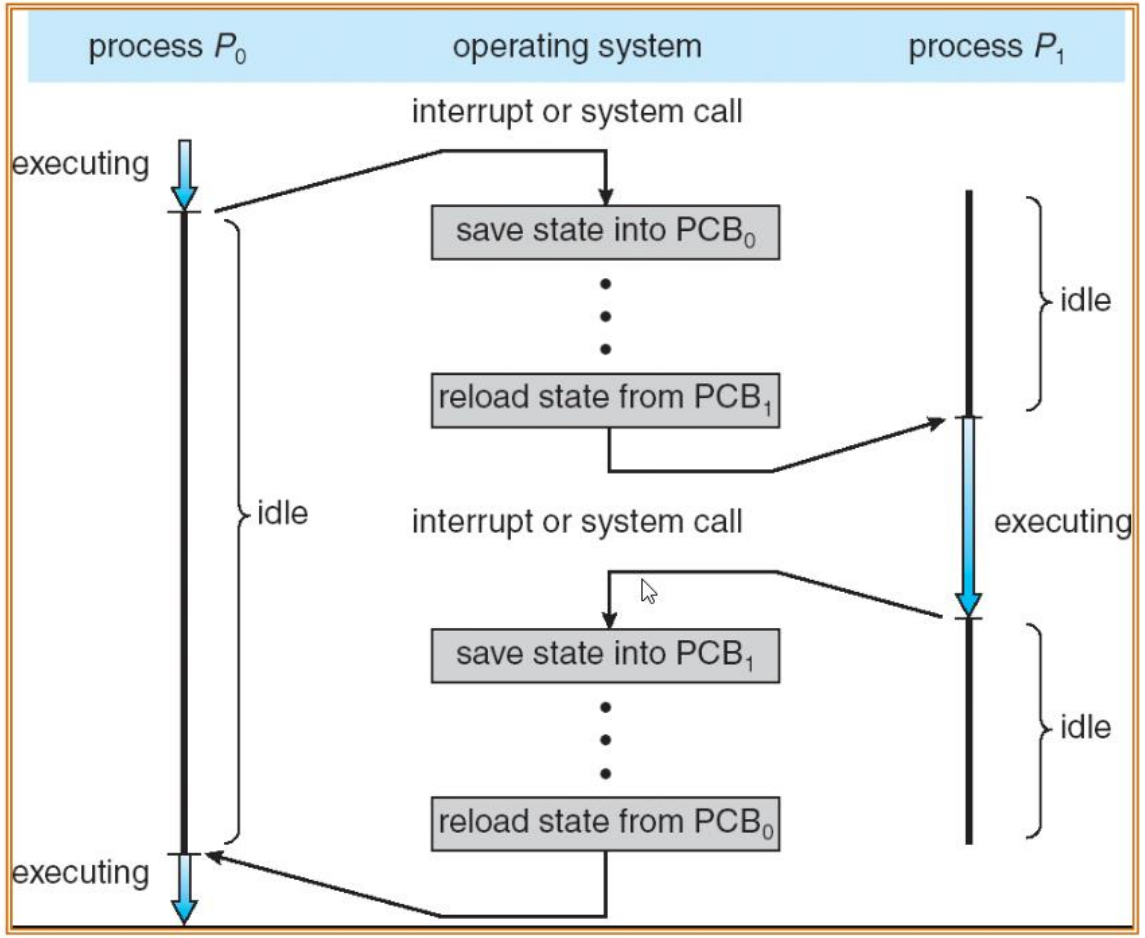
Il PCB

- Il **PCB** (*Process Control Block*) o **Process Table** è quella complessa struttura dati gestita dall'OS che si occupa di mantenere traccia completa di tutte le informazioni di ciascun processo
- Per ogni processo si possono distinguere due componenti:
  - Informazioni sulle **risorse allocate** (files aperti, sockets, figli, …)
  - Informazioni sull'**esecuzione** (codice, stato CPU, stack, …)
- Ogni volta che l'OS deve porre un processo running in un altro stato (waiting, ready) e scegliere tramite **scheduling** un nuovo processo da porre in ready, deve avvenire il cosiddetto **cambio di contesto** (**context switch**)
  - Si tratta di un'operazione inevitabile ma molto spesso onerosa in termini di risorse (a volte talmente tanto da portare il sistema in una fase di stallo più o meno totale noto come *trashing*)

Approfondimento sui processi

Il Context-Switching

› La seguente immagine è esplicativa delle problematiche di cambio contesto



https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/3_Processes.html

Approfondimento sui processi

I meccanismi base (minimi) di un processo

- Ogni OS multiprogrammato deve mettere a disposizione dei processi almeno tre meccanismi di base:
  - Per la creazione
  - Per la terminazione
  - Per l'intercomunicazione (interazione)
- Ovviamente sono tutte operazioni molto delicate ed un loro uso improprio (o abuso) potrebbe rapidamente portare il sistema in instabilità o peggio
  - Per questo motivo questi meccanismi sono gestiti solo dall'OS mediante operazioni privilegiate eseguite direttamente a livello Kernel
    - Nei sistemi Unix sono note con il nome di **System Call**

Approfondimento sui processi

La creazione dei processi

- La creazione dei processi avviene sempre ad opera di un atro processo (gerarchia)
  - Il processo originale viene chiamato padre (**parent**)
  - Il processo avviato dal padre viene chiamato processo figlio (**child**)
  - L'insieme dei processi figli di un processo padre si chiamano **siblings**
- La natura del processo figlio può però essere di due tipi:
  - La stessa del padre (si parla di clonazione o fork)
  - Diversa dal padre (si parla di esecuzione)
- I processi in breve tempo danno luogo a vere e proprie gerarchie ed annidamenti
  - Ogni processo può (e deve) avere uno ed un solo padre
  - Ogni processo padre può avere da zero a N processi figli
- Il primo processo eseguito automaticamente all'avvio di un sistema è chiamato di **bootstrap** o **init** (iniziatore) ed ha PID pari ad 1

Approfondimento sui processi

L'interazione tra processi

- Tra padre e figlio esistono delle relazioni che ne permettono varie forme di interazione
  - Generalmente i padri (e più in generale gli avi) mantengono il privilegio di interruzione dei figli
  - Gli OS Unix hanno una gerarchia ferrea e disciplinata (*process group*)
  - Gli OS Windows hanno una gerarchia più labile e i padri possono talvolta cedere dei loro diritti ai figli

Approfondimento sui processi

Aspetti caratteristici dell'interazione tra processi

- Concorrenza
  - Padre e figlio possono procedere in parallelo (Unix like)
  - Padre e figlio procedono in modo alternato (il padre attende la chiusura del figlio per procedere)
- Condivisione di risorse
  - Le risorse del padre sono condivise con il figlio (Unix like)
  - Il figlio può usare le risorse del padre solo se esplicitamente richieste
- Spazio di indirizzamento
  - Lo spazio di indirizzamento del figlio è una copia di quello del padre (in Unix con i `fork`)
  - Lo spazio di indirizzamento, relativamente sia ai dati che al codice, di padre e figlio sono distinti (in Unix con l'`exec`)

Approfondimento sui processi

La terminazione dei processi

- Per capirne le caratteristiche occorre ricordare che le gerarchie di processi prevedono che:
  - Ogni processo (tranne *init*) è figlio di qualcun altro
  - Ogni processo può avere un gran numero processi figli
- E' compito dell'OS gestire la gerarchia (parentela) e si può intuire che non è un'operazione banale
  - Il descrittore di un processo deve avere un riferimento al processo padre (**PPID**)
  - Un padre deve poter rilevare lo **stato di terminazione** dei suoi figli
    - I processi figli dopo terminati diventano **zombie** in attesa della lettura di stato del padre
  - La terminazione del padre deve far terminare anche tutti i processi figli
    - I processi figli diventano degli zombie orfani e vengono quindi ereditati da *init* e poi definitivamente terminati

Approfondimento sui processi

Approfondimento sull'interazione tra processi

- In base al livello di interazione tra due processi P_A e P_B si definiscono
  - *Processi indipendenti*, quei processi per i quali P_A e P_B non prevedono nessuna forma di interazione o condivisione (l'uno non influenza l'altro e viceversa)
  - *Processi interagenti*, quei processi per cui l'esecuzione di P_A può dipendere ed essere influenzata in qualche modo da quella di P_B (ma anche viceversa)
- Il livello di interazione può essere di vari tipi:
  - *Interazione cooperativa*: è quella prevista dal programmatore ed il cui scopo è proprio quello di gestire i processi affinché operino per uno scopo unico
  - *Interazione competitiva*: è una forma parzialmente prevedibile ma non del tutto voluta in cui i processi competono per ottenere una stessa risorsa
  - *Interazione interferente*: è una forma difficilmente prevedibile e generalmente non voluta e dannosa (da evitare con accorgimenti ad-hoc)

Introduzione alla programmazione multipla

Approfondimento sull'interazione tra processi

- Nelle modalità di interazione lo scambio di informazioni tra PA e PB può avvenire ad esempio in un ambiente *locale* o *globale*:
  - *Ambiente locale*: si usano gli scambi di messaggi (nelle varie forme previste)
    - I processi decidono cosa, come e con chi scambiarsi i messaggi (informazioni)
    - E' una modalità molto sicura ma indubbiamente lenta (per via dell'overhead del sistema di gestione dei flussi e della partecipazione diretta dell'OS nelle operazioni)
    - Difficilmente porta a problemi di interferenza
  - *Ambiente globale*: si usano le memorie condivise (e tool associati)
    - I processi e/o i thread ottengono dall'OS il permesso di condividere contemporaneamente una porzione di memoria (variabili condivise)
    - E' una modalità rapida ed efficiente
    - Espone il programmatore ed il sistema a molte problematiche di sicurezza dirette ed indirette
    - E' spesso origine di problematiche di interferenza

La duplicazione dei processi - `fork`

Duplicare (clonare) i processi

- Come abbiamo già visto, la duplicazione dei processi implica la generazione di un nuovo processo figlio PF da parte di un processo padre PP, identico a se stesso
  - I due processi hanno PID diversi
  - I due processi, pur se idealmente identici, riescono a sapere chi sono (il padre può sapere d'essere il padre, il figlio può sapere d'essere il figlio)
  - La funzione essenziale per la duplicazione di un processo è la `fork`
    - Si tratta dell'unica funzione che ritorna due volte
      - Una volta nel padre ad indicare la corretta «clonazione»
      - Una volta nel figlio ad indicare il corretto avvio
    - Si osservi che dopo la clonazione, il processo figlio non parte dal punto di ingresso (come avviene solitamente) ma dalla prima istruzione che segue il `fork`
    - Ovviamente anche il padre prosegue la propria esecuzione a partire dalla prima istruzione che segue la `fork`

La duplicazione dei processi - `fork`

Duplicare i processi

```
$ ps ax
      PID    PPID    PGID     WINPID   TTY         UID    STIME COMMAND
     9100   11948    9100       8468  cons2     197609 11:43:55 /cygdrive/c/Users/San
dro/Documents/Codice/sockets/esercizi/09-Read-chars-and-nums/Read-chars-and-nums-Clie
nt
    12280       1   12280      12280  cons1     197609 11:33:03 /usr/bin/bash
     5080    5324    9356       5080  cons0     197609 11:43:55 /cygdrive/c/Users/San
dro/Documents/Codice/sockets/esercizi/09-Read-chars-and-nums/Read-chars-and-nums-Serv
er
     9356    3188    9356      12084  cons0     197609 11:43:52 /usr/bin/make
    10440   12280   10440      10188  cons1     197609 11:43:59 /usr/bin/ps
     5324    9356    9356      10676  cons0     197609 11:43:52 /cygdrive/c/Users/San
dro/Documents/Codice/sockets/esercizi/09-Read-chars-and-nums/Read-chars-and-nums-Serv
er
    11948       1   11948      11948  cons2     197609 11:37:41 /usr/bin/bash
     3188       1    3188       3188  cons0     197609 11:32:50 /usr/bin/bash
```

Name	PID	CPU
	8896	0,01
xe	10644	0,53
xe	11788	2,38
nHookAppWIN6064.exe	11156	0,02
nHookAppWIN6032.exe	8356	0,04
xe	2172	0,31
	8616	0,20
	2660	
	5908	
	11664	
	10900	0,02
	776	
Console.exe	740	0,36
bash.exe	3188	0,22
conhost.exe	7320	0,21
bash.exe	12280	0,22
conhost.exe	11312	0,22
bash.exe	11948	0,30
conhost.exe	7368	0,28
notepad++.exe	5820	
MpCmdRun.exe	5388	
make.exe	12084	
Read-chars-and-nums-Server.exe	10676	0,02
Read-chars-and-nums-Server.exe	5080	0,02
Read-chars-and-nums-Client.exe	8468	

- Processo padre (5324) e figlio (5080) mediante `fork`
- Figlio di *Init* (1)
- WinPid

La duplicazione dei processi - `fork`

La funzione `fork`

```
<sys/types.h>
<unistd.h>
pid_t fork( void )
```

- La funzione, invocata nel processo padre, chiede all'OS di generare un processo figlio clone
- Non prevede argomenti e tecnicamente restituisce un intero che può essere:
  - Per il padre:
    - il PID del processo figlio (se l'operazione è andata a buon fine)
    - -1 se l'operazione di clonazione non è stato portato a termine
  - Per il figlio:
    - 0 (che non può essere PID di nessun processo per cui non c'è ambiguità)
- In caso di errori nella clonazione, la variabile globale `errno` permette di ottenere informazioni circa la causa

La duplicazione dei processi - `fork`

La funzione `fork`

- La variabile `errno` può essere:
  - `EAGAIN` : per indicare che l'operazione non si è potuta completare per cause probabilmente temporanee (per cui si può riprovare in seguito)
    - Le cause più tipiche sono legate alla mancanza attuale di risorse libere dell'OS per dar vita al nuovo processo (mancanza di memoria o saturazione del numero massimo di processi possibili)
  - `ENOMEM` : per indicare che l'OS non è riuscito ad allocare le strutture dati necessarie (nel kernel) per la gestione del processo
    - A fronte di questo errore generalmente non è ragionevole ritentare in seguito la stessa operazione poiché fallirà nuovamente

La duplicazione dei processi - `fork`

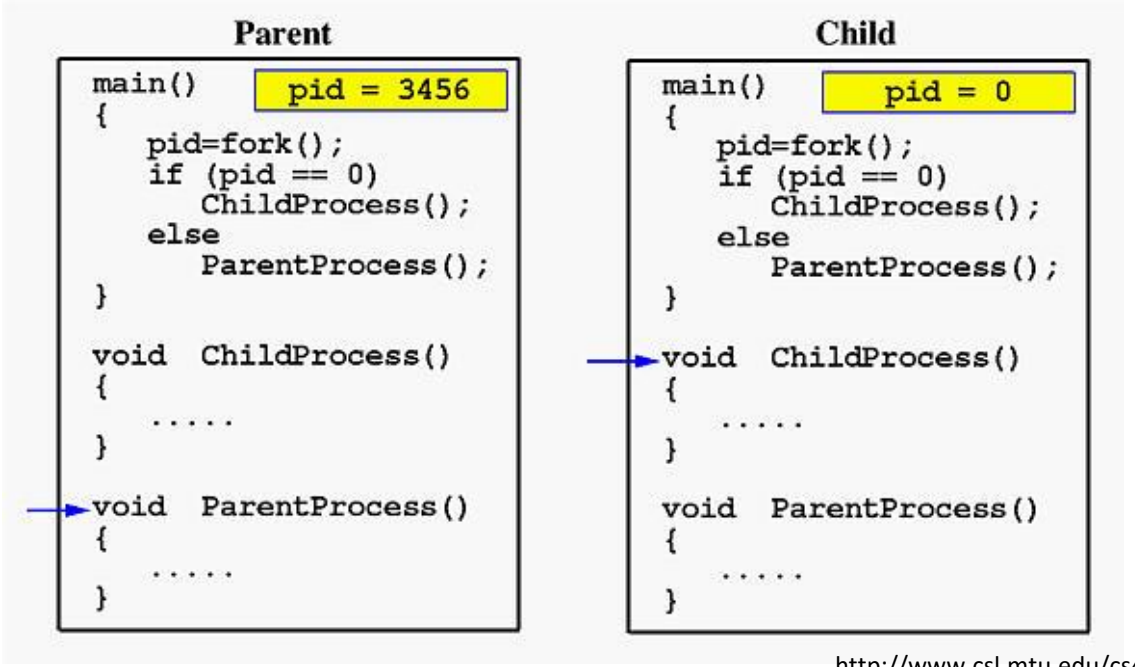
La funzione `fork`

- E' importantissimo osservare che padre e figlio sono a «quasi» tutti gli effetti processi identici (stesso codice, stesso stack, stesse variabili), ma solo inizialmente!
  - Essi infatti **non condividono la stessa memoria**, ma ne hanno semplicemente una copia che **inizialmente è identica**
  - La parte di memoria realmente condivisa è il segmento testo (costanti & co) ma è protetto in scrittura per tutti i processi (padre e figlio)
  - La parte di memoria copiata (non condivisa) è generalmente gestita con la tecnica *copy on write*
    - La memoria viene effettivamente copiata nel processo figlio solo a seguito del primo tentativo dei processi di modificarla (fino ad allora è in comune)

La duplicazione dei processi - `fork`

Esempio d'uso `fork`

› La funzione `fork` può essere gestita in molti modi diversi, ma una struttura di massima abbastanza comune è quella che prevede nel `main` l'analisi del valore di ritorno di `fork` e l'invocazione quindi di due funzioni distinte a seconda se il processo si accorge d'essere il padre (`pid!=0`) o il figlio (`pid==0`)



http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html

La duplicazione dei processi - fork

Esempio d'uso fork

```
01 #include <stdio.h>
02 #include <sys/types.h>
03 #include <unistd.h>
04 #include <stdlib.h>
05
06 int main( ) {
07   int pid;
08   if ( (pid = fork() ) <0 ) {
09     perror("Errore nella creazione del processo figlio");
10     return -1;
11   }
12   if ( pid==0 ) {// Codice svolto dal processo figlio
13     printf("Questa printf è invocata dal processo figlio\n");
14   } else {  // Codice svolto dal processo padre
15     printf("Questa printf è invocata dal processo padre\n");
16   }
17   return (EXIT_SUCCESS);
18 }
```

La duplicazione dei processi - `fork`

Le funzioni sui PID

```
<sys/types.h>
<unistd.h>
pid_t getpid( void )
pid_t getppid( void )
```

- Talvolta un processo potrebbe (per vari motivi) avere necessità di conoscere il proprio PID o il PID del proprio genitore (PPID)
  - La funzione `getpid` permette, se invocata, di ottenere il PID del processo invocante
  - La funzione `getppid` permette, se invocata, di ottenere il PPID ovvero il PID del proprio padre (*parent*)
  - Le due funzioni non prevedono segnalazione di errori

La duplicazione dei processi - `fork`

Le funzioni sui PID

```
01 #include <stdio.h>
02 #include <sys/types.h>
03 #include <unistd.h>
04 #include <stdlib.h>
05
06 int main( ) {
07   int pid;
08     if ( (pid = fork() ) <0 ) {
09     perror("Errore nella creazione del processo figlio");
10     return -1;
11   }
12   if ( pid==0 ) {// Codice svolto dal processo figlio
13     printf("Figlio: Il mio PID e':%d e quello di mio padre e':%d\n", getpid(),
getppid() );
14   } else {  // Codice dal processo padre
15     printf("Padre: Il mio PID e':%d e quello di mio padre e':%d\n", getpid(),
getppid() );
16     printf("Padre: Il valore restituito da fork e':%d\n", pid);
17   }
18     return (EXIT_SUCCESS);
19 }
```

La duplicazione dei processi - `fork`

Padre e figlio – Cos'hanno in comune

› Dopo l'esecuzione della `fork`, i processi padre e figlio hanno diverse **risorse e caratteristiche in comune**:
  – I file aperti
  – Gli identificatori d'accesso (UID, GUID, …)
  – Gli identificatori di sessione (SID, PGID, il terminale TTY)
  – La directory di lavoro e la directory radice
  – Le maschere dei permessi
  – Le maschere dei segnali bloccati
  – I segmenti di memoria condivisa agganciati al processo
  – I limiti sulle risorse
  – Le priorità e le affinità di processore
  – Le variabili d'ambiente

La duplicazione dei processi - `fork`

Padre e figlio – Cosa NON hanno in comune

- Dopo l'esecuzione della `fork`, i processi padre e figlio hanno anche diverse **risorse e caratteristiche NON in comune**:
  - Il valore di ritorno della `fork` (visto in precedenza)
  - PID e PPID
  - I valori sui tempi di esecuzione e risorse (posti a 0 nel figlio)
  - I lock sui file e sulla memoria condivisa (non ereditati)
  - Gli allarmi, i timer e i segnali pendenti (cancellati nel figlio)
  - Le operazioni di I/O asincrono (non ereditate)
  - Altre caratteristiche «minori»

La duplicazione dei processi - `fork`

Le funzioni di attesa e ricezione di stato

```
<sys/types.h>
<sys/wait.h>
pid_t wait( int *status )
```

- Chiede al processo attuale (padre) di attendere fino alla conclusione di un qualunque processo figlio
- La funzione restituisce:
  - il PID del processo figlio concluso in caso di successo
  - -1 in caso di errore (nel qual caso `errno` contiene informazioni circa le problematiche occorse)
    - Tipicamente può occorre l'errore `EINTR` che indica l'occorrenza di un segnale che ha interrotto la funzione `wait`
- La variabile `status` conterrà al termine lo *stato di terminazione* (valore di ritorno) del processo figlio osservato (accodati ad un byte LSB costituito da informazioni extra generalmente non di interesse)

La duplicazione dei processi - `fork`

Le funzioni di attesa e ricezione di stato

- Si presti attenzione che nel caso in cui il figlio sia già terminato prima dell'invocazione di `wait`, essa ritornerà immediatamente comunicando il PID del processo già terminato (*zombie*)
  - In questo caso è usato per ottenere lo stato di terminazione e liberare definitivamente le risorse del processo figlio
  - Se più processi figli erano già terminati, occorrerà invocare ripetutamente `wait` per ciascuno di essi
- L'eccessiva genericità nella risposta della funzione `wait` (nel senso che è imprevedibile sapere a quale processo figlio faccia riferimento) la rende di fatto molto complessa da usare e quindi inefficiente/inefficace
- La funzione `waitpid` pone rimedio a questo e gli andrebbe perciò sempre preferita

La duplicazione dei processi - `fork`

Le funzioni di attesa e ricezione di stato

```
01 #include <stdio.h>
02 #include <sys/types.h>
03 #include <unistd.h>
04 #include <stdlib.h>
05 #include <sys/wait.h>
06
07 int main( ) {
08   int pid, closed_pid, status;
09     if ( (pid = fork() ) <0 ) {
10     perror("Errore nella creazione del processo figlio");
11     return -1;
12   }
13   if ( pid==0 ) {  // Codice svolto dal processo figlio
14     printf("Figlio: Io sono il processo:%d e mi chiudo con lo stato %d\n", getpid(), 123 );
15     sleep(5);  // Attendo qualche secondo per accertarmi che il figlio si chiuda dopo l'invocazione di
wait nel padre
16     return 123;
17   } else {  // Codice dal processo padre
18     printf("Padre: Il mio PID e':%d e ora attendo che il mio processo figlio (%d) si chiuda\n", getpid(),
pid );
19     closed_pid = wait( &status );
20     printf("Padre: Il mio processo figlio (%d) si è chiuso restituendomi lo stato: %d\n", closed_pid,
status>>8);
21   }
22     return (EXIT_SUCCESS);
23 }
```

La duplicazione dei processi - `fork`

Le funzioni di attesa e ricezione di stato

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

- La funzione attende la chiusura di uno specifico (volendo) processo figlio `pid` e ne comunica lo stato di terminazione `status`
  - Se `pid` è -1 (che equivale a `WAIT_ANY`) si riottiene la funzionalità di `wait`, ovvero dell'attesa di qualunque processo figlio
  - Se `pid`>0 si attende uno specifico processo PID
  - Se `pid`<-1 o `pid`=0 si ottengono alcune altre funzionalità avanzate legate al PGID
- Attraverso `options` è possibile specificare alcune opzioni avanzate aggiuntive tra cui `WNOHANG` per fare in modo che la funzioni non sia bloccante (ritorna immediatamente il valore 0 nel caso non ci siano processi figli terminati)

La duplicazione dei processi - `fork`

Esercizi

- **`1-Data-not-shared :`**
  - Si scriva un piccolo programma in C che dimostra mediante output che le modifiche ad una variabile locale di un processo (il padre) non si ripercuotono anche nel figlio (e viceversa)
- **`2-Multi-fork :`**
  - Si scriva un programma che genera un processo figlio e che a sua volta genera un processo figlio (nipote). Padre e figlio devono restare bloccati in attesa che il nipote si chiuda. Dopo l'avvio, il nipote deve generare un numero intero casuale N da 1 a 10 e deve stampare quindi a video la sequenza di numeri in countdown (uno al secondo) e restituire come stato di terminazione il numero N stesso. Il processo figlio si deve quindi sbloccare ed analizzare lo stato di terminazione ricevuto (N) e quindi far partire a sua volta un countdown a partire da 2*N e restituire a sua volta lo stato di terminazione N al padre. Il padre si deve quindi sbloccare e stampare a video l'indicazione che figlio e nipote si sono conclusi dopo un tempo pari a 3*N.

La duplicazione dei processi - `fork`

Esercizi

- **`3-Guess-mathematical-operation-server-fork :`**
  - Si scriva un programma in C che deve simulare il server (non-bloccante) dell'esercizio 10 degli appunti sulla programmazione di rete con i socket, realizzandolo mediante duplicazione di processi (fork).

L'esecuzione di nuovi programmi – `exec*`

Eseguire nuovi programmi

- In molti contesti è utile per un programma (processo) invocare e mandare in esecuzione un altro programma (non banalmente una copia di se stesso)
  - Si pensi ad una shell, che accetta dei nostri parametri da riga di comando, li interpreta e poi invoca (se richiesto) nuovi programmi (non quindi copie della shell stessa)
- In questi casi la clonazione con `fork` non è la risposta giusta poiché andrei a gestire un processo del tutto identico a quello che l'ha invocato (compreso il codice)
- La funzione dedicata a questo è la famosa e temuta `exec`
  - In realtà la complessità di tale funzionalità (ed i suoi tanti casi d'uso) ha dato vita ad un'intera famiglia di funzioni, definite nella libreria Standard del C

L'esecuzione di nuovi programmi – `exec*`

Eseguire nuovi programmi

- Per comprendere come in C sia stata implementata l'esecuzione di nuovi programmi, occorre sottolineare alcune particolarità:
  - L'unico modo per eseguire un nuovo processo è l'uso della `fork` (che però abbiamo visto genera un nuovo processo clone dell'attuale)
  - L'esecuzione di un nuovo programma prevede, a differenza della più semplice clonazione, necessariamente:
    - Che ci sia la possibilità di indicare il nome del nuovo programma (eseguibile) ed il suo posizionamento sul FS (*path*)
    - Che ci sia la possibilità di indicare al nuovo programma da eseguire, eventuali parametri di configurazione (così come si fa da shell attraverso i parametri a riga di comando `argc`, `argv`)
    - Che ci sia la possibilità d indicare al nuovo programma eventuali variabili d'ambiente ereditate dal chiamante o del tutto nuove (*environment*)
  - Per dare risposte efficaci a tutte queste necessità sono state definite varie funzioni `exec*` , ma noi vedremo e studieremo solo una (`execv`)

L'esecuzione di nuovi programmi – `exec*`

Eseguire nuovi programmi

```
<unistd.h>
int execv( const char* path, char* const argv[] )
```

- La funzione esegue il programma voluto (`argv[0]`) e presente nel percorso sul FS `path`, senza cercare anche in altri percorsi
- Non genera un nuovo processo, ma rimpiazza semplicemente la sezione del codice del processo attuale con quello del nuovo programma da eseguire (generalmente va quindi eseguita dopo una `fork`)
- Non cambia il proprio PID (poiché non si tratta di un nuovo processo)
- Dopo il caricamento del nuovo segmento di codice, inizia l'esecuzione dall'entry-point del programma (esecuzione del `main`)
- Passa al `main` il vettore di argomenti da riga di comando `argv` in modo piuttosto analogo a quanto accadrebbe se lanciato da shell (non c'è `argc`, per cui la lista di `argv` va analizzata fino a trovare il puntatore `NULL`)
- Non si passa un elenco specifico di variabili d'ambiente

L'esecuzione di nuovi programmi – `exec*`

Eseguire nuovi programmi

- Il valore restituito ha senso solamente nel caso in cui la funzione abbia generato un errore, ovvero non si sia potuto avviare il nuovo programma richiesto con i parametri specificati
  - Se `execv` è invece riuscita ad eseguire il nuovo programma, essa non ritorna più semplicemente perché il corrispettivo codice del programma chiamante non esisterà più (soppiantato dal codice del programma chiamato)
  - Se la funzione ritorna un valore `<0` significa che `execv` non è stata eseguita correttamente e in questo caso `errno` contiene indicazioni sulle cause dell'errore, interpretabili con la funzione `perror`
  - Alcuni motivi per cui `execv` potrebbe fallire sono imputabili a:
    - `EACCESS` : Mancanza di privilegi e/o permessi per accedere e/o eseguire il file
    - `ENOENT` : Il file specificato non esiste
    - `ENOMEM` : Non ci sono sufficienti risorse di memoria per eseguire il nuovo programma

L'esecuzione di nuovi programmi – `exec*`

Eseguire nuovi programmi

- Il valore restituito ha senso solamente nel caso in cui la funzione abbia generato un errore, ovvero non si sia potuto avviare il nuovo programma richiesto con i parametri specificati
  - Se `execv` è invece riuscita ad eseguire il nuovo programma, essa non ritorna più semplicemente perché il corrispettivo codice del programma chiamante non esisterà più (soppiantato dal codice del programma chiamato)
  - Se la funzione ritorna un valore `<0` significa che `execv` non è stata eseguita correttamente e in questo caso `errno` contiene indicazioni sulle cause dell'errore, interpretabili con la funzione `perror`
  - Alcuni motivi per cui `execv` potrebbe fallire sono imputabili a:
    - `EACCESS` : Mancanza di privilegi e/o permessi per accedere e/o eseguire il file
    - `ENOENT` : Il file specificato non esiste
    - `ENOMEM` : Non ci sono sufficienti risorse di memoria per eseguire il nuovo programma

L'esecuzione di nuovi programmi – exec*

Eseguire nuovi programmi

```
01 #include <stdio.h>
02 #include <unistd.h>
03 #include <stdlib.h>
04
05 int main( int argc, char** argv ) {
06   int index;
07   char *argv2[10+1] = {"chiamato.exe"};
08   for (index=1; index< (argc<10?argc:10); index++)
09     argv2[index] = argv[index];
10   argv[argc] = NULL;
11   printf("CHIAMANTE: name=%s , PID=%d\n", argv[0], getpid() );
12   printf("Fra 10 secondi invocherò il chiamato\n");
13   sleep(10);
14   if ( execv("./chiamato.exe", argv2) <0 ) {
15     perror("Errore in execv");
16     exit(EXIT_FAILURE);
17   }
18   printf("Questa scritta non la leggeremo mai!\n");
19     return (EXIT_SUCCESS);
20 }
```

L'esecuzione di nuovi programmi – exec*

Eseguire nuovi programmi

```
01 #include <stdio.h>
02 #include <unistd.h>
03 #include <stdlib.h>
04
05 int main( int argc, char** argv ) {
06    int index=0;
07    printf("CHIAMATO: name=%s , PID=%d\n", argv[0], getpid() );
08    printf("Questo l'elenco dei parametri da riga di comando
ereditati dal chiamante:\n");
09    while ( argv[index]!=NULL )
10       printf("%s ", argv[index++]);
11    printf("\nQuesta scritta la leggeremo perche' la invoca il
chiamato!");
12    while(1);  // Un ciclo infinito per vedere in task manager il
nuovo programma
13       return (EXIT_SUCCESS);
14 }
```

L'esecuzione di nuovi programmi – `exec*`

L'ereditarietà

- Da un punto di vista semplificato si potrebbe pensare che in realtà il nuovo programma (processo) in esecuzione dopo `exec*` non ha più nulla a che vedere con il precedente (chiamante)
  - In realtà ci sono tanti dettagli che continuano a legare il chiamante al chiamato (oltre al PID che abbiamo già visto non cambiare)
    - Il PPID, SID, UID, GID
    - Il terminale di controllo (TTY)
    - Eventuali limiti sulle risorse
    - I tempi mancanti per un allarme lanciato nel chiamante
    - Le maschere dei segnali
    - La directory radice e di lavoro
    - Alcuni tempi di lavoro
    - ….

L'esecuzione di nuovi programmi – `exec*`

L'ereditarietà

- Ci sono però anche dettagli che, come ovvio, non continuano a legare il chiamante al chiamato (non sono ereditati):
  - I segnali pendenti
  - Le mappature dei file in memoria (handlers), ma con qualche eccezione
  - I segmenti di memoria condivisa ed i blocchi sulla memoria
  - Le funzioni registrate per l'uscita (`atexit`, `_exit`)
  - I semafori e le code di messaggi
  - …

- Per completezza si fa noti anche che:
  - La funzione `exec*` può eseguire anche programmi non nativamente eseguibili (esempio script) ma in tal caso occorrono alcune accortezze
  - Le altre funzioni della famiglia (`execve`, `execvp`, `execl`, `execle`, `execlp`) si differenziano essenzialmente per i modi di passare gli argomenti di comando e le variabili d'ambiente (ma il funzionamento è identico)

L'esecuzione di nuovi programmi – `exec*`

Esercizi

- **`4-Fork-and-execv` :**
  - Si scrivano due semplici programmi, un chiamante ed un chiamato. Il chiamante deve richiedere un parametro N (numero) da riga di comando per sapere quanti processi figli deve lanciare. Subito dopo deve creare N processi diversi e ciascuno dei quali deve eseguire poi il programma chiamato passandogli come argomento il proprio identificatore univoco (un numero intero progressivo tra 1 e N). Dopo l'esecuzione del N-simo processo il chiamante si deve mettere in attesa per leggere lo stato di terminazione di ciascuno dei suoi N figli, stampando di volta in volta il PID del processo terminato ed il suo stato (ed al termine di tutti una dicitura per comunicare il completamento). Il programma chiamato deve limitarsi invece a stampare una semplice stringa di benvenuto in cui sia compresa l'informazione dinamica circa la propria identità (il numero ricevuto come argomento dal chiamante), il numero complessivo di figli N ed il proprio PID.

I thread POSIX - `pthread`

Perché POSIX

- Negli anni la comprensione sull'importanza di un implementazione dei *thread* divenne chiara, per permettere prestazioni ed un uso di risorse migliori rispetto alla classica architettura a *processi*
  - Tuttavia la soluzione a thread è stata a lungo esente da veri standard, poiché venivano gestiti in modo più o meno nativo ma diverso dai vari produttori di hardware
- Negli anni '90 i sistemi *Unix* hanno cercato di porre rimedio a questa giungla di implementazioni proprietarie codificando una versione standard dei thread chiamata **POSIX thread** o semplicemente **pthread**
  - Si tratta però di una libreria extra (`pthread.h`) non facente parte né dello Standard C né della libreria Standard C
  - Le varie implementazioni possono differire e portare risultati diversi!

I thread POSIX - pthread

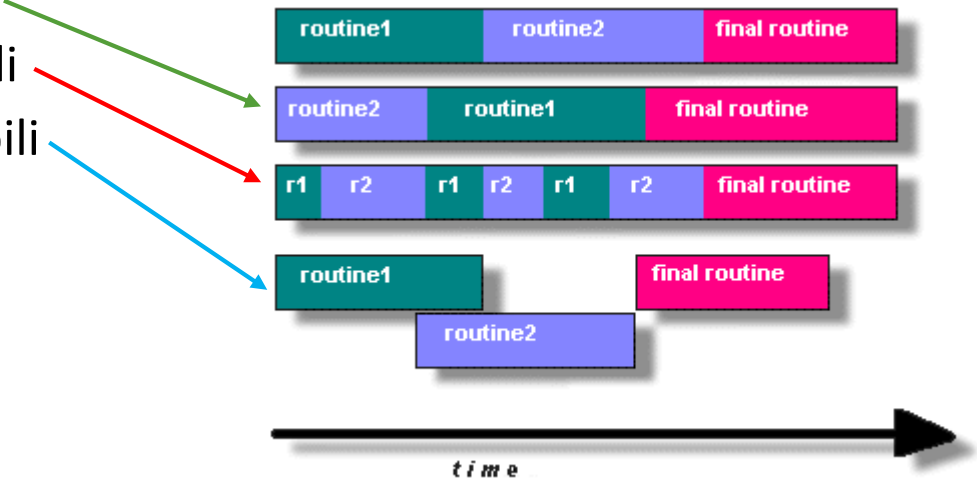
Quali vantaggi

- L'implementazione dei `pthread` gode di diversi punti forti:
  - Leggerezza
    - Confrontato con i processi, il minor tempo impiegato nella creazione è mediamente di almeno un fattore 10 (ma può avvicinarsi a 100 su architetture hardware multicore)
  - Semplice ed efficiente sistema di comunicazione
    - La comunicazione tra thread è efficiente perché non viene svolta mediante copie di dati, ma direttamente con dei dati trasversalmente disponibili (stessa locazione)
    - In questi casi il collo di bottiglia per le prestazioni è spesso il sistema di caching delle CPU o il gestore CPU-memoria
  - Funzionalità
    - Con i thread è possibile gestire facilmente I/O asincroni
    - E' semplice gestire priorità diverse per ciascun thread
- Ovviamente i thread comportano però reali benefici solo se il programma realizzato può parallelizzare effettivamente il suo lavoro

I thread POSIX - pthread

Quando usarli

- Come anticipato, non è così ovvio che l'utilizzo dei thread nel proprio programma porti davvero benefici apprezzabili
  - Affinchè il loro utilizzo sia ragionevole occorre che le routine gestite dai singoli thread siano efficacemente parallelizzabili ovvero essere ad esempio:
    - Riordinabili
    - Parcellizzabili
    - Sovrapponibili



  - L'interdipendenza incrociata dei thread è invece un chiaro esempio di caso in cui i thread risulterebbero inefficaci

I thread POSIX - pthread

Quando usarli

- Ci sono alcuni contesti d'uso particolarmente standard che garantiscono ottimi risultati con i thread
  - Quando un determinato compito da svolgere su specifici dati può essere eseguito in contemporanea
  - Quando è possibile che alcune parti del programma possano restare bloccate per lungo tempo su operazioni di I/O
  - Quando la CPU è usata in modo pesante in alcune porzioni di codice ma molto meno in alter
  - Quando il sistema deve poter rispondere rapidamente a eventi fortemente asincroni
  - Quando alcune porzioni di programma hanno più importanza di alter (priorità degli interrupts)

I thread POSIX - `pthread`

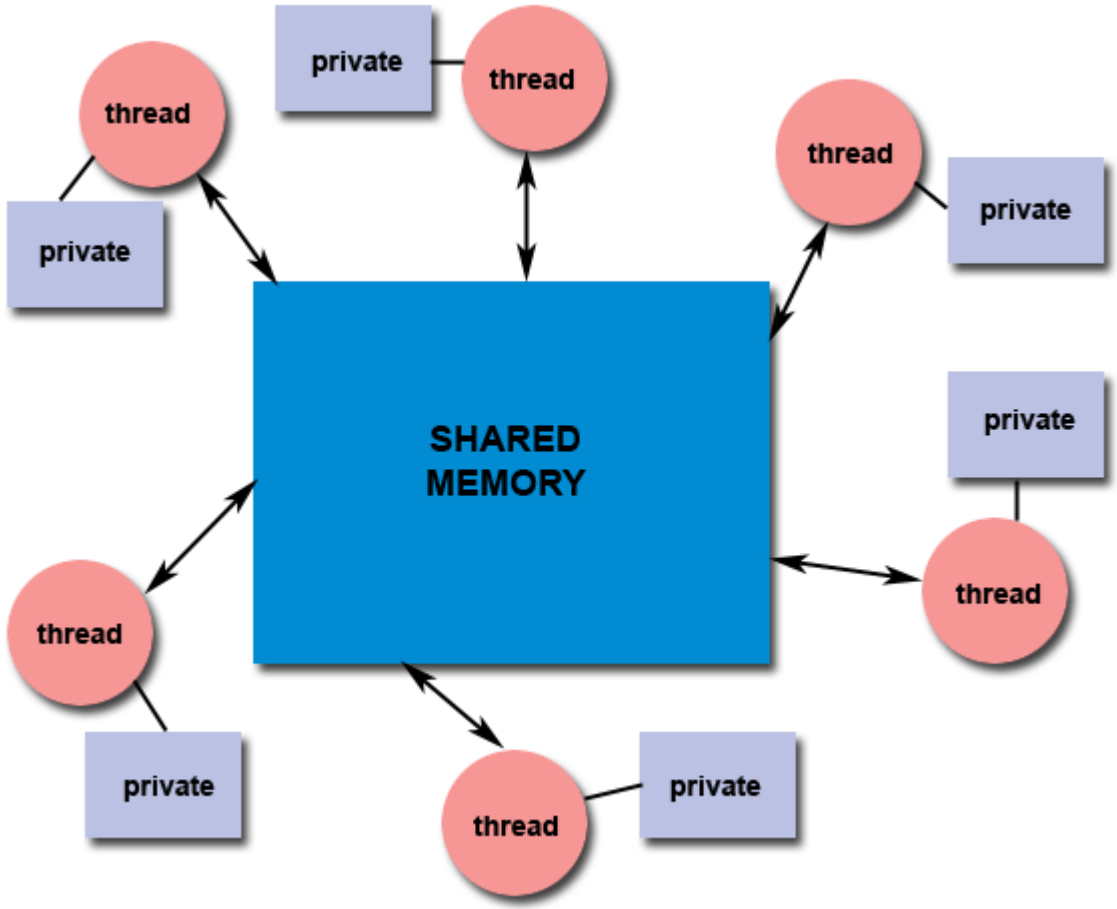
Casi tipici d'uso

- Ci sono alcune architetture tipiche di multi-programmazione che traggono generalmente ottimi benefici da un implementazione con thread
  - Manager/Worker
    - Si ha quando un unico thread (il Manger) distribuisce il lavoro a vari threads (workers)
    - Tipicamente il Manager gestisce gli input e parcellizza/ottimizza il lavoro per i workers
  - Pipeline
    - Si ha quando un lavoro complesso e lungo viene suddiviso in tanti sottolavori da svolgersi in serie ma in modo concorrente da diversi threads (come in una catena di montaggio)
  - Peer
    - Simile al Manager/Worker, con la differenza che il manager ha il compito iniziale privilegiato di creazione dei worker, ma poi partecipa in modo paritetico con gli altri all'esecuzione del lavoro

I thread POSIX - pthread

La condivisione della memoria

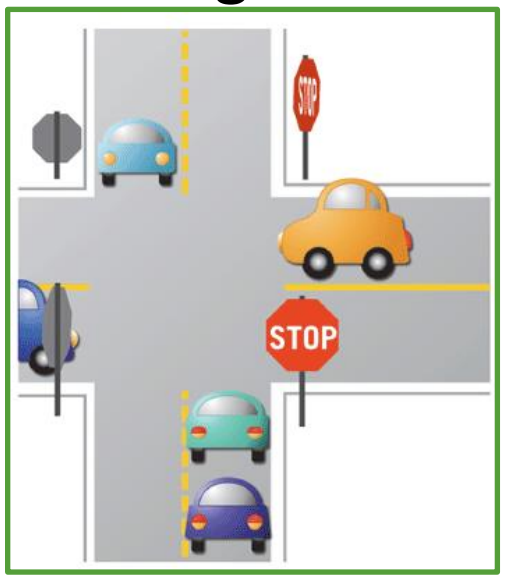
- Come ben evidenziato in figura tutti I thread condividono la memoria globale condivisa (shared memory)
- Ogni thread ha però anche una propria memoria per I dati private (non accessibile agli altri thread)
- E' compito del programmatore garantire opportuni meccanismi per la sincronizzazione degli accessi alla memoria condivisa!



I thread POSIX - pthread

La sicurezza

- La sicurezza con l'uso dei threads si riferisce essenzialmente alla capacità del programma di:
  - Evitare che un thread distrugga o inquini (*clobbering*) i dati sui cui lavora anche un altro thread
  - Evitare di innescare condizioni di instabilità logica (*Race Conditions*)
- Per questo motivo generalmente le funzioni o le librerie vengono dichiarate come:
  - *thread-safe*, quando possono garantire di non essere soggette ai problemi sopra citati)
  - *thread-unsafe*, quando la loro progettazione non ha tenuto conto delle criticità che possono emergere dal loro uso con i thread e quindi il loro uso nei thread espone automaticamente il programma a notevoli rischi



I thread POSIX - pthread

La gestione dei thread – La creazione


```
<pthread.h>
int pthread_create(pthread_t *restrict thread,
      const pthread_attr_t *restrict attr,
      void *(*start_routine)(void*), void *restrict arg)
```

- La funzione crea un nuovo thread all'interno del processo (thread) attuale, con gli attributi specificati in `attr` (se `NULL` usa i valori di default) e memorizzando l'identificatore nell'oggetto thread
  - L'entry-point del thread sarà la funzione `start_routine` (quando questa termina, il thread viene chiuso implicitamente)
  - `arg` rappresenta il parametro (unico) passato «per riferimento» alla funzione `start_routine` all'avvio (se non è necessario, si passa `NULL`)
  - Si presti attenzione al fatto che potenzialmente l'argomento `arg` può essere anche una struttura (quindi inglobare più dati in uno) e che i singoli thread potrebbero modificare il contenuto dell'argomento, che è condiviso con il chiamante e potenzialmente anche con gli altri thread
- Il valore di ritorno indica l'effettiva creazione del thread (`==0`) o l'indicazione di un errore che ha impedito la creazione del thread richiesto (il valore restituito ingloba il significato dell'errore occorso)

I thread POSIX - pthread

La gestione dei thread – La creazione

- I motivi principali per cui la funzione potrebbe fallire sono:
  - `EAGAIN` : indica che attualmente la richiesta non può essere portata a termine per il raggiungimento dei limiti di risorse imposti (ad esempio raggiunto il massimo numero di threads per processo); suggerisce però che l'invocazione potrebbe essere ritentata in seguito ed avere successo
  - `EINVAL` : i parametri `attr` non sono corretti
  - `EPERM` : non si dispone dei permessi minimi necessari per svolgere l'operazione (ad esempio a causa della scelta dei parametri di scheduling)
- Il primo thread (main) coincide con il processo stesso e non serve sia esplicitamente creato
- Si tenga presente che i `pthread` non mantengono la gerarchia e quindi una volta creati, tutti i thread sono paritetici (**peer**)
  - Questo significa che un non esistono di fatto annidamenti multipli (figli dei figli) nei thread

Nota: la creazione del nuovo thread implica contestualmente anche l'avvio dello stesso dal suo entry-point (altre implementazioni richiedono invece due step distinti: creazione ed avvio)

I thread POSIX - pthread

La gestione dei thread – L'auto-chiusura

LEARN

```
<pthread.h>
void pthread_exit(void *value_ptr)
```

- La funzione chiede la chiusura esplicita del thread stesso che l'ha invocata (auto-chiusura)
  - Per tutti i thread (escluso quello con entry-point il main) equivale alla chiusura implicita tramite ritorno della funzione entry-point
- Non ha un valore di ritorno poiché il thread interrompe di fatto l'esecuzione e quindi tale funzione non ritorna nel chiamante
  - Offre però all'ambiente chiamante (altri thread aggregati) lo stato di terminazione del thread, passato per riferimento `value_ptr`
- Non sono previsti messaggi di errore
- Andrebbe usata anche come ultima funzione del `main` al fine di mantenere il `main` stesso in «stallo» fino alla positiva chiusura di tutti i thread legati al processo (meglio evitare `return` e `exit`)

I thread POSIX - `pthread`

La gestione dei thread – La richiesta di chiusura esterna


```
<pthread.h>
int pthread_cancel(pthread_t thread)
```

- La funzione chiede la chiusura esplicita del thread specificato nell'identificatore thread
  - Si tratta di un modo con cui un thread può richiedere la chiusura di un altro thread associato allo stesso processo
  - I tempi in cui tale chiusura avvengono sono indipendenti dal momento dell'invocazione (i due eventi non sono ovviamente contemporanei e sono di fatto asincroni)
  - La richiesta di chiusura per il thread fa partire la procedura specifica di invocazione dei distruttori a cascata
- La funzione restituisce 0 se l'operazione è stata eseguita correttamente o un valore diverso da zero per indicare un'errore (ed il valore stesso ne racchiude il significato)
- *Non è in fine dei conti troppo diverso dai segnali nei processi*

I thread POSIX - pthread

Esempio di creazione di thread


```
01 #include <pthread.h>
02 #include <stdio.h>
03 #include <stdlib.h>
04
05 #define NUM_THREADS     5
06
07 void *StartRoutine(void *threadid) {
08   long tid;
09   tid = (long) threadid;
10   printf("Thread: io sono il thread n.%ld \n", tid);
11   pthread_exit(NULL);
12 }
13
14 int main(int argc, char **argv) {
15   pthread_t threads[NUM_THREADS];
16   int errorcode;
17   long index;
18   for(index=0; index<NUM_THREADS; index++) {
19      printf("Main: ora creo il thread n. %ld\n", index);
20      errorcode = pthread_create( &threads[index], NULL, StartRoutine, (void *) index);
21      if (errorcode) {
22        printf("Errore nella creazione di un nuovo thread (%d)\n", errorcode);
23        exit(EXIT_FAILURE);
24      }
25   }
26   pthread_exit(NULL);
27 }
```


I thread POSIX - `pthread`

La gestione dei thread – Join e Detach

- Secondo lo standard POSIX, i thread possono essere di tipo **joinable** o **detachable**
  - Sono modalità per mantenere sincronizzati (o no) dei thread
  - La scelta della modalità avviene tramite l'attributo `attr` in fase di creazione del thread (ma può essere fatta anche in seguito mediante modifica dell'attributo)
    - I thread *Joinable* (comportamento di default) permettono ad un thread di mettersi in attesa (bloccante) di un altro thread (che ad esempio sta facendo un lavoro che gli serve ma che deve essere completo prima di poter procedere)
    - I thread *Detachable* sono invece thread completamente autonomi per i quali non è prevista tale forma di «sudditanza» e dove quindi i meccanismi di sincronizzazione devono essere gestiti in altre forme (es. mutex)
    - E' possibile convertire un thread *Joinable* in *Detachable* ma non viceversa

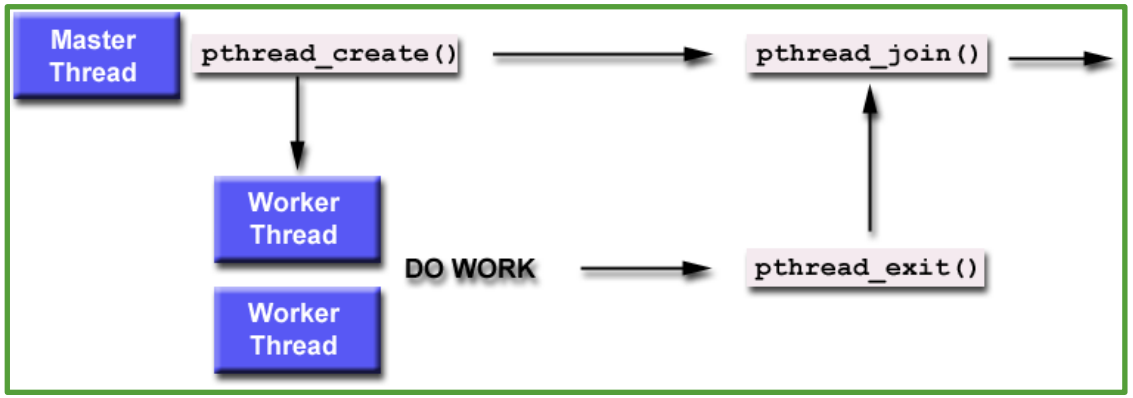
I thread POSIX - pthread

La gestione dei thread – Join e Detach

LEARN

```
<pthread.h>
int pthread_join(pthread_t thread, void **value_ptr)
int pthread_detach(pthread_t thread)
```

- Sono le due funzioni per agganciare un thread (chiamante) ad un altro (`thread`) o per sganciarsi
  - `value_ptr` serve per leggere lo stato si uscita del thread a cui si vuole agganciarsi
  - Entrambe restituiscono 0 se sono state completate correttamente o un valore diverso da 0 se c'è stato un'errore (nel qual caso il valore indica il tipo di errore)



I thread POSIX - pthread

La gestione dei thread – La dimensione dello stack

- Abbiamo visto che i thread non condividono lo stack (ovviamente) e quindi ognuno ha il suo particolare e univoco stack
  - Per un programmatore non è però facile sapere quanto grande sarà questo stack a disposizione e quindi nemmeno se sarà sufficiente per i propri scopi
  - Ci sono opportune funzioni di libreria per questo scopo:
    - `pthread_attr_getstacksize` : per conoscere la dimensione attuale dello stack
    - `pthread_attr_setstacksize` : per stabilire la dimensione del proprio stack (e per rendere così il codice maggiormente portabile, dipendendo meno dalle implementazioni)
    - `pthread_attr_getstackaddr` : per conoscere l'indirizzo dello stack attuale
    - `pthread_attr_setstackaddr` : per modificare l'indirizzo dello stack attuale (e spostarlo in un'altra regione di memoria)

I thread POSIX - `pthread`

La gestione dei thread – Altre funzioni interessanti

- Le funzioni principali per la gestione dei thread le abbiamo viste, ma ne esistono alcune altre che in certe situazioni possono essere d'interesse:
  - `pthread_self` : per ottenere l'identificativo univoco del thread che l'ha invocata
  - `pthread_equals` : per sapere se i due thread passati come argomenti rappresentano lo stesso thread oppure no
- *Nota: Una spiegazione più dettagliata sull'utilità di queste funzioni è complessa e prevede la conoscenza degli oggetti opachi (come lo sono appunto gli identificatori dei thread). Per questo motivo non viene trattata in questa sede*

I thread POSIX - `pthread`

Esercizi

› **`5-Guess-mathematical-operation-server-thread :`**

– Si scriva un programma in C che deve simulare il server (non-bloccante) dell'esercizio 10 degli appunti sulla programmazione di rete con i socket realizzandolo mediante thread (pthread).

Lo scambio dati (IPC)

Perché gli IPC

- Dopo aver visto come si creano/chiudono e gestiscono i processi ed i thread, occorre ora valutare come queste entità possano scambiarsi anche dei dati
  - Una minima forma di interazione l'abbiamo già vista in fase di creazione (con il passaggio di argomenti) e alla chiusura (con l'ottenimento del dato sullo stato di terminazione)
  - E' ovvio che però tale interazione è assolutamente inefficace ed insufficiene
  - Quello che occorre è una o più modalità con sui processi/thread possano scambiarsi continuamente dei dati (anche grosse moli), anche dopo la creazione e prima della chiusura
  - Si tratta infatti di esigenze frequentissime ed indispensabili (basti pensare alla mole di informazioni necessarie per gestire un'interfaccia grafica ed il motore di esecuzione a cui si associa)
- Queste tecniche vanno sotto il nome di sistemi di **IPC** (*Inter Process Communication*)

Lo scambio dati (IPC)

Quali IPC

- Per coerenza con lo standard visto finora, il POSIX, analizzeremo (brevemente) solo sistemi IPC riscontrabili effettivamente su sistemi POSIX
  - I socket
  - Le pipe anonime
  - Le fifo
  - I semafori (Mutex)
  - Le memorie condivise*
  - Le socketpair (o socket locali di tipo Unix)**
  - Le code di messaggi**
  - I segnali**
  - Le chiamate a procedura remote (RPC)**
- * *non saranno nel seguito approfonditi, ma solo descritti marginalmente*
- ** *non saranno descritti nel prosieguo dell'opera ma solo elencati per conoscenza*

Lo scambio dati (IPC)

I Socket

- Relativamente ai socket si è già discusso approfonditamente durante il modulo sulla programmazione di rete
  - Si sottolinea in questa sede come sia intuitivo due o più processi risiedenti sulla stessa macchina (o thread) possono interagire tra loro scambiandosi dati ed informazioni attraverso un «canale di rete virtuale» basato sui socket
  - Questo può essere fatto in diversi modi (*socket Unix*) ma se non si cercano caratteristiche eccessivamente avanzate o ben precise, può essere realizzato anche molto banalmente mediante la tipica architettura client/server su indirizzo di *loopback* (127.0.0.1) e porta a scelta non registrata (>1023)
  - Si tratta di un sistema relativamente semplice anche se un po' laborioso ed ha il vantaggio di essere enormemente portabile
  - Le prestazioni sono interessanti, pur non essendo probabilmente il sistema migliore per lo scambio di grandi quantitativi di dati in velocità

Lo scambio dati (IPC)

Le pipe anonime

- Molto meno oneroso in termini di overhead e di complicazioni di codice è l'uso delle *Pipe*
- Si tratta di un IPC nato essenzialmente sui sistemi Unix ma oggi ritrovabile anche su altri sistemi
- Possono essere pensati come una coppia di *file descriptor* che descrivono un «tubo» (o meglio due) attraverso il quale (i quali) due entità (processi o thread) possono scambiarsi rapidamente dei dati leggendo e scrivendo come farebbero su un file
  - Sono una coppia perché uno è aperto in sola lettura (per ricevere dati) e l'altro in sola scrittura (per inviare dei dati)
  - Non sono associati ad un vero file (su filesystem) ma solo ad buffer gestito dal kernel

Lo scambio dati (IPC)

Le pipe anonime

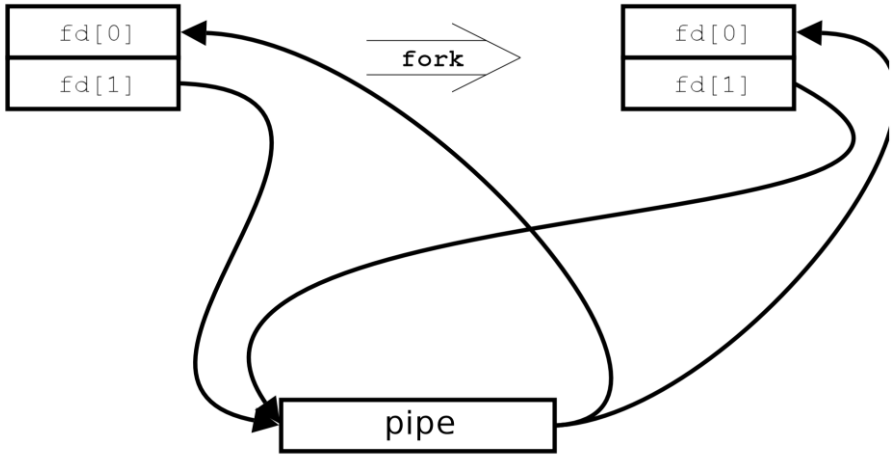
- Se è vero che si tratta di un sistema semplice, è anche vero che però hanno un prerequisito che ne limita il campo di utilizzo
  - Possono mettere in contatto solo entità (processi o thread) che abbiano una parentela tra loro (*siblings*) in quanto devono aver potuto condividere i descrittori dei file
    - Sono perciò facilmente usabili per mettere in contatto processi padri/figli (poiché i figli ereditano i descrittori già aperti dai padri) o thread associati ad un solo processo
  - Un'ulteriore limite è legato alla dimensione del buffer (`PIPE_BUF`)
    - Le implementazioni delle pipe garantiscono infatti che le operazioni di lettura e scrittura su pipe sono *atomiche* solo se i dati spediti/ricevuti sono inferiori alla dimensione del buffer
    - Se i dati da spedire/ricevere superano la dimensione del buffer, l'operazione è ancora possibile ma verrà suddivisa in più sotto-operazioni, delle quali il sistema non garantisce più però l'atomicità (con le conseguenze che vedremo più avanti)

Lo scambio dati (IPC)

Le pipe anonime

```
<unistd.h>
int pipe(int filedes[2])
```

- La funzione fa trovare nei due oggetti di `filedes` dopo la sua esecuzione i descrittori (*handler*) per i due canali (lettura e scrittura)
  - `filedes[0]` è il descrittore per la lettura o ascolto
  - `filedes[1]` è il descrittore per la scrittura o invio
- Il valore restituito è 0 in caso di successo e -1 in caso di fallimento (nel qual caso `errno` fornisce indicazioni sulle cause che hanno prodotto l'errore)



Lo scambio dati (IPC)

Le pipe anonime

- Nell'uso delle pipe si deve tenere ben presente un ulteriore caratteristica fastidiosa
  - Una pipe è descritta da due descrittori (uno per la lettura ed uno per la scrittura) ma in effetti rappresenta comunque un canale monodirezionale
  - Se si tenta infatti di usarla in modalità bidirezionale (un'unica pipe per comunicare A→B e B→A) si otterrebbe l'evidente problema per cui A rischierebbe di leggere i suoi stessi dati inviati e così farebbe anche B ed in aggiunta i dati letti per errore da A verrebbero rimossi dal buffer e quindi non giungerebbero mai a B (e viceversa)
  - La soluzione a ciò è quella di usare tipicamente una coppia di pipe (quindi 4 descrittori in tutto) in modo da realizzare due canali simplex dedicati tra A e B (uno per simulare A→B e uno per simulare B→A)
- Si tenga infine presente che la lettura di default è **bloccante** e quindi se non ci sono dati sul buffer la lettura non prosegue

Lo scambio dati (IPC)

Le pipe anonime - Esempio di codice

```
01 #include <stdio.h>
02 #include <sys/types.h>
03 #include <unistd.h>
04 #include <stdlib.h>
06 int child_routine(int*, int* );
07 int parent_routine(int* , int* );
09 int main(int argc, char** argv) {
10   int pipe_p2c[2], pipe_c2p[2], pid;
11
12   if ( pipe(pipe_p2c) <0 ) {  // Creo la pipe per i dati Parent->Child
13     perror("Errore, la funzione pipe P2C e' fallita");
14     exit(EXIT_FAILURE);
15   }
16   if ( pipe(pipe_c2p) <0 ) {  // Creo la pipe per i dati Child->Parent
17     perror("Errore, la funzione pipe C2P e' fallita");
18     exit(EXIT_FAILURE);
19   }
20     if ( (pid = fork() ) <0 ) {
21     perror("Errore nella creazione del processo figlio");
22     return -1;
23   }
24   if ( pid==0 ) {// Codice svolto dal processo figlio
25     return child_routine(pipe_c2p, pipe_p2c);
26   } else {  // Codice dal processo padre
27     return parent_routine(pipe_p2c, pipe_c2p);
28   }
29 }
```

Lo scambio dati (IPC)

Le pipe anonime - Esempio di codice

```
31 int child_routine(int pipe_write[], int pipe_read[]) {
32   unsigned char send_v, recv_v;
33   close(pipe_read[1]);  // Chiudo il capo in scrittura che non mi serve
34   for(send_v=1; send_v<10; send_v++)
35     write(pipe_write[1], &send_v, sizeof(send_v) );
36   if (close(pipe_write[1]) <0 ) {  // Chiudo la pipe per sbloccare la read
37     perror("Errore figlio, impossibile chiudere la pipe Write in scrittura");
38     exit(EXIT_FAILURE);
39   }
40   while ( read(pipe_read[0], &recv_v, sizeof(recv_v))!=0 )
41     printf("FIGLIO: Ricevuto dal padre: %d\n", recv_v);
42 }
43
44 int parent_routine(int pipe_write[], int pipe_read[]) {
45   unsigned char send_v, recv_v;
46   close(pipe_read[1]);  // Chiudo il capo in scrittura che non mi serve
47   for(send_v=100; send_v>90; send_v--)
48     write(pipe_write[1], &send_v, sizeof(send_v) );
49   if (close(pipe_write[1]) <0 ) {  // Chiudo la pipe per sbloccare la read
50     perror("Errore padre, impossibile chiudere la pipe Write in scrittura");
51     exit(EXIT_FAILURE);
52   }
53   while ( read(pipe_read[0], &recv_v, sizeof(recv_v))!=0 )
54     printf("PADRE: Ricevuto dal figlio: %d\n", recv_v);
55 }
```


Lo scambio dati (IPC)

Le FIFO o named pipe

- Le **FIFO** (sistemi di comunicazione *First In – First Out*) o **named pipe** (*pipe nominative*) sono a tutti gli effetti delle *pipe* che però hanno un limite in meno rispetto alle anonime già viste
  - Le FIFO possono essere usate anche per mettere in contatto processi non in correlazione filiare (quindi due processi nettamente distinti possono comunicare tra loro tramite una *named pipe*)
  - Si tratta di una funzionalità che espande enormemente il campo di applicazione delle pipe, rendendole di fatto molto più flessibili
  - Per risolvere il problema della conoscenza dei descrittori tra i due processi, ci si affida (anche se solo in forma virtuale) al file-system (in particolare ad un *inode*) sui cui si crea una struttura dati (chiamata appunto FIFO) a cui si agganceranno i due processi per creare le loro pipe
  - Di fatto il sistema continua ad usare un buffer nel kernel e a non usare quindi davvero il file-system per lo scambio dati (quindi è molto rapido)

Lo scambio dati (IPC)

Le FIFO o named pipe

- Come le pipe anonime, anche le FIFO:
  - Possono essere bloccanti oppure no
  - Possono essere aperte in lettura, scrittura o «entrambe»
  - Garantiscono l'atomicità delle operazioni solo per un numero di dati scambiati inferiori a `PIPE_BUF`
  - Possono essere usate anche per una comunicazione intra-processo (anche se l'utilità è dubbia e frequenti le possibili condizioni di stallo o *deadlock*)
- A differenza delle pipe anonime:
  - Occorre un primo step per la creazione dell'inode su cui poggia la FIFO
  - Alla chiusura della FIFO sarebbe opportuno rimuovere l'inode creato (non strettamente obbligatorio)
  - Di default l'operazione di apertura in scrittura (`open`) è bloccante (si sblocca solo con l'apertura in lettura dell'altro capo della fifo)

Lo scambio dati (IPC)

Le FIFO o named pipe

```
<sys/types.h>
<sys/stat.h>
int mkfifo(const char *pathname, mode_t mode)
```

- La funzione ha il compito di creare una FIFO associandola al path `pathname` e con le modalità `mode`
  - La funzione ritorna `0` se è stata eseguita correttamente e `-1` se c'è stato un'errore (nel qual caso `errno` contiene l'indicazione sulle cause dell'errore)
  - Le principali cause d'errore sono:
    - `EACCESS` : non si ha il diritto di attraversamento (*execute*) di una delle directory del percorso `pathname`
    - `EEXIST` : il percorso descritto in `pathname` corrisponde ad un elemento del FS già esistente (in certi contesti questo non va inteso come errore ma come segnalazione)

Lo scambio dati (IPC)

Le FIFO o named pipe

```
<unistd.h>
int unlink(const char *path)
```

La funzione ha il compito di rimuovere fisicamente una FIFO (ma più in generale un file/directory) dal filesystem

- L'operazione di rimozione non è contestuale all'invocazione poichè l'OS provvede realmente alla rimozione solo quando nessun processo usa più quell'oggetto (ma non può comunque essere più aperto)
- La funzione ritorna `0` se è stata eseguita correttamente e `-1` se c'è stato un'errore (nel qual caso `errno` contiene l'indicazione sulle cause dell'errore)
- Le principali cause d'errore sono:
  - `EACCESS` : non si ha il diritto di accedere al file per la sua rimozione
  - `EBUSY` : il file non può essere rimosso poichè in uso dal sistema o da un processo o si trova in una situazione di utilizzo

Nota: essendo una funzione che rimuove definitivamente oggetti dal filesystem deve essere eseguita con la massima accortezza

Lo scambio dati (IPC)

Le FIFO o named pipe - Esempio di codice

```c
#include <stdio.h>
#include <sys/types.h>  // Per mkfifo
#include <sys/stat.h>   // Per mkfifo
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>   // Per open, ...
#include <errno.h>   // Per errno, EEXIST, ...

#define FIFO_READ_PATHNAME "A2B.fifo"
#define FIFO_WRITE_PATHNAME "B2A.fifo"

int main(int argc, char** argv) {
  int fifo_read, fifo_write;
  unsigned char send_v, recv_v;
  // Creo la fifo di lettura con maschera 0622 ma non fallisce in caso di esistenza
  if ( mkfifo (FIFO_READ_PATHNAME , S_IRUSR | S_IWUSR | S_IWGRP | S_IWOTH)) {
    if (errno!=EEXIST) {
      perror ("Errore, impossibile creare la fifo per lettura");
      exit (EXIT_FAILURE);
    }
  }
  // Creo la fifo di scrittura con maschera 0622 ma non fallisce in caso di esistenza
  if ( mkfifo (FIFO_WRITE_PATHNAME , S_IRUSR | S_IWUSR | S_IWGRP | S_IWOTH)) {
    if (errno!=EEXIST) {
      perror ("Errore, impossibile creare la fifo per scrittura");
      exit (EXIT_FAILURE);
    }
  }
```

Lo scambio dati (IPC)

Le FIFO o named pipe - Esempio di codice

```
29   // la fifo per la scrittura in modalità sola-scrittura
30   if ((fifo_write = open ( FIFO_WRITE_PATHNAME , O_WRONLY )) <0) {
31     perror("Impossibile aprire in scrittura la fifo");
32     exit(EXIT_FAILURE);
33   }
34   // Apro la fifo per la lettura in modalità sola-lettura
35   if ((fifo_read = open ( FIFO_READ_PATHNAME , O_RDONLY )) <0) {
36     perror("Impossibile aprire in lettura la fifo");
37     exit(EXIT_FAILURE);
38   }
39   printf("Processo B: spedisco delle informazioni (100->90) al processo A\n");
40   for(send_v=100; send_v>90; send_v--)
41     write(fifo_write, &send_v, sizeof(send_v) );
42   close ( fifo_write );  // Chiudo la fifo in scrittura per evitare il block sulla read del processo A
43   printf("Processo B: ricevo delle informazioni dal processo A\n");
44   while ( read(fifo_read, &recv_v, sizeof(recv_v))!=0 )
45     printf("B: Ricevuto da A: %d\n", recv_v);
46  close ( fifo_read );
47  unlink(FIFO_READ_PATHNAME);
48  unlink(FIFO_WRITE_PATHNAME);
49 }
```

Nota: Il codice del processo B è praticamente identico, avendo l'accortezza di effettuare lo switch dei nomi delle due fifo, di invertire l'ordine di apertura (`open`) delle due fifo (pena una condizione di deadlock causata dalla modalità bloccante) e di modificare i dati di invio/ricezione e le diciture delle varie `printf`

Lo scambio dati (IPC)

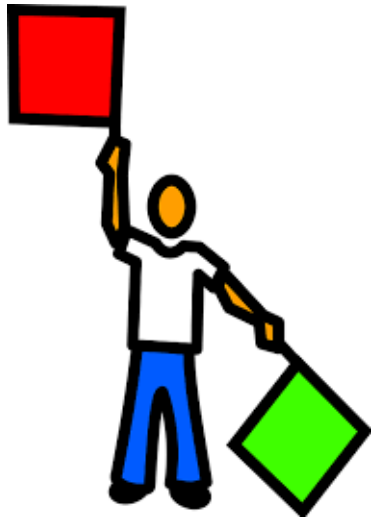
I semafori (Mutex)

- Si tratta di una modalità non per lo scambio di dati tra processi, ma piuttosto per la gestione e la sincronizzazione degli accessi ai dati mediante mutua esclusione (*MutEx*)
  - L'idea alla base di un semaforo è che il processo/thread che vuole inviare dei dati attraverso una modalità IPC debba prima controllare che tale risorsa sia libera controllando il rispettivo semaforo («verde»)
    - Se il processo/thread trova «libero» il semaforo, ne richiede immediatamente il cambio di stato per tutti gli altri processi/thread (che lo vedranno «rosso») e comincia l'invio dei dati in tutta tranquillità; al termine libera nuovamente il semaforo per tutti
    - Se il processo/thread trova invece «occupato» il semaforo (perché ad esempio c'è già qualche altro processo/thread che sta usando il sistema per inviare dati) rimane semplicemente in attesa del suo sblocco prima di procedere all'invio

  - Un'ottima analogia è quella di un lucchetto
    - Si lascia la chiave inserita quando è aperto (tutti possono accedere alla risorsa protetta)
    - Chi chiude asporta temporaneamente la chiave (così solo chi ha chiuso può riaprirlo)

Lo scambio dati (IPC)

I semafori (Mutex)



- In realtà le funzionalità dei MuTex sono molto banali e non svolgono nessuna operazione «magica» o straordinaria
  - Si limitano semplicemente a segnalare ai processi o thread la disponibilità o meno di una risorsa, ma non hanno nessun potere di bloccarla per davvero
    - Un semaforo rosso in un incrocio è un'indicazione chiara e imperativa per i guidatori per fermarsi, ma da solo non può certo impedire che materialmente qualche auto passi comunque anche con il rosso (con le conseguenze immaginabili)
  - I semafori sono quindi dei semplici mezzi per gestire in modo ragionevole un accordo (*gentlemen's agreement*) tra i processi/thread sull'accesso alla risorsa
    - E' però sempre e comunque compito del programmatore «guardare» ed «usare» i semafori per accedere alle risorse
    - Anche una sola porzione di codice «errata» in cui si tenta l'accesso alla risorsa senza osservare lo stato del mutex rende di fatto del tutto vano la sua implementazione!

Lo scambio dati (IPC)

I semafori (Mutex)

- La tipica sequenza d'uso dei semafori è:
  - Creazione e inizializzazione di un mutex
  - Vari threads (o processi) possono tentare di bloccare (chiudere/lock) il mutex
  - Solo un thread (o processo) riesce però ad ottenere il lock del mutex e ne diventa temporaneamente unico titolare
  - Il thread (o processo) che ha guadagnato il lock sul mutex può eseguire tutta la serie di operazioni che ritiene di dover eseguire in sicurezza (regione critica)
  - Il thread (o processo) può ora sbloccare (aprire/unlock) il mutex e rilasciarlo
  - Altri thread (o processi) possono ora ricompetere per guadagnare il lock sul mutex e quindi accedere alla risorsa protetta (ripartendo dal secondo punto)
  - Quando si ritiene che il semaforo non sia più necessario (perché ad esempio non c'è più competizione o la risorsa è sparita) si distrugge il mutex
- Analizzeremo di seguito brevemente solo i *mutex* dei *pthread* (POSIX) ma si tenga presente che le implementazioni possibili sono tante

Lo scambio dati (IPC)

I semafori (Mutex) – L'inizializzazione

LEARN

```
<pthread.h>
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr)
```

- E' la funzione da utilizzare per inizializzare il mutex `mutex` con le caratteristiche `attr` (il semaforo è inizialmente aperto/unlock)
  - Se `attr` è `NULL` si usano le caratteristiche di default (sufficienti per i nostri scopi
  - La funzione restituisce `0` in caso di successo ed un valore diverso in caso errore (con il significato associato)
  - Errori tipi che possono far fallire la funzione sono:
    - `EAGAIN` : quando attualmente il sistema non dispone delle risorse necessarie per inizializzare dinamicamente il mutex (ma potrebbe avere in seguito)
    - `EPERM` : quando il chiamante non dispone dei privilegi sufficienti per richiedere tale operazione
    - `ENOMEM` : quando il sistema non dispone di sufficiente memoria per inizializzare il mutex

Lo scambio dati (IPC)

I semafori (Mutex) – La distruzione


```
<pthread.h>
int pthread_mutex_destroy(pthread_mutex_t * mutex)
```

- E' la funzione da utilizzare per distruggere un mutex non più necessario
  - La funzione restituisce `0` in caso di successo ed un valore diverso in caso errore (con il significato associato)
  - Errori tipi che possono far fallire la funzione sono:
    - `EBUSY` : quando attualmente il mutex è chiuso (locked) o è comunque utilizzato in modo attivo da qualche altro thread/processo
    - `EINVAL` : quando il mutex passato come argomento non è valido

Lo scambio dati (IPC)

I semafori (Mutex) – Il lock


```
<pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

- E' la funzione con cui un processo o thread chiede il lock sul mutex `mutex`
  - Se il mutex risulta sbloccato (unlock) la funzione ha successo ed il mutex viene attribuito al processo/thread attuale e contestualmente viene chiuso (lock)
  - Se il mutex risulta invece bloccato da qualche altro processo/thread, la funzione si blocca fino all'ottenimento del lock
  - La funzione restituisce `0` se l'operazione ha avuto successo e un valore diverso da `0` in caso di fallimento (nel qual caso il valore restituito ingloba il significato delle cause del fallimento)
  - Alcune possibili cause d'errore sono:
    - `EINVAL`, `EAGAIN`, `EPERM` : significati già visti
    - `EDEADLK` : significa che il mutex è già chiuso e che il lock su tale mutex era già di proprietà del processo/thread attuale (e quindi non va considerato come vero errore)

Lo scambio dati (IPC)

I semafori (Mutex) – Il tentativo di lock


```
<pthread.h>
int pthread_mutex_trylock(pthread_mutex_t *mutex)
```

- E' la funzione con cui un processo o thread prova il lock sul mutex `mutex`
  - Se il mutex risulta sbloccato (unlock) la funzione ha successo ed il mutex viene attribuito al processo/thread attuale e contestualmente viene chiuso (lock)
  - Se il mutex risulta invece bloccato da qualche altro processo/thread, la funzione termina immediatamente restituendo l'errore (che non va inteso come errore) `EBUSY`
  - La funzione restituisce `0` se l'operazione ha avuto successo e un valore diverso da `0` in caso di fallimento (nel qual caso il valore restituito ingloba il significato delle cause del fallimento)
  - Altre cause d'errore sono analoghe alla funzione precedente

Lo scambio dati (IPC)

I semafori (Mutex) – L'unlock


```
<pthread.h>
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

- E' la funzione con cui un processo o thread rimuove il proprio lock sul mutex `mutex`
  - Se il mutex risulta bloccato (lock) dal processo/thread attuale, la funzione ha successo ed il mutex viene rilasciato dal processo/thread attuale e contestualmente viene aperto (unlock)
  - Se il mutex risulta invece bloccato da qualche altro processo/thread, la funzione termina immediatamente restituendo l'errore (che non va inteso come errore) `EPERM`
  - La funzione restituisce `0` se l'operazione ha avuto successo e un valore diverso da `0` in caso di fallimento (nel qual caso il valore restituito ingloba il significato delle cause del fallimento)
  - Alcune possibili cause d'errore sono:
    - `EINVAL`, `EAGAIN` : significati già visti
    - `EPERM` : quando il processo/thread attuale non possiede il lock sul mutex e quindi non ha il permesso di sbloccarlo

Lo scambio dati (IPC)

La memoria condivisa (shared memory)

- Si tratta di una modalità con cui i processi definiscono degli specifici segmenti di memoria della dimensione voluta e gestiti dal kernel attraverso il FS tmpfs
  - Ogni processo può aprire tale segmento (nominale su FS) in lettura, scrittura o entrambi
  - Più processi possono contemporaneamente usare in lettura, scrittura o entrambi ogni singolo segmento indipendentemente (permettendo così una comunicazione molti a molti)
  - Si tratta di un meccanismo tecnicamente molto efficiente e con basso overhead, oltre che molto semplice come interfaccia
  - Comporta però notevolissimi problemi di sincronizzazione degli accessi tra processi (e quindi possibilità di race conditions)

Lo scambio dati (IPC)

Esercizi

- **`6-Simple-pipe :`**
  - Si scriva un programma in C che richiede da riga di comando due valori interi con segno e li invia tramite una pipe ad un processo figlio che si deve occupare di svolgere tutti e quattro le operazioni possibili su quei dati e restituire i quattro risultati mediante pipe al primo processo. Al termine il processo figlio può terminare ed il processo padre deve stampare i risultati a video.
- **`7-Simple-fifo :`**
  - Si riscriva essenzialmente il programma precedente facendo uso delle fifo invece che delle pipe anonime e non usando clonazione (fork) ne nuove esecuzioni (exec) ma semplicemente gestendo ed invocando due programmi distinti da shell. A differenza del precedente però, si organizzi il protocollo di comunicazione tra il processo A (input/output) e B (calcolo) in modo che A invii la coppia di valori e l'indicazione dell'operazione e B risponda con il risultato numerico ed un indicazione testuale tipo 'O' o 'K' per indicare che il risultato è attendibile oppure no (controllo rappresentatività, divisione 0)

Lo scambio dati (IPC)

Esercizi

› 8-Client-Server-fifo

– Si riscriva un programma in C che gestisca mediante fifo la connessione bidirezionale tra due processi (il client ed il server). Non si deve fare uso di clonazione o esecuzione. Il client deve chiedere da shell all'utente il proprio nome Y con cui presentarsi poi al server. Una volta instaurata la connessione i due processi si scambiano semplicemente una stringa di saluto «Ciao, sono il server e tu sei il client n.X» e «Ciao, sono un client e mi chiamo Y» e poi il client può chiudersi ed il server rimanere in ascolto di altri client.
Nota: il programma è relativamente semplice nella sua implementazione ma richiede una notevole attenzione nelle modalità di gestione delle due fifo di comunicazione (una è semplice, ma l'altra meno…)

Analisi di alcune problematiche

Headhaches…

- La programmazione multipla ed in particolare i sistemi di IPC sono tematiche estremamente affascinanti ma anche fonte di terribili problematiche e mal di testa!
- Ci sono quantità di problemi che possono sorgere nello sviluppo di un codice multi-programmato e con sistemi di IPC
  - Race conditions (corse critiche)
  - Deadlocks (stallo)
  - Starvation (inedia)
  - …
- La cosa peggiore è che a causa della pseudo-imprevedibilità dei sistemi di scheduling dei processi, si tratta molto spesso di problemi non facilmente ricreabili e riproducibili ed il cui debug (prima ancora della risoluzione) può essere quindi assai difficoltoso!

Analisi di alcune problematiche

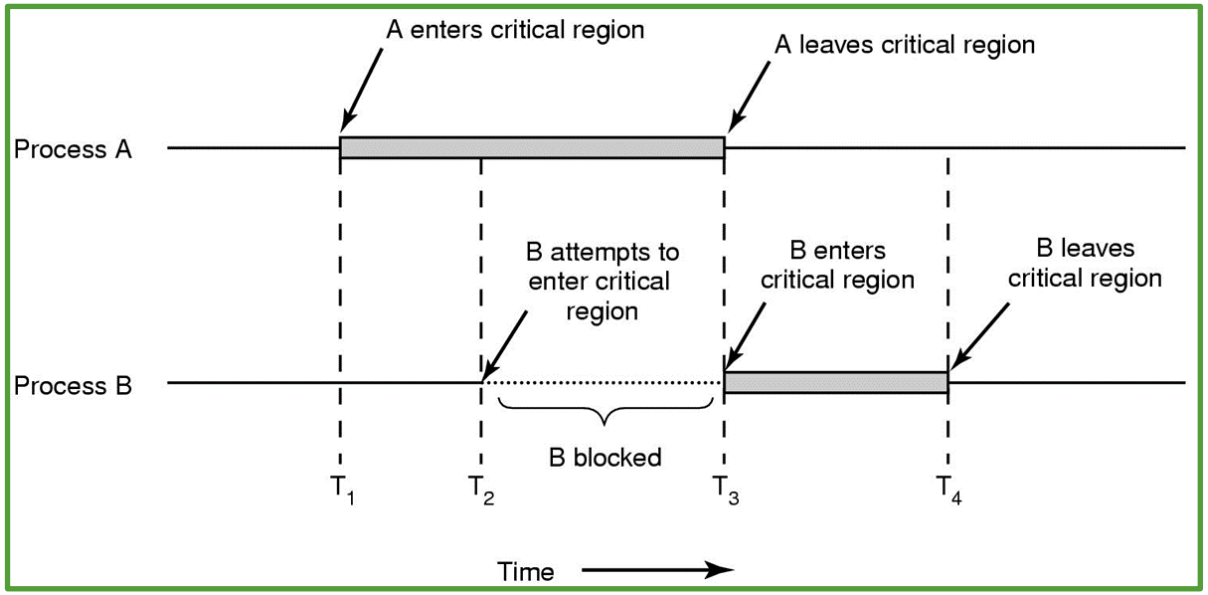
Race conditions - corse critiche

- Si tratta di problemi inerenti l'accesso concorrente (anche) in scrittura ad una risorsa da parte di più processi
  - Deriva principalmente da due fattori:
    - L'impossibilità di prevedere l'ordine con cui i processi accedono la risorsa
    - L'impossibilità di prevedere se il processo che ha ottenuto la risorsa riuscirà a completare la sua operazione prima che il tempo di esecuzione a suo disposizione termini
  - Posto che non ci sono soluzioni definitive preconfezionate per la determinazione e la risoluzione *delle race conditions*, è quantomeno disponibile una strategia che cerca di porvi rimedio alla fonte: l'uso delle **regioni critiche**
    - Si tratta praticamente di una strategia con cui si dichiarano le parti di codice dei processi che fanno uso di risorse condivise soggette a *race-conditions* all'interno delle cosiddette regioni o sezioni critiche, le quali hanno la peculiarità che non possono essere eseguite contemporaneamente

Analisi di alcune problematiche

Race conditions - corse critiche

- Alla base dell'effettiva efficacia di questo sistema ci sono 4 punti chiave che devono essere soddisfatti:
  - Due processi non devono mai trovarsi contemporaneamente nelle loro regioni critiche
  - Non può essere fatto nessun presupposto sulle velocità o sul numero delle CPU
  - Nessun processo in esecuzione al di fuori della sua regione critica può bloccare altri processi
  - Nessun processo dovrebbe restare in attesa di entrare nella sua regione critica per sempre
- Si tratta di un sistema di gestione delle mutue-esclusioni
- I modi di implementare le regioni critiche sono tanti e piuttosto complessi (busy waiting, sleep and wakeup, semafori/mutex, …)



Analisi di alcune problematiche

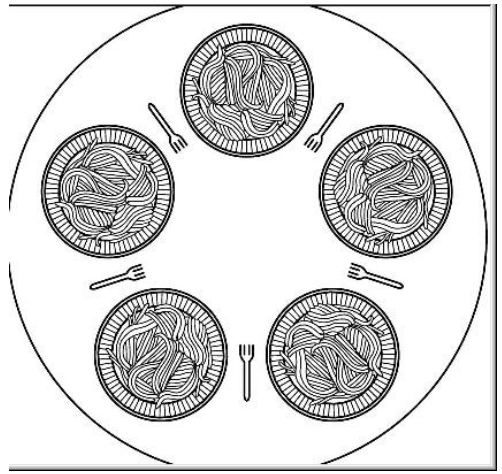
Deadlock - stallo

- Il *deadlock* o stallo è la situazione in cui due (o più) processi necessitano un'informazione o una risorsa dall'altro per poter procedere nell'esecuzione, di fatto restando bloccati su loro stessi in modo indefinito (o infinito)
  - Tipicamente è una situazione paradossale e che come tale non può avere soluzione
  - Quello che si può fare è impegnarsi a priori per evitare di cadere nelle condizioni che lo generano (migliorando la programmazione)
  - Ci sono delle condizioni ben precise (note come condizioni di Havender) che descrivono in modo razionale e scientifico la possibilità di incappare in deadlock e proprio su queste si può operare per cercare di evitare gli stalli

Analisi di alcune problematiche

Starvation - Inedia

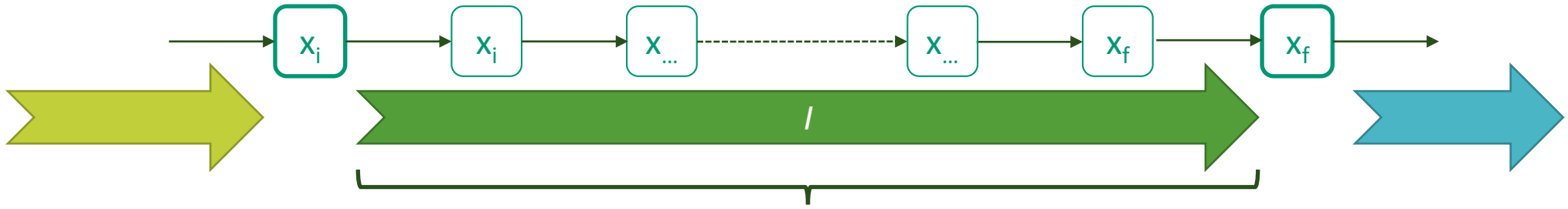
- E’ la condizione in cui un processo non riesce a progredire nell’esecuzione a causa della cronica assenza di risorse hardware e/o software (*es. problema dei cinque filosofi*)
  - E’ tipica in un sistema con scheduler inefficienti basati su priorità (in cui processi a bassa priorità rischiano di essere accantonati per un tempo indefinito)
  - Le soluzioni software non sono semplici poiché tipicamente è un accadimento legato in grande parte all’ambiente di esecuzione (OS)
    - Nel caso dello scheduler ottuso, una soluzione interessante è l’aggiunta della tecnica di *aging* dei processi, ovvero dell’invecchiamento controllato (più un processo rimane fermo in coda più invecchia e più aumenta priorità e quindi probabilità di uscirne)



Analisi di alcune problematiche

L'atomicità delle istruzioni

- Una caratteristica che permette generalmente di mitigare di molto le problematiche precedenti è quella di poter utilizzare istruzioni dichiarate come atomiche (o indivisibili)
  - Un'istruzione *I* che opera su un dato *x* si dice atomica o indivisibile se durante la sua esecuzione da parte di un processo P, il dato *x* risulta automaticamente inaccessibile per qualunque altro processo
  - Il vantaggio è che per tutto il tempo di esecuzione dell'istruzione *I*, il dato *x* può subire un'intera sequenza di variazioni tra uno stato iniziale x_i e uno finale x_f ma per gli altri processi potranno essere note (al momento giusto) solo gli estremi della sequenza (il dato x_i e il dato x_f appunto)



Bibliografia / sitografia

- «Processi e thread», appunti Sistemi Operativi, LIA Università degli Studi di Bologna
- «Reti di calcolatori», libro, A.S. Tanenbaum, D.J. Wetherall
- «I moderni Sistemi Operativi – 3° ed.», libro, A.S. Tanenbaum
- «GaPiL - Guida alla Programmazione in Linux», guida, Simone Piccardi
- «Programmazione di applicazioni di rete con socket -parte 1», appunti corso Sist. Distr., Valeria Cardellini
- «Programmazione dei socket di rete in GNU/Linux», appunti corso di Sistemi Informatici, Fulvio Ferroni
- «Controllo dei processi», Appunti corso Sistemi Università di Napoli, M. R. Guarracino
- «POSIX Threads Programming», https://computing.llnl.gov/tutorials/pthreads (ultimo accesso 20/03/2017)