



I THREAD

I thread e la libreria Pthreads

Luigi Coppolino, Luigi Romano

- W. R. Stevens, “UNIX Network Programming”, Prentice Hall Software Series, 1990
- "Using POSIX Threads: Programming with Pthreads", Brad Nichols, Dick Buttlar, Jackie Farrell, O'Reilly & Associates, Inc.
- "POSIX threads explained. A simple and nimble tool for memory sharing", D. Robbins (IBM developer site)

ROADMAP

- ❑ Cosa sono i thread
- ❑ Parallelismo potenziale
- ❑ Programmazione parallela vs programmazione sequenziale
- ❑ Sincronizzazione
- ❑ Identificazione
- ❑ Terminazione
- ❑ Threads vs processi
- ❑ Un ambiente di programmazione strutturata per i thread

COSA SONO I THREAD

□ Due mondi:

- User Mode
- Kernel Mode

□ Due tipi di thread:

- User threads
 - Astrazione di programmazione accessibile attraverso chiamate dall'interno di programmi
- Kernel threads
 - Astrazione per l'esecuzione dei thread da parte del SO

PERCHÉ USARE I THREAD

PARALLELISMO POTENZIALE

- L'esecuzione di un programma può essere spesso strutturata come un insieme di subtask con dei flussi di controllo autonomi
- Tali flussi possono vivere come processi autonomi o coesistere all'interno di un unico processo

- Miglioramento prestazionale (potenziale!)
- Aumento della complessità in termini di costo di sviluppo
 - Gestione della concorrenza
- Aumento della complessità in termini di costo di testing e debugging
 - Comparsa di errori difficilmente ripetibili (race conditions)

THREADS VS PROCESSI

I PROCESSI IN UNIX

- Ogni processo esegue un singolo programma ed è dotato di un singolo flusso di controllo:
 - Esiste un *program counter*, che indica la prossima istruzione da eseguire

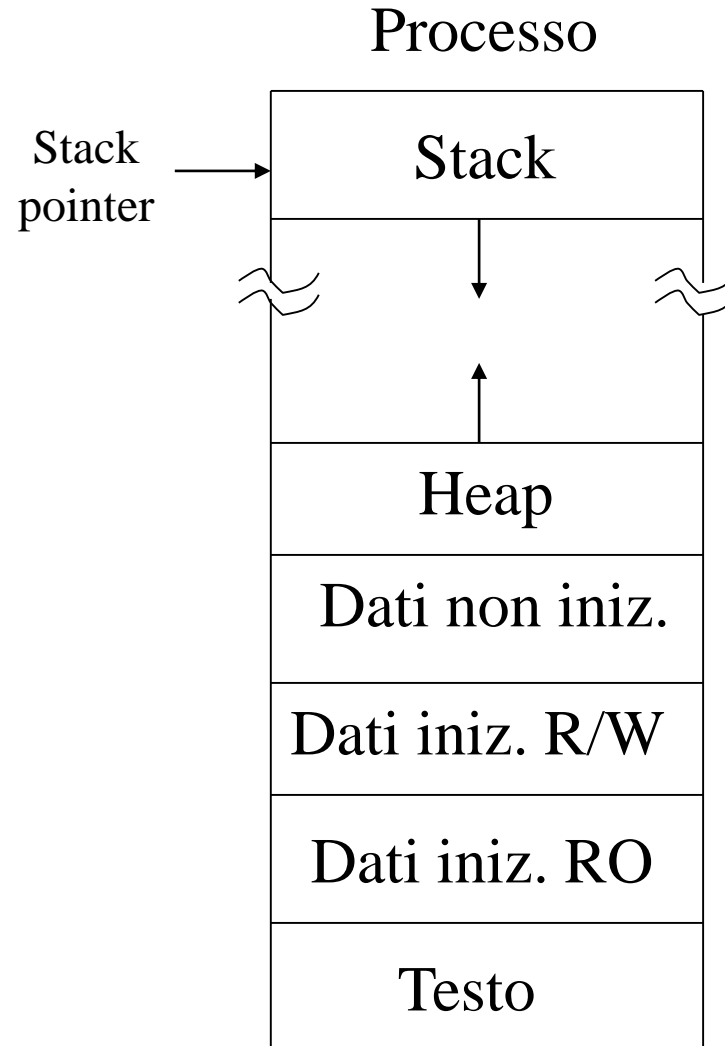
CONTESTO UTENTE DI UN PROCESSO – 1/3

- Porzioni dello spazio di indirizzi che sono accessibili al processo mentre viene eseguito nella modalità di utente:
 - *Testo*
 - Istruzioni macchina eseguite dall'hw
 - *Dati inizializzati a sola lettura*
 - Dati inizializzati dal programma, accessibili solo in lettura dal processo in esecuzione
 - *Dati inizializzati a R/W*
 - Come sopra, ma i valori possono essere anche modificati durante l'esecuzione del processo

CONTESTO UTENTE DI UN PROCESSO – 2/3

- *Dati non inizializzati*
 - Dati non inizializzati dal programma ma che sono azzerati prima che il processo cominci ad essere eseguito (modificabili dal processo in esecuzione)
- *Stack*
 - Contiene l'indirizzo di ritorno per ogni chiamata di funzione ed anche i dati richiesti da una funzione
- *Heap*
 - Utilizzato durante l'esecuzione del processo per allocare dinamicamente i dati

CONTESTO UTENTE DI UN PROCESSO – 3/3



CONTESTO KERNEL DI UN PROCESSO

- ❑ Accessibile solo al Kernel
- ❑ Informazioni necessarie al kernel per eseguire un processo:
 - Es: registri macchina associati al processo
- ❑ Non accessibile al processo mentre è in esecuzione

ESEMPIO

- Un semplice programma in un paradigma a processi (esecuzione sequenziale)

SIMPLE.C – CODICE 1/4

```
#include <stdio.h>
void do_one_thing(int *);
void do_another_thing(int *);
void do_wrap_up(int, int);
int r1 = 0, r2 = 0;
main(void)
{
    do_one_thing(&r1);

    do_another_thing(&r2);

    do_wrap_up(r1, r2);

    return 0;
}
```


SIMPLE.C – CODICE 2/4

```
void do_one_thing(int *pnum_times)
{
    int i, j, x;

    for (i = 0; i < 4; i++) {
        printf("doing one thing\n");
        for (j = 0; j < 10000; j++) x = x + i;
        (*pnum_times)++;
    }
}
```

SIMPLE.C – CODICE 3/4

```
void do_another_thing(int *pnum_times)
{
    int i, j, x;

    for (i = 0; i < 4; i++) {
        printf("doing another \n");
        for (j = 0; j < 10000; j++) x = x + i;
        (*pnum_times)++;
    }
}
```

SIMPLE.C – CODICE 4/4

```
void do_wrap_up(int one_times, int another_times)
{
    int total;

    total = one_times + another_times;
    printf("All done, one thing %d, another %d for a total of %d\n", one_times,
        another_times, total);
}
```

SIMPLE.C – COMPILAZIONE

□ Linea di comando:

- `gcc -c simple.c`
- `gcc -o simple simple.o`

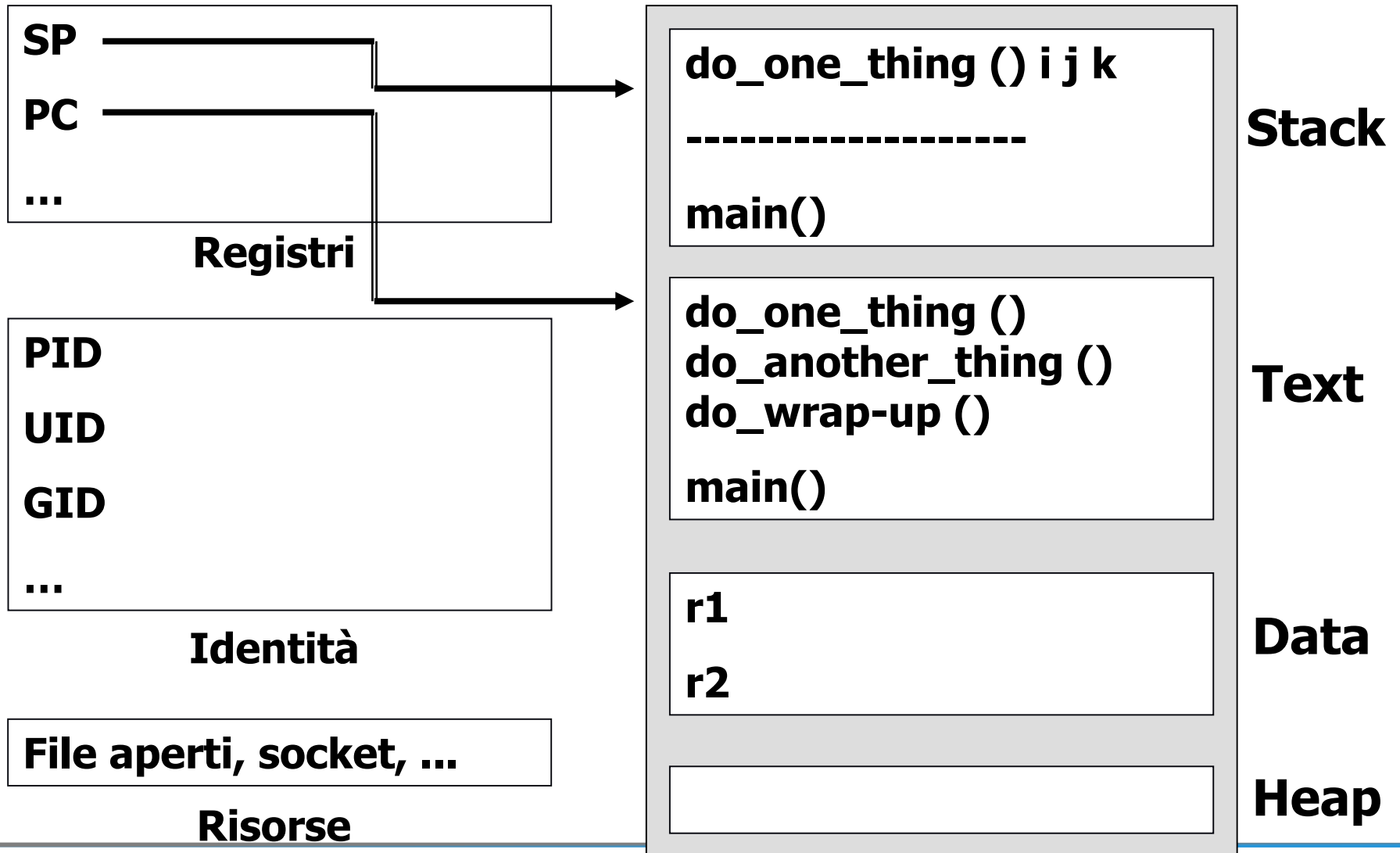
□ Makefile:

- `CCOMPILER = gcc -c`
- `CLINKER = gcc`

- `simple: simple.o`
`$(CLINKER) -o simple simple.o`

- `simple.o: simple.c`
`$(CCOMPILER) simple.c`

SIMPLE.C – MODELLO DI ESECUZIONE



ESEMPIO

- Un semplice programma in un paradigma a processi (esecuzione concorrente)

SIMPLE_PROCESSES.C – CODICE 1/2

```
int main(void)
{ ...
  if ((child1_pid = fork()) == 0) {
    /* first child */
    do_one_thing(r1p);
    return 0;
  } else if (child1_pid == -1) {
    perror("fork"), exit(1);
  }
  /* parent */
  if ((child2_pid = fork()) == 0) {
    /* second child */
    do_another_thing(r2p);
    return 0;
  } else if (child2_pid == -1) { perror("fork"), exit(1); }
```

SIMPLE_PROCESSES.C – CODICE 2/2

```
/* parent */  
if ((waitpid(child1_pid, &status, 0) == -1))  
    perror("waitpid"), exit(1);  
if ((waitpid(child2_pid, &status, 0) == -1))  
    perror("waitpid"), exit(1);  
  
do_wrap_up(*r1p, *r2p);  
  
return 0;  
}
```


SIMPLE_PROCESSES.C – COMPILAZIONE

□ Linea di comando:

- `gcc -c simple_processes.c`
- `gcc -o simple_processes simple_processes.o`

□ Makefile:

- `CCOMPILER = gcc -c`
- `CLINKER = gcc`

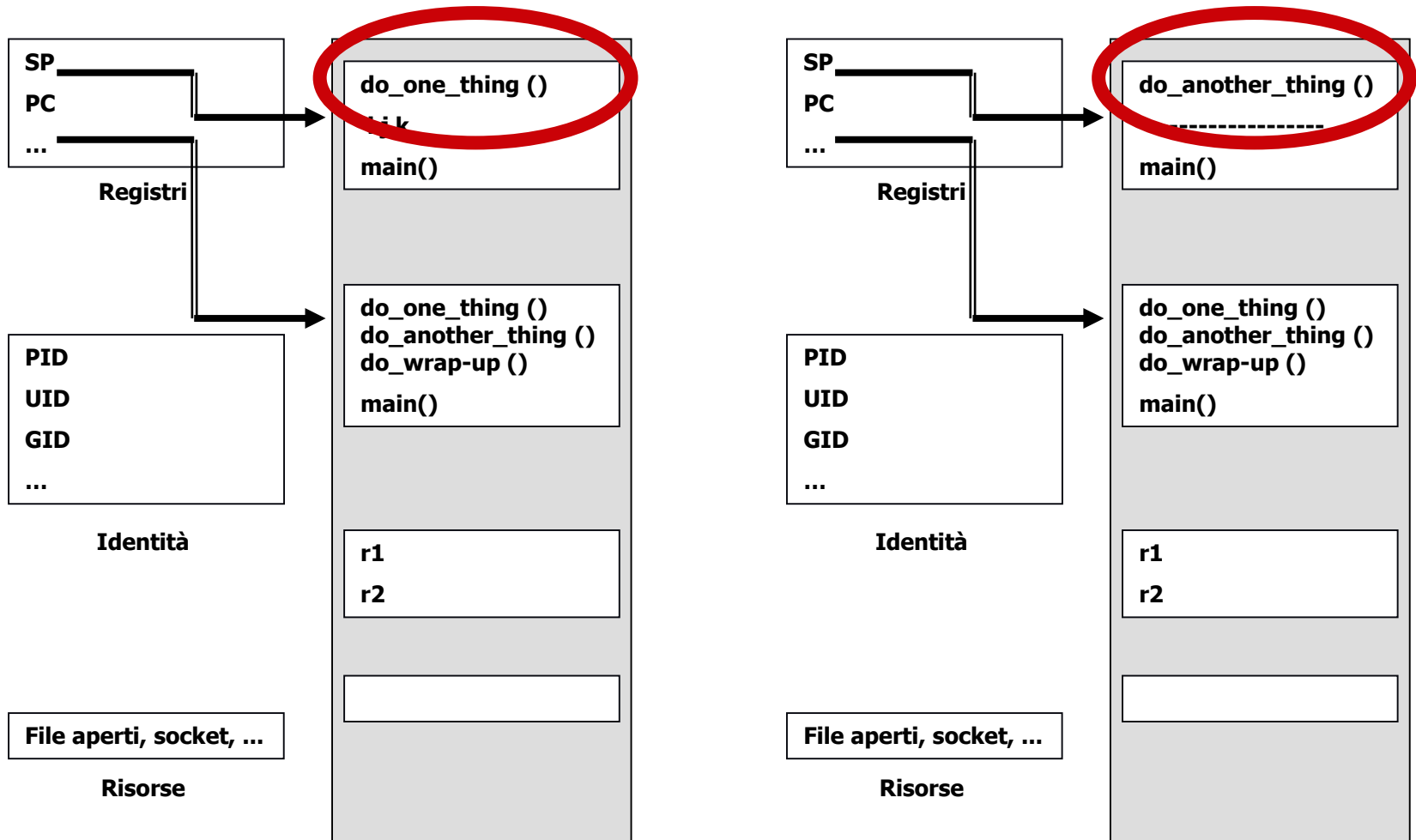
- `simple_processes: simple_processes.o`
`$(CLINKER) -o simple_processes simple_processes.o`

- `simple_processes.o: simple_processes.c`
`$(CCOMPILER) simple_processes.c`

ESERCITAZIONE

- Ricavare il modello di esecuzione dell'esempio precedente

SIMPLE_PROCESSES.C – MODELLO DI ESECUZIONE (DOPO FORK)



ESERCITAZIONE

- Compilare il programma `simple_processes.c` con il compilatore C++

SIMPLEPROCESSESDEMO.CPP – COMPILAZIONE

□ Linea di comando:

- `g++ -c SimpleProcessesDemo.cpp`
- `g++ -o SimpleProcessesDemo SimpleProcessesDemo.o`

□ Makefile:

- `CPPCOMPILER = g++ -c`
- `CPPLINKER = g++`

- `SimpleProcessesDemo: SimpleProcessesDemo.o`
`$(CPPLINKER) -o SimpleProcessesDemo SimpleProcessesDemo.o`

- `SimpleProcessesDemo.o: SimpleProcessesDemo.cpp`
`$(CPPCOMPILER) SimpleProcessesDemo.cpp`

ESERCITAZIONE

- Organizzare il programma SimpleProcessesDemo.cpp su più file

SOLUZIONE - SIMPLEPROCESSEDEMO MAIN.CPP

```
#include <stdlib.h>
```

```
...
```

```
#include "SimpleProcessesDemoFunctions.h"
```

```
int shared_mem_id;
```

```
int *shared_mem_ptr;
```

```
int *r1p;
```

```
int *r2p;
```

```
int main(void)
```

```
{
```

```
...
```

```
return 0;
```

```
}
```



SOLUZIONE - SIMPLEPROCESSESDEMOFUNCTIONS.H

```
extern int shared_mem_id;  
extern int *shared_mem_ptr;  
extern int *r1p;  
extern int *r2p;
```

```
#ifndef _SIMPLEPROCESSESDEMOFUNCTIONS  
#define _SIMPLEPROCESSESDEMOFUNCTIONS
```

```
void do_one_thing(int *);  
void do_another_thing(int *);  
void do_wrap_up(int, int);
```

```
#endif
```



SOLUZIONE - SIMPLEPROCESSESDEMOFUNCTIONS.CPP

```
#include "SimpleProcessesDemoFunctions.h"
```

```
void do_one_thing(int *pnum_times)
```

```
{
```

```
...
```

```
}
```

```
void do_another_thing(int *pnum_times)
```

```
{
```

```
...
```

```
}
```

```
void do_wrap_up(int one_times, int another_times)
```

```
{
```

```
...
```

```
}
```

SOLUZIONE - MAKEFILE

- SimpleProcessesDemoMultiple: SimpleProcessesDemoMain.o
SimpleProcessesDemoFunctions.o
\$(CPPLINKER) -o SimpleProcessesDemoMultiple
SimpleProcessesDemoMain.o SimpleProcessesDemoFunctions.o
- SimpleProcessesDemoMain.o: SimpleProcessesDemoMain.cpp
\$(CPPCOMPILER) SimpleProcessesDemoMain.cpp
- SimpleProcessesDemoFunctions.o:
SimpleProcessesDemoFunctions.cpp
\$(CPPCOMPILER) SimpleProcessesDemoFunctions.cpp

I THREAD IN UNIX

- Un unico processo
- Più flussi di esecuzione indipendenti
 - Più stati dell'esecuzione
 - Per ognuno esiste uno stack
 - Per ognuno esiste un *program counter*, che indica la prossima istruzione da eseguire

UN AMBIENTE DI PROGRAMMAZIONE STRUTTURATA PER I THREAD

I POSIX THREAD (PTHREAD)

- ❑ Esisteva la necessità di avere una libreria di programmazione per i thread che fosse indipendente dalla piattaforma e dal SO.
- ❑ IEEE ha accettato uno Standard per L'Information Technology Portable Operating System Interface (POSIX) Part 1: I Pthread (POSIX Section 1003.1c).
- ❑ Lo standard Pthread è una API (Application Programming Interface)

CREAZIONE

pthread_create (*&idthread, NULL, (void *) name_routine, (void *) ¶meter*)

- ❑ Un puntatore ad un buffer in cui viene restituito un valore che identifica il thread creato
- ❑ Un puntatore ad una struttura in cui vengono specificate le caratteristiche del nuovo thread (NULL caratteristiche di default)
- ❑ Un puntatore ad una routine da cui il thread inizierà l'esecuzione
- ❑ Un puntatore ad un parametro che deve essere passato alla routine quando il thread parte

SINCRONIZZAZIONE DEL FLUSSO DI ESECUZIONE

Pthread_join (idthread, NULL)

- pthread_join fornisce un meccanismo di sincronizzazione simili alla wait: sospende il thread finchè un altro thread termina
- La differenza con la wait, che agisce tra padre e figlio, la pthread_join può essere usata tra due thread qualsiasi in un programma

ESEMPIO

- Esecuzione concorrente di un semplice programma in un paradigma multi-thread

SIMPLE_THREADS_GLOBAL.C – CODICE

```
main(void)
{
    pthread_t    thread1, thread2;

    if (pthread_create(&thread1,
        NULL,
        (void *) do_one_thing,
        (void *) &r1) != 0)
        perror("pthread_create"), exit(1);

    if (pthread_create(&thread2,
        NULL,
        (void *) do_another_thing,
        (void *) &r2) != 0)
        perror("pthread_create"), exit(1);
```

SIMPLE_THREADS_GLOABAL.C – COMPILAZIONE

□ Linea di comando:

- `gcc -c simple_threads_global.c`
- `gcc -o -lpthread simple_threads_global simple_threads_global.o`

□ Makefile:

- `CCOMPILER = gcc -c`
- `CLINKER = gcc -lpthread`

- `simple_threads: simple_threads_global.o`
`$(CLINKER) -o simple_threads_global simple_threads_global.o`

- `simple_threads.o: simple_threads_global.c`
`$(CCOMPILER) simple_threads_global.c`

ESERCITAZIONE

- Ricavare il modello di esecuzione dell'esempio precedente

SIMPLE_THREADS_GLOBAL.C – MODELLO DI ESECUZIONE



SIMPLE_THREADS_LOCAL.C – CODICE

```
main(void)
{
    pthread_t    thread1, thread2;
    if (pthread_create(&thread1,
        NULL,
        (void *) do_one_thing,
        (void *) &r1) != 0)
        perror("pthread_create"), exit(1);

    if (pthread_create(&thread2,
        NULL,
        (void *) do_another_thing,
        (void *) &r2) != 0)
        perror("pthread_create"), exit(1);
```

CREATE_JOIN.C – CODICE

```
#include <stdio.h>

void *Test_function(void *arg){
    int i;

    for (i=0; i<10; i++){
        printf("\n\t Thread start !! %d",i);
        sleep(1);
    }

    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t MyThread;
    if (pthread_create(&MyThread, NULL,
        Test_function, NULL)){
        printf("\n\t Errore nella creazione del Thread.");
        abort();
    }
    if (pthread_join (MyThread, NULL)){
        printf("\n\t Errore nel joining del Thread.");
        abort();
    }
    else printf("\n\t Fine pthread_join\n");

    return 0;
}
```


ALTRI MECCANISMI DI SINCRONIZZAZIONE DEI PTHREAD

- Oltre alla `pthread_join` esistono:
 - `mutex variable`: attiva una mutua esclusione su di una zona critica (lock)
 - `condition variable`: fornisce un modo per rilevare eventi a cui il thread è interessato (es. attesa che un flag venga settato oppure un contatore incrementato)
 - `pthread_once` : garantisce che una routine di inizializzazione venga eseguita una ed una sola volta quando è invocata da più thread
 - Semaphore : Simile a mutex, ma sono associato adun contatore (solo su piattaforme con estensioni POSIX real-time (POSIX.1b))

VARIABILI MUTEX

VARIABILI MUTEX

- Forniscono accesso esclusivo ai dati (sezioni critiche).
- Usare le variabili mutex per i thread è semplice:
 1. Creare ed inizializzare mutex per ogni risorsa da proteggere
 2. `pthread_mutex_lock`: blocca le risorse (lock). La libreria Pthread assicura che in ogni istante un solo thread locchi la mutex
 3. `pthread_mutex_unlock`: sblocca le risorse (unlock)
- Se più thread sono in attesa di lock di una variabile mutex, quale thread accede prima?
 - Dipende dalla priorità di scheduling di ogni singolo thread. Il thread a priorità più alta effettua il lock
- Problema: priority inversion
 - Un thread a priorità più alta non può continuare fino a che un thread a priorità più bassa non rilascia il lock

ESEMPIO

- Un semplice programma con mutex

SIMPLE_MUTEX.C – CODICE

```
int r1 = 0, r2 = 0, r3 = 0;
pthread_mutex_t r3_mutex=PTHREAD_MUTEX_INITIALIZER;
```

```
int main(int argc, char **argv)
{
    pthread_t  thread1, thread2;

    if (argc > 1)
        r3 = atoi(argv[1]);

    if (pthread_create(&thread1,
        NULL,
        (void *) do_one_thing,
        (void *) &r1) != 0)
        perror("pthread_create"),exit(1);
```

...

```
...
if (pthread_create(&thread2,
    NULL,
    (void *) do_another_thing,
    (void *) &r2) != 0)
    perror("pthread_create"),exit(1);

if (pthread_join(thread1, NULL) != 0)
    perror("pthread_join"), exit(1);

if (pthread_join(thread2, NULL) != 0)
    perror("pthread_join"), exit(1);

do_wrap_up(r1, r2);

return 0;
}
```

ESERCITAZIONE

- Compilare il programma `simple_mutex.c` con il compilatore C++

SIMPLEMUTEXDEMO.CPP – CODICE

```
int r1 = 0, r2 = 0, r3 = 0;
pthread_mutex_t r3_mutex=PTHREAD_MUTEX_INITIALIZER;
```

```
int main(int argc, char **argv)
{
    pthread_t  thread1, thread2;

    if (argc > 1)
        r3 = atoi(argv[1]);

    if (pthread_create(&thread1,
        NULL,
        (void *) do_one_thing,
        (void *) &r1) != 0)
        perror("pthread_create"),exit(1);
```

...

...

```
if (pthread_create(&thread2,
    NULL,
    (void *) do_another_thing,
    (void *) &r2) != 0)
    perror("pthread_create"),exit(1);

if (pthread_join(thread1, NULL) != 0)
    perror("pthread_join"), exit(1);

if (pthread_join(thread2, NULL) != 0)
    perror("pthread_join"), exit(1);

do_wrap_up(r1, r2);

return 0;
}
```



IDENTIFICAZIONE

ID DEL PROCESSO (PID)

- Ogni processo ha un'unico identificatore di processo (Process Identifier: PID), intero tra 0 e 30000, assegnatogli dal kernel all'atto della sua creazione
- Un processo ottiene il suo pid attraverso la chiamata di sistema :
int getpid();

ID DEL PROCESSO GENITORE

- Ogni processo ha un processo genitore (eccetto il processo *init*) con il relativo pid
- Un processo ottiene il pid del padre attraverso la chiamata di sistema :
int getppid();

ESERCIZIO

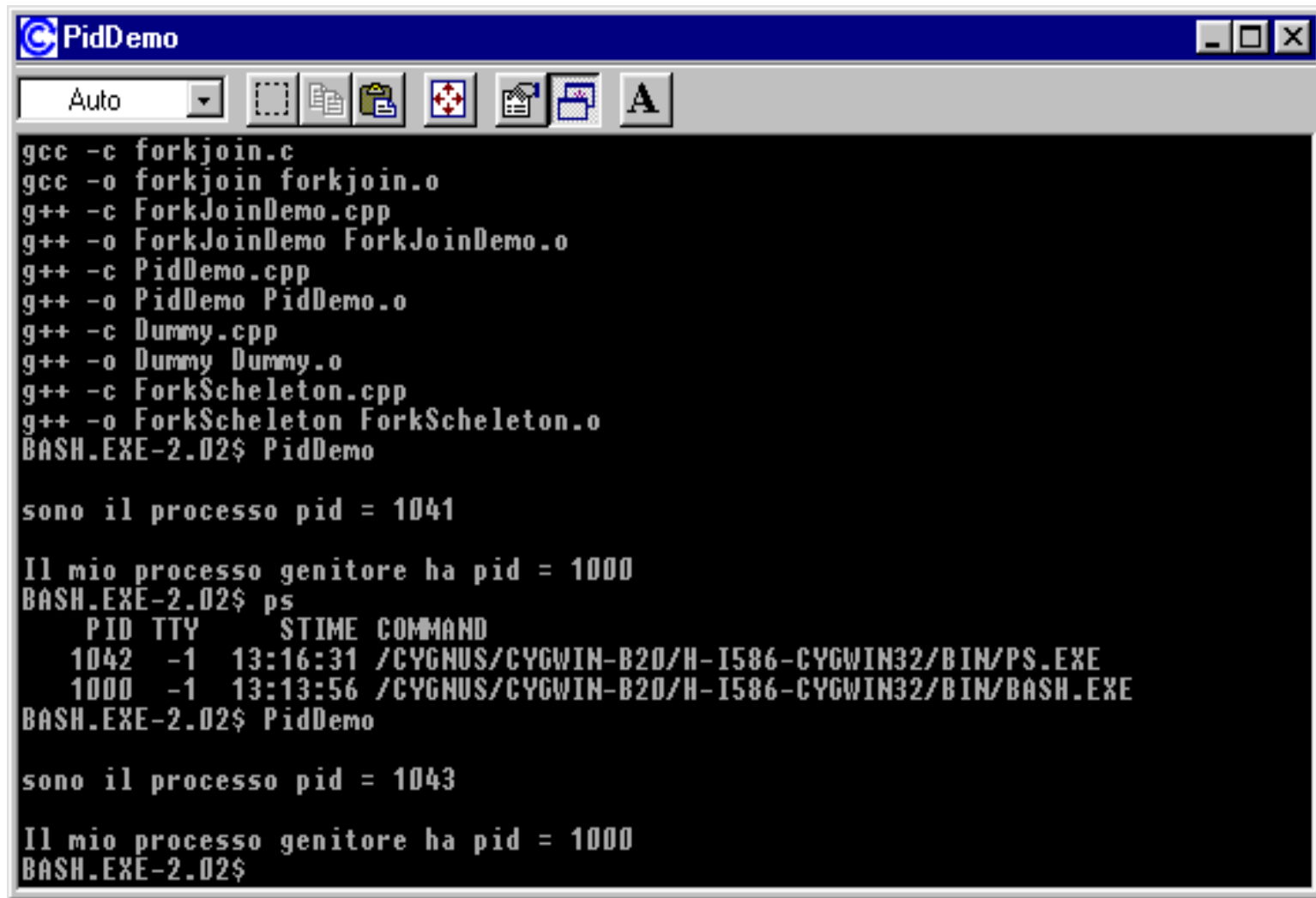
- Scrivere un semplice programma che utilizzi le chiamate di sistema *getpid()* e *getppid()*
- Nota: utilizzare
`#include <unistd.h>`

GETPID() E GETPPID() - CODICE

```
// File: PidDemo.cpp
# include <iostream.h>
# include <unistd.h>
int main(void)
{
    int pid,
        ppid;
    pid = getpid();
    cout << "\nsono il processo pid = " << pid << endl;
    ppid = getppid();
    cout << "\nIl mio processo genitore ha pid = " << ppid << endl;

    return 0;
}
```

GETPID() E GETPPID() – COMPILAZIONE ED ESECUZIONE



```
PidDemo
Auto
gcc -c forkjoin.c
gcc -o forkjoin forkjoin.o
g++ -c ForkJoinDemo.cpp
g++ -o ForkJoinDemo ForkJoinDemo.o
g++ -c PidDemo.cpp
g++ -o PidDemo PidDemo.o
g++ -c Dummy.cpp
g++ -o Dummy Dummy.o
g++ -c ForkSkeleton.cpp
g++ -o ForkSkeleton ForkSkeleton.o
BASH.EXE-2.02$ PidDemo

sono il processo pid = 1041

Il mio processo genitore ha pid = 1000
BASH.EXE-2.02$ ps
  PID TTY          STIME COMMAND
  1042 -1   13:16:31 /CYGNUS/CYGWIN-B20/H-I586-CYGWIN32/BIN/PS.EXE
  1000 -1   13:13:56 /CYGNUS/CYGWIN-B20/H-I586-CYGWIN32/BIN/BASH.EXE
BASH.EXE-2.02$ PidDemo

sono il processo pid = 1043

Il mio processo genitore ha pid = 1000
BASH.EXE-2.02$
```

PTHREAD_SELF, PTHREAD_EQUAL

- `pthread_self` : ritorna l'identificatore del thread
- `pthread_equal`: compara due identificatori di thread

.....

```
pthread_t io_thread, thread;
```

.....

```
thread = pthread_self();
```

.....

```
if (pthread_equal(io_thread,thread)) ....
```

.....

RELAZIONI DI PARENTELA

- Tra il processo padre e il processo figlio creato con una fork, esiste una relazione (es. la system call wait permette al padre di sincronizzarsi con la fine del processo figlio)
- In un programma multithread non esiste nessuna relazione tra i thread: un thread può creare un altro thread, ma i due hanno le stesse proprietà
- Chi finisce prima ? Dipende dalla politica di scheduling utilizzata.

ESEMPIO

- Un semplice programma

TIDENT.C – CODICE

```
#include <stdio.h>
#include <pthread.h>
pthread_t io_thread;
void * io_routine(void *notused)
{
    pthread_t thread;

    thread = pthread_self();
    if (pthread_equal(io_thread, thread))
        printf("hey it is me!\n");
    else
        printf("thats\' not me!\n");
    return(NULL);
}
```

```
int main()
{
    extern pthread_t io_thread;

    pthread_create(&io_thread,
        NULL,
        io_routine,
        NULL);

    pthread_join(io_thread, NULL);

    return 0;
}
```

TERMINAZIONE

ESEMPIO

- Un semplice programma

EXIT_STATUS_ALTERNATIVE.C – CODICE 1/2

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <pthread.h>
pthread_t thread;
static int arg;
static const int real_bad_error = -12;
static const int normal_error = -10;
static const int success = 1;
void * routine_x(void *arg_in)
{
    int *arg = (int *)arg_in;
    if (*arg == 1) {
        pthread_exit((void
            *)real_bad_error);
    } else if (*arg == 2) ...
```

```
... {
    return ((void *)normal_error);
} else {
    return ((void *)success);
}
}
extern int main(int argc, char **argv)
{
    extern int arg;
    pthread_t thread;
    void *status;
    if (argc < 2)
        printf("usage: %s [1-3]\n", argv[0]), exit(1);
    ...
```

EXIT_STATUS_ALTERNATIVE.C – CODICE 2/2

```
arg = atoi(argv[1]);
```

```
pthread_create(&thread,  
              NULL,  
              routine_x ,  
              &arg);
```

```
pthread_join(thread, &status);  
if (status == PTHREAD_CANCELED) {  
    printf("Thread was canceled.\n");  
} else {  
    printf("Thread completed and exit status is %ld.\n", (int)status);  
}  
return 0;  
}
```

ALTRI MECCANISMI DI SINCRONIZZAZIONE DEI PTHREAD

Oltre alla `pthread_join` esistono:

- `mutex variable`: attiva una mutua esclusione su di una zona critica (lock)
- `condition variable`: fornisce un modo per rilevare eventi di cui il thread è interessato (es. attesa che un flag venga settato oppure un contatore incrementato)
- `pthread_once` : garantisce che una routine di inizializzazione venga eseguita una ed una sola volta quando è invocata da più thread
- Semaphore : Simile a mutex, ma sono associato ad un contatore (solo su piattaforme con estensioni POSIX real-time (POSIX.1b))

CONDITION VARIABLES

CONDITION VARIABLES

Una Condition Variable permette ad un thread di sincronizzarsi su un valore di un dato.

Dopo averla inizializzata un thread può utilizzare una Conditional Variable in due modi:

1. Può mettersi in attesa su una Condition Variable
 - a) `pthread_cond_wait`: si sospende fino a che un altro thread effettua una signal sulla stessa variabile
 - b) `pthread_cond_timedwait` : permette di specificare un timeout argument. Se la condizione non è signaled in uno specifico istante, il thread è lasciato nello stato di attesa (wait)
2. Può effettuare una signal per un thread in attesa su di una Condition Variable:
 - a) `pthread_cond_sign`
 - b) `pthread_cond_broadcast`

CONDITION VARIABLES

- Prima di mettersi in attesa sulla condition variable, viene effettuata una wait sulla mutex perché il contatore è un dato condiviso
- Se la variabile count non ha raggiunto il valore desiderato, il thread (wait_thread) effettua una wait e rilascia l'area critica, così gli altri thread possono modificare la variabile count
- Se il wait_thread non rilasciasse l'area critica si verificherebbe un deadlock (nessun altro thread può entrare nell'area critica)

SBLOCCO

- Subito dopo una signal, il thread deve effettuare una `mutex_unlock`, così il wait thread può entrare nella zona critica (`mutex_lock`).
- Se più thread sono in attesa sulla stessa `C_V`, quale viene sbloccata prima a seguito di una signal ?
 - Chi ha priorità più alta.
 - Se i thread in attesa hanno la stessa priorità, l'ordine di rilascio è di tipo first-in first-out (F.I.F.O).

LA CHIAMATA PTHREAD_ONCE

IL MECCANISMO PTHREAD_ONCE

- Quando vengono creati più thread che cooperano per un unico task, spesso è necessario eseguire una singola operazione al fine di far procedere tali thread.
- Per esempio inizializzare dati, variabili mutex oppure variabili conditional)
- Ogni thread che vuole effettuare l'inizializzazione, invoca la pthread_once, ma solo il primo effettivamente effettua l'inizializzazione

TERMINAZIONE

TERMINAZIONE DI UN THREAD

- Un thread termina la sua routine o il processo in cui è eseguito termina
- `pthread_exit` : termina esplicitamente il thread
- `pthread_cancel` : termina un altro thread
- Siccome tra i thread non esiste una relazione padre-figlio, ogni thread può cancellare un altro thread
- Quando viene terminato un thread, bisogna assicurarsi che:
 - rilasci i lock (evitare situazioni di deadlock);
 - i dati siano lasciati in uno stato consistente.
- Affinché siano garantiti questi principi un thread deve modificare dinamicamente le sue proprietà di cancellabilità

TERMINAZIONE DI UN THREAD

- `pthread_setcancelstate` per disabilitare la cancellabilità del thread
- Punti di cancellazione automatici:
 - `pthread_cond_wait`, `pthread_cond_timedwait` e `pthread_join`;
- Punti di cancellazione programmabili:
 - `pthread_testcancel` abilita la cancellazione pendente nel punto in cui tale funzione viene inserita (nulla accade se non esiste una cancellazione pendente). Tali chiamate vengono inserite nei punti in cui lo stato del thread è considerato sicuro.

THREAD JOINABLE

- Se un thread termina, le sue risorse vengono prelazionate ed il suo stato salvato (thread joinable).
- Lo stato del thread terminato può essere restituito in due modi:
 - Se il thread termina esplicitamente, è possibile restituire lo stato attraverso l'argomento della funzione `thread_exit`
 - Attraverso il valore di ritorno delle routine del thread (`pthread_create` restituisce un tipo `(void *)`)

DETACHED THREAD

- Se non viene esplicitamente espressa la necessità della restituzione di uno stato al termine del thread, il suo stato viene conservato indefinitivamente (stato di zombie)
- Per un thread dichiarato detached la Pthread library non preserva lo stato di uscita di tali thread, liberando immediatamente le risorse occupate
- Un thread dichiarato detached informa la Pthread library che nessun thread può usare una `thread_join` per sincronizzarsi con la fine del thread detached
 - `PTHREAD_CREATE_DETACHED`: per thread detached
 - `PTHREAD_CREATE_JOINABLE`: per thread joinable