Lezione 5:

Concorrenza

Sistemi Operativi

Laboratorio – linea 2

5

- Concorrenza: *run together & compete*
- Un processo non è piú un programma in esecuzione che può essere considerato in isolamento
- Non determinismo: il sistema nel suo complesso ($P_1 + P_2$+Scheduler) rimane deterministico, ma se si ignora lo scheduler le esecuzioni di P_1 e P_2 possono combinarsi in molto modi, con output del tutto differenti
- Sincronizzazione: si usano meccanismi (Peterson, TSL, semafori, monitor, message passing, . . .) per imporre la combinazione voluta di P_1 e P_2

Processi senza memoria condivisa : clone vs fork 1

clone() crea un nuovo processo in maniera simile a *fork()* . A differenza di *fork()*, *clone()* permette al processo figlio di condividere **PARTE** del suo contesto di esecuzione con il processo chiamante (parent process). E' possibile scegliere **cosa** condividere, ad es. memoria, tabella dei descrittori dei file, tabella dei gestori dei segnali.

L'utilizzo principale di *clone()* è quello di implementare I thread, consentendo l'esistenza, all'interno del programma, di vari thread che eseguono in modo concorrente all'interno di uno spazio di memoria condiviso.

Quando un processo figlio è creato mediante *clone()* **esegue una funzione fn(arg)**. Questo è **diverso** da quello che succede quando usiamo fork() infatti, usando fork(), l'esecuzione continua **nel figlio dal punto in cui è stata chiamata fork()**. L'argomento **fn** è un **puntatore** a una funzione che viene chiamata dal figlio all'inizio della sua esecuzione. L'argomento **arg** è passato alla funzione fn.

Quando la funzione fn(arg) ritorna, il processo figlio termina. L'intero ritornato da fn è l'exit code per il processo figlio. Il processo figlio può anche terminare esplicitamente chiamando exit() o dopo aver ricevuto un segnale di terminazione che non può ignorare.

Processi senza memoria condivisa : clone vs fork 2

L'argomento child_stack specifica l'indirizzo dello stack usato dal processo clonato. Dato che il processo clonato (child) ed il chiamante di clone() (parent) possono condividere la memoria (virtuale) non è possibile per child eseguire utilizzando lo stesso stack del processo parent. Il processo che chiama clone(), quindi, **DEVE** occuparsi di creare uno stack per ogni processo clonato. Lo stack **CRESCE VERSO IL BASSO**. Quindi child_stack punta all'indirizzo più **ALTO** dello spazio creato dal processo parent prima di chiamare clone().

Il byte basso dell'argomento **flags** della funzione clone() contiene il numero del segnale di terminazione inviato a parent quando child termina. Se il segnale specificato è diverso da **SIGCHLD** parent **DEVE** specificare una delle opzioni __WALL (aspetta tutti) o __WCLONE (aspetta I cloni) quando chiama waitpid(). Se nessun segnale è specificato, a parent non arriva nessuna notifica quando I processi clonati terminano.

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg, … )
```

Un esempio di utilizzo di clone scritto da Linus Torvalds:
http://www.tldp.org/FAQ/Threads-FAQ/clone.c

Processi isolati : mti.c

Nel prossimo esempio che vedremo:

- C'è una variabile globale (int n).
- Usiamo **clone()** per clonare il main (parent) process.
- **NON** chiediamo a clone() di condividere la memoria virtuale → I processi sono **ISOLATI**
- I processi clonati (2) lavorano sulla loro **copia** della variabile globale (entrambi la incrementano)
- Dopo che I processi clonati (I child) terminano chiediamo a parent di stampare la var. globale.
- Dato che, a differenza dei processi clonati il processo che chiama clone() **NON** incrementa la variabile n, al momento della stampa il suo valore è sempre uguale a quello iniziale

Sistemi Operativi

Laboratorio – linea 2

5

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <signal.h>
#include <linux/sched.h>
#include <string.h>
#define _GNU_SOURCE
#define STACK_SIZE 65536
#define BUFSIZE 200
#define MAXN 1000000

int n = 0;
int Child(void *);

int main() {
    pid_t pidfirst;
    pid_t pidsecond;
    char *stackf;
    char *stacks;
    stackf = malloc(STACK_SIZE);
    stacks = malloc(STACK_SIZE);

    pidfirst =  clone(Child,stackf + STACK_SIZE, NULL);
    pidsecond = clone(Child,stacks + STACK_SIZE, NULL);
    waitpid(-1, NULL, __WALL);
    char buf[BUFSIZE];
    sprintf(buf,"Back to parent: Value of n: %d\n",n);
    write(1,buf,strlen(buf));
    return 0;
}

int Child(void *args) {
      int j = 0;
      for(j=0;j<MAXN;j++){
        n += 1;
      }
}
```

Processi
(**senza** memoria condivisa)
mti.c

```
$ gcc mti.c -o mti
$ ./mti
Back to parent: Value of n: 0
$ ./mti
Back to parent: Value of n: 0
$ ./mti
Back to parent: Value of n: 0
$ ./mti
Back to parent: Value of n: 0
```

Processi a memoria condivisa : mts.c

Nel prossimo esempio che vedremo:

- C'è una variabile globale (int n).
- Usiamo **clone()** per clonare il main (parent) process.
- **CHIEDIAMO** a clone() di **condividere** la memoria virtuale → I processi **non** sono ISOLATI
- I processi clonati (2) lavorano sulla variabile globale condivisa (entrambi **competono** per incrementarla)
- Dopo che I processi clonati (I child) terminano chiediamo a parent di stampare la var. globale.
- Secondo voi quale è il valore della variabile globale al momento della stampa da parte di parent?

Sistemi Operativi

Laboratorio – linea 2

5

Processi
(**con** memoria condivisa)
mts.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <signal.h>
#include <linux/sched.h>
#include <string.h>
#define _GNU_SOURCE
#define STACK_SIZE 65536
#define BUFSIZE 200
#define MAXN 1000000

int n = 0;      // variabile globale! possibili interferenze!
int Child(void *);

int main() {
    pid_t pidfirst;
    pid_t pidsecond;
    char *stackf;
    char *stacks;
    stackf = malloc(STACK_SIZE);
    stacks = malloc(STACK_SIZE);

    pidfirst =  clone(Child,stackf + STACK_SIZE, CLONE_VM);
    pidsecond = clone(Child,stacks + STACK_SIZE, CLONE_VM);
    waitpid(-1, NULL, __WALL);
    char buf[BUFSIZE];
    sprintf(buf,"Back to parent: Value of n: %d\n",n);
    write(1,buf,strlen(buf));
    return 0;
}

int Child(void *args) {
      int j = 0;
      for(j=0;j<MAXN;j++){
        n += 1;
      }

}
```

```
user@:~$ gcc mts.c -o mts
user@:~$ ./mts
Back to parent: Value of n: 1980887
user@:~$ ./mts
Back to parent: Value of n: 2000000
user@:~$ ./mts
Back to parent: Value of n: 1897787
user@:~$ ./mts
Back to parent: Value of n: 1894872
user@:~$ ./mts
Back to parent: Value of n: 1951541
user@:~$ ./mts
Back to parent: Value of n: 1989334
user@:~$ ./mts
Back to parent: Value of n: 1889019
user@:~$ _
```

Origine del problema osservato:

Il problema osservato nell'esempio precedente è dovuto ad una interazione tra processi che è avvenuta in maniera incontrollata.

E' possibile classificare I tipi di interazione tra processi in **tre grandi classi principali**:

1) I processi sono coscienti unicamente di sè stessi. Il fatto che le risorse sono finite crea comunque un problema di **competizione**. Es. vari processi tentano di accedere al medesimo disco o stampante. Il s.o. deve regolare gli accessi.

2) I processi sono **indirettamente** coscienti di eseguire insieme ad altri processi. L'interazione si realizza mediante risorse condivise (ad es. memoria). In questo caso si realizza una **cooperazione**.

3) I processi sono **esplicitamente** coscienti di eseguire insieme ad altri processi. Anche in questo caso si realizza una **cooperazione** ma è possibile che questa sia basata su meccanismi di **comunicazione** tra processi.

NB: Ognuno di questi casi pone problemi riguardanti il **controllo** dell'esecuzione dei processi. E' necessario assicurare che non si verifichino effetti indesiderati.

Table 5.2 Process Interaction

Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	• Results of one process independent of the action of others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation • Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Deadlock (consumable resource) • Starvation

Corsa critica:

Una corsa critica si verifica quando più processi (o thread) leggono/scrivono una variabile, e quando il risultato finale dipende unicamente dall'ordine con cui vengono eseguite le istruzioni appartenenti ai vari processi.

Esempio 1:

- P1 e P2 condividono una variabile globale a
- Ad un certo punto della sua esecuzione P1 scrive in a il valore 1
- Ad un certo punto della sua esecuzione P2 scrive in a il valore 2
- P1 e P2 competono per scrivere in a … il valore finale di a sarà determinato dall'ultimo dei due processi che riuscirà a scrivere nella variabile. In definitiva, chi "perde" la corsa per scrivere in a ne determina il valore finale.

Esempio 2:

- P3 e P4 condividono 2 variabili globali b e c con valori iniziali b=1 c=2
- Ad un certo punto della sua esecuzione P3 esegue l'assegnamento b = b+c
- Ad un certo punto della sua esecuzione P4 esegue l'assegnamento c = b+c
- Ordine di assegnazioni → P3,P4 : risultato finale b=3 c=5 P4,P3: risultato finale: b=4 c=3

Deadlock (stallo):

Definizione:

"Un insieme di processi si trova in una situazione di stallo se ogni processo dell'insieme ***aspetta*** *un evento che che solo un altro processo dell'insieme può provocare."*

… poichè **tutti** I processi stanno aspettando nessuno di essi potrà mai provocare l'evento che fa ripartire un'altro processo.

Condizioni per il deadlock (Coffman et al, 1971)

- Condizione di **mutua esclusione**: ogni risorsa è assegnata ad un processo oppure è libera
- Condizione di "prendi e aspetta" (**hold and wait**): I processi che hanno già richiesto ed ottenuto una risorsa ne possono chiedere delle altre
- **Mancanza di prerilascio**: le risorse che sono già state assegnate ad un processo non possono essergli tolte in modo forzato ma devono essere rilasciate volontariamente.
- **Attesa circolare**: deve esistere una catena circolare di processi, ognuno dei quali aspetta il rilascio di una risorsa da parte del processo che lo segue

Starvation (inedia):

Problema strettamente in relazione con deadlock.

*Dato il costante verificarsi di richieste di risorse in un sistema dinamico è necessaria qualche politica che stabilisca l'**ordine** ed il **tempo di utilizzo** di queste ultime. In alcuni casi è possibile che un processo rimanga in attesa per un tempo indefinito e resti bloccato perchè non gli viene **mai** assegnata la risorsa di cui ha bisogno (es. processo a bassa priorità in un ambiente in cui si generano spesso richieste ad alta priorità).*

Esempio:

Consideriamo tre processi P1, P2 e P3 che esibiscono questo comportamento:
P1 cerca ripetutamente di comunicare sia con P2 che con P3
P2 e P3 cercano ripetutamente di comunicare con P1

Può verificarsi una situazione in cui P1 e P2 si scambiano messaggi con alta frequenza mentre P3 è bloccato in attesa di riuscire a comunicare con P1. Non c'è deadlock (P1 rimane attivo) ma P3 è bloccato in stato di inedia.

Sezione critica:

Come abbiamo evidenziato mediante l'ultimo esempio per poter coordinare thread che condividono memoria sono necessari dei meccanismi di **sincronizzazione** che assicurino un'esecuzione "ordinata" dei thread in modo da preservare la coerenza dei dati (cfr slide 11).

La **sezione critica** è la parte di codice in cui i processi accedono a dati **condivisi**.

Consideriamo un generico thread T in cui è presente una sezione critica:

```
thread T {
   ....
   < ingresso >
   < sezione critica >
   < uscita >
   ....
}
```

vogliamo progettare il codice di ingresso e uscita dalla sezione critica in modo da garantire mutua esclusione ovvero che **un solo un thread alla volta** possa accedere alla sezione critica. Questa proprietà deve valere senza fare assunzioni sulla velocità relativa dei processi (temporizzazione), a prescindere quindi da come vengono schedulati.

Soluzioni software (niente meccanismi hardware e syscall):

Tentativo 1: lock (variabile booleana globale lock inizializzata a false

```
thread T {
   ....
   while(lock) {}
   lock = true;
   < sezione critica >
   lock = false;
   ....
}
```

Il codice di ingresso attende ciclando "a vuoto" **finché lock vale true**. Quando il lock viene rilasciato esce dal while e acquisisce il lock, ponendolo a true. Alla fine della sezione critica il lock viene settato a false e quindi rilasciato.

Problema: se due thread entrano contemporaneamente in sezione critica potrebbe accadere che superino **entrambi** il while (leggono il valore false prima che l'altro thread abbia settato lock a true). In tale caso è come se entrambi acquisissero il lock e la mutua esclusione non è garantita: **ci sono 2 thread in sezione critica**.

Tentativo 2: turno (Utilizziamo una variabile globale turno inizializzata a 1. I thread eseguono codice che dipende dal proprio ID. Es. con 2 thread)

```
thread T0 {
   ....
   while(turno != 0) {}
   < sezione critica >
   turno = 1;
   ....
}
```

```
thread T1 {
   ....
   while(turno != 1) {}
   < sezione critica >
   turno = 0;
   ....
}
```

Manca una seconda proprietà fondamentale : se nessun thread è in sezione critica un thread che chiede di accedere deve poterlo fare **immediatamente**.

Descrizione: il codice di ingresso attende ciclando "a vuoto" finché non è il proprio turno. Alla fine della sezione critica il turno viene dato all'altro thread.
Mutua esclusione: la mutua esclusione è garantita dal fatto che turno non può valere 0 e 1 contemporaneamente, di conseguenza uno dei due thread rimarrà in attesa sul ciclo while.

Problema: Anche se la soluzione garantisce mutua esclusione esistono esecuzioni problematiche: supponiamo che T0 voglia entrare **2 volte di seguito** nella sezione critica mentre T1 sta eseguendo altro codice. T0 esce dalla prima sezione critica e assegna il turno a T1, a questo punto si bloccherà sul while in attesa che T1 vada in sezione critica e ceda a sua volta il turno. Il problema è che non sappiamo se e quando T1 entrerà in sezione critica (potrebbe anche non andarci mai!).

Tentativo 3: pronto (2 thread utilizzando un array di booleani globale pronto[2] inizializzati a false. Anche in questo caso i thread eseguono codice che dipende dal proprio ID.)

```
thread T0 {
   ....
   pronto[0] = true;
   while(pronto[1]) {}
   < sezione critica >
   pronto[0] = false;
   ....
}
```

```
thread T1 {
   ....
   pronto[1] = true;
   while(pronto[0]) {}
   < sezione critica >
   pronto[1] = false;
   ....
}
```

Potenziale deadlock

Descrizione: quando il thread i-esimo vuole entrare pone a true la variabile pronto[i] per segnalare all'altro thread la sua volontà di accedere. Successivamente cicla a vuoto se anche l'altro thread è pronto. Quando esce dalla sezione critica, il thread pone a false pronto[i] per indicare che non ha più bisogno di entrare in sezione critica.

Mutua esclusione: la mutua esclusione è garantita dal fatto che se un thread è in sezione critica la sua variabile pronto è settata a true e l'altro thread si bloccherà sul ciclo while.

Problema: Anche se la soluzione garantisce mutua esclusione esistono esecuzioni problematiche: supponiamo che T0 e T1 vogliano entrare entrambi in sezione critica. Potrebbe accadere che settano simultaneamente pronto[i]=true e poi **si bloccano entrambi sul proprio ciclo while**. Notare che questa situazione di attesa è irrisolvibile e quindi inaccettabile.

Soluzione: algoritmo di Peterson

Nel 1981, Gary L. Peterson propose una soluzione al problema della soluzione critica per 2 thread che di fatto combina 2 dei tentativi che abbiamo descritto sopra: **pronto** e **turno**.

Il difetto di pronto è che esiste un caso in cui i thread vanno in stallo. Per evitare questa situazione irrisolvibile **utilizziamo una turnazione**, ma **solo nel caso problematico**: quando **entrambi** i thread sono pronti. Se uno solo è pronto, tale thread può tranquillamente accedere alla sezione critica.

```
thread T0 {
   ....
   pronto[0] = true;
   turno=1;
   while(pronto[1] && turno != 0) {}
   < sezione critica >
   pronto[0] = false;
   ....
}
```

```
thread T1 {
   ....
   pronto[1] = true;
   turno=0;
   while(pronto[0] && turno != 1) {}
   < sezione critica >
   pronto[1] = false;
   ....
   }
```

Soluzione: algoritmo di Peterson

Nel 1981, Gary L. Peterson propose una soluzione al problema della soluzione critica per 2 thread che di fatto combina 2 dei tentativi che abbiamo descritto sopra: **pronto** e **turno**.

Descrizione: quando il thread i-esimo vuole entrare pone a true la variabile pronto[i] per segnalare all'altro thread la sua volontà di accedere. Successivamente cede il turno all'altro thread e cicla a vuoto se l'altro thread è pronto **E** se non è il proprio turno. Quando esce dalla sezione critica, il thread pone a false pronto[i] per indicare che non ha più bisogno di eseguire la sezione critica.

```
thread T0 {
   ....
   pronto[0] = true;
   turno=1;
   while(pronto[1] && turno != 0) {}
   < sezione critica >
   pronto[0] = false;
   ....
}
```

```
thread T1 {
   ....
   pronto[1] = true;
   turno=0;
   while(pronto[0] && turno != 1) {}
   < sezione critica >
   pronto[1] = false;
   ....
   }
```

algoritmo di Peterson

Mutua esclusione: Per convincerci che viene garantita mutua esclusione dobbiamo considerare due casi

- T0 supera il proprio while quando T1 è prima di turno=0. In questo caso quando T1 eseguirà turno=0 si bloccherà nel ciclo while in quanto avremo che pronto[0] è true e turno è 0 cioè diverso da 1;
- T0 supera il proprio while quando T1 è dopo turno=0. Poiché T0 sta superando il while deve essere turno==0. Quindi avremo che T1 ancora una volta si blocca nel ciclo while ed attende.

Progresso: Se nessuno è in sezione critica pronto[i] vale false. Chiunque voglia entrare ci riuscirà in quanto la prima condizione nel while sarà falsa.

Assenza di stallo: Supponiamo che entrambi i thread siano bloccati nel while. In questo caso avremmo che turno!=0 e turno!=1 ma questo **non è possibile** in quanto turno viene assegnata unicamente ai valori 0 e 1 quindi **uno dei due thread uscirà necessariamente dal ciclo while**.

WARNING: aggiungere qui
#define _GNU_SOURCE

Thread
(**mutua esclusione con peterson**)
threads-peterson.c (1/2)

```
/* Copyright (C) 2009 by Mattia Monga <mattia.monga@unimi.it> */
/* $Id$ */

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sched.h>


void enter_section(int process, int* turn, int* interested)
{
        int other = 1 - process;
        interested[process] = 1;
        *turn = process;
        while  (*turn == process && interested[other]){
                printf("Busy waiting di %d\n", process);
        }
}

void leave_section(int process, int* interested)
{
        interested[process] = 0;
}

int run(const int p, void* s)
{
        int* shared = (int*)s; // alias per comodit\`a
        while (enter_section(p, &shared[1], &shared[2]), shared[0] < 10) {
                sleep(1);
                printf("Processo figlio (%d). s = %d\n",
                       getpid(), shared[0]);
                if (!(shared[0] < 10)){
                        printf("Corsa critica!!!!\n");
                        abort();
                }
                shared[0] += 1;
                leave_section(p, &shared[2]);
        }
        leave_section(p, &shared[2]);// il test nel while \`e dopo enter\_section

        return 0;
}
```

Thread
(**mutua esclusione con peterson**)
threads-peterson.c (2/2)

```
int run0(void*s){ return run(0, s); }
int run1(void*s){ return run(1, s); }


int main(void){

        int shared[4] = {0 , 0, 0, 0};

        /* int clone(int (*fn)(void *),
         *           void *child_stack,
         *           int flags,
         *           void *arg);
         * crea una copia del chiamante (con le caratteristiche
         * specificate da flags) e lo esegue partendo da fn */
        if (clone(run0, /* il nuovo
                         * processo esegue run(shared), vedi quarto
                         * parametro */
                  malloc(4096)+4096,  /* lo stack del nuovo processo
                                       *  (cresce verso il basso!) */
                  CLONE_VM | SIGCHLD, /* la (virtual) memory \`e condivisa */
                  shared) < 0){
                perror("Errore nella creazione");
                exit(1);
        }

        if (clone(run1, malloc(4096)+4096,  CLONE_VM | SIGCHLD, shared) < 0){
                perror("Errore nella creazione");
                exit(1);
        }

        /* Memoria condivisa: i due figli nell'insieme eseguono 10 o
         * 11 volte con possibili corse critiche. Il padre
         * condivide shared[0] con i figli */

        while(shared[0] < 10) {
                sleep(1);
                printf("Processo padre. s = %d %d %d %d\n",
                       shared[0],
                       shared[1],
                       shared[2],
                       shared[3]);
        }
        return 0;
}
```

```
user@:~$ gcc -o thrp threads-peterson.c
user@:~$ time ./thrp > /tmp/output

real    0m11.044s
user    0m0.011s
sys     0m0.018s
user@:~$ grep -c "Busy waiting" /tmp/output
2832046
user@:~$ _
```

algoritmo di Peterson

Conclusioni:

È possibile risolvere il problema della sezione critica puramente in software ma tali soluzioni presentano alcune problematiche importanti:

- I cicli while “a vuoto” **consumano inutilmente tempo di CPU**. Su macchine con un solo core questo problema è particolarmente rilevante. Su più core diventa più accettabile in quanto i vari core dovrebbero comunque attendere che il thread in sezione critica esca;

- Le soluzioni richiedono l’uso di **diverse variabili globali e un sequenza precisa di esecuzione delle istruzioni**. I compilatori e i processori attuali rimaneggiano e riordinano le istruzioni in modo da migliorare la performance. L’utilizzo di soluzioni software di sincronizzazione richiedere che **tali ottimizzazioni siano disabilitate** per evitare di compromettere la correttezza del codice di ingresso e uscita dalla sezione critica.

Mutua esclusione mediante istruzioni hardware

Esistono diversi meccanismi tramite I quali è possibile ottenere una mutua esclusione utilizzando delle istruzioni hardware. Il caso più banale è il seguente:

In un sistema a singola CPU l'esecuzione dei processi non può sovrapporsi nel tempo. Ogni processo continuerà ad eseguire fino al momento in cui invocherà un servizio fornito dal s.o. o riceverà una interruzione.

Disabilitando le interruzioni l'esecuzione della sezione critica di un processo non può essere interrotta e quindi la mutua esclusione è garantita!

Il prezzo, però, è molto alto:
L'efficienza di esecuzione è notevolmente ridotta (il s.o. non può più allocare in maniera dinamica ed efficiente le risorse). Inoltre questo approccio non funziona in architetture multiprocessore. Se più processi eseguono su duversi processori la mutua esclusione non è più garantita.

Test and Set Lock (TSL)

Una istruzione test-and-set è una istruzione utilizzata per scrivere qualcosa in una specifica locazione in memoria **E** ritornare il vecchio valore contenuto in memoria. Il tutto in una **UNICA e ATOMICA (impossibile da interrompere) operazione**.

Tipicamente il valore da scrivere è 1. Se più processi cercano di accedere alla medesima locazione in memoria, e se un processo sta effettuando una operazione di test-and-set, agli altri processi **non è permesso** di effettuare l'operazione test-and-set fino a quando il primo processo ha completato il test-and-set.

Questo meccanismo si può sfruttare per forzare la mutua esclusione! Esempio di funzione di ingresso in sezione critica implementata mediante istruzioni hardware.

enter.asm (1/1)

bts : bit test and set

```
section .text
global enter_section

enter_section:
        enter 0, 0              ; 0 bytes of local stack space
        mov   ebx,[ebp+8]       ; first parameter to function

spin:   lock bts dword [ebx], 0 ; The value of the first operand (base) and the second
                                ; operand (bit offset) are saved by bts into the carry flag
                                ; and then it stores 1 in the bit.
        jc spin                 ; jc : JUMP IF CF (carry flag)

        leave                   ; mov esp,ebp / pop ebp
        ret
```

Thread (**mutua esclusione con TSL**) threads-tsl.c (1/2)

WARNING: aggiungere qui
#define _GNU_SOURCE

```
/* Copyright (C) 2009 by Mattia Monga <mattia.monga@unimi.it> */
/* $Id$ */

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sched.h>


void enter_section(int *s); /* in enter.asm */
void leave_section(int *s){ *s = 0; }

int run(const int p, void* s){
        int* shared = (int*)s; // alias per comodit\`a
        while (enter_section(&shared[1]), shared[0] < 10) {
                sleep(1);
                printf("Processo figlio (%d). s = %d\n",
                       getpid(), shared[0]);
                if (!(shared[0] < 10)){
                        printf("Corsa critica!!!!\n");
                        abort();
                }
                shared[0] += 1;
                leave_section(&shared[1]);
                sched_yield();
        }
        leave_section(&shared[1]); // il test nel while \`e dopo enter\_section
        return 0;
}

int run0(void*s){ return run(0, s); }
int run1(void*s){ return run(1, s); }
```

sched_yield() causes the calling thread to relinquish the CPU. The thread is moved to the end of the queue for its static priority and a new thread gets to run.

WARNING: aggiungere qui
usleep(50)

Sistemi Operativi

Laboratorio – linea 2

5

Thread
(**mutua esclusione con TSL**)
threads-tsl.c (2/2)

```
$ nasm -f elf -o enter.o enter.asm
$ gcc -c threads-tsl.c
$ gcc threads-tsl.o enter.o -o thrt
```

```
int main(void){

        int shared[4] = {0 , 0, 0, 0};

        /* int clone(int (*fn)(void *),
         *           void *child_stack,
         *           int flags,
         *           void *arg);
         * crea una copia del chiamante (con le caratteristiche
         * specificate da flags) e lo esegue partendo da fn */
        if (clone(run0, /* il nuovo
                         * processo esegue run(shared), vedi quarto
                         * parametro */
                  malloc(4096)+4096,  /* lo stack del nuovo processo
                                       *  (cresce verso il basso!) */
                  CLONE_VM | SIGCHLD, /* la (virtual) memory \`e condivisa */
                  shared) < 0){
                perror("Errore nella creazione");
                exit(1);
        }

        if (clone(run1, malloc(4096)+4096,  CLONE_VM | SIGCHLD, shared) < 0){
                perror("Errore nella creazione");
                exit(1);
        }

        /* Memoria condivisa: i due figli nell'insieme eseguono 10 o
         * 11 volte: \`e possibile una corsa critica. Il padre
         * condivide shared[0] con i figli */

        while(shared[0] < 10) {
                sleep(1);
                printf("Processo padre. s = %d %d %d %d\n",
                       shared[0],
                       shared[1],
                       shared[2],
                       shared[3]);
        }
        return 0;
}
```

```
user@:~$ ./thrt
Processo figlio (2263). s = 0
Processo padre. s = 0 1 0 0
Processo padre. s = 1 1 0 0
Processo figlio (2263). s = 1
Processo padre. s = 2 1 0 0
Processo figlio (2263). s = 2
Processo padre. s = 3 1 0 0
Processo figlio (2263). s = 3
Processo padre. s = 4 1 0 0
Processo figlio (2263). s = 4
Processo padre. s = 5 1 0 0
Processo figlio (2263). s = 5
Processo padre. s = 6 1 0 0
Processo figlio (2263). s = 6
Processo padre. s = 7 1 0 0
Processo figlio (2263). s = 7
Processo padre. s = 8 1 0 0
Processo figlio (2263). s = 8
Processo padre. s = 9 1 0 0
Processo figlio (2263). s = 9
Processo padre. s = 10 0 0 0
user@:~$ _
```

Concorrenza:

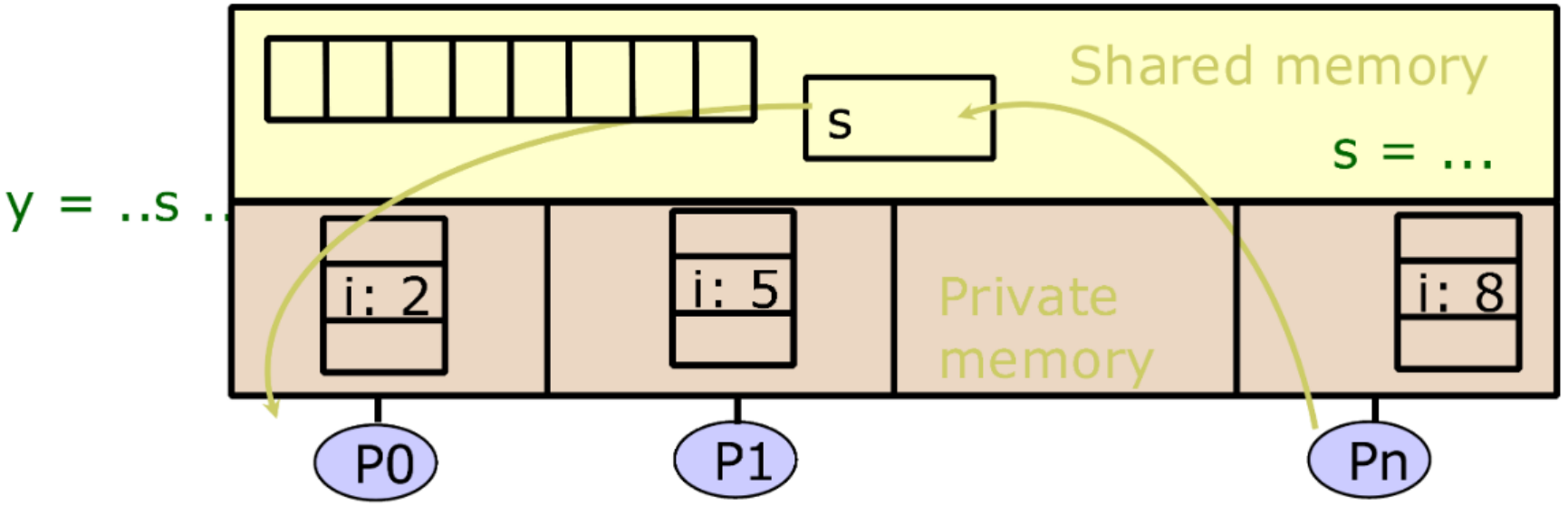
PThreads

POSIX threads

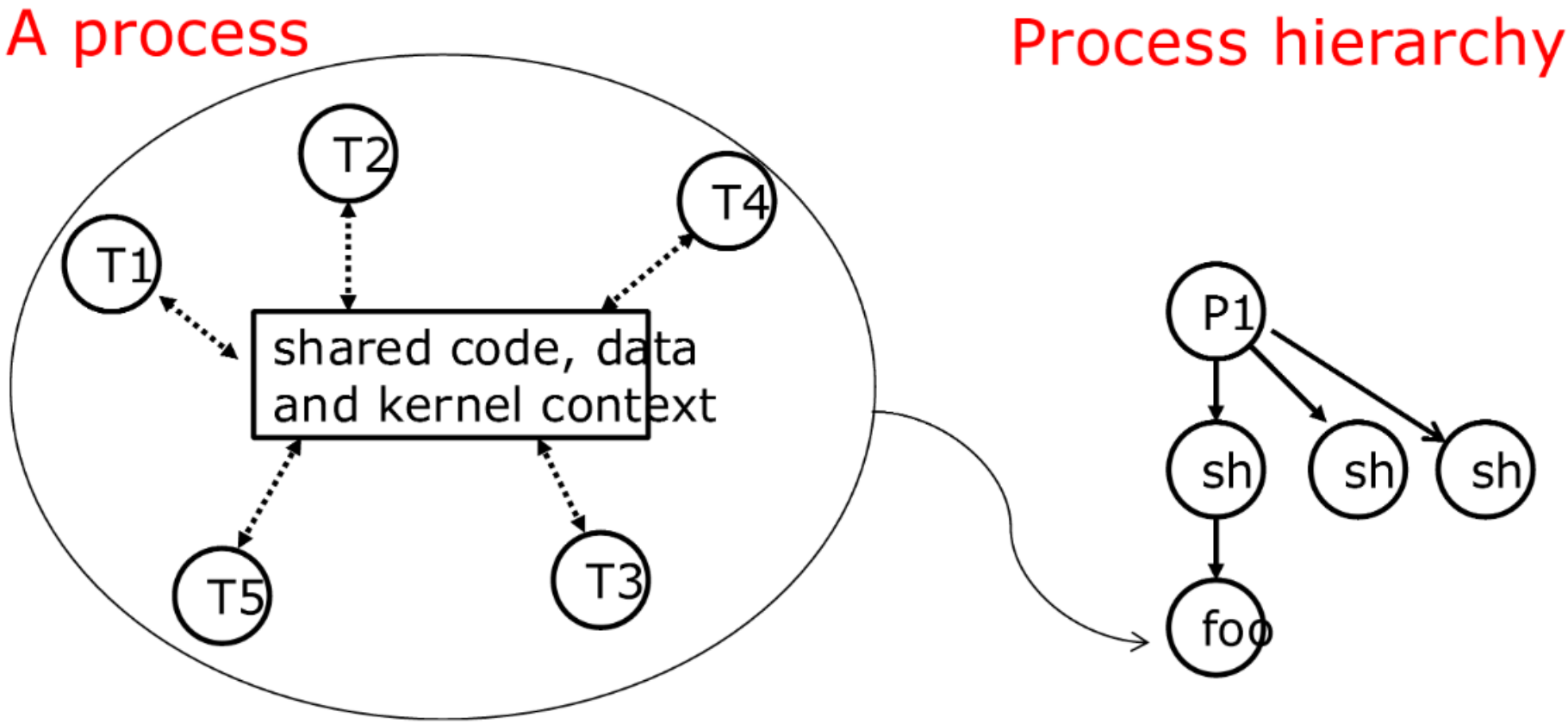
- Tecnicamente un thread è definito come una **unità di esecuzione**, una serie di istruzioni che può essere eseguita indipendentemente dall'esecuzione degli altri thread sulla base delle politiche dello schedulatore.
- Ma cosa vuol dire?
- Il concetto di “procedura” che esegue indipendentemente dal programma principale descrive bene l'idea di thread.
- Immaginate un programma principale che contiene molte procedure.
- Ora immaginate che tutte le procedure contenute nel programma principale siano in grado di eseguire simultaneamente e/o indipendentemente senza modificare l'output finale del programma.
- Questo descrive un programma "multi-threaded".
- Ma come è possibile realizzare un simile programma?

POSIX threads

- Ogni thread ha un suo set di variabili **private** (ad es. lo stack ed il program counter)
- Ogni thread ha anche "variabili" **condivise** con tutti gli altri thread (ad es. variabili statiche, heap globale).
- I thread **comunicano** implicitamente leggendo/scrivendo le variabili condivise!
- I thread vengono sincronizzati basandosi sulle variabili condivise.



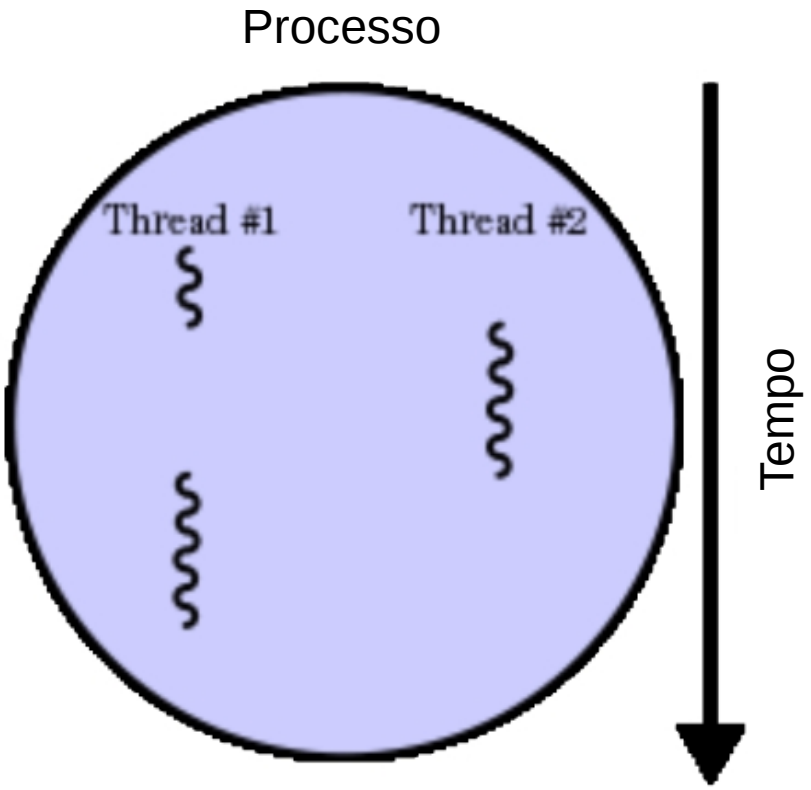
I thread vengono creati <u>all'interno</u> dei processi :



Questo rende più semplice sfruttare uno dei punti di forza dei thread:
Esecuzione / risorse sono disaccoppiate …

Benefici approccio multi threaded:

- Responsività
- Condivisione risorse (memoria condivisa)
- Economia (dal punto di vista algoritmico)
- Scalabilità (utile nel caso di CPU multi core)

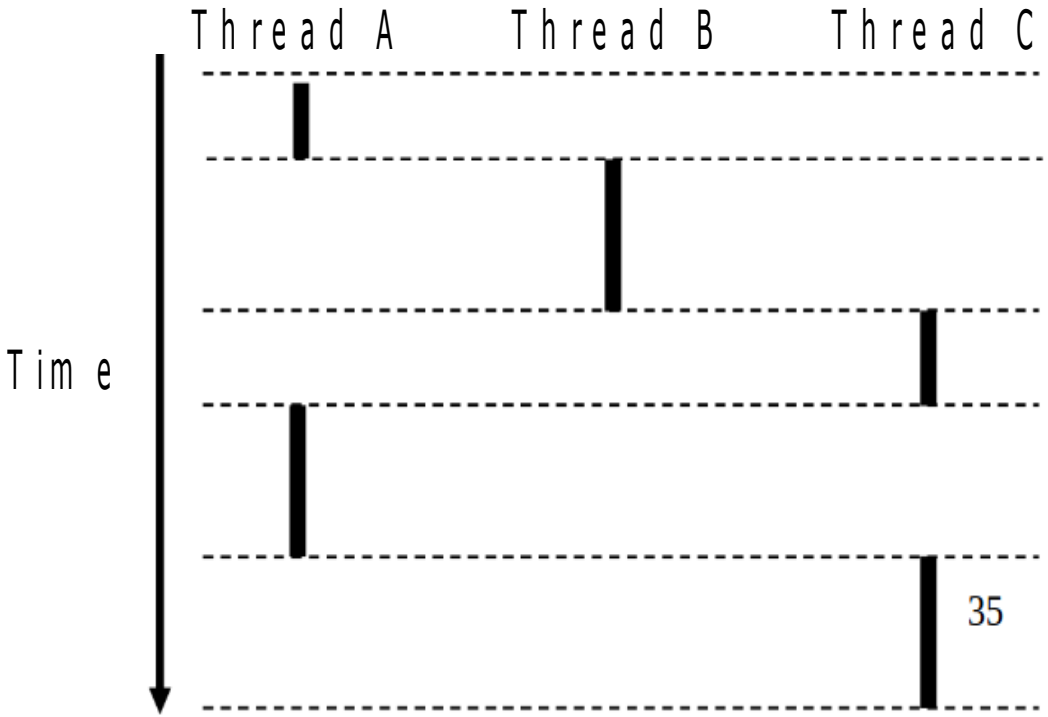


Esecuzione concorrente :

- Due thread sono concorrenti (eseguono in modo concorrente) se il loro flusso logico si sovrappone nel tempo
- In caso contrario eseguono in modo sequenziale

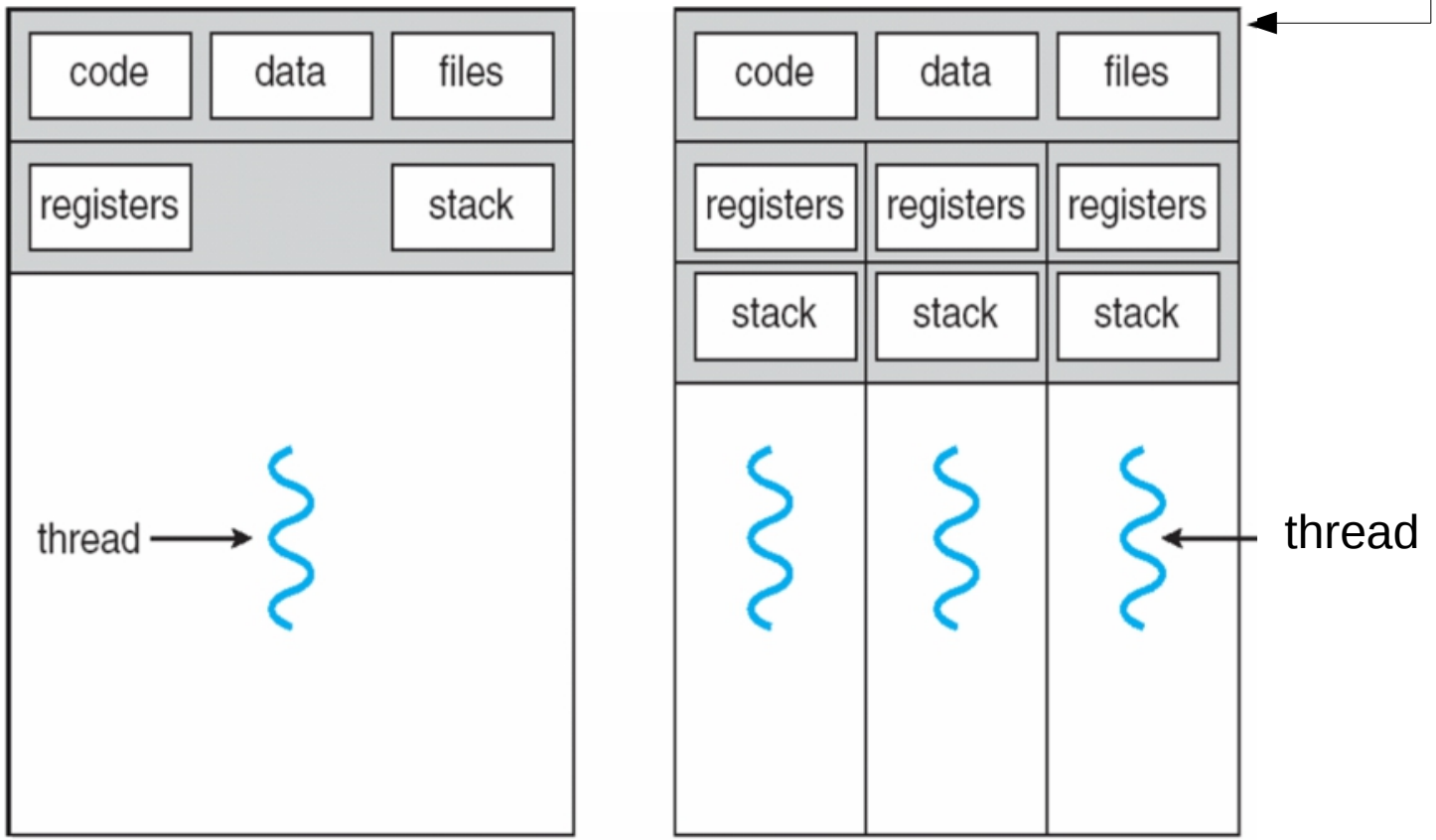
- Esempio:

Concorrenti: A & B, A & C
Sequenziali: B & C



Differenze tra processi a thread singolo e a thread multipli :

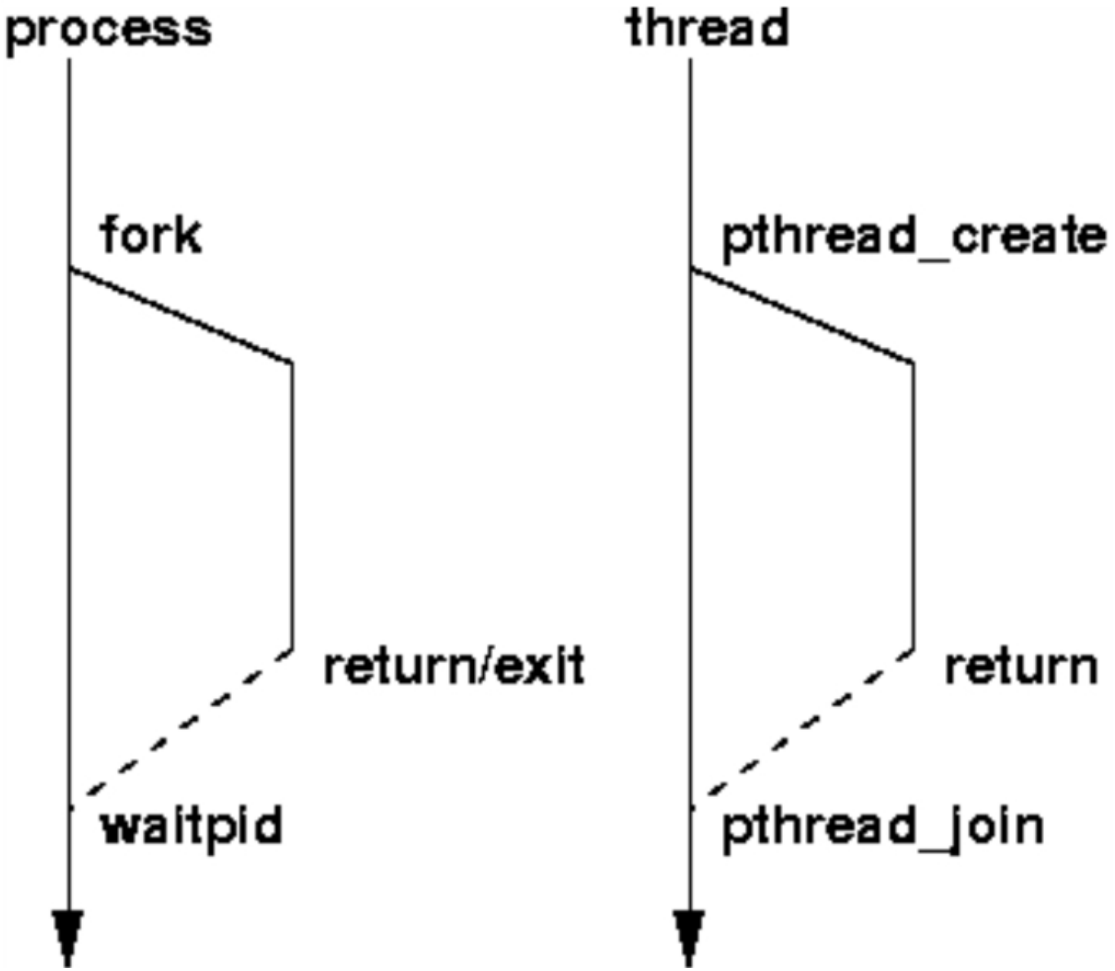
Accesso mediante memoria condivisa per codice e dati
Controllo di flusso separato (stack, registri)



POSIX e Pthreads:

- POSIX: ***P**ortable **O**perating **S**ystem **I**nterface for UNI**X***
  - Interface to Operating System utilities
- **PThreads**: The POSIX threading interface
  - System calls to create and synchronize threads
  - **compilation** with gcc -lpthread
- PThreads contain support for
  - Creating parallelism and synchronization
  - No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread

Creazione di thread:



Creazione di thread:

Signature:

```
int pthread_create(pthread_t *,
                   const pthread_attr_t *,
                   void * (*)(void *),
                   void *);
```

Example call:

```
errcode = pthread_create(&thread_id; &thread_attribute
                         &thread_fun; &fun_arg);
```

thread_id è il thread id o thread handle (si usa, ad es., per fermare il thread)

thread_attribute vari attributi
Valori di default ottenuti passando un puntatore a NULL. Esempio di attributi: minima dimensione stack

thread_fun la funzione da eseguire (argomento e tipo ritorno: void*)

fun_arg argomenti da passare a thread_fun
Se fallisce ritorna codice d'errore diverso da 0

Altre funzioni in PThreads:

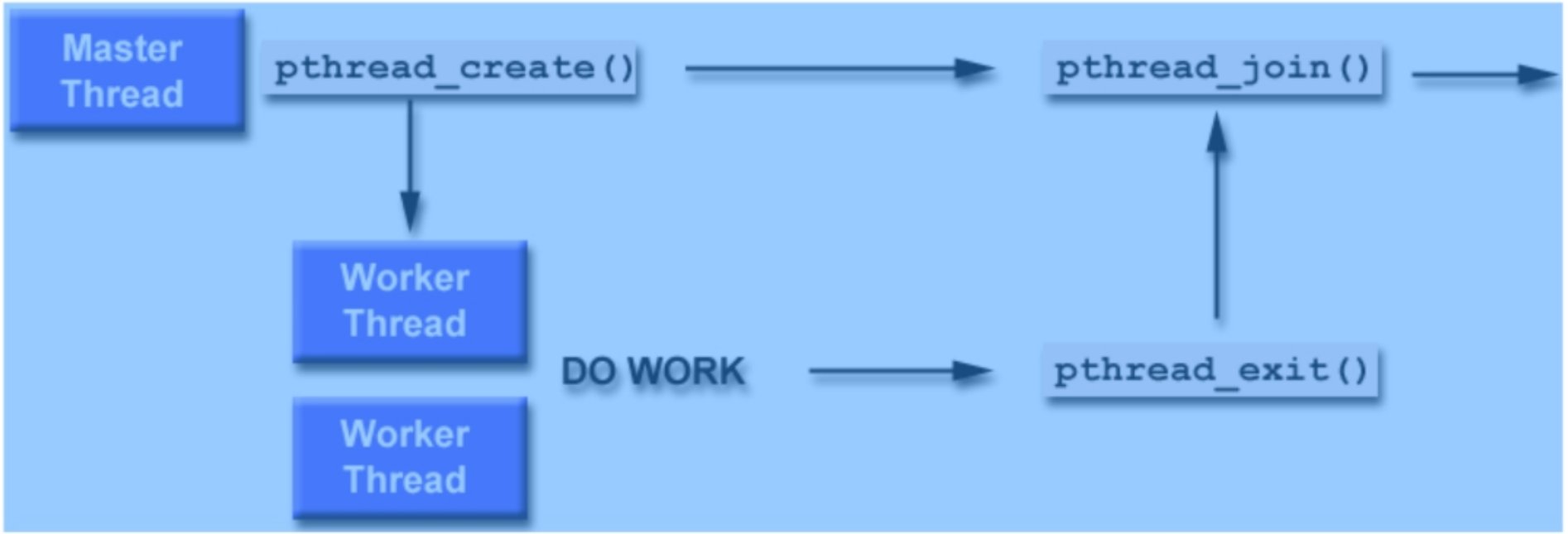
- `pthread_exit(void *value);`
  - Exit thread and pass value to joining thread (if exists)
- `pthread_join(pthread_t *thread, void **result);`
  - Wait for specified thread to finish. Place exit value into *result.

Others:

- `pthread_t me; me = pthread_self();`
  - Allows a pthread to obtain its own identifier pthread_t thread;

Join di pthreads:

pthread_join() fornisce un meccanismo molto semplice di sincronizzazione tra thread …



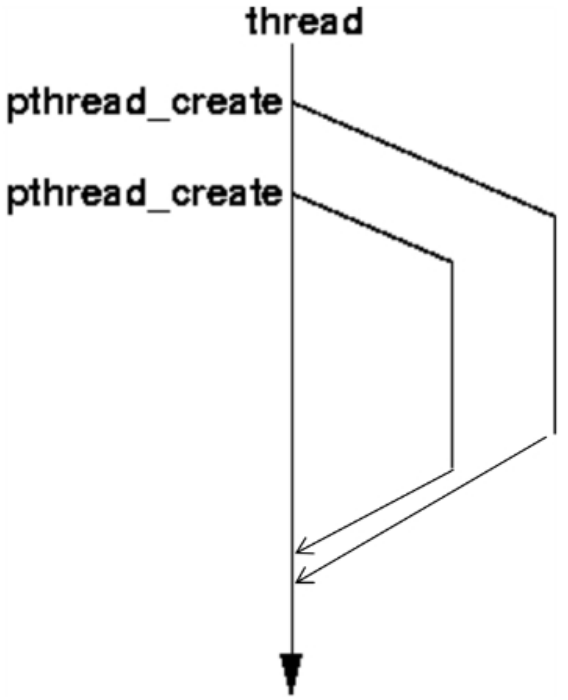
Esempio di pthread che utilizzano pthread_join() mt2.c :

```
#include <pthread.h>
#include <stdio.h>

void *PrintHello(void * id){
   printf("Thread%d: Hello World!\n", id);
}



void main (){
   pthread_t thread0, thread1;
   pthread_create(&thread0, NULL, PrintHello, (void *) 0);
   pthread_create(&thread1, NULL, PrintHello, (void *) 1);
   pthread_join(thread0, NULL);
   pthread_join(thread1, NULL);
}
```



Ricordatevi di usare

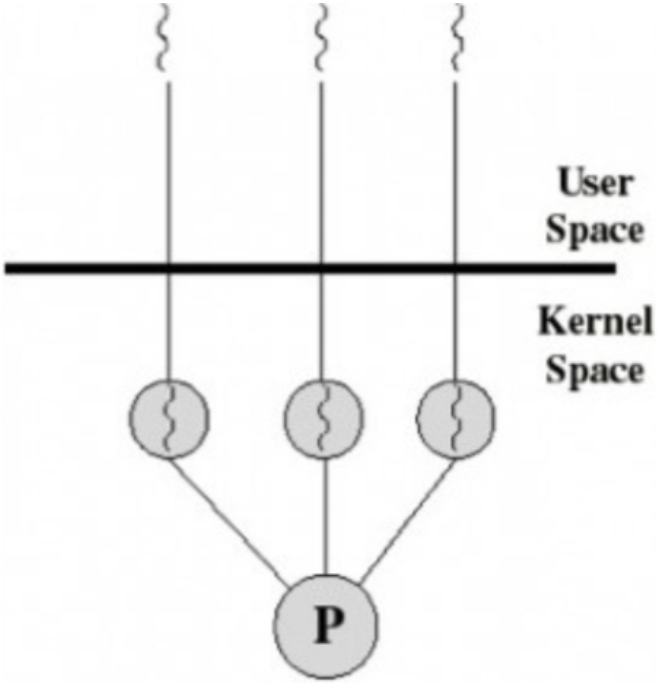
-lpthread quando compilate

Tipi di thread: KERNEL thread

Riconosciuti e gestiti dal s.o.

Scheduling e switch di contesto eseguiti **direttamente** dal kernel.

Sfruttano CPU multi core quando disponibili

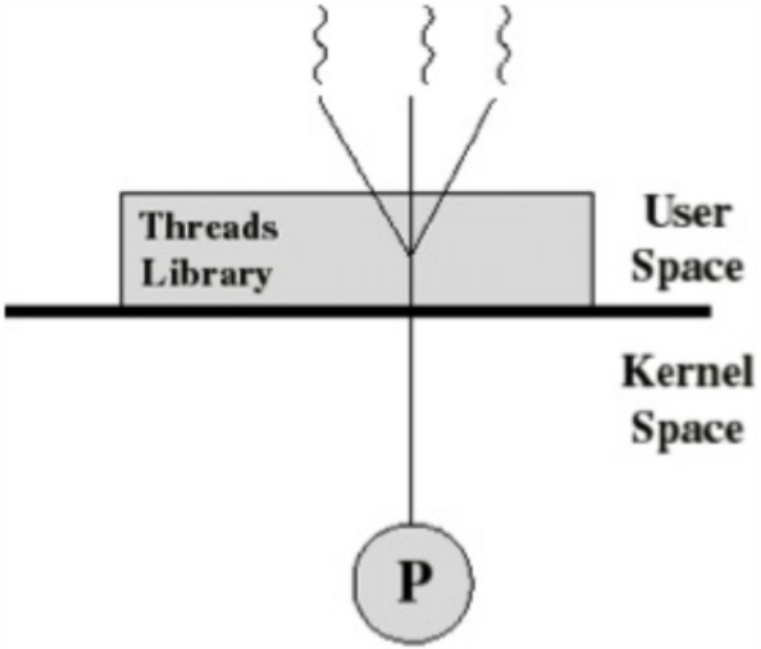


Tipi di thread: USER thread

Gestione dei thread a carico di una libreria user-level dedicata

Il s.o. non sa nulla di questi thread (vede solo processi che eseguono all'interno di un programma)

La libreria è responsabile sia dello scheduling che del context switching dei threads generati tramite il suo utilizzo.



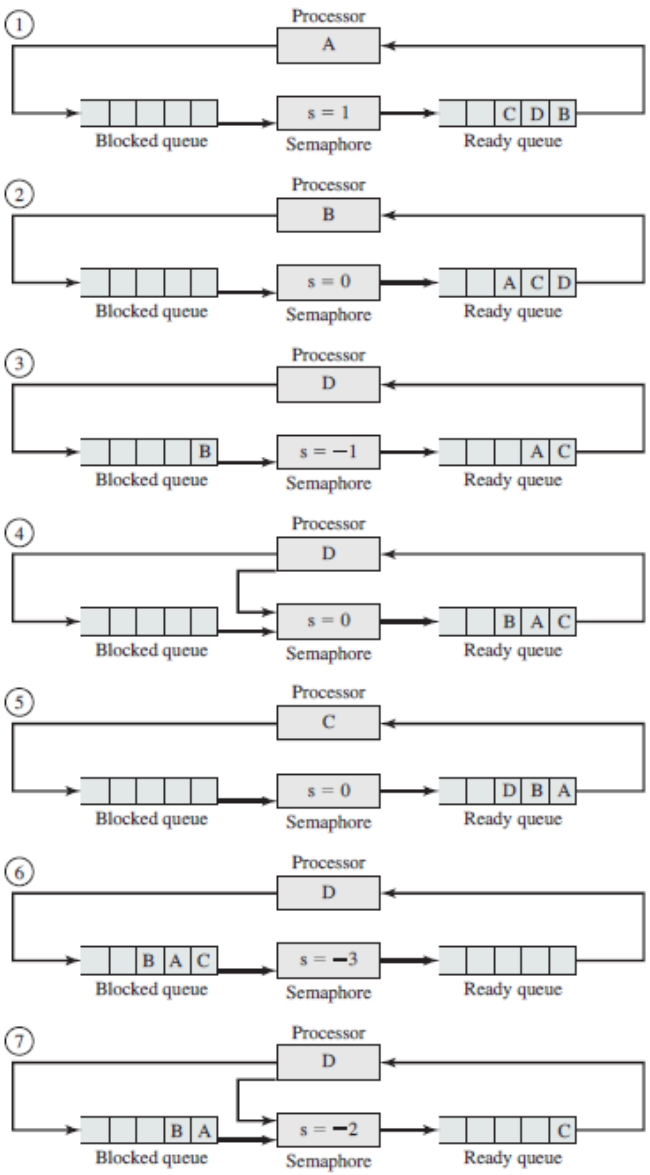
Generalizzazione dei lock : SEMAFORI

semaforo S : variabile di tipo **intero**

E' possibile accedere al semaforo unicamente mediante due operazioni indivisibili (atomiche) :

```
wait(S){
        while S <= 0
        // wait
    S--;
}

post(S){
    S++;
}
```



SEMAFORI, logica di utilizzo : produttore - consumatori

I processi A, B e C **dipendono** dai risultati prodotti dal processo D.

1) Inizialmente A è in esecuzione, B,C e D sono pronti per essere eseguiti. Il valore del semaforo è **1**, indicando che 1 risultato di D è disponibile.

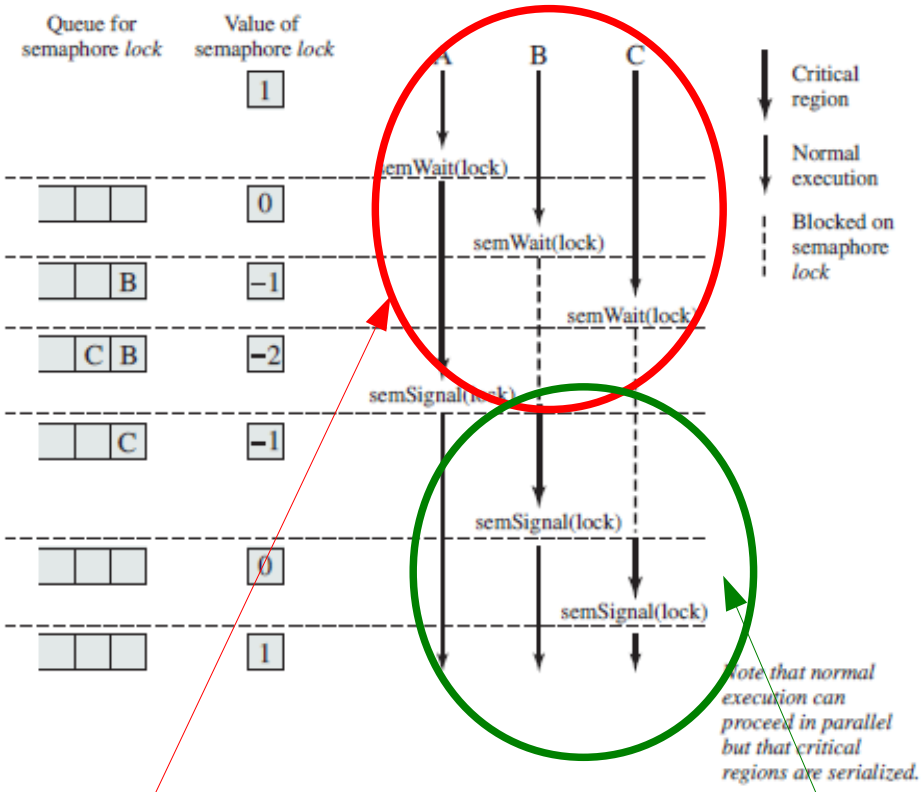
2) B inizia ad eseguire ed, eventualmente, utilizza l'istruzione *wait()*, rimanendo bloccato. Questo permette a D di rientrare in esecuzione (3).

4) Quando D completa il suo lavoro, peoducendo un nuovo risultato, lancia un segnale per notificarne la disponibilità. B può muoversi dalla coda di attesa a quella di esecuzione.

5) D rientra nella coda di esecuzione e C inizia ad eseguire ma si blocca quando unsa una istruzione wait(). Nello stesso modo A e B sono bloccati in attesa.

6) Questo permette a D di riprendere la sua esecuzione. Quando D produce un risultato emette un segnale che permette a C di rientrare nella coda dei processi pronti per l'esecuzione.

SEMAFORI: realizzazione mutua esclusione



Il semaforo s è inizializzato ad 1.

Il primo processo che chiama wait() entra immediatamente in sezione critica e setta il valore di s a 0.

Ogni processo che chiede di entrare in sezione critica, wait(), decrementa s e rimane bloccato.

Quando il processo che è entrato in sezione critica finisce di eseguirla, s è incrementato e uno dei processi in attesa può essere rimosso da una coda di attesa ed inserito in una coda di processi pronti per l'esecuzione.

Quando verrà schedulato dal s.o. potrà entrare nella sua sezione critica.

Dipendentemente dal valore iniziale di s alcune sezioni critiche possono sovrapporsi.

Ma in **breve tempo** l'esecuzione delle sezioni critiche diventa mutualmente esclusiva

SEMAFORI: realizzazione mutua esclusione

Un mutex è un semaforo utilizzato per garantire mutua esclusione. Esso è una variante più "stringente" dei semafori che permette un **unico locker** (entità che determina il blocco del semaforo) per volta.

In altre parole esso è equivalente ad un normale semaforo ma il suo valore può essere **incrementato solo a 1** e può essere sbloccato **unicamente da chi ha posto il semaforo in condizione di blocco**. Un normale semaforo può essere posto in blocco da diversi locker in modo concorrente e potrebbe non richiedere che il rilascio venga effettuato dallo stesso thread che lo ha bloccato.

L'esempio seguente dimostra l'utilizzo dei semafori per risolvere il problema delle sezioni critiche mediante mutua esclusione realizzata mediante un mutex.

int sem_init(sem_t *sem, int pshared, unsigned int value);
sem_init() è utilizzata per inizializzare il valore del semaforo. L'argomento pshared deve essere settato a 0 per semafori locali ai processi.

int sem_wait(sem_t * sem);
sem_wait() effettua il blocco del semaforo

int sem_post(sem_t * sem);
sem_post() rilascia il semaforo

int sem_destroy(sem_t * sem);
sem_destroy() dealloca in modo appropriato le risorse assegnate al semaforo.

Thread
(**mutua esclusione con semafori**)
Esempio **MUTEX**
sem-example.c (1/2)

```
/* Includes */
#include <unistd.h>     /* Symbolic Constants */
#include <sys/types.h>  /* Primitive System Data Types */
#include <errno.h>      /* Errors */
#include <stdio.h>      /* Input/Output */
#include <stdlib.h>     /* General Utilities */
#include <pthread.h>    /* POSIX Threads */
#include <string.h>     /* String handling */
#include <semaphore.h>  /* Semaphore */

/* prototype for thread routine */
void handler ( void *ptr );

/* global vars */
/* semaphores are declared global so they can be accessed in main() and in thread routine,here, the semaphore is used as a
mutex */
sem_t mutex;
int counter; /* shared variable */

int main()
{
    int i[2];
    pthread_t thread_a;
    pthread_t thread_b;

    i[0] = 0; /* argument to threads */
    i[1] = 1;

    sem_init(&mutex, 0, 1);      /* initialize mutex to 1 - binary semaphore */
                                 /* second param = 0 - semaphore is local */

    /* Note: you can check if thread has been successfully created by checking return value of pthread_create */

    pthread_create (&thread_a, NULL, (void *) &handler, (void *) &i[0]);
    pthread_create (&thread_b, NULL, (void *) &handler, (void *) &i[1]);

    pthread_join(thread_a, NULL);
    pthread_join(thread_b, NULL);

    sem_destroy(&mutex); /* destroy semaphore */

    Exit(0);      /* exit */
} /* main() */
```

Inizializzazione semaforo

distruzione semaforo

```
void handler ( void *ptr )
{
    int x;
    x = *((int *) ptr);
    printf("Thread %d: Waiting to enter critical region...\n", x);
    sem_wait(&mutex);       /* down semaphore */
    /* START CRITICAL REGION */
    printf("Thread %d: Now in critical region...\n", x);
    printf("Thread %d: Counter Value: %d\n", x, counter);
    printf("Thread %d: Incrementing Counter...\n", x);
    counter++;
    printf("Thread %d: New Counter Value: %d\n", x, counter);
    printf("Thread %d: Exiting critical region...\n", x);
    /* END CRITICAL REGION */
    sem_post(&mutex);       /* up semaphore */

    pthread_exit(0); /* exit thread */
}
```

Thread
(**mutua esclusione semafori**)
sem-example.c (2/2)

$ gcc -lpthread -o thsm sem-example.c

```
user@:/mnt/persistence/SOLABL2_lez5$ ./thsm
Thread 1: Waiting to enter critical region...
Thread 0: Waiting to enter critical region...
Thread 0: Now in critical region...
Thread 0: Counter Value: 0
Thread 0: Incrementing Counter...
Thread 0: New Counter Value: 1
Thread 0: Exiting critical region...
Thread 1: Now in critical region...
Thread 1: Counter Value: 1
Thread 1: Incrementing Counter...
Thread 1: New Counter Value: 2
Thread 1: Exiting critical region...
user@:/mnt/persistence/SOLABL2_lez5$ _
```

Variabili di condizione

- Le **Variabili di condizione** forniscono un altro meccanismo di sincronizzazione dei thread.
- **Mutex e semafori** realizzano la sincronizzazione mediante controllo dell'accesso ai dati
- Le variabili di condizione permettono ai thread di sincronizzarsi sulla base del **valore** dei dati.

Variabili di condizione

- In assenza di variabili di condizione I thread dovrebbero continuamente testare il valore di variabili globali (magari quando sono in sezione critica) fino al verificarsi di una data condizione

- Questo può consumare parecchie risorse

- Una variabile di condizione fornisce la possibilità di ottenere il medesimo risultato senza dover testare continuamente il valore di variabili globali.

- Una variabile di condizione è **SEMPRE** utilizzata in congiunzione con un mutex lock

Variabili di condizione

Main Thread

- Declare and initialize global data/variables which require synchronization (such as "count")
- Declare and initialize a condition variable object
- Declare and initialize an associated mutex
- Create threads A and B to do work

Thread A

- Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
- Lock associated mutex and check value of a global variable
- Call `pthread_cond_wait()` to perform a blocking wait for signal from Thread-B. Note that a call to `pthread_cond_wait()` automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.
- When signalled, wake up. Mutex is automatically and atomically locked.
- Explicitly unlock mutex
- Continue

Thread B

- Do work
- Lock associated mutex
- Change the value of the global variable that Thread-A is waiting upon.
- Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.
- Unlock mutex.
- Continue

Main Thread

Join / Continue

Variabili di condizione : funzioni (1)

int **pthread_cond_init**(**pthread_cond_t** ***cond**, **pthread_condattr_t** ***attr**)

Inizializza variabile di condizione cond con attributi definiti da attr
Al posto di attr si può passare NULL (valori di di default)

int **pthread_cond_destroy**(**pthread_cond_t** ***cond**)

Dealloca ogni risorsa associata con la variabile di condizione cond

int **pthread_condattr_init** (**pthread_condattr_t** ***attr**)

Inizializza la variabile attr (attributi cond) con valori di default

int **pthread_condattr_destroy** (**pthread_condattr_t** ***attr**)

Dealloca attr (nb. deallocare solo **dopo** che cond è stata inizializzata)

Variabili di condizione : funzioni (2)

int ***pthread_cond_wait(pthread_cond_t ****condition,* ***pthread_mutex_t ****m)*

Mette in attesa di un segnale nella variabile di condizione il thread chiamante. Deve essere chiamata solo quando il mutex associato è bloccato e lo sblocca (il mutex) al momento del blocco del thread. Sblocca il mutex anche nel caso in cui il segnale sia stato ricevuto.

int ***pthread_cond_signal(pthread_cond_t ****condition)*

Invia un segnale ad un thread che è bloccato in attesa del verificarsi di una condizione. Deve essere chiamata dopo che il mutex associato è in stato di blocco ed il mutex deve essere sbloccato dopo che il segnale è stato inviato.

int ***pthread_cond_broadcast*** *(****pthread_cond ****condition)*

Simile a pthread_cond_signal ma invia il segnale a tutti i thread che seguono questa variabile di condizione.