Sistemi Operativi

Lezione 5
Threads e Concorrenza

I Thread

Processi

- Con il termine processo si denota l'esecuzione di un programma (sequenza di istruzioni) nell'ambito di un determinato ambiente esecutivo caratterizzato da:
  - CPU
  - Memoria
  - File
- I processi sono tra loro scorrelati, cioè le risorse di ciascun processo sono private

Thread

- Sequenze di esecuzione diverse all'interno dello stesso ambiente esecutivo (processo)
- Un processo può essere:
  - Single threaded: nozione di processo vista sinora
  - Multiple threaded: all'interno dello stesso processo coesistono più sequenze di istruzioni indipendenti che competono per poter essere eseguite ciascuna di queste è chiamata thread

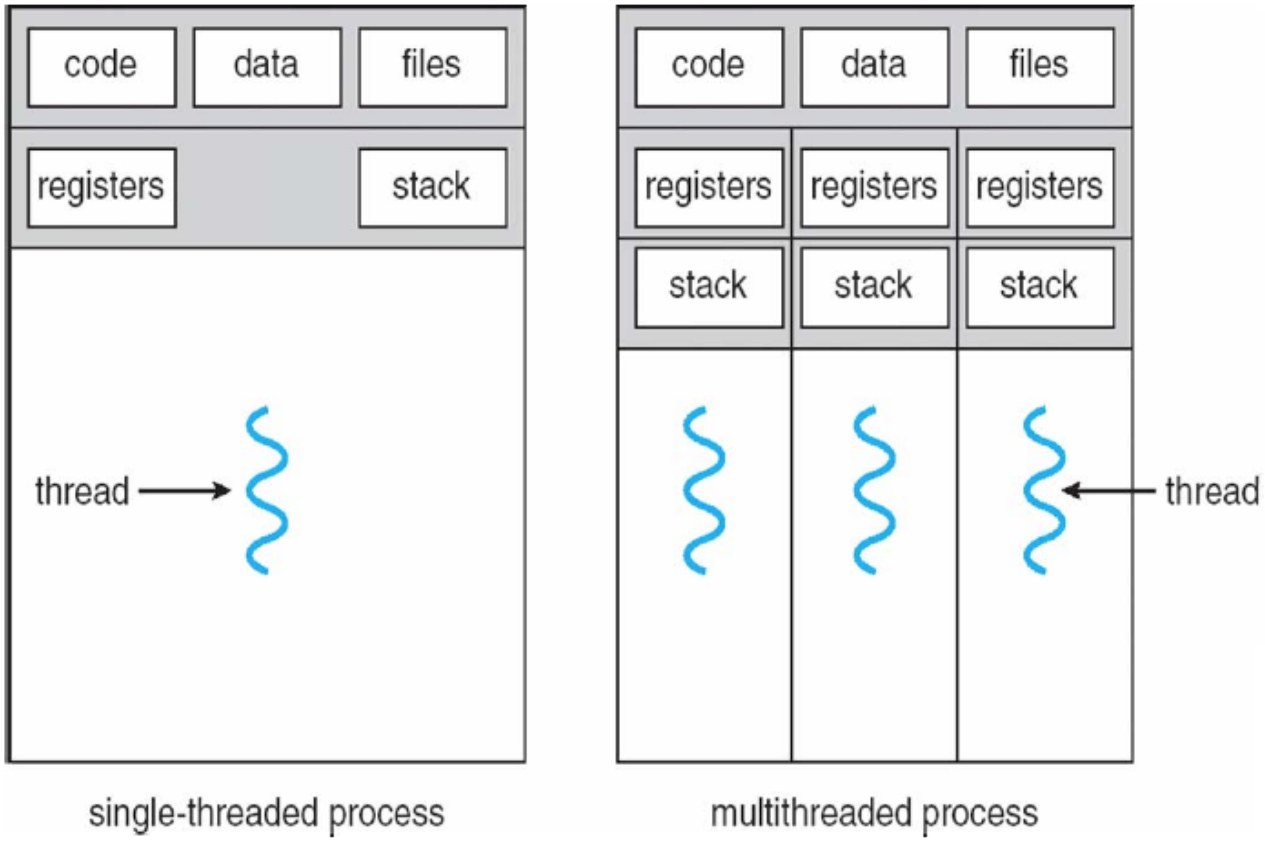
Threads esempio

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS   5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}
```

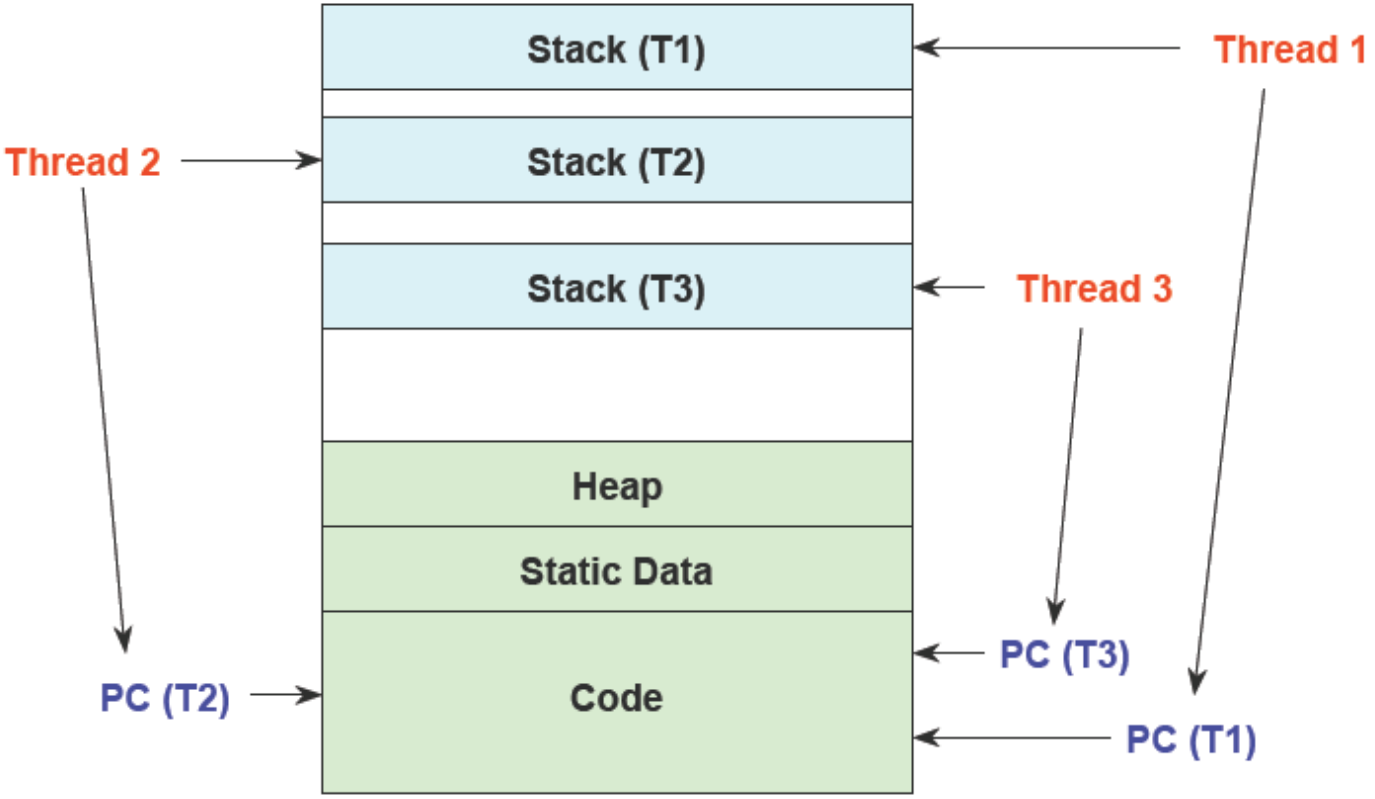
Threads esempio

```
int main(int argc, char *argv[])
{    pthread_t threads[NUM_THREADS];
     int rc;
     long t;

   for(t=0;t<NUM_THREADS;t++)
     {
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
        printf("ERROR; return code from pthread_create() is %d\n", rc);
        exit(-1);
        }
      }
     …
```

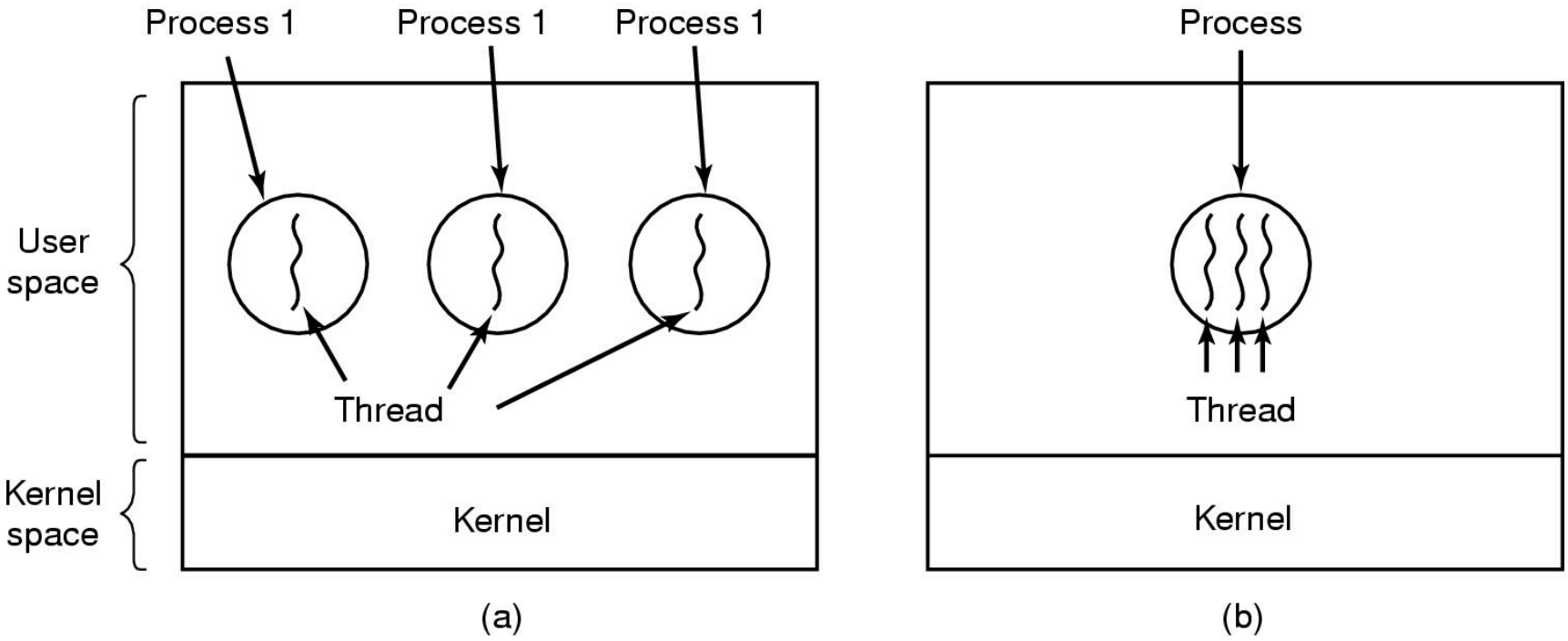
Thread vs. process



Thread e Processo



Thread: il modello (1)



Cosa condividono i thread

- Ogni thread può accedere a ciascun indirizzo all'interno dello spazio di indirizzamento del processo a cui appartiene, non c'è alcun tipo di protezione tra thread
- Un thread può essere in uno qualunque degli stati di un processo
- Il diagramma degli stati dei thread è uguale a quello dei processi

Cosa condividono i thread

- I thread all'interno di un processo condividono:
  - Process ID (PID)
  - Spazio degli indirizzi
    - Codice (istruzioni)
    - Dati non locali
  - Open file descriptors
  - Signals and signal handlers
  - Current working directory
  - User and group id

Cosa non condividono

- Thread ID (TID)
- L'insieme dei registri, compresi EIP (Program counter) e Stack pointer
- Stack per le variabili locali e record di attivazione
- Maschera signal

Schematicamente

Per process items	**Per thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Perché usare i thread

- Rendono più facile la realizzazione di applicazioni parallele (concorrenti) necessarie per:
  - la gestione di eventi concorrenti
  - l'uso efficiente di sistemi multi processing
  - L'overlapping di I/O con computazione
- Migliorano la struttura del programma

Operazioni sui Thread

- Tre sistemi (librerie) principali
  - Win32 threads
  - C-Threads
  - POSIX **Pthreads**
- Operazioni comuni
  - Create
  - Exit
  - Suspend
  - Resume
  - Sleep
  - Wake
  - Join (invece di wait())
- Le operazioni sui thread sono di solito molto più veloci che le corrispondenti operazioni sui processi

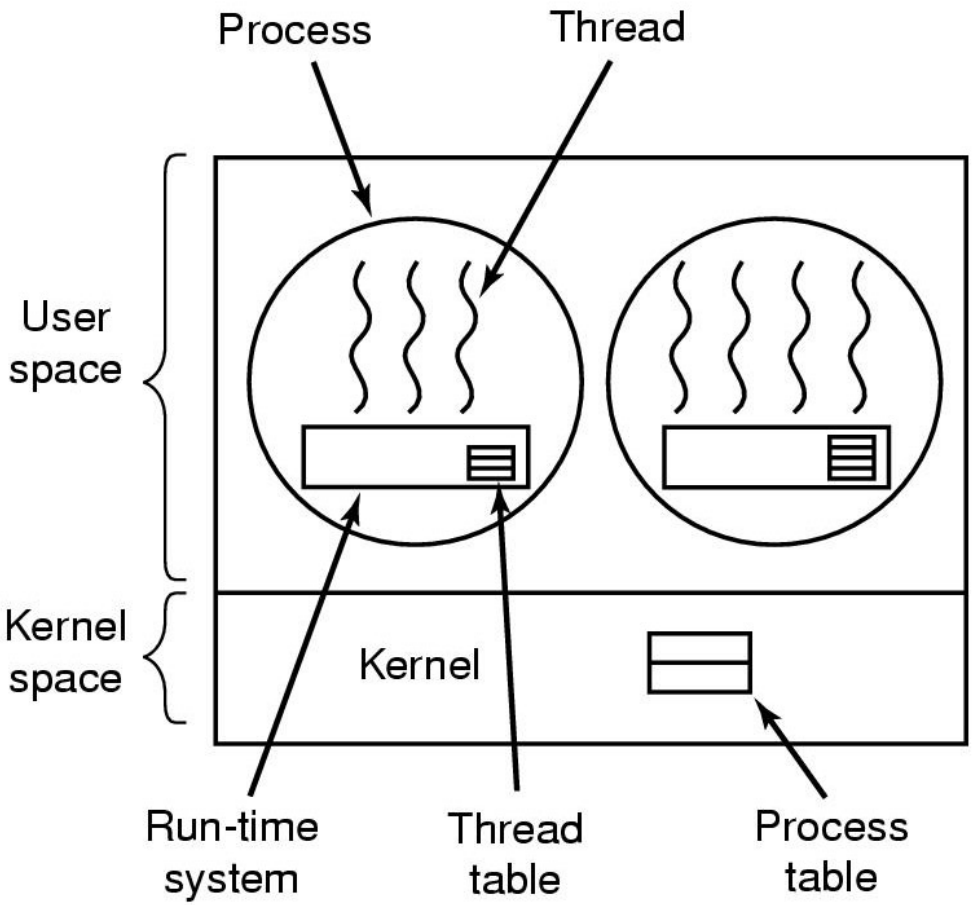
Thread Scheduling

- Abbiamo visto che lo scheduler opera sui PCB, nel caso di sistema a thread cosa succede?
- Due opzioni:
  - I thread diventano l'unità di scheduling per il sistema operativo, che adotta gli algoritmi già visti
  - Il sistema operativo continua ad operare sui PCB, e scarica su uno scheduler a livello user il compito di gestire i thread

User Level Thread

- Implementati attraverso librerie a user-level
  - Creazione, scheduling, sincronizzazione thread
  - OS ignora la presenza dei thread
  - OS vede solo i processi con un singolo thread
- Vantaggi
  - Non è richiesto alcun supporto da parte del SO → portabilità
  - Possono essere predisposte politiche di scheduling adeguate all'applicazione
  - Le operazioni su thread sono efficienti perché non richiedono l'esecuzione di syscall
- Svantaggi
  - Non può sfruttare sistemi multiprocessore
  - L'intero processo si blocca quando si blocca un solo thread

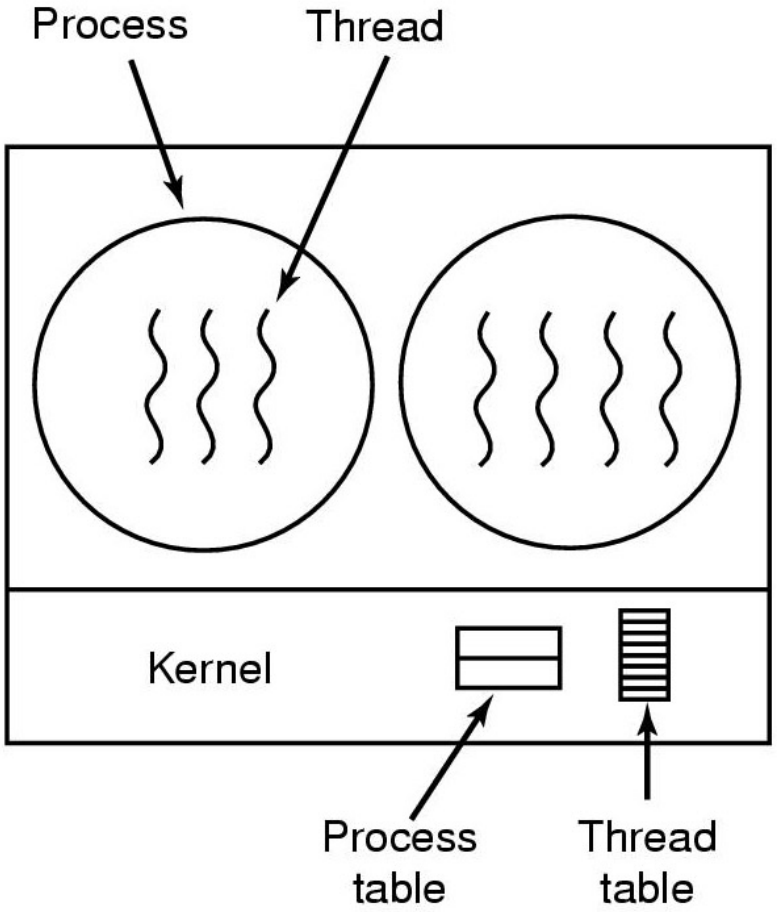
Thread in User Space



Kernel level thread

- Kernel-level threads:
  - SO associa ad ogni thread a livello user un kernel thread
  - Ogni kernel thread viene schedulato indipendentemente
  - Tutte le operazioni sui thread sono eseguite dal SO operations (creazione, scheduling, sincronizzazione)
- Vantaggi
  - Ogni kernel-level thread può essere eseguito in parallelo su un sistema multiprocessore
  - Quando un thread si blocca, altri thread appartenenti allo stesso processo possono essere eseguiti
- Svantaggi
  - Operazioni sui thread più costose in termini di tempo
  - Il SO “must scale well” con un numero crescente di thread

Thread a livello Kernel



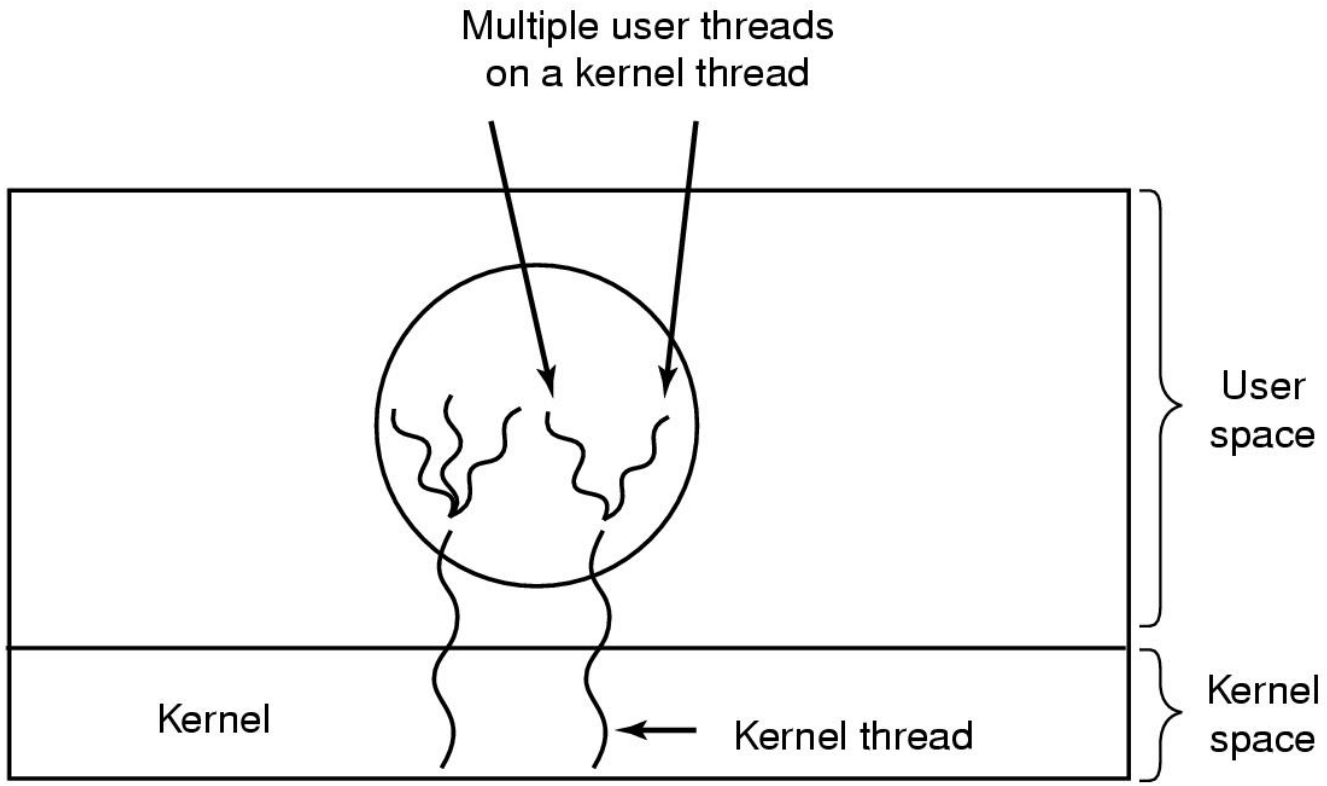
Kernel vs User

- Kernel-level
  - Integrati con OS
  - Operazioni di creazione, gestione, comunicazione lente
  - Ottimo sfruttamento parallelismo
- User-level
  - Veloci da creare, gestire e manipolare
  - Non integrati con OS, scarso sfruttamento parallelismo

Thread Modello ibrido

- Modello ibrido basato su Kernel e user-level threads
  - L' applicazione crea m threads
  - Il SO fornisce un **pool** di n kernel thread
  - Ad un kernel thread sono solitamente associati più user-level thread
- Vantaggi
  - Strategia per ottenere il meglio dalle implemenentazioni user-level e kernel-level
  - Funziona bene con molti user threads di breve durata, suddivisi in gruppi di dimensione costante
- Svantaggi
  - Complicato …
  - Selezione del mapping tra le due tipologie di thread
  - Determinazione del numero ottimale di kernel thread
    - Definito dall'utente
    - Il SO aggiusta in numero dinamicamente in funzione del carico

Hybrid Implementations



Interprocess Communication (IPC)

- Per poter cooperare fattivamente, thread e/o processi devono poter comunicare tra loro
- Esistono tre meccanismi principali per lo svolgimento di questa attività
  - Shared Memory
  - Message Passing
  - Signals

IPC: Shared Memory

- Processi
  - Ogni processo dispone di una spazio privato di indirizzamento
  - Per poter comunicare deve esplicitamente predisporre un segmento di memoria condivisa
- Thread
  - Condividono “per costruzione” lo spazio di memoria del processo
  - Vantaggi
    - La condivisione di memoria è facile e veloce
  - Svantaggi
    - Gli accessi ai dati condivisi devono essere opportunamente sincronizzati **per evitare errori**

IPC: Message Passing

- Usato principalmente tra processi
  - I dati sono trasmessi esplicitamente da un processo **sender** (src) a un processo **receiver** (destination)
  - Esempio: Unix pipes
- Vantaggi:
  - La condivisione dei dati è esplicita -> più facile individuare i legami trai vari moduli
  - Non è necessario un rapporto di fiducia tra sender e receiver
- Svantaggi:
  - Performance overhead per la copia dei messaggi
- Problematiche:
  - Come identificare mittente e destinatario?
    - Processo, insieme di processes, o mailbox (port)
  - Il processo mittente deve bloccarsi in attesa del destinatario?
    - Blocking: rallenta il mittente
    - Non-blocking: richiede buffering nei due processi

IPC: Signal

- Signal
  - Interrupt Software che notifica il verificarsi di un evento ad un processo
  - Esempi: SIGFPE, SIGKILL, SIGUSR1, SIGSTOP, SIGCONT
- Quando un processo riceve una signal può svolgere una delle seguenti azioni:
  - Catch: specificare la routine (signal handler) per la sua gestione
  - Ignore: basarsi su azioni standard previste dal SO
    - Abort, memory dump, suspensione
  - Mask: bloccare il segnale affinchè non sia consegnato
- Svantaggi
  - Non possono essere comunicati dati
  - Sono a livello processo, quindi difficili da gestire con i thread

Thread e Signal

- Problema: in un sistema multi threaded a quale thread di un processo il SO consegna la signal?
- Opzione 1: il sender specifica il thread id del destinatario
  - Il Sender può non conoscere i thread che compongono un processo
- Opzione 2: il SO sceglie il thread di destinazione
  - POSIX: ogni thread ha una signal mask (serve per disabilitare specifici signal)
  - Il SO consegna un signal a tutti i thread che non lo hanno disabilitato

INTRODUZIONE ALLA CONCORRENZA

Concorrenza

- Due o più attività sono concorrenti se la loro esecuzione è sovrapposta nel tempo
- In un sistema multiprogrammato (anche se monoprocessore) esistono generalmente più attività concorrenti
- Siccome non è possibile stabilire a priori come evolve l'esecuzione di attività concorrenti è necessario garantire che i risultati delle loro computazioni, non dipendano dalle sequenze temporali di esecuzione delle proprie istruzioni

Problemi

- Attività concorrenti possono “competere” nell’accedere alla stessa risorsa, in questi casi si può verificare una race condition
- Condizioni di corsa, o race condition
  - Due o più attività leggono e scrivono dati condivisi
  - I risultati finali (il valore dei dati) non sono sempre gli stessi ma dipendono dalla particolare sequenza di esecuzione (interleaving)
    - Comportamento non deterministico

Esempio #1 (1)

```
main() {
struct conto {
        int codice;
        char rag_sociale [120];
        float saldo; };
int codice; float importo;
file *conti;
struct conto cc;
while TRUE {
  scanf ("%d %f", &codice, &importo);
  conti = fopen("conticorrenti", "update");
  fread(&cc,sizeof(struct conto),codice, conti);
  cc.saldo = cc.saldo + importo;
  fwrite(&cc,sizeof(struct conto),codicecliente,conti);  }
}
```

Esempio #1 (2)

```
main() {
struct conto {
        int codice;
        char rag_sociale [120];
        float saldo; };
int codice; float importo;
file *conti;
struct conto cc;

while TRUE {
  scanf (“%d %f”, &codice, &importo);
  conti = fopen(“conticorrenti”, “update”);
  fread(&cc,sizeof(struct conto),codice, conti);
  cc.saldo = cc.saldo - importo;
  fwrite(&cc,sizeof(struct conto),codicecliente,conti); }
}
```

Esempio #1

- Siano P1 e P2 i processi associati ai due precedenti programmi
- Cosa succede se i due processi sono eseguiti in modo sequenziale e in modo concorrente, assumendo che P1 e P2 vengano eseguiti sullo stesso record, con i seguenti dati
  - Conto.saldo vale 1000 euro
  - Importo di addebito 200 euro
  - Importo di accredito 700 euro

Esempio #2

- Si considerino 2 thread che operano sulle seguenti variabili condivise

```
int    buffer[100];
int    counter; /* n.ro di elementi presenti
                   in  buffer*/
```

Esempio #2

```
produci()
{
int in, elem;
begin
Produci dato e ponilo in
   elem;
 while (counter == n) {};
 buffer[in]= elem;
 in= (in+1) mod n;
 counter = counter+1;
}
```

```
Consuma ()
{
int out,elem;
begin
 while (counter == 0){};
 elem =buffer[out];
 out = (out+1) mod n;
 counter = counter-1;
 Consuma dato in elem;
}
```

Esempio #2

- Si consideri l'esecuzione sequenziale e concorrente di questi thread
- Si ricorda che un'istruzione di incremento corrisponde alle seguenti istruzioni assembly

```
LW  $5, variabile
ADDI $5,1
SW $5, variabile
```

La sezione critica

- Ogni volta che due o più attività eseguite concorrentemente accedono ad una variabile condivisa si possono creare delle race condition
- La porzione di codice in cui questa condizione si può verificare è detta SEZIONE CRITICA

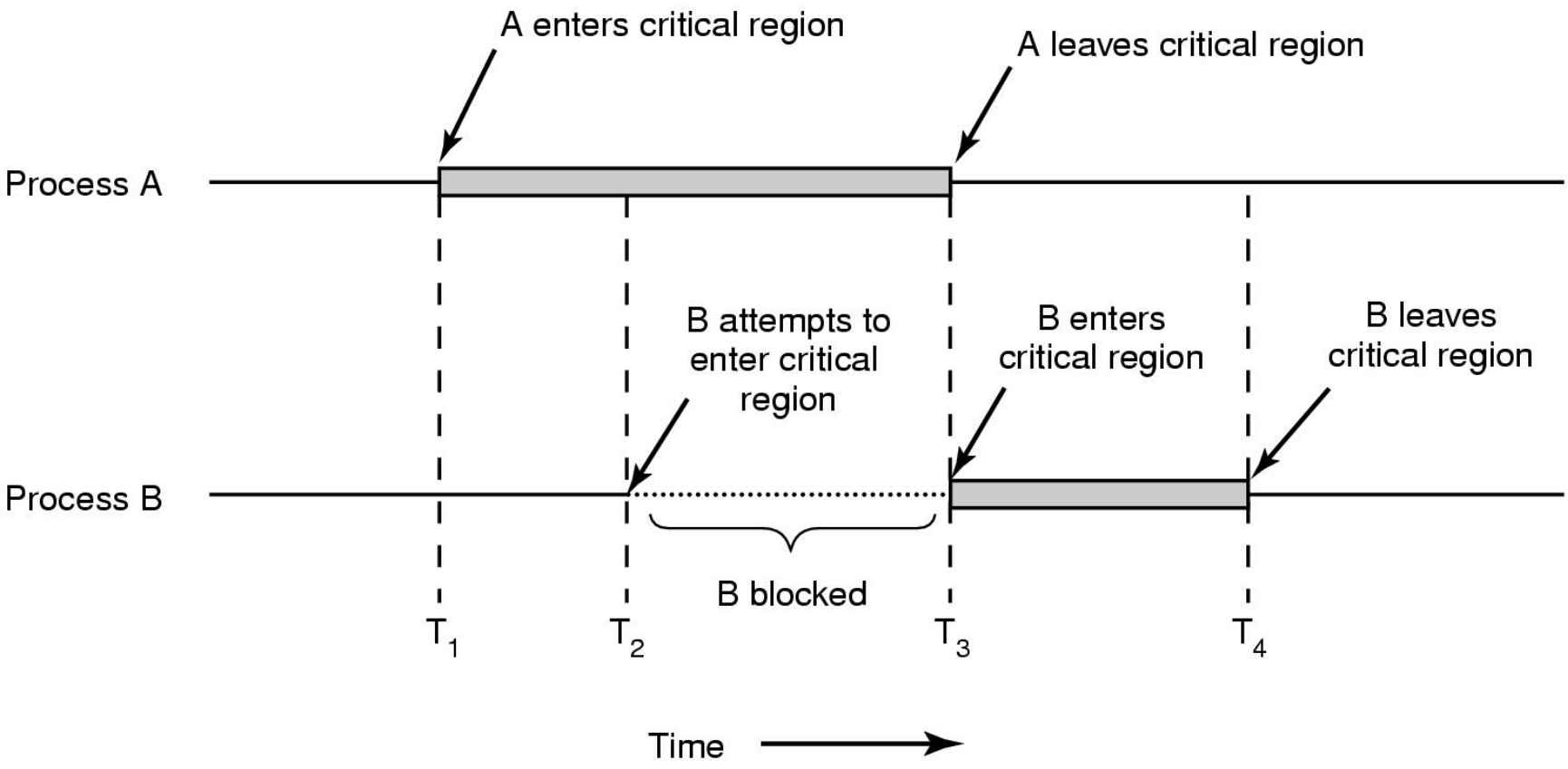
La sezione critica

- Per evitare race condition è necessario che attività concorrenti che condividono una variabile o più in generale una qualunque risorsa, e sono in grado di modificarne lo stato, vi accedano in MUTUA ESCLUSIONE
- Vanno individuati i meccanismi per garantire l'accesso in mutua esclusione alle risorse

Mutua esclusione

- Questi meccanismi devono essere in grado di garantire contemporaneamente il rispetto delle seguenti proprietà:
  - Nessuna coppia di attività può trovarsi simultaneamente in una sezione critica condivisa
  - L'accesso alla regione critica non è regolato da alcuna assunzione temporale
  - Nessuna attività che sta eseguendo codice al di fuori della regione critica può bloccare un processo interessato ad entrarvi
  - Nessun processo deve attendere indefinitamente per poter accedere alla regione critica

Mutua esclusione



L'esecuzione che vorremmo

Mutua esclusione

- Problema: individuare un algoritmo per garantire che più attività concorrenti accedano alle proprie sezioni critiche in mutua esclusione
- In genere le soluzioni adottate richiedono che per accedere ad una sezione critica si svolgano una serie di azioni preliminari

```
non_critical_section
ENTER_REGION
Sezione_critica
LEAVE_REGION
non_critical_section
```

Mutua esclusione con busy waiting

- Soluzione generale: si fa aspettare chi deve entrare quando la sezione critica è già occupata, valutando continuamente una variabile
  - Busy waiting

Mutua esclusione con busy waiting (2)

Stretta alternanza

(a) Processo 0

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

(b) Processo 1

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Mutua esclusione con busy waiting (2)

- Problema quando uno dei due processi è molto più lento dell'altro
  - Viola la condizione che un processo al di fuori della sezione critica non deve poter impedire ad un altro di entrarci, se la sezione critica è libera

Mutua esclusione con busy waiting (3)

```
#define TRUE 1
#define FALSE 0
#define N 2
int interested[N];

void enter_region(int process);
 {
   int other;  /* numero dell'altro processo */
   other = 1 - process;
   interested[process] = TRUE;
   while (interested[other] == TRUE);
}
SEZIONE CRITICA

void leave_region (int process);
{
  interested[process] = FALSE;
}
```

Mutua esclusione con busy waiting (3)

- Con la soluzione proposta si può verificare la situazione in cui ciascun processo è in attesa di un evento
  - interested[other] diventa FALSE

  che può accedere solo grazie all'altro processo
  - esecuzione di leave_region
- Fenomeno degenere dei sistemi concorrenti noto come deadlock

Mutua esclusione con busy waiting (4)

- Modifichiamo la soluzione precedente per evitare il deadlock
  - interested[process] deve cambiare
  - Invece di aspettare a vuoto, ciascun processo modifica la propria variabile interested[process] nella speranza che ciò possa sbloccare la situazione

Mutua esclusione con busy waiting (4)

```
#define TRUE 1
#define FALSE 0
#define N 2
int interested[N];

void enter_region(int process);
 {
   int other;    /* numero dell'altro processo */
   other = 1 - process;
   interested[process] = TRUE;
   while (interested[other] == TRUE) {
        interested[process] = FALSE;
        interested[process] = TRUE;}
}
void leave_region (int process);
{
  interested[process] = FALSE;
}
```

Mutua esclusione con busy waiting (4)

- Si consideri il seguente interleaving:
  1. P0 pone interested[0] a TRUE
  2. P1 pone interested[1] a TRUE
  3. P1 verifica interested[0] e pone interested[1] a FALSE
  4. P0 accede alla sezione critica
  5. P0 chiede di accedere di nuovo alla sezione critica
  6. P1 pone interested[1] a TRUE
  7. Torna al passo 3

Mutua esclusione con busy waiting (4)

- La sequenza appena descritta può ripetersi indefinitamente
- P0 continua ad eseguire la propria sezione critica
- P1 non riesce ad accedervi
- Non c'è deadlock, ma c'è starvation di P1
  - Si viola la quarta condizione

Algoritmo di Peterson

```
#define FALSE  0
#define TRUE   1
#define N      2                    /* number of processes */

int turn;                           /* whose turn is it? */
int interested[N];                  /* all values initially 0 (FALSE) */

void enter_region(int process);     /* process is 0 or 1 */
{
      int other;                    /* number of the other process */

      other = 1 – process;          /* the opposite of process */
      interested[process] = TRUE;   /* show that you are interested */
      turn = process;               /* set flag */
      while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)      /* process: who is leaving */
{
      interested[process] = FALSE;  /* indicate departure from critical region */
}
```