Processi e Thread

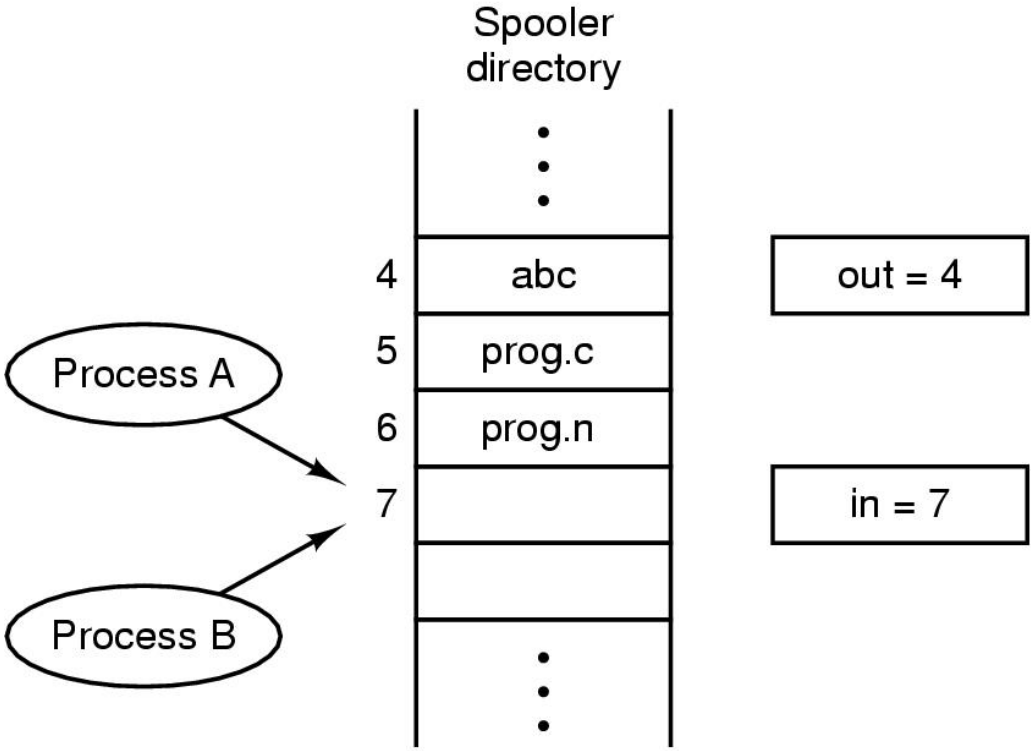
Meccanismi di IPC (1)

Comunicazioni fra processi/thread

- Processi/thread eseguiti concorrentemente hanno bisogno di interagire per comunicare e sincronizzarsi :
  - scambiare dati
  - utilizzare correttamente strutture dati condivise
  - eseguire azioni nella sequenza corretta
- Molti meccanismi proposti
- Per semplicità ci riferiremo principalmente ai processi
  - maccanismi simili sono disponibili per i thread

Comunicazioni fra processi

Race Condition (Interferenza)



Due processi accedono alla memoria condivisa contemporaneamente

l'esito dipende dall'ordine in cui vengono eseguiti gli accessi

Sezioni/Regioni Critiche (1)

Regione Critica : porzione di un processo che accede a strutture dati condivise

- punti potenziali di interferenza

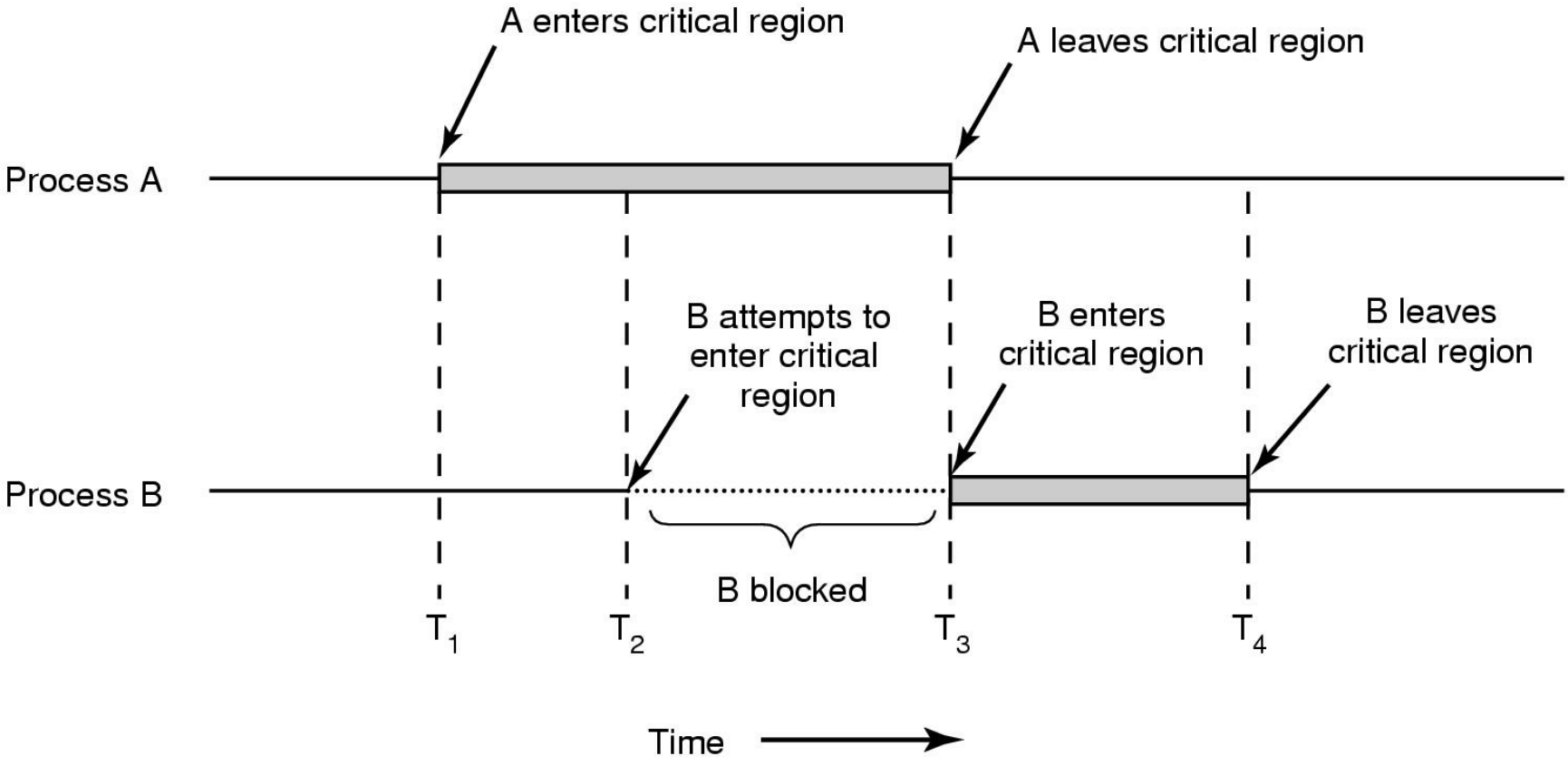
Obiettivo : fare in modo che le regioni critiche di due processi non vengano mai eseguite contemporaneamente (mutua esclusione)

Sezioni/Regioni Critiche (2)

4 condizioni per assicurare la mutua esclusione

- un solo processo per volta esegue la sezione critica
- non viene fatta nessuna assunzione sulla velocità relativa dei processi
- nessun processo che sta eseguendo codice esterno alla sezione critica può bloccare un altro processo
- nessun processo attende indefinitamente di entrare nella sezione critica

Sezioni/Regioni Critiche (3)



Mutua esclusione con sezioni critiche

Mutua Esclusione : soluzioni hw

- Disabilitare le interruzioni
  - impedisce che un altro processo vada in esecuzione
  - non utilizzabile in modo utente
  - utilizzabile per poche istruzioni in modo kernel
  - non risolve il problema se il sistema ha più di una CPU

Mutua Esclusione con attesa attiva (*busy waiting*) (1)

- Soluzioni software
  - alternanza stretta
  - soluzione di Peterson
- Soluzioni hardware-software
  - l'istruzione TSL

Mutua Esclusione con attesa attiva (2)

Alternanza stretta

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

(a)

Processo 0

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

(b)

Processo 1

Una soluzione non soddisfacente per il problema della ME

- 0 può bloccare 1 quando si trova fuori dalla SC

Mutua Esclusione con attesa attiva (3)

```
#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 */
}
```

Soluzione di Peterson (semplificata)

Mutua Esclusione con attesa attiva (4)

- Istruzione assembler Test and Set Lock (TSL)
  - eseguibile in modalità utente
- es: TSL R1, X
  - esegue due accessi indivisibili alla memoria
  - 1) copia il contenuto della cella di indirizzo X in R1
  - 2) scrive 1 in X

Mutua Esclusione con attesa attiva (5)

```
enter_region:
        TSL REGISTER,LOCK                | copy lock to register and set lock to 1
        CMP REGISTER,#0                  | was lock zero?
        JNE enter_region                 | if it was non zero, lock was set, so loop
        RET | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0                     | store a 0 in lock
        RET | return to caller
```

Ingresso ed uscita dalla sezione critica utilizzando l'istruzione TSL

Soluzioni senza attesa attiva
Le primitive *Sleep* e *Wakeup* (1)

- Idea di base : un processo viene bloccato finché non è in grado di entrare nella sezione critica (in modo da non sprecare cicli di CPU)
- Due primitive realizzate come system call
  - sleep() :: blocca il processo che la invoca
  - wakeup(P) :: sveglia il processo P

- Esempio : il problema del produttore e del consumatore

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
      int item;

      while (TRUE) {                            /* repeat forever */
            item = produce_item( );             /* generate next item */
            if (count == N) sleep( );           /* if buffer is full, go to sleep */
            insert_item(item);                  /* put item in buffer */
            count = count + 1;                  /* increment count of items in buffer */
            if (count == 1) wakeup(consumer);   /* was buffer empty? */
      }
}


void consumer(void)
{
      int item;

      while (TRUE) {                            /* repeat forever */
            if (count == 0) sleep( );           /* if buffer is empty, got to sleep */
            item = remove_item( );              /* take item out of buffer */
            count = count – 1;                  /* decrement count of items in buffer */
            if (count == N – 1) wakeup(producer);  /* was buffer full? */
            consume_item(item);                 /* print item */
      }
}
```

Sol. Errata al problema del produttore–consumatore (con race condition)

Semafori (1)

- Problema con *sleep* e *wakeup* : una *wakeup* non utilizzata immediatamente viene persa
- Semafori : variabili intere
  - contano quanti eventi si sono verificati (es. Wakeup)
  - il valore è 0 se non ci sono eventi pendenti e > 0 altrimenti
- Due operazioni atomiche standard *Up* e *Down* (P e V)
  - down(S)
    - se S > 0 allora S= S - 1 ed il processo continua l'esecuzione
    - se S==0 ed il processo si blocca senza completare la primitiva
  - up(S)
    - se ci sono processi in attesa di completare la down su quel semaforo (e quindi necessariamente S == 0) uno di questi viene svegliato e S rimane a 0, altrimenti S viene incrementato;
    - in caso contrario (S > 0), allora S=S + 1

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );               /* generate something to put in buffer */
          down(&empty);                         /* decrement empty count */
          down(&mutex);                         /* enter critical region */
          insert_item(item);                    /* put new item in buffer */
          up(&mutex);                           /* leave critical region */
          up(&full);                            /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                          /* decrement full count */
          down(&mutex);                         /* enter critical region */
          item = remove_item( );                /* take item from buffer */
          up(&mutex);                           /* leave critical region */
          up(&empty);                           /* increment count of empty slots */
          consume_item(item);                   /* do something with the item */
     }
}
```

Mutex (1)

- Semaforo con solo due stati *aperto* (*unlocked*) o *chiuso* (*locked*)
- Popolare nei thread user-level
- Due primitive :
  - mutex_lock(), corrisponde alla down()
  - mutex_unlock(), corrisponde alla up()
- Utilizzato per realizzare sezioni critiche su dati condivisi
- Puo essere implementato efficientemente senza passare in stato kernel (se è disponibile la TSL)

Mutex (2)

```
mutex_lock:
        TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
        CMP REGISTER,#0             | was mutex zero?
        JZE ok                      | if it was zero, mutex was unlocked, so return
        CALL thread_yield           | mutex is busy; schedule another thread
        JMP mutex_lock              | try again later
ok:     RET | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0               | store a 0 in mutex
        RET | return to caller
```

Implementazione delle primitive di *mutex_lock* e *mutex_unlock*