

# Sistemi Operativi

Lezione 5-6

La comunicazione tra processi

# Introduzione

- I processi devono poter comunicare tra loro in modo strutturato e preciso
  - Per passarsi informazioni
  - Per non disturbarsi reciprocamente quando svolgono attività critiche
  - Per ordinare correttamente le reciproche esecuzioni in presenza di dipendenze
- Problema analogo con i thread tranne il primo caso
  - Condividono la memoria

# Processi concorrenti

- Due o più processi sono concorrenti se la loro esecuzione è sovrapposta nel tempo
- In un sistema monoprocesso l'unico modo per eseguire in modo "concorrente" dei processi è quello di eseguire un interleaving delle istruzioni del loro codice
- In particolare non potendo fare alcuna assunzione temporale, un sistema che supporta l'esecuzione concorrente di processi deve garantire la sua correttezza indipendentemente dall'interleaving eseguito

# Processi concorrenti

- Processi concorrenti possono avere la necessità di interagire tra loro per poter evolvere
  - Comunicazione
    - due o più processi si scambiano informazioni
  - Contesa
    - due o più processi competono per l'uso della stessa risorsa
  - Cooperazione
    - due o più processi collaborano alla soluzione di uno stesso problema

# Problemi

- Condizioni di corsa, o race conditions
  - Due o più processi leggono e scrivono dati condivisi
  - I risultati finali dipendono dalla particolare sequenza di esecuzione (interleaving)
    - Trovare l'errore è difficile perché si manifesta solo in presenza di particolari temporizzazioni dell'esecuzione

# Esempio #1 (1)

```
Program accredito (nro_conto:integer; importo:real);
struct conto {
    int codice;
    char rag_sociale [120];
    float saldo; };

int accredito (int codicecliente, float importo){
    FILE *conti;
    struct conto cc;

    conti = fopen("conticorrenti", "update");
    fread(&cc,sizeof(struct conto),codicecliente, conti);
    cc.saldo := cc.saldo + importo;
    fwrite(&cc,sizeof(struct conto),codicecliente,conti);
}
```

## Esempio #1 (2)

```
int addedito (int codicecliente, float importo)
{
    FILE *conti;
    struct conto cc;
    conti = fopen("conticorrenti", "update");
    fread(&cc, sizeof(struct conto), codicecliente,
        conti);
    cc.saldo := cc.saldo - importo;
    fwrite(&cc, sizeof(struct conto), codicecliente,
        conti);
}
```

# Esempio #1

- Siano P1 e P2 i processi associati ai programmi accredito e addebito
- 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 processi 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 considerino più processi, ciascuno associato ad uno ed uno solo dei programmi suddetti
- Si consideri l'esecuzione sequenziale e concorrente di tali processi
- Si ricorda che un'istruzione di incremento corrisponde alle seguenti istruzioni assembly
  - LW \$5, variabile
  - ADDI \$5,1
  - SW \$5, variabile

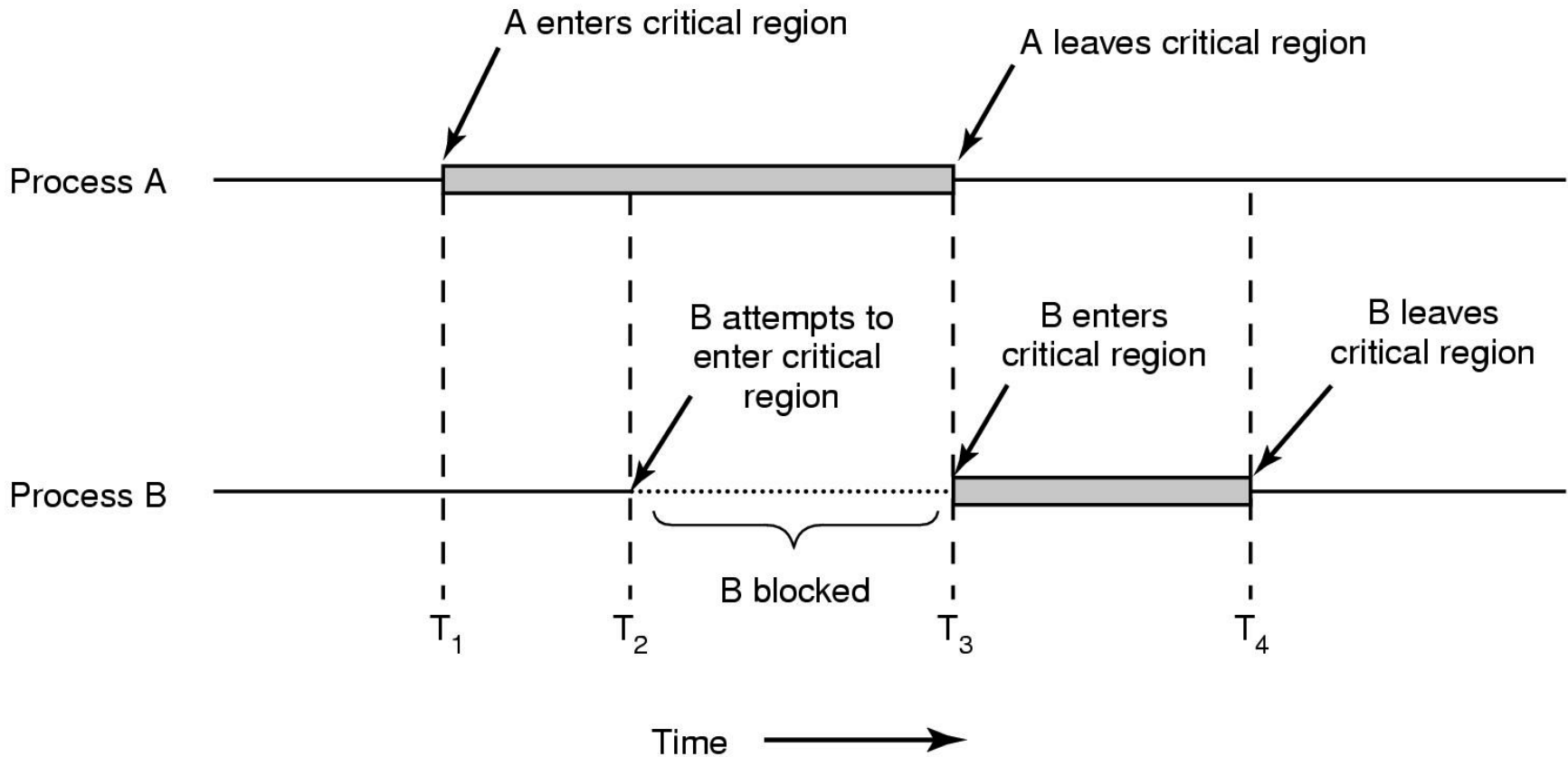
# La sezione critica

- Ogni volta che due o più processi eseguiti concorrentemente accedono ad una variabile condivisa si possono creare dei problemi in esecuzione
- La porzione di codice in cui un processo accede a variabili condivise con altri processi è chiamata **SEZIONE CRITICA**

# La sezione critica

- Per evitare i problemi visti con gli esempi #1, #2 è necessario che processi concorrenti che condividono una variabile o più in generale una qualunque risorsa, vi accedano in **MUTUA ESCLUSIONE**

# Mutua esclusione



L'esecuzione che vorremmo

# Mutua esclusione

- Due o più processi accedono a risorse in mutua esclusione quando soddisfano le seguenti proprietà:
  - Nessuna coppia di processi può trovarsi simultaneamente nella sezione critica
  - L'accesso alla regione critica non è regolato da alcuna assunzione temporale o sul numero di CPU
  - Nessun processo 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

- Problema: quale algoritmo deve essere utilizzato per garantire che più processi concorrenti accedano alle proprie sezioni critiche in mutua esclusione
- In genere le soluzioni adottate richiedono che un processo che vuole accedere alla sezione critica svolga 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

(b) Processo 1

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

(a)

```
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);
}
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 FALSEche può accadere solo grazie all'altro processo
  - esecuzione di leave\_region
- Fenomeno degenerare 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

# Mutua esclusione con busy waiting (5)

```
#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 */
}
```

# Una soluzione alternativa

- Due processi possono trovarsi in sezione critica simultaneamente solo perché chi vi è entrato per primo è stato interrotto e la CPU è stata passata ad un altro processo
- La CPU passa da un processo ad un altro solo in seguito ad un interrupt
- Disabilitando gli interrupt prima di accedere alla sezione critica e riabilitandoli all'uscita, la CPU non può essere passata ad altri
  - un processo può leggere e aggiornare le variabili condivise senza essere interrotto da un altro processo

# Disabilitare gli interrupt

- Si può garantire l'accesso in mutua esclusione ad una sezione critica disabilitando gli interrupt
  - Utile per il kernel per aggiornare le sue variabili
  - Non è raccomandabile dare all'utente il potere di disabilitare gli interrupt
    - Istruzione privilegiata
  - L'utente potrebbe non riabilitarli più e non cedere più la CPU
  - Rallenta la risposta agli eventi
  - Inutile in un sistema multiprocessore
  - Non è strutturata e rende di difficile comprensione il codice

# Soluzioni basate su istruzioni

- La soluzione sw al problema della mutua esclusione è abbastanza complessa
  - Algoritmo di Peterson
- La soluzione hw mediante disabilitazione degli interrupt non è adeguata per programmi utente
  - Codice dal comportamento oscuro, scarsa efficienza, inutile nei sistemi multiprocessore
- Una soluzione migliore può essere ottenuta con un certo supporto da parte del linguaggio macchina del sistema su cui operiamo

# Test and Set Lock

- I primi a notare questa necessità furono i progettisti dell'OS/360
- Aggiunsero nel linguaggio macchina del sistema l'istruzione

TEST AND SET LOCK (TSL)

# TSL

- TSL opera nel seguente modo
  - Sintassi
    - TSL register, flag
  - Semantica
    - `register := flag /* copia in register il valore corrente di flag */`
    - `flag := 1 /* assegna 1 a flag */`

con la garanzia di indivisibilità o atomicità delle due operazioni

- sono eseguite come se si trattasse di un'unica istruzione

# TSL

enter\_region:

    tsl register,lock

    cmp register,#0

    jne enter\_region

    ret

| copy lock to register and set lock to 1

| was lock zero?

| if it was non zero, lock was set, so loop

| return to caller; critical region entered

leave\_region:

    move lock,#0

    ret

| store a 0 in lock

| return to caller

# Osservazioni

- Svantaggi di soluzioni basate su istruzioni speciali
  - Busy wait
  - Starvation possibile a causa della scelta del prossimo processo da eseguire quando uno lascia la sezione critica
  - Deadlock per inversione di priorità
    - se il processo in sezione critica P1 viene interrotto e la CPU assegnata ad un processo P2 con priorità superiore che cerca di entrare, non può a causa di P1 che però non può eseguire avendo priorità inferiore a P2, che è attivo in busy wait

# Semafori

- I semafori sono delle variabili intere
  - condivise tra più processi
  - possono assumere come valore 0 e 1
    - in questo caso parliamo di semafori binari
  - oppure un intero  $\geq 0$ 
    - in questo caso parliamo di semafori generalizzati
- Le operazioni definite sui semafori sono due
  - P(sem) o Down(sem) o wait(sem)
    - la P sta per Proberen (provare)
  - V(sem) o Up(sem) o signal(sem)
    - la V sta per Verhogen (incrementare)
- A queste si aggiunge l'inizializzazione
  - semaforo = valore

# Semafori binari

- Le operazioni Wait(sem) e Signal(sem) hanno la seguente semantica

down (sem) :

```
if (sem == 0) then wait on sem;  
      else sem = 0;
```

up (sem) :

```
if (some process is waiting on sem)  
  then awake it  
  else sem = 1;
```

- Il sistema ne garantisce l'atomicità
  - eseguite come se fossero un'unica istruzione macchina

# Semafori binari

- I processi non attendono più in busy wait
- A ciascun semaforo è associata una coda in cui vengono posti i processi bloccati sul particolare semaforo
- La disciplina di gestione della coda non è specificata nella definizione dei semafori
  - solitamente FIFO
- I processi sono risvegliati dalla coda quando un altro processo esegue un'operazione di up sul semaforo su cui sono bloccati

# Semafori binari

- Con i semafori binari il problema della mutua esclusione tra n processi può essere risolto molto facilmente
- Ogni processo esegue la sequenza

```
down(sem) /* enter_region */
Sezione critica
up(sem) /* leave_region */
```
- Il primo processo che riesce ad eseguire `down(sem)`, entra in regione critica

# Semafori binari

- I semafori binari possono anche essere usati per sincronizzare due o più processi
  - Determinare una specifica sequenza di esecuzione
- Esempio

```
semaphore S1 = 1;
```

```
semaphore S2 = 0;
```

```
semaphore S3 = 0;
```

P1	P2	P3
down (S1)	down (s2)	down (s3)
:	:	:
up (S2)	up (S3)	up (S1)

# I semafori generalizzati

- Le operazioni `down(sem)` e `up(sem)` hanno la seguente semantica

`down(sem) :`

```
if (sem == 0) then wait on sem;  
      else sem = sem - 1;
```

`up(sem) :`

```
if (some process is waiting on sem)  
  then awake it  
  else sem = sem + 1;
```

- Il sistema ne garantisce l'atomicità

# Producer-Consumer Problem

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
} . . .
```

*/\* number of slots in the buffer \*/*  
*/\* semaphores are a special kind of int \*/*  
*/\* controls access to critical region \*/*  
*/\* counts empty buffer slots \*/*  
*/\* counts full buffer slots \*/*

*/\* TRUE is the constant 1 \*/*  
*/\* generate something to put in buffer \*/*  
*/\* decrement empty count \*/*  
*/\* enter critical region \*/*  
*/\* put new item in buffer \*/*  
*/\* leave critical region \*/*  
*/\* increment count of full slots \*/*

# Producer-Consumer Problem

```
void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item( );
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

*/\* infinite loop \*/*  
*/\* decrement full count \*/*  
*/\* enter critical region \*/*  
*/\* take item from buffer \*/*  
*/\* leave critical region \*/*  
*/\* increment count of empty slots \*/*  
*/\* do something with the item \*/*

# Osservazioni

- La programmazione della concorrenza con i semafori non è facile
  - P e V, sparpagliate nel codice
  - problema del corretto ordine di esecuzione di P e V
    - l'uso di P e V in ordine errato può portare a deadlock o a violazione della mutua esclusione
- Proposte soluzioni alternative a livello di linguaggio di programmazione
  - monitor
  - primitiva di sincronizzazione di alto livello

# Message Passing

- Basato sull'uso di due primitive di sincronizzazione
  - `send (destination, &message)` : invia un messaggio ad un processo destinatario
  - `receive (source, &message)` : riceve un messaggio da un processo. Se il messaggio non è ancora disponibile:
    - Il processo si mette in attesa
    - Ritorna un codice d'errore

# Prod-Cons

```
#define N 100                                /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                                /* message buffer */

    while (TRUE) {
        item = produce_item( );              /* generate something to put in buffer */
        receive(consumer, &m);              /* wait for an empty to arrive */
        build_message(&m, item);            /* construct a message to send */
        send(consumer, &m);                 /* send item to consumer */
    }
}
```

# Prod-Cons

```
void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m);                /* get message containing item */
        item = extract_item(&m);              /* extract item from message */
        send(producer, &m);                    /* send back empty reply */
        consume_item(item);                   /* do something with the item */
    }
}
```

# Indirizzamento- Mailbox

- In un sistema di scambio messaggi è fondamentale l'identificazione dei processi coinvolti:
  - Un identificatore univoco per ogni processo
  - Mailbox: buffer gestiti dai processi e usati nelle send e receive in sostituzione degli indentificatori di processo, un receive su mailbox vuota è bloccante e allo stesso modo una senda su mailbox piena

# Rendezvous

- Meccanismo di scambio messaggio sincrono che comporta la completa eliminazione del buffering
  - Se send viene effettuata prima di receive, il processo che ha eseguito la send si blocca in attesa che il destinatario effettui la receive
  - Lo stesso avviene in caso di receive “anticipata”
  - Sender/receiver obbligati ad un’esecuzione di tipo lockstep