

## Trasparenze del Corso di *Sistemi Operativi*

Marina Lenisa  
Università di Udine

Copyright © 2000-04 Marino Miculan (miculan@dimi.uniud.it)

La copia letterale e la distribuzione di questa presentazione nella sua integrità sono permesse con qualsiasi mezzo, a condizione che questa nota sia riprodotta.

1

## Cooperazione tra Processi

- Principi
- Il problema della sezione critica: le *race condition*
- Supporto hardware
- Semafori
- Monitor
- Scambio di messaggi
- Barriere
- Problemi classici di sincronizzazione

201

## Processi (e Thread) Cooperanti

- Processi *independenti* non possono modificare o essere modificati dall'esecuzione di un altro processo.
- I processi *cooperanti* possono modificare o essere modificati dall'esecuzione di altri processi.
- Vantaggi della cooperazione tra processi:
  - Condivisione delle informazioni
  - Aumento della computazione (parallelismo)
  - Modularità
  - Praticità implementativa/di utilizzo

202

## IPC: InterProcess Communication

Meccanismi di comunicazione e interazione tra processi (e thread)

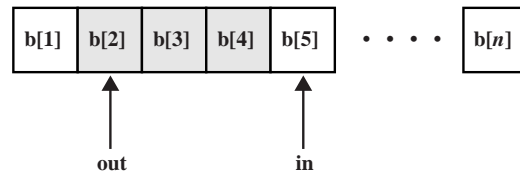
Questioni da considerare:

- Come può un processo passare informazioni ad un altro?
- Come evitare accessi inconsistenti a risorse condivise?
- Come sequenzializzare gli accessi alle risorse secondo la causalità?

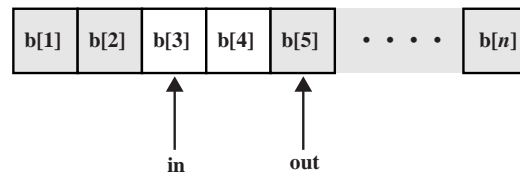
Mantenere la consistenza dei dati richiede dei meccanismi per assicurare l'esecuzione ordinata dei processi cooperanti.

203

## Esempio: Problema del produttore-consumatore



(a)



(b)

- Tipico paradigma dei processi cooperanti: il processo *produttore* produce informazione che viene consumata da un processo *consumatore*
- Soluzione a memoria condivisa: tra i due processi si pone un buffer di comunicazione di dimensione fissata.

204

Processo produttore

```
repeat
  ...
  produce un item in nextp
  ...
  while counter = n do no-op;
  buffer[in] := nextp;
  in := in + 1 mod n;
  counter := counter + 1;
until false;
```

Processo consumatore

```
repeat
  while counter = 0 do no-op;
  nextc := buffer[out];
  out := out + 1 mod n;
  counter := counter - 1;
  ...
  consuma l'item in nextc
  ...
until false;
```

- Le istruzioni

- *counter* := *counter* + 1;
- *counter* := *counter* - 1;

devono essere eseguite *atomicamente*: se eseguite in parallelo non atomicamente, possono portare ad inconsistenze.

## Produttore-consumatore con buffer limitato

- Dati condivisi tra i processi

```
type item = ... ;
var buffer: array [0..n-1] of item;
    in, out: 0..n-1;
    counter: 0..n;
    in, out, counter := 0;
```

205

## Race conditions

**Race condition (corsa critica):** più processi accedono concorrentemente agli stessi dati, e il risultato dipende dall'ordine di interleaving dei processi.

- Frequenti nei sistemi operativi multitasking, sia per dati in user space sia per strutture in kernel.
- Estremamente pericolose: portano al malfunzionamento dei processi cooperanti, o anche (nel caso delle strutture in kernel space) dell'intero sistema.
- Difficili da individuare e riprodurre: dipendono da informazioni astratte dai processi (decisioni dello scheduler, carico del sistema, utilizzo della memoria, numero di processori, ...)

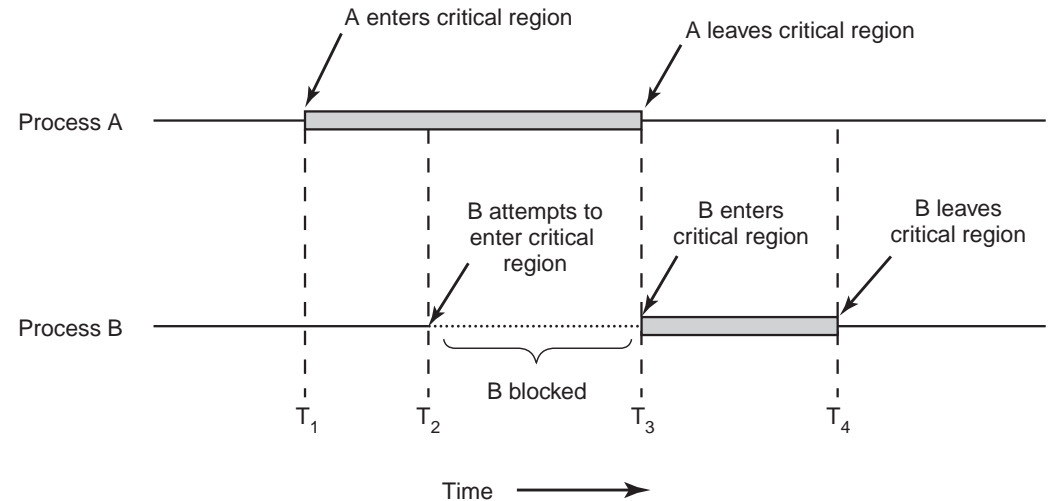
206

## Problema della Sezione Critica

- $n$  processi che competono per usare dati condivisi
- Ogni processo ha un segmento di codice, detto *sezione critica* in cui accede ai dati condivisi.
- Per evitare race condition serve assicurare che quando un processo esegue la sua sezione critica, nessun altro processo possa entrare nella propria sezione critica.
- Come? Proteggendo la sezione critica con apposito *codice di controllo*

```
while (TRUE) {  
    entry section  
    sezione critica  
    exit section  
    sezione non critica  
};
```

207



208

## Criteri per una Soluzione del Problema della Sezione Critica

1. **Mutua esclusione:** se il processo  $P_i$  sta eseguendo la sua sezione critica, allora nessun altro processo può eseguire la propria sezione critica.
  2. **Progresso:** nessun processo in esecuzione fuori dalla sua sezione critica può bloccare processi che desiderano entrare nella propria sezione critica.
  3. **Attesa limitata:** se un processo  $P$  ha richiesto di entrare nella propria sezione critica, allora il numero di volte che si concede agli altri processi di accedere alla propria sezione critica prima del processo  $P$  deve essere limitato.
- Si suppone che ogni processo venga eseguito ad una velocità non nulla.
  - Non si suppone niente sulla velocità *relativa* dei processi (e quindi sul numero e tipo di CPU)

209

## Soluzioni hardware: controllo degli interrupt

- Il processo può disabilitare TUTTI gli interrupt hw all'ingresso della sezione critica, e riabilitarli all'uscita
  - Soluzione semplice; garantisce la mutua esclusione
  - ma pericolosa: il processo può non riabilitare più gli interrupt, acquisendosi la macchina
  - può allungare di molto i tempi di latenza
  - non si estende a macchine multiprocessore (a meno di non bloccare tutte le altre CPU)
- Inadatto come meccanismo di mutua esclusione tra processi utente
- Adatto per brevi(ssimi) segmenti di codice affidabile (es: in kernel, quando si accede a strutture condivise)

210

## Soluzioni software

- Supponiamo che ci siano solo 2 processi,  $P_0$  e  $P_1$

- Struttura del processo  $P_i$  (l'altro sia  $P_j$ )

```
while (TRUE) {  
    entry section  
    sezione critica  
    exit section  
    sezione non critica  
}
```

- Supponiamo che i processi possano condividere alcune variabili (dette di *lock*) per sincronizzare le proprie azioni

211

## Tentativo sbagliato

- Variabili condivise

– **var** *occupato*: (0..1);

inizialmente *occupato* = 0

– *occupato* = 0  $\Rightarrow$  un processo può entrare nella propria sezione critica

- Processo  $P_i$

```
while (TRUE) {  
    ↓  
    while (occupato  $\neq$  0); occupato := 1;  
    sezione critica  
    occupato := 0;  
    sezione non critica  
};
```

- Non funziona: lo scheduler può agire dopo il ciclo, nel punto indicato.

212

## Alternanza stretta

- Variabili condivise

– **var** *turn*: (0..1);

inizialmente *turn* = 0

– *turn* =  $i \Rightarrow P_i$  può entrare nella propria sezione critica

- Processo  $P_i$

```
while (TRUE) {  
    while (turn  $\neq$   $i$ ) no-op;  
    sezione critica  
    turn :=  $j$ ;  
    sezione non critica  
};
```

213

## Alternanza stretta (cont.)

- Soddisfa il requisito di mutua esclusione, ma non di progresso (richiede l'alternanza stretta)  $\Rightarrow$  inadatto per processi con differenze di velocità

- È un esempio di *busy wait*: attesa *attiva* di un evento (es: testare il valore di una variabile).

– Semplice da implementare

– Può portare a consumi inaccettabili di CPU

– In genere, da evitare, ma a volte è preferibile (es. in caso di attese molto brevi)

- Un processo che attende attivamente su una variabile esegue uno *spin lock*.

214

## Algoritmo di Peterson (1981)

```
#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 */
}
```

215

## Algoritmo di Peterson (cont)

- Basato su una combinazione di *richiesta* e *accesso*
- Soddisfa tutti i requisiti; risolve il problema della sezione critica per 2 processi
- Si può generalizzare a  $N$  processi
- È ancora basato su spinlock

216

## Algoritmo del Fornaio

Risolve la sezione critica per  $n$  processi, generalizzando l'idea vista precedentemente.

- Prima di entrare nella sezione critica, ogni processo riceve un numero. Chi ha il numero più basso entra nella sezione critica.
- Eventuali conflitti vengono risolti da un ordine statico: Se i processi  $P_i$  and  $P_j$  ricevono lo stesso numero: se  $i < j$ , allora  $P_i$  è servito per primo; altrimenti  $P_j$  è servito per primo.
- Lo schema di numerazione genera numeri in ordine crescente, i.e., 1,2,3,3,3,4,5

217

## Istruzioni di Test&Set

- Istruzioni di Test-and-Set-Lock: testano e modificano il contenuto di una parola atomicamente

```
function Test-and-Set (var target: boolean): boolean;
begin
    Test-and-Set := target;
    target := true;
end;
```

- Questi due passi devono essere implementati come atomici in assembler. (Es: le istruzioni BTC, BTR, BTS su Intel). Ipoteticamente:

```
TSL RX, LOCK
```

Copia il contenuto della cella LOCK nel registro RX, e poi imposta la cella LOCK ad un valore  $\neq 0$ . Il tutto atomicamente (viene bloccato il bus di memoria).

218

## Istruzioni di Test&Set (cont.)

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
```

- corretto e semplice
- è uno spinlock — quindi busy wait
- Problematico per macchine parallele

219

## Evitare il busy wait

- Le soluzioni basate su spinlock portano a
    - busy wait: alto consumo di CPU
    - inversione di priorità: un processo a bassa priorità che blocca una risorsa può essere bloccato all'infinito da un processo ad alta priorità in busy wait sulla stessa risorsa.
  - Idea migliore: quando un processo deve attendere un evento, che venga posto in *wait*; quando l'evento avviene, che venga posto in *ready*
  - Servono specifiche syscall o funzioni di kernel. Esempio:
    - *sleep()*: il processo si autosospende (si mette in *wait*)
    - *wakeup(pid)*: il processo *pid* viene posto in *ready*, se era in *wait*.
- Ci sono molte varianti. Molto comune: con *evento* esplicito.

220

## Produttore-consumatore con sleep e wakeup

```
#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 */
    }
}
```

221

## Produttore-consumatore con sleep e wakeup (cont.)

- Risolve il problema del busy wait
- Non risolve la corsa critica sulla variabile *count*
- I segnali possono andare *perduti*, con conseguenti *deadlock*
- Soluzione: salvare i segnali “in attesa” in un contatore

222

## Semafori

Strumento di sincronizzazione generale (Dijkstra '65)

- Semaforo  $S$ : variabile intera.
- Vi si può accedere solo attraverso 2 operazioni **atomiche**:
  - $up(S)$ : incrementa  $S$
  - $down(S)$ : attendi finché  $S$  è maggiore di 0; quindi decrementa  $S$
- Normalmente, l'attesa è implementata spostando il processo in stato di *wait*, mentre la  $up(S)$  mette uno dei processi eventualmente in attesa nello stato di *ready*.
- I nomi originali erano  $P$  (*proberen*, testare) e  $V$  (*verhogen*, incrementare)

223

## Esempio: Sezione Critica per $n$ processi

- Variabili condivise:
  - **var**  $mutex$  : semaphore
  - inizialmente  $mutex = 1$
- Processo  $P_i$

```
while (TRUE) {
    down(mutex);
    sezione critica
    up(mutex);
    sezione non critica
}
```

224

## Esempio: Sincronizzazione tra due processi

- Variabili condivise:
  - **var**  $sync$  : semaphore
  - inizialmente  $sync = 0$
- Processo  $P_1$     Processo  $P_2$ 

:	:
$S_1$ ;	$down(sync)$ ;
$up(sync)$ ;	$S_2$ ;
:	:
- $S_2$  viene eseguito solo dopo  $S_1$ .

225

## Esempio: Produttore-Consumatore con semafori

```
#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 */
    }
}
```

226

```

void consumer(void)
{
    int item;

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

```

## Implementazione dei semafori

- La definizione classica usava uno *spinlock* per la *down*: facile implementazione (specialmente su macchine parallele), ma inefficiente
- Alternativa: il processo in attesa viene messo in stato di *wait*
- In generale, un semaforo è un record

```

type semaphore = record
    value: integer;
    L: list of process;
end;

```

- Assumiamo due operazioni fornite dal sistema operativo:
  - *sleep()*: sospende il processo che la chiama (rilascia la CPU)
  - *wakeup(P)*: pone in stato di *ready* il processo *P*.

227

## Implementazione dei semafori (Cont.)

- Le operazioni sui semafori sono definite come segue:

```

down(S): S.value := S.value - 1;
        if S.value < 0
            then begin
                aggiungi questo processo a S.L;
                sleep();
            end;
up(S):   S.value := S.value + 1;
        if S.value ≤ 0
            then begin
                toglì un processo P da S.L;
                wakeup(P);
            end;

```

228

## Implementazione dei semafori (Cont.)

- *value* può avere valori negativi: indica quanti processi sono in attesa su quel semaforo
- le due operazioni *down* e *up* devono essere *atomiche* fino a prima della *sleep* e *wakeup*: problema di sezione critica, da risolvere come visto prima:
  - disabilitazione degli interrupt: semplice, ma inadatto a sistemi con molti processori
  - uso di istruzioni speciali (test-and-set)
  - ciclo busy-wait (spinlock): generale, e sufficientemente efficiente (le due sezioni critiche sono molto brevi)

229



## Mutex

- I mutex sono semafori con due soli possibili valori: *bloccato* o *non bloccato*
- Utili per implementare mutua esclusione, sincronizzazione, ...
- due primitive: *mutex\_lock* e *mutex\_unlock*.
- Semplici da implementare, anche in user space (p.e. per thread). Esempio:

```
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
```

230

## Memoria condivisa?

Implementare queste funzioni richiede una qualche memoria condivisa.

- A livello kernel: strutture come quelle usate dai semafori possono essere mantenute nel kernel, e quindi accessibili da tutti i processi (via le apposite system call)
- A livello utente:
  - all'interno dello stesso processo: adatto per i thread
  - tra processi diversi: spesso i S.O. offrono la possibilità di condividere segmenti di memoria tra processi diversi (*shared memory*)
  - alla peggio: file su disco

231

## Deadlock con Semafori

- **Deadlock (stallo):** due o più processi sono in attesa indefinita di eventi che possono essere causati solo dai processi stessi in attesa.
- L'uso dei semafori può portare a deadlock. Esempio: siano *S* e *Q* due semafori inizializzati a 1

```

P0      P1
down(S); down(Q);
down(Q); down(S);
:         :
up(S);   up(Q);
up(Q);   up(S);
```

- Programmare con i semafori è molto delicato e pronò ad errori, difficilissimi da debuggare. Come in assembler, solo peggio, perché qui gli errori sono race condition e malfunzionamenti non riproducibili.

232

## Monitor

- Un *monitor* è un tipo di dato astratto che fornisce funzionalità di mutua esclusione
  - collezione di dati privati e funzioni/procedure per accedervi.
  - i processi possono chiamare le procedure ma non accedere alle variabili locali.
  - *un solo* processo alla volta può eseguire codice di un monitor
- Il programmatore raccoglie quindi i dati condivisi e tutte le sezioni critiche relative in un monitor; questo risolve il problema della mutua esclusione
- Implementati dal compilatore con dei costrutti per mutua esclusione (p.e.: inserisce automaticamente `lock_mutex` e `unlock_mutex` all'inizio e fine di ogni procedura)

**monitor example**  
**integer i;**  
**condition c;**

**procedure producer();**

·  
·  
·

**end;**

**procedure consumer();**

·  
·  
·

**end;**

**end monitor;**

233

## Monitor: Controllo del flusso di controllo

Per sospendere e riprendere i processi, ci sono le variabili *condition*, simili agli eventi, con le operazioni

- *wait(c)*: il processo che la esegue si blocca sulla condizione *c*.
- *signal(c)*: uno dei processi in attesa su *c* viene risvegliato.

A questo punto, chi va in esecuzione nel monitor? Due varianti:

- chi esegue la *signal(c)* si sospende automaticamente (*monitor di Hoare*)
- la *signal(c)* deve essere l'ultima istruzione di una procedura (così il processo lascia il monitor) (*monitor di Brinch-Hansen*)
- i processi risvegliati possono provare ad entrare nel monitor solo dopo che il processo attuale lo ha lasciato

Il successivo processo ad entrare viene scelto dallo scheduler di sistema

- i *signal* su una condizione senza processi in attesa vengono persi

234

## Produttore-consumatore con monitor

```
monitor ProducerConsumer
```

```
condition full, empty;
```

```
integer count;
```

```
procedure insert(item: integer);
```

```
begin
```

```
    if count = N then wait(full);
```

```
    insert_item(item);
```

```
    count := count + 1;
```

```
    if count = 1 then signal(empty)
```

```
end;
```

```
function remove: integer;
```

```
begin
```

```
    if count = 0 then wait(empty);
```

```
    remove = remove_item;
```

```
    count := count - 1;
```

```
    if count = N - 1 then signal(full)
```

```
end;
```

```
count := 0;
```

```
end monitor;
```

```
procedure producer;
```

```
begin
```

```
    while true do
```

```
        begin
```

```
            item = produce_item;
```

```
            ProducerConsumer.insert(item)
```

```
        end
```

```
end;
```

```
procedure consumer;
```

```
begin
```

```
    while true do
```

```
        begin
```

```
            item = ProducerConsumer.remove;
```

```
            consume_item(item)
```

```
        end
```

```
end;
```

235

## Monitor (cont.)

- I monitor semplificano molto la gestione della mutua esclusione (meno possibilità di errori)
- Veri costrutti, non funzioni di libreria  $\Rightarrow$  bisogna modificare i compilatori.
- Implementati (in certe misure) in veri linguaggi.  
Esempio: i metodi *synchronized* di Java.
  - solo un metodo *synchronized* di una classe può essere eseguito alla volta.
  - Java non ha variabili *condition*, ma ha *wait* and *notify* (+ o - come *sleep* e *wakeup*).
- Un problema che rimane (sia con i monitor che con i semafori): è necessario avere *memoria condivisa*  $\Rightarrow$  questi costrutti non sono applicabili a sistemi distribuiti (reti di calcolatori) senza memoria fisica condivisa.

236

## Passaggio di messaggi

- Comunicazione non basata su memoria condivisa con controllo di accesso.
- Basato su due primitive (chiamate di sistema o funzioni di libreria)
  - *send(destinazione, messaggio)*: spedisce un messaggio ad una certa destinazione; solitamente non bloccante.
  - *receive(sorgente, &messaggio)*: riceve un messaggio da una sorgente; solitamente bloccante (fino a che il messaggio non è disponibile).
- Meccanismo più astratto e generale della memoria condivisa e semafori
- Si presta ad una implementazione su macchine distribuite

237

## Problematiche dello scambio di messaggi

- Affidabilità: i canali possono essere inaffidabili (es: reti). Bisogna implementare appositi protocolli fault-tolerant (basati su acknowledgment e timestamping).
- Autenticazione: come autenticare i due partner?
- Sicurezza: i canali utilizzati possono essere intercettati
- Efficienza: se prende luogo sulla stessa macchina, il passaggio di messaggi è sempre più lento della memoria condivisa e semafori.

238

## Produttore-consumatore con scambio di messaggi

- Comunicazione *asincrona*
  - I messaggi spediti ma non ancora consumati vengono automaticamente bufferizzati in una *mailbox* (mantenuto in kernel o dalle librerie)
  - L'oggetto delle *send* e *receive* sono le mailbox
  - La *send* si blocca se la mailbox è piena; la *receive* si blocca se la mailbox è vuota.
- Comunicazione *sincrona*
  - I messaggi vengono spediti direttamente al processo destinazione
  - L'oggetto delle *send* e *receive* sono i processi
  - Le *send* e *receive* si bloccano fino a che la controparte non esegue la chiamata duale (*rendez-vous*).

239

```

#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 */
    }
}

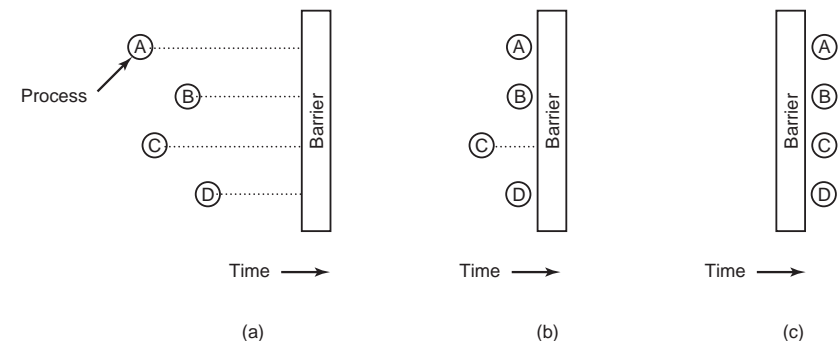
void consumer(void)
{
    int item, i;
    message m;

    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 */
    }
}

```

## Barriera

- Meccanismo di sincronizzazione per *gruppi* di processi, specialmente per calcolo parallelo a memoria condivisa (es. SMP, NUMA)
  - Ogni processo alla fine della sua computazione, chiama la funzione *barrier* e si sospende.
  - Quando tutti i processi hanno raggiunto la barriera, la superano *tutti assieme* (si sbloccano).



240

## I Grandi Classici

Esempi paradigmatici di programmazione concorrente. Presi come testbed per ogni primitiva di programmazione e comunicazione.

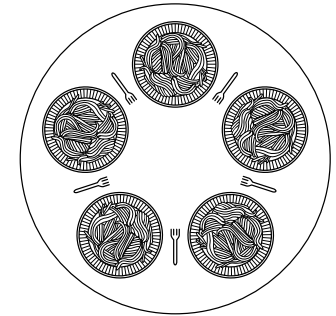
(E buoni esempi didattici!)

- Produttore-Consumatore a buffer limitato (già visto)
- I Filosofi a Cena
- Lettori-Scrittori
- Il Barbiere che Dorme

241

## I Classici: I Filosofi a Cena (Dijkstra, 1965)

$n$  filosofi seduti attorno ad un tavolo rotondo con  $n$  piatti di spaghetti e  $n$  forchette (bastoncini). (nell'esempio,  $n = 5$ )



- Mentre pensa, un filosofo non interagisce con nessuno
- Quando gli viene fame, cerca di prendere le bacchette più vicine, una alla volta.
- Quando ha due bacchette, un filosofo mangia senza fermarsi.
- Terminato il pasto, lascia le bacchette e torna a pensare.

Problema: programmare i filosofi in modo da garantire

- assenza di deadlock: non si verificano mai blocchi
- assenza di starvation: un filosofo che vuole mangiare, prima o poi mangia.

242

## I Filosofi a Cena—Una non-soluzione

```
#define N 5                /* number of philosophers */

void philosopher(int i)    /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think();           /* philosopher is thinking */
        take_fork(i);      /* take left fork */
        take_fork((i+1) % N); /* take right fork; % is modulo operator */
        eat();             /* yum-yum, spaghetti */
        put_fork(i);       /* put left fork back on the table */
        put_fork((i+1) % N); /* put right fork back on the table */
    }
}
```

Possibilità di deadlock: se tutti i processi prendono contemporaneamente la forchetta alla loro sinistra...

243

## I Filosofi a Cena—Tentativi di correzione

- Come prima, ma controllare se la forchetta dx è disponibile prima di prelevarla, altrimenti rilasciare la forchetta sx e riprovare daccapo.
  - Non c'è deadlock, ma possibilità di starvation.
- Come sopra, ma introdurre un ritardo casuale prima della ripetizione del tentativo.
  - Non c'è deadlock, la possibilità di starvation viene ridotta ma non azzerata. Applicato in molti protocolli di accesso (CSMA/CD, es. Ethernet). Inadatto in situazione mission-critical o real-time.

244

## I Filosofi a Cena—Soluzioni

- Introdurre un semaforo `mutex` per proteggere la sezione critica (dalla prima `take_fork` all'ultima `put_fork`):
  - Funziona, ma solo un filosofo per volta può mangiare, mentre in teoria  $\lfloor n/2 \rfloor$  possono mangiare contemporaneamente.
- Tenere traccia dell'*intenzione* di un filosofo di mangiare. Un filosofo ha tre stati (THINKING, HUNGRY, EATING), mantenuto in un vettore `state`. Un filosofo può entrare nello stato EATING solo è HUNGRY e i vicini non sono EATING.
  - Funziona, e consente il massimo parallelismo.

245

```
void take_forks(int i)           /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                /* enter critical region */
    state[i] = HUNGRY;           /* record fact that philosopher i is hungry */
    test(i);                     /* try to acquire 2 forks */
    up(&mutex);                  /* exit critical region */
    down(&s[i]);                  /* block if forks were not acquired */
}

void put_forks(i)                /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                /* enter critical region */
    state[i] = THINKING;        /* philosopher has finished eating */
    test(LEFT);                 /* see if left neighbor can now eat */
    test(RIGHT);                /* see if right neighbor can now eat */
    up(&mutex);                  /* exit critical region */
}

void test(i)                     /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

```
#define N          5           /* number of philosophers */
#define LEFT      (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT     (i+1)%N    /* number of i's right neighbor */
#define THINKING  0           /* philosopher is thinking */
#define HUNGRY    1           /* philosopher is trying to get forks */
#define EATING    2           /* philosopher is eating */
typedef int semaphore;        /* semaphores are a special kind of int */
int state[N];                /* array to keep track of everyone's state */
semaphore mutex = 1;         /* mutual exclusion for critical regions */
semaphore s[N];              /* one semaphore per philosopher */

void philosopher(int i)       /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {            /* repeat forever */
        think();              /* philosopher is thinking */
        take_forks(i);        /* acquire two forks or block */
        eat();                /* yum-yum, spaghetti */
        put_forks(i);         /* put both forks back on table */
    }
}
```

## I Classici: Lettori-Scrittori

Un insieme di dati (es. un file, un database, dei record), deve essere condiviso da processi *lettori* e *scrittori*

- Due o più lettori possono accedere contemporaneamente ai dati
- Ogni scrittore deve accedere ai dati in modo esclusivo.

Implementazione con i semafori:

- Tenere conto dei lettori in una variabile condivisa, e fino a che ci sono lettori, gli scrittori non possono accedere.
- Dà maggiore priorità ai lettori che agli scrittori.

246

```

typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

/* use your imagination */
/* controls access to 'rc' */
/* controls access to the database */
/* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {
        /* repeat forever */
        down(&mutex);
        /* get exclusive access to 'rc' */
        rc = rc + 1;
        /* one reader more now */
        if (rc == 1) down(&db);
        /* if this is the first reader ... */
        up(&mutex);
        /* release exclusive access to 'rc' */
        read_data_base();
        /* access the data */
        down(&mutex);
        /* get exclusive access to 'rc' */
        rc = rc - 1;
        /* one reader fewer now */
        if (rc == 0) up(&db);
        /* if this is the last reader ... */
        up(&mutex);
        /* release exclusive access to 'rc' */
        use_data_read();
        /* noncritical region */
    }
}

```

```

void writer(void)
{
    while (TRUE) {
        /* repeat forever */
        think_up_data();
        /* noncritical region */
        down(&db);
        /* get exclusive access */
        write_data_base();
        /* update the data */
        up(&db);
        /* release exclusive access */
    }
}

```

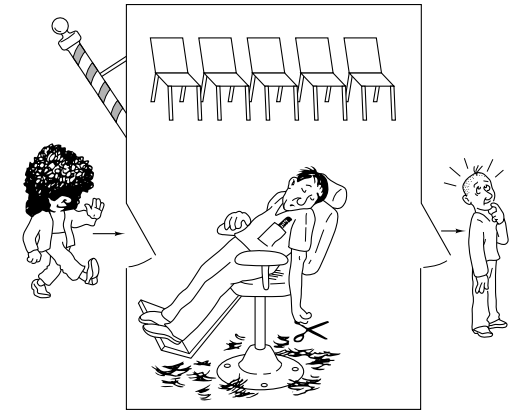
## Il Barbiere—Soluzione

- Tre semafori:
  - customers: i clienti in attesa (contati anche da una variabile waiting)
  - barbers: conta i barbieri in attesa
  - mutex: per mutua esclusione
- Ogni barbiere (uno) esegue una procedura che lo blocca se non ci sono clienti; quando si sveglia, serve un cliente e ripete.
- Ogni cliente prima di entrare nel negozio controlla se ci sono sedie libere; altrimenti se ne va.
- Un cliente, quando entra nel negozio, sveglia il barbiere se sta dormendo.

## I Classici: Il Barbiere che Dorme

In un negozio c'è un solo barbiere, una sedia da barbiere e  $n$  sedie per l'attesa.

- Quando non ci sono clienti, il barbiere dorme sulla sedia.
- Quando arriva un cliente, questo sveglia il barbiere se sta dormendo.
- Se la sedia è libera e ci sono clienti, il barbiere fa sedere un cliente e lo serve.
- Se un cliente arriva e il barbiere sta già servendo un cliente, si siede su una sedia di attesa se ce ne sono di libere, altrimenti se ne va.



Problema: programmare il barbiere e i clienti filosofi in modo da garantire assenza di deadlock e di starvation.

```

#define CHAIRS 5
/* # chairs for waiting customers */

typedef int semaphore;
/* use your imagination */

semaphore customers = 0;
/* # of customers waiting for service */
semaphore barbers = 0;
/* # of barbers waiting for customers */
semaphore mutex = 1;
/* for mutual exclusion */
int waiting = 0;
/* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);
        /* go to sleep if # of customers is 0 */
        down(&mutex);
        /* acquire access to 'waiting' */
        waiting = waiting - 1;
        /* decrement count of waiting customers */
        up(&barbers);
        /* one barber is now ready to cut hair */
        up(&mutex);
        /* release 'waiting' */
        cut_hair();
        /* cut hair (outside critical region) */
    }
}

```

```
void customer(void)
{
    down(&mutex);           /* enter critical region */
    if (waiting < CHAIRS) { /* if there are no free chairs, leave */
        waiting = waiting + 1; /* increment count of waiting customers */
        up(&customers);       /* wake up barber if necessary */
        up(&mutex);          /* release access to 'waiting' */
        down(&barbers);      /* go to sleep if # of free barbers is 0 */
        get_haircut();       /* be seated and be serviced */
    } else {
        up(&mutex);          /* shop is full; do not wait */
    }
}
```