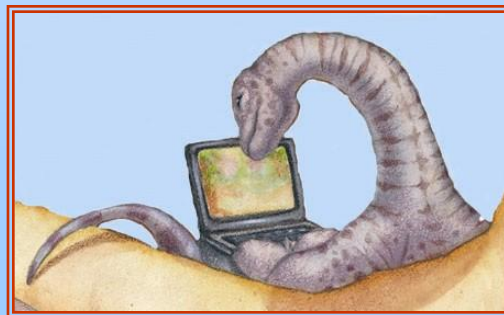


Sincronizzazione dei processi





Sincronizzazione dei processi

- ✿ Background
- ✿ Il problema della sezione critica
- ✿ Hardware di sincronizzazione
- ✿ Semafori
- ✿ Problemi classici di sincronizzazione
- ✿ Monitor





Background – 1

- ✿ Un processo cooperante può influenzare gli altri processi in esecuzione nel sistema o subirne l'influenza
- ✿ I processi cooperanti possono...
 - ✗ ...condividere uno spazio logico di indirizzi, cioè codice e dati (thread, memoria condivisa)
 - ✗ ...oppure solo dati, attraverso i file
- ✿ L'accesso concorrente a dati condivisi può causare inconsistenza nei dati
- ✿ Per garantire la coerenza dei dati occorrono meccanismi che assicurano l'esecuzione ordinata dei processi cooperanti





Background – 2

- ✿ La soluzione con memoria condivisa e buffer limitato del problema del produttore–consumatore permette la presenza contemporanea nel buffer di al più $\text{DIM_VETTORE}-1$ elementi
- ✿ Una soluzione in cui vengano impiegati tutti gli elementi del buffer presuppone...
 - ✗ ...la modifica del codice del produttore–consumatore con l'aggiunta di una variabile *contatore*, inizializzata a zero
 - ✗ *contatore* viene incrementato ogni volta che un nuovo elemento viene inserito nel buffer, decrementato dopo ogni estrazione





Buffer limitato – 1

❄ Dati condivisi

```
#define DIM_VETTORE 10
typedef struct {
    . . .
} ELEMENTO;
ELEMENTO vettore[DIM_VETTORE];
int inserisci = 0;
int preleva = 0;
int contatore = 0;
```





Buffer limitato – 2

Processo produttore

inserisci — indice della successiva posizione libera nel buffer

```
ELEMENTO appena_prodotto;
while (1) {
    while (contatore == DIM_VETTORE)
        ; /* do nothing */
    vettore[inserisci] = appena_prodotto;
    inserisci = (inserisci + 1) % DIM_VETTORE;
    contatore++;
}
```

```
ELEMENTO da_consumare;
while (1) {
    while (contatore == 0)
        ; /* do nothing */
    da_consumare = vettore[preleva];
    preleva = (preleva + 1) % DIM_VETTORE;
    contatore--;
}
```

Processo consumatore

preleva — indice della prima posizione piena nel buffer





Buffer limitato – 3

- ✱ Sebbene siano corrette se si considerano separatamente, le procedure del produttore e del consumatore possono non “funzionare” se eseguite in concorrenza
- ✱ In particolare, le istruzioni:

```
contatore++;  
contatore--;
```

devono essere eseguite *atomicamente*





Buffer limitato – 4

- ✱ Un'operazione è atomica quando viene completata senza subire interruzioni
- ✱ Le istruzioni di aggiornamento del contatore vengono realizzate in linguaggio macchina come...

registro₁ = contatore
registro₁ = registro₁ + 1
contatore = registro₁

LOAD R1, CONT
ADD 1, R1
STORE R1, CONT

registro₂ = contatore
registro₂ = registro₂ - 1
contatore = registro₂

LOAD R2, CONT
SUB 1, R2
STORE R2, CONT

I due registri
possono
coincidere...





Buffer limitato – 5

- ✱ Se il produttore ed il consumatore tentano di accedere al buffer contemporaneamente, le istruzioni in linguaggio macchina possono risultare *interfogliate*
 - ✱ La sequenza effettiva di esecuzione dipende da come i processi produttore e consumatore vengono schedulati
 - ✱ **Esempio:** inizialmente contatore=5
- | | | |
|--------------------|---|----------------------------|
| T_0 produttore: | registro ₁ =contatore | (registro ₁ =5) |
| T_1 produttore: | registro ₁ =registro ₁ +1 | (registro ₁ =6) |
| T_2 consumatore: | registro ₂ =contatore | (registro ₂ =5) |
| T_3 consumatore: | registro ₂ =registro ₂ -1 | (registro ₂ =4) |
| T_4 produttore: | contatore=registro ₁ | (contatore=6) |
| T_5 consumatore: | contatore=registro ₂ | (contatore=4) |
- ✱ *contatore* varrà 4, 5 o 6, mentre dovrebbe rimanere uguale a 5 (un elemento prodotto ed uno consumato)





Race condition

- ✱ *Race condition* — Più processi accedono in concorrenza e modificano dati condivisi; l'esito dell'esecuzione (il valore finale dei dati condivisi) dipende dall'ordine nel quale sono avvenuti gli accessi
- ✱ Per evitare le **corse critiche** occorre che i processi concorrenti vengano **sincronizzati**
 - ✱ Tali situazioni si verificano spesso nei SO, nei quali diverse componenti compiono operazioni su risorse condivise
 - ⇒ Problema particolarmente significativo nei sistemi multicore, in cui diversi thread vengono eseguiti in parallelo su unità di calcolo distinte





Il problema della sezione critica – 1

- ✱ n processi $\{P_0, P_1, \dots, P_{n-1}\}$ competono per utilizzare dati condivisi
- ✱ Ciascun processo è costituito da un segmento di codice, chiamato **sezione critica** (o *regione critica*), in cui accede a dati condivisi
- ✱ **Problema** — assicurarsi che, quando un processo accede alla propria sezione critica, a nessun altro processo sia concessa l'esecuzione di un'azione analoga
- ✱ L'esecuzione di sezioni critiche da parte di processi cooperanti è **mutuamente esclusiva** nel tempo





Il problema della sezione critica – 2

- ✱ **Soluzione** — progettare un protocollo di cooperazione fra processi:
 - ✗ Ogni processo deve chiedere il permesso di accesso alla sezione critica, tramite una *entry section*
 - ✗ La sezione critica è seguita da una *exit section*
 - ✗ Il rimanente codice è non critico

```
do {  
    entry section  
    sezione critica  
    exit section  
    sezione non critica  
} while (1);
```






Soluzione al problema della sezione critica

1. **Mutua esclusione** — Se il processo P_i è in esecuzione nella sezione critica, nessun altro processo può eseguire la propria sezione critica
2. **Progresso** — Se nessun processo è in esecuzione nella propria sezione critica ed esiste qualche processo che desidera accedervi, allora la selezione del processo che entrerà prossimamente nella propria sezione critica non può essere rimandata indefinitamente
3. **Attesa limitata** — Se un processo ha effettuato la richiesta di ingresso nella sezione critica, è necessario porre un limite al numero di volte che si consente ad altri processi di entrare nelle proprie sezioni critiche, prima che la richiesta del primo processo sia stata accordata (politica *fair* per evitare la *starvation*)
 - Si assume che ciascun processo sia eseguito ad una velocità diversa da zero
 - Non si fanno assunzioni sulla velocità relativa degli n processi





Soluzione di Peterson (1981)

- ✱ Soluzione per due processi P_0 e P_1 ($i=1-j$)
- ✱ Supponiamo che le istruzioni **LOAD** e **STORE** siano atomiche (non necessariamente vero per le architetture con pipeline)
- ✱ I due processi condividono due variabili:
 - ✕ `int turno`  `typedef enum{FALSE,TRUE} boolean;`
 - ✕ `boolean flag[2]`
- ✱ La variabile **turno** indica il processo che "è di turno" per accedere alla propria sezione critica
- ✱ L'array **flag** si usa per indicare se un processo è pronto ad entrare nella propria sezione critica
 - ✕ `flag[i]=TRUE` implica che il processo P_i è pronto per accedere alla propria sezione critica





Algoritmo per il processo P_i

```
while (1) {  
    flag[i] = TRUE;  
    turno = j;  
    while (flag[j] && turno==j)  
        \\do no-op;  
    SEZIONE CRITICA  
    flag[i] = FALSE;  
    SEZIONE NON CRITICA  
}
```

- ✿ **Soluzione** — P_i entra nella sezione critica (progresso) al massimo dopo un ingresso da parte di P_j (attesa limitata): **alternanza stretta**





Hardware di sincronizzazione – 1

- ✿ In generale, qualsiasi soluzione al problema della sezione critica richiede l'uso di un semplice strumento, detto **lock** (lucchetto)
 - ✗ Per accedere alla propria sezione critica, il processo deve acquisire il possesso di un lock, che restituirà al momento della sua uscita

```
do {  
    acquisisce il lock  
    sezione critica  
    restituisce il lock  
    sezione non critica  
} while (1);
```





Hardware di sincronizzazione – 2

- ✱ Molti sistemi forniscono supporto hardware per risolvere efficacemente il problema della sezione critica
- ✱ In un sistema monoprocessoore è sufficiente interdire le interruzioni mentre si modificano le variabili condivise
 - ✗ Il job attualmente in esecuzione viene eseguito senza possibilità di prelazione
 - ✗ Soluzione inefficiente nei sistemi multiprocessore
 - SO non scalabili





Hardware di sincronizzazione – 3

- ✱ Molte architetture attuali offrono particolari istruzioni atomiche che...
 - ✗ ...permettono di controllare e modificare il contenuto di una parola di memoria (TestAndSet)
 - ✗ oppure, di scambiare il contenuto di due parole di memoria (Swap)

```
boolean lock = FALSE;
```

```
boolean TestAndSet (boolean *object)
{
    boolean value = *object;
    *object = TRUE;
    return value;
}
```

Definizione di TestAndSet

```
do {
    while TestAndSet (&lock);
    sezione critica
    lock = FALSE;
    sezione non critica
} while (1);
```

Realizzazione della mutua esclusione





I semafori – 1

- ✱ I semafori sono strumenti di sincronizzazione
- ✱ Il semaforo S è una variabile intera
- ✱ Si può accedere al semaforo S (dopo l'inizializzazione) solo attraverso due operazioni indivisibili (operazioni atomiche, primitive) *wait()* e *signal()* (originariamente in olandese: *P* per *proberen*/verificare e *V* per *verhogen*/incrementare)

```
wait(S) {  
    while  $S \leq 0$   
        ; // do no-op  
     $S--$ ;  
}
```

Spinlock

```
signal(S) {  
     $S++$ ;  
}
```





I semafori – 2

- ✱ Tutte le modifiche al valore del semaforo contenute nelle operazioni *wait()* e *signal()* devono essere eseguite in modo indivisibile
 - ✗ Mentre un processo cambia il valore del semaforo, nessun altro processo può contemporaneamente modificare quello stesso valore
 - ✗ Nel caso di *wait()* devono essere effettuate atomicamente sia la verifica del valore del semaforo che il suo decremento





Sezione critica con n processi

✿ Variabili condivise:

```
semaphore mutex = 1 ; // inizialmente mutex = 1
```

✿ Processo P_i :

```
do {  
    wait(mutex);  
    sezione critica  
    signal(mutex);  
    sezione non critica  
} while (1);
```

Gli n processi condividono un semaforo comune, *mutex* per *mutual exclusion*





Tipologie di semafori

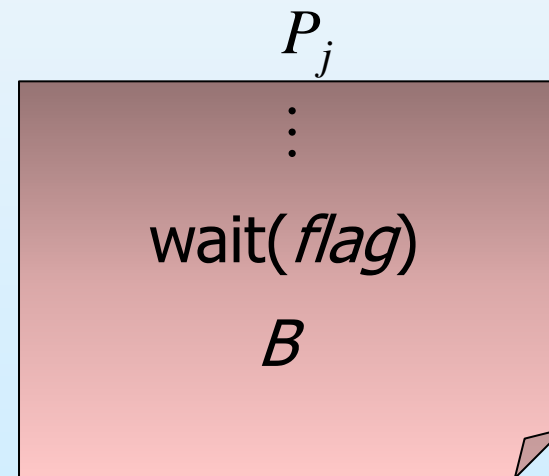
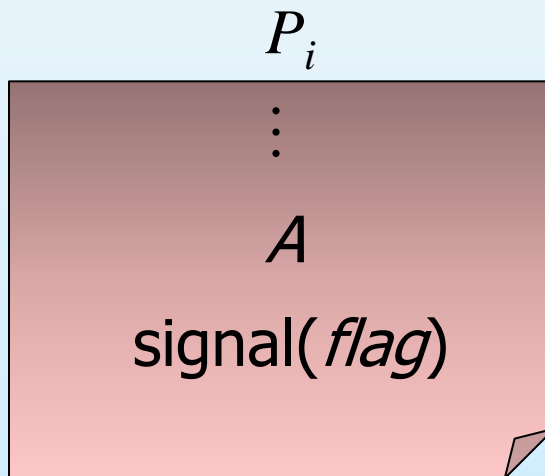
- ✱ **Semaforo contatore** — intero che può assumere valori in un dominio non limitato
 - ✗ Trova applicazione nel controllo dell'accesso a risorse presenti in un numero finito di esemplari
 - ✗ Il semaforo è inizialmente impostato al numero di esemplari disponibili
 - ✗ I processi che desiderano utilizzare un'istanza della risorsa, invocano una *wait()* sul semaforo, decrementandone così il valore
 - ✗ I processi che ne rilasciano un'istanza, invocano *signal()*, incrementando il valore del semaforo
 - ✗ Quando il semaforo vale 0, tutte le istanze della risorsa sono allocate e i processi che le richiedono devono sospendersi sul semaforo
- ✱ **Semaforo binario** — intero che può essere posto solo a 0 o 1, noto anche come *mutex lock*
 - ✗ Più facile da implementare





Il semaforo: uno strumento di sincronizzazione

- ✿ Si esegue B in P_j solo dopo che A è stato eseguito in P_i
- ✿ Si impiega un semaforo $flag$ inizializzato a 0
- ✿ Codice:





Busy waiting

- ✱ Mentre un processo si trova nella propria sezione critica, qualsiasi altro processo che tenti di entrare nella sezione critica si trova nel ciclo del codice della entry section
- ✱ Il *busy waiting* o **attesa attiva** costituisce un problema per i sistemi multiprogrammati, perché la condizione di attesa spreca cicli di CPU che altri processi potrebbero sfruttare produttivamente
- ✱ L'attesa attiva è vantaggiosa solo quando è molto breve perché, in questo caso, può evitare un context switch





Implementazione dei semafori – 1

- ✱ Per evitare di lasciare un processo in attesa nel ciclo *while* si può ricorrere ad una definizione alternativa di semaforo
 - ✗ Associata ad ogni semaforo, vi è una coda di processi in attesa
 - ✗ La struttura semaforo contiene:
 - un valore intero (numero di processi in attesa)
 - un puntatore alla testa della lista dei processi
- ⇒ Si definisce un semaforo come un record:

PCB ⇒

```
typedef struct {  
    int valore;  
    struct processo *lista;  
} semaforo;
```





Implementazione dei semafori – 2

- ✱ Si assume che siano disponibili due operazioni (fornite dal SO come system call):
 - ✗ *block* – posiziona il processo che richiede di essere bloccato nell'opportuna coda di attesa, ovvero sospende il processo che la invoca
 - ✗ *wakeup* – rimuove un processo dalla coda di attesa e lo sposta nella ready queue





Implementazione dei semafori – 3

- ✱ Le operazioni sui semafori possono essere definite come (S.valore inizializzato a 1)...

```
void wait(semaforo S) {  
    S.valore--;  
    if (S.valore < 0) {  
        aggiungi il processo P a S.lista;  
        block();  
    }  
}
```

```
void signal(semaforo S) {  
    S.valore++;  
    if (S.valore <= 0) {  
        rimuovi il processo P da S.lista;  
        wakeup(P);  
    }  
}
```





Implementazione dei semafori – 4

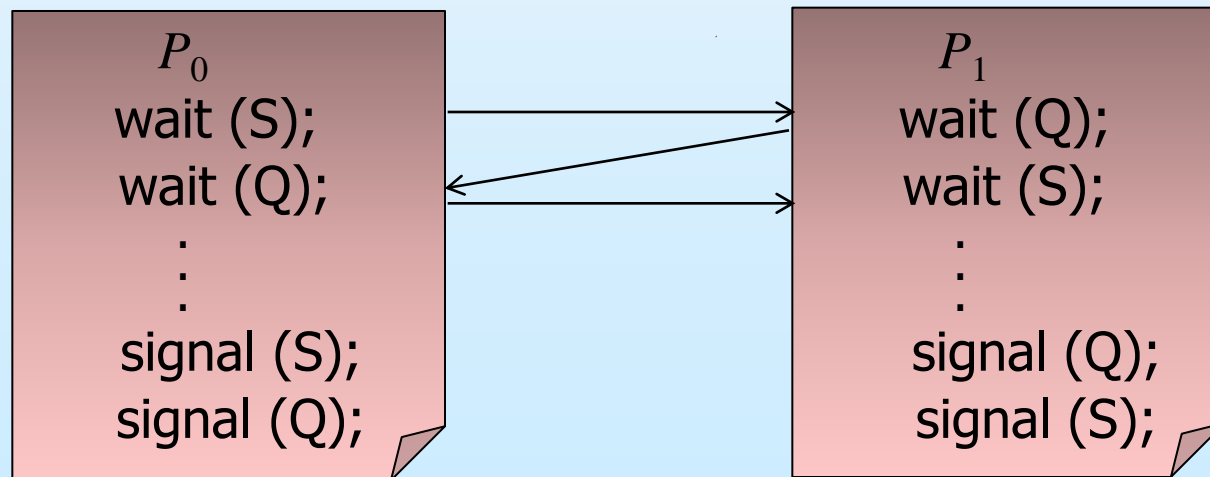
- ☀ Mentre la definizione classica di semaforo ad attesa attiva è tale che il valore del semaforo non è mai negativo, il semaforo appena descritto può assumere valori negativi (nel campo *valore*)
- ☀ Se *valore* è negativo $|valore|$ è il numero dei processi che attendono al semaforo
 - ✗ Ciò avviene a causa dell'inversione dell'ordine delle operazioni di decremento e verifica del valore nella *wait()*





Deadlock

- ✱ La realizzazione di un semaforo con coda di attesa può condurre a situazioni in cui ciascun processo attende l'esecuzione di un'operazione *signal()*, che solo uno degli altri processi in coda può causare
- ✱ Più in generale, si verifica una situazione di **deadlock**, o **stallo**, quando due o più processi attendono indefinitamente un evento che può essere causato soltanto da uno dei due processi in attesa
- ✱ Siano S e Q due semafori inizializzati entrambi ad 1





Starvation

- ✱ **Starvation** – l'attesa indefinita (senza blocco) nella coda di un semaforo si può verificare qualora i processi vengano rimossi dalla lista associata al semaforo in modalità LIFO (*Last-In-First-Out*)
- ✱ Viceversa, per aggiungere e togliere processi dalla "coda al semaforo", assicurando un'attesa limitata, la lista può essere gestita in modalità FIFO





Sezione critica e kernel – 1

- ✱ In un dato istante, più processi in modalità utente possono richiedere servizi al SO
- ⇒ Il codice del kernel deve regolare gli accessi a dati condivisi
- ✱ **Esempio:**
 - ✗ Una struttura dati del kernel mantiene una lista di tutti i file aperti nel sistema
 - Deve essere modificata ogniqualvolta si aprono/chiudono file
 - Due processi che tentano di aprire file in contemporanea potrebbero ingenerare nel sistema una corsa critica





Sezione critica e kernel – 2

✿ Kernel con diritto di prelazione

- ✗ Critici per sistemi SMP, in cui due processi in modalità kernel possono essere eseguiti su processori distinti
- ✗ Necessari in ambito real-time: permettono ai processi in tempo reale di far valere il loro diritto di precedenza
- ✗ Utili nei sistemi time-sharing: diminuiscono il tempo medio di risposta
- ✗ Linux, dal kernel ver. 2.6

✿ Kernel senza diritto di prelazione

- ✗ Immuni dai problemi legati all'ordine degli accessi alle strutture dati del kernel
- ✗ Windows XP/2000





Inversione di priorità – 1

- ✱ Quando: un processo a priorità più alta ha bisogno di leggere o modificare dati a livello kernel utilizzati da un processo, o da una catena di processi, a priorità più bassa
- ✱ Perché: poiché i dati a livello kernel sono tipicamente protetti da un lock, il processo a priorità maggiore dovrà attendere finché il processo a priorità minore non avrà finito di utilizzare le risorse
- ✱ Esempio famoso: **Mars Pathfinder/Sojourner** (1997)





Inversione di priorità – 2

✿ Esempio

- ✗ Siano dati 3 task, T_1, T_2, T_3 , con rispettive priorità $p_1 > p_2 > p_3$
- ✗ Supponiamo che T_1 e T_3 debbano accedere alla risorsa a cui è associato il mutex S
- ✗ Se T_3 inizia la sua fase di elaborazione ed esegue una *wait(S)* prima che T_1 inizi la propria, quando T_1 tenterà di eseguire *wait(S)* verrà bloccato per un tempo non definito, cioè fino a quando T_3 non eseguirà *signal(S)*
 - T_1 viene penalizzato a favore di T_3 a dispetto delle priorità $p_1 > p_3$ (**blocco diretto**)
- ✗ Se, prima che T_3 esegua *signal(S)*, inizia l'esecuzione di T_2 , T_3 verrà sospeso per permettere l'elaborazione di $T_2 \Rightarrow T_1$ dovrà attendere che anche T_2 finisca (**blocco indiretto**)





Inversione di priorità – 3

✱ Soluzioni

- ✗ Limitare a due il numero di possibili valori di priorità (poco realistico)
- ✗ Implementare un **protocollo di ereditarietà delle priorità**
 - Tutti i processi che stanno accedendo a risorse di cui hanno bisogno processi con priorità maggiore ereditano la priorità più alta finché non rilasciano l'uso delle risorse contese
 - Quando hanno terminato, la loro priorità ritorna al valore originale





Problemi classici di sincronizzazione

- ✱ Problema del produttore–consumatore con buffer limitato
- ✱ Problema dei lettori–scrittori
- ✱ Problema dei cinque filosofi





Problema del buffer limitato – 1

✱ Variabili condivise:

```
semaphore piene, vuote, mutex;
```

```
// inizialmente piene = 0, vuote = n, mutex = 1
```

- ✱ Il buffer ha n posizioni, ciascuna in grado di contenere un elemento
- ✱ Il semaforo *mutex* garantisce la mutua esclusione sugli accessi al buffer
- ✱ I semafori *vuote* e *piene* contano, rispettivamente, il numero di posizioni vuote ed il numero di posizioni piene nel buffer





Problema del buffer limitato – 2

```
do {  
    ...  
    produce un elemento in np  
    ...  
    wait(vuote);  
    wait(mutex);  
    ...  
    inserisce np nel buffer  
    ...  
    signal(mutex);  
    signal(piene);  
} while (1);
```

Processo produttore

```
do {  
    wait(piene)  
    wait(mutex);  
    ...  
    sposta un elemento dal buffer in nc  
    ...  
    signal(mutex);  
    signal(vuote);  
    ...  
    consuma un elemento in nc  
    ...  
} while (1);
```

Processo consumatore





Problema dei lettori–scrittori – 1

- ✿ Un insieme di dati (ad es., un file, una base di dati, etc.) deve essere condiviso da più processi concorrenti che possono:
 - ✗ richiedere la sola lettura del contenuto...
 - ✗ o l'accesso ai dati sia in lettura che in scrittura
- ✿ **Problema:** permettere a più lettori di accedere ai dati contemporaneamente; solo uno scrittore alla volta, viceversa, può accedere ai dati condivisi
- ✿ Variabili condivise (oltre ai dati):

semaphore mutex, scrittura;

int numlettori;

// inizialmente mutex = 1, scrittura = 1, numlettori = 0





Problema dei lettori–scrittori – 2

```
while (1) {  
    wait (mutex);  
    numlettori++;  
    if (numlettori==1) wait (scrittura);  
    signal (mutex)  
        // lettura  
    wait (mutex);  
    numlettori--;  
    if (numlettori==0) signal (scrittura);  
    signal (mutex);  
}
```

Processo lettore

```
while (1) {  
    wait (scrittura);  
    // scrittura  
    signal (scrittura);  
}
```

Processo scrittore

Possibile
starvation per gli
scrittori...





Problema dei lettori–scrittori – 3

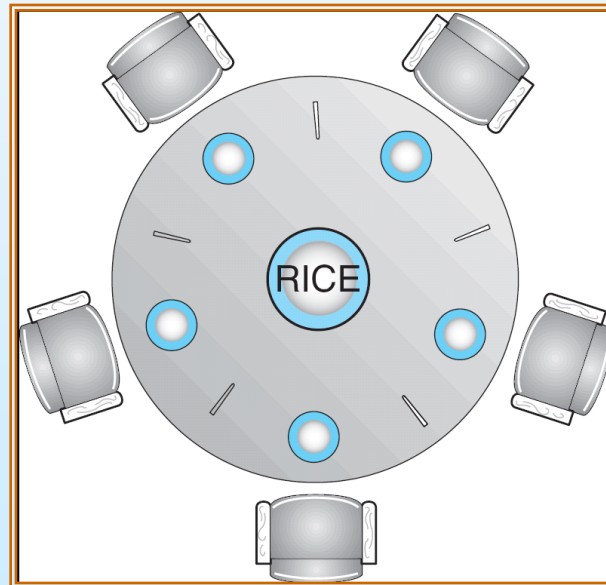
- ✱ Il semaforo *scrittura* è comune ad entrambi i tipi di processi
- ✱ Il semaforo *mutex* serve per assicurare la mutua esclusione all'atto dell'aggiornamento di *numlettori*
- ✱ La variabile *numlettori* contiene il numero dei processi che stanno attualmente leggendo l'insieme di dati
- ✱ Il semaforo *scrittura* realizza la mutua esclusione per gli scrittori e serve anche al primo ed all'ultimo lettore che entra ed esce dalla sezione critica
 - ✗ Non serve ai lettori che accedono alla lettura in presenza di altri lettori





Problema dei cinque filosofi – 1

- ✿ Cinque filosofi passano la vita pensando e mangiando, attorno ad una tavola rotonda
- ✿ Al centro della tavola vi è una zuppiera di riso e la tavola è apparecchiata con cinque bacchette





Problema dei cinque filosofi – 2

- ✿ Quando un filosofo pensa, non interagisce con i colleghi, quando gli viene fame, tenta di impossessarsi delle bacchette che stanno alla sua destra ed alla sua sinistra
- ✿ Il filosofo può appropriarsi di una sola bacchetta alla volta
- ✿ Quando un filosofo affamato possiede due bacchette contemporaneamente, mangia; terminato il pasto, appoggia le bacchette e ricomincia a pensare
- ✿ Variabili condivise:
 - ✗ La zuppiera di riso (l'insieme dei dati)
 - ✗ Cinque semafori, **bacchetta[5]**, inizializzati ad 1





Problema dei cinque filosofi – 3

- ❖ Codice per l' i -esimo filosofo

```
while (1) {  
    wait (bacchetta[i]);  
    wait (bacchetta[(i+1) % 5]);  
    // mangia  
    signal (bacchetta[i]);  
    signal (bacchetta[(i+1) % 5]);  
    // pensa  
}
```

- ❖ Soluzione che garantisce che due vicini non mangino contemporaneamente, ma non riesce ad evitare lo stallo





Problema dei cinque filosofi – 4

- ✱ **Stallo:** i filosofi hanno fame contemporaneamente ed ognuno si impossessa della bacchetta alla propria sinistra
- ✱ **Soluzioni:**
 - ✗ Solo quattro filosofi possono stare a tavola contemporaneamente
 - ✗ Un filosofo può prendere le bacchette solo se sono entrambe disponibili
 - ✗ I filosofi di posto “pari” raccolgono prima la bacchetta alla loro sinistra, quelli di posto “dispari” la bacchetta alla loro destra





Errori comuni

✿ Errori comuni nell'uso dei semafori:

✗ *signal(mutex) ... wait(mutex)*

inversione delle chiamate di sistema: più processi possono eseguire le proprie sezioni critiche in contemporanea

✗ *wait(mutex) ... wait(mutex)*

doppia chiamata di una stessa system call: si genera un deadlock

✗ Omissione di *wait(mutex)*, *signal(mutex)* o di entrambe
violazione della mutua esclusione o stallo





Esempio 1

- ✱ Si considerino due processi, un produttore P ed un consumatore C, ed un buffer con capacità pari ad 1

- ✱ Sia

```
semaphore empty=1;
```

l'unica variabile condivisa (oltre al buffer)

- ✱ Si considerino le seguenti porzioni di codice includenti la sezione critica per l'accesso al buffer da parte di P e C

Processo P

```
while(true) {  
p1      wait(empty);  
p2      enter_item(item);  
}
```

Processo C

```
while(true) {  
c1      remove_item(item);  
c2      signal(empty);  
}
```

- ✱ Supponendo che le istruzioni p1 e p2 vengano eseguite per prime, il sistema si comporta correttamente come un produttore-consumatore? Descrivere una sequenza di esecuzione delle istruzioni di P e di C non corretta





Esempio 1 (Cont.)

- ✱ Il comportamento non è corretto, non è possibile realizzare il meccanismo produttore–consumatore con un solo semaforo
- ✱ Sequenza scorretta
 - ✗ $p1, p2, c1, c2, c1$: attenzione, il consumatore consuma da un buffer vuoto!





Esempio 2

- ✱ Si consideri il seguente programma concorrente:

Variabili condivise
semaphore $s1:=0, s2:=0$;

```
process P1 {  
    signal(s1);  
    wait(s2);  
    print "P"  
    wait(s2);  
    print "R"  
    signal(s1);  
}
```

```
process P2 {  
    signal(s2);  
    wait(s1);  
    print "A";  
    signal(s2);  
    wait(s1);  
    print "I";  
}
```

- ✱ Descrivere il comportamento ed i possibili output dell'esecuzione parallela di P1 e P2





Esempio 2 (cont.)

- ✱ Inizialmente uno qualsiasi tra P1 e P2 può essere selezionato per l'esecuzione della prima istruzione
- ✱ Il processo P1 deve attendere l'esecuzione di *signal(s2)* da parte di P2 e viceversa P2 attende l'esecuzione di *signal(s1)*
- ✱ L'esecuzione delle prime due stampe "P" in P1 e "A" P2 può avvenire in qualsiasi ordine; dopo aver stampato "P", P1 attende che P2 esegua *signal(s2)* e quindi stampa "R" e "segnala" sul semaforo s1 sul quale attende P2
- ✱ Solo dopo la *signal(s1)* eseguita da P1, P2 può stampare "I"
- ✱ I possibili output sono quindi due: la stringa "PARI" e la stringa "APRI"





Esempio 3

- ✱ **Il problema delle molecole di H_2O** – Un sistema è costituito da due processi concorrenti: un processo costruttore di atomi di idrogeno e un processo costruttore di atomi di ossigeno. Ogni processo ciclicamente produce un atomo e poi invoca una procedura comune (nell'esempio è una procedura di stampa). Si vuole che l'attività dei due processi sia sincronizzata in modo che le chiamate di questa procedura siano eseguite con gli input nell'ordine HHOHHOHHO...

- ✱ Siano

semaphore $H_y=0$, $Oxy=2$;

le variabili condivise





Esempio 3 (Cont.)

- ✱ I processi Idrogeno ed Ossigeno possono essere descritti come segue:

Processo Idrogeno

```
{  
while(1)  
{  
    <make a hydrogen atom>  
    wait(Oxy);  
    write(H);  
    signal(Hy);  
}  
}
```

Processo Ossigeno

```
{  
while(1)  
{  
    <make an oxygen atom>  
    wait(Hy);  
    wait(Hy);  
    write(O);  
    signal(Oxy);  
    signal(Oxy);  
}  
}
```





Esempio 4

- ✱ Una tribù di N selvaggi mangia in comune da una pentola che può contenere fino ad M (con $M < N$) porzioni di stufato di missionario; quando un selvaggio ha fame controlla la pentola: se non ci sono porzioni, sveglia il cuoco ed attende che questo abbia riempito di nuovo la pentola; se la pentola contiene almeno un pasto, se ne appropria. Il cuoco controlla inizialmente che ci siano delle porzioni e, se ci sono, si addormenta, altrimenti cuoce M porzioni e le serve ai cannibali in attesa
- ✱ Si descriva una soluzione al problema che impieghi i semafori
- ✱ Siano:

```
semaphore mutex=1, pieno=0, vuoto=0;  
int porzioni=M;
```

le risorse condivise





Esempio 4 (Cont.)

- ✱ I processi Cuoco e Cannibale sono:

Processo Cuoco

```
{  
  ...  
  while(1)  
  {  
    wait(vuoto);  
    <riempi la pentola>  
    signal(pieno);  
  }  
}
```

Processo Cannibale

```
{  
  ...  
  wait(mutex);  
  if (porzioni==0)  
  {  
    signal(vuoto);  
    wait(pieno);  
    porzioni = M;  
  }  
  porzioni—;  
  signal(mutex);  
  <mangia porzione>  
  ...  
}
```





Esercizio 1

✱ Variabili condivise:

```
semaphore S=1, M=1;  
int x=10;
```

```
{  
    wait(M);  
    x=x+1;  
    signal(M);  
  
    wait(M);  
    write(x);  
    signal(M);  
  
    signal(S);  
}
```

Processo P_1

```
{  
    wait(S);  
  
    wait(M);  
    x=x-1;  
    signal(M);  
  
    wait(M);  
    write(x);  
    signal(M);  
}
```

Processo P_2





Esercizio 1 (Cont.)

- ✱ Individuare le regioni critiche nel codice di P_1 e P_2 ; si possono verificare race condition per la variabile condivisa x ?
- ✱ Determinare tutti i possibili output del programma concorrente (P_1 in parallelo con P_2)
- ✱ Supponiamo di inizializzare il semaforo S a 0 invece che a 1; determinare tutti i possibili output del programma (P_1 in parallelo con P_2) così modificato





Esercizio 2

* Variabili condivise:

```
semaphore S=2, T=0;  
int x=0;
```

```
{  
    wait(T);  
    x=5;  
    signal(S);  
    x=6;  
}
```

Processo P_1

```
{  
    wait(S);  
    x=1;  
    signal(T);  
    wait(S);  
    x=10;  
    write(x);  
}
```

Processo P_2





Esercizio 2 (Cont.)

- ✱ Si supponga che i processi vengano eseguiti concorrentemente sulla stessa CPU
 - ✗ Determinarne tutti i possibili output
 - ✗ Cosa succede se inizialmente i semafori hanno valore $S=1$ e $T=0$?
 - ✗ Nel caso $S=1$ e $T=0$, inserire una sola nuova chiamata *wait* su S o T in uno dei processi P_1 e P_2 in modo da ottenere come unico possibile output il valore 10





Semafori e monitor

- ✱ Benché i semafori costituiscano un meccanismo pratico ed efficace per la sincronizzazione dei processi, il loro uso scorretto può generare errori difficili da individuare, in quanto si manifestano solo in presenza di particolari sequenze di esecuzione
- ✱ I monitor semplificano molto la gestione della mutua esclusione (meno possibilità di errore)
- ✱ Sono veri costrutti, non funzioni di libreria \Rightarrow modifica dei compilatori
- ✱ Problema comune sia con i semafori che con i monitor: è necessario avere memoria condivisa (costrutti non applicabili in sistemi distribuiti)





I monitor – 1

- ✱ Il monitor è un costrutto di sincronizzazione di alto livello che permette la condivisione sicura di un tipo astratto di dati fra processi concorrenti
- ✱ Incapsula i dati privati mettendo a disposizione metodi pubblici per operare su tali dati
- ✱ La rappresentazione di un tipo monitor è formata da:
 - ✗ dichiarazioni di variabili i cui valori definiscono lo stato di un'istanza del tipo
 - ✗ procedure o funzioni che realizzano le operazioni sul tipo stesso
- ✱ Supportati (implementati in certa misura) da Concurrent Pascal, C#, Java (metodi *synchronized*)

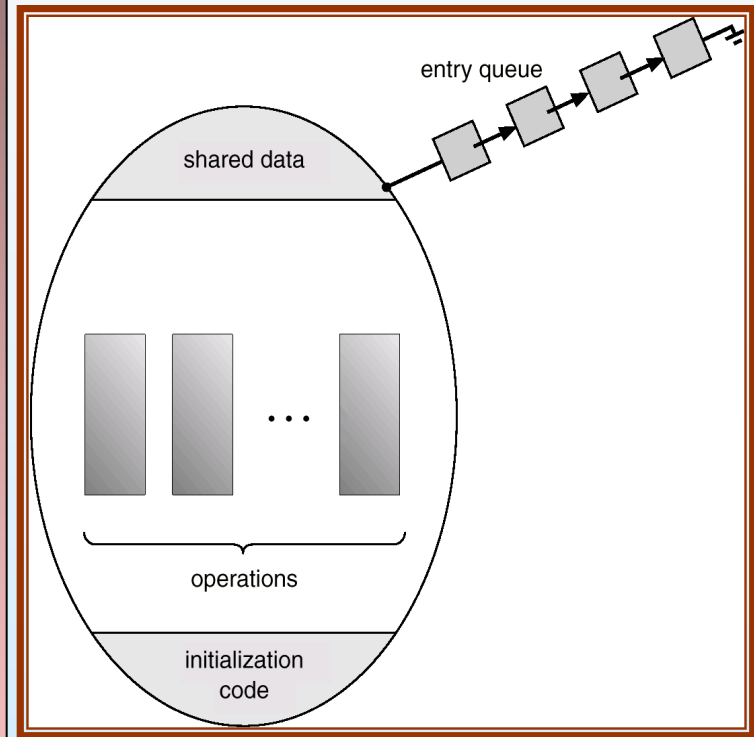




I monitor – 2

```
monitor monitor-name
{
    dichiarazione delle variabili condivise

    procedure  $P_1$  (...) {
        ...
    }
    procedure  $P_2$  (...) {
        ...
    }
    procedure  $P_n$  (...) {
        ...
    }
    codice di inizializzazione
}
```





I monitor – 3

- ✱ Per effettuare l'allocazione di risorse tramite monitor, un processo deve richiamare una routine d'ingresso al monitor
 - ✱ La mutua esclusione è rigidamente forzata ai confini del monitor stesso, dove può accedere un solo processo alla volta
 - ✱ Tuttavia, tale definizione di monitor non è sufficientemente potente per modellare alcuni schemi di sincronizzazione
- ⇒ per permettere ad un processo di attendere dopo l'ingresso al monitor, causa occupazione della risorsa richiesta, si dichiarano apposite variabili *condition*

`condition x,y;`





I monitor – 4

- ✱ Le variabili *condition* possono essere usate solo in relazione a specifiche operazioni *wait* e *signal*

- ✗ L'operazione

x.wait();

fa sì che il processo chiamante rimanga sospeso fino a che un diverso processo non effettui la chiamata...

x.signal();

- ✗ L'operazione *x.signal* attiva esattamente un processo sospeso; se non esistono processi sospesi l'operazione *signal* non ha alcun effetto





I monitor – 5

- ✿ Un processo utilizza una variabile *condition* (interna al monitor) per attendere il verificarsi di una certa condizione, uscendo dal monitor nell'attesa
- ✿ Un monitor associa una diversa variabile *condition* ad ogni situazione che potrebbe determinare un processo in attesa
- ✿ Un processo può uscire dal monitor mettendosi in attesa su una variabile *condition* o completando l'esecuzione del codice protetto dal monitor





I monitor – 6

- ✱ Le variabili *condition* si differenziano dalle variabili convenzionali poiché ognuna di esse ha una coda associata
 - ✗ Un processo che richiama una *wait* su una variabile *condition* particolare è posizionato nella sua coda e, finché vi rimane, è al di fuori del monitor, in modo che un altro processo possa entrarvi e richiamare una *signal*
 - ✗ Un processo che richiama una *signal* su una variabile *condition* particolare fa sì che un processo in attesa su quella variabile sia rimosso dalla coda associata e possa accedere al monitor
- ✱ Gestione FIFO o a priorità delle code dei processi





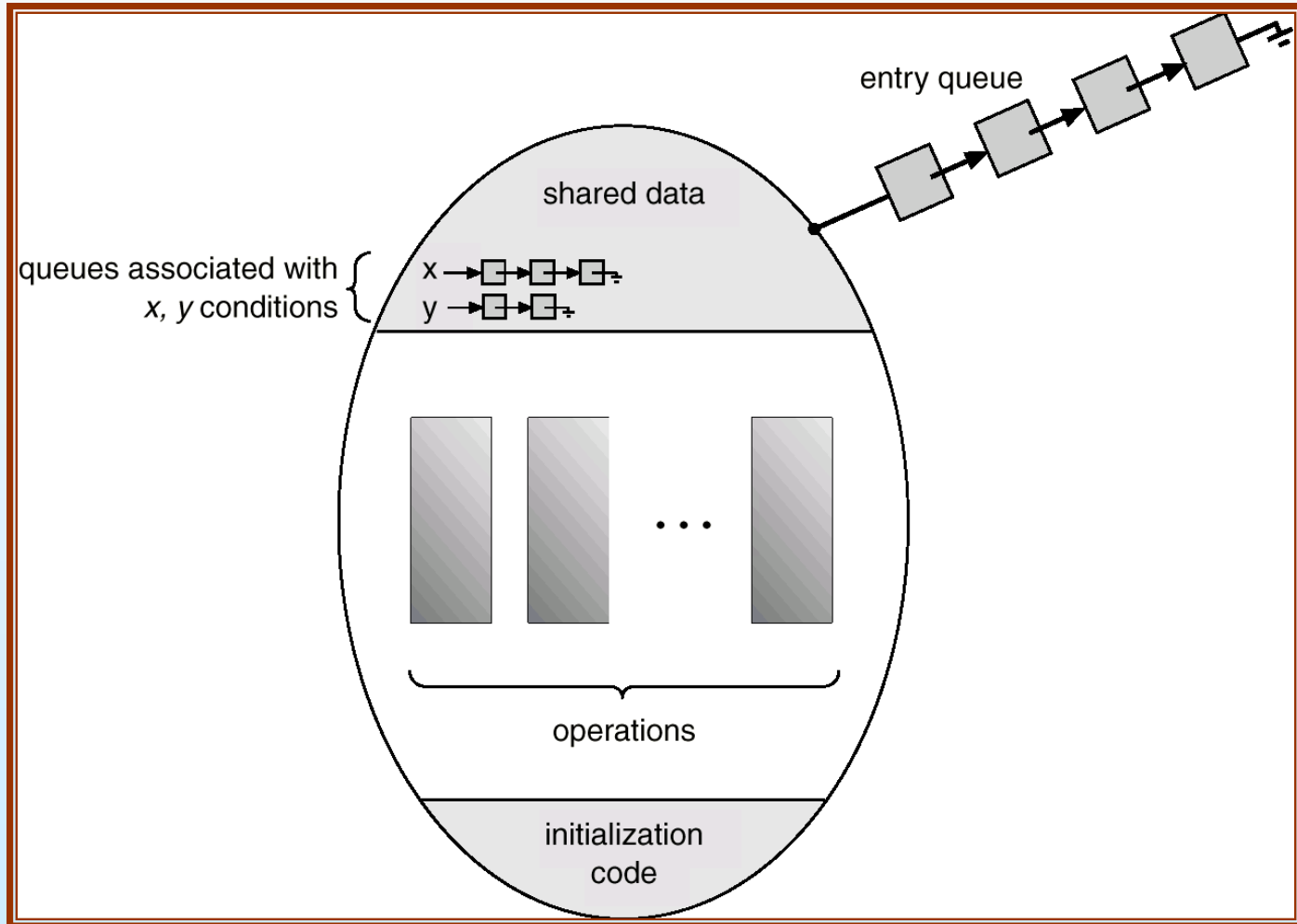
I monitor – 7

- ✱ Se il processo P invoca *signal* su una variabile *condition* ed esiste un processo Q in coda su quella variabile...
 - ✗ **Segnalare ed attendere**: nel monitor entra Q e P attende (fuori dal monitor) che Q abbia terminato o si metta in attesa su una *condition*
 - ✗ **Segnalare e proseguire**: Q attende che P lasci il monitor o si metta in attesa su una *condition*
- ✱ Soluzione adottata da Concurrent Pascal: P deve eseguire la *signal* come ultima istruzione prima di uscire dal monitor





Monitor con variabili condition





Soluzione del problema dei cinque filosofi – 1

```
monitor filosofo
{
    enum{PENSA, AFFAMATO, MANGIA} stato[5] ;
    condition self[5];

    void prende (int i) {
        stato[i] = AFFAMATO;
        test(i);
        if (stato[i] != MANGIA) self[i].wait;
    }

    void posa (int i) {
        stato[i] = PENSA;
        // controlla i vicini a sinistra ed a destra
        test((i+4) % 5);
        test((i+1) % 5);
    }
}
```





Soluzione del problema dei cinque filosofi – 2

```
void test (int i) {  
    if ( (stato[(i+4) % 5] != MANGIA) &&  
        (stato[i] == AFFAMATO) &&  
        (stato[(i+1) % 5] != MANGIA) ) {  
        stato[i] = MANGIA;  
        self[i].signal() ;  
    }  
}  
  
codice_iniz() {  
    for (int i = 0; i < 5; i++)  
        stato[i] = PENSA;  
}  
}
```





Soluzione del problema dei cinque filosofi – 3

- ✱ Il filosofo i -esimo invocherà le operazioni *prende()* e *posa()* nell'ordine:

```
filosofo.prende(i)
... ..
mangia
... ..
filosofo.posa(i)
```

- ✱ La soluzione non provoca deadlock, ma la starvation è ancora possibile!





Esempio:

Monitor per la gestione di mailbox – 1

- ✱ Utilizziamo il costrutto monitor per risolvere il problema della comunicazione tra processi:
 - ✗ il monitor incorpora il buffer dei messaggi (gestito in modo circolare)
 - ✗ i processi produttori (o consumatori) inseriranno (o preleveranno) i messaggi mediante le funzioni *send* (o *receive*) definite nel monitor
- ✱ La struttura dati che rappresenta il buffer fa parte delle variabili locali al monitor e quindi le operazioni *send* e *receive* possono accedere solo in modo mutuamente esclusivo a tale struttura





Esempio:

Monitor per la gestione di mailbox – 2

```
monitor buffer_circolare
{
    messaggio buffer[N];
    int contatore=0; int testa=0; int coda=0;
    condition pieno;
    condition vuoto;

    void send(messaggio m)
    {
        if (contatore==N) pieno.wait;
        buffer[coda]=m;
        coda=(coda + 1)%N;
        ++contatore;
        vuoto.signal;
    }
}
```

```
messaggio receive()
{
    messaggio m;
    if (contatore == 0) vuoto.wait;
    m=buffer[testa];
    testa=(testa + 1)%N;
    --contatore;
    pieno.signal;
    return m;
}
/* fine monitor */
```





Esempio: Parcheggio – 1

- ✱ Il gestore di un parcheggio vuole automatizzare il sistema di ingresso/uscita in modo da impedire l'accesso quando il parcheggio è pieno (cioè la sbarra d'ingresso deve sollevarsi solo se ci sono posti disponibili) e da consentire l'uscita solo in caso di parcheggio non vuoto (cioè la sbarra d'uscita deve rimanere abbassata in caso di parcheggio vuoto)
- ✱ Scrivere un monitor che permetta di gestire l'accesso al parcheggio





Esempio: Parcheggio – 2

```
monitor CarPark
  condition notfull, notempty;
  integer spaces, capacity;

  procedure enter();
  begin
    while(spaces=0) do notfull.wait;
    spaces := spaces-1;
    notempty.signal;
  end;
  procedure exit();
  begin
    while(spaces=capacity) do notempty.wait;
    spaces := spaces+1;
    notfull.signal;
  end;

  spaces := N;
  capacity := N;
end monitor;
```





Sincronizzazione in Pthreads

- ✿ La API Pthreads fornisce diversi tipi di lock e variabili condizionali per la sincronizzazione dei thread
- ✿ I lock mutex rappresentano la tecnica di sincronizzazione fondamentale
 - ✗ Un thread, che sia in procinto di accedere alla propria sezione critica, si appropria del lock, salvo rimuoverlo all'uscita
- ✿ I semafori non rientrano nello standard Pthreads, appartengono invece all'estensione POSIX SEM (non totalmente portabile)

