Meccanismi di sincronizzazione: i monitor

Il monitor è un *costrutto linguistico* per creare nuovi tipi di dato, simile alla *classe* in un linguaggio ad oggetti

Useremo la seguente *sintassi* per la definizione:

monitor *nome_monitor*

dichiarazione di variabili condivise, interne al monitor

    // modificabili solo tramite le procedure del monitor stesso

nome_proc(*parametri*) // procedure *pubbliche* del monitor

    body della procedura;

    ...

…. // eventuali procedure private

…. ; // codice di inizializzazione

end

Meccanismi di sincronizzazione: i monitor

Permette di sincronizzare l'esecuzione delle funzioni definite per quel tipo di dato con due meccanismi, uno automatico, l'altro no

1) L'esecuzione su una stessa struttura dati (oggetto) di tipo «monitor» di procedure (metodi) da parte di processi diversi avviene in **mutua esclusione**, sia nel caso in cui la procedura richiesta sia la stessa, sia nel caso i due processi vogliano eseguire procedure diverse

La mutua esclusione è automatica, sarà il compilatore del linguaggio ad inserire le opportune chiamate del sistema operativo all'inizio e alla fine di ogni procedura

Questo risolve automaticamente problemi come quello del conto corrente, definendo procedure per:

- prelievo
- deposito
- saldo

che verranno automaticamente eseguite in mutua esclusione

Meccanismi di sincronizzazione: i monitor

2) Un altro *tipo di sincronizzazione* è da programmare esplicitamente con:

- variabili di tipo *condition*
- l'operazione *wait(cond)* su una variabile *condition,* operazione che sospende il processo, in attesa che venga «segnalata» *cond*; viene *rilasciata la mutua esclusione*
- l'operazione *signal(cond)* che risveglia, se esiste, un processo in attesa su cond

N.B. 1: affinché il processo risvegliato possa girare, deve riprendere la mutua esclusione, in particolare rispetto al processo che esegue *signal* (sono possibili diverse scelte, es.: signal comporta automaticamente l'uscita dalla procedura e il «passaggio» della mutua esclusione al processo svegliato)

N.B. 2: si chiamano condizioni, ma non sono vere/false; sono un meccanismo di sincronizzazione che *può* essere usato per attendere che una condizione booleana diventi vera/falsa, e segnalare che lo è diventata

Meccanismi di sincronizzazione: i monitor

N.B. 3: non c'è il problema della «perdita della segnalazione» come per sleep/wakeup?
Consideriamo una istruzione:

if (*espressione booleana)* wait(cond)

per via della mutua esclusione, se P1 esegue tale codice, non può capitare che P2 effettui *signal(cond) dopo* che P1 ha valutato l'espressione ma *prima* che arrivi a *wait(cond)*

Esempio di utilizzo dei monitor: il produttore e il consumatore

```
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;
```

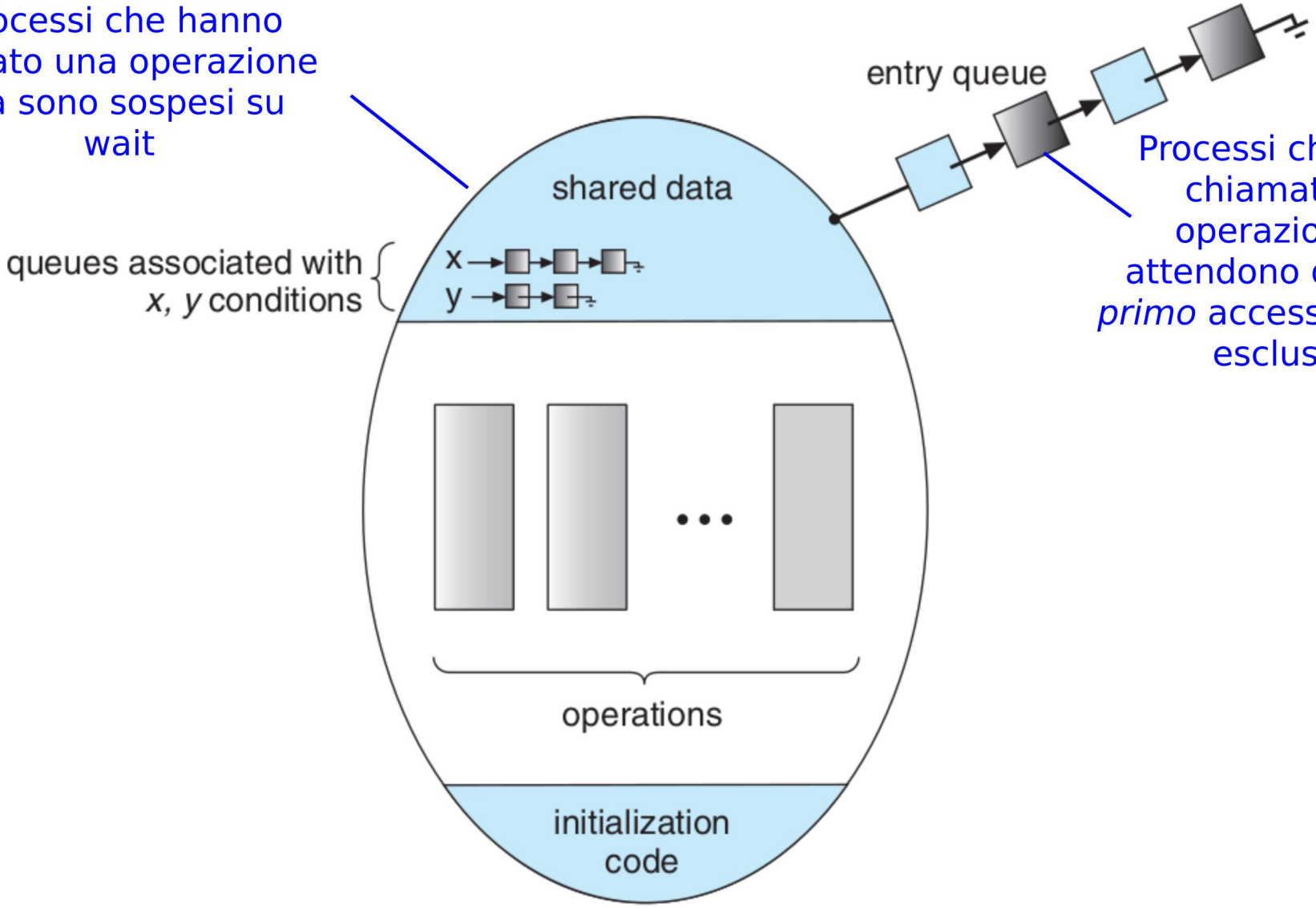
```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

Attenzione ai nomi, qui wait(full) = aspetta *perché* è pieno = aspetta che sia *non pieno*

il while al posto di if serve se non è garantito che il processo risvegliato abbia subito l'accesso in mutua esclusione (anche ad esempio rispetto ad altri processi produttori, per un produttore) e quindi, ad esempio, non sia più *count = N* quando il produttore che ha ricevuto la segnalazione su "full" ottiene la CPU

Meccanismi di sincronizzazione: i monitor

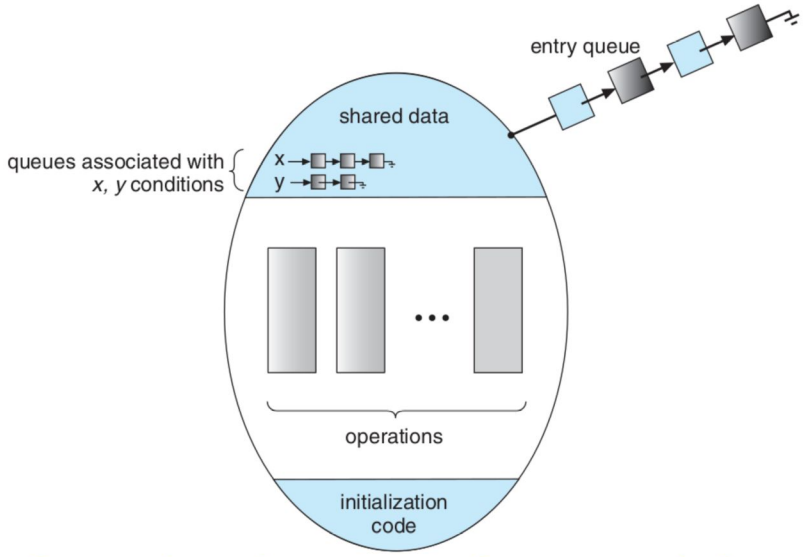
Processi che hanno iniziato una operazione ma sono sospesi su wait



Processi che hanno chiamato una operazione ma attendono di avere il *primo* accesso in mutua esclusione

Meccanismi di sincronizzazione: i monitor

Se P1 fa *signal(cond)* e ci sono processi in attesa su *cond*, ne viene scelto uno, P2, che non sarà più in attesa su *cond;*



ma non possono procedere entrambi ad operare sulle variabili condivise; ci sarebbero 3 soluzioni:

1. Una *signal* comporta l'uscita dalla procedura (va bene nel caso del produttore-consumatore, è l'ultima operazione)
2. P2 diventa «pronto», P1 viene parcheggiato in una ulteriore coda di processi, uno dei quali riacquisirà la mutua esclusione (meglio se non competendo anche con quelli nella «entry queue») quando verrà rilasciata
3. P1 continua, P2 rimane in attesa della mutua esclusione

La sincronizzazione di thread in Java

In un programma Java si possono creare thread multipli

- nelle classi si possono dichiarare metodi *synchronized*
  - l'esecuzione di metodi *synchronized* su uno stesso oggetto da parte di thread diversi avviene in mutua esclusione
  - per una stessa classe vi possono essere metodi *synchronized* e non
- non ci sono variabili condizione, ma i metodi *wait() , notify()* e *notifyAll():*
  - c'è una unica coda di attesa (come se ci fosse una sola condizione)*;*
  - *wait()* sospende il thread rilasciando la mutua esclusione
  - *notify()* ne «sveglia» uno (se c'è), *notifyAll()* tutti;
  - «sveglia» nel senso che per girare devono riprendere la mutua esclusione
  - essendoci un'unica coda e non potendo fare assunzioni sulla gestione della mutua esclusione, *in generale* è bene usare *while (...) wait(),* non *if (...) wait()*

Deadlock

Definizione:

un insieme di processi è in deadlock (stallo) quando ciascuno è in attesa di un evento che può essere causato solo da un altro processo dell'insieme

Nessuno dei processi può girare,

quindi:

nessuno può causare eventi,

quindi:

tutti rimangono sospesi

Tipi di risorse

Molti fenomeni di deadlock coinvolgono risorse che devono essere acquisite:

- Records in DB
- CPU
- Lettore blue ray

Le risorse possono essere di due tipi:

- Prelazionabili (preemptable) – può essere requisita al processo che la detiene senza effetti indesiderati
  - Memoria
  - CPU

- Non prelazionabili (nonpreemptable) – non può essere requisita senza causare effetti indesiderati (es. falimenti)
  - Blue ray
  - Stampante

Deadlock

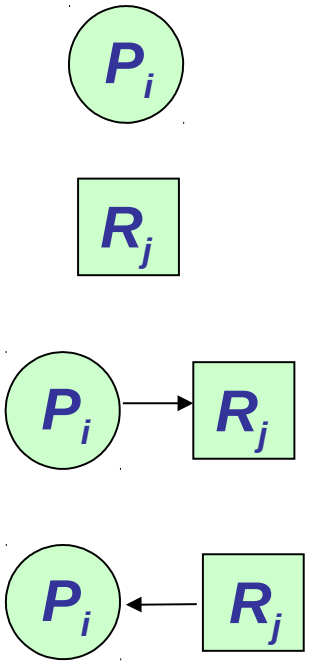
Avviene in particolare nel caso di allocazione di risorse; in questo caso l'evento che i processi attendono è il rilascio di una risorsa da parte di un altro processo

Affinché vi sia deadlock per l'allocazione di risorse devono valere 4 condizioni (tutte necessarie):

1. le risorse sono allocate in mutua esclusione (e se no, perché un processo dovrebbe attendere)
2. le risorse non sono *preemptive:* non ha senso portarle via al processo che le sta usando (es. stampante; mentre per la CPU il costo è accettabile)
3. un processo a cui sono allocate risorse ne può richiedere altre (*hold and wait*, allocazione parziale)
4. si ha attesa circolare nel senso già indicato: ogni processo attende una risorsa detenuta da un altro processo dell'insieme

Rappresentazione con grafo di allocazione delle risorse (una risorsa per tipo)

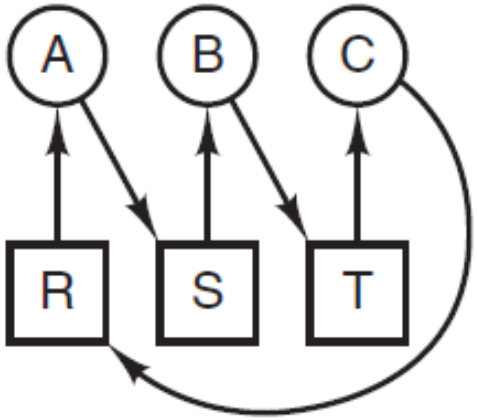
- Processo
- Risorsa
- P_i richiede R_j e rimane in attesa
- R_j allocata a P_i



Deadlock

Rappresentazione con grafo di allocazione delle risorse (una risorsa per tipo)

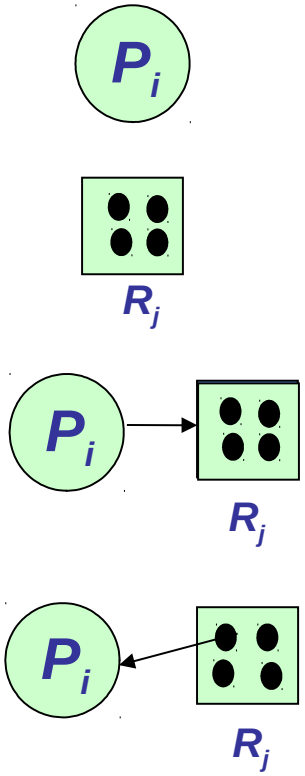
Vi è deadlock se e solo se c'è un ciclo nel grafo (la presenza di un ciclo è condizione necessaria e sufficiente)



Rappresentazione con grafo di allocazione delle risorse (con più risorse per tipo)

- Processo
- Tipo di risorse con 4 risorse
- P_i richiede una risorsa di tipo R_j e rimane in attesa

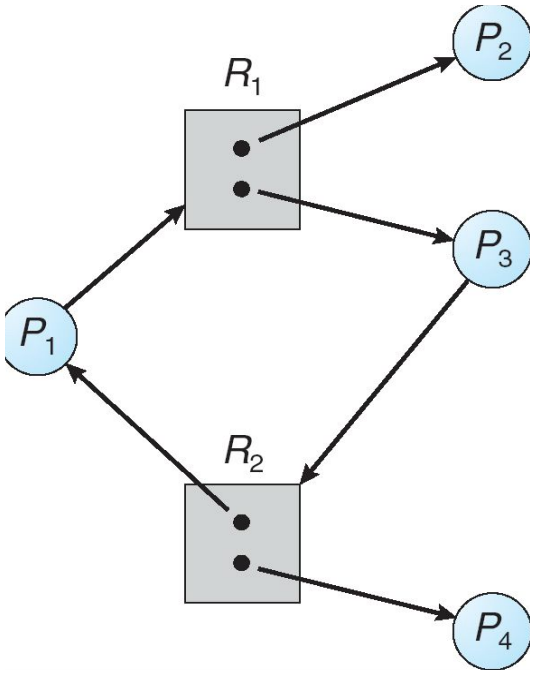
✂ P_i possiede una risorsa di tipo R_j



Deadlock

Rappresentazione con grafo di allocazione delle risorse (con più risorse per tipo)

La presenza di un ciclo (fra i nodi P_i , R_j) è condizione necessaria per avere deadlock, ma non sufficiente



Quando P2 rilascia R1,
o P4 rilascia R2,
si può continuare

Approcci al deadlock

- ignorare il problema
  es. quanto più spesso accade di un crash?
  questa è la soluzione adottata di fatto dai sistemi operativi
- algoritmi per rilevare il deadlock e approcci per recovery senza fare *reboot* (il modo più brutale è terminare qualche processo, sempre meglio che terminarli tutti)
- gestione delle risorse per prevenire il deadlock rendendo falsa una delle 4 condizioni; sebbene il sistema operativo non la imponga si possono seguirne i criteri nello scrivere i programmi
- algoritmi di allocazione delle risorse per evitare il deadlock, che possono rifiutare una allocazione se, usando informazione sulle richieste massime (*che devono essere dichiarate*) dei vari processi, verificano che è possibile che la situazione evolva in deadlock
  (es. algoritmo “del banchiere”)

Prevenzione del deadlock

Per qualche tipo di risorsa si può evitare di assegnarla direttamente, in mutua esclusione, ai processi utente

Es. per stampanti si fa *spooling:*

i processi scrivono (con nomi diversi) i file di output in una directory aggiornando (in mutua esclusione) una coda di file da stampare

un processo di sistema (“daemon”) periodicamente ispeziona il contenuto della coda e, una volta che c’è un file completo, lo invia alla stampante

NB: l’accesso in mutua esclusione è necessario solo per il tempo di aggiornamento della coda di stampa, *non per la stampa*

Prevenzione del deadlock

Per evitare *hold and wait* si può imporre di:

- chiedere tutte le risorse di cui si può avere bisogno prima di utilizzarne una, oppure
- se si detengono risorse e ne servono altre, prima si rilasciano quelle detenute (non è detto che abbia senso, per lo stesso motivo per cui non si può evitare che certe risorse siano “non preemptive”)

Per evitare *attesa circolare* si può imporre di richiedere le risorse in un ordine prefissato:

se nell’ordine A è prima di B, può capitare che un processo detenga A e attenda B, ma non viceversa (analogamente per A_1, ..., A_n)