



Modelli di interazione tra processi

Modelli di interazione tra processi

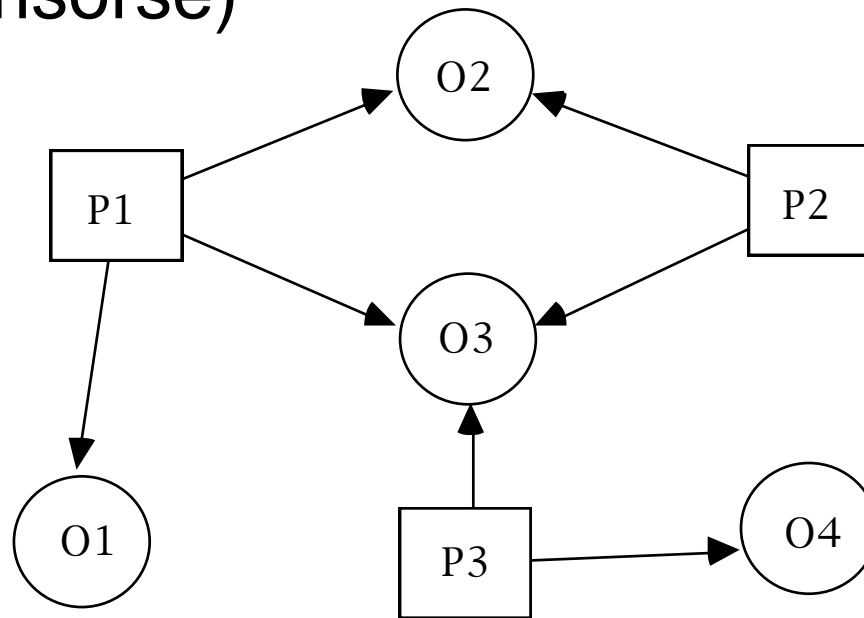
- Modello ad ambiente globale o modello a memoria comune (*global environment*)
- Modello ad ambiente locale o modello a scambio di messaggi (*message passing*)


Tipi di interazione tra processi:

- competizione
- cooperazione

Modello ad ambiente globale

- Il sistema è visto come un insieme di processi e oggetti (risorse)



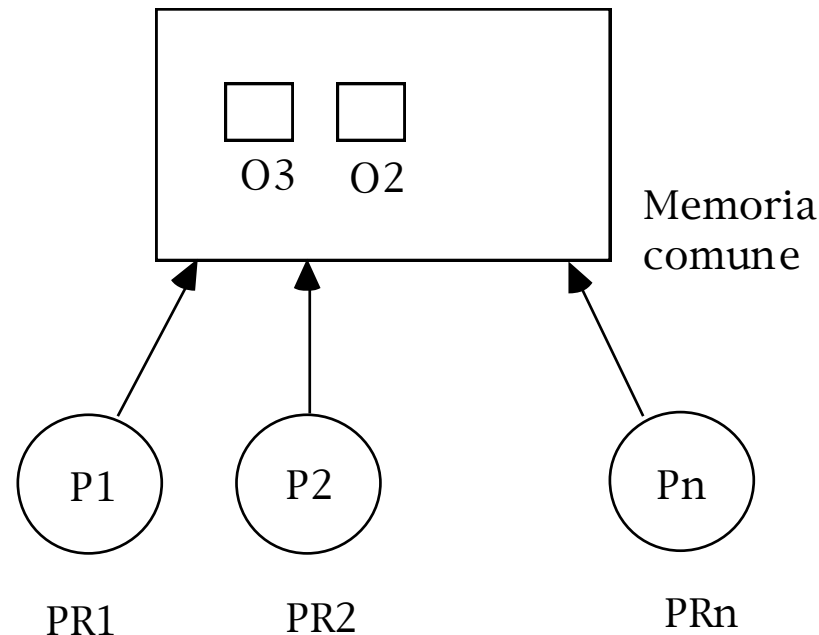
-  diritto di accesso
- O1, O4 risorse private (class)
- O2, O3 risorse comuni (monitor)

Modello ad ambiente globale

SisOp
2018/19



- Il modello ad ambiente globale rappresenta la naturale astrazione di un *sistema in multiprogrammazione* costituito da uno o più processori che hanno accesso ad una *memoria comune*.



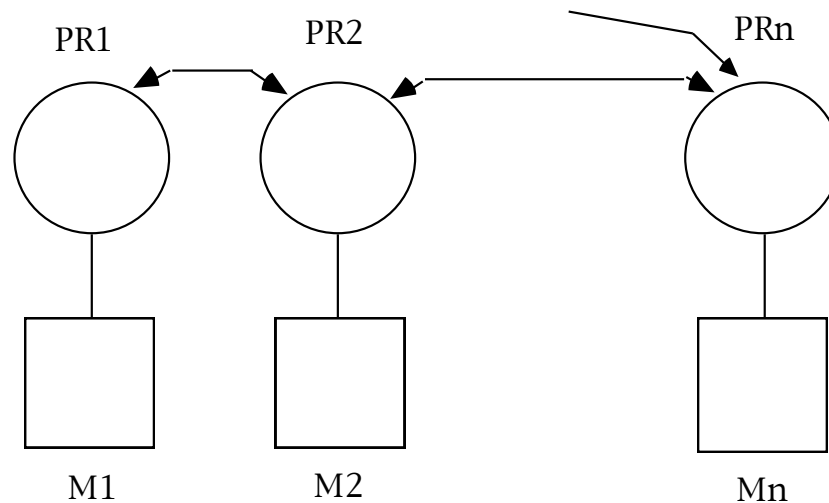
- Ad ogni processore può essere eventualmente associata una memoria privata, ma *ogni interazione* avviene tramite oggetti contenuti nella *memoria comune*.

Modello a scambio di messaggi

- Il sistema è visto come un insieme di processi ciascuno operante in un *ambiente locale* non accessibile direttamente a nessun altro processo.
- Ogni forma di interazione tra processi (comunicazione, sincronizzazione), avviene tramite *scambio di messaggi*.
- Non esiste il concetto di *risorsa accessibile direttamente ai processi*. Sono possibili due casi:
 - alla risorsa è associato un *processo servitore*
 - la risorsa viene passata da un processo ad un altro *sotto forma di messaggi*

Modello a scambio di messaggi

- Il modello a scambio di messaggi rappresenta la naturale astrazione di un sistema *privo di memoria comune* (sistema distribuito), in cui a ciascun processore è associata una memoria privata.



- Il modello a scambio di messaggi può essere realizzato anche in presenza di memoria comune, che viene utilizzata per realizzare canali di comunicazione.

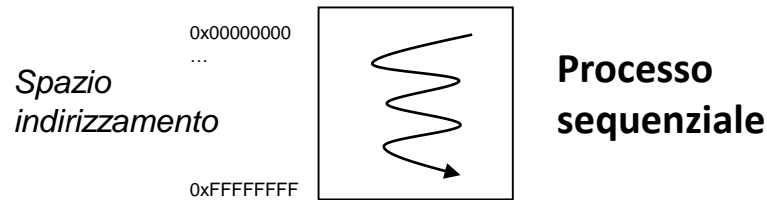
Modelli e strumenti per la interazione dei processi in ambiente globale

- Costrutti linguistici e modelli per la comunicazione e sincronizzazione tra processi in ambiente globale:
 - semafori e primitive di sincronizzazione (P, V) (Dijkstra, 1965)
 - regioni critiche e monitor (Brinch Hansen, Hoare, 1973)
 - path expressions (Campbell, Habermann, 1974)
 - serializer (Hewitt, Atkinson, 1977)
- Strumenti per programmazione in ambiente globale
 - Threads (più flussi di controllo per un unico spazio di indirizzamento) – ad es. la libreria Pthreads (Posix 1003.1c)
 - Java (semafori, monitor)

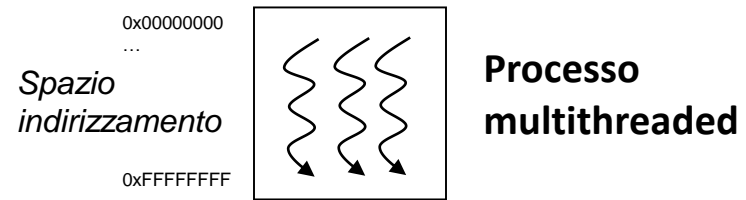
Entrambi gli strumenti permettono anche lo scambio di messaggi

Processi e thread

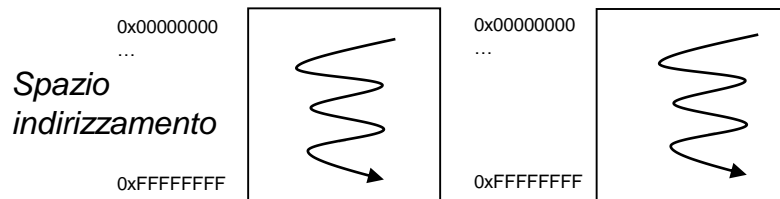
SisOp
2018/19



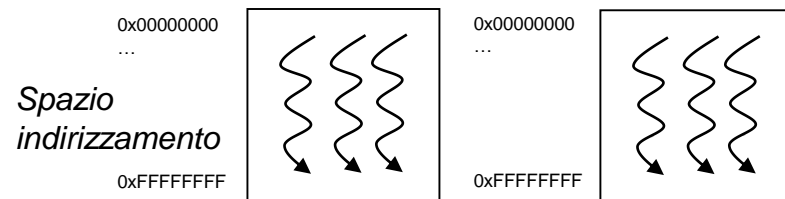
un thread per processo



più thread per processo (i thread condividono la memoria del processo)



Processi multipli -
un thread per processo



Processi multipli -
più thread per processo

Processo UNIX
Thread

⇒ processo pesante (creato con `fork` – identico al padre)
⇒ processo leggero (creato con `pthread_create` – esegue la funzione indicata come argomento)

Strumenti per la comunicazione in ambiente a scambio di messaggi

Si basano su due primitive fondamentali:

send (m), receive (m)

m: messaggio;

```
type messaggio =      record
    origine:           ...;
    destinazione:     ...;
    contenuto:         ...;
end
```



Interazione tra processi in ambiente globale

Tipi di interazione tra processi

SisOp
2018/19



1. Cooperazione:

- Comprende tutte le interazioni *prevedibili e desiderate*, insite cioè nella logica dei programmi
- Prevede *scambio di informazioni*:
 - segnale temporale (senza trasferimento di dati)
 - messaggi (dati)
- In entrambi i casi esiste un *vincolo di precedenza (sincronizzazione)* tra gli eventi di processi diversi. Nel secondo caso è presente anche una comunicazione tra i processi.

2. Competizione:

- La "macchina concorrente" su cui i processi sono eseguiti mette a disposizione un *numero limitato di risorse*.
- Competizione per l'uso di risorse comuni che *non possono essere usate contemporaneamente*.
- Interazione *prevedibile e non desiderata ma necessaria*.

Tipi di interazione tra processi

SisOp
2018/19



- Cooperazione => Sincronizzazione diretta o esplicita
- Competizione => Sincronizzazione indiretta o implicita

3. Interferenza:

Provocata da *errori di programmazione*:

- inserimento nel programma di interazioni tra processi non richieste dalla natura del problema,
 - erronea soluzione a problemi di interazione (cooperazione e competizione) necessari per il corretto funzionamento del programma.
- Interazione *non prevista e non desiderata*.
 - Dipende dalla *velocità relativa* tra i processi:
⇒ "gli effetti possono o meno manifestarsi nel corso dell'esecuzione del programma a seconda delle diverse condizioni di velocità di esecuzione dei processi" (*errori dipendenti dal tempo*).

Il problema delle interferenze

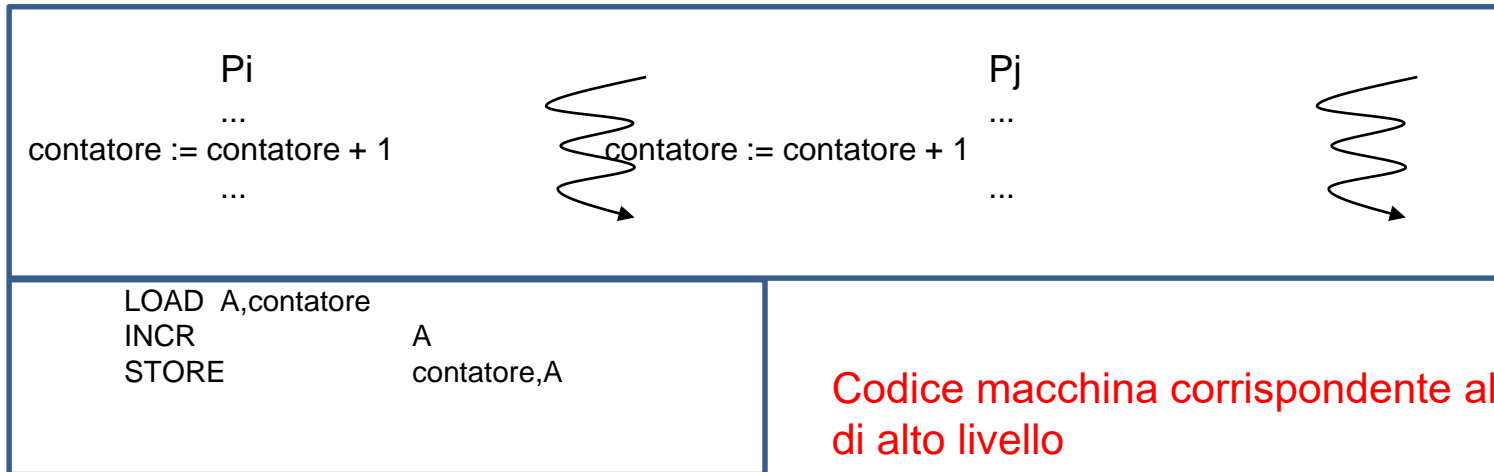
SisOp
2018/19



- Un problema fondamentale della programmazione concorrente è l'*eliminazione delle interferenze*.
- L'eliminazione delle interferenze del primo tipo risulta semplificata se la macchina concorrente fornisce *meccanismi di protezione degli accessi*.
- Per evitare le interferenze del secondo tipo, trattandosi di interazioni previste ma programmate in modo errato, è opportuno adottare tecniche di *multiprogrammazione strutturata*.

Esempio

SisOp
2018/19



P_i e P_j
incrementano
una variabile
comune

Codice macchina corrispondente all'istruzione
di alto livello

t ₀ :	LOAD	A,contatore	(P _i)	
t ₁ :	LOAD	A,contatore	(P _j)	
t ₂ :	INCR	A		(P _j)
t ₃ :	STORE	contatore,A	(P _j)	
t ₄ :	INCR	A		(P _i)
t ₅ :	STORE	contatore,A	(P _i)	

Possibile sequenza di
esecuzione da parte della
CPU

L'incremento del contatore eseguito da P_j non ha lasciato alcuna traccia!

Esempio

Processo P: incrementa una variabile v di 1
Processo Q: stampa il valore di v e lo azzera

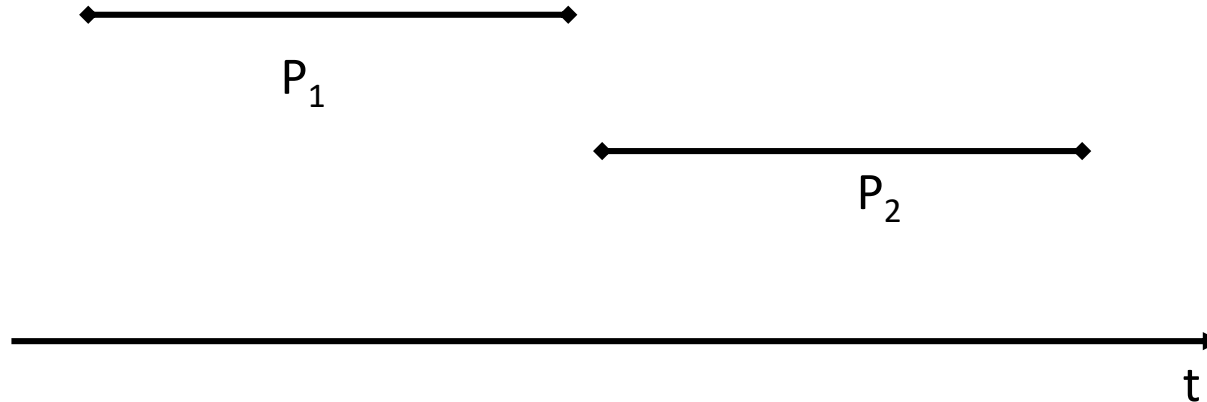
P	Q
...	...
$v := v+1;$	<i>print</i> $v;$
...	$v := 0;$
...	...

Le istruzioni di P e Q possono mescolarsi arbitrariamente e dar luogo a diverse possibili sequenze di esecuzione:

$v := v+1;$ (P)	<i>print</i> $v;$ (Q)	<i>print</i> $v;$ (Q)
<i>print</i> $v;$ (Q)	$v := 0;$ (Q)	$v := v+1;$ (P)
$v := 0;$ (Q)	$v := v+1;$ (P)	$v := 0;$ (Q)
corretta	corretta	sbagliata

Mutua Esclusione

- Si ha un requisito di mutua esclusione quando non più di un processo alla volta può accedere a *variabili comuni*



- Nessun vincolo* è imposto sull'ordine con il quale le operazioni sulle variabili comuni sono eseguite.

Esempio di mutua esclusione

- P1 e P2 utilizzano una stessa telescrivente (periferica tradizionale simile a stampante).
- La telescrivente deve essere assegnata ad un solo processo alla volta per tutta la durata del suo uso.
- Ipotesi di soluzione (errata!) :

Richiesta

...
repeat until libera;
libera := false;

...
Valore iniziale *libera := true;*

Possibile sequenza errata di esecuzione:

t_0 :	<i>repeat until libera</i>	(P1)
t_1 :	<i>repeat until libera</i>	(P2)
t_2 :	<i>libera := false</i>	(P1)
t_3 :	<i>libera := false</i>	(P2)

Rilascio

...
libera := true;
...

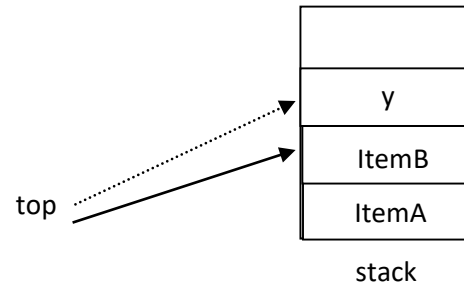
⇒ La telescrivente risulta *assegnata ad entrambi i processi*.

Esempio di mutua esclusione

SisOp
2018/19



- Due processi hanno accesso ad una struttura dati organizzata a *pila* (*stack*) per *depositare* e *prelevare* messaggi:



Inserimento (y)

...
top := top + 1;
stack[top] := y;

...

- Un'*esecuzione contemporanea* delle due procedure può portare ad un uso scorretto della risorsa.

Esempio : P1 inserisce e P2 preleva:

t ₀ :	top := top + 1	(P1)
t ₁ :	x := stack [top]	(P2)
t ₂ :	top := top - 1	(P2)
t ₃ :	stack[top] := y	(P1)

Prelievo (x)

...
x := stack [top];
top := top - 1;

...

Lo stesso problema si ha con riferimento all'esecuzione contemporanea di una qualunque delle due operazioni da parte dei due processi

Sezione critica

- La sequenza di istruzioni con le quali un processo accede e modifica un *insieme di variabili comuni* prende il nome *di sezione critica*.
- Ad un insieme di variabili comuni possono essere associate *una sola* sezione critica (usata da tutti i processi) *o più* sezioni critiche (*classe di sezioni critiche*).
- La *regola di mutua esclusione* stabilisce che:

"sezioni critiche appartenenti alla stessa classe devono escludersi mutuamente nel tempo",
ovvero:
"una sola sezione critica di una classe può essere in esecuzione ad ogni istante".

Soluzione al problema della mutua esclusione

- *Tempificazione dell'esecuzione* dei singoli processi da parte del programmatore (*errori "time dependent"*)
- *Inibizione delle interruzioni* del processore sul quale sono eseguite le sezioni critiche durante l'esecuzione di ciascuna di esse (*soluzione parziale ed inefficiente*)
- *Strumenti di sincronizzazione: semafori e altri*

Semafori

SisOp
2018/19



- Un semaforo è una *variabile intera non negativa* ($s \geq 0$) con valore iniziale $s_0 \geq 0$
- Al semaforo è associata una *lista di attesa* Q_s nella quale sono posti i *descrittori* dei processi che attendono l'autorizzazione a proseguire nell'esecuzione
- Sul semaforo sono ammesse *solo* due operazioni *indivisibili* (*primitive*):

wait (s)	signal (s)
$P(s)$	$V(s)$
- *wait* e *signal* sono realizzate tramite chiamate al S.O. (SVC) ed eseguite in modo *monitor*, cioè la variabile semaforo è *protetta*
- Semafori e primitive di sincronizzazione sono stati introdotti da Dijkstra nel 1963

wait e signal

wait(s) :

```
begin
  if s = 0 then
    <il processo viene sospeso e
    il suo descrittore inserito in  $Q_s$ >;
  else s := s - 1;
end;
```

La *wait* può essere *passante* ($s > 0$) o *bloccante* ($s = 0$), nel qual caso si verifica un *context switch*

signal(s) :

```
begin
  if <esiste un processo in coda> then
    <il suo descrittore viene rimosso da  $Q_s$  e
    il suo stato modificato in pronto>;
  else s := s + 1;
end;
```

La *signal* è
sempre
passante

- L'esecuzione della *signal* (s) non comporta concettualmente alcuna modifica allo stato del processo che l'ha eseguita.
- La scelta del processo sospeso avviene tramite politica FIFO.

Mutua esclusione tramite semaforo

- Ad ogni classe di sezioni critiche viene associata una variabile semaforo s ; prologo ed epilogo vengono realizzati rispettivamente tramite *wait* (s) e *signal* (s).
- A, B sezioni critiche della stessa classe; s semaforo (valore iniziale: $s_0 = 1$):

processo P1

...

wait (s);

<sezione critica A>;

signal (s);

...

processo P2

...

wait (s);

<sezione critica B>;

signal (s);

...

- La natura primitiva di *wait* e *signal* assicura la proprietà di mutua esclusione.
- La soluzione è corretta per qualunque numero di processi e per velocità relative arbitrarie.
- Sono risolti i problemi di attesa attiva e attesa indefinita (gestione opportuna della coda dei processi bloccati, es. FIFO). Un processo non può riappropriarsi della sezione critica che ha appena liberato se ci sono altre richieste pendenti (nella *signal* è rimasto $s = 0$).

Indivisibilità di wait e signal

SisOp
2018/19



- Occorre garantire che l'azione di analisi e modifica del semaforo non sia separata dalla azione di sospensione.
- Esempio con wait e signal non atomiche:

```
t0:           // semaforo s con valore corrente s = 0
t0:   if s = 0           (P1 – inizio wait)
t1:   s := s + 1         (P2 - signal)
t2:   sospensione      (P1 – fine wait)
```

Si ha come conseguenza un processo sospeso (P1) su un semaforo che vale 1.

- Si può ottenere indivisibilità *inibendo le interruzioni durante l'esecuzione di wait e signal*.
- Tale soluzione vale *solo se tutte le wait e signal relative allo stesso semaforo sono eseguite sullo stesso processore*.
- Nel caso di sistema multiprocessore occorre considerare *wait e signal come sezioni critiche brevi e proteggerle mediante un meccanismo di più basso livello denominato lock*.

Lock e Unlock

- x indicatore associato alla classe di sezioni critiche (inizializzato a 0):
x = 0 nessuna sezione critica in esecuzione
x = 1 una sezione critica in esecuzione

```
lock (x):    begin
               repeat until x = 0;           // test del valore di x
               x := 1;                       // modifica del valore di x
            end;
```

```
unlock (x): begin
               x := 0; // indivisibile
            end;
```

- lock e unlock devono essere *indivisibili*
- Nell'ipotesi che l'*hardware* garantisca la mutua esclusione solo a livello di singola lettura o scrittura di una cella di memoria, *solo unlock (x) è indivisibile*.
- Per rendere indivisibile anche lock(x) occorrono speciali istruzioni dei processori come TSL (Test-and-Set-Lock) e CMPXCHG (Intel)

Lock con TSL

L'istruzione macchina `TSL RX, LOCK` legge il valore della variabile `LOCKVAR` nel registro `RX` e atomicamente scrive 1 in `LOCKVAR` (un unico ciclo non interrompibile di lettura/scrittura sul bus)

```
lock:                # lock(x) dove x è rappresentata da LOCKVAR
tsl R1, LOCKVAR      # è un unico ciclo di lettura e scrittura
cmp R1, 0            # R1 vale 0 ?
jne lock              # se R1 non è uguale a 0, vale 1 il che significa che
                      # la sezione critica era già occupata da un altro processo:
                      # occorre riprovare (si ritorna a lock)

                      # se R1 ora vale 0, significa che il processo corrente
                      # è riuscito a prendere possesso della sezione critica
ret                  # ritorno
-----

unlock:              # unlock(x) dove x è rappresentata da LOCKVAR
mov LOCKVAR, 0       # è un'istruzione atomica
ret
```

Indivisibilità di wait e signal

SisOp
2018/19



Nel caso generale in cui *wait* e *signal* relative allo stesso semaforo possono essere eseguite su processori diversi si ha:

```
wait (sem):    begin  
                <disabilitazione interruzioni>;  
                lock (x);  
                <codice della wait>;  
                unlock (x);  
                <abilitazione interruzioni>;  
            end;
```

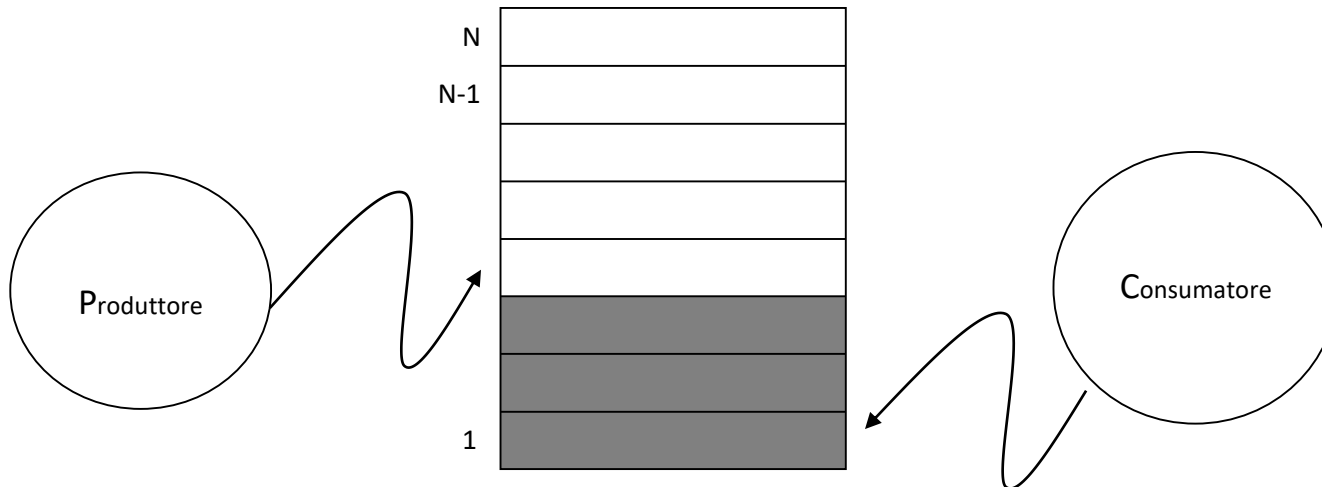
```
signal (sem): begin  
                <disabilitazione interruzioni>;  
                lock (x);  
                <codice della signal>;  
                unlock (x);  
                <abilitazione interruzioni>;  
            end;
```

Livelli di sezioni critiche

- I Livello:
sezioni critiche: S1, S2 (codice a livello utente)
mutua esclusione tramite *wait* e *signal*
- II Livello:
sezioni critiche: *wait(s)* e *signal(s)* (codice a livello S.O.)
mutua esclusione tramite *lock(x)* e *unlock(x)*
- III Livello:
sezioni critiche: *lock(x)*, *unlock(x)* (codice a livello S.O.)
mutua esclusione tramite hardware (istruzione TSL)

Esempio: Produttore-Consumatore

- Buffer in grado di contenere N messaggi, a cui accedono il processo P per l'inserimento di un messaggio ed il processo C per il prelievo di un messaggio:



- Il produttore *non può inserire un messaggio nel buffer se questo è pieno.*
- Il consumatore *non può prelevare un messaggio dal buffer se questo è vuoto.*
- Indicando con:
d = numero dei messaggi depositati
N = dimensione del buffer

e = numero dei messaggi estratti

$$0 \leq d - e \leq N$$

Produttore-Consumatore

- La soluzione richiede due semafori:

"messaggio disponibile"
"spazio disponibile"

mess-disp valore iniziale 0
spazio-disp valore iniziale N

- | <u>Produttore (P)</u> | <u>Consumatore (C)</u> |
|---------------------------|-----------------------------|
| begin | begin |
| repeat | repeat |
| <produzione messaggio> | wait (mess-disp) |
| wait (spazio-disp) | <prelievo messaggio> |
| <deposito messaggio> | signal (spazio-disp) |
| signal (mess-disp) | <consumazione messaggio> |
| forever | forever |
| end | end |

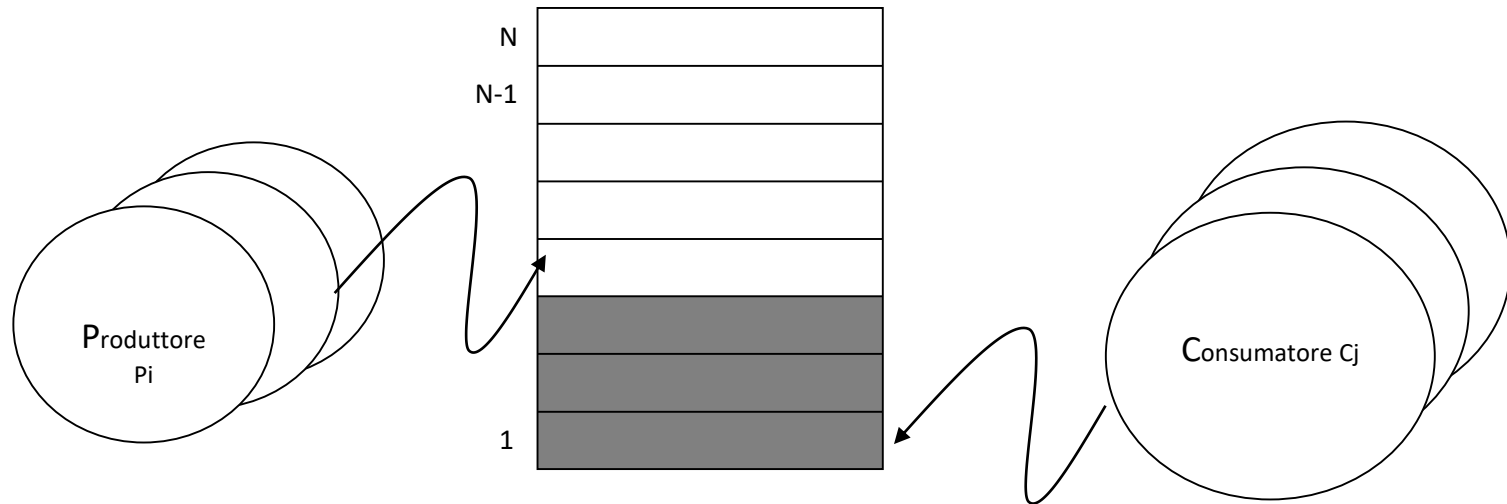
- E' una soluzione *simmetrica*, non privilegia nessun processo.
- P e C possono operare in parallelo sul buffer su messaggi diversi: P e C non possono operare sul medesimo messaggio, indipendentemente dalla sua lunghezza. (P e C tentano di accedere allo stesso messaggio solo nelle condizioni limite di buffer pieno e buffer vuoto; in tali condizioni uno dei due processi è bloccato dalla wait).

Produttori-Consumatori

SisOp
2018/19



- Se in generale L produttori P_i ed M consumatori C_j accedono al buffer limitato:



- In aggiunta ai vincoli del problema che prevedeva un solo produttore e un solo consumatore, ora i produttori *non possono inserire contemporaneamente messaggi nel buffer* per evitare interferenze. Per lo stesso motivo, *due o più consumatori non possono accedere simultaneamente al buffer*.
- «deposito» è una sezione critica per i produttori; «prelievo» è una sezione critica per i consumatori.

Produttori-Consumatori

SisOp
2018/19



- La soluzione ora richiede l'impiego più semafori:

mess-disp	valore iniziale 0	// per regolare i consumatori
spazio-disp	valore iniziale N	// per regolare i produttori
mutex1, mutex2	valore iniziale 1	// per la mutua esclusione

- | <u>Produttore Pi</u> | <u>Consumatore Cj</u> |
|---------------------------|-----------------------------|
| begin | begin |
| repeat | repeat |
| <produzione messaggio> | wait (mess-disp) |
| wait (spazio-disp) | wait (mutex2) |
| wait (mutex1) | <prelievo messaggio> |
| <deposito messaggio> | signal (mutex2) |
| signal (mutex1) | signal (spazio-disp) |
| signal (mess-disp) | <consumazione messaggio> |
| forever | forever |
| end | end |

- Con due semafori mutex, un Pi ed un Cj possono operare in parallelo sul buffer su messaggi diversi e si ottiene il massimo parallelismo
- Con un solo mutex la soluzione resterebbe corretta ma si avrebbe una serializzazione dei processi sul buffer

Regolazione dell'esecuzione di processi e thread

- Problema:
 - n processi (o thread) P_1, P_2, \dots, P_n devono essere attivati ad intervalli di tempo prefissati da un processo gestore P_0
 - l'esecuzione di P_i non può iniziare prima che sia giunto il segnale da P_0
 - ad ogni segnale inviato da P_0 deve corrispondere un'attivazione di P_i

- Soluzione:

- Definiamo n semafori s_i con valore iniziale $s_{i0}=0$

P_i		P_0
...		...
<i>repeat</i>		<i>repeat</i>
	wait (s_i);	...
	<do something>	signal (s_i);
<i>forever</i>		...
		<i>forever</i>

Programmazione concorrente e semafori

SisOp
2018/19



- I semafori sono uno strumento *potente e generale* per la soluzione dei problemi di programmazione concorrente in ambiente globale, sia per competizione che per cooperazione di thread e processi
- I semafori sono uno strumento potente ma di *basso livello*: è difficile risolvere problemi complessi utilizzando semafori; è facile commettere errori nel loro uso e data la generalità del loro uso l'ambiente di sviluppo può fornire scarso supporto per rilevarli
- Si utilizzano pertanto *meccanismi di più alto livello*:
 - costrutti come *Monitor, Regioni Critiche e oggetti sincronizzati* supportati a livello di linguaggio (es. Concurrent Pascal, Mesa, Java)
 - librerie per la programmazione multithread (es. API POSIX Pthread)
 - *separando* la funzione di *mutua esclusione* (variabili di lock, mutex, parole chiave synchronized o shared) da quelle di *segnalazione di eventi* (variabili condizione, esplicite o anonime)
 - adottando *pattern progettuali* di provata affidabilità ed efficacia



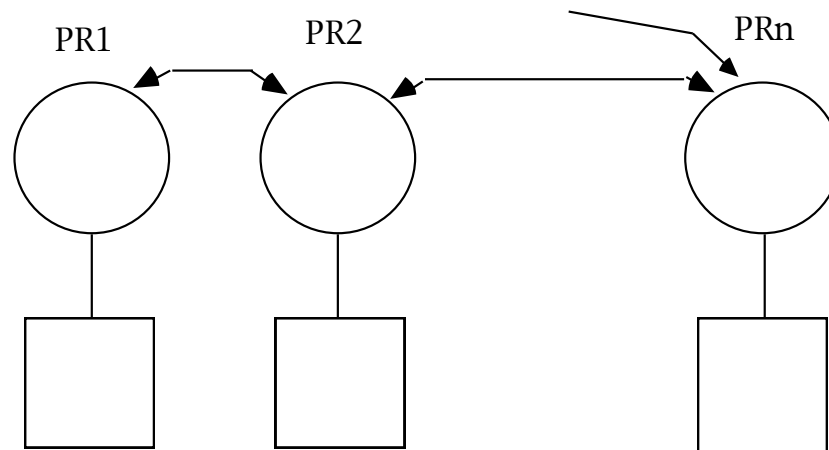
Modello ad ambiente locale

Modello a scambio di messaggi

- Il sistema è concepito come un insieme di *processi* ciascuno operante in un *ambiente locale* non accessibile direttamente a nessun altro processo.
- Ogni forma di interazione tra processi (comunicazione, sincronizzazione), avviene tramite *scambio di messaggi*.
- Non esiste il concetto di *risorsa accessibile direttamente ai processi*. Sono possibili due casi:
 - alla risorsa è associato un *processo servitore*
 - la risorsa viene trasferita da un processo ad un altro *sotto forma di messaggi*

Modello a scambio di messaggi

- Il modello a scambio di messaggi rappresenta la naturale astrazione di un sistema privo di memoria comune (sistema distribuito), in cui a ciascun processore è associata una memoria privata.



- Il modello a scambio di messaggi può essere realizzato anche in presenza di memoria comune, che viene utilizzata per realizzare canali di comunicazione.

Primitive per lo scambio di messaggi

SisOp
2018/19



- Un messaggio si può considerare costituito da: origine, destinazione, contenuto

```
type messaggio =      record
                        origine:      ...;
                        destinazione:  ...;
                        contenuto:     ...;
                        end
```

- Nel caso più semplice si può supporre che:
 - ad ogni processo sia associata una coda per i messaggi in arrivo;
 - le primitive di comunicazione usate dai processi sono:

send (m) receive (m) ove: var m: message
 - la primitiva *send (m)* inserisce il messaggio *m* nella coda del destinatario
 - la primitiva *receive (m)* preleva un messaggio dalla coda o sospende il processo se la coda è vuota.

Scambio di messaggi

Con lo scambio di messaggi viene realizzata:

- la **comunicazione**:
un processo, attraverso la ricezione di un messaggio, *ottiene valori da un processo mittente*;
- la **sincronizzazione**:
un messaggio può essere ricevuto solo dopo che è stato trasmesso; tale relazione di causa-effetto *vincola* l'ordine in cui i due eventi possono avvenire.
- La *mutua esclusione* non è più un problema, perché nel modello ad ambiente locale tutte le risorse sono *private*.

Buffer di comunicazione con monitor (modello ad ambiente globale)

Realizzazione attraverso *monitor* (oggetti/tipo di dato astratto con metodi sincronizzati)

/ possibile uso per interazione Produttori-Consumatori */*

```
type IO_buffer = monitor
    var          buffer: block; inuse: boolean;
                free, loaded: condition;

procedure entry deliver (in: block);
    begin
        if inuse then free.wait;
        buffer := in;
        inuse := true;
        loaded.signal
    end

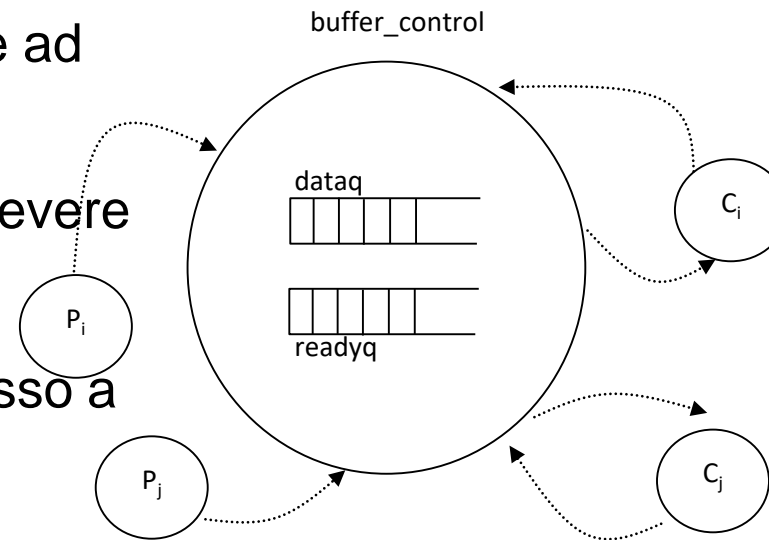
procedure entry retrieve (out: block);
    begin
        if not inuse then loaded.wait;
        out := buffer;
        inuse := false;
        free.signal
    end

begin    inuse := false    end
end type;
```


Buffer di comunicazione nel modello a scambio di messaggi

E' necessario un processo gestore (buffer_control) della risorsa buffer che serve i processo produttori P_i e i processi consumatori C_j

- P_i manda un messaggio al processo buffer_control che a sua volta lo deve inviare ad uno dei processi C_j .
- Ciascun C_j deve inviare un messaggio a buffer_control per indicare che è pronto a ricevere il messaggio.
- Esistono quindi due tipi di messaggi in ingresso a buffer_control:
 - "data" inviato da P_i
 - "ready" inviato da C_j
- Devono esistere *due code entro buffer_control* per memorizzare i due tipi di messaggi:
 - "dataq" e "readyq"



Buffer di comunicazione nel modello a scambio di messaggi

```
process buffer_control;  
    var inputm, outputm: message;  
        dataq, readyq: queue of message;  
  
repeat forever  
    receive (inputm);  
    case inputm.contents.type of  
        "data":    if <ci sono messaggi nella coda readyq>  
                    thenbegin  
                        <prepara outputm>;  
                        send (outputm)  
                    end;  
        else <inserisci inputm nella coda dataq>;  
        "ready":   if <ci sono messaggi nella coda dataq>  
                    thenbegin  
                        <prepara outputm>;  
                        send (outputm)  
                    end;  
        else <inserisci inputm nella coda readyq>;  
    end case  
end
```

- Ipotesi di produttori e consumatori indistinguibili: un messaggio prodotto può essere consegnato ad un qualsiasi consumatore.

Buffer di comunicazione nel modello a scambio di messaggi

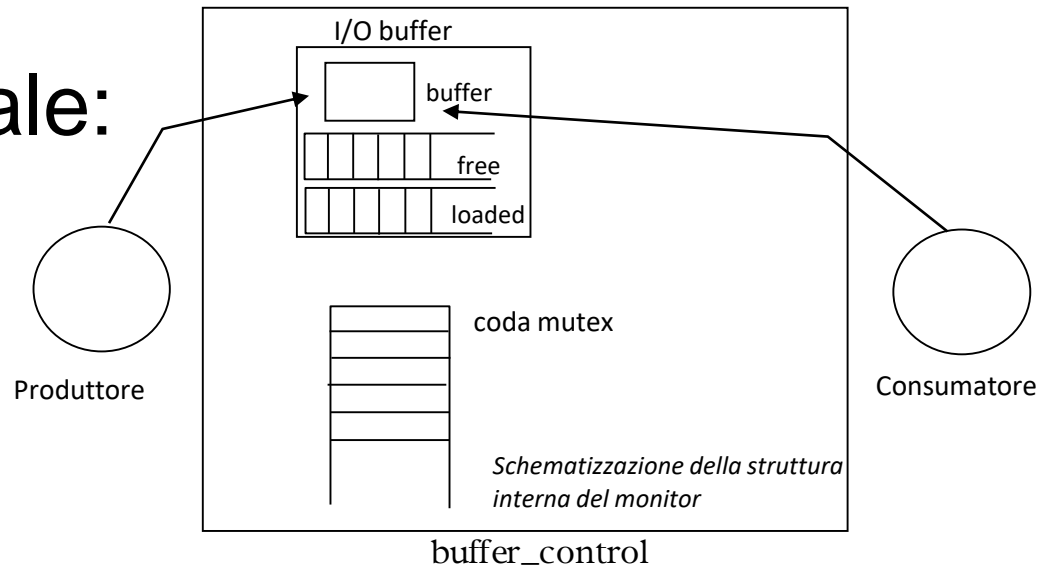
```
process prod_i;
    var    mess:      message;

    repeat forever
        ...
        <produzione informazione>;
        mess.contents.type := "data";           /* costruzione mess. */
        mess.contents.info := <informazione>;
        mess.destination := buffer_control;
        send (mess);                             /* <<----- */
        ...
    end ;

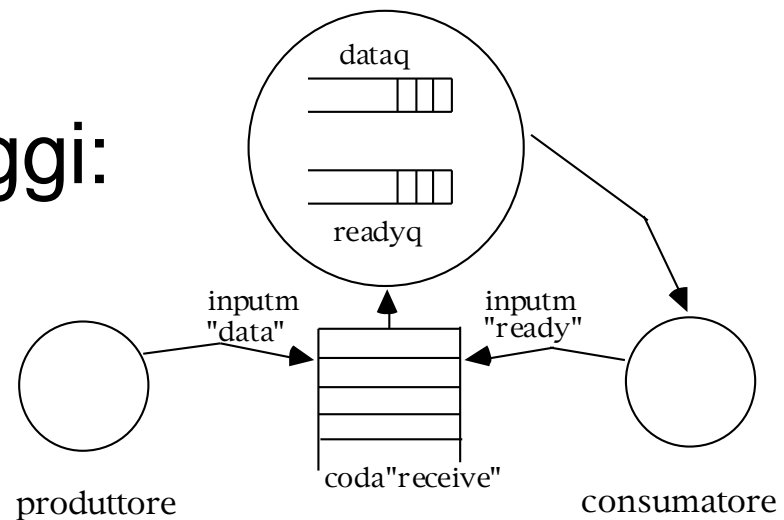
process cons_j;
    var    mess1, mess2:      message;
    repeat forever
        ...
        mess1.contents.type := "ready";
        mess1.destination := buffer_control;
        send (mess1);                             /* <<----- */
        receive (mess2);                          /* <<----- */
        <estrazione e uso informazione>;
        ...
    end
```

Confronto tra le due strutture

- ambiente globale:



- scambio di messaggi:



Osservazioni

- *Sincronizzazione tra processi attraverso scambio di messaggi*
 - I processi produttori non sono regolati. Può nascere l'esigenza di un buffer illimitato (coda dataq).
 - Nell'esempio precedente prima di inviare un messaggio di tipo "data" il processo produttore può inviare un messaggio "ready_to_send" a buffer_control, che risponderà "ok_to_send" se c'è posto nella coda.
- *Protezione*
 - Chi assicura che i produttori rispettino il protocollo definito? Buffer_control potrebbe ritornare con "ok_to_send" una chiave di accesso per poi eseguire una *autenticazione*.
- *Riduzione del parallelismo*
 - Non è possibile l'accesso contemporaneo ad una risorsa da parte di più processi. Gli *accessi sono sequenzializzati dal processo servitore*. Talvolta si può ovviare al problema suddividendo la risorsa in più risorse gestite da processi diversi.

Costrutti linguistici per il modello a scambio di messaggi

Classificazione

- a) **designazione** dei processi sorgente e destinatario di ogni comunicazione
 - designazione diretta o esplicita
 - simmetrica
 - asimmetrica
 - designazione indiretta o globale
 - mailbox
 - porte

- b) *tipo di* **sincronizzazione** tra i processi comunicanti
 - sincrona
 - asincrona

Caratteristiche *ortogonali*: le soluzioni proposte per a) e b) sono tra loro indipendenti

Primitive con designazione esplicita

send <expression_list> **to** <destination_designator>

receive <variable_list> **from** <source_designator>

- L'esecuzione della *send* determina il contenuto del messaggio mediante la valutazione delle espressioni in <expression_list>.
- <destination_designator> dà al programmatore il controllo su dove inviare il messaggio.
- L'esecuzione della *receive* determina l'assegnamento dei valori contenuti nel messaggio alle variabili in <variable_list>, e la successiva distruzione del messaggio.
- <source_designator> dà al programmatore il controllo sull'origine dei messaggi.

Designazione esplicita

SisOp
2018/19



- La coppia (`<destination_designator>`, `<source_designator>`) definisce un *canale di comunicazione*.
- Schema *simmetrico*
I processi si nominano *esplicitamente* (*direct naming*) l'un l'altro:

send message to P2
receive message from P1

- P1 invia un messaggio che può essere ricevuto solo da P2
- P2 riceve un messaggio che può provenire solo da P1
- Semplice da realizzare e da utilizzare: un processo può controllare in maniera selettiva gli intervalli di tempo in cui riceve messaggi dagli altri processi.
- E' usato nei modelli del tipo *pipeline*: collezione di processi concorrenti in cui l'output di un processo costituisce l'input di un altro. Il sistema è concepito in termini di flusso di informazione.

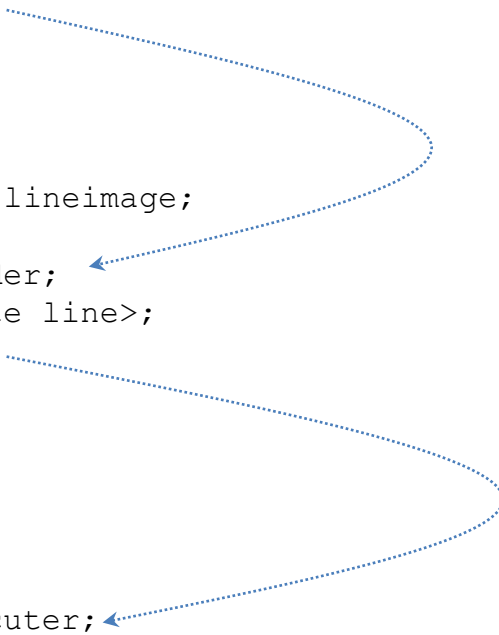
Direct naming - esempio

- Elaborazione batch mediante scambio di messaggi. Esempio di sistema a paradigma *pipeline*.

```
process reader;
  var card: cardimage;
  loop
    <read card from cardreader>;
    send card to executer
  end
end;

process executer;
  var card: cardimage; line: lineimage;
  loop
    receive card from reader;
    <process card and generate line>;
    send line to printer
  end
end;

process printer;
  var line: lineimage;
  loop
    receive line from executer;
    <print line on line printer>
  end
end;
```



Direct naming: schema asimmetrico

SisOp
2018/19



- Il mittente nomina *esplicitamente* il destinatario, mentre questi, al contrario, non esprime il nome del processo con cui desidera comunicare. (Vedi esempio "buffer_control".)
- Notazione:
 send <message> to P2
 process_id := *receive* <message>
- oppure:
 send (P2, message)
 receive (process_id, message)
- In process_id il ricevente dispone dell'*identità del mittente* per l'eventuale messaggio di risposta.
- La designazione asimmetrica facilita la organizzazione della interazione tra processi secondo il *paradigma Cliente-Servitore*, in cui un processo gestore di una risorsa (servitore) riceve richieste da più processi cliente.

Modello Cliente-Servitore

- Corrisponde all'uso di un *processo come gestore di una risorsa*.

Pi (cliente)

...

send <richiesta>;

receive <risultato>;

...

Pj (servitore)

...

receive <richiesta>;

<esecuzione>;

send <risultato>;

- Schema *da-molti-a-uno* :

I processi cliente *specificano il destinatario* delle loro richieste. Il processo servitore è pronto a ricevere messaggi *da qualunque cliente*.

- Schema *da-uno-a-molti* o *da-molti-a-molti* :

I processi cliente inviano richieste non ad *un particolare servitore*, ma ad uno qualunque scelto tra un insieme di *servitori equivalenti*.

E' di difficile realizzazione con designazione diretta. Richiede di passare ad una *designazione indiretta o globale (mailbox)*.

Modello client-server e naming

- Il direct naming è in generale poco adatto al modello client-server.
- In presenza di *più clienti* la `receive` di un servitore dovrebbe consentire la ricezione di un messaggio da un qualsiasi cliente. Nel caso di designazione esplicita simmetrica sarebbe necessaria almeno una `receive` per ogni cliente.
- In presenza di *più servitori* equivalenti la `send` di un cliente dovrebbe produrre un messaggio che possa essere ricevuto da un qualsiasi servitore.
- Occorre uno schema più sofisticato per la definizione dei canali di comunicazione: *designazione globale o indiretta*. Fa uso di nomi globali detti **mailbox**.

Designazione globale

SisOp
2018/19



- Una mailbox può apparire come `<destination_designator>` o come `<source_designator>` nelle istruzioni di *send* e *receive* di qualunque processo.
- I messaggi inviati ad una specifica mailbox possono essere ricevuti da *qualsiasi processo* che effettui una *receive* designando tale mailbox.
- Notazione:

```
send    message to  A_mbox  
process_id := receive message from  A_mbox
```
- oppure:

```
send    (A-mbox, message)  
receive (A-mbox, message)
```
- I processi possono *selezionare i tipi di messaggio* che desiderano ricevere effettuando *receive* sulle mailbox opportune.

Uso delle mailbox

SisOp
2018/19



- La mailbox consente in modo immediato la programmazione delle interazioni cliente-servitore anche nel caso *da-molti-a-molti*. I clienti eseguono una *send* sulla mailbox associata al servizio, i servitori una *receive*.
- La realizzazione delle mailbox in ambiente distribuito presenta problemi di *natura realizzativa*. Il supporto a tempo di esecuzione del linguaggio deve garantire che:
 - un messaggio di richiesta indirizzato ad una mailbox è inviato a *tutti i processi* che possono eseguire una *receive* su di essa;
 - non appena il messaggio è ricevuto da un processo, esso deve *diventare indisponibile per tutti gli altri servitori*.

Porte

SisOp
2018/19



- Sono mailbox il cui nome può comparire solamente *in un processo* come *<source-designator>* in uno statement di *receive*.
- Sono di *realizzazione più semplice* delle mailbox:
⇒ tutte le *receive* che indicano una porta compaiono in un solo processo.
- Forniscono una soluzione al problema "più clienti - un solo servitore" (ma non a quello "più clienti - più servitori").
- Un processo può *selezionare* i messaggi che desidera ricevere attraverso l'uso di porte distinte.
- Se un processo effettua *receive su una sola porta*, lo schema di designazione è logicamente equivalente ad un direct naming asimmetrico, a meno di aspetti di modularità e flessibilità.

Naming

- direct naming simmetrico \Rightarrow comunicazione one to one
- direct naming asimmetrico e port naming \Rightarrow comunicazione many to one
- global naming \Rightarrow comunicazione many to many
- Il global naming è il caso più generale. Gli altri schemi limitano i *tipi di interazione direttamente programmabili* ma sono più semplici da realizzare da parte del SO.
- La designazione dei canali può avvenire *staticamente, a tempo di compilazione, o dinamicamente*, a tempo di esecuzione.
- Naming statico:
 - impedisce ad un programma di comunicare tramite canali non noti a tempo di compilazione; ne limita le capacità di sopravvivenza in un *ambiente dinamico*.
 - il *potenziale* accesso di un programma ad un canale deve essere assicurato fin dall'inizio, e cioè *permanentemente*.
- Naming dinamico:
 - uno schema statico di base di designazione dei canali viene arricchito mediante variabili per la designazione di sorgente o destinazione.

Sincronizzazione

- Send asincrona
- Send sincrona ("rendez-vous" semplice)
- Send di tipo "chiamata a procedura remota" ("rendez-vous" esteso)
- Receive sincrona
- Receive asincrona e Interrogazione dello stato di un canale

Send asincrona

- Il processo mittente *continua la sua esecuzione* immediatamente dopo che il messaggio è stato inviato.
- Il messaggio ricevuto contiene informazioni che non possono essere associate *allo stato attuale del mittente*. Ciò comporta notevoli difficoltà nella verifica dei programmi.
- L'interazione viene definita come *scambio di messaggi asincrono*.
- Per la memorizzazione dei messaggi il supporto del linguaggio deve mettere a disposizione *una coda in ingresso ad ogni processo* nel caso di direct naming, ed una coda in ingresso ad *ogni porta o mailbox* nel caso di global naming.

Send asincrona

- In analogia con il meccanismo semaforico, la *send asincrona* è caratterizzata da:
 - *flessibilità di uso* (i costrutti di più alto livello possono essere realizzati mediante send asincrone)
 - *carenza espressiva*
- Richiede, a livello realizzativo, un *buffer di capacità illimitata*. Si può ovviare *modificandone la semantica*:
 - a) un processo mittente *si blocca* qualora la coda dei messaggi sia piena;
 - b) la primitiva send, in caso di coda piena, solleva un'*eccezione* che viene notificata al processo mittente.

Send sincrona

- "Rendez-vous" semplice:
- Il processo mittente *si blocca* in attesa che il messaggio sia stato ricevuto.
- Un messaggio ricevuto contiene informazioni corrispondenti allo *stato attuale* del processo mittente. Ciò semplifica la scrittura e la verifica dei programmi.
- L'invio di un messaggio costituisce *un punto di sincronizzazione* sia per il mittente che per il destinatario: il trasferimento delle informazioni avviene quando entrambi i processi sono pronti a comunicare (rendez-vous).
- L'interazione viene definita come *scambio di messaggi sincrono*.
- Ai processi sono associati canali *privi di memoria*, uno per ogni tipo di messaggio che il processo può ricevere.

Send di tipo "chiamata di procedura remota"

- "Rendez-vous" esteso: il processo mittente *rimane in attesa* fino a che il ricevente non ha *terminato di svolgere l'azione richiesta*.
- La *send con rendez-vous esteso* o *remote procedure call (RPC)* ha una *analogia semantica* (e spesso *sintattica*) con la chiamata di procedura:
 - un processo cliente "chiama" una procedura eseguita da un processo servitore su una macchina potenzialmente remota;
 - il nome della procedura remota identifica *un processo in caso di direct naming, oppure un servizio* in caso di port o mailbox naming.
- I programmi risultano più facilmente verificabili grazie alla localizzazione dei vincoli di sincronizzazione.
- La *send di tipo RPC* è *orientata al modello cliente-servitore*.
- L'interazione tra i processi presenta una *riduzione di parallelismo*, spesso solo apparente: in un modello client-server normalmente i clienti comunque si bloccano in attesa del completamento del servizio, effettuando una *receive* subito dopo la *send*.

Receive

SisOp
2018/19



- Normalmente è *bloccante* se non vi sono messaggi sul canale. Costituisce *un punto di sincronizzazione* per il processo ricevente.
- *Problema*: un processo desidera ricevere solo alcuni messaggi ritardando l'elaborazione di altri. (Esempio: processi gestori di risorse che intendono effettuare ricezione di messaggi compatibili con lo stato delle risorse).
- *Soluzione*: *specificare più canali di ingresso* per ogni processo, ciascuno dedicato a messaggi di tipo diverso.
 - Deve essere possibile specificare su quali canali attendere, *sulla base dello stato interno della risorsa*.
- Si ricorre ad una primitiva che *verifica* lo stato del canale e restituisce un messaggio se esso è presente, ovvero *un'indicazione di canale vuoto (receive non bloccante)*.
 - Ciò consente ad un processo di selezionare l'insieme di canali da cui prelevare un messaggio.
- *Inconveniente*: per l'attesa di messaggi da *specifici canali* occorre fare uso di *cicli di attesa attiva*.

Chiamata di procedura remota (RPC)

- Consente di esprimere a più alto livello e in maniera più sintetica le interazioni di tipo client-server.
- Specifica del *lato cliente*:

```
call service (<parametri di ingresso>; <parametri di uscita>);
```

- service è il nome di un canale:
 - Se la designazione è diretta, service indica il processo servitore.
 - Se la designazione è indiretta (porte o mailbox), service indica il tipo di servizio richiesto.
- La *call* può essere tradotta in una *send* seguita immediatamente da una *receive*. Il cliente quindi non si può "dimenticare" di attendere la risposta.
- Specifica del *lato servitore*:
 1. come *procedura dichiarata separatamente*,
 2. come *statement collocato in un punto qualunque di un processo*.

RPC: specifica del lato servitore

- Specifica del *lato servitore* come procedura :

```
remote procedure service
(in  <parametri di ingresso>; out  <parametri di uscita>)
<body>
end
```

- La procedura remota viene dichiarata come una procedura in un linguaggio sequenziale e realizzata come un *processo servitore* che attende la ricezione di un messaggio, esegue il corpo e trasmette un messaggio di risposta.
- Può essere realizzata come *un singolo processo* che esegue le richieste una alla volta in modo sequenziale, oppure con la creazione di *un nuovo processo per ogni chiamata*. Le varie istanze sono eseguite concorrentemente, e potranno eventualmente doversi sincronizzare tra loro.

RPC: specifica del lato servitore

- Specifica del *lato servitore come statement* :
- La procedura remota è uno *statement*, e come tale può essere collocato in un punto qualunque del processo servitore.

```
accept service  
(in <parametri di ingresso>; out <parametri di uscita>) --> body
```

- L'esecuzione della *accept* sospende il servitore fino all'arrivo di un messaggio corrispondente alla *call* del servizio.
L'esecuzione del corpo può fare uso dei valori dei parametri e di tutte le variabili accessibili dallo scope dello *statement*.
Al termine viene trasmesso il messaggio di risposta al processo chiamante, dopo di che il processo servitore *continua la propria esecuzione*.
- Ad ogni servizio è associata una coda distinta, generalmente di tipo FIFO.

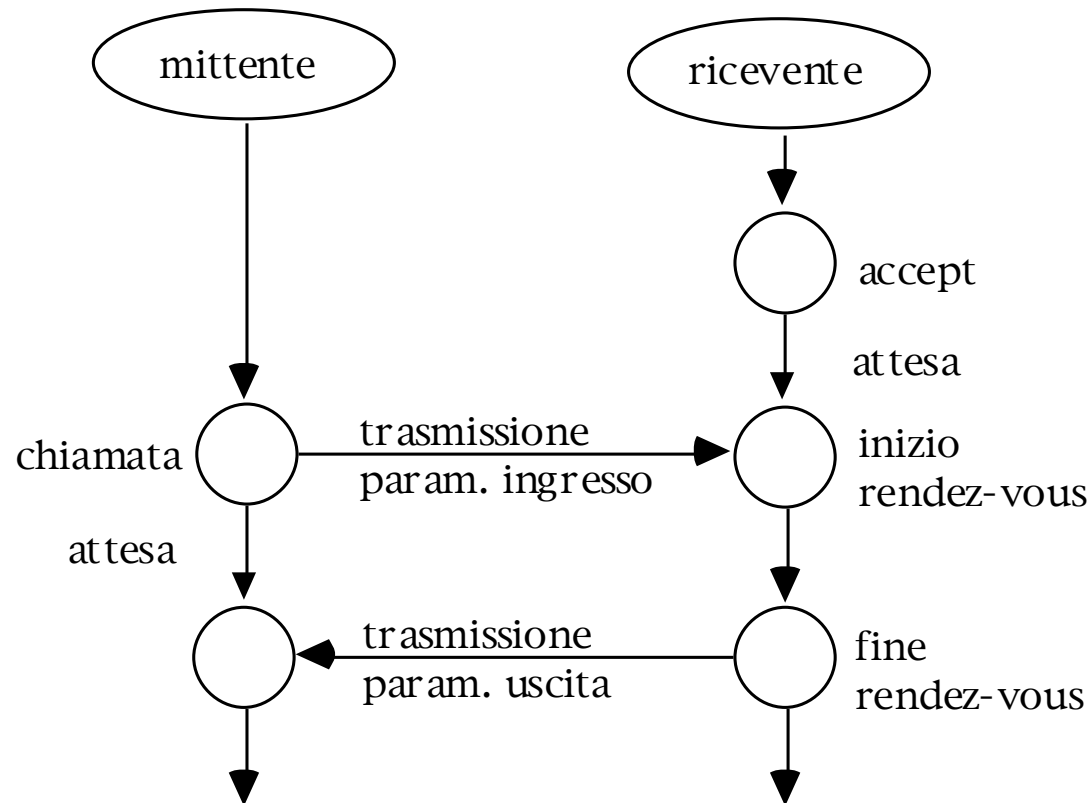
Uso della *accept*

SisOp
2018/19



- Quando il lato servitore è specificato con una *accept* (ad es. linguaggio ADA), la RPC viene chiamata *extended rendez-vous*: cliente e servitore si "incontrano" per la durata della esecuzione del corpo della *accept* per poi proseguire separatamente.
- Vantaggi:
 - il servitore può fornire *più tipi di servizi* (*accept* diverse)
 - il servitore può decidere *quando* servire *le call* dei clienti
 - il servitore può selezionare *quali tipi di call* servire
 - le istruzioni di *accept* possono essere alternate o innestate
 - vi possono essere più *accept* di chiamate allo stesso servizio con diverso *body* (ad esempio per l'inizializzazione).
- La *accept* viene spesso combinata con comunicazioni selettive per consentire ad un servitore di attendere e selezionare una tra diverse richieste di servizio.
- La possibilità di *accept* diverse per lo stesso servizio fa sì che ad una richiesta possano corrispondere *azioni* diverse in funzione dello stato del processo servitore. Ciò introduce una netta distinzione rispetto al concetto di procedura.

Schema di comunicazione realizzato dalla RPC



Uso della RPC

SisOp
2018/19



- Lo schema di comunicazione realizzato dal meccanismo della chiamata a procedura remota è di tipo asimmetrico e da molti ad uno.
- L'accoppiamento tra una *chiamata priva di parametri* ed una *accept priva di corpo* rappresenta la trasmissione ed il relativo riconoscimento di un *segnale di sincronizzazione*.
- Una *chiamata con soli parametri di ingresso* ed una *accept priva di corpo* definiscono invece un *rendez-vous stretto*.
- L'istruzione *accept* consente di considerare un processo come un modulo che *incapsula* un insieme di *funzioni chiamabili dall'esterno ed eseguibili una alla volta*, con analogie con il monitor in ambiente globale.
- Problemi della RPC:
 - interazioni non client-server,
 - perdita di messaggi nelle architetture distribuite.

II deadlock

SisOp
2018/19



Deadlock (blocco critico)

SisOp
2018/19



- Più processi possono entrare in competizione per ottenere l'uso di risorse. Se la risorsa richiesta *non è disponibile* il processo viene posto in condizione *di attesa*.
- Se un processo in attesa non cambia più il suo stato, cioè se le risorse richieste sono trattenute da altri processi essi pure in attesa, si ha una *situazione di deadlock* (blocco critico).
- Per blocco critico si intende una situazione nella quale *uno o più processi* rimangono *indefinitamente bloccati* a causa della impossibilità del verificarsi delle condizioni necessarie per il loro proseguimento.
- Nel caso di un solo processo bloccato si parla anche di *blocco individuale*.
- Si possono avere situazioni di deadlock sia nel caso di *interazione indiretta* tra processi (risorse riusabili) che nel caso di *interazione diretta* (risorse consumabili).

Risorse riusabile e risorse consumabili

Risorse riusabili: dopo il loro uso da parte di un processo possono essere usate da altri processi
→ *serially reusable resources*

Proprietà:

- devono essere usate in modo esclusivo,
- (in genere) non possono essere sottratte al processo durante l'uso.

Esempi:

- risorse fisiche (stampanti, telescriventi, etc.)
- risorse logiche (files, tabelle, etc.).

Sono divise in *tipi*, ciascuno dei quali costituito da una o più unità *equivalenti* (usabili in maniera indifferenziata dai processi).

Risorse consumabili:

- Sono *segnali* o *messaggi* scambiati tra processi (ad esempio i segnali e i messaggi delle pipe in UNIX)
- *Cessano di esistere* non appena acquisite da un processo.
- Sono potenzialmente in *numero infinito*.

Esempio di deadlock

- Due processi A e B necessitano entrambi delle risorse R1 (disco) e R2 (stampante)
 1. A richiede e ottiene R1
 2. B richiede e ottiene R2
 3. A richiede R2 e viene bloccato
 4. B richiede R1 e viene bloccato

Esempio di deadlock

Deadlock provocato da un uso scorretto delle primitive di sincronizzazione

→ usare correttamente!

Deadlock provocato da un accesso non ordinato alle risorse

Esempio: accesso a due risorse R1 ed R2 da assumere in mutua esclusione

→ Semafori s1, s2 con valore iniziale s1=s2=1

processo P1

```
...  
wait (s1)  
<uso di R1>  
wait (s2)  
<uso di R1 e R2>  
signal (s2)  
<uso di R1>  
signal (s1)  
...
```

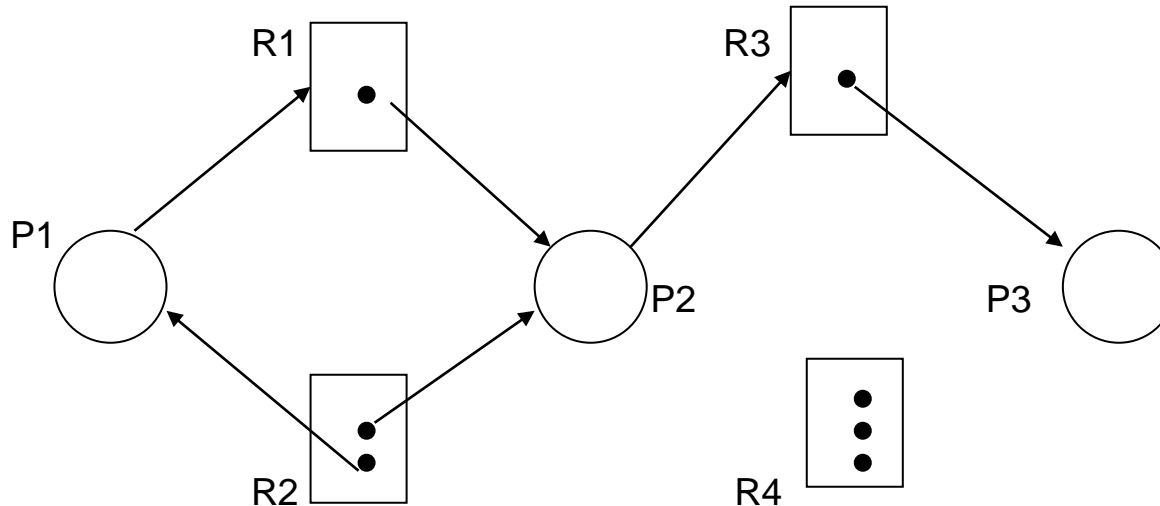
processo P2

```
...  
wait (s2)  
<uso di R2>  
wait (s1)  
<uso di R1 e R2>  
signal (s1)  
<uso di R1>  
signal (s2)  
...
```

Grafo di allocazione delle risorse

- $G = \{V, E\}$, V insieme di vertici, E insieme di lati
- L'insieme V è suddiviso in due tipi (*grafo bipartito*):
 - $P = \{P_1, P_2, \dots, P_n\}$ insieme dei processi
 - $R = \{R_1, R_2, \dots, R_m\}$ insieme dei tipi di risorse
- Ogni elemento di E è una coppia ordinata (P_i, R_j) o (R_j, P_i)
- (P_i, R_j) rappresenta un lato diretto da P_i a R_j e significa che P_i richiede una istanza del tipo di risorsa R_j ed è in attesa per questa risorsa (*lato di richiesta*)
- (R_j, P_i) rappresenta un lato diretto dal tipo di risorsa R_j al processo P_i e significa che una istanza di R_j è stata allocata a P_i (*lato di assegnamento o di allocazione*)
- Quando una richiesta è *soddisfatta* il lato di richiesta viene trasformato in lato di assegnazione. Quando il processo rilascia la risorsa, il lato viene *eliminato*

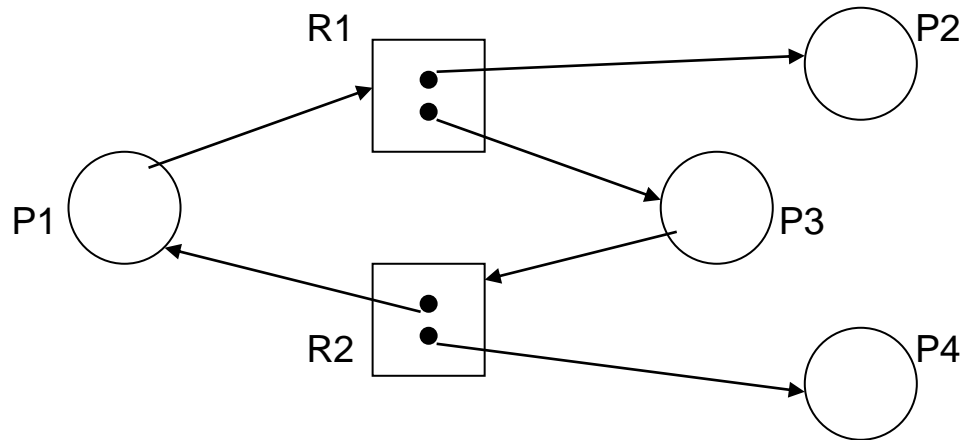
Grafo di allocazione delle risorse



- Se il grafo non contiene cicli, non c'è deadlock
- Se il grafo contiene un ciclo ci può essere deadlock (condizione necessaria); il SO può effettuare una verifica dei processi coinvolti nel ciclo per verificare se c'è deadlock
- Se è presente una sola istanza per ogni tipo di risorsa e c'è un ciclo allora c'è deadlock (necessaria e sufficiente)
- Se, a partire dallo stato in figura, P3 richiede un'istanza di R2?

Grafo di allocazione delle risorse

- Il ciclo P1-R1-P3-R2-P1 non comporta deadlock!



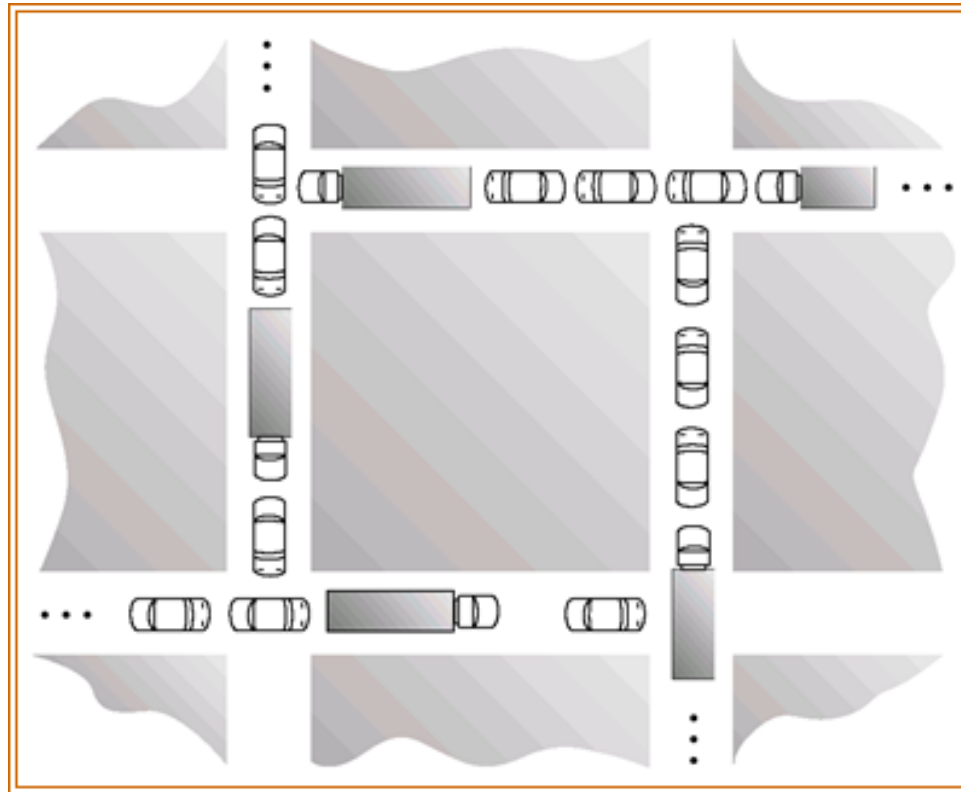
Condizioni per il deadlock

Coffman et al. (1971) hanno dimostrato che è **necessario** che si verifichino quattro condizioni perché vi sia deadlock:

1. **Mutua Esclusione:** ogni risorsa risulta assegnata esattamente ad un processo oppure è disponibile.
2. **Possesso e attesa (hold and wait):** i processi che detengono risorse assegnategli in precedenza possono richiederne di nuove.
3. **Assenza di revoca:** le risorse precedentemente assegnate non possono essere revocate forzatamente.
4. **Attesa circolare:** è presente una situazione di attesa circolare tra un gruppo di processi: ogni processo è in attesa di una risorsa posseduta dal processo che segue nella lista, e l'ultimo processo è in attesa di una risorsa detenuta dal primo processo considerato

Un esempio di deadlock

- Deadlock con risorse riusabili:



Strategie per evitare il deadlock

1. Ignorare il problema

- Algoritmo dello struzzo ...

Anche UNIX può soffrire di deadlock

- Esempio: si supponga che la tabella dei processi abbia dimensione fissa di 100 elementi (nel kernel Linux 2.6 il limite è la memoria fisica disponibile)
 - Se la tabella è piena, fork() fallisce e il processo può decidere di riprovare dopo un intervallo casuale
 - 10 processi ciascuno dei quali vuole creare 12 processi figli
 - Dopo che ogni processo ha creato 9 figli c'è il deadlock

Meglio accettare deadlock occasionali piuttosto che dover utilizzare una sola risorsa alla volta

- compromesso tra utilità correttezza
- eliminare deadlock è costoso
- plausibile nei sistemi interattivi di tipo personale per il codice utente

Il deadlock in UNIX viene comunque evitato nel codice del SO

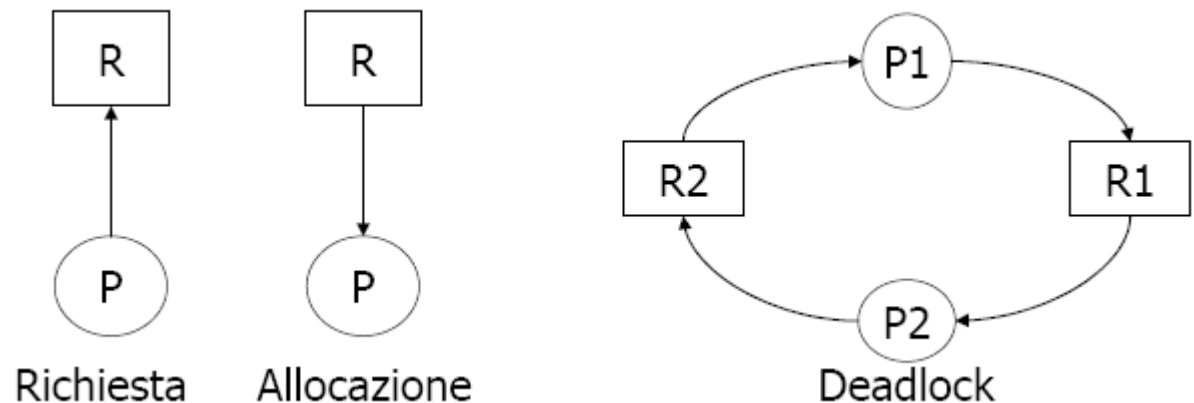
Strategie per evitare il deadlock

SisOp
2018/19



2. Rilevare la presenza di deadlock e risolverlo

- Approccio denominato *detection & recovery*
- Periodicamente il S.O. controlla un grafo di allocazione delle risorse (costo quadratico nel numero di processi e risorse)
- Se c'è un ciclo (deadlock) si terminano uno o più processi nel ciclo (recuperando le loro risorse) fino a quando non vi è più deadlock
- Quale processo terminare? A caso o in modo informato? E' possibile sottrarre solo alcune risorse e far ripartire il processo?



Strategie per evitare il deadlock

3. Prevenzione dinamica

- allocazione attenta delle risorse

Vincolare i processi in modo che il deadlock risulti strutturalmente impossibile mediante decisioni prese a tempo di esecuzione

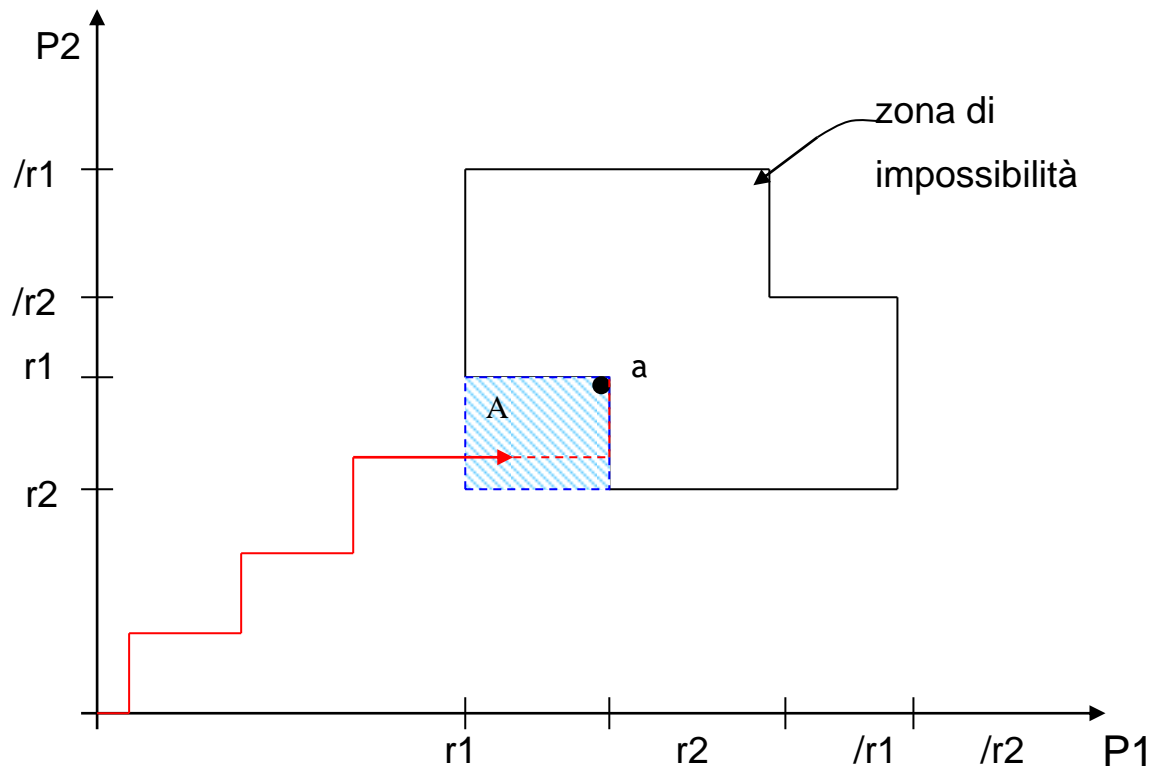
–Algoritmo del Banchiere (Dijkstra)

Il banchiere (SO) non soddisfa le richieste di credito (risorse) dei clienti (processi) che possono portare a stati di deadlock

- L'algoritmo richiede di conoscere a priori il massimo numero di risorse richieste da ciascun processo e presuppone, come caso peggiore, che ogni processo possa richiedere contemporaneamente il *numero massimo* di risorse dichiarato e *mantenerle tutte* durante l'esecuzione (caso peggiore).
- L'algoritmo identifica le situazioni rischiose e nega l'assegnazione di risorse disponibili quando la loro attribuzione potrebbe portare ad un deadlock
- Il costo quadratico viene pagato ad ogni richiesta di allocazione di una risorsa libera
- Appropriato per sistemi di elaborazione con carico statico noto (es. gestionali)

Algoritmo del Banchiere

- Visualizzazione di esempio con due risorse e due processi:
 - **A** stato non salvo, **a** stato di deadlock





4. Prevenzione Strutturale (statica)

- Occorre negare una delle 4 condizioni di Havender (o Coffman)
- Normalmente si evita l'instaurarsi di una **condizione di attesa circolare** imponendo *un ordine sulla richiesta delle risorse*
 - Richiede la *collaborazione attiva* dei programmatori (in qualche caso con supporto da parte del SO)
 - Metodo utilizzato all'interno del SO
 - Utilizzato nei sistemi in tempo reale
 - Ragionevolmente efficiente, occorre prestare attenzione ai costi indiretti
- Come fare se l'ordine delle richieste di un processo P_i non coincide con l'ordine fissato per prevenire il deadlock?

Esempio di prevenzione statica del deadlock

Esercizio: dato il problema di accesso a due risorse da assumere in mutua esclusione (codice sottostante che fa uso di due semafori di mutua esclusione s1, s2 con v.i.=1), individuare una soluzione alternativa esente da rischio del deadlock basata su prevenzione statica

processo P1

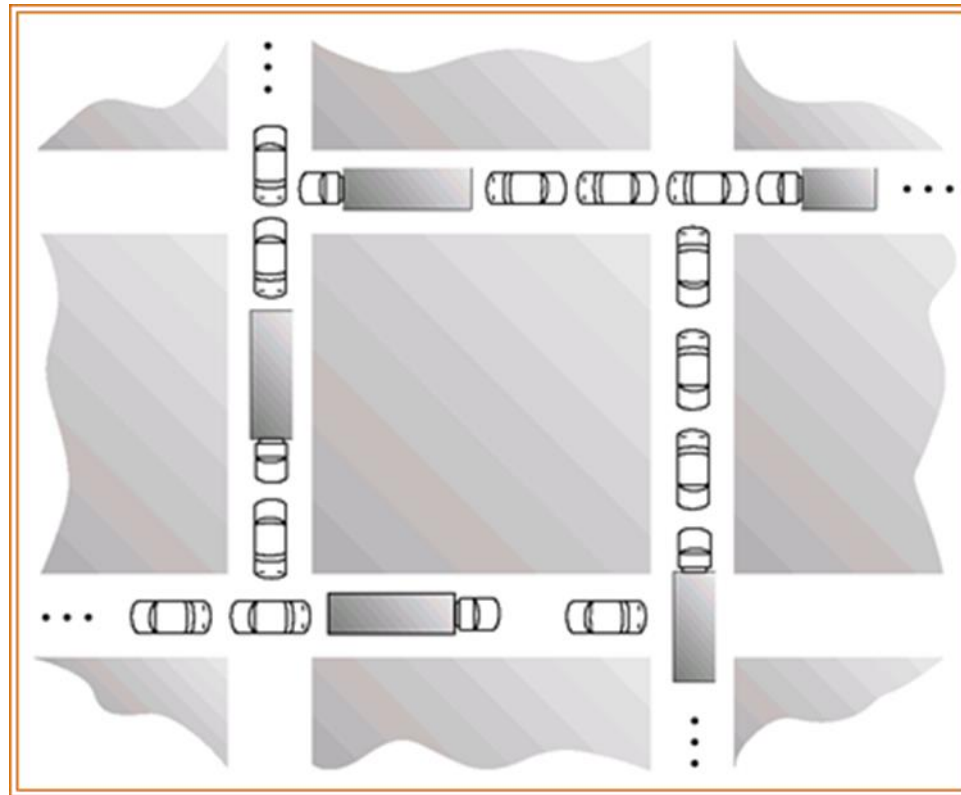
```
...  
wait (s1)  
...  
wait (s2)  
...  
signal (s2)  
...  
signal (s1)
```

processo P2

```
...  
wait (s2)  
...  
wait (s1)  
...  
signal (s1)  
...  
signal (s2)
```

Prevenzione statica del deadlock

- Applicare le tecniche di prevenzione statica in modo da impedire il deadlock del traffico



Il deadlock nei sistemi reali

SisOp
2018/19



- Nei sistemi interattivi, il deadlock è in genere ignorato nel codice utente (l'utente termina l'applicazione che «non risponde»)
- Il deadlock è sempre prevenuto staticamente nei sistemi in tempo reale
- Il deadlock è prevenuto all'interno del kernel del SO (con preallocazione in blocco di parte delle risorse e loro successiva allocazione gerarchica); i programmatori del SO seguono un ordine concordato nell'acquisire le risorse
- Il deadlock viene talvolta prevenuto dinamicamente o con tecniche di detection & recovery nei sistemi gestionali con carico transazionale stabile e in alcuni sistemi di calcolo
- A volte i sistemi integrano strategie miste
- Come utenti e programmatori dobbiamo essere a conoscenza del problema del deadlock anche se non intendiamo gestirlo attivamente!

Lo scheduling della CPU

SisOp
2018/19



Scheduling della CPU

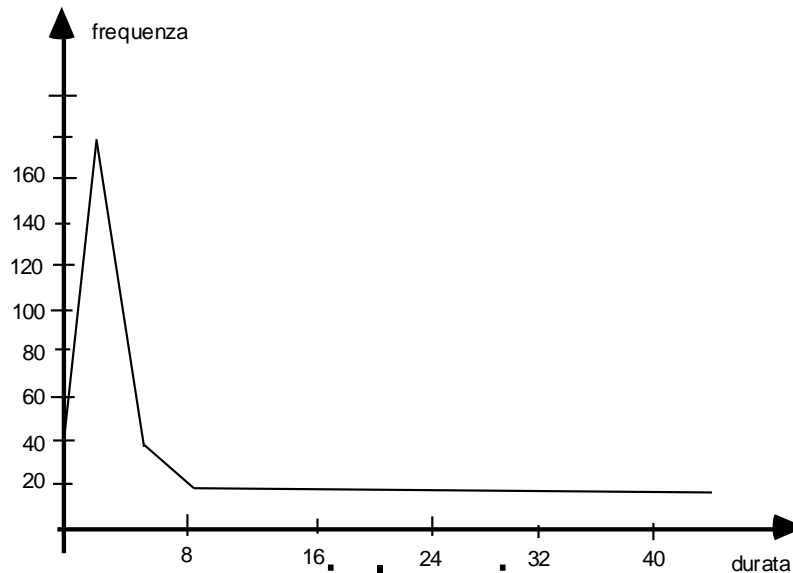
- La gestione delle risorse impone al SO di prendere decisioni sulla loro assegnazione in base a criteri di efficienza e funzionalità
- Le risorse più importanti, a questo riguardo, sono la CPU e la memoria principale.
- Scheduler (della CPU):
 - parte del S.O. che decide a quale dei *processi* (o *thread*) pronti presenti nel sistema *assegnare il controllo della CPU*
- Algoritmo di scheduling (assegnazione della CPU):
 - realizza *un particolare criterio di scelta tra i processi pronti (politica)*
 - in base a quali elementi ?

Scheduling della CPU

- Proprietà dei processi: l'esecuzione di un processo *alterna* attività di CPU e attesa per I/O

```
...  
LOAD                }  
STORE               }  
ADD                 } CPU burst  
STORE               } (sequenza di operazioni di CPU contigue)  
READ from file      }  
  
WAIT for I/O        } I/O burst  
  
STORE               }  
INCREMENT index     } CPU burst  
WRITE to file       }  
  
WAIT for I/O        } I/O burst  
  
LOAD                }  
STORE               }  
ADD                 } CPU burst  
STORE               }  
READ from file      }  
  
WAIT for I/O        } I/O burst  
...
```

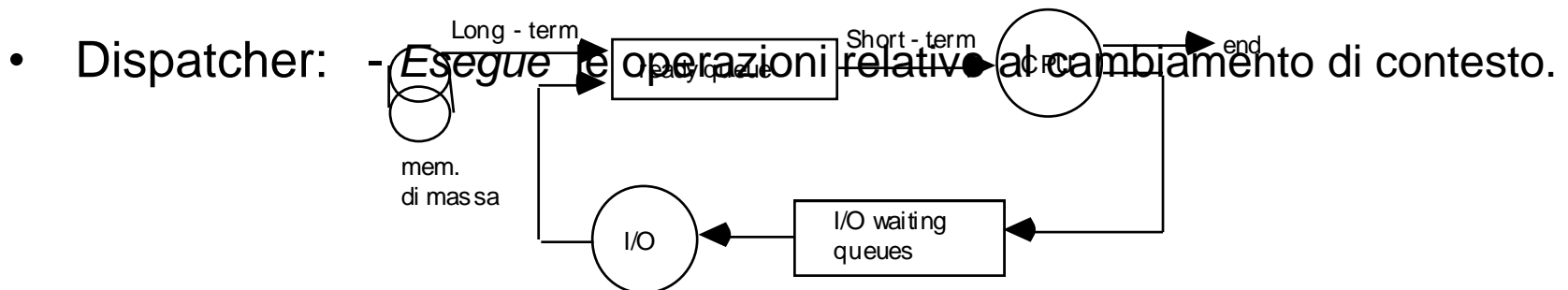
CPU bursts



- Andamento esponenziale o iperesponenziale: un gran numero di burst molto brevi ed un piccolo numero di burst molto lunghi
- Un programma I/O bound ha molti burst di CPU, brevi
- Un programma CPU bound ha pochi burst di CPU, lunghi

CPU schedulers

- Long-term scheduler (o job scheduler):
 - Determina *quali processi* dalla memoria di massa devono essere caricati in memoria principale pronti per l'esecuzione.
 - *Controlla il grado di multiprogrammazione* (numero di processi in memoria). Interviene, di regola, quando un processo lascia il sistema.
 - Il *criterio di selezione* è basato su un mix equilibrato di job I/O bound e CPU bound.
- Short-term scheduler:
 - *Seleziona* tra tutti i processi in memoria pronti per l'esecuzione quello *cui assegnare la CPU*.
 - deve essere *efficiente* in quanto interviene *frequentemente*.



CPU schedulers

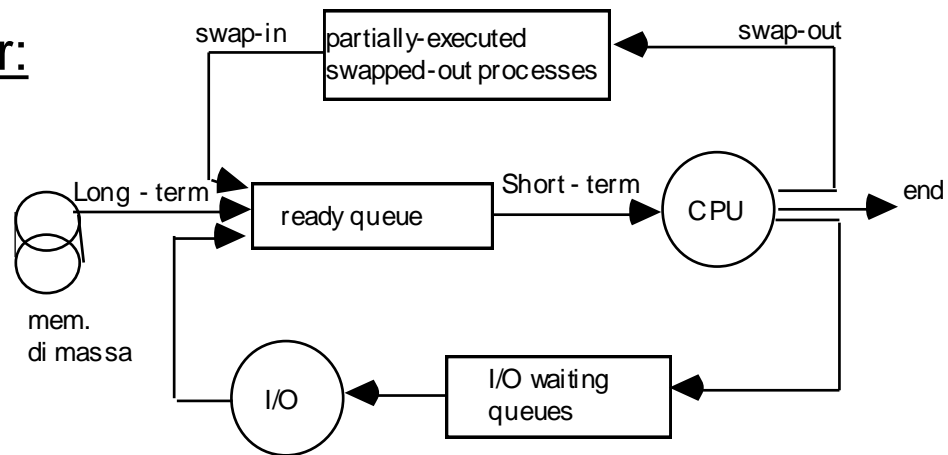
SisOp
2018/19



- Nei sistemi *time-sharing / interattivi* non esiste long-term scheduling.

I processi entrano immediatamente in memoria centrale. Il limite è imposto o dal numero di terminali connessi o dal tempo di risposta che diviene troppo lungo (sconsigliando l'uso del sistema quando sovraccarico).

- Medium-term scheduler:



- Può risultare vantaggioso, talvolta, rimuovere alcuni processi dalla memoria e *ridurre il grado di multiprogrammazione*. I processi vengono successivamente reintrodotti (*swapping*).

Scheduling e revoca della CPU

SisOp
2018/19



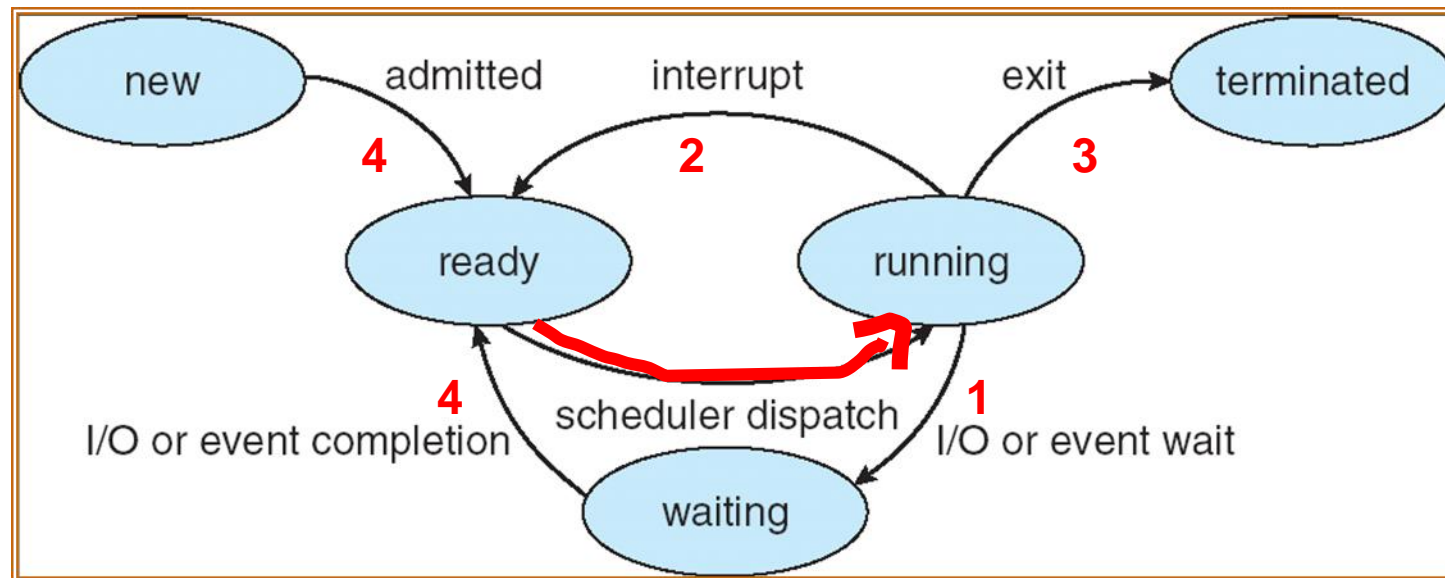
- La riassegnazione della CPU può avvenire a seguito di uno dei seguenti *eventi*:
 1. Un processo commuta *dallo stato di esecuzione a sospeso* (ad es., per richiesta di operazione di I/O, wait su semaforo, etc.).
 2. Un processo commuta *dallo stato di esecuzione a pronto* (ad es., a seguito della elaborazione di un interrupt, oppure perché il processo esegue una *yield*).
 3. Il processo in esecuzione *termina*.
 4. Un processo commuta *dallo stato sospeso a pronto* (ad es., per il completamento di un'operazione di I/O).
- Scheduling *non-preemptive*:
 - un processo in esecuzione prosegue fino al rilascio spontaneo della CPU;
 - la riassegnazione della CPU avviene solo a seguito di eventi di tipo 1 e 3, o di tipo 2 nel caso di yield.
- Scheduling *preemptive*:
 - il processo in esecuzione può perdere il controllo della CPU anche se "logicamente" in grado di proseguire;
 - la riassegnazione della CPU può avvenire anche a seguito di eventi di tipo 2 e 4.

Scheduling CPU

SisOp
2018/19



- Eventi che possono determinare revoca e riassegnazione della CPU:



Algoritmi di scheduling

- Criteri:
 - CPU utilization (40% - 90%)
 - Throughput
 - Turnaround time (tempo in mem. di massa, coda "pronti", esecuzione, I/O)
 - Waiting time (tempo speso nella coda dei processi pronti)
 - Response time
 - Fairness (assenza di privilegi)
- Scelto un criterio, si cerca di ottimizzare (minimizzare o massimizzare).
- Per sistemi interattivi (time sharing) è più importante minimizzare la *varianza nel tempo di risposta piuttosto che il tempo medio di risposta*.
- Misura di confronto spesso scelta: tempo medio di attesa (*waiting time*).
- Un'analisi accurata dovrebbe comprendere molti processi, ciascuno costituito da una sequenza di centinaia o migliaia di CPU burst ed I/O burst. Si utilizzano tracce di workload reali.
- Per semplicità nel seguito consideriamo per ciascun processo *un solo burst di CPU*.

FIRST COME, FIRST SERVED (F.C.F.S.)

- La CPU viene assegnata al processo che l'ha richiesta per primo.
- La realizzazione di questa politica è ottenuta con code gestite in modo FIFO.
- Le prestazioni di questo algoritmo sono in genere *scadenti in termini di tempo medio di attesa*.
- Pro: semplice da realizzare, *fair* (equo); Contro: prestazioni scadenti

- Esempio

Processi	burst di CPU
1	24
2	3
3	3

I processi arrivano nell'ordine 1, 2, 3 e sono serviti con politica FCFS

- Tempi di attesa: per il processo 1 è 0
 per 2 è 24
 per 3 è 24+3=27
- Tempo medio di attesa: $(0 + 24 + 27) / 3 = 17$
- Se i processi fossero arrivati nell'ordine 2, 3, 1 si avrebbe:
tempo medio di attesa: $(6 + 0 + 3) / 3 = 3$

Shortest Job First (S.J.F.)

SisOp
2018/19



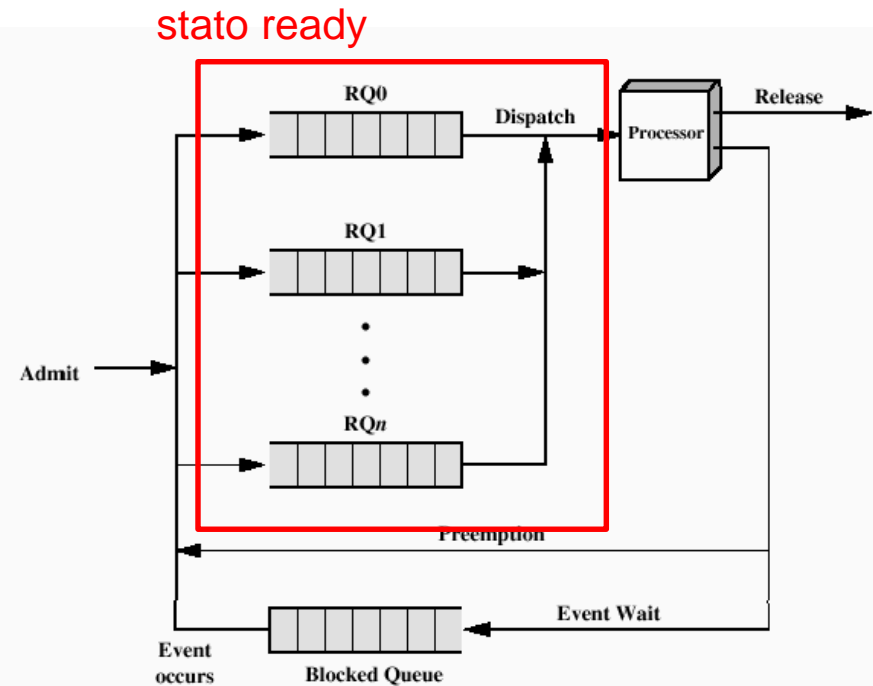
- A ciascun processo è associata *la lunghezza del successivo burst di CPU*. Quando la CPU è libera, essa viene assegnata al processo con il burst di CPU più breve.

Processi	Burst time
1	6
2	8
3	7
4	3

- Adottando SJF il tempo medio di attesa è 7 (con FCFS si sarebbe ottenuto 10.25)
- Si può dimostrare che SJF fornisce la soluzione *ottima*.**
Infatti, se si esegue un processo breve prima di uno lungo il tempo di attesa del processo breve diminuisce più di quanto aumenti il tempo di attesa di quello lungo.
- La difficoltà sta nel definire la lunghezza della successiva richiesta di CPU. Si può *predire* tale lunghezza facendo ad esempio l'ipotesi che sia simile a quella del burst precedente.
- Modelli analitici:* il burst successivo di CPU viene stimato come una media esponenziale delle lunghezze dei precedenti burst
- $$T_{n+1}^* = a t_n + (1 - a) T_n^* \quad 0 \leq a \leq 1$$
- t_n = lunghezza dell' n -esimo burst di CPU
- T_{n+1}^* = stima della lunghezza dell' $n+1$ -esimo burst di CPU

Priorità

- Il SO mantiene i processi pronti in code separate per i diversi livelli di priorità.
- Lo scheduler seleziona sempre il primo processo nella coda al livello massimo di priorità che contiene processi pronti.
- In presenza di preemption, in ogni istante è in esecuzione un processo a priorità massima.

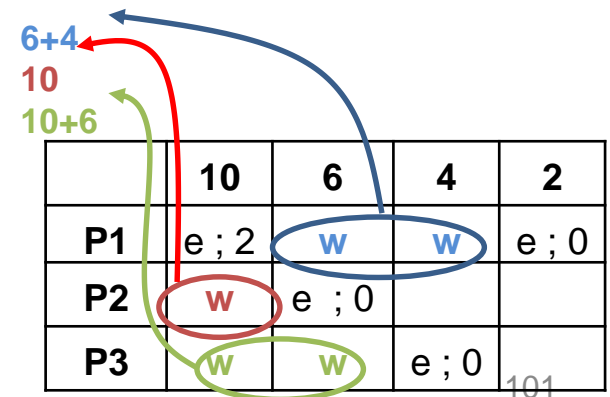


Priorità

- Valutata *internamente* o *esternamente*.
- Nel primo caso la priorità è individuata sulla base di qualche quantità misurabile. Ad es., limiti di tempo, richieste di memoria, numero di file aperti, rapporto tra le durate medie dei burst di I/O e di CPU, etc.
- Nel secondo caso la priorità è imposta da condizioni esterne.
- Problema di *starvation*: processi a bassa priorità possono rimanere indefinitamente ritardati.
- Una soluzione è quella di *aumentare* gradualmente la priorità dei job che attendono.

Priorità

- Priorità *statica*: attribuita ai processi all'atto della creazione in base alle loro caratteristiche o a politiche riferite al tipo di utente.
- In generale vengono favoriti i processi di sistema e di I/O; inoltre:
 - processi foreground (interattivi): => alta priorità
 - processi background (batch): => bassa priorità
- Possibilità di starvation.
- Priorità *dinamica*: modificata durante l'esecuzione del processo.
 - per penalizzare i processi che impegnano troppo la CPU
 - per evitare starvation
 - per favorire i processi che si dimostrano I/O-bound
 - per mantenere (indirettamente) un buon job-mix.



Algoritmi di scheduling

- Gli algoritmi FCFS, SJF e a priorità visti in precedenza sono di tipo *non-preemptive*, cioè quando la CPU è stata assegnata ad un processo questi ne mantiene il controllo fino al proprio completamento, o alla richiesta di una operazione di I/O, o all'esecuzione di una sincronizzazione sospensiva.
- Gli algoritmi SJF e a priorità possono essere anche di tipo *preemptive*. Tale possibilità nasce quando, durante l'esecuzione di un processo, un nuovo processo entra nella coda dei processi pronti. Il nuovo processo può richiedere un tempo di CPU inferiore o avere una priorità maggiore di quello in esecuzione.
- L'algoritmo SJF di tipo preemptive viene chiamato anche *shortest remaining time first (SRTF)*.

Code a più livelli

SisOp
2018/19



- Suddivisione tra job *foreground* (*interattivi*) e *background* (*batch*). Algoritmi diversi in quanto sono diverse le esigenze.
- Più in generale, possono esistere *classi diverse* di job che vengono assegnati *staticamente* ad una coda. Ogni coda ha un proprio algoritmo di scheduling. Ad esempio, per i job foreground l'algoritmo R.R., per quelli background l'algoritmo FCFS.
- In taluni casi può essere consentito ad un job di cambiare, eventualmente temporaneamente, coda. Un job che usa troppo tempo di CPU può passare ad un livello inferiore; un job che attende da troppo tempo può passare ad un livello superiore.
- Occorre definire:
 - il numero di code
 - l'algoritmo di scheduling per ogni coda
 - un criterio per decidere quando spostare un job ad una coda di priorità più elevata
 - un criterio per decidere quando spostare un job ad una coda di priorità più bassa
 - un metodo per determinare in quale coda un job deve entrare quando inizia il servizio.

Valutazione degli algoritmi

- *Scelta dei criteri. Ad esempio:*
 - massimizzare l'utilizzazione della CPU con il vincolo che il tempo di risposta max sia 1 sec
 - massimizzare il throughput in modo che il tempo di risposta sia (in media) proporzionale al tempo di esecuzione totale.
- Valutazione analitica
 - a) *Modelli deterministici:* Fissato un carico di lavoro viene definita la performance di ciascun algoritmo.
- Esempio: supponiamo che al tempo 0 arrivino 5 job nel seguente ordine:

Processo	burst time	priorità
1	10	2
2	29	3
3	3	4
4	7	5
5	12	1
- Si considerino gli algoritmi FCFS, SJF, PRIO, RR con quanto = 10. Determinare il minimo dei tempi medi di attesa.
- Si ottiene:

FCFS	$T = (0 + 10 + 39 + 42 + 49) / 5 = 28$
SJF	$T = (10 + 32 + 0 + 3 + 20) / 5 = 13$
PRIO	$T = (12 + 22 + 51 + 54 + 0) / 5 = 27,8$

Valutazione degli algoritmi

Processo	burst time
1	10
2	29
3	3
4	7
5	12

- Si ottiene: RR ($q = 10$) $T = (0 + 32 + 20 + 23 + 40) / 5 = 23$

Esecuzione dei job
con scheduling RR
($q = 10$)

	10	10	3	7	10	10	2	9
J1	e ; 0							
J2	w	e ; 19	w	w	w	e : 9	w	e ; 0
J3	w	w	e ; 0					
J4	w	w	w	e ; 0				
J5	w	w	w	w	e ; 2	w	e ; 0	

Valutazione degli algoritmi

SisOp
2018/19



- b) *Modelli basati sulla teoria delle code.*
Il sistema di calcolo è descritto come una *rete di servitori*. Ciascun servitore ha una coda di job in attesa. Conoscendo le frequenze di arrivo e i tempi di servizio (in termini di distribuzione di probabilità) si può calcolare la lunghezza media delle code, i tempi medi di attesa, etc.
- Limitazioni del metodo. Occorre conoscere le distribuzioni di probabilità dei burst di CPU e di I/O e dei tempi di arrivo nel sistema dei job. Per rendere il problema trattabile occorre definire distribuzioni per l'arrivo ed il servizio sufficientemente semplici (uniforme, esponenziale, etc.). Inoltre è difficile o impossibile esprimere sincronizzazioni. La soluzione è approssimata e spesso la sua validità è incerta.
- c) *Modelli basati su reti di Petri temporizzate.*
Il sistema viene descritto in termini di *eventi* (sparo di transizioni) e *condizioni* (marcatura dei posti). Le attività sono tipicamente rappresentate tramite transizioni temporizzate.
- Occorre caratterizzare le durate delle attività dei job in maniera analoga a quanto avviene con le reti di code. E' possibile esprimere sincronizzazioni ma la risolubilità analitica è condizionata da limitazioni forti sulle distribuzioni ammesse. Inoltre la modellazione di alcune politiche di scheduling, possibile in linea di principio, dà luogo a reti complicate.

Valutazione degli algoritmi

SisOp
2018/19



- Simulazione
- Si costruisce un modello del sistema utilizzando le reti di code o le reti di Petri come strumento di rappresentazione. Non si risolve analiticamente il modello ma lo si simula tramite un programma di calcolo.
- Le distribuzioni possono essere definite matematicamente o empiricamente, desumendole da misure effettive sul sistema.
- Spesso si fa uso di tracce di esecuzione (storicamente *trace tapes*) create registrando la *sequenza reale* di eventi nel sistema e fornendola come dati al simulatore. Viene usata per confrontare algoritmi diversi.

Lo scheduling in UNIX

SisOp
2018/19



- L'algoritmo di scheduling favorisce i *job di tipo interattivo (foreground)*.
- Si tratta di un algoritmo *round-robin con priorità (variabile)*.
- Ad ogni processo è associata una priorità di scheduling. La priorità è rappresentata in senso decrescente: più è basso il valore, più è elevata la priorità.
- Processi che svolgono attività di I/O su disco hanno priorità negativa e non possono essere interrotti.
- La priorità *varia dinamicamente*: al crescere del tempo di CPU utilizzato da un processo diminuisce la sua priorità. Analogamente, al crescere del tempo di attesa di un processo aumenta anche la sua priorità (per evitare *starvation*).
- UNIX 4.2 BSD, ampiamente documentato in letteratura, ha un quanto di tempo di 0.1 secondi e ricalcola la priorità *ogni secondo*.

Lo scheduling in UNIX (BSD4.3)

- Le priorità variano tra 0 (massima) e 127 (minima). Da 0 a 49 per i processi che eseguono in modo kernel, da 50 a 127 per i processi in modo utente.
- Calcolo della priorità di un processo in modo utente (ad es. ogni 40 msec):

$$p_usrpri = PUSER + (p_cpu / 4) + 2 * p_nice$$

- ove:
 - p_usrpri saturato a 127
 - $PUSER = 50$ (base priority for user mode execution)
 - p_nice varia tra -20 e 20, default 0
 - p_cpu incrementato ad ogni tick in cui il processo viene trovato in esecuzione
 - un correttivo, applicato ogni 1 sec, fa decadere il 90% di p_cpu in circa 5 sec
 - un correttivo diminuisce p_cpu per i processi a lungo sospesi.

Sistemi real-time

SisOp
2018/19



- In un sistema *real-time* la correttezza della elaborazione dipende sia dalla sua correttezza logica sia *dall'istante in cui il risultato viene generato*. Il mancato rispetto dei vincoli temporali equivale ad un guasto del sistema (*system failure*).
- Garantire la correttezza del comportamento temporale richiede che il sistema sia *altamente predicibile*. Le esigenze di tempo reale sono spesso in conflitto con gli usuali requisiti di efficienza nell'uso delle risorse.
- Sistemi *soft real-time*: i vincoli sui tempi di risposta sono espressi come distribuzioni statistiche e scostamenti massimi ammessi.
- Sistemi *hard real-time*: i vincoli devono essere rispettati in modo rigoroso.
- Nei sistemi *real-time* organizzati a processi tipicamente sono presenti sia *processi critici*, che devono soddisfare i vincoli temporali, che *processi non critici*, eseguiti con algoritmi di scheduling convenzionali (es. FCFS) negli intervalli di tempo residui.

Lo scheduling in UNIX

SisOp
2018/19



- Viene usato il meccanismo di *time-out*: ogni quanto di tempo (0.1 sec, 1 sec) l'interruzione di clock mette in funzione una procedura che esegue l'azione richiesta (cambiamento di contesto, ricalcolo delle priorità) e predispone il clock per essere nuovamente chiamata.
- Un processo sospende l'esecuzione tramite la primitiva del nucleo *sleep* che ha come parametro l'indirizzo di una struttura dati del kernel relativa ad un *event* che il processo attende prima di risvegliarsi.
- Quando si verifica un *event*, il nucleo provvede a risvegliare *tutti* i processi in attesa di *event*. I processi vengono messi in coda per essere scelti dal meccanismo di scheduling.
- Possono nascere "condizioni di corsa" relative al meccanismo degli eventi. Se un processo decide di sospendersi in attesa di un evento e l'evento si verifica prima che il processo completi la primitiva *sleep*, il processo rimane in attesa indefinita (deadlock). (Non c'è memoria associata agli eventi).
- Una soluzione al problema consiste nell'impedire all'evento di verificarsi durante l'esecuzione della primitiva (innalzando la priorità hardware della CPU in modo che non si possano verificare interruzioni).