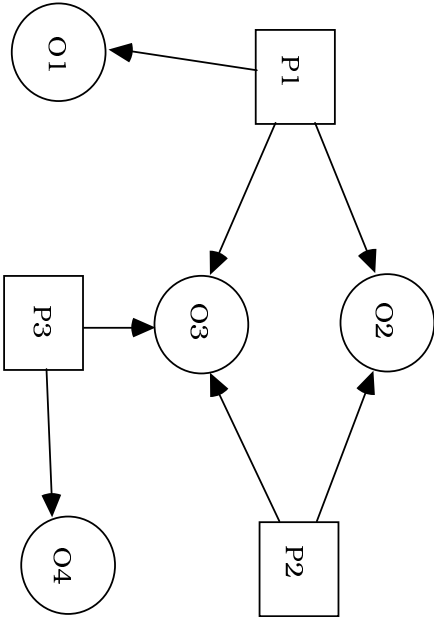


Modelli di interazione tra processi

- Modelli ad ambiente globale - global environment
- Modelli a scambio di messaggi - message passing

Modello ad ambiente globale

- Il sistema é visto come un insieme di:
 - processi
 - oggetti (risorse)

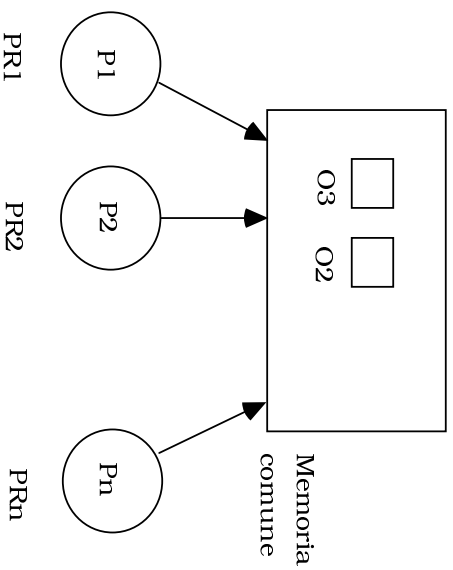


-  diritto di accesso
- O1, O4 risorse private (class)
- O2, O3 risorse comuni (monitor)

- Tipi di interazione tra processi:
 - competizione
 - cooperazione

Modello ad ambiente globale

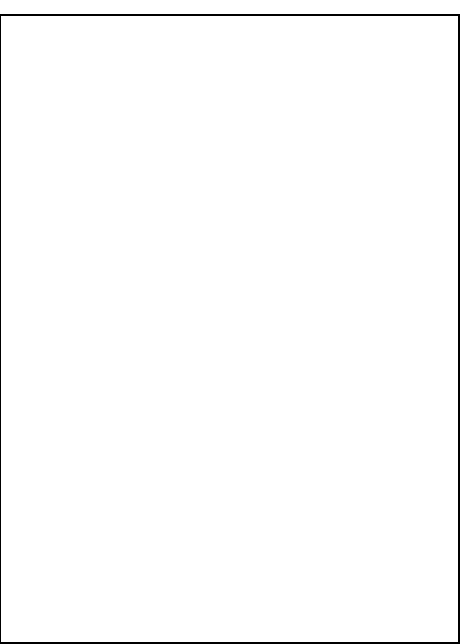
- Il modello ad ambiente globale rappresenta la naturale astrazione di un *sistema in multiprogrammazione* costituito da uno o più processori che hanno accesso ad una memoria comune.



- 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

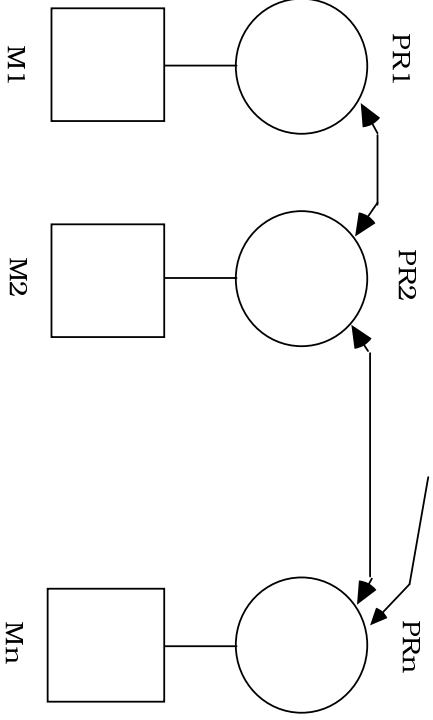
- Il sistema è visto come un insieme di processi ciascuno operante in un *ambiente locale* che non è accessibile direttamente a nessun altro processo.



- 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.

Strumenti di comunicazione

- Strumenti per la comunicazione e sincronizzazione tra processi in ambiente globale:

- semafori e primitive di sincronizzazione (P, V) (Dijkstra, 1965)
- monitor (Brinch Hansen, Hoare, 1973)
- path expressions (Campbell, Habermann, 1974)
- serializer (Hewitt, Atkinson, 1977)

Programmazione in ambiente globale

- Threads (più flussi di controllo per un unico spazio di indirizzamento)
- Java (semafori, monitor)

Entrambi permettono anche lo scambio di messaggi

- Strumenti per la comunicazione tra processi in ambiente a scambio di messaggi:

- **send (m), receive (m)**

m: messaggio;

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

Interazione tra processi

- Cooperazione:
 - Comprende tutte le interazioni *prevedibili e desiderate* insite cioè nella *logica dei programmi* (archi nel grato di precedenza ad ordinamento parziale)
 - 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.
- 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*.

Interazione tra processi

- Cooperazione => Sincronizzazione diretta o esplicita
- Competizione => Sincronizzazione indiretta o implicita
- 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*).

Processi interferenti

- 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

	Pi	Pj
	...	...
	contatore := contatore + 1	contatore := contatore + 1
	...	...
	LOAD A,contatore	
	INCR A	
	STORE contatore,A	
t0:	LOAD A,contatore	(Pi)
t1:	LOAD A,contatore	(Pj)
t2:	INCR A	(Pj)
t3:	STORE contatore,A	(Pj)
t4:	INCR A	(Pi)
t5	STORE contatore,A	(Pi)

Esempio

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

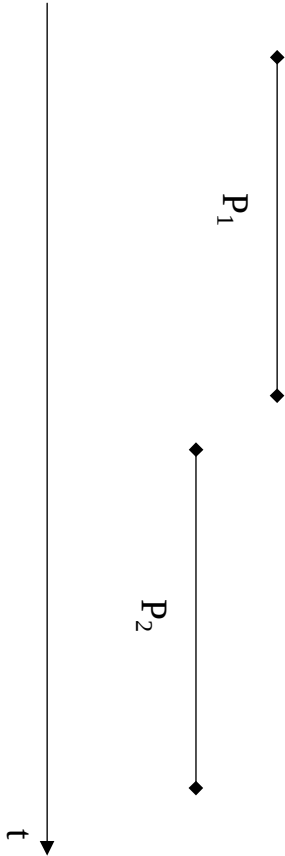


- Le istruzioni di P e Q possono mescolarsi arbitrariamente e dar luogo a diverse possibili sequenze di esecuzione:

v := v+1;	(P)	print v;	(Q)	print v;	(Q)
print 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 mutua esclusione quando non piu` di un processo alla volta puo` accedere a *variabili comuni*



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

Esempio di mutua esclusione

- P1 e P2 utilizzano una stessa telescrivente.

- La telescrivente deve essere assegnata ad un solo processo alla volta per tutta la durata del suo uso.

- Ipotesi di soluzione:

<u>Richiesta</u>	<u>Rilascio</u>
...	...
<i>repeat until</i> libera;	libera := true;
libera := false;	...
...	...

Valore iniziale libera := true;

- Possibile sequenza errata di esecuzione:

t0: <i>repeat until</i> libera	(P1)
t1: <i>repeat until</i> libera	(P2)
t2: libera := false	(P1)
t3: libera := false	(P2)

- La telescrivente risulta assegnata ad entrambi i processi.

Esempio di mutua esclusione

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

top	
stack	

<u>Inserimento (y)</u>	<u>Prelievo (x)</u>
...	...
top := top + 1;	x := stack [top];
stack [top] := y;	top := top - 1;
...	...

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

-Esempio - P1 inserisce e P2 preleva:

t0: top := top + 1	(P1)
t1: x := stack [top]	(P2)
t2 top := top - 1	(P2)
t3: stack [top] := y	(P1)

- 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 *piu`* 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 puo` essere in esecuzione ad ogni istante".

Soluzione al problema della mutua esclusione

- *Templificazione dell'esecuzione* dei singoli processi da parte del programmatore (errori "*time dependent*")
- *Inibizione delle interruzioni* del processor 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

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

$wait(s)$
 $P(s)$

$signal(s)$
 $V(s)$
- *wait* e *signal* sono realizzate tramite chiamate al S.O. (SVC) ed eseguite in modo *monitor*, cioè la variabile semaforo è *protetta*.

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;
```
- ```
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 *wait* può essere *passante* ($s > 0$) o *bloccante* ($s = 0$), nel qual caso si verifica un *context switch*. 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 e primitive wait e signal

- 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 sez. crit. della stessa classe; s semaforo (valore iniziale: $s_0 = 1$):

processo P1

processo P2

...

...

wait (s);

wait (s);

<sezione critica A>;

<sezione critica B>;

signal (s);

signal (s);

...

...

Indivisibilità di wait e signal

- Occorre garantire che l'azione di analisi e modifica del semaforo non sia separata dalla azione di sospensione.
- Esempio: $s = 0$
t0: if $s = 0$ (P1)
t1: $s := s + 1$ (P2)
t2: sospensione (P1)
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 la lock.

Indivisibilità di wait e signal

- Nel caso più 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

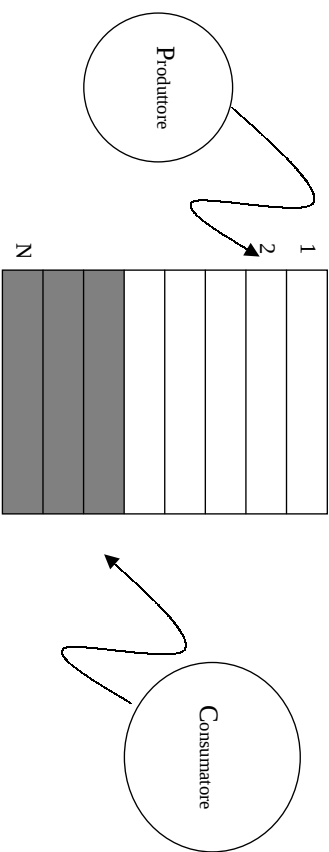
- I Livello:
sezioni critiche: S1, S2
mutua esclusione tramite wait e signal

- II Livello:
sezioni critiche: wait(s) e signal(s)
mutua esclusione tramite lock(x) e unlock(x)

- III Livello:
sezioni critiche: lock(x), unlock(x)
mutua esclusione tramite hardware (test-and-set)

Esempio: Produttore-Consumatore

- Buffer in grado di contenere N messaggi, a cui accedono il processo P in scrittura ed il processo C in lettura:



- 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
 - e = numero dei messaggi estratti
 - N = dimensione del buffer

$0 \leq d \leq N$

Produttore-Consumatore

- La soluzione richiede due semafori:
 - "messaggio disponibile" mess-disp valore iniziale 0
 - "spazio disponibile" spazio-disp valore iniziale N

```
Produttore (P)                      Consumatore (C)
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*).

Primitive per lo scambio di messaggi

- 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*: messaggio
- 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 é 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 é stato trasmesso; tale relazione di causa-effetto *vincola l'ordine* in cui i due eventi possono avvenire.

- La *mutua esclusione* non é più un problema, perché nel modello ad ambiente locale tutte le risorse sono *private*.

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

- 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 implementare 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 é 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

- 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) Pj (servitore)

```
...                               ...
send <richiesta>;                receive <richiesta>;
receive <risultato>;              <esecuzione>;
...                               send <risultato>;
...                               ...
```

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

I processi cliente *specificano* il *destinatario* delle loro richieste. Il processo servitore é 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 esplicita. Richiede di passare ad una *designazione indiretta* o *globale* (mailbox).

Modello *client-server* e *naming*

- Il *direct naming* é 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

- 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

- 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*.
- C'è una analogia con il caso della *mailbox* in ambiente a *memoria comune*, in cui pure non è specificata l'identità del particolare servitore.
- L'implementazione 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 *non è più disponibile per tutti gli altri servitori*.

Porte

- 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 --> comunicazione one to one
- port naming --> comunicazione many to one
- global naming --> comunicazione many to many
- Il global naming é il caso più generale. Gli altri due schemi limitano i *tipi di interazione direttamente programmabili* ma sono più semplici da implementare.
- 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*. (Difficoltà di 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 asincrona*),
 - *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

- 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: 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 implementata come un *processo servitore* che attende la ricezione di un messaggio, esegue il corpo e trasmette un messaggio di risposta.

- Può essere implementata 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.

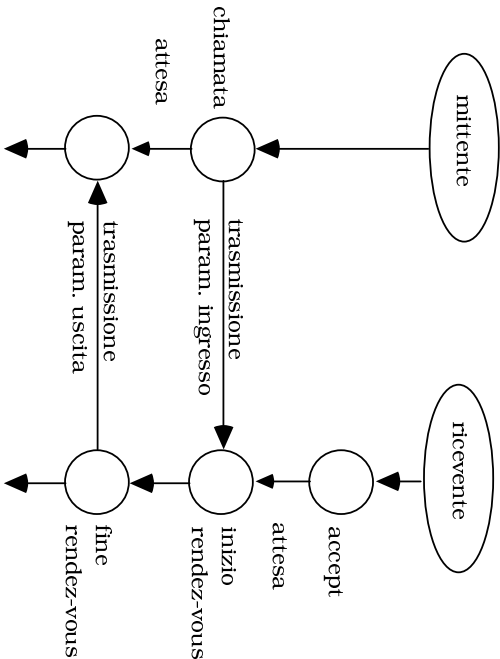
Al termine viene trasmesso il messaggio di risposta al processo chiamante, dopo di che il processo servitore *continua la propria esecuzione*.

- Ad ogni servizio è associata una coda distinta, generalmente di tipo FIFO.

Uso della *accept*

- 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 alla definizione di procedura.

Schema di comunicazione realizzato dalla RPC



Uso della RPC

- 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.

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 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 ?

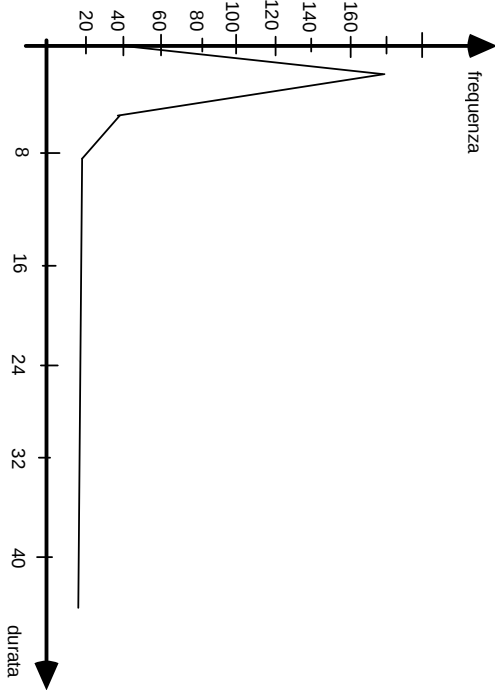
Assegnazione della CPU (CPU scheduling)

- Proprietà dei processi:

l'esecuzione di un processo alterna attività di CPU e attesa per I/O

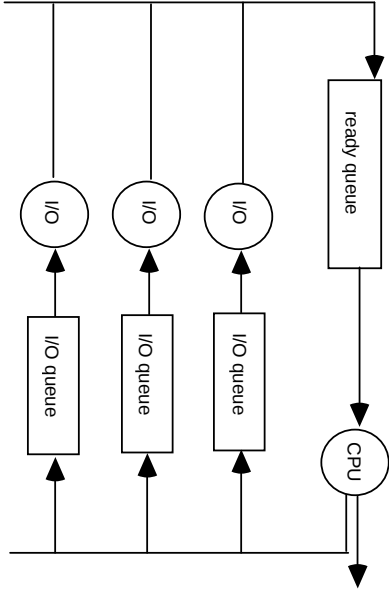
...	
LOAD	}
STORE	}
ADD	} <u>CPU burst</u>
STORE	} (seq. di operazioni di CPU contigue)
READ from file	}
WAIT for I/O	} <u>I/O burst</u>
STORE	}
INCREMENT index	} CPU burst
WRITE to file	}
WAIT for I/O	} I/O burst
LOAD	}
STORE	}
ADD	} CPU burst
STORE	}
READ from file	}
WAIT for I/O	} I/O burst
...	

CPU bursts



- Andamento esponenziale o iperesponenziale: un gran numero di burst molto brevi ed un piccolo numero di burst molto grandi

- 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*):

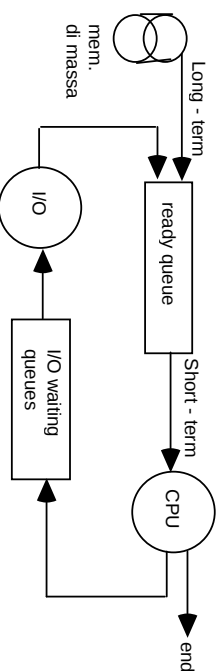
- Determina *quali processi* dalla memoria di massa devono essere caricati in memoria principale pronti per l'esecuzione.
- *Controlla il grado di multiprogrammazione* (numero di processi in memoria). Interviene, di regola, quando un processo abbandona il sistema.
- Il *criterio di selezione* e' basato su un mix equilibrato di jobs 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*.

- Dispatcher:

- *Esegue* le operazioni relative al cambiamento di contesto.

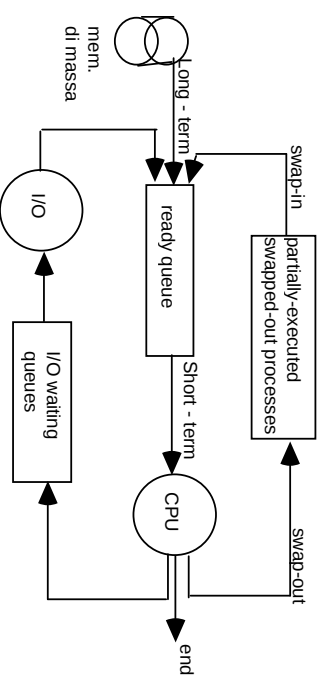


CPU schedulers

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

I processi entrano immediatamente in memoria centrale. Il limite e' 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 i processi dalla memoria e ridurre il grado di multiprogrammazione. I processi vengono successivamente reintrodotti. (*Swapping*)

Scheduling e revoca della CPU

- 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).
 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.
- 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/o 4.

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 tenta 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 scelta: *tempo medio di attesa* (waiting time).
- Un'analisi accurata dovrebbe comprendere molti processi, ciascuno costituito da una sequenza di diverse centinaia di CPU burst ed I/O burst. Per semplicità viene considerato 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 é ottenuta con code gestite FIFO.
- Le prestazioni di questo algoritmo sono in genere *basse* in termini di *tempo medio di attesa*.
- Es.:

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 é 0
 per 2 é 24
 per 3 é 27

- Tempo medio di attesa: $(0 + 24 + 27) / 3 = 17$

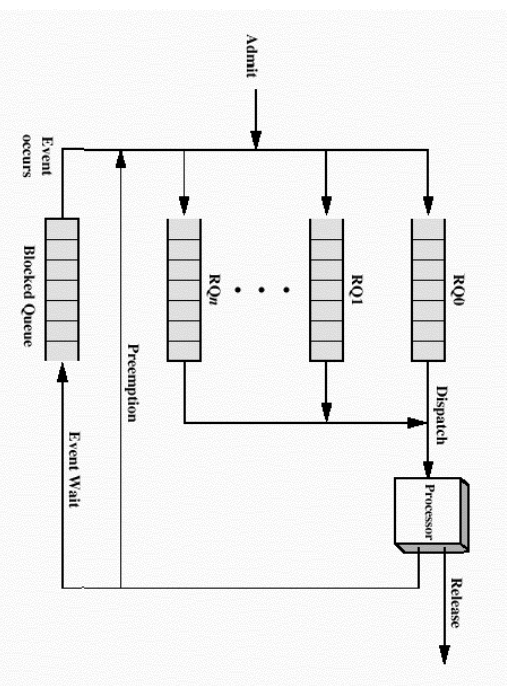
- Se i processi arrivano nell'ordine 2, 3, 1 si ha:
tempo medio di attesa: $(6 + 0 + 3) / 3 = 3$

Shortest-Job-First (S.J.F.)

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

Processi	Burst-time
1	6
2	8
3	7
4	3
- Adottando S.J.F., il tempo medio di attesa é 7 (con F.C.F.S. si sarebbe ottenuto 10,25)
- Si può dimostrare che S.J.F. 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
$$T^*_{n+1} = a \cdot t_n + (1 - a) \cdot T^*_n$$
$$0 < a < 1$$
$$t_n = \text{lunghezza dell'n-esimo burst di CPU}$$
$$T^*_{n+1} = \text{stima della lunghezza dell'n+1-esimo burst di CPU}$$

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 é in esecuzione un processo a priorità massima.

Priorità

- Valutata *internamente* o *esternamente*.
- Nel primo caso la priorità é individuata sulla base di qualche quantità misurabile. Ad es., limiti di tempo, richieste di memoria, numero di file aperti, rapporto tra i burst medi di I/O e di CPU, etc.
- Nel secondo caso la priorità é imposta da condizioni esterne.
- Problema di *starvation*: processi a bassa priorità possono rimanere indefinitamente ritardati.
- Una soluzione é 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 I/O-bound
 - per mantenere (indirettamente) un buon job-mix.

Round Robin (R.R.)

- Time sharing systems. Quanto di tempo di tempo assegnato a tutti i processi (10 - 100 msec).
- La coda dei processi pronti è *circolare* e la CPU è assegnata a ciascuno dei processi per un quanto di tempo.
- Un processo è interrotto per l'esaurimento del suo quanto, viene inserito come *ultimo* nella coda dei processi pronti (preemption).
- La politica di scheduling round-robin è caratterizzata da *elevata fairness* e *assenza di starvation*, tuttavia il context-switch *imposto* all'esaurimento del quanto di tempo determina un *incremento dell'overhead*.

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*.

Code a più livelli

- 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 massimo sia 1 secondo.
 - 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:

<u>job</u>	<u>burst time</u>
1	10
2	29
3	3
4	7
5	12

- Si considerino i tre algoritmi FCFS, SJF, RR con quanto = 10. Determinare il minimo dei tempi medi di attesa.

- Si ottiene:
- | | |
|-------------|--|
| FCFS | $T = (0 + 10 + 39 + 42 + 49) / 5 = 28$ |
| SJF | $T = (10 + 32 + 0 + 3 + 20) / 5 = 13$ |
| RR (q = 10) | $T = (0 + 32 + 20 + 23 + 40) / 5 = 23$ |

Valutazione degli algoritmi

- b) *Modelli basati sulla teoria delle code.*
- Il sistema di calcolo è descritto come una *rete di server*. 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

- 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 *trace tapes* create registrando la sequenza reale di eventi nel sistema e fornendola come dati al simulatore. Viene usata per confrontare *algoritmi diversi*.
- La simulazione affidabile di un modello è sempre molto più costosa della sua risoluzione analitica.

Lo scheduling in UNIX

- L'algoritmo di scheduling favorisce i *job di tipo interattivo* (foreground).
- Si tratta di un algoritmo *round-robin con priorità* (Variable).
- 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).
- 4.2 BSD ha un quanto di tempo di 0.1 sec 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.

Lo scheduling in UNIX

- 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 *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).

Sistemi real-time

- 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.