


Si illustra il concetto di processo, descrivendo anche i possibili stati in cui si può trovare in un SO multiprogrammato (P. 63)

Il processo è il sistema funzionale nel SO ed è controllato da un programma.

Esso ha bisogno di un processare per essere eseguito, che può essere privato (è permanentemente in esecuzione) o comune (come ad esempio la CPU).

Ci sono 2 tipi di processi:

Processo del SO, che esegue il codice del sistema.

Processo utente, che esegue il codice utente.

Il SO può: creare e cancellare i processi, sospenderli e riattivarli, ma deve anche fornire gli strumenti per la sincronizzazione e la comunicazione e per il trattamento delle condizioni di deadlock (cioè quando A detiene delle risorse utili a B e viceversa).

Un processo può anche essere definito come l'insieme del programma, unito ad altre informazioni.

Un processo si può trovare in 3 stati: esecuzione, pronto e bloccato.

- Si dice in **esecuzione** se detiene la CPU
- Se ad un certo punto vuole stampare, ma la stampante è occupata, il processo viene **bloccato** e messo in coda.
- Se è in attesa che lo scheduli gli di La CPU

Si dice che è PRONTO.

Quando passa da stato di esecuzione a stato di pronto lo un interrupt

Un programma è l'algoritmo che viene compilato,

Quando si esegue un programma diventa un processo

2) Si illustra una classificazione delle modalità di designazione e sincronizzazione delle primitive di interazione tra processi nel modello ad ambiente locale \rightarrow send/receive (P.50)

Nel modello a scambio di messaggi ogni processo opera in un ambiente locale che è protetto.

Le interazioni avvengono tramite scambio di messaggi e tutte le risorse sono private, accessibili soltanto indirettamente.

STRUTTURA DEL MESSAGGIO

Type messaggio: ... ,

origini: ... ;

destinazione: ... ;

contenuto: ... ;

end

Il messaggio può essere anche privo di contenuto, ovvero quando due processi devono sincronizzarsi e fissa con un segnale che è costituito dall'arrivo del messaggio.

Le primitive usate nello scambio di messaggi sono due:

send (m) che inserisce il messaggio nella coda del destinatario e

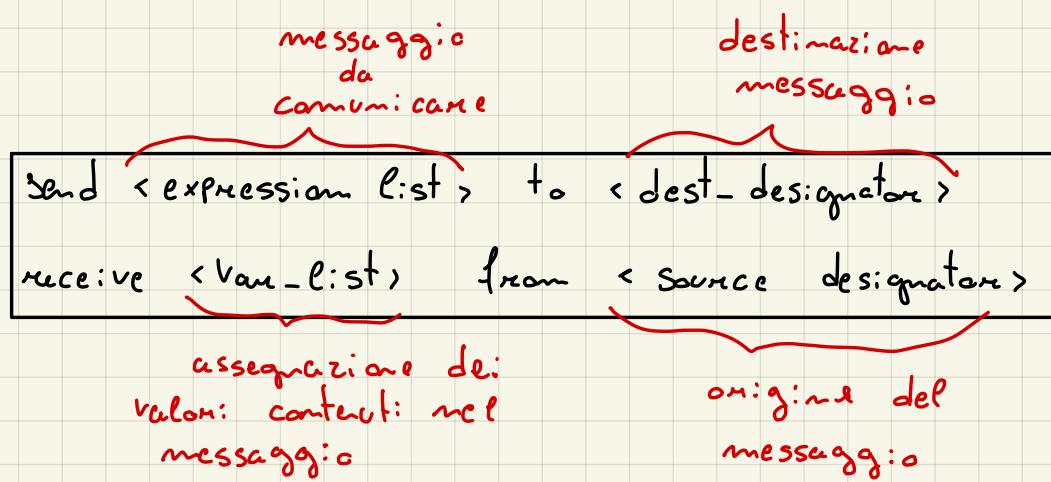
receive (m) che preleva il messaggio dalla coda del processo corrente.

I costruttori linguistici che realizzano lo scambio di messaggi:

- ↳ differenziano in base al modo in cui vengono **designati**:
- i processi **targete e destinatario** e in base al tipo di **Sincronizzazione**

La **designazione** può essere diretta o indiretta.

Nel caso **diretta** send e receive assumono questa forma:



La designazione diretta è detta **simmetrica**. Perché i processi si nominano esplicitamente e simmetricamente, stabilendo un canale di comunicazione individuato dalla coppia (**dest- e source-designator**).

Questa tipologia di scambio di messagg: è semplice da fare e usare.

La designazione diretta può essere anche **asimmetrica**.

La comunicazione non avviene da un processo all'altro, ma da molti ad uno.

Il mittente esplicita il nome del destinatario ma questi non esprime con quale processo vuole comunicare.

Per la comunicazione da molti a molti si usa la designazione **indiretta**.

Essa prevede l'esistenza di un mailbox al quale si possono mandare messaggi e prelevarne (ricopra il ruolo di dest. e source-generatore).

Il messaggio può essere prelevato da un qualunque processo che prevede nella propria receive il nome di quella mailbox.

I processi possono selezionare la tipologia di messaggi da ricevere con operazioni di receive solo su certe mailbox.

I client mandano i messaggi alla mail box alle quali sono associati i servizi, mentre i server prelevano i messaggi solo per i servizi che competono a loro.

L'implementazione di una mailbox in ambiente distribuito presenta dei problemi legati alla realizzazione, in quanto il server che riceve bisogna che sia in grado di soddisfare le richieste e inoltre appena un processo effettua la receive il messaggio deve essere reso indisponibile agli altri server.

Da qui si ha la realizzazione della **designazione globale tramite porta**.

Le porte sono delle mbox il cui nome può comporre solo in un processo come source-designatore in una receive, ovvero solo un processo può accedere in receive alla porta.

In questo modo tutte le receive che indicano una porta compaiono in un solo processo.

Abbiamo risolto il problema da molti ad uno ma non quello da molti a molti.

Ogni porta è associata ad un tipo di richiesta, quindi un processo decide quali tipi di richieste accettare utilizzando le porte associate ad esse.

Riassunto

Il direct messaging simmetrico è utilizzato per uno-uno quello asimmetrico per multi-uno e l'indiretto con inbox per multi-molti e quello indiretto con le porte per molti-uno.

L'indirect messaging è il caso più grande, ma anche il più difficile da realizzare, gli altri sono più facili da implementare ma limitano i tipi di interazione direttamente programmabili:

Parlano ora della **Sincronizzazione**

Se send può essere sincrona, asincrona e di tipo RPC.

Nel caso **Sincrona** il mittente si blocca e rimane in attesa fin quando il messaggio non viene ricevuto.

Un messaggio ricevuto contiene informazioni sullo stato attuale del processo ammesso perché questo non va avanti con l'esecuzione fino alla ricezione.

Trasferimento dati solo quando entrambi i processi sono pronti a cominciare → **punto di Sincronizzazione**

Siccome il trasferimento avviene diretto da uno all'altro non c'è bisogno di interporne tra i disposit. v. d: memoria.

Nel caso asincrono il mittente continua l'esecuzione dopo aver inviato il msg.

Informazion: del msg non legate allo stato attuale del processo perché la continua nell'esecuzione.

Per realizzare send asincrona serve una coda di ingresso ad ogni processo nel caso direct naming, oppure una coda in ingresso ad ogni mbox nel caso di indirect naming.

In teoria lo spazio per contenere il msg (buffer) dovrebbe essere illimitato; si arriva il problema segnalando al mittente quando il buffer è pieno.

Nel caso RPC il mittente rimane in attesa finché il destinatario ha terminato di svolgere la procedura richiesta.

Nel caso d: direct naming la procedura identif. ca un processo, nel caso d: indirect naming identif. ca un servizio.

Anche la receive può essere sincrona o asincrona.

Nel caso d: sincrona la receive è bloccante se non ci sono messagg: sul canale d: comunicazione.

Costituisce un punto d: sincronizzazione per il processo ricevente.

Un problema è che in certe applicazioni si vuole ricevere solo alcun: msg ritardano elaborazione d: altri:

Per questo c'è la receive asincrona con interazioni dello stato del canale.

Per ogni processo specifico più canali di input, ciascuno dedicato a tipologie diverse di messaggi.

Per fare in modo che il processo decida quale messaggio ricevere specificando su quali canali attendere, per questo usiamo una primitiva che verifica lo stato del canale e comunica se c'è un messaggio o se il canale è vuoto. La verifica avviene con un criterio di polling e la primitiva risulta non bloccante.

Il problema qua è invece l'attesa attiva che si fa quando si attendono i messaggi specifici dai canali.

3) Concetto di busy waiting (P.54)

Considero un modello di interazione tra CPU e dispositivo d'I/O: i due blocchi comunicano, cioè si scambiano dati attraverso il processore del dispositivo in cui c'è un Buffer. Per i comandi inviati dalla CPU, un bit è un flag busy e un bit o flag ready.

La CPU vuole ad esempio che la periferica stampi: mette il segnale nel Buffer dei comandi e setta ad 1 la F.B.
⇒ avvisa la stampante che deve andare in funzione.

Il processore del dispositivo se non sta eseguendo alcun comando ispeziona continuamente la F.B., oppure trova il valore 1 lo accetta e va a leggere nel Buffer l'operazione che deve svolgere. Terminata l'esecuzione la stampante mette a 1 la F.R., avvertendo la CPU che ha svolto la richiesta. La CPU, che ispeziona di continuo la F.R., si accorgere di ciò e lo accetta, inserendo poi un nuovo comando nel buffer.

Questo sistema è insufficiente poiché finché la F.R. non va a 1 la CPU perde tempo a controllarne e non elabora alcun dato.

È qui che abbiamo la busy wait: la CPU compie operazioni di efficienza zero.

Essendo le fasi d'I/O: IOC (input-output control). IOC controlla tutti i dispositivi, quindi sa quando sono pronti o occupati: e quindi riesce a sovrapporre le fasi al meglio.

l'OC manda i comandi ai dispositivi pronti di continuo, e quando è insufficiente solo quando sono tutti occupati, che è raro.

Per migliorare ulteriormente si può sovrapporre le fasi di I/O con quelle di elaborazione. Si realizza facendo eseguire alla CPU altri programmi mentre è in attesa per il completamento dell'esecuzione di comandi di I/O.

Se si pensa che non c'è nessuna operazione di I/O la l'OC gira a vuoto (**busy waiting**).

Qui entra in gioco il SO che passa il controllo della CPU ai programmi che devono elaborare dati.

Quindi la CPU ha 2 ruoli: elaborazione e supporto della l'OC.

Per ottenerne questo introduciamo **C'è interrupt** utilizzata tramite HW.

C'sono 2 tipi: da dispositivi e da timer.

Quella da dispositivo consiste in un segnale HW che viene inviato da un dispositivo di I/O alla CPU per segnalare che un comando è stato eseguito e quindi il dispositivo è pronto a ricevere il comando successivo.

Per cui abbiamo la CPU contenente un oggetto (vetture di bit) IRR dove ogni bit memorizza l'interrupt di un particolare dispositivo: se sono tutti a 0 allora nessun dispositivo ha fatto richiesta di essere eseguito e la CPU continua ad elaborare, se c'è almeno un bit a 1 la CPU si ferma e passa il controllo alla l'OC che va a servire la periferica.

L'interrupt routine salva lo stato del programma interrotto, azzera quel bit, chiama il programma di controllo del dispositivo, ripristina lo stato e fa ripartire il programma interrotto.

L'interrupt da timer è un unico interrupt (invece di uno per ogni dispositivo) generato periodicamente da un timer H/w con frequenza programmabile.

Viene attivato l'interrupt con cadenza regolabile e si controlla se qualche dispositivo vuole essere servito:

In tal caso si passa il controllo della CPU al programma che gestisce il dispositivo, altrimenti la CPU continua ad elaborare.

La procedura è questa: salvo lo stato del programma interrotto, controllo tutti i dispositivi =>

Suppongo sia una CR e un LP => se CR è in stato pronto chiama il programma di controllo di CR, se LP è pronto e LS > 0 (è il contatore di linea da stampare) chiama il programma di controllo di LP.

Infine ripristina lo stato e continua il programma interrotto.

Questo sistema è poco efficiente

4) Descrivere algoritmi di scheduling della CPU e confrontare i tempi medi di attesa considerando i processi P_1, \dots, P_4 . Per gli algoritmi: FCFS, SJF e con priorità (in ordine decrescente) (PS7)

Processo	tempo di Burst	priorità
P_1	9	3
P_2	12	1
P_3	4	2
P_4	7	4

Se si deve decidere come distribuire le risorse tra i vari processi; le più importanti sono la CPU e la memoria centrale.

Lo scheduler decide a quale dei processi prenti assegna il controllo della CPU.

Abbiamo cioè una coda di processi in stato di pronto che possono passare in stato di esecuzione qualora gli venga assegnata la CPU.

La scelta del processo avviene tramite un algoritmo di scheduling.

Il primo algoritmo che consideriamo è il F.C.F.S.

La CPU viene assegnata al processo che l'ha richiesta per prima (le code vengono gestite in modo FIFO).

Le prestazioni sono scadenti in termini di tempo medio d'attesa, ma è semplice da realizzare. Per esempio avremo che il tempo medio di attesa è:

$$\frac{[0+7+(7+4)+(7+4+12)]}{4} = 10,25$$

L'algoritmo S.J.F. assegna la CPU, quando si libera, al processo col tempo di burst più breve. Per fare in modo che lo scheduler decide correttamente. Ad ogni processo viene assegnata la lunghezza del successivo burst di CPU.

La difficoltà sta proprio qua, quindi distinguiamo i modelli analitici, che sfruttano le informazioni che si hanno sui precedenti burst del processo:

$$T_{n+1} = \alpha T_n + (1-\alpha) T_m \quad 0 < \alpha < 1$$

T_{n+1} = stima lung. burst $n+1$ -esimo

T_m = lung. burst n -esimo

Oppure supponiamo che $T_{n+1} \approx t_n$

nel nostro esempio il tempo medio di attesa sarà:

$$\frac{0 + 4 + (4 + 7) + (4 + 7 + 9)}{4} = 8,75$$

L'algoritmo di priorità consiste nel realizzare code di processi pronti a seconda della priorità.

Lo scheduler seleziona il processo al livello nuovo di priorità che contiene processi pronti.

In presenza di preemptive è in esecuzione un processo di priorità nuova in ogni istante.

Ciò vuol dire che un processo può lasciare forzatamente la CPU, quindi possiamo avere alcuni programmi favoriti su altri.

→ **starvation**, cioè i processi a bassa priorità possono rimanere bloccati.

Per risolverlo si aumenta gradualmente la priorità di questi processi distinguendo quindi priorità **dinamica** e **statica**.

nel nostro esempio il tempo di attesa è:

$$\frac{0 + 12 + (12 + 4) + (12 + 4 + 9)}{4} = 13,25$$

5) Concetto di starvation (P. 48)

In un sist. di calcolo multiprogrammato in memoria principale sono presenti più processi, anche in stati diversi.

disponendo la macchina di risorse limitate questi processi sono tra loro in concorrenza, soprattutto per la CPU.

Da **starvation** avviene quando un processo o più, non vengono eseguiti rimanendo in stato di pronto o occupato.

Si può avere quando lo scheduler utilizza un **algoritmo di priorità**. Un altro esempio: si vogliono risolvere problemi che derivano dall'accesso, in ordine errato, dei **processi a dati condivisi**.

La parte di programma che prevede questo è detta **sezione critica**.

Quando 2 o più processi interfiscano sulla stessa risorsa si fa invece una **corsa critica**.

Due sezioni critiche appartenenti alla stessa classe se operano sugli stessi dati.

Se una prevede operazioni di **scrittura** devono essere eseguite naturalmente nel tempo, se solo **lettura** nella stessa area di memoria non c'è problema.

Per risolvere il problema delle corse critiche si corre alla **mutua esclusione**, cioè una sola sezione critica per classe può essere in esecuzione ad ogni istante.

Una soluzione di mutua esclusione è il **test and set lock** che avviene tramite HW.

6) Sistema di protezione di un SO (P.61)

In presenza di più processi in esecuzione fa nascere problemi di interferenza, ad esempio un processo può tentare di modificare il programma o i dati di un altro processo o di puntare del SO. Il S.O. fornisce perciò una **protezione**: politiche e meccanismi (caso e come) per controllare l'accesso di processi alle risorse del sistema di elaborazione, cioè gestire gli accessi alla stessa periferica o alla stessa localizzazione di memoria.

Bisogna ricordare che i processi devono anche comunicare fra loro, quindi il S.O. deve **proteggere** ma anche permettere **l'interazione**.

Un metodo è la condivisione dello spazio di memoria.

Il HW deve rilevare gli errori: come op code illegali o riferimenti in memoria non consentiti.

Gli errori vengono segnalati e affidati alla gestione del S.O. tramite il meccanismo delle **trap**.

In tal caso il S.O. termina il processo, segnala la terminazione anomala (il processo non è arrivato alla fine) ed effettua un **dump** della memoria, cioè tutta la memoria viene salvata sul disco.

Questo file dump contiene lo stato del processore; si può quindi effettuare il dialogo del programma e capire il perché dell'errore.

Questo pone via molto tempo e memoria.

Per ogni utente viene definito un **dominio di protezione** di modo che ogni processo lanciato da questo operi in tale dominio.

Essa specifica le risorse al quale il programma può accedere e le operazioni consentite.

Tale dominio è costituito da accessi di diritto, cioè coppie in cui sono specificate risorse e diritti che il processo ha su di essa, ovvero le operazioni che può compiere.

Due tipi di operazioni: comuni e privilegiate per questo la CPU deve avere più modi di funzionamento, supervisor mode e user mode.

In supervisor mode la CPU può eseguire qualsiasi operazione, nell'altro caso no.

Il passaggio da user e supervisor avviene tramite interrupt che può essere esterno (asincrono) o interno (sincrono). La commutazione inversa avviene tramite un'istruzione speciale di cambiamento di modo eseguita dal SO prima della cessione del controllo ad un processo di utente.

Le istruzioni privilegiate eseguibili solo in supervisor sono:

- I/O

- evitare che un programma modifichi della propria area di azione.
- manipolazione del sistema di istruzione.
- Cambiamento di modo
- Halt

Lo scambio di informazioni tra S.O. e programma avviene tramite system call.

L'alternanza user/supervisor avviene all'interno del S.O. ed è invisibile al programma.

Il passaggio di eventuali parametri avviene tramite registri: per indirizzarli.

Le categorie principali di system call sono:

- controllo: processi e lavori.
- manipolazione di file e dispositivi.
- gestione informazioni.
- comunicazioni.

7) deadlock (P. 66)

Il deadlock avviene quando 2 processi detengono ciascuno una risorsa che serve all'altro per proseguire. Se la risorsa non è disponibile il processo viene posto in attesa e così anche l'altro rimanendo bloccato.

Se condizioni affinché si verifichi deadlock sono:

- 1)- **mutua esclusione**: ogni risorsa può trovarsi in due stati, o è assegnata ad un solo processo oppure è disponibile
- 2)- **prende e aspetta**: i processi che detengono già delle risorse possono richiederne altre
- 3)- **assenza di preemptive**: risorse già assegnate non possono essere revocate forzatamente.
- 4)- **attesa circolare**: ogni processo è in attesa di una risorsa occupata da quello che lo segue nelle liste, l'ultimo è in attesa del primo.

Per evitare che si verifichi il deadlock si può:

- 1)- **ignorare il problema**: meglio accettarne se occasionalmente piuttosto che uscire una risorsa alla volta
- 2)- **riservarlo e risolverlo**: per ad esempio il S.O controlla il grado di allocazione delle risorse, se è presente deadlock si fermano uno o più processi fino a terminare il problema.
- 3)- **Prevenzione dinamica**: mediante algoritmo del bancheere questo richiede di conoscere il max numero di risorse richieste da ciascun processo e presupporre che tale processo

Posso mantenerle tutte durante l'esecuzione.

L'algoritmo identifica le situazioni rischiose e mega l'assegnazione delle risorse disponibili.

- 4)- Prevenzione strutturale: evita che si verifichi una situazione di attesa circolare imponendo un ordine sulla richiesta delle risorse.
Richiede la collaborazione attiva dei programmatori.

8) tecnica di spooling dell'I/O (P. 63)

Nasce dalla necessità d: massimizzare l'efficienza d: un esecutore nel sistema batch.

Se l'elaboratore resta in attesa durante le fasi d: I/O che sono lunghe, e ciò non va bene.

da qui la divisione in tre blocchi del calcolatore:

dispositivo d: I/O, elaboratore e periferica d: O, ognuno con la propria capacità elaborativa (non è più un blocco unico)

Una volta che A ha terminato la fase d: input e va in elaborazione, B può subito iniziare la sua fase d: I/O.

Per accrescere la velocità d: esecuzione si ricorre allo spooling, ovvero la sovrapposizione d: fasi d: I/O mediante l'utilizzo delle memorie come un buffer d: grandi dimensioni.

Se ad esempio A e B vogliono utilizzare due periferiche d: O diverse, possono farlo simultaneamente e i risultati vengono poi memorizzati nella memoria.

I due programmi d: spool-in e spool-out s: occupano del trasferimento de: dati da: dispositivi d: i alla memoria e dalla memoria ai dispositivi d: O.

3) Si descrivano le varianti preemptive e non-p. d: alcuni algoritmi: d: scheduling

Preemptive: Prevede che la CPU possa essere tolta ad un processo forzatamente, in quanto la miassegnazione può avvenire anche per eventi del tipo 3-4

Non-preemptive: La miassegnazione della CPU avviene per eventi 1 e 2 è quando un processo in esecuzione prosegue fino al rilascio spontaneo della CPU.

La miassegnazione della CPU può avvenire a seguito di questi evet:

- 1) Un processo commuta dallo stato di esecuzione a sospeso.
- 2) " " " " " esecuzione a pronto
- 3) Un processo in esecuzione termina.
- 4) " " " " " sospeso a pronto.

Pur quanto riguarda gli algoritmi d: scheduling; FCFS, SJF e di priorità sono di tipo non-preemptive, cioè quando la CPU viene assegnato ad un processo, questo ne detiene il controllo fino quando non finisce.

Gli algoritmi SJF e di priorità possono essere anche di tipo preemptive, quando durante l'esecuzione d: un processo, uno nuovo entra in coda.

Quello nuovo può richiedere un tempo d: CPU inferiore o avere maggiore priorità d: quello in esecuzione.

10) Concetto di multi-programmazione, vantaggi in termini di efficienza rispetto alla mono programmazione.

Nella multi-programmazione sono presenti più programmi in memoria. Quando uno attende il completamento di una fase di I/O, il controllo della CPU viene assegnato ad un altro processo. L'insieme dei programmi in memoria viene scelta in modo da ottenere la max occupazione della CPU e della memoria. In questo modo vengono ridotti i tempi morti dove la CPU è inattiva.
» efficienza \Leftrightarrow » complessità

Pur massimizzando il throughput utilizza la gestione batch dei job, cioè gestisce i programmi da svolgere in coda.

Nella monoprogrammazione abbiamo i programmi ordinati in liste, appena A finisce inizia B $\Rightarrow \text{thr} = 1$

Nella multi-programmaz. il S.O sovrappone i programmi correttamente (che non devono utilizzare la stessa risorsa). Se un programma ha finito con una risorsa, questa va subito ad un altro che la richiede.

Il time-sharing permette all'elaboratore di servire, nello stesso intervallo di tempo, più utenti, dedicando a ciascuno tutte le risorse per quanti di tempo fissati. Questo migliora i temp. di risposta, ma peggiora l'utilizzo delle risorse.

11) Strumento di sincronizzazione dei semafori e illustrare mediante codice il problema dell'interazione tra produttore e consumatore (P.GG)

Il semaforo è una variabile non negativa ($S \geq 0$)

con valore iniziale $S = S_0$.

ad esso è assegnata una lista d'attesa Q_S che contiene informazioni sui processi in attesa.

Ci sono due primitive: `wait(S)` e `signal(S)`.

`wait` valuta se la sezione critica è già occupata, e se lo è il processo viene messo in coda altrimenti viene eseguita la sua SC.

`Signal` serve per segnalare che la SC è libera e vi si può accedere.

La `wait` può essere bloccante ($S=0$): il processo viene sospeso e le sue inf. messe in coda \Rightarrow un altro processo è in SC; oppure può essere passante ($S>0$) e il processo può eseguire la SC.

La `signal` è sempre passante: se c'è un processo in coda le sue inf. vengono cancellate da Q_S e il suo stato modificato in "pronto", cioè quando lo scheduler gli assegna la CPU il processo inizia la SC.

La riattività dei processi in sospesa avviene tramite politica FIFO, la SC viene occupata dal primo che ne ha fatto richiesta.

non ci sono fecciai d: starvation.

Viene risoltò il problema d: busy waiting: se un processo cerca d: accedere ad una SC che è occupata non cicla sempre tra le stesse istruzioni: occupando la CPU, ma viene bloccato così che non consumi risorse.

wait e signal sono come SC brevi perché operano su dati condivisi (ad esempio S), per questo deve essere garantita la loro individabilità, ottenute con `lock()` e `unlock()`.

Una delle applicazioni dei semafori è nel problema del produttore e del consumatore: P e C si scambiano msg accedendo ad un buffer, il 1° in scrittura e il 2° in lettura: il buffer può contenere n messaggi.

P deve comunicare a C se ci sono msg nel buffer, mentre C deve comunicare a P se c'è spazio per inserire altri.

Questa lo si realizza mediante l'uso di due semafori: uno indica che c'è un msg disponibile e l'altro indica lo spazio che rimane nel buffer.

- il produttore non può inserire il msg se il buffer è pieno.
- il consumatore non può prelevare msg se il buffer è vuoto.