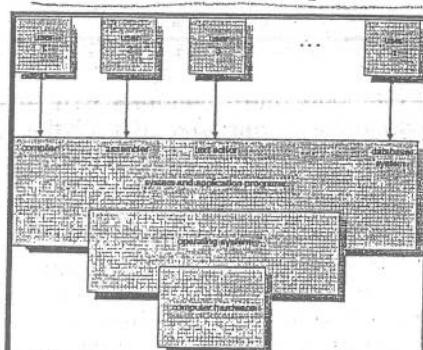


RIASSUNTI Sistemi Operativi

Lezione 1.

Un sistema di elaborazione può essere visto come l'insieme di:
hardware, sistema operativo, programmi applicativi, utenti.



L'hardware è il complesso delle unità fisiche che compongono la macchina; il sistema operativo è un software (costituito da più programmi) che gestisce e controlla le risorse del sistema. I programmi applicativi sono tutti quei software che vengono utilizzati dagli utenti per i loro scopi: ci sono programmi di sistema, quali il compilatore, l'assembler e l'editor di testo, e programmi applicativi (CAD, database, giochi ecc.). Ovviamente perché il tutto funzioni c'è bisogno degli utenti, che

devono inserire i dati da elaborare (ed interagire in genere con la macchina). Attenzione perché si parla di utenti non di un solo utente: la necessità di gestire più operatori sulla stessa macchina genera molti problemi, soprattutto di ottimizzazione.

Un sistema di calcolo può essere visto come un insieme di risorse Hw e Sw utilizzate per lo sviluppo e l'esecuzione dei programmi utente. Tali risorse devono essere gestite in modo opportuno: di questo si occupa il Sistema Operativo. Per gestione si intendono molte cose: l'utilizzo delle risorse secondo un determinato ordine (ottimizzazione dei tempi di esecuzione), rendere tali risorse disponibili a più utenti (quando questi lo richiedono), proteggerle contro accessi non autorizzati, sia voluti (un utente vuole accedere ai dati di un altro utente) sia non voluti (es.: un programma per errore cerca di scrivere in una parte di memoria dove ci sono già dei dati). Un altro aspetto della gestione è l'organizzazione delle risorse, che deve essere mirata a garantire la sopravvivenza del sistema in caso di guasti (es.: se viene interrotta l'alimentazione il sistema deve essere in grado di ripartire, oltre a non perdere i dati che si stavano elaborando in quel momento).

Uno dei compiti più importanti del SO è la gestione delle risorse volta a rendere semplice ed efficiente il loro utilizzo: per efficienza si intende in completo utilizzo di tutte le risorse, mentre la semplificazione si ottiene fornendo agli utenti un'interfaccia grafica (o macchina virtuale) più agevole da utilizzare della macchina "nuda".

Il SO può essere visto come un allocatore di risorse Hw e Sw. Esso si occupa ad esempio di fornire agli utenti (cioè ai programmi che essi utilizzano) l'uso della CPU (in genere è una sola), quindi di frazionare tra di essi il tempo di CPU, facendo girare alternativamente i programmi per un determinato intervallo di tempo; il SO deve anche attribuire ad ogni programma una parte della memoria, decidere a chi vada assegnato l'utilizzo dei dispositivi di I/O e secondo quali politiche (priorità, ordine di richiesta, ecc.). Il SO può essere visto anche come un programma

di controllo: controlla l'esecuzione dei programmi per prevenire errori ed usi impropri del calcolatore (in particolare per i dispositivi di I/O).

In sintesi gli obiettivi del SO sono due: rendere più semplice l'utilizzo di un sistema di calcolo e rendere più efficiente l'uso delle risorse dello stesso.

Il SO è costituito dall'insieme dei programmi che rendono praticamente utilizzabile l'elaboratore agli utenti cercando contemporaneamente di ottimizzarne le prestazioni. Ci sono due modi di osservare il SO: dall'alto verso il basso (top-down), ovvero considerare il SO come una macchina estesa: in questo caso il SO fornisce astrazione, perché non si vede quello che effettivamente la macchia fa dentro; oppure dal basso verso l'alto (bottom-up): il SO è visto come un gestore di risorse che fornisce anche protezione e risoluzione di conflitti.

Le risorse Hw di un sistema di elaborazione sono i processori (in genere uno solo), le memorie (memoria centrale, memoria cache e memoria secondaria), i canali di comunicazione (bus) e i dispositivi di I/O (stampanti, tastiere, monitor ecc.). Gestire queste risorse significa tenere traccia di esse, cioè sapere ad esempio quanto spazio è rimasto in memoria oppure quali sono i dispositivi di I/O in utilizzo. La gestione implica anche adottare strategie di assegnazione, allocare le risorse, recuperare le risorse inutilizzate e rilevarne eventuali usi impropri.

Le funzioni specifiche che un SO svolge sono:

- la gestione della memoria principale: il SO carica in questa memoria da quella secondaria dati e programmi, in modo che questi possano essere trasferiti alla CPU più velocemente; siccome la prima operazione è relativamente lenta il SO ~~si~~ deve cercare di minimizzare i trasferimenti dalla memoria secondaria a quella principale. Inoltre bisogna evitare che i programmi interferiscano tra di loro invadendo lo spazio assegnato ad altri programmi ed anche assegnare la memoria in base a criteri di efficienza
- la gestione dei processori: il SO deve decidere quale programma utilizzerà il processore, cioè pianificare (scheduling) con che ordine i programmi si alterneranno nell'uso della CPU, seguendo criteri di corretto funzionamento e di efficienza. A questo compito si aggiunge quello di verificare che i programmi rilascino il processore entro il tempo stabilito.
- la gestione di dispositivi periferici: il SO si occupa di mascherare la complessità delle operazioni di I/O, cioè l'utente non deve occuparsi ad esempio dei movimenti interni della stampante, deve soltanto dare l'ordine di stampa; altre funzioni sono: controllare il corretto funzionamento delle operazioni (es.: verificare se la stampante ha stampato), risolvere conflitti nell'utilizzo di una stessa periferica da parte di più programmi (gestire quindi le code di utilizzo), consentire il massimo sfruttamento delle periferiche (es.: se un programma termina di stampare il successivo occupa subito la stampante in un'altra operazione).

- la gestione della memoria secondaria: il SO consente l'accesso all'informazione in base alla sua organizzazione logica, che può essere un insieme di file, cartelle, sottocartelle ecc. (File System) anziché fisica, cioè il modo in cui sono effettivamente suddivisi i dati (dischi, tracce, settori). Altro compito è quello di controllare i diritti di accesso ai file da parte degli utenti e, se permesso, consentirne la manipolazione.

Le aree di applicazione di un SO sono due: SO utilizzati per sistemi di tipo generale e SO impiegati per la gestione di sistemi in tempo reale. Ovviamente a seconda dell'area di applicazione il SO si comporta in un determinato modo. In particolare nel primo caso, facendo riferimento alla gestione della CPU, il SO alterna l'utilizzo del processore tra i vari programmi che sono in esecuzione, assegnando ad ognuno di questi un "quanto" di tempo (rimpallo di processi). Ipotizziamo per esempio che ci siano in esecuzione i programmi A e B e in un istante di tempo sia A ad avere a disposizione la CPU: se B avesse bisogno utilizzare la CPU in quell'istante non potrebbe, perché c'è A in esecuzione; quindi B deve aspettare che termini il quanto di tempo per poi svolgere le sue operazioni. In un sistema in tempo reale questo tempo di attesa a volte può influire molto sul corretto funzionamento del sistema stesso (si pensi ad un sistema di controllo di un processo ad esempio). In questo caso, il SO consente al programma B di interrompere l'esecuzione di A e di avere immediatamente (o quasi) a disposizione la CPU. Tutto questo non è valido solo per il processore ma anche per le altre risorse gestite dal SO. Un SO generale si può avvicinare al comportamento del SO in tempo reale se si riduce il tempo di utilizzo della CPU (quanto di tempo).

Gli utenti del SO si dividono in:

- utenti finali: per loro il SO è trasparente e si limitano ad utilizzare i programmi applicativi;
- programmatori applicativi: utilizzano i servizi del SO per la realizzazione e l'esecuzione dei loro programmi;
- programmatori di sistema: aggiornano e modificano i programmi del SO per adeguarli a nuove necessità del sistema o degli utenti applicativi;
- operatori: controllano il funzionamento e rispondono alle richieste di intervento da parte del sistema;
- amministratore del sistema: stabilisce le politiche di gestione del sistema e ne cura l'osservanza.

Come già detto il SO gestisce le risorse Hw (e Sw) del sistema di elaborazione. L'ideale per un programmatore di SO sarebbe quello di conoscere a fondo la struttura Hw del sistema di calcolo, in modo da sfruttare in modo ottimale le risorse di ogni tipo di macchina. I SO realizzati secondo questi criteri sono detti proprietari, ed in genere vengono realizzati dalla stessa casa costruttrice dell'Hw. E' scontato che un SO di questo tipo funzioni benissimo sulla macchina per cui è stato progettato, ma presenta una scarsa efficienza se viene cambiato l'Hw: non è flessibile. Inoltre i sistemi di calcolo che utilizzano questo tipo di SO hanno interfacce grafiche diverse a seconda della casa

che li ha progettati. I SO standard, invece, vengono realizzati per potersi adattare sulla gran parte delle macchine; avendo una maggiore flessibilità però perdono in efficienza se le loro prestazioni vengono confrontate con quelle dei SO proprietari. Questa differenza di prestazioni è stata ridotta nel tempo grazie all'introduzione dei DRIVER: software allegati ad ogni periferica Hw che permettono a queste di funzionare in modo più efficiente, visto che il Sw è creato dalla stessa casa costruttrice che ha realizzato l'Hw. Altre differenza fondamentale tra SO proprietari e SO standard è che l'interfaccia grafica nel primo caso cambia tra le diverse case costruttrici, nel secondo invece, siccome il SO si adatta a vari tipi di macchine, l'interfaccia utente rimane costante, il che rende virtualmente uguali due macchine che sotto il profilo Hw sono diverse.

Lezione 2.

I sistemi di elaborazione hanno subito nel corso del tempo un processo evolutivo contraddistinto da vari stadi. Questa evoluzione ha riguardato sia la parte Hw (le unità fisiche), che quella Sw (il SO). Facendo riferimento alla parte Hw si possono identificare quattro fasi: sistemi isolati, sistemi centralizzati, sistemi decentrati (minielaboratori) e sistemi distribuiti.

- ①. **Sistemi isolati:** sono detti anche Stand Alone. Sono costituiti da un'unica unità di calcolo in grado di svolgere una sola operazione alla volta. L'elaborazione è di tipo batch, ovvero a lotti: viene redatta una lista di istruzioni o programmi da eseguire che, una volta lanciata, viene svolta dall'elaboratore; questo, terminate le istruzioni, comunica i risultati. Non c'è quindi alcuna comunicazione diretta tra l'utente e la macchina, le interazioni si limitano allo scambio istruzioni/risultati.
- ②. **Sistemi centralizzati:** in questa fase gli elaboratori assumono dimensioni maggiori rispetto al caso precedente poiché è richiesta maggiore potenza di calcolo. Il sistema è costituito da un unico elaboratore centrale cui sono collegati più utenti attraverso dei terminali passivi (accesso remoto tramite rete telefonica), cioè dei dispositivi che consentono di interagire con il computer centrale ma che non effettuano nessun altro tipo di operazione (fanno solo da tramite). La necessità di gestire più di un operatore porta alla nascita delle tecniche di time-sharing: viene fornita ad ogni utente una fetta di tempo durante il quale può servirsi della macchina, cioè inviare i dati da elaborare e ricevere i risultati dal sistema attraverso il terminale. Utilizzando questo tipo di sistema si può distribuire il lavoro tra più utenti che interagiscono con l'elaboratore da lontano.
- ③. **Sistemi decentrati:** sono costituiti da un elaboratore centrale e più minielaboratori collegati ad esso. Rispetto a prima i terminali hanno una loro potenza di calcolo, il che permette di ottenere tempi di elaborazione minori relativamente al caso dei sistemi centralizzati. La decentralizzazione consente di scegliere i calcolatori in base allo scopo che devono svolgere. Ad esempio in una industria che adotti questo sistema il calcolatore che controlla il processo produttivo avrà caratteristiche diverse da quello che opera nel magazzino. Ogni elaboratore è quindi autonomo e ognuno di essi gestisce i dati inerenti alla sua area di servizio. Un difetto di questo sistema è che ha bisogno di maggiore manutenzione rispetto al precedente, perché si passa da un solo elaboratore a più di uno.
- ④. **Sistemi distribuiti:** insieme di unità dotate di capacità operativa autonoma ed al tempo stesso in grado di scambiare mutuamente dati e risorse attraverso una rete di comunicazione. In alcune applicazioni di questo tipo di sistema i componenti sono specializzati in determinate operazioni in modo tale che un utente che vuole svolgere una certa operazione utilizzi l'elaboratore specializzato. Con il sistema distribuito si

possono modellizzare molti sistemi in base al tipo di distribuzione: distribuzione funzionale, è il caso dei calcolatori in cui le funzioni sono suddivise; distribuzione locale, cioè le reti in cui le funzioni sono distribuite localmente; distribuzione geografica, utilizzata nelle reti come Internet. Il sistema distribuito può essere schematizzato come un insieme di nodi collegati tra loro ognuno dei quali presenta delle capacità, che sono poi le proprietà distribuite: elaborazione (ognuno ha il suo processore), memorizzazione (ognuno ha la sua memoria), comunicazione (ognuno ha la sua scheda di rete).

I vantaggi di un sistema distribuito sono svariati. Tra questi vi è la tolleranza al guasto: il verificarsi di un guasto, ad esempio un terminale che non funziona, non blocca il sistema ma ne riduce soltanto le prestazioni. Essendoci poi la possibilità di condividere i dati si possono creare varie copie dei database, dislocati in luoghi diversi, in modo da tutelarsi da evenienze catastrofiche. Un altro vantaggio sono le prestazioni: la distribuzione dell'intelligenza ai vari terminali provoca un abbattimento dei tempi di elaborazione rispetto ai sistemi centralizzati, perché l'elaborazione viene effettuata sul luogo, evitando il trasferimento dei dati. Infine la capacità di condivisione dei dati e delle risorse permette di suddividere l'elaborazione tra più utenti.

I SO si sono evoluti insieme ai modelli Hw dei sistemi di elaborazione. Le prime macchine (sistemi stand alone) non avevano alcun SO. In questi casi il programmatore è anche operatore interattivo (la persona addetta a mantenere la macchina) ed ha la visione diretta della macchina e della disponibilità di tutte le risorse. L'accesso da parte di più utenti è ottenuto mediante meccanismi di prenotazione, che portano però a dei problemi poiché è il sistema risulta troppo rigido: nel caso un utente non utilizzi la macchina questa rimane inattiva, fatto che abbassa l'efficienza.

La prima generazione dei SO introduce la separazione del programmatore dalla macchina tramite l'operatore; in questo modo non è più il programmatore che deve svolgere anche le funzioni di mantenimento della macchina, al contrario di quanto accade nel caso precedente. Uno dei problemi che il programmatore deve affrontare è il set-up dei job, cioè con che ordine mettere in esecuzione i programmi in modo da sfruttare al massimo le potenzialità della macchina e ridurre i tempi di esecuzione. La riduzione dei tempi di set-up si ottiene con tre accorgimenti: gestione dei job di tipo batch, ovvero organizzazione dei programmi da svolgere in code; gestione delle periferiche con tecniche di spooling, che permettono di rendere parallele le operazioni di I/O; automatic job sequencing, tecnica che consiste nell'automaticizzare l'allineamento dei programmi in modo da massimizzare le prestazioni e quindi ridurre i tempi. Quest'ultima operazione è il primo compito che un SO è chiamato a svolgere: nasce il SO come stratificazione successiva di funzioni volte ad aumentare l'efficienza e la semplicità d'uso della macchina.

La seconda generazione di SO introduce principalmente due novità.
La prima è l'indipendenza tra i programmi ed i dispositivi di I/O usati: il programmatore non si preoccupa di come è fatto il dispositivo, lui lancia la primitiva di utilizzo e a questo punto è il SO che gestisce la periferica. Questo fatto porta il grande vantaggio di poter cambiare le periferiche senza dover alterare il programma che le utilizza.

La seconda novità è la parallelizzazione degli utenti tramite multiprogrammazione e time-sharing: più utenti possono utilizzare la macchina contemporaneamente (o quasi), alternando tra di essi l'uso della CPU.

Con la terza generazione i SO diventano adattabili a diversi elaboratori (sempre però della stessa famiglia). Questo porta a vedere delle macchine diverse allo stesso modo, poiché si usano gli stessi programmi con le stesse istruzioni (macchine virtuali). I sistemi riescono a svolgere più funzioni, anche se i linguaggi di comando rimangono molto complessi.

E' con l'avvento della quarta generazione che i SO assumono un'interfaccia grafica amichevole, per facilitare l'uso della macchina all'utente. Questa generazione (l'ultima in ordine di tempo) riassume in sé le caratteristiche migliori delle precedenti versioni, tra cui l'adattabilità ad elaboratori diversi (macchine virtuali). Questi SO sono in grado di gestire anche sistemi multiprocessore e sistemi distribuiti, dove più CPU sono collegate tra loro in modo da aumentare le prestazioni di calcolo.
Esistono due tecniche per la gestione di un sistema di calcolo:
monoprogrammazione e multiprogrammazione.

Un sistema monoprogrammato gestisce in modo sequenziale nel tempo i diversi programmi: soltanto quando il programma precedente è terminato il successivo può iniziare l'esecuzione. In questo modo tutte le risorse sia Hw che Sw del sistema sono dedicate ad un solo programma. Così facendo si ottiene uno scarso utilizzo della macchina; basti pensare che se un programma prevede una fase di elaborazione seguita da una di stampa, durante quest'ultima la CPU rimane inutilizzata con conseguente perdita di efficienza.

L'utilizzazione delle risorse di un sistema si misura in base al grado di utilizzo della CPU. Questo si può quantificare (nel caso di un sistema monoprogrammato) facendo il rapporto tra T_p e T_t , dove T_p è il tempo che la Cpu impiega ad eseguire il programma e T_t è il tempo totale di permanenza del programma nel sistema.

Se un programma prevede una fase di elaborazione ed una di stampa il T_p è molto piccolo (la CPU è veloce) e il $T_t = T_p + T_s$ (T_s tempo di stampa) è molto più grande di T_p : è immediato riconoscere che l'utilizzo della CPU in un sistema di questo tipo è molto basso.

Un'altra grandezza che permette di quantificare l'utilizzo delle risorse è il throughput, definito come il numero di programmi eseguiti per unità di tempo. In un sistema monoprogrammato il throughput è sempre uguale ad uno, visto che i programmi vengono eseguiti in modo sequenziale.

Per far crescere il Thr si ricorre ai sistemi multiprogrammati: si gestiscono simultaneamente più programmi indipendenti, facendo sì che un programma possa iniziare l'esecuzione prima che il

precedente sia terminato (a differenza dei sist. mono.). L'aggettivo simultaneo non va preso alla lettera, poiché nei sistemi di cui ci occupiamo c'è una sola CPU che si dedica all'elaborazione, quindi può essere eseguita una sola istruzione alla volta. In un sistema di questo tipo si migliorano le prestazioni perché vengono ridotti i tempi morti, cioè quegli intervalli di tempo in cui la CPU è inattiva.

Ovviamente una maggiore efficienza si paga con una maggiore complessità del SO, visto che in questo caso bisogna gestire le risorse, prima fra tutte la CPU, il cui utilizzo viene rimpallato di continuo tra i programmi in esecuzione; poi la memoria, le periferiche ecc.

Uno dei metodi per massimizzare il Thr è la gestione batch dei job. Raggruppando i programmi in liste ci si assicura, nel caso di sistema monoprogrammato, che il Thr sia sempre uguale ad uno (valore massimo che si può raggiungere): se, ad esempio, si devono eseguire due programmi A e B, mettendoli in lista si fa in modo che appena A termina comincia subito B, in modo che il Thr non vada mai sotto l'unità. Passando ai sistemi multiprogrammati il Thr può (deve) essere maggiore di uno: il SO elabora degli algoritmi che consentono di allineare i programmi da eseguire in modo tale da sfruttare al massimo le risorse. Il SO deve lavorare cercando di non sovrapporre programmi che devono utilizzare nello stesso istante la medesima risorsa, anzi se il programma che è partito prima ha finito di utilizzare una risorsa questa va subito assegnata ad un altro programma. Un'idea sarebbe quella di far partire insieme tutti i programmi da eseguire e decidere dopo a chi assegnare le risorse, ma ciò provoca problemi di saturazione delle risorse stesse (si pensi alla memoria cache).

Il time-sharing (presente sia nei sistemi mono. che multi.), è una tecnica che permette all'elaboratore di servire "simultaneamente" più utenti, dotati di terminali, dedicando a ciascuno di essi tutte le risorse del sistema per quanti di tempo fissati. La parola simultaneamente non si riferisce allo stesso istante di tempo, ma ad un intervallo abbastanza esteso, visto che una sola CPU può eseguire una sola istruzione alla volta. Un discorso a parte va fatto per quanto riguarda l'ampiezza del quanto di tempo. Questa dipende principalmente dal grado di interattività che si richiede alla macchina: se bisogna eseguire dei processi di elaborazione non si hanno particolari problemi sull'ampiezza di dt; se invece uno dei programmi è interattivo (ad esempio un programma di videoscrittura) bisogna garantire che se l'utente fornisce l'input (schiaccia i tasti), l'output (le parole appaiono sullo schermo) sia abbastanza ravvicinato. Questo si può fare se il processore rimane occupato per poco tempo mentre girano gli altri programmi, in modo da passare subito la palla all'output. Il tempo di risposta dipende ovviamente dall'applicazione. Allo stesso tempo però ci sono delle limitazioni sul valore minimo di dt. Infatti quando i programmi si alternano nell'utilizzo della CPU, ad ogni cambio i dati in essa contenuti devono essere salvati e successivamente ripristinati, c'è il cosiddetto context switch. Se il dt viene ridotto di molto queste operazioni in più che

impegnano la CPU senza partecipare all'elaborazione (hanno efficienza zero) diventano predominanti in termini di tempo di occupazione del processore. Si deve trovare un compromesso che permetta di ottenere un buona interattività ma che allo stesso tempo non riduca di molto l'efficienza della macchina.

Abbiamo visto che la gestione batch dei processi porta a massimizzare il throughput. I primi sistemi batch sono costituiti da un unico blocco in cui sono accoppiati un dispositivo di input, un elaboratore ed un dispositivo di output. In questi sistemi la macchina è un tutt'uno: le informazioni vengono inserite, elaborate dalla CPU e successivamente c'è una fase di output (ad esempio di stampa). Durante una delle fasi di I o di O l'elaboratore è inattivo: se si tiene conto del fatto che queste fasi sono in genere molto lunghe si capisce che il sistema monolitico è inefficiente, perché non sfrutta al massimo

I'elemento più costoso che lo costituisce. L'evoluzione dei sistemi batch ha portato ad una soluzione di questo problema, con la divisione del calcolatore in tre blocchi separati ognuno con la propria capacità elaborativa: dispositivo di I, elaboratore e periferica di Output. In questo modo se il programma A ha terminato la fase di input ed ha iniziato l'elaborazione, il programma B può subito iniziare con il suo input, senza aspettare che A finisca di elaborare e magari anche di stampare, come avviene nei sistemi monolitici. L'utilizzo dei tre dispositivi avviene tramite prenotazione: la gestione delle prenotazioni è un compito del SO che diventa quindi più complicato. Se il sistema fosse costituito da più periferiche di I e di O, l'ideale sarebbe di farle funzionare insieme. Con una sola prenotazione per l'I ed una per l'O questo non si può realizzare. Per poterle gestire al meglio si associa una prenotazione ad ognuna di esse, aumentando così ancora di più la complessità del SO.

Per accrescere la velocità di esecuzione dei programmi si ricorre allo spooling. Questa tecnica permette di sovrapporre (rendere parallele) più operazioni di I/O (SPOOL - Simultaneous Peripheral Operation On Line), mediante l'utilizzo della memoria come un buffer di grosse dimensioni ed il decentramento di capacità d'elaborazione sulle periferiche (già effettuato nell'evoluzione dei sistemi batch).

Per comprendere il funzionamento dello spooling ricorriamo ad un esempio. Il programma A è in fase di esecuzione, terminata la quale comincia la fase di output (supponiamo di stampa). B entra in esecuzione non appena A libera la CPU. Supponiamo che B, una volta che ha finito di elaborare, voglia utilizzare un'altra periferica di O, mentre la fase di O di A è ancora in svolgimento. Come si comporta il sistema? Se la periferica di uscita fosse una sola, bisognerebbe aspettare che A termini completamente per poi permettere a B di eseguire il suo O. Nel caso di due dispositivi di O lo spooling permette di eseguire gli O simultaneamente: terminate le rispettive fasi di elaborazione dei due programmi A e B i risultati vengono memorizzati nella memoria. A questo punto sono i processori delle periferiche che vanno a prendere i dati in memoria, guidati dai rispettivi programmi di spool-out. In questo

modo la CPU è svincolata da altre operazioni riguardanti A e B nel momento in cui termina la fase di memorizzazione dei risultati del programma B, visto che le fasi di O sono simulate dalla memorizzazione.

I programmi di spool-in e di spool-out si occupano quindi del trasferimento dei dati dai dispositivi di I alla memoria e dalla memoria ai dispositivi di O. Questi programmi vengono gestiti dal SO: essi non sono sempre attivi, ma sono sempre pronti per l'esecuzione. Il SO deve occuparsi anche delle memorie di SI e SO, dei processori delle periferiche e della sincronizzazione tra i dispositivi (ad esempio stabilire chi deve usare il bus). A questo proposito è ovvio che i due dispositivi di O non potranno leggere i dati dalla memoria nello stesso istante di tempo, perché il bus è condiviso.

Lezione 3.

Abbiamo visto come si riescano ad ottimizzare le prestazioni di un sistema di calcolo modificandone l'Hw: si è passati da un sistema monolitico che prevede I E ed O tutti in uno, ad un sistema con i tre componenti separati, ognuno con il proprio processore di potenza diversa. Abbiamo successivamente replicato i dispositivi di I e di O e siamo riusciti a rendere paralleli singolarmente gli I e gli O mediante lo spooling. Ora vediamo come si ottimizza il Sw.

Consideriamo un modello di interazione tra CPU e dispositivi di I/O: i due blocchi comunicano, cioè si scambiano dati in entrambe le direzioni, attraverso il processore del dispositivo in cui c'è un buffer per i comandi inviati dalla CPU, un bit o flag ready ed un bit o flag ready. Vediamo con un esempio come interagiscono CPU e periferica (a.e. una stampante): la CPU vuole che la periferica stampi: mette il segnale nel buffer dei comandi e setta ad uno il FB. Questo è l'avviso per la stampante che deve entrare in funzionamento. Il processore del dispositivo se non sta eseguendo alcun comando ispeziona continuamente il FB: non appena trova il valore uno, lo azzerà e va a leggere nel buffer dei comandi l'operazione che deve svolgere. Terminata l'esecuzione del comando la stampante mette a uno il FR, avvertendo così la CPU che l'operazione richiesta è stata eseguita. La CPU, che ispeziona di continuo il FR, si accorge di ciò e, azzerato il FR, inserisce un nuovo comando nel buffer. Questo sistema funziona ma è molto inefficiente: dal momento in cui la CPU mette il comando nel buffer finché il FR non va ad uno, la CPU è impegnata nel controllo del flag ready, cioè non sta elaborando alcun dato. In queste situazioni si parla di attesa attiva o busy waiting: la CPU in queste situazioni compie delle operazioni che hanno efficienza zero. Questo abbassa di molto l'efficienza dell'intero sistema visto che le fasi di I/O sono in genere molto più lunghe di quelle di elaborazione.

Il modello di comunicazione può essere espresso più precisamente come segue:

CPU	I/O Processor
repeat	repeat
invio comando;	repeat esamina BUSY
BUSY:=1;	until BUSY=1
repeat esamina READY	BUSY:=0;
until READY=1	<esegui comando>
READY:=0;	READY:=1;
until transmission_completed until device_halt	

Le righe rosse costituiscono un loop in cui la CPU viene intrappolata e che rende tremendamente inefficiente la macchina. I due processi appena descritti sono completamente indipendenti, visto che girano su due differenti processori. Essi si sincronizzano solo quando c'è bisogno di interazione, cioè durante la stampa: sono due processi asincroni che possono sincronizzarsi. Generalmente il tempo dedicato all'I/O costituisce una parte fondamentale del tempo complessivo di esecuzione di un programma.

Volendo migliorare le prestazioni di un sistema di calcolo, cioè eseguire più programmi in poco tempo, si deve ridurre il tempo dedicato alle attività di I/O, ad esempio sovrapponendo l'I di un processo con l'O di un altro. Vediamo come si raggiunge questo obiettivo considerando un semplice SO che deve gestire una macchina composta da un lettore di schede (CR), una CPU, una memoria principale ed una stampante (LP). La sequenza di operazioni che questo sistema svolge sia composta da una fase di lettura delle schede, una fase di elaborazione ed una di stampa dei risultati. In codice:

```
repeat
    leggi il pacco di schede
    compila
    carica
    esegui
    stampa i risultati
until il calcolatore si ferma
```

(fase di elaborazione)

Il compito del SO è di controllare la sequenza di passi e gestire tutti i dispositivi di I/O, che in questo caso sono il lettore di schede e la stampante. Prima di vedere quali sono i programmi di controllo dei due dispositivi dobbiamo definire i vari stati in cui una periferica può trovarsi. Per il CR e la LP lo stato dipende rispettivamente dall'interruttore, dal comando e dal pacco di schede (CR) e dall'interruttore e dal comando (LP). Per il CR gli stati sono LIBERO, PRONTO e OCCUPATO: lo stato di LIBERO si ha sempre quando l'interruttore è aperto, indipendentemente se c'è un comando o se c'è un pacco di schede. Il CR permane in questo stato anche se l'interruttore è a 1, ma non c'è un pacco di schede in ingresso. Lo stato di PRONTO si raggiunge quando CR è acceso, c'è il pacco di schede e non è stato ancora inviato il comando. Quando questo avviene il CR diventa OCCUPATO. Per la LP il ragionamento è simile: permane dello stato di LIBERO fin quando l'interruttore non viene chiuso. Se non c'è un comando allora la LP è nello stato PRONTO (con l'interruttore chiuso), quando arriva il comando commuta in OCCUPATO. Gli stati più importanti sono PRONTO e OCCUPATO: è solo attraverso comandi Sw che si può passare dall'uno all'altro, mentre lo stato LIBERO è influenzato da comandi fisici (interruttore). Vediamo ora un esempio di un programma di controllo del CR. Definiamo INDCR l'indirizzo dell'area di memoria riservata a contenere le informazioni sulle schede, SL la variabile che rappresenta il numero di schede lette e IC2 l'indirizzo corrente dell'area di memoria:

```
programma di controllo CR:
fissa INDCR
SL := 0
IC2 := INDCR
attendi fino a che CR diventa PRONTO
repeat
```

```

invia comando (IC2) /* leggi scheda */
SL := SL + 1
calcola il nuovo indirizzo IC2
attendi mentre CR e' OCCUPATO
until CR diventa LIBERO
Fissato l'indirizzo in cui mettere le informazioni della prima
scheda letta e inizializzato il valore di SL a zero il programma
comincia a leggere quando il CR diventa PRONTO. Infatti non è
detto che CR sia già PRONTO perché potrebbe essere impegnato in
un'altra lettura oppure spento. Iniziata la fase di lettura il
programma esegue le istruzioni: legge la scheda, incrementa SL e
calcola il nuovo indirizzo IC2. Prima di proseguire con un'altra
scheda però c'è una fase di attesa, perché il CR impiega più tempo
a leggere una scheda rispetto al tempo di esecuzione da parte
della CPU di due istruzioni. La fase di input termina quando CR
diventa LIBERO, cioè quando viene spento o quando non ci sono più
schede da leggere.
L'aspetto più significativo di questo esempio sono le due fasi di
attesa: durante questo tempo, cioè quando il CR è occupato (o nel
primo caso anche spento), la CPU è impegnata nel continuo
controllo di un bit (il flag ready). Si parla in questo caso di
attesa attiva o busy waiting. Questo intervallo di tempo che è in
genere molto lungo se confrontato con i tempi di esecuzione della
CPU abbassa notevolmente le prestazioni della macchina.
Analogni ragionamenti possono essere fatti per la LP. Definendo:
INDLP = indirizzo di inizio dell'area di memoria
contenente le linee da stampare
LS = contatore linee da stampare
IC1 = indirizzo corrente area di memoria
il programma di controllo della LP è:

```

```

IC1 := INDLP
repeat
    attendi fino a che LP PRONTA
    invia comando(IC1) /* stampa una linea */
    LS := LS - 1
    calcola nuovo indirizzo IC1
until LS = 0

```

Anche in questo caso c'è una fase di attesa: la CPU continua a controllare un bit (flag), impiegando così del tempo in delle operazioni che hanno efficienza nulla (attesa attiva). Il processo funziona ma non in modo efficiente.

Entrambe queste fasi di attesa, sia per la LP che per il CR, sono dovute alla diversa velocità della CPU che esegue i due programmi di controllo nei confronti dei dispositivi. Questa differenza di velocità fa in modo che i programmi trascorrano gran parte del loro tempo in attesa, durante il quale la CPU è pronta per mandare un nuovo comando ma deve attendere l'esecuzione fisica del comando precedente da parte del dispositivo. L'efficienza può essere aumentata sovrapponendo fasi di input a fasi di output, in modo da ridurre i tempi morti ed aumentare il valore del throughput.

Ricordiamo che il thr è il numero di programmi eseguiti per unità di tempo:

$$\text{throughput} = n / t$$

dove n è il numero di programmi e t è il tempo necessario ad eseguirli, dato dalla somma di $I + CLE + O$: I è la somma dei tempi di input, CLE è la somma dei tempi per compile, load, execute e O è la somma dei tempi di output. Per massimizzare il thr è necessario diminuire il denominatore della frazione n/t e quindi sovrapporre le tre fasi I , CLE , O nel tempo. Per realizzare questo obiettivo occorre che i dispositivi di I/O e la CPU operino in modo il più possibile indipendente.

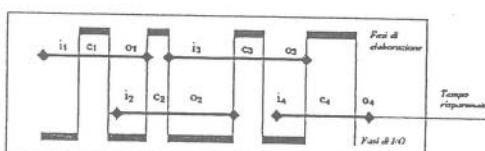
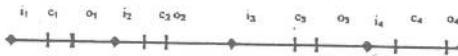
Per un sistema seriale (monoprogrammato), in cui tutte le risorse vengono affidate ad un programma alla volta, se devono essere eseguiti quattro ($n=4$) programmi, ognuno con una fase di I una di E ed una di O

il thr è

$$\frac{n}{t} = \frac{n}{CLE + I + O} = \frac{n}{CLE + \sum_{k=1}^n (I_k + O_k)}$$

dove CLE è la somma dei tempi di elaborazione di tutti i singoli programmi, tempi che non possono essere sovrapposti visto che la CPU a disposizione è unica.

Per un sistema non seriale, potendo sovrapporre I e O si ottiene un guadagno di tempo visibile anche graficamente rispetto al caso precedente



corta bisognerà attendere che termini O_3 prima di far partire C_4 (ricordiamo che non stiamo trattando la sovrapposizione tra I/O ed E). Ciò implica che può capitare che un processo di elaborazione non abbia fasi contigue.

Il tempo di esecuzione dei programmi si riduce rispetto al caso del sistema seriale. Numericamente esse è dato dalla somma di CLE e di M

$$M = I_1 + \sum_{k=1}^{n-1} m_k + O_n$$

dove I_1 e O_n vanno messi a parte perché sono fasi che non si sovrappongono con nessun'altra, il temine m_k invece è dato dal massimo tra la fase di output del programma k e quella di input del programma $k+1$.

Per raggiungere l'obiettivo della sovrapposizione dell'ingresso di un programma con l'uscita di un altro occorre un unico programma di controllo di I/O: IOC (input-output control). IOC controllando tutti i dispositivi sa quando questi sono pronti oppure occupati e di conseguenza riesce a sovrapporre le fasi al meglio. Esso manda comandi ai dispositivi PRONTI di continuo, andando in attesa e quindi diventando inefficiente solo quando tutti i dispositivi sono occupati, il che avviene molto raramente. La sua efficienza quindi è più alta dei singoli programmi di gestione di ogni periferica.

Riprendendo l'esempio del sistema costituito dal CR dalla CPU e dalla LP vediamo come si comporta il programma unico di gestione. Definendo

INDLP = indirizzo di inizio area di memoria contenente le linee da stampare

INDCR = indirizzo di inizio area di memoria in cui inserire le schede dati

IC1 = indirizzo corrente relativo a INDLP

IC2 = indirizzo corrente relativo a INDCR
il programma IOC è il seguente:

```
fissa INDCR
IC1 := INDLP, IC2 := INDCR
SL := 0
repeat
    attendi mentre CR and LP OCCUPATI
    if CR PRONTO then
        invia comando (IC2)
        SL := SL + 1
        calcola il nuovo indirizzo IC2
    fi
    if LP PRONTO then
        invia comando (IC1)
        LS := LS - 1
        calcola nuovo indirizzo IC1
    fi
until LS = 0 and CR LIBERO
```

Fissati i valori iniziali il programma gestisce la LP e CR fintanto che uno dei due o tutti e due sono pronti: invia loro dei comandi che sono quelli visti prima. L'unico caso in cui il programma diventa inefficiente ovvero va in attesa attiva è quando entrambi (c'è l'and logico) i dispositivi sono impegnati (supponiamo che siano sempre accesi) in operazioni di lettura o di stampa. Appena uno termina la stampa o la lettura viene subito gestito dal programma. Nel caso di soli due dispositivi è abbastanza probabile che entrambi risultino occupati, soprattutto se le due fasi sono lunghe, ma la probabilità si abbassa se cresce il numero delle periferiche.

Avendo la centralità dell'azione si possono rendere paralleli l'I e l'O, riducendo di molto i tempi di busy waiting della CPU rispetto ad un sistema che utilizzi programmi separati. Va notato

che le due fasi di I/O e di elaborazione sono sequenzializzate: non potrebbe essere altrimenti, perché la CPU o viene assegnata a IOC o elabora i dati dei programmi (è unica). Altra cosa da sottolineare è che la fase di attesa è sempre presente, anche se ridotta. Questa attesa è dannosa perché oltre ad occupare la CPU in operazioni ad efficienza zero, disturba anche l'accesso dei processori dei dispositivi di I/O alla memoria.

Per migliorare ulteriormente le prestazioni del sistema è necessario sovrapporre le fasi di I/O e di elaborazione. Questo si realizza facendo eseguire alla CPU altri programmi mentre il programma di controllo è in attesa per il completamento dell'esecuzione di comandi di I/O. Pensandoci bene se non c'è nessuna operazione di I/O da svolgere il IOC gira a vuoto verificando di continuo che non deve fare niente (busy waiting). E' inutile quindi chiamare il IOC quando non è richiesta nessuna operazione di I/O.

E' qui che entra in gioco il SO: questo deve sapere quando IOC è in attesa e passare in questo caso il controllo della CPU ai programmi che devono elaborare dati. La CPU è chiamata quindi a svolgere due attività fondamentali: elaborazione e supporto a IOC. La sovrapposizione può essere ottenuta dedicando a queste due fasi, alternativamente, un certo intervallo di tempo, ma è una soluzione poco efficiente perché se durante il suo dt IOC è in attesa la CPU rimane inutilizzata. Altra soluzione è quella di inserire nei programmi, ad intervalli regolari, la richiesta di esecuzione dei programmi di controllo. Questa soluzione è inaccettabile perché fa ricadere sul programmatore un compito del SO.

La soluzione si ha attraverso il concetto di interruzione, realizzato tramite Hw. Si distinguono due tipi di interruzione: da dispositivo e da timer.

L'interrupt 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, per cui è terminata la fase di esecuzione fisica del comando precedente ed il dispositivo è pronto a ricevere il comando successivo. In assenza di interruzioni il ciclo base di fetch-execute della CPU è il seguente:

```
repeat
    IR:=M[PC]
    PC:=PC+1
    execute (istruzione in IR)
until CPU halt
```

la CPU esegue i comandi contenuti nell'indirizzo presente in PC fino a quando non viene spenta.

In un sistema che presenta l'interruzione da dispositivo si può supporre che la CPU contenga un registro (IRR - Interrupt Request Register) in cui ogni bit memorizza l'interruzione di un particolare dispositivo. La CPU ad ogni ciclo controlla questo registro: se tutti i bit sono a zero allora nessun dispositivo ha fatto richiesta di essere servito e la CPU continua ad elaborare;

se c'è almeno un bit a 1 la CPU stoppa l'elaborazione e passa il controllo a IOC che va a servire la periferica. Il ciclo diventa:

```
repeat
    if IRR = 0 then
        IR := M[ PC]
        PC := PC + 1
    else IR := M[ 0] fi
    execute( istruz. in IR)
until CPU halt
```

dove M[0] è l'indirizzo in memoria di una procedura chiamata interrupt routine. Questa procedura opera nel seguente modo: salva lo stato del programma interrotto, azzerà il bit che riguarda la chiamata ricevuta, chiama il programma di controllo del dispositivo, ripristina lo stato e fa ripartire il programma interrotto.

L'interrupt da timer consiste in un unico interrupt, invece di uno per ogni dispositivo. Questo è generato periodicamente da un timer Hw (real-time clock) con frequenza programmabile dal SO. Con cadenza regolare viene attivato l'interrupt e si controlla se qualche dispositivo necessita di essere servito: in caso affermativo di passa il controllo della CPU al programma che gestisce quel dispositivo, altrimenti la CPU continua ad elaborare.

La procedura di interrupt in questo caso compie le seguenti operazioni: salva lo stato del programma interrotto, controlla tutti i dispositivi (supponiamo che siano un CR ed una LP): se CR è in stato di PRONTO chiama il programma di controllo di CR, se LP è PRONTO e LS>0 (ci sono altre linee da stampare) chiama il programma di controllo di LP. Infine ripristina lo stato e continua il programma interrotto.

Questo sistema ha il pregio di essere molto semplice ma allo stesso tempo è poco efficiente. Se ad esempio nessuna delle periferiche ha necessità di essere servita si ha un overhead dovuto al salvataggio ed al ripristino del programma interrotto. Altro difetto è che un programma può non ottenere subito il comando successivo, cioè appena ha terminato di eseguire il precedente, ma deve aspettare, al massimo per un periodo di real-time clock.

Lezione 4.

Il SO deve gestire le interruzioni attraverso le interrupt routines. Come visto prima per migliorare le prestazioni di un sistema di calcolo si ricorre alla sovrapposizione delle fasi di E, I ed O. Per un semplice sistema monoprogrammato, questo accorgimento, però, porta dei benefici limitati. Infatti al massimo si riesce a sovrapporre una fase di I/O ad una di E, visto che c'è un solo programma in esecuzione. Inoltre è difficile sovrapporre anche due fasi di I o di O poiché molto probabilmente il programma in esecuzione userà sempre le stesse periferiche. Da queste considerazioni nasce la necessità di avere più programmi in esecuzione per ottenere prestazioni maggiori, ciò significa operare con sistemi multiprogrammati. Il fatto di avere più programmi "contemporaneamente" in esecuzione permette di avere una maggiore varietà di richieste, riuscendo così a sovrapporre E, I e O di più programmi. Ovviamente c'è un limite superiore al numero di programmi che si riescono a gestire. Questa limitazione dipende dal fatto che non si hanno sufficienti risorse per gestire molti programmi contemporaneamente. Una tra tutte queste risorse è la memoria principale dove sono caricati i programmi. Quando uno di questi attende per il completamento di una operazione di I/O, il controllo della CPU può essere assegnato ad un altro: CPU multiplexing. L'obiettivo è di organizzare la successione dei programmi che occupano la CPU in modo tale da sfruttarla al massimo. Per passare il controllo della CPU ad un altro programma si utilizzano le interruzioni: se in esecuzione c'è E1 ed ad un certo punto c'è un'interruzione, il controllo della CPU passa a IOC che gestisce la richiesta; quando questo termina la CPU non viene assegnata nuovamente a E1, ma ad un altro programma, ad esempio E2. La scelta del programma cui passare la CPU è effettuata tramite un algoritmo di scheduling. Ci si domanda a questo punto quale sia il modo migliore di gestire le interruzioni, alla luce di quanto detto: è meglio utilizzare l'interrupt da Hw o il polling? La soluzione migliore è la seconda, perché nel caso in cui ci sia in esecuzione un programma di sola elaborazione (compute bound) e le interruzioni vengono dall'Hw, se non c'è una richiesta il programma può tenere la CPU per un tempo indeterminato, facendo perdere al sistema la caratteristica di multiprogrammazione. Con il polling invece ci si assicura che ogni dt (che dipende dal periodo del real time clock) lo scheduler venga invocato. Un inconveniente del polling è l'overhead che si genera quando c'è un solo programma in esecuzione.

In un sistema multiprogrammato il SO deve mantenere una tabella di stato dei dispositivi per sapere in ogni istante il loro stato, se sono occupati, se sono liberi, chi li occupa ecc. Ciò è necessario perché se un programma tenta di accedere ad una risorsa occupata il SO deve sosporarlo, mettendolo in coda per l'utilizzo di quella periferica. La gestione di queste code è un altro compito del SO. Esso deve decidere in base a degli algoritmi di gestione a quale programma affidare l'utilizzo della risorsa (l'ideale sarebbe conoscere per quanto tempo i programmi vogliono occupare

la risorsa, in modo da assegnarne l'utilizzo al più rapido, ma il SO non lo sa). Altro compito del SO è proteggere i dati di un programma...

Vi sono vari tipi di interruzioni:...

Va notato che non è detto che una richiesta di interruzione implichi una gestione immediata: ciò avviene solo se quella interruzione è abilitata. Il sistema di abilitazione delle interruzioni può essere schematizzato nel seguente modo:

IMR e AG possono essere modificati mediante istruzioni speciali o istruzioni generali, e vista l'importanza di questi comandi la loro esecuzione è affidata al...

Quando si verifica un'interruzione bisogna svolgere le seguenti operazioni:

- a- salvare tutte le informazioni necessarie per la ripresa del programma interrotto
 - b- individuare la causa dell'interruzione
 - c- servire l'interruzione
 - d- ripristinare lo stato del programma interrotto e riavviarlo
- A seconda della sofisticazione dell'Hw queste azioni impegnano in maniera maggiore o minore il Sw. Vediamo le operazioni una alla volta:
- a- l'Hw salva due registri fondamentali: PC dove è memorizzato l'indirizzo dell'istruzione da cui dovrà riprendere il programma interrotto e PSW (Program State Word) che contiene alcune informazioni quali il bit di riporto, il bit di parità ecc. I registri generali vengono salvati dal Sw. Non vengono salvati tutti quelli presenti nella CPU, ma solo quelli che la routine di interruzione va a modificare, è per questo che questo compito è affidato proprio alla int.rout. Il salvataggio dei registri deve avvenire ad interrupt disabilitati; generalmente è l'Hw che li disabilita, altrimenti lo fa il Sw.
 - b- il controllo può essere trasferito dall'Hw sempre al medesimo programma di risposta, che provvede ad individuare la causa dell'interruzione tramite skip chain, oppure direttamente a programmi di risposta separati
 - c- durante il servizio dell'interruzione si possono riabilitare tutti gli interrupt oppure mantenerne solo qualcuno, basta che sia possibile l'annidamento delle interruzioni tramite stack
 - d- durante la fase di ripristino ~~dell'~~ dei registri gli interrupt devono rimanere disabilitati, per evitare che lo stato sia ripristinato solo in parte. Alla fine di questa fase lo stato degli interrupt è quello che si aveva prima dell'interruzione (o viene riabilitato?????????)

La gestione di interrupt più o meno importanti porta a definire un livello di priorità per gli interrupt: ad ognuno di questi viene assegnato un livello che serve ad esempio durante la gestione di un'interruzione: se è possibile l'annidamento si lasciano abilitati gli interrupt a priorità maggiore. Un altro utilizzo dei livelli di priorità è nella gestione delle code degli interrupt:

se all'atto del ritorno esistono più richieste pendenti si serve quella a livello più elevato (è una delle tante scelte). Ecco un esempio di un programma di risposta ad un'interruzione di livello L:....

Si può notare che non c'è una fase di riabilitazione degli interrupt. Questa operazione è infatti automatica: in genere il bit delle interruzioni è salvato nella PSW, riabilitando questo registro si ripristina il valore del bit precedente all'interruzione.

Un processo (o task, o job) è l'unità funzionale in un SO. Esso è controllato da un programma ed ha bisogno di un processore per essere eseguito. Alcuni processi dispongono di un processore privato e pertanto sono permanentemente in esecuzione (ad esempio i controllori delle periferiche, vedi spooling). Altri processi condividono un processore comune, come può essere la CPU quando sono in esecuzione i processi utente e di sistema.

Il processo è l'unità di lavoro di un sistema che quindi consiste in una collezione di processi. Vi sono due tipi di processi: processi del SO che eseguono il codice del sistema e processi utente che eseguono il codice utente. Per capire meglio la differenza tra processo e programma si tenga conto che un programma è l'algoritmo che viene compilato, quando il programma viene eseguito allora diventa un processo. Ha senso parlare di programma in esecuzione solo quando ce n'è solo uno.

Facendo riferimento ai processi il SO deve svolgere le seguenti funzioni: creare e cancellare i processi; sospenderli e riattivarli ad esempio quando un programma (?) cerca la stampante e questa è occupata; fornire gli strumenti per la sincronizzazione e la comunicazione e gli strumenti per il trattamento delle condizioni di deadlock. Queste situazioni si hanno ad esempio quando un il processo A detiene una risorsa che serve al processo B e questo a sua volta ne occupa una che serve ad A: nessuno dei due va avanti. Il SO deve capire quando si verifica una condizione di deadlock e sbloccare i processi.

Un processo può essere definito anche come l'insieme del programma unito ad altre informazioni. Queste sono costituite dallo stato del processo che quindi lo corredano. Gli stati in cui un processo si può trovare sono tre: esecuzione, pronta e bloccato. Un processo è in esecuzione se detiene la CPU; se ad un certo punto vuole stampare per esempio, e la stampante è occupata il processo viene bloccato e messo in coda. Se il processo è in attesa che lo scheduler gli dia la CPU, si dice che è pronto. Volendo fare un esempio del ciclo in figura si prenda un processo che dopo una fase di esecuzione prevede un input da tastiera: esso va dallo stato di E a quello di B fintanto che qualcuno non schiaccia il tasto. Nel frattempo il processo è inattivo quindi la CPU viene assegnata ad un altro processo. Quando l'utente dà l'input necessario il processo va nello stato di pronto: non appena lo scheduler gli dà la CPU torna in esecuzione. La freccia che

collega lo stato E a P significa che si può revocare la CPU ad un processo durante l'esecuzione, ad esempio a causa di un interrupt.

Algoritmo programma e processo....

Lezione 5.

La presenza di più programmi in esecuzione fa nascere dei problemi di interferenza. Ad esempio un processo potrebbe tentare di modificare il programma o i dati di un altro processo o di parte del SO stesso. Evitare queste interferenze è compito del SO, che deve fornire un'adeguata protezione: politiche e meccanismi per controllare l'accesso di processi alle risorse del sistema di elaborazione. Il SO deve sapere cosa fare e come gestire gli accessi alla stessa periferica o alla stessa locazione di memoria. Bisogna ricordare però che a volte c'è bisogno che i processi comunichino tra di loro. I SO quindi deve proteggere ma anche permettere l'interazione. Uno dei metodi utilizzati per raggiungere questo obiettivo è la condivisione dell'area di memoria.

All'Hw (non al SO) è affidato il compito di rilevare gli errori, come op code illegali o riferimenti in memoria non consentiti, conseguenze di errori di programmazione o di intrusioni volute. Per evitare, ad esempio, che un processo acceda ad una locazione che va fuori dalla memoria che gli è stata concessa si usa l'Hw di indirizzamento. L'I/O system invece impedisce l'accesso diretto da parte dei processi ai dispositivi, che invece deve avvenire sempre tramite il SO perché esso deve tenere traccia delle periferiche. Una volta che l'Hw individua l'errore (o violazione della protezione) questi vengono segnalati al SO (mediante il meccanismo delle trap), il quale provvede a gestirli. In sintesi il SO provvede a terminare il processo, a segnalare la terminazione anomala (anomala perché il processo non è arrivato alla fine) ed inoltre effettua il dump della memoria. Questo è uno degli aspetti più importanti: terminato il processo tutta la memoria viene salvata sul disco. Questo file di dump contiene anche lo stato del processore, il che permette di effettuare il debug del programma e capire quindi perché è stato commesso quell'errore. Questa operazione però porta via molto tempo e molto spazio sul disco. Questi inconvenienti possono essere contenuti, limitando le dimensioni del file di dump.

Per ogni utente viene definito un dominio di protezione, in modo che ogni processo lanciato dall'utente operi nell'ambito di questo dominio. Esso specifica le risorse a cui il processo può accedere (a.e. può accedere alla stampante ma non al microfono, può accedere ad una locazione ma non ad un'altra, ecc.) e le operazioni consentite (a.e. un processo utente non potrà mai modificare gli interrupt). Il dominio di protezione è costituito da una collezione di diritti d'accesso, cioè di coppie ordinate in cui sono specificate risorsa e diritti che il processo ha su di essa, ovvero le operazioni che può compiere sulla risorsa.

Il sistema di protezione porta a definire operazioni comuni ed operazioni privilegiate. Per questo motivo devono esistere più modi di funzionamento della CPU: supervisor mode e user mode.

Quando la CPU si trova nel primo modo di funzionamento può eseguire qualunque operazione; nel secondo caso il suo campo d'azione è limitato. Il passaggio da user a supervisor avviene tramite interruzione. Questa può essere esterna (asincrona), come

nel caso della richiesta da parte di una periferica, oppure può essere interna (sincrona) ovvero generata da una SuperVisor Call (all'interno di un programma). Un esempio di commutazione tra user e supervisor si ha quando il controllo della CPU passa dall'elaborazione di un programma all'IOC, cioè quando si deve gestire una richiesta di I/O: non potendo il programma dialogare direttamente con le periferiche passa i dati al SO (in particolare all'IOC) il quale può operare in supervisor mode. La commutazione inversa invece (da supervisor a user) viene effettuata tramite un'istruzione speciale di cambiamento di modo eseguita dal SO prima della cessione del controllo ad un processo utente.

Tra le istruzioni privilegiate, eseguibili solo in modalità supervisor vi sono:

- I/O: il SO deve tenere traccia delle periferiche -> tutti gli accessi avvengono attraverso il suo tramite
- modifica dei registri che delimitano le partizioni logiche di memoria: un programma confinato ad operare in una certa area non può modificarne i limiti perché andrebbe in interferenza
- manipolazione del sistema di interruzione
- cambiamento di modo: ovvio
- halt

Tutte queste istruzioni non possono essere effettuate direttamente da un programma ma solo tramite il SO. Questo è l'unica interfaccia che i programmi utente hanno con il resto della macchina:

Lo scambio di informazioni tra SO e programma avviene tramite le supervisor call o system call. Tramite queste si richiede al SO di eseguire l'operazione di I/O: il SO verifica i diritti di accesso alla periferica, vede se questa è libera ed in seguito esegue la sc, che è memorizzata nell'area riservata al SO, essendo una procedura predefinita.

L'accesso alle procedure, trovandosi queste in una parte di memoria riservata al SO, deve avvenire in modalità sv, quindi ci deve essere un cambio di stato u->sv che è generato da questa interruzione interna. La CPU esegue le istruzioni e torna ad eseguire il programma in modalità user. L'alternanza user/supervisor avviene all'interno del codice del SO: è invisibile al programma.

Il tipo di sc richiesta dal programma è identificato dall'operando della sc stessa: INT n. Il passaggio degli eventuali parametri avviene tramite registri o per indirizzo. Ad esempio se si vogliono stampare un certo numero di righe, una volta create si chiama la sc corrispondente alla stampa indicando in qualche modo dove andare a prendere quello che deve stampare (i parametri). Si può scegliere di memorizzare le stringhe in memoria a partire da un certo indirizzo, e si memorizza questo indirizzo in un registro. Chiamata la sc, questa sa che deve andare in quel

registro a leggere un indirizzo e conosce di conseguenza dove sono salvate le stringhe. La CPU può andare a leggere nell'area di memoria del programma che deve stampare perché è in modalità sv. In questo caso si dice che i parametri sono stati passati tramite tabella.

Le categorie principali di system call sono:

- a) controllo dei processi e dei job (end, abort, load, execute ...)
- b) manipolazione dei file e dei dispositivi (create, open, close...)
- c) gestione delle informazioni (get, set time or date...)
- d) comunicazione (send, receive message...)

Lezione 6.

In un sistema multiprogrammato ci sono più programmi in esecuzione, più precisamente esistono vari processi che si trovano in diversi stati. Durante la loro esistenza due processi possono interagire tra di loro, sia per necessità che a causa di errori. Quando l'interazione è voluta i processi si scambiano informazioni, si sincronizzano ecc. Esistono due modelli di comunicazione tra processi (ICP - modello ad ambiente globale e scambio di messaggi (o ambiente locale)):

Nel modello ad ambiente globale il sistema è visto come un insieme di processi e di risorse. I processi hanno determinati diritti di accesso alle risorse le quali possono essere private (accessibili ad uno solo) o condivise (accessibili a tutti). Questo modello è la naturale astrazione per un sistema in multiprogrammazione; esso è costituito da uno o più processori che hanno accesso ad una memoria comune, che è il mezzo che permette le interazioni tra i processi.

Nel modello a scambio di messaggi invece ogni processo opera in un ambiente locale che è protetto. Le interazioni avvengono tramite lo scambio di messaggi e tutte le risorse sono private, accessibili soltanto indirettamente: il processo Pj accede ad una risorsa di Pi attraverso Pi stesso che viene chiamato per questo processo server. Per queste caratteristiche il modello a scambio di messaggi è ideale per un sistema privo di memoria comune, come un sistema distribuito. Le comunicazioni, cioè lo scambio dei messaggi, può avvenire tramite rete o tramite una memoria condivisa.

Gli strumenti di programmazione che consentono la comunicazione in questi modelli sono per il modello ad ambiente globale: monitor, semafori e primitive di sincronizzazione; per il modello a scambio di messaggi ci sono le primitive send(msg) e receive(msg). Lo strumento monitor consiste nel far vedere le risorse ai processi non direttamente ma attraverso alcune procedure del SO che sono delle funzioni che la risorsa può compiere. In questo modo, ad esempio, a P1 è garantito l'accesso alla risorsa mentre a P2 no: un solo processo alla volta accede alla risorsa, mentre gli altri sono in attesa.

Esistono tre modi in cui due processi possono interagire: cooperazione, competizione ed interferenza.

Due processi sono in cooperazione se lavorano per raggiungere lo stesso scopo: è un'interazione programmata cioè prevedibile e desiderata, ad esempio per distribuire un calcolo complesso. Per raggiungere lo stesso obiettivo e quindi per cooperare tra loro i due processi devono scambiarsi informazioni, il che avviene tramite messaggi. Oltre a questo i due processi si sincronizzano tra loro attraverso un segnale temporale e visto che questa sincronizzazione è gestita dal programmatore, viene detta diretta o esplicita.

Nel caso di interazione prevedibile ma non desiderata si parla di competizione tra processi. Ciò avviene ad esempio quando due

processi vogliono accedere alla stessa risorsa (comune ai due) che non può essere usata contemporaneamente da entrambi. Essa è quindi una interazione necessaria in tutte le macchine che dispongono di risorse limitate. La competizione porta dei vantaggi, se sfruttata bene: se più processi vogliono utilizzare la stessa risorsa, qualcuno di loro dovrà aspettare, ma l'efficienza di quella risorsa è molto alta poiché è sfruttata al massimo essendo sempre utilizzata. Ovviamente questi vantaggi ci sono se il SO che gestisce l'accesso dei processi alle risorse riesce a sincronizzare i processi in modo tale da far funzionare la risorsa di continuo. In questo caso la sincronizzazione è detta indiretta o implicita.

L'interferenza invece si ha ogni volta l'interazione tra due processi non è corretta oppure quando per errore c'è un'interazione non richiesta dalla natura del problema. Un esempio di interferenza del primo tipo si ha quando due processi devono cooperare tra loro e non si sincronizzano bene.

L'interferenza è quindi un'interazione non prevedibile e non desiderata. Nel caso di un'interazione non richiesta dal problema è facile porre rimedio: basta che la macchina fornisca meccanismi di protezione degli accessi. Per evitare che si verifichino problemi nelle interazioni volute ma programmate male bisogna adottare tecniche per sincronizzare l'accesso alle risorse condivise.

Un esempio di interazione programmata male è il seguente, in cui due programmi devono incrementare la stessa variabile una volta ciascuno e lo scheduler passa il controllo (in seguito ad un'interruzione) al secondo prima che il primo abbia terminato:

Quando c'è il context switch tra P1 e P2 il contenuto del processo uno viene salvato e con esso anche il registro A. Terminato P2 lo stato precedente viene ripristinato: così facendo in A c'è il valore originario di count che viene incrementata ma solo una volta.

Un altro esempio che può dare diversi risultati è quello di due processi in cui uno incrementa la variabile v di 1 e l'altro la stampa e la azzerà. A seconda di come lo scheduler organizza le tre istruzioni macchina si hanno tre diversi risultati, a seconda a chi decide di dare il controllo della CPU negli istanti successivi alle interruzioni che causano il context switch:

I due esempi illustrati creano dei problemi derivanti dall'accesso, in ordine errato, dei processi a dati condivisi. La parte di un programma che prevede l'accesso a dati condivisi (sia in lettura che in scrittura) è detta sezione critica. Quando due o più processi interferiscono su una stessa risorsa si ha invece una corsa critica.

Due sezioni critiche appartengono alla stessa classe se operano sugli stessi dati. Nel caso in cui almeno una delle due prevede

operazioni di scrittura le due sc devono essere eseguite mutuamente nel tempo.

Se entrambe vogliono leggere nella stessa area di memoria non c'è nessun problema.

Nel caso di competizione tra processi per l'utilizzo di una risorsa come una stampante si possono avere delle interferenze se lo scheduler organizza male la sequenza delle operazioni:

in questo caso il flag libera viene impostato ad un valore logico falso dopo che la stampante è stata assegnata a P1. Così facendo P2 legge ancora libera = true e la occupa anch'esso: la stampante è assegnata ad entrambi!

Un altro esempio di interferenza si può fare quando due processi interagiscono per lo scambio di messaggi. Essi utilizzano lo stack seguendo queste primitive:

Se lo scheduler combina male le operazioni si può avere una situazione in cui P1 inserisce mentre P2 preleva:

In questo modo P2 legge un messaggio non definito perché l'indirizzo corrente dello stack è stato incrementato e P1 sovrascrive l'ultimo elemento. Il problema nasce quando non si completa la fase di inserimento poiché in seguito ad un'interruzione si passa il controllo della CPU al processo che legge: le due sc devono essere eseguite una alla volta.

Per risolvere il problema delle corse critiche si ricorre alla mutua esclusione. Una buona soluzione soddisfa queste ipotesi: i due processi non devono mai essere insieme in sezioni critiche della stessa classe (altrimenti interferirebbero); nessuna ipotesi va fatta sulla velocità relativa dei processi perché la soluzione deve adeguarsi a qualsiasi situazione. Altra ipotesi è che nessun processo fuori della sc possa bloccarne altri e nessuno deve attendere indefinitamente prima di entrare in sc.

Una soluzione di mutua esclusione è il Test and Set Lock, che avviene tramite Hw. Prima di entrare in sc un processo testa la variabile di lock attraverso questo algoritmo:

il contenuto di lock viene copiato in un registro, viene comparato col valore 0 e solo se è 0 viene avviata la sc. Se il confronto tra il registro (cioè lock) e zero dà risultato positivo allora il processo si riporta alla prima riga, ciclando. Terminata la sc viene svolta la procedura di unlock:

lock viene azzerato e si ritorna al processo. La lettura e la scrittura di lock sono due fasi indivisibili e finché non termina l'esecuzione di TSL (cioè della prima rica di lock()) nessun altro processore può accedere alla locazione di memoria.
In forma compatta:

x è l'indirizzo della locazione di memoria che contiene la variabile lock.

L'utilizzo di queste due primitive ha degli aspetti negativi. Innanzitutto porta inefficienza perché se un processo vuole entrare in una sc già occupata, la lock() lo fa ciclare di continuo finché lock=0: c'è busy waiting.

Un altro problema è legato all'inversione di priorità. Supponiamo che ci siano due processi, uno ad alta priorità (H) ed uno a bassa priorità (L). Se L è in sc la variabile di lock è ad 1: se H diventa "pronto" (magari dopo aver finito di eseguire qualche operazione di I/O) e vuole entrare in sezione critica, non lo può fare perché c'è L. Lo scheduler dà ad H la CPU poiché ha priorità maggiore, e questi entra nella lock(), andando in busy waiting. H non partirà con la sc rimanendo nella lock() fino a quando L non eseguirà unlock(). A questo punto se lo scheduler continua a seguire un criterio di priorità nell'assegnare la CPU L non viene schedulato, mentre H è in esecuzione ma in attesa attiva. Tutto questo porta al risultato che anche se H ha priorità alta non viene eseguito, si ha cioè un fenomeno di starvation.

Una soluzione alle corse critiche che elimina i fenomeni di busy waiting e di starvation è quella dei semafori. Un semaforo è una variabile intera non negativa ($s \geq 0$) con valore iniziale $s=s_0$. Al semaforo è associata una lista di attesa Q_s che contiene informazioni sui processi in attesa dell'autorizzazione a procedere. Ci sono solo due primitive: Wait(s) e Signal(s) che operano sulla variabile semaforo. La prima serve a valutare se la sc è già occupata; se lo è il processo viene messo in coda altrimenti viene eseguita la sua sc. Signal(s) invece serve per segnalare che la sc è libera e vi si può accedere. Esse sono realizzate tramite chiamate al SO e vengono eseguite in modo monitor.

Nel dettaglio le due primitive sono:

se $s=0$ il processo viene sospeso e le informazioni che lo riguardano vengono messe in coda, ciò significa che qualche altro processo è in sc. Altrimenti s viene decrementata ed il processo può eseguire la sc. Nel primo caso la Wait è detta bloccante e la CPU passa dal processo bloccato ad un altro: context switch. Nel caso $s>0$ la wait è detta passante.

La Signal(s) ha questa struttura:

se c'è qualche processo in coda le informazioni relative vengono cancellate da Q_s ed il suo stato è modificato in "pronto", ovvero appena lo scheduler gli consegna la CPU il processo inizia la sc. Se Q_s è vuota non c'è nessun processo in coda ed s viene incrementata. Va notato che la signal non blocca mai il processo che la esegue (anzi non ne modifica proprio lo stato): è sempre passante.

La riattivazione dei processi sospesi avviene tramite politica FIFO - First In First Out, la sc viene occupata dal primo che ne ha fatto richiesta: non ci sono fenomeni di starvation. Per capire perché si viene incrementata e decrementata si faccia riferimento a questo esempio in cui i processi P1 e P2 devono eseguire una sc ciascuno (A e B) tutte e due appartenenti alla stessa classe. Il valore iniziale di S è 1:

La soluzione dei semafori è corretta e soddisfa le ipotesi: wait e signal così come sono strutturate assicurano la mutua esclusione, qualunque sia il numero di processi e indipendentemente dalla loro velocità relativa. Nessun processo rimane indefinitamente in attesa, infatti grazie alla gestione della coda con politica FIFO un processo non si può riappropriare della sc che ha appena liberato se esistono altre richieste pendenti. Altro aspetto importante è che sono risolti i problemi di busy waiting: se un processo cerca di accedere ad una sc che è occupata non cicla sempre tra le stesse istruzioni occupando inutilmente la CPU, ma viene bloccato, in questo modo non consuma risorse.

Le primitive Wait() e Signal() sono considerate come sc brevi, perché operano su dati condivisi come la stessa variabile s. Per questo deve essere garantita la loro indivisibilità, che viene ottenuta tramite lock() e unlock(). Nel caso che il sistema sia monoprocesso ancora prima di eseguire la lock() vengono disabilitate le interruzioni, in modo tale da evitare le attese attive degli altri processi dovute alla presenza della stessa lock(). Le due primitive atomiche sono quindi:

L'uso dei semafori è utile in molte situazioni. Una delle applicazioni di questo strumento di sincronizzazione tra processi è nella risoluzione del cosiddetto problema del produttore e del consumatore: due processi P e C si scambiano messaggi (in modo unidirezionale) accedendo ad un buffer, il primo in scrittura ed il secondo in lettura. Questo buffer ha dimensioni limitate; può contenere al massimo, poniamo, n messaggi. Il processo P deve comunicare a C se ci sono messaggi nel buffer e C deve segnalare a P se c'è spazio nel buffer per inserire altri messaggi. Questo obiettivo si realizza mediante l'uso di due semafori, uno che indica che c'è un messaggio disponibile ed uno che indica lo spazio che rimane nel buffer.

P viene bloccato dalla wait() se il semaforo spzdisp è =0, cioè se il buffer è pieno la wait gli impedisce di inserire un altro messaggio. Una volta che si libera dello spazio la signal di C incrementa spzdisp e P può inserire il messaggio. Ragionamento analogo nel caso in cui il buffer sia vuoto: il semaforo msgdisp è

=0 e C viene bloccato dalla wait finché P non inserisce un messaggio e la sua signal incrementa il semaforo msgdisp.

Lezione 7.

In molte applicazioni un processo necessita di accedere a più risorse in modo esclusivo. In questi casi si possono verificare delle condizioni di deadlock: due processi detengono ciascuno una risorsa che serve all'altro per proseguire. Quando due o più processi entrano in deadlock rimangono bloccati per sempre. Ecco un esempio di deadlock ad alto livello: due processi leggono dei dati dall'unità DVD e stampano su un plotter

- A chiede l'accesso all'unità DVD -> lo ottiene
 - B chiede l'accesso al plotter -> lo ottiene
 - A chiede l'accesso al plotter -> si blocca
 - B chiede l'accesso all'unità DVD -> si blocca
- i due processi si bloccano perché continuano a tenere la risorsa che serve all'altro.

Dal punto di vista della programmazione un deadlock si verifica quando, ad esempio, il codice ha questa struttura:

processo A	processo B
...	...
wait(S1)	wait(S2)
<SC-R1>	<SC-R2>
wait(S2)	wait(S1)
<SC-R1+R2>	<SC-R2+R1>
signal(S2)	signal(S1)
signal(S1)	signal(S2)

il processo A aspetta di accedere alla risorsa R1 e la ottiene, eseguendo la sezione critica, così come il processo B che richiede la risorsa R2. Quando A cerca di accedere a R2 va in wait aspettando che il semaforo S2 diventi passante, e B fa la stessa cosa su R1. I due processi rimangono bloccati in questa parte del codice, perché nessuno dei due ha sbloccato la risorsa che detiene.

Si definisce **formalmente** la condizione di **deadlock** come quella situazione in cui dato un insieme S di processi ognuno è in attesa di un evento che solo un altro processo appartenente ad S può causare. Questo evento in genere è il rilascio di una risorsa posseduta da un altro membro dell'insieme e nessuno può provocarlo.

E' dimostrato che le condizioni affinché si verifichi un deadlock sono quattro:

- ① **mutua esclusione**: ogni risorsa può trovarsi in due stati, o è assegnata ad un processo (uno solo) oppure è disponibile; se non ci fosse la mutua esclusione allora nell'esempio fatto prima i due processi avrebbero potuto accedere insieme alle due risorse.
- ② **prendi e aspetta**: i processi che detengono già delle risorse possono richiederne altre.
- ③ **assenza di preemption**: le risorse assegnate non possono essere revocate forzatamente, solo il processo può decidere quando rilasciarle.

④ **attesa circolare:** deve esistere una lista circolare di almeno due processi ognuno dei quali è in attesa di una risorsa posseduta dal processo che segue nella lista.

Per evitare il deadlock si può decidere di adottare delle strategie che consentano di risolvere il problema quando si verifica o di prevenirlo dinamicamente. Quest'ultima soluzione si ottiene allocando con attenzione le risorse in modo i deadlock risultino impossibili. Si utilizzano opportuni algoritmi che evitano di soddisfare le richieste delle risorse da parte dei processi se ciò comporta situazioni di deadlock.

Il deadlock si può evitare anche negando una delle quattro condizioni precedenti. Analizzandole bene si vede che l'unica che conviene negare è la terza: assenza di preemption. Infatti non conviene allocare una sola risorsa alla volta; inoltre il principio della mutua esclusione deve sempre essere soddisfatto per evitare che più processi interferiscano. Consentendo il rilascio forzato delle risorse il deadlock può essere individuato e per eliminarlo si terminano dei processi a caso nel ciclo.

Esistono due modelli che permettono la comunicazione tra processi (ICP - Inter Process Communication): ambiente globale e scambio di messaggi. Nel modello a scambio di messaggi ogni processo opera in un ambiente locale che è protetto. Le interazioni avvengono tramite lo scambio di messaggi e tutte le risorse sono private, accessibili soltanto indirettamente: il processo Pj accede ad una risorsa di Pi attraverso Pi stesso che viene chiamato per questo processo server. Per queste caratteristiche il modello a scambio di messaggi è ideale per un sistema privo di memoria comune, come un sistema distribuito. Le comunicazioni, cioè lo scambio dei messaggi, può avvenire tramite rete o tramite una memoria condivisa.

La struttura di un messaggio può essere pensata in questo modo:

Type messaggio
origine: ...;
destinazione: ...;
contenuto: ...;
end

il messaggio può anche essere privo di contenuto; ciò avviene quando due processi devono sincronizzarsi per cui basta soltanto un segnale che è costituito dall'arrivo del messaggio.

Nel caso più semplice si può supporre che per ogni processo esista una coda per i messaggi in arrivo ed il processo stesso controlla se è arrivato qualche messaggio. Le primitive utilizzate 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.

Come detto prima il contenuto di un messaggio può essere anche vuoto. Lo scambio di messaggi si presta quindi ad un duplice ruolo: comunicazione e sincronizzazione. Nel primo caso un processo ottiene dei dati dal processo che gli manda il messaggio. Nel secondo caso visto, che un messaggio può essere ricevuto solo

dopo che è stato trasmesso, l'invio e la successiva ricezione di un messaggio possono servire da segnale di sincronizzazione. Al contrario del modello ad ambiente globale, nel modello a scambio di messaggi non esiste il problema della mutua esclusione, perché ogni risorsa è privata e l'accesso ad essa avviene serialmente, cioè un solo processo alla volta può accedere alla risorsa (tramite un meccanismo di tipo monitor).

I costrutti linguistici che realizzano lo scambio di messaggi si differenziano in base al modo in cui vengono designati i processi sorgente e destinatario e in base al tipo di sincronizzazione. La designazione può essere diretta (o esplicita) oppure indiretta (o globale). Nel caso di designazione diretta le primitive send e receive assumono questa forma:

```
send <expression_list> to <dest_designator>
receive <var_list> from <source designator>
```

Nella send il termine <expression_list> contiene il messaggio ovvero i dati da comunicare, mentre <dest_designator> indica la destinazione del messaggio. L'esecuzione della receive determina l'assegnamento dei valori contenuti nel messaggio alle variabili in <var_list> e la successiva distruzione del messaggio. Il termine <source designator> indica l'origine del messaggio. In questo caso la designazione diretta è detta simmetrica, perché i processi si nominano esplicitamente e simmetricamente, stabilendo un canale di comunicazione individuato dalla coppia (<dest_designator>, <source designator>). Questa tipologia di scambio di messaggi è molto semplice da realizzare e da utilizzare. Viene applicata nei modelli di tipo pipeline (condutture), costituiti da una catena di processi in cui l'output di uno è l'input di un altro. Ecco un esempio:

Nei sistemi di tipo pipeline i dati passano da un processo all'altro costituendo il flusso dell'informazione.

La designazione diretta può essere anche asimmetrica. In questo caso la comunicazione non avviene da uno ad uno, cioè da un processo ad un altro, ma da molti ad uno: il mittente esplicita il nome del destinatario ma questi non esprime il nome del processo con cui desidera comunicare, al contrario del caso precedente. Un esempio di questo tipo di comunicazione è quello del server e dei client: i client specificano nel corpo del messaggio il destinatario dei loro messaggi ma il server non esplicita il nome del client, perché è pronto a ricevere richieste da qualunque client. Il server comunque deve sapere chi gli manda il messaggio per sapere a chi deve comunicare una eventuale risposta. Il modello client-server è utilizzato nel caso in cui un processo disponga di risorse (server) a cui gli altri non possono accedere direttamente (client), come succede nel caso dello spooler che in questo modello è il server. I client ed il server possono anche risiedere su macchine remote:

Nel caso si voglia gestire la comunicazione tra uno o più client e più server le cose si fanno più complicate. Per questi modelli, che sono detti rispettivamente da uno a molti e da molti a molti, non va bene la designazione diretta. Infatti se si cercasse di adattare la designazione diretta simmetrica al modello da molti a uno la receive di un server dovrebbe garantire la ricezione da qualsiasi client il che comporterebbe l'esistenza di almeno una receive per ogni client. In questi casi viene utilizzato un altro tipo di designazione detta globale o indiretta. Essa prevede l'esistenza di un mailbox alla quale si possono mandare i messaggi e da cui si possono prelevare, ricoprendo il duplice ruolo di <dest_designator> e di <source_designator> nelle istruzioni di send e receive di un qualunque processo. Una volta inviato un messaggio ad una mailbox questo può essere prelevato da un qualunque processo che preveda nella propria receive il nome di quella mailbox. La notazione in questo caso è del tipo

```
send msg to A_mbox  
Pid:= receive message from A_mbox  
oppure  
send(A_mbox, msg)  
receive(A_mbox, msg)
```

I processi possono selezionare quali tipologie di messaggi ricevere, effettuando l'operazione di receive solo su determinate mbox. Il modello a designazione indiretta consente di programmare facilmente le interazioni client-server anche nel caso da molti a molti. I client mandano i messaggi alle mailbox alle quali sono associati i servizi, mentre i server prelevano i messaggi con le receive solo per i servizi che competono a loro.

L'implementazione di una mailbox in ambiente distribuito presenta dei problemi legati alla realizzazione. Infatti il supporto runtime del linguaggio deve garantire sia che una richiesta inviata ad una mailbox possa essere ricevuta dal server che possono soddisfarla ed allo stesso tempo non appena un processo ha effettuato la receive il messaggio deve essere reso indisponibile a tutti gli altri servitori, perché già un server si sta occupando della richiesta. Tutto questo è di difficile realizzazione per cui il modello viene sostituito da uno più semplice: designazione globale tramite porte. Le porte sono delle mailbox il cui nome può comparire solo in un processo come <source_designator> 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, risolvendo il problema della comunicazione da molti ad uno ma non quello da molti a molti. Ogni porta è associata ad un tipo di richiesta così come le mailbox, quindi un processo decide quale tipo di richieste accettare utilizzando le porte associate ad esse. Nel caso di un processo che svolga una receive su una sola porta si ha uno schema di designazione equivalente ad un direct naming asimmetrico. Nel direct naming i client possono inviare richieste diverse al server che deve selezionare cosa fare; con le porte invece richieste diverse giungono a porte diverse.

Riassumendo il direct naming simmetrico è utilizzato per la comunicazione da uno a uno, quello asimmetrico per il caso da molti ad uno, la designazione globale con mailbox è implementata nei casi da molti a molti mentre quella che utilizza le porte nei casi da molti ad uno. Il global naming è il caso più generale, ma allo stesso tempo è il più difficile da realizzare; gli altri schemi sono più facili da implementare ma limitano i tipi di interazione direttamente programmabili.

La designazione dei canali di comunicazione può avvenire staticamente o dinamicamente. Nel primo caso i canali sono definiti al tempo di compilazione e quindi un programma non può comunicare attraverso altri canali. Un programma con naming statico che viene eseguito in un ambiente dinamico vede diminuite le sue possibilità di sopravvivenza, perché se un altro programma utilizza il suo canale il programma si blocca. Questo implica che il potenziale accesso di un programma ad un canale deve essere assicurato fin dall'inizio in modo permanente.

Nel caso di naming dinamico invece lo schema statico di base che si occupa della designazione dei canali viene arricchito tramite variabili per la designazione di sorgente o destinazione.

Un altro criterio che discrimina i vari modelli per lo scambio di messaggi è la sincronizzazione. La send ad esempio può essere sincrona, asincrona e di tipo RPC. Nel primo caso 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 mittente, proprio perché questo non va avanti con l'esecuzione fino alla ricezione. Il trasferimento dei dati avviene quindi solo quando entrambi i processi sono pronti a comunicare, realizzando così un punto di sincronizzazione per entrambi. Siccome il passaggio dati avviene direttamente dal mittente al destinatario non c'è bisogno di interporre tra i due dispositivi di memoria.

La send può anche essere asincrona: il mittente continua l'esecuzione dopo aver inviato il messaggio. Per questo motivo le informazioni contenute nel messaggio non possono essere collegate allo stato attuale del processo, perché questo ha continuato nell'esecuzione. Altra differenza con il caso precedente è che per realizzare una send asincrona c'è bisogno di una coda di ingresso ad ogni processo nel caso direct naming e una coda di ingresso per ogni mailbox nel caso global naming. Lo spazio messo a disposizione per contenere i messaggi (buffer) dovrebbe essere in teoria illimitato. Si ovvia a questo problema modificando la semantica della send in modo tale da bloccare il processo quando il buffer è pieno e questo viene segnalato al mittente.

La send di tipo RPC prevede un appuntamento esteso: il mittente rimane in attesa fino a quando il destinatario ha terminato di svolgere la procedura richiesta. Il meccanismo è del tutto simile a quello della chiamata di procedura: un processo cliente chiama la procedura eseguita da un processo server su una macchina potenzialmente remota. Nel caso di direct naming la procedura identifica un processo, nel caso di global naming identifica un servizio.

Anche la receive può essere sincrona o asincrona. La receive sincrona è normalmente bloccante se non ci sono messaggi sul canale di comunicazione. Essa costituisce un punto di sincronizzazione per il processo ricevente. Un inconveniente è costituito dal fatto che in certe applicazioni si desidera ricevere solo alcuni messaggi ritardando l'elaborazione di altri. La soluzione viene dalla receive asincrona con interrogazione dello stato del canale: per ogni processo si specificano più canali di input ciascuno dedicato a messaggi di tipo diverso. Ovviamente per fare in modo che il processo possa decidere quali messaggi ricevere deve essere possibile specificare su quali canali attendere. In genere si ricorre ad una primitiva che verifica lo stato del canale e comunica se c'è un messaggio o se il canale è vuoto; la verifica è eseguita con un criterio di polling e la primitiva così risulta essere non bloccante. L'inconveniente di questa soluzione è costituito dall'attesa attiva che si ha quando un si attendono messaggi da specifici canali.

Lezione 8.

Tra i compiti più importanti del SO c'è quello di gestire le risorse del sistema. Il SO deve decidere in base a criteri di efficienza e funzionalità ben definiti come distribuire le risorse tra i vari processi. Tra tutte le risorse del sistema le più importanti sono la CPU e la memoria centrale. Ci occuperemo della sola CPU. La sezione del SO che decide a quale dei processi pronti presenti nel sistema assegnare il controllo della CPU è detta scheduler. Esiste quindi una coda di processi che sono nello stato di 'pronto' che possono passare nello stato di esecuzione qualora lo scheduler decida di assegnare loro la CPU. La scelta del processo tra tutti quelli in coda è effettuata tramite un algoritmo di scheduling, realizzato in base ad una particolare politica.

Un processo tipicamente alterna attività di CPU e di attesa per svolgere operazioni di I/O. Le parti del programma che utilizzano la CPU sono dette di CPU burst, quelle che prevedono un'attesa per I/O sono dette I/O burst. Generalmente è più probabile che i burst di CPU siano di breve durata. Infatti la frequenza con cui vengono eseguiti burst di CPU brevi è molto maggiore rispetto a burst di CPU lunga durata. Questa osservazione porta alla conclusione che i programmi di CPU bound, cioè quelli che hanno pochi burst di CPU di lunga durata, sono più difficili da trovare rispetto ai programmi di I/O bound che sono quelli con molti burst di CPU, brevi.

Uno schema molto semplice di un Cpu scheduler è il seguente:

i processi che sono nello stato di pronto e che quindi possono essere eseguiti sono nella ready queue. Quando lo scheduler entra in azione dà la CPU ad uno di questi, che può lasciare la CPU se termina l'esecuzione (freccia a dx) oppure può andare nello stato di sospeso perché ha richiesto un'operazione di I/O e viene messo nelle coda relativa alla periferica che vuole utilizzare. Terminata l'operazione rientra nella coda dei processi pronti. In realtà lo schema prevede altri elementi. Uno tra questi è il long-term scheduler: determina quali programmi dalla memoria di massa devono essere caricati in memoria principale e quindi trasformati in processi. Oltre a quali programmi caricare decide anche quanti processi mantenere in memoria, controllando così il grado di multiprogrammazione. Questo ultimo compito è molto delicato: avendo un certo numero di programmi da eseguire si potrebbe pensare di eseguirli tutti insieme, di farli partire contemporaneamente. Questo però potrebbe saturare le risorse disponibili. E' il LTS che si occupa di trovare il giusto equilibrio tra numero di programmi e buon utilizzo delle risorse, evitando di aumentare troppo il numero dei processi per non far lievitare il numero dei context switch. Proprio per assicurare che tutte le risorse siano sfruttate al meglio il criterio di selezione dei programmi da caricare è basato su un mix equilibrato tra processi di I/O bound e CPU bound. Altro elemento mancante allo schema precedente è lo short-term scheduler: seleziona tra tutti i processi in memoria pronti per

l'esecuzione quello cui assegnare la CPU. Va notato che ogni qual volta che c'è un'interruzione o un qualsiasi altro evento che causi un context switch lo short-term scheduler deve decidere a chi dare la CPU. Questi eventi sono molto frequenti il che implica che lo STS debba intervenire molto di frequente durante il funzionamento del sistema, senza tuttavia contribuire all'elaborazione. E' per questi motivi che lo STS deve essere molto efficiente, cioè l'esecuzione dell'algoritmo che individua il processo cui assegnare al CPU deve essere molto veloce. Lo schema con l'aggiunta di questi elementi diviene:

dove per semplicità è stato rappresentato un solo dispositivo di I/O. E' ovvio che quando un processo termina il LTS mette in memoria principale altri processi, il cui numero dipende dai criteri descritti prima (utilizzo risorse, limitare i context switch, ecc.).

Non in tutti i tipi di sistemi la gestione delle risorse segue lo schema mostrato. Nei sistemi time-sharing ad esempio non esiste il long-term scheduling, proprio per la natura del sistema stesso in cui gli utenti comunicano attraverso dei terminali con il sistema centrale dove girano i programmi e lo utilizzano ciascuno per un determinato dt. I processi quindi devono già essere in esecuzione. Il limite al loro numero è imposto o dal numero dei terminali connessi o dal tempo di risposta che diviene troppo elevato. Infatti se il numero dei processi è molto grande, una volta servito un utente dovrà passare molto tempo prima di tornare da lui. Si potrebbe pensare di ridurre il dt ma allo stesso tempo aumenta la frequenza dei contest switch con l'abbassamento delle prestazioni del sistema. Si deve trovare il compromesso ottimo tra numero di programmi e lunghezza di dt.

Può succedere che avviata l'esecuzione di un processo questi voglia più memoria per continuare, fatto di cui il LTS non era a conoscenza e che quindi ha allocato per il processo uno spazio minore. In questo modo non è più bilanciato l'utilizzo delle risorse perché il processo si vuole espandere e le assunzioni di LST non sono più vere. Per risolvere questo tipo di problema viene inserito il medium-term scheduler, che prende i processi già in esecuzione e li mette in una coda esterna salvando la memoria del processo sul disco: swap out. A questo punto i processi in ready queue sono diminuiti, ma non ne vengono inseriti altri perché il processo che ha subito lo swap non è terminato. Infatti il MTS provvede a reinserirlo in coda (swap in) mettendo magari altri processi in memoria di massa. Queste operazioni di trasferimento dati ovviamente costano in termini di tempo, rallentando l'esecuzione dei processi ma sono necessarie quando il bilanciamento non ottimizza l'utilizzo delle risorse.

Lo STS interviene con la riassegnazione della CPU in seguito ad uno di questi quattro eventi:

- ① un processo commuta dallo stato di esecuzione allo stato di sospeso, in seguito magari alla richiesta di un'operazione di I/O, ad una wait su semaforo, ecc.
- ② il processo in esecuzione termina

- (3) un processo commuta dallo stato di esecuzione allo stato di pronto, in seguito all'elaborazione di un interrupt che sappiamo provoca un context switch
- (4) un processo commuta dallo stato sospeso a pronto

In quest'ultimo caso alla CPU non è successo niente, perché allora c'è una riassegnazione? Può essere successo che un processo abbia terminato una fase di I/O quindi il programma di IOC ha lasciato la CPU e questo provoca un context switch. Un'altra situazione può essere quella in cui un processo ad elevata priorità, terminato l'I/O, riottiene la CPU. Gli ultimi due casi si rappresentano nello schema con una linea che collega la Cpu con la ready queue. Si distinguono due tipi di scheduling: non-preemptive e preemptive. Il primo tipo di scheduling si ha quando la riassegnazione della CPU avviene per soli eventi di tipo 1 e 2. In questo caso quindi un processo in esecuzione prosegue fino al rilascio spontaneo della CPU. Lo scheduling preemptive, invece, prevede che la CPU possa essere tolta ad un processo anche in modo forzato, in quanto la riassegnazione può avvenire anche per eventi del tipo 3 e 4. Un processo in esecuzione quindi si può vedere revocata la CPU anche se è in grado di continuare l'elaborazione. Gli algoritmi di scheduling operano la scelta dei processi secondo determinati criteri: scelto il criterio l'algoritmo punta ad ottimizzarlo. Ad esempio si impiegano algoritmi che cercano di aumentare l'utilizzazione della CPU, altri che cercano di aumentare il throughput, alcuni puntano a diminuire il waiting time, cioè il tempo trascorso dal processo nella ready queue, altri algoritmi cercano di diminuire il tempo di risposta, altri ancora hanno come obiettivo l'assenza di privilegi (fairness), ovvero permettere a tutti i processi di essere eseguiti. A seconda delle applicazioni si sceglie l'algoritmo più adatto. Per esempio per i sistemi interattivi (time sharing) è più importante minimizzare la varianza nel tempo di risposta piuttosto che il tempo medio di risposta.

Per analizzare e mettere a confronto vari algoritmi bisogna scegliere un criterio di confronto. Inoltre un'analisi accurata dovrebbe comprendere molti processi, ciascuno costituito da una sequenza di diverse centinaia di CPU burst ed I/O burst. Per semplicità considereremo un solo burst di CPU per ciascun processo mentre il criterio di confronto sarà il tempo medio di attesa. Il primo algoritmo che consideriamo è il first-come-first-served, che assegna la CPU al processo che l'ha richiesta per primo. La realizzazione di questa politica è ottenuta con code gestite FIFO. Vediamo un esempio: consideriamo 3 processi ognuno costituito da un solo burst di CPU di durata nota:

processi	burst di CPU
1	24
2	3
3	3

e calcoliamo il tempo medio di attesa nel caso in cui i processi arrivino nel seguente ordine: 1, 2, 3. I tempi di attesa sono, per

il processo 1 t=0, per il 2 t=24 e per il 3 t=3+24=27. Facendo la media dei tempi si ottiene $(0+24+27)/3=17$.
Nel caso i processi arrivino nell'ordine 2, 3, 1 si ha

Come si vede il risultato dipende fortemente dall'ordine di arrivo dei processi. I due casi visti sono i due casi limite in termini di risultato finale. In genere le prestazioni di questo algoritmo sono basse se si usa come criterio quello del tempo di attesa. Analizziamo ora l'algoritmo shortest-job-first: quando la CPU è libera viene eseguito per prima il processo con burst di CPU più breve. Per fare in modo che lo scheduler sappia qual è il più breve ad ogni processo deve essere associata la lunghezza del successivo burst di CPU. Ad esempio

processi	burst time
1	6
2	8
3	7
4	3

il tempo medio di attesa è 7, mentre con l'algoritmo FCFS si sarebbe ottenuto 10.25. Si può dimostrare che l'algoritmo SJF fornisce la soluzione ottima ma la difficoltà sta nel conoscere la lunghezza della successiva richiesta di CPU. Ci sono dei modelli analitici che cercano di predire questa lunghezza, sfruttando le informazioni che si hanno sui precedenti burst del processo. Un altro criterio che può essere usato per decidere a quale processo assegnare la CPU è attribuire ai vari processi un indice di priorità. Si realizzano così varie code di processi pronti a seconda della priorità:

Lo scheduler seleziona sempre il primo processo nella coda al livello massimo di priorità che contiene i processi pronti. In presenza di preemption, inoltre, in ogni istante è in esecuzione un processo a priorità massima, perché con la preemption i processi possono abbandonare la CPU anche forzatamente e quindi lo scheduler assegna la CPU al processo a priorità maggiore. Questo va in contrasto con il principio di fairness (correttezza) che consiste nell'evitare che alcuni programmi siano favoriti rispetto ad altri. Le priorità devono comunque esistere innanzitutto per distinguere i processi di sistema da quelli utente. Questo meccanismo può innescare un altro problema: processi a bassa priorità possono rimanere indefinitamente bloccati se lo scheduler seleziona sempre processi ad alta priorità, si ha cioè starvation. Una soluzione a questo problema è quella di aumentare gradualmente la priorità dei job che attendono, in modo che anche un processo che prima era a bassa priorità possa scavalcare nella coda i processi ad alta priorità, dato che è rimasto in attesa per molto tempo. Si possono definire quindi due tipi di priorità: statica e dinamica. La priorità statica è attribuita ai processi all'atto della loro 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 i processi foreground

(interattivi) hanno priorità alte mentre quelli background (batch) hanno priorità bassa. Ovviamente se la priorità è di tipo statico c'è la possibilità che si manifestino eventi di **starvation**. La priorità dinamica viene modificata durante l'esecuzione del processo. Così facendo si evita il verificarsi della starvation che era l'obiettivo fissato. Inoltre modificando la priorità si riesce a mantenere un buon job-mix, perché si possono penalizzare i processi che impegnano troppo la CPU e favorire quelli I/O bound.

Un altro algoritmo è il **round robin**, che prevede una coda circolare di processi pronti ai quali viene assegnata la CPU per un certo quanto di tempo (10-100 msec), senza nessuna priorità. Quando un processo viene interrotto per l'esaurimento del suo quanto di tempo, viene inserito nell'ultima posizione della coda dei processi, c'è quindi **preemption**, perché il processo viene bloccato dall'esterno. Questa politica di scheduling comporta un elevata fairness ed anche l'assenza di starvation. Tuttavia il context switch imposto all'esaurimento del quanto di tempo determina un incremento dell'**overhead**.

Gli algoritmi **SJF** e a priorità possono anche essere di tipo **preemptive**. Tale possibilità nasce quando, durante l'esecuzione di un processo, un nuovo processo entra nella coda dei processi pronti. Questo può richiedere un tempo di CPU inferiore o avere una priorità maggiore di quello in esecuzione, per cui il processo che detiene la CPU può essere sospeso ed il processore può essere assegnato al job che è appena entrato nella coda.

Avendo processi con diverse priorità si può pensare di dividerli in code diverse, creando ad esempio due code una per i job foreground ed una per i job background. In questo modo si può assegnare ad ogni coda un algoritmo di scheduling differente. Ad esempio si può assegnare alla coda dei job foreground l'algoritmo RR perché essendo a priorità maggiore si vuole dare a tutti la possibilità di essere eseguiti, mentre alla coda dei job background può essere assegnato l'algoritmo FCFS perché non interessa la priorità e quindi si utilizza quello più facile da implementare. Unendo a questo concetto quello della priorità dinamica si può consentire ad un processo di cambiare coda, anche temporaneamente, magari perché il job usa troppo tempo di CPU o perché attende da troppo tempo. Occorre definire quindi il numero di code, l'algoritmo associato ad ogni coda, i criteri che fanno spostare i job dall'alto verso il basso e viceversa ed un metodo per determinare la coda in cui un job deve entrare quando inizia il servizio.

Esistono diversi metodi per valutare analiticamente le prestazioni degli algoritmi di scheduling:

1. modelli deterministic
2. modelli basati sulla teoria delle code
3. modelli basati su reti di Petri temporizzate
4. modelli simulativi
5. ...

Per confrontare le prestazioni di più algoritmi utilizzando un modello deterministico bisogna prima fissare un carico di lavoro

che è uguale per tutti gli algoritmi e poi si possono valutare le prestazioni di ognuno. Supponiamo ad esempio che all'istante 0 arrivino 5 job in questo ordine:

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

Considerando gli algoritmi FCFS, SJF e RR con quanto =10, i tempi medi di attesa sono:

$$\text{FCFS } T = (0 + 10 + 39 + 42 + 49) / 5 = 28$$

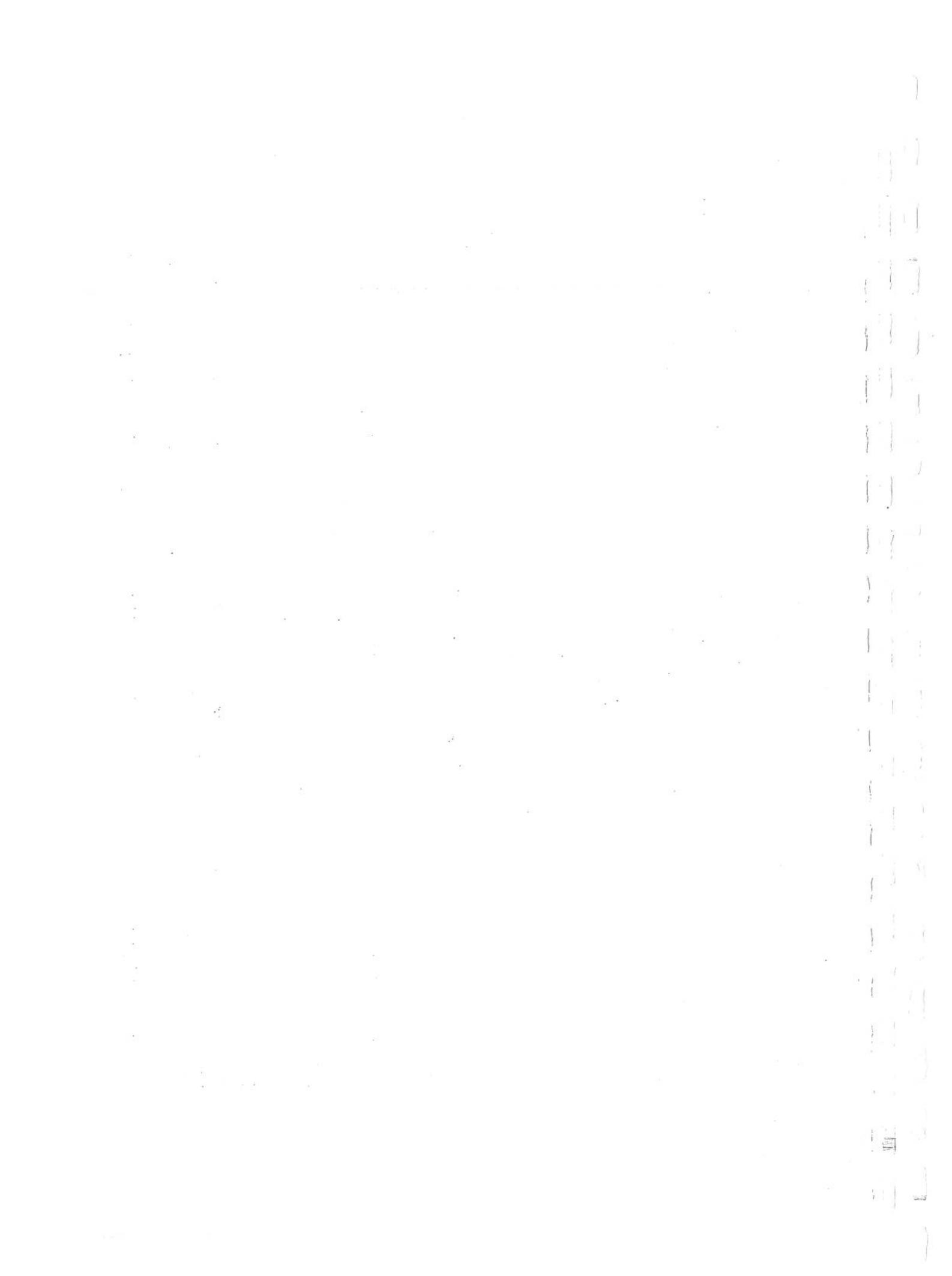
$$\text{SJF } T = (10 + 32 + 0 + 3 + 20) / 5 = 13$$

$$\text{RR (q = 10) } T = (0 + 32 + 20 + 23 + 40) / 5 = 23$$

DOMANDE

|

RISPOSTE.



1. Si illustri il concetto di **processo**, descrivendo anche i possibili stati in cui si può trovare in un S.O. multiprogrammato. (Punti 15).

Un **processo** (o task, o job) è l'unità funzionale in un SO. Esso è controllato da un programma ed ha bisogno di un processore per essere eseguito. Alcuni processi dispongono di un processore **privato** e pertanto sono permanentemente in esecuzione (ad esempio i controllori delle periferiche, vedi spooling). Altri processi condividono un processore **comune**, come può essere la CPU quando sono in esecuzione i processi utente e di sistema.

Il processo è l'unità di lavoro di un sistema che quindi consiste in una collezione di processi. Vi sono due tipi di processi: **processi del SO** che eseguono il codice del sistema e **processi utente** che eseguono il codice utente. Per capire meglio la differenza tra **processo** e **programma** si tenga conto che un programma è l'algoritmo che viene compilato, quando si esegue il programma diventa un processo. Ha senso parlare di **programma** in esecuzione quando ce n'è solo uno.

Facendo riferimento ai processi il SO deve svolgere le seguenti funzioni: **creare e cancellare i processi; sospenderli e riattivarli**, ad esempio quando un programma (?) cerca la stampante e questa è occupata; fornire gli strumenti per la sincronizzazione e la comunicazione e gli strumenti per il trattamento delle condizioni di deadlock. Queste situazioni si hanno ad esempio quando un processo A detiene una risorsa che serve al processo B e questo a sua volta ne occupa una che serve ad A: nessuno dei due va avanti. Il SO deve capire quando si verifica una condizione di deadlock e sbloccare i processi.

Un processo può essere definito anche come l'insieme del programma unito ad altre informazioni. Queste sono costituite dallo **stato del processo** che quindi lo corredano. Gli stati in cui un processo si può trovare sono tre: **esecuzione, pronto e bloccato**. Un processo è in esecuzione se detiene la CPU; se ad un certo punto vuole stampare, per esempio, e la stampante è occupata il processo viene bloccato e messo in coda. Se il processo è in attesa che lo scheduler gli dia la CPU, si dice che è pronto. Volendo fare un esempio del ciclo in figura si prenda un processo che dopo una fase di esecuzione prevede un input da tastiera: esso va dallo stato di E a quello di B fintanto che qualcuno non schiaccia il tasto. Nel frattempo il processo è inattivo quindi la CPU viene assegnata ad un altro processo. Quando l'utente dà l'input necessario il processo va nello stato di pronto: non appena lo scheduler gli dà la CPU torna in esecuzione. La freccia che collega lo stato E a P significa che si può revocare la CPU ad un processo durante l'esecuzione, ad esempio a causa di un interrupt.

2. Si descriva lo strumento di sincronizzazione dei semafori e si illustri mediante pseudo codice il problema dell'interazione tra produttore e consumatore. (Punti 10).

Durante la loro esistenza due processi possono interagire tra di loro, sia per necessità che a causa di errori. Quando l'interazione è voluta i processi si scambiano informazioni, si sincronizzano ecc. Esistono due modelli di comunicazione tra processi (ICP -): modello ad ambiente globale e scambio di messaggi (o ambiente locale).

Nel modello ad ambiente globale il sistema è visto come un insieme di processi e di risorse. I processi hanno determinati diritti di accesso alle risorse le quali possono essere private (accessibili ad uno solo) o condivise (accessibili a tutti). Questo modello è la naturale astrazione per un sistema in multiprogrammazione; esso è costituito da uno o più processori che hanno accesso ad una memoria comune, che è il mezzo che permette le interazioni tra i processi.

Esistono tre modi in cui due processi possono interagire: cooperazione, competizione ed interferenza.

Due processi sono in cooperazione se lavorano per raggiungere lo stesso scopo: è un'interazione programmata cioè prevedibile e desiderata, ad esempio per distribuire un calcolo complesso. Per raggiungere lo stesso obiettivo e quindi per cooperare tra loro i due processi devono scambiarsi informazioni, il che avviene tramite messaggi. Oltre a questo i due processi si sincronizzano tra loro attraverso un segnale temporale e visto che questa sincronizzazione è gestita dal programmatore, viene detta diretta o esplicita.

Nel caso di interazione prevedibile ma non desiderata si parla di competizione tra processi. Ciò avviene ad esempio quando due processi vogliono accedere alla stessa risorsa (comune ai due) che non può essere usata contemporaneamente da entrambi. Essa è quindi una interazione necessaria in tutte le macchine che dispongono di risorse limitate.

L'interferenza invece si ha ogni qual volta l'interazione tra due processi non è corretta oppure quando per errore c'è un'interazione non richiesta dalla natura del problema. Un esempio di interferenza del primo tipo si ha quando due processi devono cooperare tra loro e non si sincronizzano bene.

L'interferenza è quindi un'interazione non prevedibile e non desiderata. Nel caso di un'interazione non richiesta dal problema è facile porre rimedio: basta che la macchina fornisca meccanismi di protezione degli accessi. Per evitare che si verifichino problemi nelle interazioni volute ma programmate male bisogna adottare tecniche per sincronizzare l'accesso alle risorse condivise.

Un esempio di interferenza è quello di due processi in cui uno incrementa la variabile v di 1 e l'altro la stampa e la azzerà. A seconda di come lo scheduler organizza le tre istruzioni macchina si hanno tre diversi risultati, a seconda a chi decide di dare il

controllo della CPU negli istanti successivi alle interruzioni che causano il context switch:

```
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)
V+1, V=0        V, V=1        V, V=0
```

Nell'esempio ci sono dei problemi che derivano dall'accesso, in ordine errato, dei processi a dati condivisi. La parte di un programma che prevede l'accesso a dati condivisi (sia in lettura che in scrittura) è detta **sezione critica**. Quando due o più processi interferiscono su una stessa risorsa si ha invece una corsa critica.

Due sezioni critiche appartengono alla stessa classe se operano sugli stessi dati. Nel caso in cui almeno una delle due prevede operazioni di scrittura le due sc devono essere eseguite mutuamente nel tempo. Se entrambe vogliono leggere nella stessa area di memoria non c'è nessun problema.

Per risolvere il problema delle corse critiche si ricorre alla mutua esclusione.

Una soluzione di mutua esclusione è il **Test and Set Lock**, che avviene tramite Hw. Prima di entrare in sc un processo testa la variabile di lock: solo se lock è 0 viene avviata la sc. Altrimenti il processo continua nel controllo, ciclando. Terminata la sc viene svolta la procedura di unlock: lock viene azzerato e si ritorna al processo. La lettura e la scrittura di lock sono due fasi indivisibili e finché non termina l'esecuzione di TSL nessun altro processore può accedere alla locazione di memoria.

L'utilizzo di queste due primitive ha degli aspetti negativi. Innanzitutto porta inefficienza perché se un processo vuole entrare in una sc già occupata, la lock() lo fa ciclare di continuo finché lock=0: c'è **busy waiting**.

Un altro problema è legato all'inversione di priorità. Supponiamo che ci siano due processi, uno ad alta priorità (H) ed uno a bassa priorità (L). Se L è in sc la variabile di lock è ad 1: se H diventa "pronto" (magari dopo aver finito di eseguire qualche operazione di I/O) e vuole entrare in sezione critica, non lo può fare perché c'è L. Lo scheduler dà ad H la CPU poiché ha priorità maggiore, e questi entra nella lock(), andando in busy waiting. H non partirà con la sc rimanendo nella lock() fino a quando L non eseguirà unlock(). A questo punto se lo scheduler continua a seguire un criterio di priorità nell'assegnare la CPU L non viene schedulato, mentre H è in esecuzione ma in attesa attiva. Tutto questo porta al risultato che anche se H ha priorità alta non viene eseguito, si ha cioè un fenomeno di **starvation**.

Una soluzione alle corse critiche che elimina i fenomeni di busy waiting e di starvation è quella dei **semaphore**. Un semaforo è una variabile intera non negativa ($s \geq 0$) con valore iniziale $s=s_0$. Al semaforo è associata una lista di attesa Qs che contiene informazioni sui processi in attesa dell'autorizzazione a procedere. Ci sono solo due primitive: **Wait(s)** e **Signal(s)** che operano sulla variabile semaforo. La prima serve a valutare se la sc è già occupata; se lo è il processo viene messo in coda altrimenti viene eseguita la sua sc. Signal(s) invece serve per segnalare che la sc è libera e vi si può accedere. Esse sono

realizzate tramite chiamate al SO e vengono eseguite in modo monitor.

Nel dettaglio le due primitive sono:

```
wait(s)
Begin
if s = 0 then
<Processo sospeso>
<Descrittore del Processo inserito in Qs>
else
s := s - 1
end
signal(s)
Begin
if <Esiste un processo in Qs> then
<Descrittore del Processo Rimosso dal Qs>
<Stato del Processo Rimosso modificato in Pronto>
else
s := s + 1
End
```

se $s=0$ il processo viene sospeso e le informazioni che lo riguardano vengono messe in coda, ciò significa che qualche altro processo è in sc. Altrimenti s viene decrementata ed il processo può eseguire la sc. Nel primo caso la Wait è detta **bloccante** e la CPU passa dal processo bloccato ad un altro: context switch. Nel caso $s>0$ la wait è detta **passante**.

Per quanto riguarda la signal(), se c'è qualche processo in coda le informazioni relative vengono cancellate da Qs ed il suo stato è modificato in "pronto", ovvero appena lo scheduler gli consegna la CPU il processo inizia la sc. Se Qs è vuota non c'è nessun processo in coda ed s viene incrementata. Va notato che la signal non blocca mai il processo che la esegue (anzi non ne modifica proprio lo stato): è sempre passante.

La riattivazione dei processi sospesi avviene tramite politica FIFO - First In First Out, la sc viene occupata dal primo che ne ha fatto richiesta: non ci sono fenomeni di starvation.

Per capire perché s viene incrementata e decrementata si faccia riferimento a questo **esempio** in cui i processi P1 e P2 devono eseguire una sc ciascuno (A e B) tutte e due appartenenti alla stessa classe. Il valore iniziale di S è 1:

Processo P1	Processo P2
...	...
Wait(s)	Wait(s)
<Sezione Critica A>	<Sezione Critica B>
Signal(s)	Signal(s)
...	...

La soluzione dei semafori è corretta e soddisfa le ipotesi: wait e signal così come sono strutturate assicurano la **mutua esclusione**, qualunque sia il numero di processi e indipendentemente dalla loro velocità relativa. Nessun processo rimane indefinitamente in attesa, infatti grazie alla gestione della coda con politica FIFO

un processo non si può riappropriare della sc che ha appena liberato se esistono altre richieste pendenti. Altro aspetto importante è che sono risolti i problemi di **busy waiting**: se un processo cerca di accedere ad una sc che è occupata non cicla sempre tra le stesse istruzioni occupando inutilmente la CPU, ma viene bloccato, in questo modo non consuma risorse.

Le primitive **Wait()** e **Signal()** sono considerate come sc brevi, perché operano su dei dati condivisi come la stessa variabile s. Per questo deve essere garantita la loro indivisibilità, che viene ottenuta tramite **lock()** e **unlock()**. Nel caso che il sistema sia monoprocesso ancora prima di eseguire la **lock()** vengono disabilitate le interruzioni, in modo tale da evitare le attese attive degli altri processi dovute alla presenza della stessa **lock()**. Le due primitive atomiche sono quindi:

```
Wait(Semaphore):
begin;
<Disabilitazione Interruzioni>;
lock(x);
<Codice della Wait>;
unlock(x);
<Abilitazione delle Interruzioni>;
end;
```

L'uso dei semafori è utile in molte situazioni. Una delle applicazioni di questo strumento di sincronizzazione tra processi è nella **risoluzione del cosiddetto problema del produttore e del consumatore**: due processi P e C si scambiano messaggi (in modo unidirezionale) accedendo ad un **buffer**, il primo in scrittura ed il secondo in lettura. Questo buffer ha dimensioni limitate; può contenere al massimo, poniamo, n messaggi. Il processo P deve comunicare a C se ci sono messaggi nel buffer e C deve segnalare a P se c'è spazio nel buffer per inserire altri messaggi. Questo obiettivo si realizza mediante l'uso di due semafori, uno che indica che c'è un messaggio disponibile ed uno che indica lo spazio che rimane nel buffer.

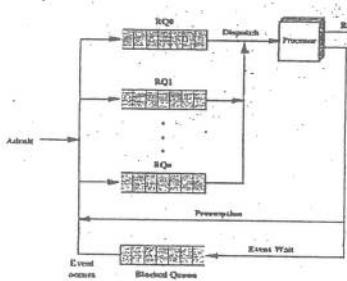
- "Messaggio disponibile"	MsgDisp	$(S_0=0)$	P viene bloccato dalla wait() se il semaforo spzdisp è =0, cioè se il buffer è pieno la wait gli impedisce di inserire un altro messaggio. Una volta che si libera dello spazio la signal di C incrementa spzdisp e P può inserire il messaggio. Ragionamento analogo nel caso in cui il buffer sia vuoto: il semaforo msgdisp è =0 e C viene bloccato dalla wait finché P non inserisce un messaggio e la sua signal incrementa il semaforo msgdisp .
- "Spazio Disponibile"	SpzDisp	$(S_0=N)$	
Produttore (P)		Consumatore (C)	
begin		Begin	
repeat		repeat	
<Produzione Msg>		Wait(MsgDisp)	
Wait(SpzDisp)		<Prelievo Msg>	
<Deposito Msg>		Signal(SpzDisp)	
Signal(MsgDisp)		<Consumazione Msg>	
forever		forever	
end		end	

STARVATION

3. Si descriva il concetto di ~~starvation~~. (Punti 5).

In un sistema di calcolo multiprogrammato in memoria principale sono presenti più processi, che si possono trovare in stati diversi. Disponendo la macchina di **risorse limitate** questi processi sono in concorrenza tra di loro, soprattutto per utilizzare la CPU. Per diverse cause, che sono poi degli errori di programmazione del SO, può capitare che uno o più processi non vengano eseguiti, rimanendo in stato di pronto o di occupato generando così una situazione di **starvation** (letteralmente *fame*). Il sistema continua ad elaborare ma questi processi rimangono bloccati per sempre.

Un **esempio** di starvation si ha quando lo scheduler di un sistema utilizza un algoritmo a priorità: si attribuiscono ai vari processi un indice di priorità e si realizzano così varie code di processi pronti a seconda della priorità. Lo scheduler seleziona



sempre il primo processo nella coda al livello massimo di priorità che contiene i processi pronti. In presenza di **preemption**, inoltre, in ogni istante è in esecuzione un processo a priorità massima, perché con la preemption i processi possono abbandonare la CPU anche forzatamente e quindi lo scheduler assegna la CPU al processo a priorità maggiore. Questo va in contrasto con il principio di **fairness** (correttezza) che consiste nell'evitare

che alcuni programmi siano favoriti rispetto ad altri. Le priorità devono comunque esistere innanzitutto per distinguere i processi di sistema da quelli utente. In questo caso quindi i processi a bassa priorità possono rimanere indefinitamente bloccati se lo scheduler seleziona sempre processi ad alta priorità, si ha cioè **starvation**. Una soluzione a questo problema è quella di aumentare gradualmente la priorità dei job che attendono, in modo che anche un processo che prima era a bassa priorità possa scavalcare nella coda i processi ad alta priorità, dato che è rimasto in attesa per molto tempo.

Si possono definire quindi due tipi di priorità: statica e dinamica. La priorità statica è attribuita ai processi all'atto della loro 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 i processi foreground (interattivi) hanno priorità alte mentre quelli background (batch) hanno priorità bassa. Ovviamente se la priorità è di tipo statico c'è la possibilità che si manifestino eventi di **starvation**.

La priorità dinamica viene modificata durante l'esecuzione del processo. Così facendo si evita il verificarsi della starvation che era l'obiettivo fissato. Inoltre modificando la priorità si riesce a mantenere un buon job-mix, perché si possono penalizzare i processi che impegnano troppo la CPU e favorire quelli I/O bound.

Un altro esempio di starvation si ha quando si vogliono risolvere i problemi che derivano dall'accesso, in ordine errato, dei processi a dati condivisi. La parte di un programma che prevede l'accesso a dati condivisi (sia in lettura che in scrittura) è

detta **sezione critica**. Quando due o più processi interferiscono su una stessa risorsa si ha invece una **corsa critica**.

Due sezioni critiche appartengono alla stessa classe se operano sugli stessi dati. Nel caso in cui almeno una delle due prevede operazioni di scrittura le due sc devono essere eseguite mutuamente nel tempo. Se entrambe vogliono leggere nella stessa area di memoria non c'è nessun problema.

Per risolvere il problema delle corse critiche si ricorre alla **mutua esclusione**.

Una soluzione di mutua esclusione è il **Test and Set Lock**, che avviene tramite Hw. Prima di entrare in sc un processo testa la variabile di lock: solo se lock è 0 viene avviata la sc e la variabile viene posta ad 1. Altrimenti il processo continua nel controllo, ciclando. Terminata la sc viene svolta la procedura di unlock: lock viene azzerata e si ritorna al processo. La lettura e la scrittura di lock sono due fasi indivisibili e finché non termina l'esecuzione di TSL nessun altro processore può accedere alla locazione di memoria.

L'utilizzo di queste due primitive ha degli **aspetti negativi**. Innanzitutto porta **inefficienza** perché se un processò vuole entrare in una sc già occupata, la lock() lo fa ciclare di continuo finché lock=0: c'è **busy waiting**.

Un altro problema è legato all'**inversione di priorità**. Supponiamo che ci siano due processi, uno ad alta priorità (H) ed uno a bassa priorità (L). Se L è in sc la variabile di lock è ad 1: se H diventa "pronto" (magari dopo aver finito di eseguire qualche operazione di I/O) e vuole entrare in sezione critica, non lo può fare perché c'è L. Lo scheduler dà ad H la CPU poiché ha priorità maggiore, e questi entra nella lock(), andando in busy waiting. H non partirà con la sc rimanendo nella lock() fino a quando L non eseguirà unlock(). A questo punto se lo scheduler continua a seguire un criterio di priorità nell'assegnare la CPU L non viene schedulato, mentre H è in esecuzione ma in attesa attiva. Tutto questo porta al risultato che anche se H ha priorità alta non viene eseguito, si ha cioè un fenomeno di **starvation**.

Una soluzione alle corse critiche che elimina i fenomeni di busy waiting e di starvation è quella dei **semafori**.

1. Si illustri una classificazione delle modalità di designazione e di sincronizzazione delle primitive di interazione tra processi nel modello ad ambiente locale (`send/receive`). (Punti 15).

Nel modello a **scambio di messaggi** ogni processo opera in un ambiente locale che è protetto. Le interazioni avvengono tramite lo scambio di messaggi e tutte le risorse sono private, accessibili soltanto indirettamente.

La struttura di un messaggio può essere pensata in questo modo:

```
Type messaggio  
origine: ...;  
destinazione: ...;  
contenuto: ...;  
end
```

il messaggio può anche essere privo di contenuto; ciò avviene quando due processi devono sincronizzarsi per cui basta soltanto un segnale che è costituito dall'arrivo del messaggio.

Le primitive utilizzate 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.

Come detto prima il contenuto di un messaggio può essere anche vuoto. Lo scambio di messaggi si presta quindi ad un duplice ruolo: **comunicazione e sincronizzazione**.

I costrutti linguistici che realizzano lo scambio di messaggi si differenziano in base al modo in cui vengono designati i processi sorgente e destinatario e in base al tipo di **sincronizzazione**.

La designazione può essere **diretta** (o esplicita) oppure **indiretta** (o globale). Nel caso di designazione **diretta** le primitive `send` e `receive` assumono questa forma:

```
send <expression_list> to <dest_designator>  
receive <var_list> from <source designator>
```

Nella `send` il termine `<expression_list>` contiene il **messaggio** ovvero i dati da comunicare, mentre `<dest_designator>` indica la **destinazione** del messaggio. L'esecuzione della `receive` determina l'assegnamento dei valori contenuti nel messaggio alle **variabili** in `<var_list>` e la successiva distruzione del messaggio. Il termine `<source designator>` indica l'**origine** del messaggio.

In questo caso la designazione diretta è detta **simmetrica**, perché i processi si nominano esplicitamente e simmetricamente, stabilendo un **canale di comunicazione** individuato dalla coppia (`<dest_designator>, <source designator>`).

Questa tipologia di scambio di messaggi è molto semplice da realizzare e da utilizzare. Viene applicata nei modelli di tipo **pipeline** (condutture), costituiti da una catena di processi in cui l'**output** di uno è l'**input** di un altro. In questi sistemi i dati passano da un processo all'altro costituendo il **flusso** dell'informazione.

La **designazione diretta** può essere anche **asimmetrica**. In questo caso la comunicazione non avviene da uno ad uno, cioè da un

processo ad un altro, ma da molti ad uno: il mittente esplicita il nome del destinatario ma questi non esprime il nome del processo con cui desidera comunicare, al contrario del caso precedente. Un esempio di questo tipo di comunicazione è quello del server e dei client (spooler): i client specificano nel corpo del messaggio il destinatario dei loro messaggi ma il server non esplicita il nome del client, perché è pronto a ricevere richieste da qualunque client. Il server comunque deve sapere chi gli manda il messaggio per sapere a chi deve comunicare una eventuale risposta.

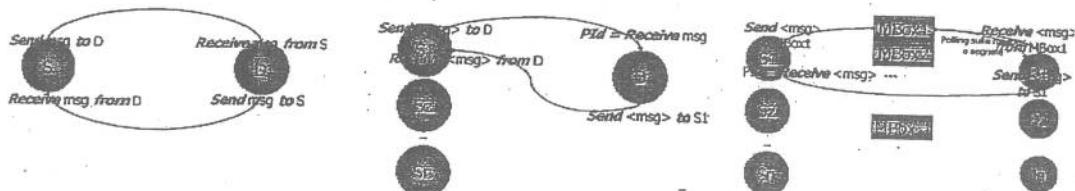
Nel caso si voglia gestire la comunicazione tra uno o più client e più server le cose si fanno più complicate. Per questi modelli, che sono detti rispettivamente da uno a molti e da molti a molti, non va bene la designazione diretta. Infatti se si cercasse di adattare la designazione diretta simmetrica al modello da molti a uno la receive di un server dovrebbe garantire la ricezione da qualsiasi client il che comporterebbe l'esistenza di almeno una receive per ogni client. In questi casi viene utilizzato un altro tipo di designazione detta globale o indiretta. Essa prevede l'esistenza di un mailbox alla quale si possono mandare i messaggi e da cui si possono prelevare, ricoprendo il duplice ruolo di <dest_designator> e di <source designator> nelle istruzioni di send e receive di un qualunque processo. Una volta inviato un messaggio ad una mailbox questo può essere prelevato da un qualunque processo che preveda nella propria receive il nome di quella mailbox. La notazione in questo caso è del tipo

```
send msg to A_mbox  
Pid:= receive message from A_mbox  
oppure  
send(A_mbox, msg)  
receive(A_mbox, msg)
```

I processi possono selezionare quali tipologie di messaggi ricevere, effettuando l'operazione di receive solo su determinate mbox. Il modello a designazione indiretta consente di programmare facilmente le interazioni client-server anche nel caso da molti a molti. I client mandano i messaggi alle mailbox alle quali sono associati i servizi, mentre i server prelevano i messaggi con le receive solo per i servizi che competono a loro.

L'implementazione di una mailbox in ambiente distribuito presenta dei problemi legati alla realizzazione. Infatti il supporto runtime del linguaggio deve garantire sia che una richiesta inviata ad una mailbox possa essere ricevuta dai server che possono soddisfarla ed allo stesso tempo non appena un processo ha effettuato la receive il messaggio deve essere reso indisponibile a tutti gli altri servitori, perché già un server si sta occupando della richiesta. Tutto questo è di difficile realizzazione per cui il modello viene sostituito da uno più semplice: designazione globale tramite porta. Le porte sono delle mailbox il cui nome può comparire solo in un processo come <source designator> 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, risolvendo il problema della comunicazione da molti ad uno ma non quello da molti a molti. Ogni porta è associata ad un tipo di richiesta così come le mailbox, quindi un processo decide quale tipo di richieste accettare utilizzando le porte associate ad esse. Nel caso di un processo che svolga una receive su una sola porta si ha uno schema di designazione equivalente ad un direct naming asimmetrico. Riassumendo il direct naming simmetrico è utilizzato per la comunicazione da uno a uno, quello asimmetrico per il caso da molti ad uno, la designazione globale con mailbox è implementata nei casi da molti a molti mentre quella che utilizza le porte nei casi da molti ad uno. Il global naming è il caso più generale, ma allo stesso tempo è il più difficile da realizzare; gli altri schemi sono più facili da implementare ma limitano i tipi di interazione direttamente programmabili.



Un altro criterio che discrimina i vari modelli per lo scambio di messaggi è la sincronizzazione. La **send** ad esempio può essere sincrona, asincrona e di tipo RPC. Nel primo caso 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 mittente, proprio perché questo non va avanti con l'esecuzione fino alla ricezione. Il trasferimento dei dati avviene quindi solo quando entrambi i processi sono pronti a comunicare, realizzando così un punto di sincronizzazione per **entrambi**. Siccome il passaggio dati avviene direttamente dal mittente al destinatario non c'è bisogno di interporre tra i due dispositivi di memoria.

La **send** può anche essere asincrona: il **mittente** continua l'esecuzione dopo aver inviato il messaggio. Per questo motivo le informazioni contenute nel messaggio non possono essere collegate allo stato attuale del processo, perché questo ha continuato nell'esecuzione. Altra differenza con il caso precedente è che per realizzare una **send** asincrona c'è bisogno di una **coda** di ingresso ad ogni processo nel caso direct naming e una coda di ingresso per ogni **mailbox** nel caso global naming. Lo spazio messo a disposizione per contenere i messaggi (**buffer**) dovrebbe essere in teoria illimitato. Si ovvia a questo problema modificando la semantica della **send** in modo tale da **bloccare** il processo quando il **buffer** è pieno e questo viene segnalato al mittente. La **send** di tipo **RPC** prevede un appuntamento esteso: il mittente **rimane in attesa** fino a quando il destinatario ha terminato di svolgere la procedura richiesta. Il meccanismo è del tutto simile a quello della chiamata di procedura: un processo cliente chiama

la procedura eseguita da un processo server su una macchina potenzialmente remota. Nel caso di direct naming la procedura identifica un **processo**, nel caso di global naming identifica un **servizio**.

Anche la **receive** può essere sincrona o asincrona. La receive sincrona è normalmente bloccante se non ci sono messaggi sul canale di comunicazione. Essa costituisce un punto di sincronizzazione per il processo ricevente. Un inconveniente è costituito dal fatto che in certe applicazioni si desidera ricevere solo alcuni messaggi ritardando l'elaborazione di altri. La soluzione viene dalla **receive asincrona con interrogazione dello stato del canale**: per ogni processo si specificano più canali di input ciascuno dedicato a messaggi di tipo diverso. Ovviamente per fare in modo che il processo possa decidere quali messaggi ricevere deve essere possibile specificare su quali canali attendere. In genere si ricorre ad una primitiva che verifica lo stato del canale e comunica se c'è un messaggio o se il canale è vuoto; la verifica è eseguita con un criterio di polling e la primitiva così risulta essere non bloccante. L'inconveniente di questa soluzione è costituito dall'attesa attiva che si ha quando un si attendono messaggi da specifici canali.

2. Si descriva il concetto di **busy waiting**. (Punti 5).

Consideriamo un **modello di interazione** tra CPU e dispositivi di I/O: i due blocchi comunicano, cioè si scambiano dati in entrambe le direzioni, attraverso il **processore** del dispositivo in cui c'è un **buffer** per i comandi inviati dalla CPU, un bit o flag **busy** ed un bit o flag **ready**. Vediamo con un esempio come interagiscono CPU e periferica (a.e. una stampante): la CPU vuole che la periferica stampi: mette il segnale nel buffer dei comandi e setta ad uno il FB. Questo è l'avviso per la stampante che deve entrare in funzionamento. Il processore del dispositivo se non sta eseguendo alcun comando ispeziona continuamente il FB: non appena trova il valore uno, lo azzerà e va a leggere nel buffer dei comandi l'operazione che deve svolgere. Terminata l'esecuzione del comando la stampante mette a uno il FR, avvertendo così la CPU che l'operazione richiesta è stata eseguita. La CPU, che ispeziona di continuo il FR, si accorge di ciò e, azzerato il FR, inserisce un nuovo comando nel buffer.

Questo sistema funziona ma è molto **inefficiente**: dal momento in cui la CPU mette il comando nel buffer finché il FR non va ad uno, la CPU è impegnata nel controllo del flag ready, cioè non sta elaborando alcun dato. In queste situazioni si parla di attesa attiva o **busy wait**: la CPU in queste situazioni compie delle operazioni che hanno efficienza zero. Questo abbassa di molto l'efficienza dell'intero sistema visto che le fasi di I/O sono in genere molto più lunghe di quelle di elaborazione.

Il modello di comunicazione può essere espresso più precisamente come segue (mediante pseudocodice):

CPU	I/ O Processor
repeat	repeat
invio comando;	repeat esamina BUSY
BUSY:=1;	until BUSY=1
repeat esamina READY	BUSY:=0;
until READY=1	<esegui comando>
READY:=0;	READY:=1;
until transmission_completed until device_halt	

Le righe rosse costituiscono un **loop** in cui la CPU viene intrappolata e che rende tremendamente inefficiente la macchina. I due processi appena descritti sono completamente indipendenti, visto che girano su due differenti processori. Essi si sincronizzano solo quando c'è bisogno di interazione, cioè durante la stampa: sono due processi asincroni che possono sincronizzarsi. Generalmente il tempo dedicato all'I/O costituisce una parte fondamentale del tempo complessivo di esecuzione di un programma. Questa fase di attesa è dovuta alla diversa velocità della CPU che esegue il programma di controllo del dispositivo (quello che manda i comandi che poi vanno nel buffer) nei confronti del dispositivo. Questa differenza di velocità fa in modo che i programmi che gestiscono le periferiche trascorrano gran parte del loro tempo in attesa, durante il quale la CPU è pronta per mandare un nuovo

comando ma deve attendere l'**esecuzione fisica** del comando precedente da parte del dispositivo.

L'efficienza può essere aumentata sovrapponendo fasi di input a fasi di output, in modo da ridurre i tempi morti ed aumentare il valore del throughput.

Per raggiungere l'obiettivo della sovrapposizione dell'ingresso di un programma con l'uscita di un altro occorre un unico programma di controllo di I/O: IOC (input-output control). IOC controllando tutti i dispositivi sa quando questi sono pronti oppure occupati e di conseguenza riesce a sovrapporre le fasi al meglio. Esso manda comandi ai dispositivi PRONTI di continuo, andando in attesa e quindi diventando inefficiente solo quando tutti i dispositivi sono occupati, il che avviene molto raramente. La sua efficienza quindi è più alta dei singoli programmi di gestione di ogni periferica.

Per migliorare ulteriormente le prestazioni del sistema è necessario sovrapporre le fasi di I/O e di elaborazione. Questo si realizza facendo eseguire alla CPU altri programmi mentre il programma di controllo è in attesa per il completamento dell'esecuzione di comandi di I/O. Pensandoci bene se non c'è nessuna operazione di I/O da svolgere il IOC gira a vuoto verificando di continuo che non deve fare niente (busy waiting). E' inutile quindi chiamare il IOC quando non è richiesta nessuna operazione di I/O.

E' qui che entra in gioco il SO: questo deve sapere quando IOC è in attesa e passare in questo caso il controllo della CPU ai programmi che devono elaborare dati. La CPU è chiamata quindi a svolgere due attività fondamentali: elaborazione e supporto a IOC. Interrupt.

La sovrapposizione può essere ottenuta dedicando a queste due fasi, alternativamente, un certo intervallo di tempo, ma è una soluzione poco efficiente perché se durante il suo dt IOC è in attesa la CPU rimane inutilizzata. Altra soluzione è quella di inserire nei programmi, ad intervalli regolari, la richiesta di esecuzione dei programmi di controllo. Questa soluzione è inaccettabile perché fa ricadere sul programmatore un compito del SO.

La soluzione si ha attraverso il concetto di interruzione, realizzato tramite Hw. Si distinguono due tipi di interruzione: da dispositivo e da timer.

L'interrupt 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, per cui è terminata la fase di esecuzione fisica del comando precedente ed il dispositivo è pronto a ricevere il comando successivo. In assenza di interruzioni il ciclo base di fetch-execute della CPU è il seguente:

```
repeat
    IR:=M[PC]
    PC:=PC+1
    execute (istruzione in IR)
```

```
until CPU halt
la CPU esegue i comandi contenuti nell'indirizzo presente in PC
fino a quando non viene spenta.
In un sistema che presenta l'interruzione da dispositivo si può
supporre che la CPU contenga un registro (IRR - Interrupt Request
Register) in cui ogni bit memorizza l'interruzione di un
particolare dispositivo. La CPU ad ogni ciclo controlla questo
registro: se tutti i bit sono a zero allora nessun dispositivo ha
fatto richiesta di essere servito e la CPU continua ad elaborare;
se c'è almeno un bit a 1 la CPU stoppa l'elaborazione e passa il
controllo a IOC che va a servire la periferica. Il ciclo diventa:

repeat
    if IRR = 0 then
        IR := M[ PC ]
        PC := PC + 1
    else IRR := M[ 0 ] fi
    execute( istruz.. in IR)
until CPU halt
```

dove M[0] è l'indirizzo in memoria di una procedura chiamata interrupt routine. Questa procedura opera nel seguente modo: salva lo stato del programma interrotto, azzera il bit che riguarda la chiamata ricevuta, chiama il programma di controllo del dispositivo, ripristina lo stato e fa ripartire il programma interrotto.

L'interrupt da timer consiste in un unico interrupt, invece di uno per ogni dispositivo. Questo è generato periodicamente da un timer Hw (real-time clock) con frequenza programmabile dal SO. Con cadenza regolare viene attivato l'interrupt e si controlla se qualche dispositivo necessita di essere servito: in caso affermativo di passa il controllo della CPU al programma che gestisce quel dispositivo, altrimenti la CPU continua ad elaborare.

La procedura di interrupt in questo caso compie le seguenti operazioni: salva lo stato del programma interrotto, controlla tutti i dispositivi (supponiamo che siano un CR ed una LP): se CR è in stato di PRONTO chiama il programma di controllo di CR, se LP è PRONTO e LS>0 (ci sono altre linee da stampare) chiama il programma di controllo di LP. Infine ripristina lo stato e continua il programma interrotto.

Questo sistema ha il pregio di essere molto semplice ma allo stesso tempo è poco efficiente. Se ad esempio nessuna delle periferiche ha necessità di essere servita si ha un overhead dovuto al salvataggio ed al ripristino del programma interrotto. Altro difetto è che un programma può non ottenere subito il comando successivo, cioè appena ha terminato di eseguire il precedente, ma deve aspettare, al massimo per un periodo di real-time clock.

1. Si descrivano i principali algoritmi di scheduling della CPU e si confrontino i tempi medi di attesa considerando i seguenti processi P1.....P4 (in ordine di arrivo) per gli algoritmi FCFS, SJF e con priorità:

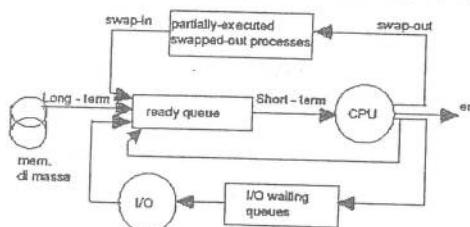
Processo Tempo di burst Priorità

P1	9	3
P2	12	1
P3	4	2
P4	7	4

N.B. La priorità è rappresentata in ordine decrescente.
(Punti 15).

Tra i compiti più importanti del SO c'è quello di gestire le risorse del sistema. Il SO deve decidere in base a criteri di efficienza e funzionalità ben definiti come distribuire le risorse tra i vari processi. Tra tutte le risorse del sistema le più importanti sono la CPU e la memoria centrale. Ci occuperemo della sola CPU. La sezione del SO che decide a quale dei processi pronti presenti nel sistema assegnare il controllo della CPU è detta scheduler. Esiste quindi una coda di processi che sono nello stato di 'pronto' che possono passare nello stato di esecuzione qualora lo scheduler decida di assegnare loro la CPU. La scelta del processo tra tutti quelli in coda è effettuata tramite un algoritmo di scheduling, realizzato in base ad una particolare politica.

Uno schema di un Cpu scheduler è il seguente:



i processi che sono nello stato di pronto e che quindi possono essere eseguiti sono nella ready queue. Quando lo scheduler entra in azione dà la CPU ad uno di questi, che può lasciare la CPU se termina l'esecuzione (freccia a dx) oppure può andare nello stato

di sospeso perché ha richiesto un'operazione si I/O e viene messo nelle coda relativa alla periferica che vuole utilizzare. Terminata l'operazione rientra nella coda dei processi pronti. In realtà lo schema prevede altri elementi. Uno tra questi è il long-term scheduler: determina quali programmi dalla memoria di massa devono essere caricati in memoria principale e quindi trasformati in processi. Oltre a quali programmi caricare decide anche quanti processi mantenere in memoria, controllando così il grado di multiprogrammazione. Questo ultimo compito è molto delicato: avendo un certo numero di programmi da eseguire si potrebbe pensare di eseguirli tutti insieme, di farli partire contemporaneamente. Questo però potrebbe saturare le risorse disponibili. E' il LTS che si occupa di trovare il giusto equilibrio tra numero di programmi e buon utilizzo delle risorse, evitando di aumentare troppo il numero dei processi per non far lievitare il numero dei context switch. Proprio per assicurare che

tutte le risorse siano sfruttate al meglio il criterio di selezione dei programmi da caricare è basato su un mix equilibrato tra processi di I/O bound e CPU bound.

Altro elemento mancante allo schema precedente è lo **short-term scheduler**: seleziona tra tutti i processi in memoria pronti per l'esecuzione quello cui assegnare la CPU. Va notato che ogni volta che c'è un'interruzione o un qualsiasi altro evento che causi un context switch lo short-term scheduler deve decidere a chi dare la CPU. Questi eventi sono molto frequenti il che implica che lo STS debba intervenire molto di frequente durante il funzionamento del sistema, senza tuttavia contribuire all'elaborazione. E' per questi motivi che lo STS deve essere molto efficiente, cioè l'esecuzione dell'algoritmo che individua il processo cui assegnare al CPU deve essere molto veloce. Può succedere che avviata l'esecuzione di un processo questi voglia più memoria per continuare, fatto di cui il LTS non era a conoscenza e che quindi ha allocato per il processo uno spazio minore. In questo modo non è più bilanciato l'utilizzo delle risorse perché il processo si vuole espandere e le assunzioni di LTS non sono più vere. Per risolvere questo tipo di problema viene inserito il **medium-term scheduler**, che prende i processi già in esecuzione e li mette in una coda esterna salvando la memoria del processo sul disco: **swap out**. A questo punto i processi in ready queue sono diminuiti, ma non ne vengono inseriti altri perché il processo che ha subito lo swap non è terminato. Infatti il MTS provvede a reinserirlo in coda (**swap in**) mettendo magari altri processi in memoria di massa. Queste operazioni di trasferimento dati ovviamente costano in termini di tempo, rallentando l'esecuzione dei processi ma sono necessarie quando il bilanciamento non ottimizza l'utilizzo delle risorse.

Lo STS interviene con la riassegnazione della CPU in seguito ad uno di questi quattro eventi:

1. un processo commuta dallo stato di esecuzione allo stato di sospeso, in seguito magari alla richiesta di un'operazione di I/O, ad una wait su semaforo, ecc.
2. il processo in esecuzione termina
3. un processo commuta dallo stato di esecuzione allo stato di pronto, in seguito all'elaborazione di un interrupt che sappiamo provoca un context switch
4. un processo commuta dallo stato sospeso a pronto

In quest'ultimo caso alla CPU non è successo niente, perché allora c'è una riassegnazione? Può essere successo che un processo abbia terminato una fase di I/O quindi il programma di IOC ha lasciato la CPU e questo provoca un context switch. Un'altra situazione può essere quella in cui un processo ad elevata priorità, terminato l'I/O, riottiene la CPU. Gli ultimi due casi si rappresentano nello schema con una linea che collega la Cpu con la ready queue. Si distinguono due tipi di scheduling: non-preemptive e preemptive. Il primo tipo di scheduling si ha quando la riassegnazione della CPU avviene per soli eventi di tipo 1 e 2. In questo caso quindi un processo in esecuzione prosegue fino al rilascio spontaneo della CPU. Lo scheduling preemptive, invece, prevede che la CPU possa essere tolta ad un processo anche in modo forzato, in quanto la riassegnazione può avvenire anche per eventi del tipo 3 e 4. Un processo in esecuzione quindi si può vedere revocata la CPU anche se è in grado di continuare l'elaborazione.

Gli algoritmi di scheduling operano la scelta dei processi secondo determinati criteri: scelto il criterio l'algoritmo punta ad ottimizzarlo. Ad esempio si impiegano algoritmi che cercano di aumentare l'utilizzazione della CPU, altri che cercano di aumentare il throughput, alcuni puntano a diminuire il waiting time, cioè il tempo trascorso dal processo nella ready queue, altri algoritmi cercano di diminuire il tempo di risposta, altri ancora hanno come obiettivo l'assenza di privilegi (fairness), ovvero permettere a tutti i processi di essere eseguiti. A seconda delle applicazioni si sceglie l'algoritmo più adatto. Per esempio per i sistemi interattivi (time sharing) è più importante minimizzare la varianza nel tempo di risposta piuttosto che il tempo medio di risposta.

Per analizzare e mettere a confronto vari algoritmi bisogna scegliere un criterio di confronto. Inoltre un'analisi accurata dovrebbe comprendere molti processi, ciascuno costituito da una sequenza di diverse centinaia di CPU burst ed I/O burst. Per semplicità considereremo un solo burst di CPU per ciascun processo mentre il criterio di confronto sarà il tempo medio di attesa. Il primo algoritmo che consideriamo è il **first-come-first-served**, che assegna la CPU al processo che l'ha richiesta per primo. La realizzazione di questa politica è ottenuta con code gestite FIFO. Vediamo un esempio: consideriamo 3 processi ognuno costituito da un solo burst di CPU di durata nota:

processi	burst di CPU
1	24
2	3
3	3

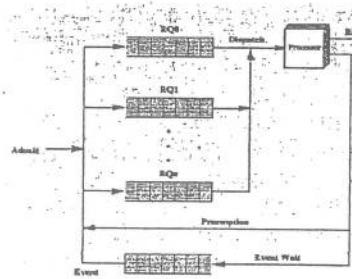
e calcoliamo il tempo medio di attesa nel caso in cui i processi arrivino nel seguente ordine: 1,2,3. I tempi di attesa sono, per il processo 1 $t=0$, per il 2 $t=24$ e per il 3 $t=3+24=27$. Facendo la media dei tempi si ottiene $(0+24+27)/3=17$.

Nel caso i processi arrivino nell'ordine 2, 3, 1 si ha ... Come si vede il risultato dipende fortemente dall'**ordine** di arrivo dei processi. I due casi visti sono i due casi limite in termini di risultato finale. In genere le prestazioni di questo algoritmo sono basse se si usa come criterio quello del tempo di attesa. Analizziamo ora l'algoritmo **shortest-job-first**: quando la CPU è libera viene eseguito per prima il processo con burst di CPU più breve. Per fare in modo che lo scheduler sappia qual è il più breve ad ogni processo deve essere associata la lunghezza del successivo burst di CPU. Ad esempio

processi	burst time
1	6
2	8
3	7
4	3

il tempo medio di attesa è 7, mentre con l'algoritmo FCFS si sarebbe ottenuto 10.25. Si può dimostrare che l'algoritmo SJF fornisce la soluzione ottima ma la difficoltà sta nel conoscere la lunghezza della successiva richiesta di CPU. Ci sono dei modelli

analitici che cercano di predire questa lunghezza, sfruttando le informazioni che si hanno sui precedenti burst del processo.



Un altro criterio che può essere usato per decidere a quale processo assegnare la CPU è attribuire ai vari processi un indice di priorità. Si realizzano così varie code di processi pronti a seconda della priorità. Lo scheduler seleziona sempre il primo processo nella coda al livello massimo di priorità che contiene i processi pronti. In presenza di **preemption**, inoltre, in ogni istante è in esecuzione un processo a priorità

massima, perché con la **preemption** i processi possono abbandonare la CPU anche forzatamente e quindi lo scheduler assegna la CPU al processo a priorità maggiore. Questo va in contrasto con il principio di **fairness** (correttezza) che consiste nell'evitare che alcuni programmi siano favoriti rispetto ad altri. Le priorità devono comunque esistere innanzitutto per distinguere i processi di sistema da quelli utente. Questo meccanismo può innescare un altro problema: processi a bassa priorità possono rimanere indefinitamente bloccati se lo scheduler seleziona sempre processi ad alta priorità, si ha cioè **starvation**. Una soluzione a questo problema è quella di aumentare gradualmente la priorità dei job che attendono, in modo che anche un processo che prima era a bassa priorità possa scavalcare nella coda i processi ad alta priorità, dato che è rimasto in attesa per molto tempo. Si possono definire, quindi due tipi di priorità: statica e dinamica. La priorità statica è attribuita ai processi all'atto della loro 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 i processi foreground (interattivi) hanno priorità alte mentre quelli background (batch) hanno priorità bassa. Ovviamente se la priorità è di tipo statico c'è la possibilità che si manifestino eventi di **starvation**.

La priorità dinamica viene modificata durante l'esecuzione del processo. Così facendo si evita il verificarsi della **starvation** che era l'obiettivo fissato. Inoltre modificando la priorità si riesce a mantenere un buon job-mix, perché si possono penalizzare i processi che impegnano troppo la CPU e favorire quelli I/O bound.

Un altro algoritmo è il **round robin**, che prevede una coda circolare di processi pronti ai quali viene assegnata la CPU per un certo quanto di tempo (10-100 msec), senza nessuna priorità. Quando un processo viene interrotto per l'esaurimento del suo quanto di tempo, viene inserito nell'**ultima posizione** della coda dei processi, c'è quindi **preemption**, perché il processo viene bloccato dall'esterno. Questa politica di scheduling comporta un elevata **fairness** ed anche l'assenza di **starvation**. Tuttavia il context switch imposto all'esaurimento del quanto di tempo determina un incremento dell'**overhead**.

Gli algoritmi SJF e a priorità possono anche essere di tipo **preemptive**. Tale possibilità nasce quando, durante l'esecuzione di un processo, un nuovo processo entra nella coda dei processi pronti. Questo può richiedere un tempo di CPU inferiore o avere una priorità maggiore di quello in esecuzione, per cui il processo che detiene la CPU può essere sospeso ed il processore può essere assegnato al job che è appena entrato nella coda.

1. Discutere eventuali varianti **preemptive** degli algoritmi di scheduling.

Perché si schedula; come è fatto lo scheduler; come si passa da un processo ad un altro; pree e non; gli algoritmi sono...; versioni pree per SJF e priorità

2. Si descriva il sistema di protezione di un S.O. (Punti 10).

La presenza di più programmi in esecuzione fa nascere dei problemi di interferenza. Ad esempio un processo potrebbe tentare di modificare il programma o i dati di un altro processo o di parte del SO stesso. Evitare queste interferenze è compito del SO, che deve fornire un'adeguata protezione: politiche e meccanismi per controllare l'accesso di processi alle risorse del sistema di elaborazione. Il SO deve sapere cosa fare e come gestire gli accessi alla stessa periferica o alla stessa locazione di memoria. Bisogna ricordare però che a volte c'è bisogno che i processi comunichino tra di loro. Il SO quindi deve proteggere ma anche permettere l'interazione. Uno dei metodi utilizzati per raggiungere questo obiettivo è la condivisione dell'area di memoria.

All'Hw (non al SO) è affidato il compito di rilevare gli errori, come op code illegali o riferimenti in memoria non consentiti, conseguenze di errori di programmazione o di intrusioni volute. Per evitare, ad esempio, che un processo acceda ad una locazione che va fuori dalla memoria che gli è stata concessa si usa l'Hw di indirizzamento. L'I/O system invece impedisce l'accesso diretto da parte dei processi ai dispositivi, che invece deve avvenire sempre tramite il SO perché esso deve tenere traccia delle periferiche. Una volta che l'Hw individua l'errore (o violazione della protezione) questi vengono segnalati al SO (mediante il meccanismo delle trap), il quale provvede a gestirli. In sintesi il SO provvede a terminare il processo, a segnalare la terminazione anomala (anomala perché il processo non è arrivato alla fine) ed inoltre effettua il dump della memoria. Questo è uno degli aspetti più importanti: terminato il processo tutta la memoria viene salvata sul disco. Questo file di dump contiene anche lo stato del processore, il che permette di effettuare il debug del programma e capire quindi perché è stato commesso quell'errore.

Questa operazione però porta via molto tempo e molto spazio sul disco. Questi inconvenienti possono essere contenuti, limitando le dimensioni del file di dump.

Per ogni utente viene definito un **dominio di protezione**, in modo che ogni processo lanciato dall'utente operi nell'ambito di questo dominio. Esso specifica le risorse a cui il processo può accedere (a.e. può accedere alla stampante ma non al microfono, può accedere ad una locazione ma non ad un'altra, ecc.) e le operazioni consentite (a.e. un processo utente non potrà mai modificare gli interrupt). Il dominio di protezione è costituito da una collezione di diritti d'accesso, cioè di coppie ordinate in cui sono specificate risorsa e diritti che il processo ha su di essa, ovvero le operazioni che può compiere sulla risorsa.

Il sistema di protezione porta a definire **operazioni comuni ed operazioni privilegiate**. Per questo motivo devono esistere più modi di funzionamento della CPU: **supervisor mode** e **user mode**.

Quando la CPU si trova nel primo modo di funzionamento può eseguire qualunque operazione; nel secondo caso il suo campo d'azione è limitato. Il passaggio da **user** a **supervisor** avviene tramite **interruzione**. Questa può essere esterna (asincrona), come nel caso della richiesta da parte di una periferica, oppure può essere interna (sincrona) ovvero generata da una **SuperVisor Call** (all'interno di un programma). Un esempio di commutazione tra user e supervisor si ha quando il controllo della CPU passa dall'elaborazione di un programma all'IOC, cioè quando si deve gestire una richiesta di I/O: non potendo il programma dialogare direttamente con le periferiche passa i dati al SO (in particolare all'IOC) il quale può operare in supervisor mode.

La commutazione inversa invece (da **supervisor** a **user**) viene effettuata tramite un'istruzione speciale di cambiamento di modo eseguita dal SO prima della cessione del controllo ad un processo utente.

Tra le **istruzioni privilegiate**, eseguibili solo in modalità **supervisor** vi sono:

- **I/O**: il SO deve tenere traccia delle periferiche -> tutti gli accessi avvengono attraverso il suo tramite
- **modifica dei registri che delimitano le partizioni logiche di memoria**: un programma confinato ad operare in una certa area non può modificarne i limiti perché andrebbe in interferenza
- **manipolazione del sistema di interruzione**
- **cambiamento di modo**: ovvio
- **halt**

Tutte queste istruzioni non possono essere effettuate direttamente da un programma ma solo tramite il SO. Questo è l'unica interfaccia che i programmi utente hanno con il resto della macchina.

Lo scambio di informazioni tra SO e programma avviene tramite le **supervisor call** o **system call**. Tramite queste si richiede al SO di eseguire l'operazione di I/O: il SO verifica i diritti di accesso alla periferica, vede se questa è libera ed in seguito esegue la sc, che è memorizzata nell'area riservata al SO, essendo una procedura predefinita.

L'accesso alle procedure, trovandosi queste in una parte di memoria riservata al SO, deve avvenire in modalità sv, quindi ci deve essere un cambio di stato u->sv

che è generato da questa interruzione interna. La CPU esegue le istruzioni e torna ad eseguire il programma in modalità user. L'alternanza user/supervisor avviene all'interno del codice del SO: è invisibile al programma. Il tipo di sc richiesta dal programma è identificato dall'operando della sc stessa: INT n. Il passaggio degli eventuali parametri avviene tramite registri o per indirizzo. Ad esempio se si vogliono stampare un certo numero di righe, una volta create si chiama la sc corrispondente alla stampa indicando in qualche modo dove andare a prendere quello che deve stampare (i parametri). Si può scegliere di memorizzare le stringhe in memoria a partire da un certo indirizzo, e si memorizza questo indirizzo in un registro. Chiamata la sc, questa sa che deve andare in quel registro a leggere un indirizzo e conosce di conseguenza dove sono salvate le stringhe. La CPU può andare a leggere nell'area di memoria del programma che deve stampare perché è in modalità sv. In questo caso si dice che i parametri sono stati passati tramite tabella. Le categorie principali di system call sono:

- a) controllo dei processi e dei job (end, abort, load, execute ...)
- b) manipolazione dei file e dei dispositivi (create, open, close...)
- c) gestione delle informazioni (get, set time or date...)
- d) comunicazione (send, receive message...)

3. Si descriva la tecnica dello spooling dell'I/O. (Punti 5).

I primi sistemi batch sono costituiti da un unico blocco in cui sono accorpati un dispositivo di input, un elaboratore ed un dispositivo di output. In questi sistemi la macchina è un tutt'uno: le informazioni vengono inserite, elaborate dalla CPU e successivamente c'è una fase di output (ad esempio di stampa). Durante una delle fasi di I o di O l'elaboratore è inattivo: se si tiene conto del fatto che queste fasi sono in genere molto lunghe si capisce che il sistema monolitico è inefficiente, perché non sfrutta al massimo l'elemento più costoso che lo costituisce. L'evoluzione dei sistemi batch ha portato ad una soluzione di questo problema, con la divisione del calcolatore in tre blocchi separati ognuno con la propria capacità elaborativa: dispositivo di I, elaboratore e periferica di Output. In questo modo se il programma A ha terminato la fase di input ed ha iniziato l'elaborazione, il programma B può subito iniziare con il suo input, senza aspettare che A finisca di elaborare e magari anche di stampare, come avviene nei sistemi monolitici. Per accrescere la velocità di esecuzione dei programmi si ricorre allo spooling. Questa tecnica permette di sovrapporre (rendere parallele) più operazioni di I/O (SPOOL - Simultaneous Peripheral Operation On Line), mediante l'utilizzo della memoria come un buffer di grosse dimensioni ed il decentramento di capacità d'elaborazione sulle periferiche (già effettuato nell'evoluzione dei sistemi batch). Per comprendere il funzionamento dello spooling ricorriamo ad un esempio. Il programma A è in fase di esecuzione, terminata la quale comincia la fase di output (supponiamo di stampa). B entra in esecuzione non appena A libera la CPU. Supponiamo che B, una volta che ha finito di elaborare, voglia utilizzare un'altra periferica di O, mentre la fase di O di A è ancora in svolgimento. Come si comporta il sistema? Se la periferica di uscita fosse una sola, bisognerebbe aspettare che A termini completamente per poi

permettere a B di eseguire il suo O. Nel caso di due dispositivi di O lo spooling permette di eseguire gli O **simultaneamente**: terminate le rispettive fasi di elaborazione dei due programmi A e B i risultati vengono memorizzati nella memoria. A questo punto sono i processori delle periferiche che vanno a prendere i dati in memoria, guidati dai rispettivi programmi di spool-out. In questo modo la CPU è svincolata da altre operazioni riguardanti A e B nel momento in cui termina la fase di memorizzazione dei risultati del programma B, visto che le fasi di O sono simulate dalla memorizzazione (a meno di comunicare l'indirizzo dove prelevare le informazioni).

I programmi di spool-in e di spool-out si occupano quindi del trasferimento dei dati dai dispositivi di I alla memoria e dalla memoria ai dispositivi di O. Questi programmi vengono gestiti dal SO: essi non sono sempre attivi, ma sono sempre pronti per l'esecuzione. Il SO deve occuparsi anche delle memorie di SI e SO, dei processori delle periferiche e della sincronizzazione tra i dispositivi (ad esempio stabilire chi deve usare il bus). A questo proposito è ovvio che i due dispositivi di O non potranno leggere i dati dalla memoria nello stesso istante di tempo, perché il bus è condiviso.

2. Differenza tra sistemi operativi proprietari e standard: discutere problemi e vantaggi.

Un sistema di calcolo può essere visto come un insieme di risorse Hw e Sw utilizzate per lo sviluppo e l'esecuzione dei programmi utente. Tali risorse devono essere gestite in modo opportuno: di questo si occupa il Sistema Operativo. Per gestione si intendono molte cose: l'utilizzo delle risorse secondo un determinato ordine (ottimizzazione dei tempi di esecuzione), rendere tali risorse disponibili a più utenti (quando questi lo richiedono), proteggerle contro accessi non autorizzati, sia voluti (un utente vuole accedere ai dati di un altro utente) sia non voluti (es.: un programma per errore cerca di scrivere in una parte di memoria dove ci sono già dei dati).

In sintesi gli obiettivi del SO sono due: rendere più semplice l'utilizzo di un sistema di calcolo e rendere più efficiente l'uso delle risorse dello stesso (oltre ad effettuare una costante azione di controllo).

L'ideale per un programmatore di SO sarebbe quello di conoscere a fondo la struttura Hw del sistema di calcolo, in modo da sfruttare in modo ottimale le risorse di ogni tipo di macchina. I SO realizzati secondo questi criteri sono detti proprietari, ed in genere vengono realizzati dalla stessa casa costruttrice dell'Hw. E' scontato che un SO di questo tipo funzioni benissimo sulla macchina per cui è stato progettato, ma presenta una scarsa efficienza se viene cambiato l'Hw: non è flessibile. Inoltre i sistemi di calcolo che utilizzano questo tipo di SO hanno interfacce grafiche diverse a seconda della casa che li ha progettati. I SO standard, invece, vengono realizzati per potersi adattare sulla gran parte delle macchine; avendo una maggiore flessibilità però perdono in efficienza se le loro prestazioni vengono confrontate con quelle dei SO proprietari. Questa differenza di prestazioni è stata ridotta nel tempo grazie all'introduzione dei DRIVER: software allegati ad ogni periferica Hw che permettono a queste di funzionare in modo più efficiente, visto che il Sw è creato dalla stessa casa costruttrice che ha realizzato l'Hw. Altre differenza fondamentale tra SO proprietari e SO standard è che l'interfaccia grafica nel primo caso cambia tra le diverse case costruttrici, nel secondo invece, siccome il SO si adatta a vari tipi di macchine, l'interfaccia utente rimane costante, il che rende virtualmente uguali due macchine che sotto il profilo Hw sono diverse.

0. Deadlock

In molte applicazioni un processo necessita di accedere a più risorse in modo esclusivo. I questi casi si possono verificare delle condizioni di deadlock: due processi detengono ciascuno una risorsa che serve all'altro per proseguire. Quando due o più processi entrano in deadlock rimangono bloccati per sempre.. Ecco un esempio di deadlock ad alto livello: due processi leggono dei dati dall'unità DVD e stampano su un plotter

- A chiede l'accesso all'unità DVD -> lo ottiene
 - B chiede l'accesso al plotter -> lo ottiene
 - A chiede l'accesso al plotter -> si blocca
 - B chiede l'accesso all'unità DVD -> si blocca
- i due processi si bloccano perché continuano a tenere la risorsa che serve all'altro.

Dal punto di vista della programmazione un deadlock si verifica quando, ad esempio, il codice ha questa struttura:

processo A	processo B
...	...
wait(S1)	wait(S2)
<SC-R1>	<SC-R2>
wait(S2)	wait(S1)
<SC-R1+R2>	<SC-R2+R1>
signal(S2)	signal(S1)
signal(S1)	signal(S2)

il processo A aspetta di accedere alla risorsa R1 e la ottiene, eseguendo la sezione critica, così come il processo B che richiede la risorsa R2. Quando A cerca di accedere a R2 va in wait aspettando che il semaforo S2 diventi passante, e B fa la stessa cosa su R1. I due processi rimangono bloccati in questa parte del codice, perché nessuno dei due ha sbloccato la risorsa che detiene.

Si definisce formalmente la condizione di deadlock come quella situazione in cui dato un insieme S di processi ognuno è in attesa di un evento che solo un altro processo appartenente ad S può causare. Questo evento in genere è il rilascio di una risorsa posseduta da un altro membro dell'insieme e nessuno può provocarlo.

E' dimostrato che le condizioni affinché si verifichi un deadlock sono quattro:

1. **mutua esclusione**: ogni risorsa può trovarsi in due stati, o è assegnata ad un processo (uno solo) oppure è disponibile; se non ci fosse la mutua esclusione allora nell'esempio fatto prima i due processi avrebbero potuto accedere insieme alle due risorse.
2. **prendi e aspetta**: i processi che detengono già delle risorse possono richiederne altre.
3. **assenza di preemption**: le risorse assegnate non possono essere revocate forzatamente, solo il processo può decidere quando rilasciarle.

4. attesa circolare: deve esistere una lista circolare di almeno due processi ognuno dei quali è in attesa di una risorsa posseduta dal processo che segue nella lista.

Per evitare il deadlock si può decidere di adottare delle strategie che consentano di risolvere il problema quando si verifica o di prevenirlo dinamicamente. Quest'ultima soluzione si ottiene allocando con attenzione le risorse in modo i deadlock risultino impossibili. Si utilizzano opportuni algoritmi che evitano di soddisfare le richieste delle risorse da parte dei processi se ciò comporta situazioni di deadlock.

Il deadlock si può evitare anche negando una delle quattro condizioni precedenti. Analizzandole bene si vede che l'unica che conviene negare è la terza: assenza di preemption. Infatti non conviene allocare una sola risorsa alla volta; inoltre il principio della mutua esclusione deve sempre essere soddisfatto per evitare che più processi interferiscano. Consentendo il rilascio forzato delle risorse il deadlock può essere individuato e per eliminarlo si terminano dei processi a caso nel ciclo.

4. Si illustri il concetto di processo, descrivendo anche i possibili stati in cui si può trovare in un S.O. multiprogrammato. (Punti 15).
5. Si descriva lo strumento di sincronizzazione dei semafori e si illustri mediante pseudo codice il problema dell'interazione tra produttore e consumatore. (Punti 10).

6. Si descriva il concetto di ~~starvation~~. (Punti 5).

3. Si illustri una classificazione delle modalità di designazione e di sincronizzazione delle primitive di interazione tra processi nel modello ad ambiente locale (send/receive). (Punti 15).

4. Si descriva lo strumento di sincronizzazione dei semafori e l'implementazione delle primitive wait e signal. (Punti 10).

5. Si descriva il concetto di busy waiting. (Punti 5).

4. Si descrivano i principali algoritmi di scheduling della CPU e si confrontino i tempi medi di attesa considerando i seguenti processi P1.....P4 (in ordine di arrivo) per gli algoritmi FCFS, SJF e con priorità:

Processo	Tempo di burst	Priorità
P1	9	3
P2	12	1
P3	4	2
P4	7	4

N.B. La priorità è rappresentata in ordine decrescente.
(Punti 15).

5. Si descriva il sistema di protezione di un S.O. (Punti 10).

6. Si descriva la tecnica dello spooling dell'I/O. (Punti 5).

3. Designazione e sincronizzazione per le primitive send/receive.

4. Discutere eventuali varianti ~~preemptive~~ degli algoritmi di scheduling.

5. Differenza tra sistemi operativi proprietari e standard: discutere problemi e vantaggi.

