Architettura dei sistemi operativi

Definizione di Processo

Si definisce processo ***l'istanza di un programma*** caricato in memoria centrale e in esecuzione su un certo insieme di dati. Ogni programma può generare più processi indipendenti (tramite le funzionefork o spawn).
Istanza significa allocazione dinamica (in continua evoluzione) di un programma, che viceversa è una entità statica.
Due istanze di word aperte su file differenti possono essere viste come due processi separati.
I termini ***task*** e ***job*** sono sinonimi di processo. Il ***processore*** è il dispositivo che esegue il processo.

Definizione di File

Archivio elettronico, cioè archivio di dati memorizzati su un supporto elettronico (memoria di massa).

Definizione di Risorsa

Si definisce risorsa di un computer un qualunque componente hardware e software che può essere utilizzato da un processo in esecuzione e che ne condiziona l'avanzamento (CPU, memoria, dispositivi di IO e relativi driver, ma anche, ad esempio, i files di un disco).

Definizione di Sistema Operativo (SO)

Il SO un software di base mirato alla gestione delle risorse hw e sw presenti un personal computer, facendo in modo che un utente possa utilizzare la macchina nel modo più semplice ed efficiente possibile.
Il SO, mediante un vasto insieme di funzioni e procedure, esegue una virtualizzazione della macchina, mostrando all'utente una macchina virtuale "più bella" e molto più semplice, i cui componenti sono simulati sulla base del calcolatore reale. Più in dettaglio si possono individuare i seguenti obiettivi fondamentali:

- ***Gestione delle risorse hardware della macchina*** mostrando all'utente *un'immagine astratta delle risorse disponibili*, nascondendo i dettagli dell'hardware ed esponendo un insieme di funzioni dette ***primitive*** (es le *API* di Windows) che il programmatore può utilizzare nei suoi programmi. Grazie all'astrazione realizzata dal SO, due macchine utilizzanti hardware differenti ma lo stesso SO sono viste dall'utente allo stesso modo. Questo costituisce il primo passo verso la *portabilità dei programmi*. Modifiche o upgrade sull'hardware(cambio di monitor, stampante, modem, HD) vengono "assorbite" dal SO senza modifica delle applicazioni.
- ***Gestione delle applicazioni utente***: dal caricamento dei programmi in memoria alla loro esecuzione, evitando che un processo possa accedere ad aree di memorie esterne rispetto a quelle di sua competenza, ed evitando eventuali conflitti fra processi che tentano di utilizzare una medesima risorsa (ad esempio devono entrambi accedere alla stampante o ad uno stesso file).
- ***Implementazione di una interfaccia utente semplice e funzionale***, attraverso la quale l'utente possa interagire con il SO (e quindi con la macchina) ed inviare i propri comandi all'interprete dei comandi del SO (in inglese ***shell***). L'interfaccia utente può essere
  - di tipo ***CLI*** (*Command Line Interface*) cioè a linea di comando, es DOS, sessioni TTY di Linux)
  - di tipo ***GUI*** (*Graphical User Interface*) cioè tale da consentire l'accesso al sistema tramite un insieme di elementi grafici (finestre, icone e menù) con il puntamento mediante mouse. Sono oggi sempre più ricche di funzionalità e user friendly, cioè facili da usare, anche da parte di personale non specializzato.
- ***Realizzazione di un meccanismo di protezione e sicurezza nell'accesso ai dati***: ogni utente deve poter accedere ai propri dati con la possibilità di concedere o negare l'accesso anche agli altri.

Poiché un SO non può conoscere a priori i dettagli hardware di ogni possibile dispositivo presente sul mercato, è indispensabile che ogni dispositivo sia dotato di un apposito software denominato ***device driver*** (*pilota del dispositivo*) e che il SO, in qualunque momento, possa essere ampliato con l'installazione di un nuovo driver. Il SO deve essere in grado di installare correttamente il driver in modo tale che non si creino conflitti con dispositivi preesistenti, assegnandogli ad esempio un numero di interrupt libero ed aggiornando la propria Interrupt Vector Table

Portabilità di un programma

Un software si dice portabile quando può essere compilato ed eseguito indipendentemente sia dalla piattaforma hardware sottostante, sia dal SO stesso.

- ***L'indipendenza dell'hardware*** viene assicurata dal SO: i software applicativi non comunicano con l'hardware ma con il SO

- ***L'indipendenza del SO*** è più difficile da realizzare. Occorre che i vari SO mettano a disposizione delle applicazioni client le stesse chiamate di sistema (detta ***System Call***), realizzate internamente in modo differente, ma che presentano al programma sorgente (testuale) la stessa firma. Il primo esempio di software portabile è stato ***JAVA*** che ha introdotto il concetto di un ulteriore substrato di interfacciamento fra Applicazione utente e SO, cioè la *Java Virtual Machine*. Per poter eseguire un programma JAVA su una certa macchina occorre installare sulla macchina la Java Virtual Machine relativa al SO in uso e che con esso si interfaccia. Dopo di che l'applicazione finale non comunica più con il SO, ma soltanto con la Java Virtual Machine, che gli espone le stesse primitive di interfacciamento in modo indipendente dal SO sottostante

Classificazione ed evoluzione dei SO

Sistemi Dedicati

I sistemi di elaborazione, nella loro struttura iniziale, erano in grado di gestire un solo processo per volta, completamente residente in memoria a cui erano destinate tutte le risorse a disposizione. Il SO ha lo scopo di gestire caricamento, inizializzazione e terminazione del programma, oltre che fornire supporto base per l'accesso all'hardware. Anche i primi Personal Computer erano sistemi dedicati (MS DOS), mono utente e single task.

Sistemi Batch Sequenziali

Fin dagli anni 60 apparve evidente che era inaccettabile che un sistema di elaborazione dal costo estremamente elevato fosse a disposizione di un unico utilizzatore che occupava tempo macchina con lunghe e onerose procedure di caricamento del programma. Nacquero così i primi sistemi batch, cioè sistemi di elaborazione a lotti, in cui i vari lavori (job) caricati tramite schede da utenti diversi (ciascuno con il proprio UID), venivano poi uno alla volta caricati in memoria centrale dal SO ed eseguiti in rigida sequenza.
Se un job andava in errore veniva terminato e si passava immediatamente al job successivo. L'utente non ha interazione con l'elaboratore. Pur essendo i sistemi batch ormai caduti in disuso, il termine batch è ancora oggi utilizzato per indicare una qualunque sequenza di comandi da eseguire uno dopo l'altro (file .BAT del DOS).

Sistemi Batch Multiprogrammati (multitask non preemptive)

In un sistema di elaborazione la risorsa più importante, sia come capacità di elaborazione sia come costo, è la CPU.
Nei sistemi batch il processore risultava decisamente sottoutilizzato in quanto doveva spesso fermarsi per attendere l'esecuzione delle operazioni di IO, infinitamente più lente rispetto ai tempi di elaborazione di CPU. Nacque così molto presto il concetto di ***multiprogrammazione*** che consiste nel caricare simultaneamente più programmi in memoria centrale, ciascuno all'interno di una sua ben precisa zona di competenza. Quando un processo si arresta per eseguire una operazione di IO viene avviato il processo successivo. Il processo sospeso potrà riprendere la propria attività nel momento in cui il nuovo processo termina o richiede a sua volta una operazione di IO.

- A livello di terminologia il termine ***task*** sostituisce il termine ***job***. Pur essendo sostanzialmente dei sinonimi, il termine job indica stretta sequenzialità, mentre task indica parallelismo.
- Il termine ***non preemptive*** significa che i sistemi batch multiprogrammati erano non prelazionali, nel senso che il SO non aveva la prelazione, cioè la facoltà, (il diritto) di interrompere un processo a meno che questo non decidesse spontaneamente di fermarsi per eseguire una operazione di IO.

Sistemi multitask time sharing

Si tratta di sistemi in grado di eseguire più task in parallelo, assegnando alternativamente ad ognuno di essi una porzione di tempo detta ***time slice***(*quanto di tempo*): Un task può sospendere la propria attività:

- Di sua volontà, perché deve eseguire una operazione di IO
- Contro la propria volontà, perché è scaduto il suo time slice.

Si tratta appunto di un sistema preemptive, in quanto il SO ha la prelazione (facoltà, diritto) di interrompere un processo dopo un certo tempo di esecuzione, anche contro la volontà del processo stesso. Grazie al time sharing, è come se ogni processo avesse una macchina virtuale completamente dedicata. Il Time Sharing nasce con i grandi mainframe degli anni 70 (ad esempio il VAX) in cui gli utenti erano collegati all'elaboratore centrale tramite semplici terminali utente. Poiché gli utenti operano con tempi molto più lunghi rispetto ai tempi diCPU, si crea nell'utente l'illusione di avere tutto il sistema a propria disposizione.
Il time sharing determina un notevole appesantimento del SO, che deve continuamente eseguire dei ***Context Switch*** da un processo all'altro, cioè:

- Memorizzare lo stato del processo corrente (in modo da poterlo poi riprendere da dove era arrivato)
- Caricare in CPU lo stato del nuovo processo, aggiornando le pipeline

Occorre inoltre gestire i problemi di condivisione delle risorse disponibili, gestendo eventuali conflitti che possonosorgere per l'assegnazione di una risorsa a diversi processi in esecuzione parallela

Concetto di System Overhead

Nel time sharing il tempo di CPU non è più interamente utilizzato per l'esecuzione dei programmi utente, ma viene ripartito fra questi e le routine del SO. Cioè parte del tempo di CPU viene "sprecato" per l'esecuzione del sistema operativo anziché dedicato all'esecuzione dei programmi utente, diminuendo l'efficienza del processore. Il ***SystemOverhead*** rappresenta appunto la percentuale di tempo di CPU utilizzato per l'esecuzione del SO rispetto al tempo utilizzato per l'esecuzione dei processi utente.

Tipologie di sistemi operativi

I sistemi operativi possono essere classificati nel modo seguente:

- ***single-user, single task***: il sistema prevede un unico utente e una sola attività in esecuzione (task);
- ***single-user, multi-tasking***: è il tipico sistema operativo di personal computer nel quale un singolo utente può interagire con il sistema eseguendo però diversi programmi "contemporaneamente" come, per esempio, la scrittura di un testo con Word e la creazione di una tabella con Excel. Anche se un PC connesso in rete è "utilizzato" da molti utenti, in realtà fornisce semplicemente servizi agli utenti remoti;
- ***multi-user***: consente a più utenti (anche molte migliaia) di eseguire programmi contemporaneamente. Di norma ogni utente utilizza un proprio terminale, ma è possibile anche operare in multiutenza da un solo terminale alternandolo tra diversi utenti. Un SO multiuser consente la condivisione delle risorse del computer tra più utenti e quindi deve gestire un'equa attribuzione delle risorse, il coordinamento dell'utilizzo delle stesse e la protezione delle attività degli utenti e del sistema da reciproche interferenze. UNIX è un esempio di SO multiutente; mentre per quanto riguarda i Personal Computer, DOS è single user mentre Windows a partire dalla versione XP è diventato multi user.
- ***multitasking***: detto anche *multiprocessing*, consente l'esecuzione di diversi programmi contemporaneamente;
- ***multithreading***: consente l'esecuzione concorrente di parti diverse dello stesso task (*processo*);
- ***multiprocessor***: consente l'esecuzione di un programma su più di una CPU. Un SO ***AMP*** (*Asymmetric Multi Processing*) utilizza una CPU per sé e tutte le altre per i processi di utente. Un SO ***SMP*** (*Symmetric Multi Processing*) utilizza invece tutte le CPU per distribuire qualsiasi tipo di carico di lavoro;
- ***real time***: risponde agli input in tempi molto rapidi e conosciuti (tempo di latenza massimo noto), comunque adeguati a controllare il processo esterno. Questi SO sono detti ***RTOS*** (*Real Time Operating System*) e sono adatti al controllo di macchinari industriali e strumentazione scientifica. Hanno una limitata interfaccia utente, poche o nessuna utility e in genere un ambiente di sviluppo spartano;
- ***batch processing***: è un SO che esegue una serie di lavori (job) sequenzialmente. Si tratta della modalità tipica dei primi mainframe, tuttora adottata come possibile modalità per eseguire lavori che non necessitano di interazione di tempi di risposta brevi. In effetti il batch processing consentiva all'operatore del centro di calcolo di organizzare i lavori in modo da sfruttare al massimo il mainframe, per esempio accorpando assieme tutti i job che richiedevano un particolare compilatore; un'altra ottimizzazione tipica era quella di far eseguire alcuni job di notte, quando non c'erano particolari urgenze. Il batch processing non è interattivo: l'utente lasciava il job sotto forma di un pacco di schede perforate e tornava il giorno successivo a prendere i tabulati, ovvero le stampe con i risultati. Se il programma falliva e andava in halt, al più era prodotto un *PDM* (*Post Mortem Debug*) dai cui tabulati il programmatore cerca di risalire alle cause degli errori;
- ***time-sharing***: con la diffusione dei terminali "stupidi" (monitor, tastiera e limitate capacità di calcolo per i protocolli di comunicazione) e delle prime reti di calcolatori, sì aprì il campo all'elaborazione interattiva; molti utenti da terminali diversi possono richiedere l'esecuzione dello stesso programma o di programmi diversi e ottenere risposte in tempo ragionevole per lo svolgimento di compiti di ufficio. Si pensi per esempio al sistema di prenotazione di posti delle linee aeree, certamente impossibile con il batch processing. La soluzione fu il time-sharing, grazie al quale un singolo computer dedica la propria CPU a intervalli di tempo a ciascun utente, cosicché questo dispone di un proprio "calcolatore virtuale". Questa soluzione fu resa possibile dallo sfruttamento dei "tempi morti" di I/O e dall'enorme potenza di calcolo delle CPU, certamente sovradimensionata per un singolo utente e che poteva essere quindi assegnata a intervalli di tempi servendosi di un'interruzione periodica dal real time clock del sistema. Il time-sharing è correlato al multitasking in quanto un'unica CPU esegue in modo concorrente diversi processi: nel caso del multitask non devono esserci necessariamente più utenti, ma può capitare che una sola applicazione software sia frazionata in più processi;
- ***transaction processing***: per transazione si intende un insieme di operazioni logiche che devono o avvenire tutte con successo o tutte fallire. I SO progettati per questo scopo si dicono *transaction processing*. Un esempio può essere

un'operazione di prelievo Bancomat, nel corso della quale sarebbe molto grave che l'addebito in conto fosse effettuato senza erogazione del contante dallo sportello.

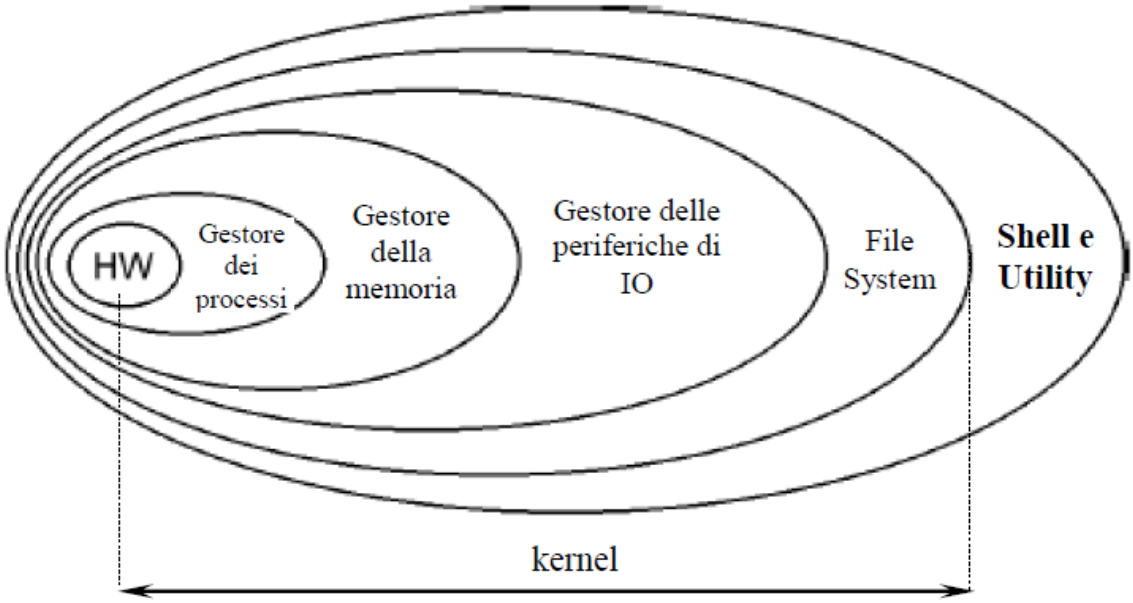
Il Kernel di un SO

Un SO in prima analisi può essere suddiviso in due parti principali:

- Il ***kernel*** (*nocciolo*), che costituisce il cuore del SO. Esso comunica direttamente con l'hardware e si occupa della gestione dei processi fornendo loro un accesso sicuro e controllato alle risorse hardware della macchina. Il kernel viene caricato in memoria centrale al momento dello start up della macchina e vi risiede per tutta la durata del funzionamento del PC.
- La ***shell dei comandi***, che funge da interfaccia tra kernel e utente, consentendo all'utente di poter lanciare in esecuzione allo stesso modo qualsiasi tipo di programma, anche programmi molto differenti fra loro. La shell può essere di due tipi: ***CLI*** o ***GUI***. Nel caso del CLI il SO presenta un semplice PROMPT in corrispondenza del quale l'utente digita i comandi in modalità testuale. Nel caso della GUI l'utente interagisce in modo molto più *user-friendly* con icone e immagini visualizzate sullo schermo. La maggior parte dei SO mette a disposizione entrambe queste interfacce. Nel caso di Linux l'interfaccia GUI è detta ***DesktopManager*** (es *Gnome*, *KDE*, *OpenBox*). UNIX è stato il primo SO a introdurre il concetto di interfaccia utente(shell) come processo esterno al kernel, con la possibilità di cambiare shell senza dover ricompilare il kernel.
- Insieme alla shell, il SO rende di solito disponibili anche un insieme di ***utility aggiuntive*** come un editor di testi(es *vi* nel CLI o *gedit* nella GUI). Nei sistemi Unix/Linux anche il compilatore C fa parte delle Utilità del SO

Il modello OnionSkin

Modello proposto nel 1983 da H. Deitel. Fornisce una rappresentazione modulare gerarchica (*Onion Skin*, cioè *a buccia di cipolla*) su come dovrebbe essere strutturato il kernel di un SO, cioè suddiviso in 4 moduli che rappresentano le quattro principali funzionalità del SO :



- Il ***Gestore dei Processi*** si occupa di creare e cancellare i processi utente e i processi di sistema; mantenere aggiornato lo stato dei vari processi e del processore; decidere a quale processo assegnare il processore e per quanto tempo. Gestisce inoltre la comunicazione fra i vari processi. Al sopra di questo livello i processi vedranno ciascuno un proprio processore virtuale completamente dedicato.
- Il ***Gestore della Memoria*** si occupa di mantenere uno stato della memoria; assegnare ai vari processi un apposito spazio in memoria (assicurandosi che i vari processi non possano, per errore o per volontà, andare ad operare nelle aree di memoria dedicate ad altri processi o al SO); liberare spazio in memoria quando è piena. Al di sopra di questo livello gli altri livelli vedranno un'unica area di memoria completamente dedicata al processo. Se un processo tenta di accedere ad un'area esterna a quella di sua competenza viene generato un interrupt di sistema (*trap*).
- Il ***Gestore delle Periferiche*** si occupa di mantenere aggiornato lo stato di tutte le periferiche e dei dispositivi di controllo; virtualizzare le risorse, facendo in modo che ogni processo veda la risorsa tutta per se; gestire il mapping tra risorsa reale e risorsa fisica. In realtà il processo dispone di risorse virtuali che il SO simula servendosi delle risorse del calcolatore reale. Ad esempio la tecnica di ***spool*** (*Simultaneous Peripheral Operations On-line*) consente al SO di

scaricare temporaneamente su disco i dati destinati alla stampa, gestendo così una coda di stampa condivisibile fra più processi. Un apposito modulo si occupa di inoltrare progressivamente la coda di stampa alla stampante.

- Il ***Gestore del File System*** si occupa della gestione dei dischi. Deve mantenere traccia di tutti i file e directory memorizzati, cioè la loro dislocazione, la loro lunghezza, nonché i diritti di accesso su di essi; gestire l'assegnazione degli archivi ai vari processi; supportare primitive per la manipolazione di file e cartelle.

Nella gerarchia Onion Skin il centro è costituito dall'hardware ed ogni livello comunica soltanto con i livelli adiacenti. Compito di ogni livello è quello di fornire servizi al livello superiore ed è visibile dal livello superiore soltanto attraverso un insieme ben definito di funzioni dette ***primitive***.
Il pregio fondamentale del modello Onino Skin è quello di individuare i 4 moduli fondamentali di un SO. L'elevata modularità del modello Onion Skin garantisce una notevole robustezza al SO che potrebbe facilmente essere portato su macchine differenti sostituendo soltanto i moduli più interni che operano sull'hardware.
D'altro canto, il modello Onion Skin risulta estremamente carente dal punto di vista delle prestazioni. Se, ad esempio, la shell necessita di servizi forniti dal Gestore dei Processi deve passare diversi confini. L'attraversamento dei vari confini si traduce ovviamente in elevati tempi di risposta, in contrasto con il fatto che il SO dovrebbe offrire un ambiente per l'esecuzione delle applicazioni il più efficiente e rapido possibile. Tutto ciò che si contrappone a questa esigenza è detto *sovraccarico* (***overhead***). In una struttura Onion Skin l'overhead è massimizzato: elevato overhead significa bassa efficienza.

Interrupt e System Call

Nei SO più datati non esisteva il concetto di System Call. I processi comunicavano con il SO soltanto attraverso i cosiddetti Interrupt Software. Nei SO più moderni gli Interrupt Software entrano a far parte di un più ampio gruppo di procedure messe a disposizione dal SO denominate ***System Call***, cioè *chiamate di sistema*, che il processo utente può utilizzare per richiamare una primitiva del kernel.
Le system call rappresentano lo strumento principale attraverso cui un processo utente comunica col kernel il quale a sua volta comunica con l'hardware (virtualizzazione dell'hardware). Ad esempio se si vuole stampare un file, invece che inviare i comandi fisici al gestore dell'HD per la ricerca del file e poi i relativi comandi fisici alla stampante, è sufficiente fare una System Call ad una apposita primitiva del SO indicandogli quale file stampare e su quale stampante.
Il termine Interrupt è ora riservato soltanto più per gli interrupt hardware, generati dai dispositivi esterni per avvisare la CPU riguardo alla terminazione di una certa operazione. In corrispondenza del sopraggiungere dell'interrupt, il SO avvia la ***Routine di Risposta all'Interrupt*** (*RRI*), detta normalmente ***interrupt handler.***

Gli stati di un processore: Kernel Mode e User Mode

Tutti i processori Intel con architettura IA32 (a partire dal 80386 fino ai processori attuali) sono in grado gestire 4livelli di protezione delle istruzioni (*Kernel Mode*, *System Services*, *OS Extension*, *User Mode*), cioè in sostanza ad ogni istruzione è assegnato un livello di protezione e può essere eseguita soltanto da processi aventi un livello di protezione uguale o superiore a quello dell'istruzione. Scopo di queste protezioni è quello di impedire o consentire l'accesso a particolari aree di memoria in base allo stato del processore.
A tal fine le istruzioni del processore sono state suddiviso in due categorie: le ***istruzioni standard***, eseguibili da chiunque in ogni momento, e le ***istruzioni privilegiate*** che interagiscono con il sistema (ad es le istruzioni IN e OUT) che possono essere eseguite soltanto in particolari condizioni.
In corrispondenza si possono individuare 2 stati del processore: uno *stato utente* (***user mode***) in cui è consentita soltanto l'esecuzione delle istruzioni standard, ed uno stato *supervisore* (***kernel mode***) con diversi livelli di privilegio in cui è consentita qualunque istruzione.
Un processo utente, normalmente, viene avviato con un livello di protezione pari a "user mode". Nel momento in cui il processo esegue una System Call al SO, il SO provvede ad eseguire un ***Mode Switch*** (*cambio di modo*) del processore elevandolo dal livello user mode al livello kernel mode (modalità privilegiata), avviando il processo di sistema richiesto. Terminato il processo di sistema, il SO provvede a riportare il processore in stato utente e assegnarlo al processo utente: in questo modo solo i processi di sistema possono utilizzare le istruzioni privilegiate.
Esempio tipico di istruzioni privilegiate sono le istruzioni ***IN*** e ***OUT*** che possono essere eseguite soltanto in kernel mode in modo da evitare che il programma utente possa eseguire un accesso diretto e incontrollato all'hardware della macchina. Per cui, per poter eseguire le istruzioni IN e OUT, il programma utente dovrà necessariamente eseguire una System Call ai driver del SO i quali, soltanto loro, potranno eseguire le istruzioni IN e OUT sui registri di IO.

Sistemi Operativi monolitici

Un SO monolitico è composto da un insieme di moduli e procedure tutte compilati insieme e caricate all'interno dello stesso spazio di memoria. Ogni funzione ha una ben definita interfaccia (in termini di parametri e risultati) e può indifferentemente chiamare tutte le altre in qualsiasi momento ne abbia bisogno.
Per costruire un sistema operativo monolitico occorre compilare tutti i vari moduli e poi collegarli insieme mediante il linker in un unico file eseguibile di sistema. Anche se ogni modulo è normalmente separato dal resto (programmazione modulare), l'integrazione del codice è comunque molto stretta. Il programma utente può accedere ai servizi forniti dal SO tramite speciali istruzioni di trap, le System Call, che cambiano la modalità della macchina da user mode a kernel mode trasferendo il controllo al sistema operativo.
I parametri da passare al SO vengono posizionati in locazioni ben definite (tipicamente i registri di CPU o lo stack).
Quando l'implementazione del sistema è completa e sicura, la stretta integrazione interna dei componenti rende un buon kernel monolitico estremamente efficiente e veloce. I difetti principali riguardano il fatto che, siccome tutti i moduli operano nello stesso spazio di memoria, un bug in uno di essi può bloccare l'intero sistema e, soprattutto, non è possibile aggiungere un nuovo dispositivo hardware senza aggiungere il relativo modulo al kernel, operazione che richiede la ricompilazione del kernel.
Un esempio di sistema monolitico è il kernel di Unix.

Microkernel e modello client-server

L'approccio microkernel consiste nel definire delle macchine virtuali molto semplici sopra l'hardware, con un set di primitive per implementare servizi minimali quali semafori, gestione dei thread, spazi di indirizzamento o comunicazione inter-processo, spostando quanto più possibile il codice di sistema verso i livelli superiori con un kernel minimale. L'approccio consueto consiste nell'implementare molti servizi di sistema all'interno di processi esterni al kernel. Per richiedere un servizio, come la lettura di un blocco di un file, un processo utente (chiamato ora ***processo client***) invia la richiesta a un processo server esterno al kernel che effettua il lavoro e ritorna la risposta.
In questo modello il kernel, dovendo solo gestire la comunicazione tra processi client e server, diventa estremamente leggero e si limita a svolgere la funzione di Gestore dei Processi, demandando gran parte delle altre funzionalità a moduli esterni che vengono caricati in memoria soltanto al momento della necessità (soprattutto i driver). Inoltre un servizio non funzionante non provoca il blocco dell'intero sistema, ma il singolo servizio può essere riavviato indipendentemente dal resto.
Un altro vantaggio del modello client-server è la sua adattabilità all'utilizzo in sistemi distribuiti. Se un client comunica con un server inviandogli messaggi, il client non ha bisogno di sapere se il suo messaggio viene gestito localmente sulla macchina o se viene inviato ad un processo server su una macchina remota. I processi server possono girare anche su macchine con altri SO dal client. Unica differenza è che nei sistemi distribuiti il kernel non può occuparsi solo del traffico dei messaggi ma deve saper gestire la comunicazione con altri nodi della rete.
Sia Unix che Linux utilizzano sostanzialmente un kernel monolitico, molto più modulare nel caso di Linux, scritti entrambi prima che fosse dimostrato che i microkernel puri potevano comunque avere performance elevate confrontabili con quelle dei kernel monolitici. Comunque è tutt'ora aperta la disputa sul fatto che sia migliore il sistema monolitico o il microkernel. Il progetto di Linux nato come kernel monolitico anziché come microkernel è stato uno degli argomenti della famosa guerra di religione fra Linus Torvalds (creatore di Linux) e Andrew Tanenbaum (celebre docente di SO, autore di Minix).
In ogni caso i kernel monolitici più moderni come il Kernel Linux e FreeBSD possono caricare dei moduli in fase di esecuzione, a patto che questi fossero previsti in fase di compilazione, permettendo così l'estensione del kernel quando richiesto e mantenendo al contempo le dimensioni del codice nello spazio del kernel al minimo indispensabile. Non si tratta più di un kernel monolitici puri ma sostanzialmente di ***kernel ibridi***.
I kernel ibridi sono essenzialmente dei microkernel che hanno del codice "non essenziale" al livello di spazio del kernel in modo che questo codice possa girare più rapidamente che se fosse implementato ad alto livello, compromesso adottato da molti sviluppatori di SO, in particolar modo Windows che introduce il concetto di ***DLL***.

I processi

Process Control Block

Un processo in un sistema operativo è rappresentato da una struttura dati detta PCB (Process Control Block) o descrittore di processo. Il PCB contiene in genere le seguenti informazioni:

Contesto volatile:
- program counter;
- area per il salvataggio dei registri,
- area salvataggio registro di stato;

Process kernel data:
- stato corrente di avanzamento del processo (ready, running, waiting ecc.);
- identificatore unico del processo;
- un puntatore al processo parent;
- puntatori ai processi child, se esistenti;
- livello di priorità;
- informazioni per il memory management (in particolare memoria virtuale) del processo;
- identificatore della CPU su cui è in esecuzione;
- informazioni per lo scheduling del processo, come il tempo di run o wait accumulato;
- informazioni di accounting del processo;
- signal ed eventi pendenti;
- informazioni sullo stato di I/O del processo: open file, socket ecc.

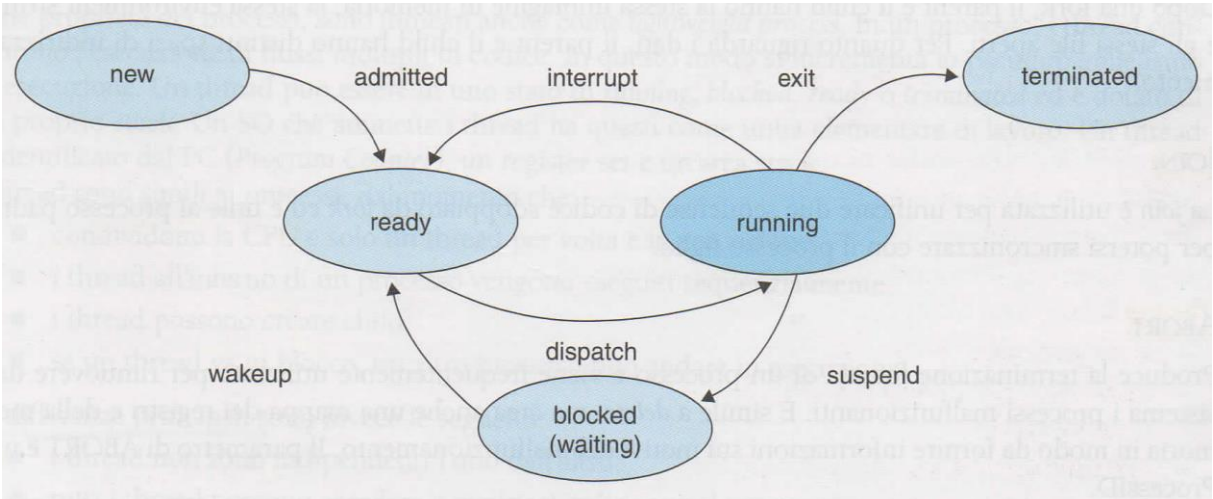
Il PCB è quindi la struttura dati mediante la quale il SO gestisce un processo e di fatto ne rappresentalo stato globale. La gestione degli stati di un processo avviene principalmente tramite opportune listedi PCB in memoria centrale

Stati di un processo

Lo stato di un processo consiste in tutte le informazioni necessarie per riprenderne l'esecuzione dopo una temporanea sospensione, durante la quale la CPU è assegnata a qualche altro processo.
Un processo evolve attraverso una serie di stati di avanzamento discreti:

- ***new***: stato iniziale dopo la creazione;
- ***ready***: un processo si dice ready se è pronto per essere eseguito ma la CPU non è disponibile;
- ***running***: quando un processo ha assegnata la CPU;
- ***blocked*** (*waiting*): quando un processo si blocca significa che non può continuare ad avanzare poiché è in attesa (waiting) di una risorsa al momento non disponibile o di un evento su cui sincronizzarsi come, per esempio, il completamento di un'operazione di I/O;
- ***terminated***: il processo ha terminato l'esecuzione.

Un ulteriore stato è ***suspended***: il processo comanda a se stesso o a un altro processo di sospendersi(*sleep*) in attesa della segnalazione di un evento. Esso sarà "risvegliato" quando un altro processo effettuerà una procedura di ***wakeup*** (*risveglio*).
Le sei possibili transizioni tra i cinque stati suddetti sono le seguenti:



- ***Admitted***(*New → Ready*): si verifica quando viene creato un processo.
- ***Dispatch*** (*Ready → Running*): avviene quando un processo deve andare in esecuzione.
- ***Interrupt*** (*Running → Ready*): avviene quando lo scheduler decide che il processo running ha esaurito il quanto di tempo a lui assegnato, o nel caso sia applicato lo schema detto preemption e lo scheduler deve cedere la CPU a un processo a più alta priorità entrato in ready.
- ***Suspend***(*Running →Block*): avviene quando un processo non può continuare l'esecuzione. Per esempio, se un processo inizia un'operazione di I/O, può rilasciare volontariamente la CPU, oppure si può autosospendere in attesa di un timeout.
- ***Wakeup*** (*Blocked → Ready*): avviene quando si genera l'evento esterno su cui era in attesa il processo.
- ***Exit*** (*Running →Terminated*): si verifica quando un processo termina l'esecuzione.

Operazioni sui processi

Create

Nei sistemi general-purpose occorre un meccanismo per creare processi durante il funzionamento del sistema. I principali eventi che determinano la creazione di un processo sono i seguenti:

- inizializzazione del sistema;
- un processo in esecuzione invoca una system call creation;
- un utente richiede la creazione di un nuovo processo interattivo;
- viene attivato un job batch.

Un processo può essere creato, per esempio, quando un utente effettua il login o esegue un programma. Il SO crea processi per fornire servizi, per esempio la stampa. Un processo in esecuzione può quindi chiedere la creazione di altri processi per realizzare servizi particolari.
I parametri di Create sono il *ProcessID* e i *ProcessAttribute*.

Fork

Un processo può creare un nuovo processo con procedure dette fork. Il processo creatore viene detto ***parent*** (*genitore*) mentre quello creato è chiamato ***child process*** (*figlio*). Questo procedimento di creazione determina una struttura gerarchica di processi, nella quale un parent può avere diversi child, ma un child può avere un unico parent.
Dopo una Fork, il parent e il child hanno la stessa immagine in memoria, la stessa environment string e gli stessi file aperti. Per quanto riguarda i dati, il parent e il child hanno distinti spazi di indirizzamento.

Join

La Join è utilizzata per unificare due sequenze di codice sdoppiato da fork ed è utile al processo padre per potersi sincronizzare con il processo figlio.

Abort

Produce la terminazione forzata di un processo e viene frequentemente utilizzata per rimuovere dal sistema i processi malfunzionanti. È simile a delete, ma crea anche una mappa dei registri e della memoria in modo da fornire informazioni sui motivi del malfunzionamento. Il parametro di Abort è un *ProcessID*.

Suspend

Un processo può sospendere se stesso o un altro processo, in base ovviamente al suo livello di privilegio e di priorità. Il parametro di Suspend è un *ProcessID*.

Resume

Questa system call riattiva il processo indicato come parametro (*ProcessID*). Assieme a suspend implementa un meccanismo di sincronizzazione.

Delay

I parametri sono un *ProcessID* e un tempo. In genere, Delay può essere applicata dal processo a sé stesso o a un altro processo, in relazione ai livelli di privilegio e priorità. Con questa system call si gestiscono i timeout, ovvero l'attesa che accada un certo evento, in mancanza del quale il processo riprende il controllo.

Get_attribute

Riceve come parametri un *ProcessID* e una structure in grado di contenere gli attributi del processo.

Terminate

Un processo termina quando finisce di eseguire la sua ultima istruzione. Le risorse tornano al sistema(memoria, I/O), il processo è purgato dagli elenchi di servizio del SO e il suo Process Control Block è cancellato. La terminazione può essere di diversi tipi, elencati di seguito:

- ***Normal exit***: molti processi terminano quando hanno regolarmente completato il loro compito.
- ***Error exit***: se un processo rileva un errore parametrico come, per esempio, un compilatore che verifica la non esistenza del file sorgente.
- ***Fatalerror***: un errore grave dovuto a un bug nel codice come, per esempio, l'esecuzione di un'istruzione illegale, il riferimento a una locazione di memoria non accessibile o una divisione per zero.
- ***Killed by another process***: un processo esegue una system call chiedendo al sistema operativo di terminare un altro processo.

Thread

Nonostante il fatto che un thread deve essere eseguito nel contesto di un processo, il processo e i thread associati sono entità differenti. I processi sono utilizzati per raggruppare e isolare risorse, mentre i thread sono le entità schedulate per l'esecuzione da parte della CPU.
Un thread è una ***single sequence stream*** all'interno di un processo. Dal momento che i thread hanno alcune proprietà dei processi, sono indicati anche come ***lightweight process***. In un processo, i thread consentono l'esecuzione di flussi multipli di codice. In questo modo si incrementa lo pseudoparallelismodi esecuzione.
Un thread può essere in uno stato di *running*, *blocked*, *ready* o *terminated* ed è dotato di un proprio stack. Un SO che ammette i thread ha questi come unità elementare di lavoro. Un thread è identificato dal PC (Program Counter), un register set e un'area stack.
I thread sono simili ai processi, dal momento che:

- condividono la CPU e solo un thread per volta è in esecuzione;
- i thread all'interno di un processo vengono eseguiti sequenzialmente;
- i thread possono creare child;
- se un thread va in blocco, un altro processo può andare in esecuzione.

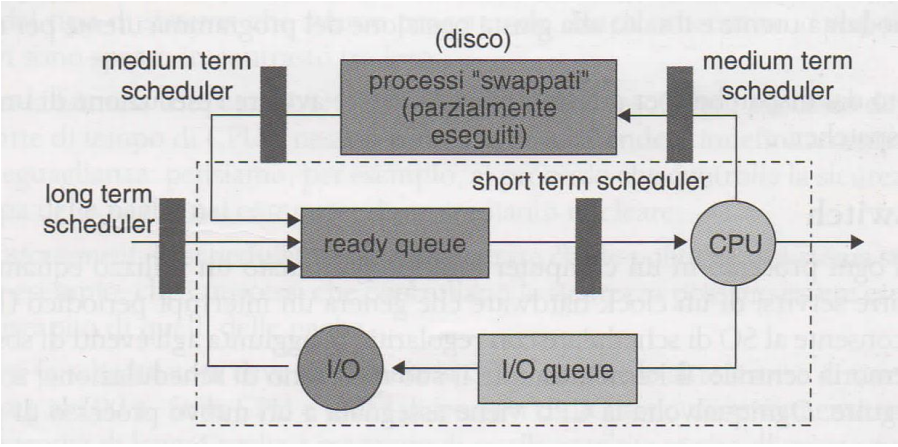
Le differenze principali sono invece le seguenti:

- i thread non sono indipendenti l'uno dall'altro;
- tutti i thread possono accedere a qualsiasi indirizzo nel processo;
- i thread sono progettati per cooperare l'uno con l'altro;
- non vi è alcuna protezione tra i thread. Per esempio un thread può sovrascrivere lo stack di un altro thread o alterare dati condivisi anche se la filosofia di base è quella della cooperazione di thread all'interno di un processo.

Scheduler e dispatcher

Livelli di scheduling

Lo scheduling può operare fino a tre livelli:



- ***long term scheduling***: regola l'accesso dei programmi, di norma i job batch, al sistema per l'esecuzione; controlla quindi il grado di multiprogrammazione cercando di massimizzarlo anche regolando il mix di processi CPU-bound e I/O bound. È importante ricordare che con il termine job batch intendiamo un insieme di informazioni che codificano tutti i

dati, i programmi e i comandi di sistema necessari per svolgere completamente l'esecuzione senza necessità di interazione;

- ***medium term scheduling***: solo un sottoinsieme dei processi viene mantenuto in memoria, mentre i rimanenti sono trasferiti su disco (swapped). Vengono rimossi i processi che sono stati in memoria centrale per un tempo sufficientemente elevato per caricare un nuovo insieme di processi dal disco (swapping). Si tratta di un livello connesso alla gestione della memoria virtuale e quindi è un misto di scheduling di CPU e memory management;
- ***short term scheduling***: è il normale scheduler di CPU dei processi residenti in memoria centrale; spesso si identifica con il dispatcher. È invocato su ogni evento che può portare a eseguire un context switch come l'interruzione *RTC* (*Real Time Clock*), interruzioni I/O, system call e trap, signal. Nel prosieguo della trattazione con il termine scheduler indicheremo lo *short term scheduler*.

Scheduler

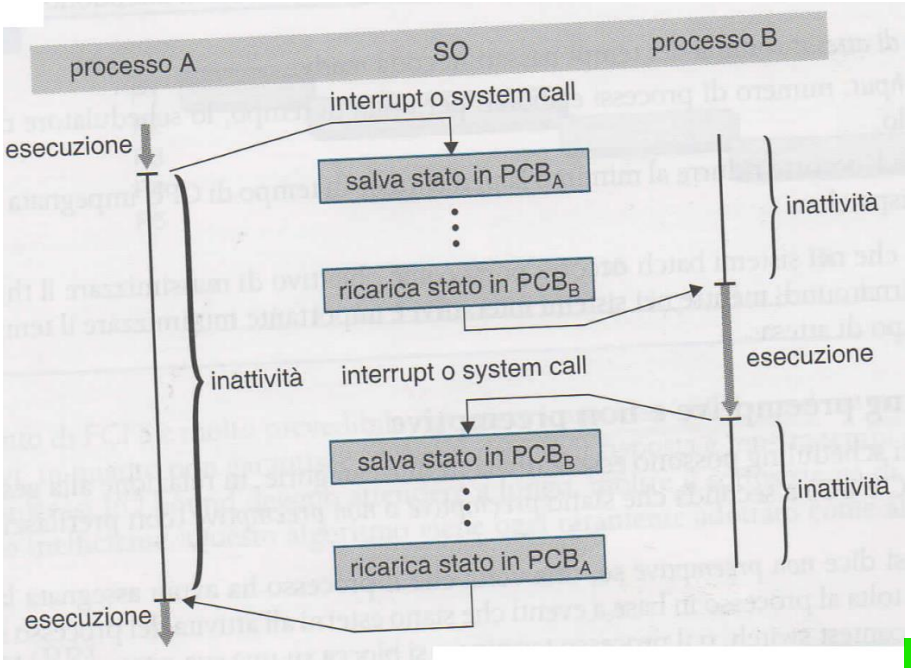
Lo scheduling della CPU (*short term sheduler*) consiste nella scelta, qualora sia necessario cambiare il processo in esecuzione, di un nuovo processo cui assegnare la CPU. L'effettiva assegnazione della CPU al processo prescelto è effettuata dal ***dispatcher***. Scheduler e dispatcher agiscono sulle code di attesa dei processi, i cui elementi sono i PCB
Quando più di un processo si trova nello stato ready, lo scheduler decide quale tra questi processi debba andare in esecuzione. Gli algoritmi utilizzati da questo componente si dicono algoritmi di schedulazione. Lo scheduling della CPU è alla base dei sistemi multiprogrammati: più processi sono mantenuti in memoria e la CPU è assegnata loro dinamicamente.

Dispatcher

Il dispatcher è il modulo del kernel del SO che passa effettivamente il controllo della CPU ai processi scelti dallo scheduler a breve termine; effettua il cambio di contesto (***context switch***) di elaborazione, il passaggio alla modalità utente e il salto alla giusta posizione del programma utente per riavviarne l'esecuzione. Il tempo richiesto dal dispatcher per arrestare un processo e avviare l'esecuzione di un altro definisce la latenza del dispatcher.

Context switch

Per assicurare a ogni processo in un computer multiprogrammato un utilizzo equamente condiviso della CPU, occorre servirsi di un clock hardware che genera un interrupt periodico (***RTC***, *Real TimeClock*). Questo consente al SO di schedulare con regolarità, in aggiunta agli eventi di sospensione, tuttii processi in memoria centrale: il kernel, tramite il suo algoritmo di schedulazione, sceglie un nuovo processo da eseguire. Ogniqualvolta la CPU viene assegnata a un nuovo processo di parla di *context switching*.
I valori dei registri sono salvati nel PCB del processo che viene sospeso e i registri sono caricati con i nuovi valori prelevati dal PCB del nuovo processo. In un sistema monoprocessore multiprogrammato, il context switch occorre così di frequente che tutti i processi sembrano essere eseguiti in parallelo.



Il context switch tra processi è relativamente lento. Prima che un processo possa essere switched, il suo PCB deve essere salvato dal SO, dopodiché occorre caricare il nuovo PCB. Queste sono operazioni che richiedono non poco tempo. Il context switch tra processi è ulteriormente rallentato dallo svuotamento della pipeline e della cache del microprocessore.

La commutazione dei thread è invece meno costosa e molto veloce: i thread dello stesso processo condividono molte risorse e riguarda solo registri, stack e program counter.

Scheduling preemptive e non preemptive

Gli algoritmi di scheduling possono essere divisi in due categorie, in relazione alla gestione delle interruzioni (RTC e I/O) a seconda che siano ***preemptive*** o ***non preemptive*** (con *prerilascio* o *senza prerilascio*).
Lo scheduling si dice ***non preemptive*** se, una volta che il processo ha avuta assegnata la CPU, questa non può essere tolta al processo in base a eventi che siano esterni all'attività del processo stesso; quindi, per attivare un contest switch, o il processo termina o si blocca su una sua operazione di I/O o comunque sull'attesa per acquisire una risorsa condivisa.
Con questa tecnica, i processi piccoli spesso devono attendere il completamento di processi lunghi. In compenso, i tempi di risposta sono più prevedibili in quanto i processi lunghi non sono disturbati da processi piccoli ad alta priorità. Uno scheduler non preemptive interviene quando il processo passa dallo stato run a quello wait oppure termina.
Uno scheduler è ***preemptive*** se un processo, dopo aver avuto assegnata la CPU, può averla tolta in base a eventi estranei all'attività del processo stesso e quindi può essere sospeso anche se ha tutte le risorsefunzionalmente necessarie per proseguire. Uno scheduler preemptive interviene non solo quando il processo passa dallo stato run a quello wait, oppure termina, ma viene richiamato anche da RTC o da un event handler di interrupt e comunque interviene se in coda ready si inserisce un processo a priorità maggiore di quello in esecuzione.

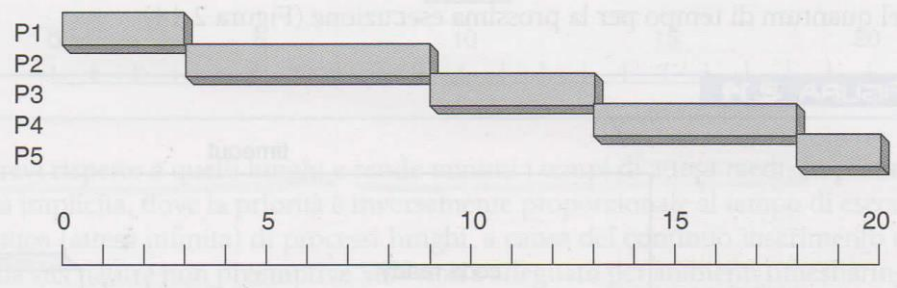
Algoritmi di schedulazione

Per valutare meglio gli algoritmi di schedulazione, supponiamo di avere i processi P1, P2, P3, P4 e P5come nella tabella seguente. Con tempo di servizio si intende il tempo totale di CPU necessario per eseguire il processo senza interruzioni.

Processo	Tempo di arrivo	Tempo di servizio
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

FIRST-COME-FIRST-SERVED (FCFS)

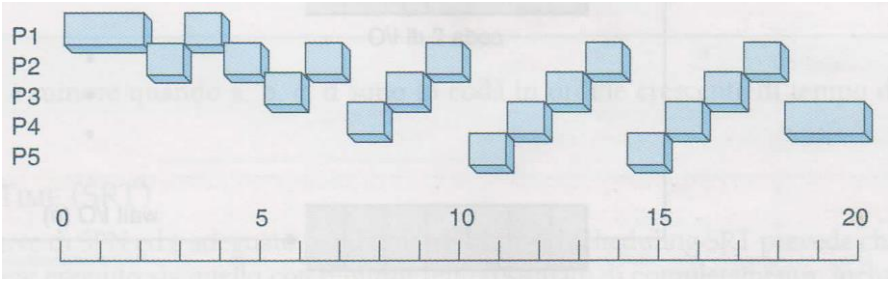
Viene detto anche FIFO (First-In-First-Out) o Run-to-Completion o Run-Until-Done. I processi sono schedulati in base al tempo di arrivo nella coda dei ready. È un algoritmo non preemptive e quindi un processo a cui viene assegnata la CPU viene eseguito finché non viene completato o si pone in attesa di una risorsa condivisa o si sospende sincronizzandosi sulla fine di un'operazione di I/O. Si tratta di un algoritmo teoricamente equo, ma in realtà non lo è in quanto un processo lungo e poco importante potrebbe far attendere molto un processo breve e importante.



Il comportamento di FCFS è molto prevedibile, favorisce i processi CPU bound ma non è adatto per utenti interattivi, in quanto non garantisce un buon tempo di risposta e mostra tempi di attesa medi elevati. Brevi processi I/O bound devono attendere a lungo, inoltre il sottosistema di I/O viene utilizzato in modo inefficiente. Questo algoritmo viene oggi raramente adottato come algoritmo principale.

ROUND ROBIN (RR)

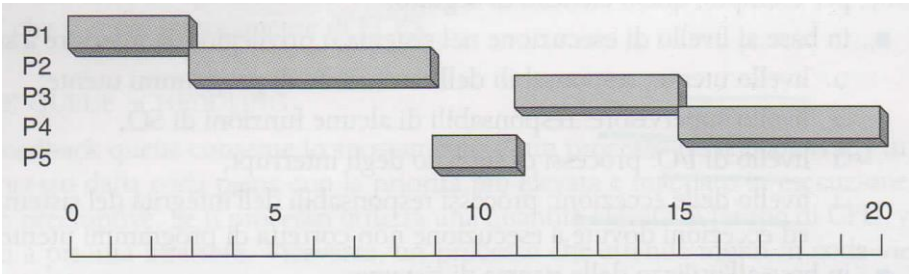
Nello scheduling Round Robin (RR) i processi sono schedulati in modo simile al FCFS, ma a ciascuno è assegnato un intervallo di tempo di esecuzione limitato detto ***time-slice*** o ***quantum***. Se un processo non termina o si sospende prima dello scadere del quantum, lo schedulatore è richiamato tramite RTC, la CPU è preempted e assegnata al processo successivo che attende nella coda ready. Il processo preempted è posto alla fine della coda ready, che è di tipo circolare.



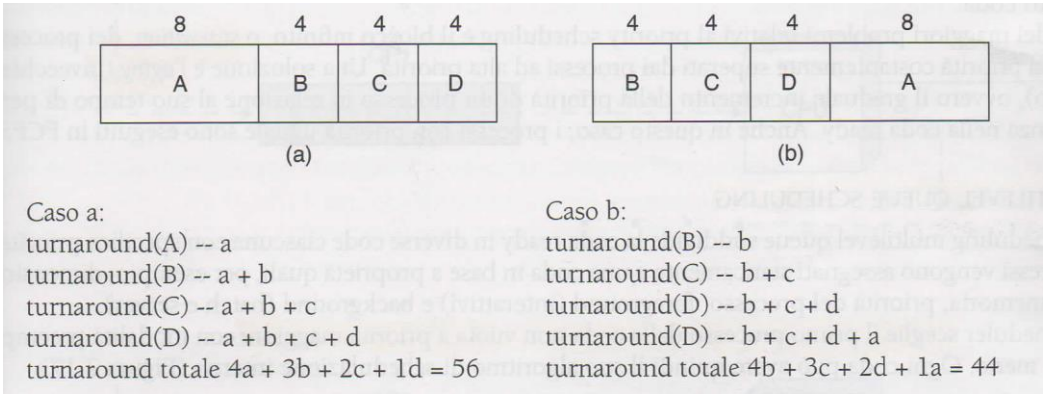
Lo scheduling Round Robin è preemptive (solo in relazione alla fine del time-slice) e quindi è adatto nei sistemi time-sharing nei quali è importante un basso tempo di risposta per gli utenti interattivi.
Ovviamente risulta importante definire il valore del quantum. Se è troppo corto avremo molti context switch con aumento dell'overhead, mentre se troppo lungo tenderà a comportarsi come FCFS con i relativi problemi. Anche in questo caso, però, il tempo di attesa medio per entrare in servizio può essere troppo lungo.
Quando il quantum è molto breve, e il context switch è progettato per essere il più breve possibile, si parla di ***time sharing***: N processi avanzano al limite come se disponessero di N CPU ciascuna, con capacità di elaborazione pari a 1/N di quella reale.

SHORTEST-PROCESS-NEXT FIRST (SPN)

SPN non è preemptive e prevede di mandare in esecuzione, alla fine del processo corrente, quello in coda ready che ha il minor tempo stimato di completamento. Questo algoritmo è molto indicato per i processi dei job batch, di cui sono noti a priori i tempi stimati di esecuzione Certamente in applicazioni gestionali standard è possibile monitorare il sistema e valutare con una certa accuratezza i tempi di esecuzione. La stima dei tempi di esecuzione è invece molto difficile per i processi interattivi di durata non nota a priori.



SPN favorisce i processi brevi rispetto a quelli lunghi e rende minimi i tempi di attesa medi. In pratica è uno scheduling a priorità implicita, dove la priorità è inversamente proporzionale al tempo di esecuzione. È possibile la starvation (attesa infinita) di processi lunghi, a causa del continuo inserimento diprocessi brevi. A causa della sua natura non preemptive SPN non è adeguato per ambienti timesharing, bensì per quelli batch. Se in coda abbiamo processi con tempi stimati uguali, questi sono schedulati in modalità FCFS. Vediamo ora un semplice esempio per il calcolo del tempo medio di attesa, supponendo di avere in coda ready 4 processi A, B, C, D con tempi di esecuzione a, b, c, d.



Il tempo medio di attesa è minore quando a, b, c, d sono in coda in ordine crescente di tempo diesecuzione.

SHORTEST REMAINING TIME (SRT)

SRT è la versione preemptive di SPN ed è adeguato per il time-sharing. Lo scheduling SRT prevede che il processo seguente a essere eseguito sia quello con il minor tempo stimato di completamento, includendo i nuovi arrivi in coda ready. Mentre SPN prevede il completamento dell'esecuzione del processo, in SRT, se entra in coda ready un processo con tempo di completamento stimato minore di quello in esecuzione, gli viene assegnata la CPU. Ovviamente SRT ha un overhead maggiore di SPN, in quanto deve aggiornare il tempo stimato per finire il processo in esecuzione e gestire

le preemption. Con questo algoritmo processi corti attendono pochissimo tempo per essere eseguiti mentre quelli più lunghi avranno ritardi maggiori.

PRIORITY SCHEDULING

A ogni processo è assegnata una priorità e viene mandato in esecuzione il processo a priorità maggiore; nel caso di processi con uguale priorità, si opera con FCFS. La priorità può essere impostata internamente dal sistema o esternamente dal programmatore.

COMPETIZIONE E COOPERAZIONE

Interferenza tra processi

In ambiente multiprocesso, i processi vengono eseguiti per lo più in maniera asincrona l'uno dall'altro e quindi possono accedere a risorse condivise in un modo "disordinato", il che può comportare la corruzione del valore di variabili o effetti gravi sulle periferiche. La soluzione per evitare questa situazione è proibire che più di un processo possa leggere e scrivere dati "simultaneamente", o utilizzare simultaneamente alcune periferiche. Abbiamo scritto simultaneamente tra virgolette per ricordare che a livello non macroscopico in realtà i processi si alternano sulla CPU. Che cosa accadrebbe se un processo sta aggiornando i campi di un vettore di struct, ma a un certo punto l'esecuzione passa a un processo che aggiorna anche lui i dati dello stesso array e infine anche questo viene sospeso e va in esecuzione un processo che legge i dati dall'array? Quest'ultimo leggerebbe dati incoerenti, prodotti un po' da un processo, un po' dall'altro.
Pensiamo ora alla stampa che otterremmo qualora un processo A che, impegnata la stampante, inizia a stampare un testo A ma viene sospeso dal processo B, che inizia a stampare a sua volta: vedremmo stampato parte del testo A e parte del testo B.
La parte del codice che implementa l'accesso alla risorsa condivisa si dice ***sezione critica***.
In particolare le sezioni critiche si identificano laddove abbiamo il riferimento a una o più variabili condivise in "read-update-write": Il punto è che mentre un processo sta eseguendo codice in sezione critica nessun altro processo deve fare altrettanto. In pratica l'esecuzione di sezioni critiche deve essere soggetta alle seguenti regole:

1. ***mutua esclusione***: se il processo P sta eseguendo la sua sezione critica, nessun altro processo può eseguire la propria sezione critica;
2. ***progresso***: se nessun processo è nella sezione critica ed esiste un processo che desidera entrare nella propria sezione critica, allora l'esecuzione di tale processo non può essere posposta indefinitamente;
3. ***attesa limitata***: se un processo P ha richiesto di entrare nella propria sezione critica, il numero di volte che si concede agli altri processi di accedere alla propria sezione critica prima del processo P deve essere limitato.

Mutua esclusione

Occorre quindi serializzare l'esecuzione della sezione critica; il primo processo che impegna la sezione critica deve obbligare gli altri a mettersi in coda di attesa e, quando finisce l'esecuzione, deve in qualche
modo determinare l'accesso al primo processo in coda di attesa. Questa procedura è chiamata *mutua esclusione*. Dobbiamo quindi prevedere un protocollo di impegno della sezione critica e uno di disimpegno.

Soluzione 1: DISABILITAZIONE DELLE INTERRUZIONI

Una prima proposta prevede la disabilitazione delle interruzioni quando un processo entra in sezione critica e la riabilitazione quando ne esce. In questo modo, se non ci sono altri master sul bus in grado di modificare la memoria, siamo certi di bloccare il multiplex della CPU. Indubbiamente non è molto prudente dare a un processo di utente la gestione delle interruzioni; che cosa accadrebbe se per un errore non venissero riabilitate o, comunque, se rimanessero troppo a lungo disabilitate, dal momento che la decisione è lasciata al processo?

Soluzione 2: BLOCCO DELLO SCHEDULATORE

Alcuni sistemi operativi prevedono le system call ***lock*** e ***unlock*** che sospendono e ripristinano il multiprocessing, in pratica basandosi sulla disabilitazione dell'interrupt RTC, dando modo al processo di eseguire una fase in single processing. Anche questa soluzione è molto pericolosa, in quanto si dà ai processi di utente uno strumento che, utilizzato erroneamente o maliziosamente, può alterare il comportamento di tutto il sistema.

Soluzione 3: TEST AND SET

Questo metodo prevede l'utilizzo di una variabile in Memoria centrale (un byte) condivisa, detta ***LOCK***, inizializzata a 1, che indica la presenza di un processo in sezione critica quando vale 0. Un processo testa LOCK prima di entrare in sezione critica; se vale 1 il processo la resetta e entra nella sezione critica, altrimenti esegue un'istruzione di loop sul test, attendendo che LOCK diventi a 1, cosa che accade a carico del processo attualmente in sezione critica al momento dell'uscita. Questa soluzione ha un grave difetto: se il processo A testa LOCK e la trova a 1; ma prima di

eseguire l'istruzione di set viene schedulato un processo B che testa e setta con successo LOCK, quando A torna in esecuzione, supponendo che B non sia uscito dalla sezione critica, procede settando LOCK (in realtà già settata da B) ed entra nella sezione critica. Ambedue i processi saranno in sezione critica, violando una delle regole di mutua esclusione.
Per questo motivo, tutte le moderne CPU hanno un'istruzione macchina di tipo ***test and set*** che in un ciclo istruzione (non interrompibile per definizione) legge la variabile LOCK, imposta il flag Z e setta LOCK. Vediamo un segmento di programma assembly 8086 che effettua la test and set:

```
        MOV AX, 0
P_LOCK:
        XCHG AX, LOCK
        CMP AX,0
        JZ P_LOCK
```

L'operazione di rilascio è la seguente:

```
P_UNLOCK:
        MOV LOCK, 1
```

L'istruzione XCHG scambia in un unico ciclo-istruzione il contenuto di AX con quello della locazione di memoria LOCK, impostando i flag di stato secondo il valore di LOCK.
Il problema che rimane è il fatto che se un processo A ha impegnato la sezione critica, gli altri processi che eseguono la test and set (supponiamo che l'algoritmo di scheduler sia il Round Robin), spenderanno il loro quantum eseguendo continuamente il loop di attesa (***attesa attiva*** o ***busywaiting***). Questa attività è ovviamente uno spreco di tempo di CPU, nel senso che, per esempio, nello stesso tempo un altro processo potrebbe progredire nella sua esecuzione.

SEMAFORI

Un semaforo è una ***variabile di tipo intero non negativo protetta***, nel senso che il suo valore può essere manipolato solo dalle operazioni atomiche P, V e dalla procedura di inizializzazione. Il termine ***atomico*** si riferisce al fatto che il flusso di esecuzione delle procedure P e V è indivisibile.
Mentre si esegue una P o V, deve essere garantito che nessun altro processo possa accedere al semaforo.
Se più processi cercano di eseguire contemporaneamente la P sul semaforo, verranno bloccati ma l'implementazione di P e V garantisce che questo avvenga solo per un tempo limitato.
I semafori binari possono assumere solo il valore 0 e 1, mentre i semafori generali possono assumere valori interi positivi o nulli.
Vediamo ora le implementazioni di P (detta anche ***wait***) e V (detta anche ***signal***) con ***attesa attiva del processo***.

P(S)	***V(S)***
while (S==0); /* *attesa attiva: c'è istruzione vuota prima del ;* */ S=S-1; /* *decremento semaforo* */	S=S+1;

Sempre con attesa attiva, i semafori possono essere implementati con la test and set:

P(S)	***V(S)***
while (plock(S)==0) ;/**attesa attiva*/	S=1;

Nelle implementazioni reali i semafori hanno una coda associata che permette di implementare la forma sospensiva che mette in attesa un processo che tenta di entrare in una sezione critica occupata e che pertanto non andrà a consumare inutilmente tempo di CPU. Nella forma sospensiva la Pe la V sono realizzate nel seguente modo:

P(S)	***V(S)***
IF (S > 0) THEN S= S - 1 ELSE (*metti il processo nella coda di S)*	IF (*uno o più processi sono in attesa su S*) THEN (*consenti a uno dei processi in wait su S di proseguire*) ELSE S = S +1

Queste implementazioni di P e V sono sospensive: se un processo che esegue la P non può proseguire in sezione critica perché il semaforo ha valore 0, viene richiamato il dispatcher che provvede a spostare il suo PCB nella coda di attesa associata a quel semaforo, a selezionare un nuovo processo dalla coda ready e quindi a caricare il suo PCB mettendolo in run. La V invece provvede, nel caso la lista sul semaforo non sia vuota, a spostare il PCB del primo processo in attesa nella coda ready; a questo punto lo schedulatore può intervenire o meno a seconda che il sistema sia o meno preemptive.

La mutua esclusione si implementa nel modo seguente:

```
semaphore mutex = 1 /* inizializzazione*/
while (TRUE){
        P(mutex);
        …..                    /* sezione critica*/
        V(mutex);
      }
```

Ai fini della mutua esclusione, un processo deve quindi eseguire prima la P e in seguito la V sullo stesso semaforo binario. Vedremo in seguito un utilizzo diverso per risolvere il problema della cooperazione tra processi.

Cooperazione tra processi

Sincronizzazione

Un altro aspetto della dinamica dei processi è la sincronizzazione. È ovvio che i processi che implementano un'applicazione in qualche misura si coordinino tra loro per realizzare il compito applicativo e quindi devono in qualche modo sincronizzarsi e comunicare.
Focalizziamo ora l'attenzione sulla sincronizzazione, ovvero sui meccanismi che consentono a un processo di sospendersi in attesa della segnalazione da parte di un altro processo di via libera all'avanzamento. Anche se la sincronizzazione in sé potrebbe non prevedere alcuno scambio di messaggi, spesso ciò è implicitamente previsto, in quanto il processo PB in attesa di sincronismo potrebbe proprio aspettare che un altro processo PA svolga un'elaborazione e quindi, in genere, prepari dati che saranno poi elaborati da PB. Per esempio, pensiamo a un sistema di acquisizione dati nel quale il processo PA si occupa di raccogliere i dati dai sensori e organizzarli in un array, mentre il processo PB esegue un algoritmo di calcolo di variabili statistiche sui dati raccolti.

PB in attesa di PA

È il caso elementare di sincronizzazione tra processi: lo statement S1 del processo PA deve essere eseguito prima dello statement S2 del processo PB (per esempio S1 prevede lo sblocco di una valvola di un impianto industriale, mentre S2 prevede il controllo della portata attraverso la valvola).
La soluzione prevede l'utilizzo di un classico semaforo detto di sincronizzazione in quanto viene inizializzato a 0 dal processo PA (sincronizzante) e da questo sottoposto a V, mentre la P viene eseguita da PB (sincronizzato):

```
semaphore: sync=0;

processo PA          processo PB
statement S1         P(sync);
V(sync);             statement S2;
```

Produttore consumatore

Una tipica problematica di sincronizzazione è quella relativa a un insieme di ***k*** processi Produttori (P)che inseriscono i messaggi in una coda FIFO di ***n*** posizioni da dove vengono letti da un insieme ***m*** di processi Consumatori (C).

Produttore consumatore singoli con buffer infinito

Il caso più semplice si ha per *k=m=1* e *n infinito*. È evidente come l'unico Consumatore debba unicamente sincronizzarsi sulla condizione *buffer vuoto*; l'unico produttore deve definire un solo semaforo intero (non binario) di sincronismo, chiamato *semitem*:

```
semaphore: semitem=0;

Processo Produttore                Processo Consumatore
while (TRUE) {                     …….
   produci un item                 while (TRUE) {
   inserisci l'item nel buffer        P(semitem); /* attesa su buffer vuoto*/
   V(semitem);                        estrai l'item dal buffer
}                                     consuma l'item
                                   }
```

Produttore consumatore singoli con buffer circolare finito

Supponiamo ora che la coda FIFO sia di dimensioni finite e realizzata mediante un vettore gestito con la tecnica del buffer circolare. In questo caso occorre aggiungere un secondo semaforo intero di sincronismo sul quale attende il produttore nel caso di buffer pieno, chiamato *space*, inizializzato a *max* (dimensione massima del buffer):

```
semaphore: semitem=0, space=max;

Processo Produttore                 Processo Consumatore
put=0;                              get=0;
while (TRUE) {                      …….
    produci un item                 while (TRUE){
    P (space);                          P(semitem); /* attesa su buffer vuoto*/
    buffer[put]= new item;              new item=buffer[get];
    put=(put+1)%max;                    get=(get+1)%max;
    V(semitem);                         V(space);
}                                       consuma l'item
                                    }
```

Produttore consumatore multipli con buffer circolare finito

In questo caso utilizziamo tre semafori: i due per la sincronizzazione *space* e *semitem*, e uno per la mutua esclusione chiamato *mutex*. Il semaforo intero *space* è utilizzato per contare il numero di posizioni libere nel buffer, mentre il semaforo *semitem* conta il numero di posizioni che sono vuote; infine il semaforo *mutex* ci assicura la protezione delle sezioni critiche del codice nelle quali si accede a dati condivisi e modificabili del buffer: un solo processo Produttore per volta deve poter inserire un dato, un solo processo Consumatore deve poter estrarre un dato, pena l'alterazione di dati strutturati e dei puntatori di servizio al buffer circolare.

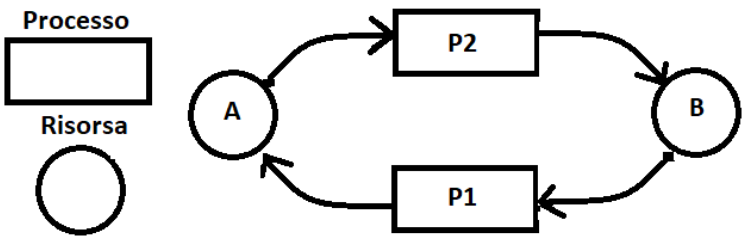
```
semaphore: semitem=0, space=max, mutex=1;
Processo Produttore                 Processo Consumatore
put=0;                              get=0;
while (TRUE) {                      …….
    produci un item                 while (TRUE){
    P (space);                          P(semitem); /* attesa su buffer vuoto */
    P(mutex);                           P(mutex);
    buffer[put]= new item;              new item=buffer[get];
    put=(put+1)%max;                    get=(get+1)%max;
    V(mutex);                           V(mutex);
    V(semitem);                         V(space);
}                                       consuma l'item
                                    }
```

Stallo

Un insieme di processi si trova in stato di ***stallo*** o ***deadlock*** quando ciascuno di essi sta aspettando un evento che deve essere generato da un altro processo dell'insieme, che a sua volta si trova nella stessa condizione. Nessuno dei processi può avanzare, nessuno può rilasciare risorse e nessuno può essere posto in ready.

Le risorse possono essere fisiche oppure logiche: stampanti, nastri, memoria centrale, tempo di CPU, oppure file, semafori, monitor.

Un esempio semplice di deadlock è il seguente: il processo 1 acquisisce in mutex la risorsa A (per esempio, un nastro) mentre il processo 2 acquisisce in mutex la risorsa B (per esempio, una stampante). Il bon processo 1 richiede quindi la risorsa B mentre il processo 2 richiede la risorsa A: nessuno dei due processi può più avanzare. Il sistema è in deadlock, poiché ogni processo dispone di una risorsa richiesta dall'altro e non può rilasciarla.



Il grafo delle risorse schematizza il sistema e ha 2 tipi di nodi, processi o risorse e 2 tipi di archi; un arco dal processo alle risorse indica che il processo richiede la risorsa e dalla risorsa al processo indica che il processo possiede la risorsa.

La differenza tra risorse preemptable e non preemptable è la seguente:

- una risorsa preemptable può essere sottratta al processo senza conseguenze;
- una risorsa non preemptable invece non può essere tolta al processo senza conseguenze negative.

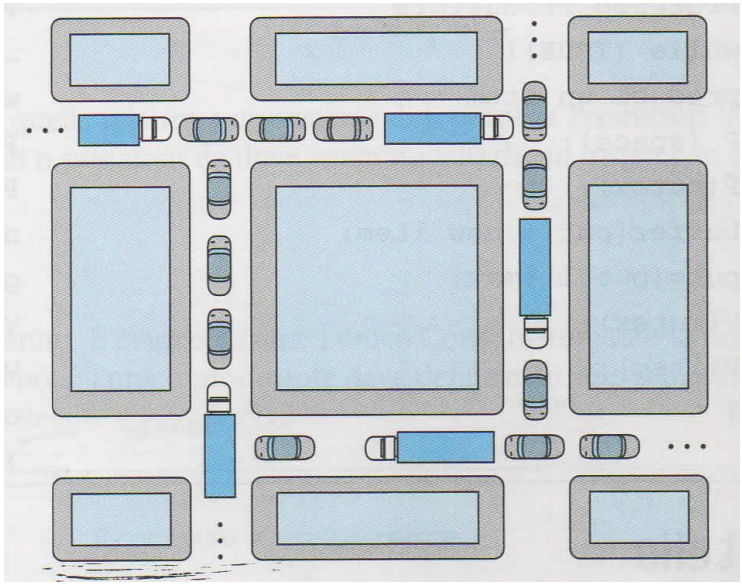
La riallocazione delle risorse può risolvere il deadlock se le risorse coinvolte sono preemptable.

Condizioni necessarie e sufficienti per il deadlock

Abbiamo 4 condizioni che devono verificarsi simultaneamente per causare un deadlock (necessarie ma non sufficienti):

1. condizione di ***mutual exclusion***: le risorse coinvolte non sono condivisibili, ovvero si accede tramite mutua esclusione;
2. condizione ***hold and wait***: un processo richiede risorse ed è in attesa di altre risorse;
3. condizione ***non-preemptive***: le risorse già allocate non sono preemptable dal kernel o rilasciabili volontariamente dal processo;
4. condizione di ***attesa circolare***.

I processi nel sistema formano una lista circolare di attesa nella quale ogni processo della lista è in attesa di una risorsa posseduta dal successivo processo in lista.
Nell'esempio della figura:



- le risorse sono i tratti delle strade;
- la condizione di ***mutua esclusione*** è applicata: infatti solo un veicolo per volta può impegnare l'incrocio;
- la condizione ***hold and wait*** è anch'essa applicata: ogni veicolo impegna una risorsa (un tratto di strada) e desidera impegnare il successivo;
- La condizione ***non preemptive*** è verificata, in quanto non è possibile togliere il tratto di strada al veicolo che la impegna;
- La condizione di ***attesa circolare*** è anch'essa vera: ogni veicolo è in attesa che un altro rilasci il suo tratto di strada. Nella realtà questa condizione non si verifica se l'automobilista scrupoloso comprende che impegnando un incrocio questo si bloccherebbe, bloccando le altre direzioni di traffico.

Gestione del deadlock

Esistono quattro modi di gestire il deadlock, elencati di seguito:

1. ***ignorare del tutto il problema*** (*ostrich algorithm*);
2. ***rilevare e recuperare il deadlock***; viene implementato un monitor che riconosce l'esistenza del deadlock e identifica i processi e le risorse coinvolti. Per effettuare l'operazione di recupero è possibile disallocare temporaneamente risorse ai processi in deadlock, riportare questi processi a uno stato di avanzamento precedente al deadlock, ultimare processi fino a sbloccare il deadlock. È chiaro che si tratta di una procedura molto onerosa;
3. ***evitare il deadlock***, mediante un'attenta schedulazione delle risorse; l'algoritmo più noto è quello del Banchiere, proposto da Dijkstra (1965), così chiamato perché simile alla procedura che utilizzano i banchieri per determinare se un prestito possa essere o meno concesso senza alcun rischio; i processi sono analoghi ai clienti, le unità alle risorse, il banchiere al sistema operativo.
   La tabella mostra quattro clienti, i relativi crediti assegnati e il massimo credito loro assegnabile; il banchiere però decide di assegnare in totale al massimo dieci unità.

Clienti	Crediti	Max	
A	0	6	
B	0	5	Unità disponibili = 10
C	0	4	
D	0	7	

Ipotizziamo che, a un certo punto, si verifichi la situazione illustrata nella tabella seguente.

Clienti	Crediti	Max	
A	1	6	
B	1	5	Unità disponibili = 2
C	2	4	
D	4	7	

In ogni stato ci deve essere almeno un modo per ogni utente di terminare affinché sia uno stato sicuro. Nella tabella precedente siamo in uno stato sicuro, poiché il banchiere può soddisfare C ritardando A, B e D. Quando C finisce, rilascia 4 unità e il banchiere può soddisfare altre richieste.
Uno stato non sicuro si verifica invece se viene data una unità a B. Si verificherà la situazione mostrata nella tabella seguente.

Clienti	Crediti	Max	
A	1	6	
B	2	5	Unità disponibile = 1
C	2	4	
D	4	7	

Se tutti i clienti A, B, C e D chiedessero il massimo di credito residuo, il banchiere non potrebbe soddisfare nessuno e i clienti resterebbero in attesa all'infinito o andrebbero in deadlock.
Se il sistema si trova in uno stato non sicuro, non necessariamente si verificherà il deadlock anche se è un'eventualità plausibile. L'algoritmo procede quindi valutando se, soddisfacendo una richiesta, si passa da uno stato sicuro a uno non sicuro e in questo caso ritarda il soddisfacimento

4. ***Prevenzione del deadlock***: schedulando le risorse in modo da evitare almeno una delle quattro condizioni.
   - è difficile eliminare la condizione di mutua esclusione, in quanto alcune risorse come la stampante devono essere soggette a mutua esclusione.
   - per eliminare la condizione "Hold and Wait" dovremo garantire tutte le risorse necessarie al processo immediatamente all'inizio del processo. Il processo non può iniziare se non allochiamo tutte le risorse richieste. Questa strategia porta a uno spreco delle risorse che rimangono impegnate per tutto il tempo di esecuzione del processo anche se sono effettivamente utilizzate per poco tempo. Inoltre si può verificare la ***starvation***, ovvero l'attesa indefinita di inizio.
   - per evitare la condizione no-preemption si può tentare di forzare un processo a rilasciare quante più risorse possibili se entra in attesa di una risorsa. Le risorse saranno di nuovo richieste in uni secondo momento. Ovviamente il rilascio delle risorse in uso può provocare la perdita del lavoro già svolto e anche la starvation.
   - infine si può provare a evitare la condizione di coda circolare imponendo un ordinamento totale di tutti i tipi di risorse e forzando i processi a richiedere le risorse in base a un ordine crescente o decrescente. In questo modo il grafico di allocazione delle risorse non può mai generare un ciclo.