Sistemi operativi

Giacomo Fantoni
Telegram: @GiacomoFantoni
Filippo Momesso
Telegram: @Momofil31
Github: https://github.com/giacThePhantom/SistemiOperativi

23 aprile 2020

Indice

Capitolo 1

Introduzione

Un sistema operativo è un insieme di programmi che agiscono come intermediario tra l'hardware e l'uomo per facilitare l'uso del computer, rendere efficiente l'uso dell'hardware e evitare conflitti nell'allocazione di risorse tra hardware e software. Offre pertanto un ambiente per controllare e coordinare l'utilizzo dell'hardware da parte dei programmi applicativi. I suoi compiti principali sono di gestore di risorse e di controllore dell'esecuzione dei programmi e il corretto utilizzo del sistema. La struttura dei sistemi operativi è soggetta a notevole variabilità ad è adattabile a criteri di organizzazione estremamente differenti. È pertanto un programma sempre in esecuzione sul calcolatore che generalmente viene chiamato kernel al quale si aggiungono programmi di sistema e programmi applicativi. Nel progettare un sistema operativo si deve tipicamente fare un trade-off tra l'astrazione che semplifica l'utilizzo del sistema e l'efficienza.

Componenti

Un sistema di calcolo si può dividere in 4 componenti:

- Dispositivi fisici: sono composti dall'unità centrale di elaborazione (CPU), dalla memoria e dall'I/O.
- Programmi applicativi: definiscono il modo in cui utilizzare le risorse per risolvere i problemi computazionali da parte degli utenti.
- Sistema operativo: Controlla e coordina l'udo dei dispositivi da parte degli utenti.
- Utenti.

1.0.1 Punti chiave nel progetto di calcolatori

Punto di vista dell'utente

La percezione di un calcolatore dipende dall'interfaccia impiegata. Il più comune metodo di utilizzo è il PC composto da schermo, tastiera e mouse. Il sistema operativo in questo caso si progetta considerando la facilità di utilizzo con qualche attenzione alle prestazioni ma non all'utilizzo delle risorse. Nel caso di un utente che utilizza terminali connessi ad un mainframe condividendo risorse con altri utenti il sistema operativo andrebbe ottimizzato per massimizzare l'utilizzo delle risore.

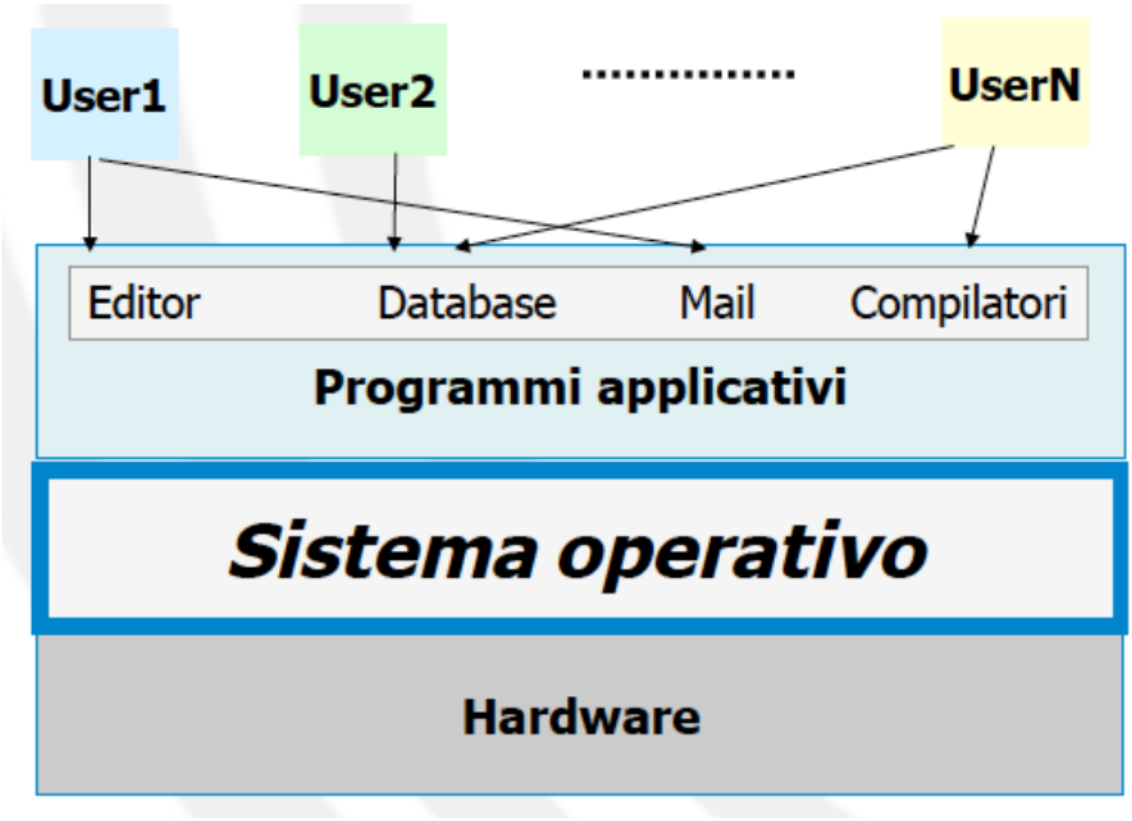

Figura 1.1: Stack del sistema operativo

Punto di vista del sistema

Il sistema operativo è il programma collegato più strettamente ai suoi elementi fisici ed è assimilabile ad un assegnatore di risorse o come programma di controllo che gestisce l'esecuzione dei programmi utente in modo da impedire che si verifichino errori o che il calcolatore sia utilizzato in modo scorretto.

1.1 Storia dei sistemi operativi

Si possono identificare 5 generazioni di calcolatori che riflettono direttamente l'evoluzione dei sistemi operativi dovuta all'aumento dell'utilizzo del processore.

1.1.1 Prima generazione (1946-1955)

In questa generazione i calcolatori erano enormi e a valvole, non esisteva il sistema operativo e l'operatore del calcolatore era equivalente al programmatore. L'accesso alla macchina era gestito tramite prenotazioni e i programmi venivano eseguiti da console caricando in memoria un'istruzione alla volta agendo su interruttori. Il controllo degli errori era fatto attraverso spie della console. Il processing era seriale.

Evoluzione

Durante la prima generazione si diffondono periferiche come il lettore/perforatore di schede, nastri e stampanti che rendono necessari programmi di interazione con periferiche detti device driver. Viene sviluppato del software come librerie di funzioni comuni e compilatori, linker e loader. Queste evoluzioni portano a una scarsa efficienza in quanto pur essendo la programmazione facilitata le operazioni erano complesse con tempi di setup elevati e un basso utilizzo relativo della CPU per eseguire il programma.

1.1.2 Seconda generazione (1955-1965)

In questa generazione si introducono i transistor nei calcolatori. Viene separato il ruolo di programmatore e operatore eliminando lo schema a prenotazione e il secondo permette di eliminare dei tempi morti. I programmi o jobs simili nell'esecuzione vengono raggruppati in batch in modo da aumentare l'efficienza ma aumentando i problemi in caso di errori o malfunzionamenti.

Evoluzione

Nasce l'automatic job sequencing in cui il sistema si occupa di passare da un job all'altro: il sistema operativo fa il lavoro dell'operatore e rimuove i tempi morti. Nasce pertanto il monitor residente, il primo esempio di sistema operativo, perennemente caricato in memoria. Le componenti del monitor erano i driver per i dispositivi di I/O, il sequenzializzatore dei job e l'interprete delle schede di controllo (per la loro lettura ed esecuzione). La sequenzializzazione avviene tramite un linguaggio di controllo o job control language attraverso schede o record di controllo.

Limitazioni

L'utilizzo del sistema risulta ancora basso a causa del divario di velocità tra I/O e CPU. Una soluzione è la sovrapposizione delle operazioni di I/O e di elaborazione. Nasce così l'elaborazione off-line grazie alla diffusione di nastri magnetici capienti e veloci. La sovrapposizione avviene su macchine diverse: da scheda a nastro su una macchina e da nastro a CPU su un'altra. La CPU viene limitata ora dalla velocità dei nastri, maggiore di quella delle schede.

Sovrapposizione tra CPU e I/O

È possibile attraverso un opportuno supporto strutturale far risiedere sulla macchina le operazioni off-line di I/O e CPU.

Polling Il polling è il meccanismo tradizionale di interazione tra CPU e I/O: avviene l'interrogazione continua del dispositivo tramite esplicite istruzioni bloccanti. Per sovrapporre CPU e I/O è necessario un meccanismo asincrono o richiesta I/O non bloccante come le interruzioni o interrupt e il DMA (direct memory access).

Interrupt e I/O In questo caso la CPU programma il dispositivo e contemporaneamente il dispositivo controllore esegue. La CPU, se possibile prosegue l'elaborazione. Il dispositivo segnala la fine dell'elaborazione alla CPU. La CPU riceve un segnale di interrupt esplicito e interrompe l'istruzione corrente salvando lo stato, salta a una locazione predefinita, serve l'interruzione trasferendo i dati e riprende l'istruzione interrotta.

DMA e I/O Nel caso di dispositivi veloci gli interrupt sono molto frequenti e porterebbero a inefficienza. Si rende pertanto necessario creare uno specifico controllore hardware detto DMA controller che si occupa del trasferimento di blocchi di dati tra I/O e memoria senza interessare la CPU. Avviene pertanto un solo interrupt per blocco di dati.

Buffering Si dice buffering la sovrapposizione di CPU e I/O dello stesso job. Il dispositivo di I/O legge o scrive più dati di quanti richiesti e risulta utile quando la velocità dell'I/O e della CPU sono simili. Nella realtà i dispositivi di I/O sono più lenti della CPU e pertanto il miglioramento è marginale.

Spooling Si dice spooling (simultaneous peripheral operations on-line))la sovrapposizione di CPU e I/O di job diversi, Nasce un problema in quanto i nastri magnetici sono sequenziali e pertanto il lettore di schede non può scrivere su un'estremità del nastro mentre la CPU legge dall'altra. Si devono pertanto introdurre dischi magnetici ad accesso causale. Il disco viene utilizzato come un buffer unico per tutti i job. Nasce il paradigma moderno di programma su disco che viene caricato in memoria, la pool di job e il concetto di job scheduling (la decisione di chi deve o può essere caricato su disco).

1.1.3 Terza generazione (1965-1980)

In questa generazione viene introdotta la multiprogrammazione e i circuiti integrati. La prima nasce dal fatto che un singolo job è incapace di tener sufficientemente occupata la CPU e pertanto si rende necessaria una loro competizione. Sono presenti più job in memoria e le fasi di attesa vengono sfruttate per l'esecuzione di un nuovo job. Con la presenza di più job nel sistema diventa possibile modificare la natura dei sistemi operativi: si passa ad una tendenza a sodisfare molti utenti che operano interattivamente e diventa importante il tempo di risposta di un job (quanto ci vuole perchè inizi la sua esecuzione). Nasce pertanto il multitasking o time sharing, estensione logica della multiprogrammazione in cui l'utente ha l'impressione di avere la macchina solo per sè e si migliora l'interattività con la gestione di errori e l'analisi di risultati. Nascono i sistemi moderni con tastiera che permette decisioni dell'evoluzione del sistema in base ai comandi dell'utente e un monitor che permette un output immediato durante l'esecuzione. Il file system inoltre è un'astrazione del sistema operativo per accedere a dati e programmi.

Protezione

Con la condivisione si rende necessario introdurre delle capacità di protezione per il sistema:

- I/O: programmi diversi non devono usare il dispositivo contemporaneamente, viene realizzata tramite il modo duale di esecuzione: modo user in cui i job non possono accedere direttamente alle risorse di I/O e modo supervisor o kernel in cui il sistema operativo può accedere a tali risorse. Tutte le operazioni di I/O sono privilegiate: le istruzioni di accesso invocano una system call, un interrupt software che cambia la modalità da user a supervisor e al termine della system call viene ripristinata la modalità supervisor.
- Memoria: un programma non può leggere o scrivere ad una zona di memoria che non gli appartiene: realizzata associando dei registri limite ad ogni processo, che possono essere modificati unicamente dal sistema operativo con istruzioni privilegiate.
- CPU: prima o poi il controllo della CPU deve tornare al sistema operativo, realizzata attraverso un timer legato ad un job, al termine del quale il controllo passa al monitor.

1.1.4 Quarta generazione (1980-1990)

- Diffusione di sistemi operativi per PC e workstation, utilizzo personale degli elaboratori e nascita delle interfacce grafiche (GUI).
- Sistemi operativi di rete in cui esiste una separazione logica delle risorse remote in cui l'accesso alle risorse remote è diverso rispetto a quello delle risorse locali.
- Sistemi operativi distribuiti: le risorse remote non sono separate logicamente e l'accesso alle risorse remote e locali è uguale.

Quinta generazione (1990- oggi)

Sistemi real-time vincolati sui tempi di risposta del sistema, sistemi operativi embedded per applicazioni specifiche, per piattaforme mobili e per l'internet of things.

Capitolo 2

Componenti di un sistema operativo

Un sistema operativo offre un ambiente on cui eseguire i programmi e fornire servizi che naturalmente variano in base al sistema operativo. Si possono comunque identificare alcune classi di servizi comuni.

2.1 Servizi di gestione

Gestione dei processi

Si intende per processo un programma in esecuzione che necessita di risorse e viene eseguito in modo sequenziale (un'istruzione alla volta). Si differenzia tra processi del sistema operativo e quelli utente. Il sistema operativo è responsabile della creazione, distruzione, sospensione, riesumazione e ella fornitura di meccanismi per la sincronizzazione e la comunicazione tra processi e fornisce meccanismi per la sincronizzazione.

Gestione della memoria primaria

La memoria primaria conserva dati condivisi dalla CPU e dai dispositivi di I/O. Un programma deve essere caricato in memoria prima di poter essere eseguito. Il sistema operativo è responsabile della gestione dello spazio di memoria (quali parti e da chi sono usate), ovvero della decisione su quale processo caricare in memoria in base allo spazio disponibile e dell'allocazione e rilascio dello spazio di memoria.

Gestione della memoria secondaria

Essendo la memoria primaria volatile e piccola si rende necessaria una memoria secondaria per mantenere grandi quantità di dati in modo permanente. È formata tipicamente da un insieme di dischi magnetici (che stanno per essere sostituiti dai dischi a stato solido -SSD- più veloci e performanti). Il sistema operativo è responsabile della gestione dello spazio libero su disco, dell'allocazione dello spazio su disco e dello scheduling degli accessi su disco.

Gestionde dell'I/O

Il sistema operativo nasconde all'utente le specifiche caratteristiche dei dispositivi di I/O per motivi di efficienza e protezione. Viene impegato un sistema per accumulare gli accessi ai dispositivi

(buffering), una generica interfaccia verso i device driver, con device driver specifici per alcuni dispositivi.

Gestione dei file

Le informazioni sono memorizzate su supporti fisici diversi controllati da driver con caratteristiche diverse. Si crea pertanto un file, un'astrazione logica per rendere conveniente l'uso di memoria non volatile grazie alla raccolta di informazioni correlate. Il sistema operativo è responsabile della creazione e cancellazione di file e directory, del supporto di operazioni primitive per la loro gestione (copia, incolla, modifica), della corrispondenza tra file e spazio fisico su disco e del salvataggio delle informazioni a scopo di backup.

Protezione

Si intende con protezione un meccanismo per controllare l'accesso alle risorse da parte di utenti e processi. Il sistema operativo deve definire quali sono gli accessi autorizzati e quali no, i controlli da imporre e fornire gli strumenti per verificare le politiche di accesso. La sicurezza di un sistema operativo comincia con l'obbligo di identificazione di ciascun utente che permette l'accesso alle risorse.

Rete (sistemi distribuiti)

Si intende per sistema distribuito una collezione di elementi di calcolo che non condividono nè la memoria nè un clock: le risorse di calcolo vengono connesse tramite una rete. Il sistema operativo è responsabile della gestione in rete delle varie componenti.

2.2 Interprete dei comandi (shell)

Vi sono due modi fondamentali per gli utenti di comunicare con il sistema operativo: un primo basato su un'interfaccia a riga d comando (o interprete dei comandi) e un secondo bastato su un'interfaccia grafica o GUI. Il primo lascia inserire direttamente agli utenti le istruzioni che il sistema deve eseguire. L'interprete dei comandi è più comunemente conosciuto come shell. La funzione principale dell'interprete è quella di prelevare ed eseguire il successivo comando impartito dall'utente. A questo livello si utano nuovi comandi per la gestione dei file che possono essere implementati internamente all'interprete o attraverso programmi speciali. Nel secondo caso l'interprete non capisce il comando in sè ma prende il nome per caricare l'opportuno file in memoria per eseguirlo.

Interfaccia grafica

L'interfaccia grafica è una modalità di comunicazione tra utente e il sistema operativo. È più intuitiva della riga di comando e la GUI è l'equivalente del desktop e rimane strettamente legata a mouse, tastiera e schermo. Puntando le icone col mouse è possibile accedere a file, cartelle e applicazioni.

2.2.1 System calls

Le chiamate di sistema costituiscono l'interfaccia di comunicazione tra il processo e il sistema operativo. Sono tipicamente scritte in linguaggi di alto livello come *C* o *C++*. I programmatori non si devono preoccupare dei dettagli di implementazione delle *sys.call* in quanto solitamente utilizzano

un'API (application program interface) che specifica un'insieme di funzioni a disposizione dei programmatori e dettaglia i parametri necessari all'invocazione di queste funzioni e i valori restituiti. Le due API più comuni sono *win32* e *POSIX API*, rispettivamente per Windows e UNIX. I parametri delle system calls possono essere pasati per valore o riferimento, ma vanno fisicamente messi da qualche parte: vengono pertanto posizionati nei registri (molto veloci, ma pochi e di dimensione fissa) nello stack del programma o in una tabella di memoria il cui indirizzo è passato in un registro o nello stack. Le system calls possono essere implementate in due modi:

- L'interprete legge il comando e cerca all'interno della shell per cercare il programma da avviare, non viene utilizzata in quanto rende necessario modifiche al kernel e non è efficiente.
- L'interprete legge il comando e possiede una tabella che collega tale comando al programma da avviare.

Le system calls si differiscono in cotrollo dei processi, gesione dei file, dei dispositivi, delle comunicazioni e della protezione. Un processo inoltre deve essere sia in grado di essere chiuso normalmente (*end*) che in modo anomalo (*abort*), con la conseguente generazione di un messaggio di errore e copia dello stato del processo abortito.

Capitolo 3

Architettura di un sistema operativo

Un principio importante è la separazione tra meccanismi e criteri o policy: i primi determinano come eseguire qualcosa, mentre i secondi cosa si deve fare. Questa distinzione è importante ai fini della flessibilità in quanto i criteri sono soggetti a cambiamenti di luogo e tempo.Principi importanti da tenere a mente durante lo sviluppo di sistemi operativi è il KISS (keep it small and simple), semplice dal punto di vista del codice, per mantenere affidabilità e mantenibilità e il POLA (principle of least privilege): un programm deve poter accedere unicamente ai dati strettamente necessari, fondamentale per il mantenimento di sicurezza e affidabilità.Questi cambiamenti devono richiedere il cambio di meccanismi solo nel caso pessimo. Le principali tipologie di architettura di sistemi operativi sono:

- Sistemi monolitici: sistemi senza gerarchia e con un unico strato software tra utente e hardware, tutti i componenti sono sullo stesso livello e possono chiamarsi vicendevolmente. In questa tipologia il codice dipende direttamente dall'architettura hardware e rende test e debugging complesso.
- Sistemi a struttura semplice: si ha un minimo di gerarchia e di struttura, non esiste ancora la suddivisione modalità utente e modalità kernel. Questa struttura è mirata a ridurre i costi di sviluppo e manutenzione.
- Sistema operativo a livelli. I servizi sono organizzati su livello gerarchici con al livello più alto l'interfaccia utente e al più basso l'hardware. Ogni livello può utilizzare funzioni di livelli inferiori. La modularità rende più semplice la manutenzione, ma diminuisce l'efficienza e richiede un'attenta definizione dei livelli.
- Sistemi basati su kernel: vengono utilizzati due livelli, i cui servizi sono distinti tra kernel e non-kernel. Presenta i vantaggi del sistema a livelli come modularità ma senza avere troppi livelli. Tra i servizi al di fuori del kernel non si trova nessuna organizzazione e si tende a pensare al kernel come a una struttura monolitica.
- Sistemi a micro-kernel: i micro-kernel sono un insieme di piccoli kernel che svolgono poche funzioni fondamentali. Occupano meno memoria e sono più affidabili e mantenibili, ma presentano scarse prestazioni: ogni volta che si deve accedere ad un programma applicativo si deve fare un cambio tra modalità kernel a modalità utente e viceversa una volta terminato il processo. Vengono utilizzati da quando le prestazioni di CPU e memoria sono sufficienti a non far percepire all'utente il cambio di modalità. La modularità offre inoltre maggiore sicurezza e portabilità.

3.1 Modello client-server

Una variazione del'idea del microkernel è quella di distinguere due classi di processi: i server che forniscono un servizio e i client che lo utilizzano. Spesso il livello più basso è un microkernel, ma non è richiesto. La comunicazione tra client e server avviene tramite scambio di messaggi: per ottenere un servizio il client deve costruire un messaggio e inviarlo al server, che quando lo riceve restituisce la risposta. Se client e server operano sulla stessa macchina sono possibili delle ottimizzazioni.

3.2 macchine virtuali

Le macchine virtuali sono introdotte nel 1972 da IBM come estremizzazione dell'approccio a livelli persata per offrire un sistema di timesharing multiplo ovvero che permette la multiprogrammazione e una macchina estesa che abbia un interfaccia più semplice del solo hardware. La base della macchina virtuale è la separazione di questi due aspetti. La sua parte centrale era il virtual machine monitor che permette la multiprogrammazione offrendo diverse macchine virtuali al livello superiore. Un tipo di macchina virtuale utilizza un type 1 hypervisor, usato comuniemente che si trova sull'hardware e permette di eseguire diversi sistemi operativi sulla stessa macchina. Il type 2 hypervisor viene utilizzato su un sistema operativo host nel quale l'hypervisor installa il sistema operativo guest in un disco virtuale che il sistema host vede come un file di grandi dimensioni. La differenza tra i due tipi di hypervisor sta nel fatto che quello di tipo 1 si trova direttamente sull'hardware, mentre il tipo 2 viene creato in un sistema operativo host.

3.2.1 Esokernel

Piuttosto che clonare la macchina, un'altra strategia per ottenere un sottinsieme delle risorse è partizionarla. Al livello più basso si trova un programma eseguito in modalità kernel che alloca risorse alle virtual machine e controlla le loro prove di utilizzarle. Il vantaggio dell'esokernel è che evita un livello di mappatura: in altri metodi è necessaria una mappatura tal disco virtuale a quello fisico, come per tutte le altre risorse.

3.3 Processi e thread

Attributi (Process Control Block): contiene un puntatore alla cella di memoria che contiene l'immagine, contiene lo stato del processo in un determinato momento, contiene i registri, le informazioni relative allo stato dell'I/O. Un processo può essere in diversi stati. All'inizio il processo viene creato, poi può essere in esecuzione se gli viene assegnata la CPU o non in esecuzione se non gli viene assegnata la CPU. Il Dispatcher assegna la CPU ai processi che sono pronti, ma non in esecuzione (posso avere diversi processi in memoria, ma solo uno per volta usa la CPU e quindi è effettivamente in esecuzione, a meno di CPU multicore). Quando un processo è pronto e viene creato viene messo nella ReadyQueue (oppure nella coda di un dispositivo cioè la coda in cui viene messo un processo che sta aspettando di accedere a un determinato dispositivo). In realtà esistono diverse code in cui può essere messo un processo. Dispatch e Scheduler sono due componenti diversi, lo scheduler sceglie mentre il dispatcher implementa questa cosa. Context-Switch: salvo tutto quello che c'era nel PCB, tutto quello che stavo utilizzando al momento dell'esecuzione. Due operazioni fondamentali che fa un sistema operativo è la creazione e la terminazione del processo. Ogni processo può creare altri processi e questi prendono il nome di processi figli. Il figlio può essere creato in modalità sincrona, cioè finché il figlio è in vita io non faccio niente, oppure in modalità asincrona cioè io creo il figlio

e continuo la mia esecuzione. Con la fork il figlio lo creo esattamente uguale al padre, con la exec posso caricare sul figlio un programma diverso rispetto al padre. Con la wait creo un'esecuzione sincrona tra padre e figlio. Una fork può fallire perché o il padre non ha abbastanza memoria o perché non ha i privilegi per creare un figlio. L'exec cambia l'immagine in memoria del figlio.

3.4 Processi e thread

Attributi (Process Control Block): contiene un puntatore alla cella di memoria che contiene l'immagine, contiene lo stato del processo in un determinato momento, contiene i registri, le informazioni relative allo stato dell'I/O. Un processo può essere in diversi stati. All'inizio il processo viene creato, poi può essere in esecuzione se gli viene assegnata la CPU o non in esecuzione se non gli viene assegnata la CPU. Il Dispatcher assegna la CPU ai processi che sono pronti, ma non in esecuzione (posso avere diversi processi in memoria, ma solo uno per volta usa la CPU e quindi è effettivamente in esecuzione, a meno di CPU multicore). Quando un processo è pronto e viene creato viene messo nella ReadyQueue (oppure nella coda di un dispositivo cioè la coda in cui viene messo un processo che sta aspettando di accedere a un determinato dispositivo). In realtà esistono diverse code in cui può essere messo un processo. Dispatch e Scheduler sono due componenti diversi, lo scheduler sceglie mentre il dispatcher implementa questa cosa. Context-Switch: salvo tutto quello che c'era nel PCB, tutto quello che stavo utilizzando al momento dell'esecuzione. Due operazioni fondamentali che fa un sistema operativo è la creazione e la terminazione del processo. Ogni processo può creare altri processi e questi prendono il nome di processi figli. Il figlio può essere creato in modalità sincrona, cioè finché il figlio è in vita io non faccio niente, oppure in modalità asincrona cioè io creo il figlio e continuo la mia esecuzione. Con la fork il figlio lo creo esattamente uguale al padre, con la exec posso caricare sul figlio un programma diverso rispetto al padre. Con la wait creo un'esecuzione sincrona tra padre e figlio. Una fork può fallire perché o il padre non ha abbastanza memoria o perché non ha i privilegi per creare un figlio. L'exec cambia l'immagine in memoria del figlio.

Capitolo 4

Processi e threads

Il concetto centrale per ogni sistema operativo é quello di processo, l'astrazione di un programma che sta venendo eseguito. Tali astrazioni permettono di avere operazioni pseudo-concorrenti anche quando esiste una sola CPU, trasformandola in diverse CPU virtuali.

4.1 Processi

Tutti i computer moderni svolgono diverse funzioni allo stesso tempo. In un sistema multiprogramma la CPU cambia da processo a processo rapidamente, eseguendo ognuno per decine o centinaia di millisecondi. Si parla di pseudoparallelismo in contrasto con il parallelismo dei sistemi multiprocessore.
.

4.1.1 Il modello dei processi

In questo modello tutti il software eseguibile sul computer è organizzato in un numero di processi sequenziali, istanze di un programma che sta venendo eseguito con i valori per il contatore, i registri e le variabili. Ogni processo possiede la propria CPU virtuale, anche se è la CPU che cambia tra i processi, chiamato multiprogramming. Ogni programma in questo caso viene eseguito in maniera indipendente. Essendo che esiste un unico contatore di programma fisico, quando un processo viene eseguito il suo contatore logico è inserito in quello reale. Quando si passa ad un altro processo il contatore fisico è salvato nel contatore logico del processo. Si assuma che ci sia una sola CPU. Con essa che cambia tra i processi, il tasso di computazione di un processo non sarà uniforme nè riproducibile, pertanto i processi non possono essere programmati con assunzioni riguardo alla temporizzaizone. Quando un processo richiede dei criteri real-time eventi si devono prendere delle misure speciali. Il processo può essere considerato come un'istanza del programma, con un input, un output e uno stato. Si noti come se un programma viene eseguito due volte conta come due proessi.

4.1.2 Creazione dei processi

I sistemi operativi necessitano di un modo per creare processi. In sistemi progettati per eseguire una singola applicazione potrebbe essere possibile avere tutti i processi che saranno necessari presenti allo startup. In sistemi general-purpose è neccessario avere qualche modo per creare e terminare processi quando sono necessari durante l'operazione. Ci sono quattro princupali cause per la creazione di un processo:

- Inizializzazione del sistema. Durante la fase di boot sono creati numerosi processi, alcuni che interagiscono con l'utente e altri di background che non sono associati con utenti particolari ma hanno funzioni specifiche. I processi che stanno nel background per gestire delle attività sono detti daemons e sistemi grossi ne possiedono a dozzine.
- Esecuzione di una system call da un processo che sta venendo eseguito che crea un processo attraverso una system call. Creare un nuovo processo è utile quando il lavoro che deve essere eseguito può essere formulato come un insieme di processi che interagiscono ma sono indipendenti.
- L'utente richiede di creare il processo in sistemi interattivi attraverso un comando o cliccando su un'icona. In sistemi basati su UNIX il nuovo processo rileva la finestra da dove è eseguito. In windows il processo può creare una o più finestre. In entrambi i casi l'utente può avere più finestre aperte contempoaneamente.
- Inizializzazione di un batch job su sistemi batch che si trovano su grandi mainframes. Gli utenti possono inviare batch jobs al sistema e quando il sistema decide che ha le risorse necessarie per eseguirne un altro crea un nuovo processo e svolge il job successivo nella coda.

Tecnicamente in tutti questi casi un nuovo processo è creato avendo un processo esistente che esegue una system call che dice al sistema operativo di creare un nuovo processo e indica direttamente o indirettamente quale programma eseguire in esso. In UNIX esiste un'unica system call per creare un nuovo processo: *fork* che crea un clone esatto del processo chiamante. Dopo la *fork* i due processi hanno la stessa immagine di memoria, le stesse stringhe ambientali e gli stessi file aperti. Tipicamente il processo figlio esegue *execve* o una system call simile per cambiare l'immagine di memoria e eveguire un nuovo programma. Quando l'utente inserisce un comando la shell forka il processo figlio che poi esegue il comando. QUesti due passaggi permettono al figlio di manipolare i propri file descriptors dopo la *fork* ma prima dell'*execve*. In Windows una singola chiamata alla funzione *CreateProcess* gestisce entrambi i passaggi con 10 parametri che includono la creazione e il caricamento del programma corretto da eseguire, i parametri di liena di comando, attributi di sicurezza, bit di controllo e un puntatore alla struttura in cui le informazioni sul processo sono ritornate al chiamante. Dopo che un processo è creato il genitore e il figlio possiedono i propri spazi di indirizzamento distinti. In UNIX quello del figlio è inizialmente una copia del genitore ma non è condivisa memoria scrivibile.

4.1.3 Terminazione di un processo

Dopo che un processo è stato creato e ha svolto il suo lavoro termina in:

- Normal exit (volontaria), la maggior parte dei processi termina in questo modo eseguendo una system call in modo da dire al sistema operativo che ha finito. Questa call è *exit* in UNIX e *ExitProcess* in Windoes. Programmi screen-oriented supportano la terminazione volontaria.
- Error exit (volontaria), avviene quando il processo scopre un errore, lo annuncia ed esce, le applicazioni screen oriented tipicamente creano una dialog box.
- Fatal error (involontaria) è causata da un bug nel programma, come l'esecuzione di un'istruzione illegale.
- Ucciso da un altro processo (involontario) attraverso una system call che dice al sistema operativo di terminarlo in UNIX è *kill*, in Windows è *TerminateProcess*.

In alcuni sistemi quando un processo termina sono uccisi anche tutti i processi che ha creato.

4.1.4 Gerarchie di processi

In alcuni sistemi quando un processo ne crea un altro rimangono associati in certi modi. in UNIX un progesso e tutti i suoi discendenti formano un gruppo di processi. Quando un segnale viene inviato dalla tastiera viene ricevuto da tutti i membri del gruppo di processi associati alla tastiera. Individualmente ogni processo può catturare il segnale, ignorarlo o svolgere un'azione default, che è di essere ucciso dal sengale. Un altro esempio di gerarchia è dato dal inizializzazione di UNIX successivamente la fase di boot. Un processo speciale detto *init* è presente nella immagine di boot e quando viene eseguito legge un file che dice quanti terminali ci sono. Successivamente forka ad un nuovo processo per terminale. Questi processi aspettano per un login che se hanno successo esegue una shell per eseguire comandi che potrebbero iniziarne altri e così via. Pertanto tutti i processi del sistema si basano su un singolo albero con *init* alla radice. Windows non possiede un concetto di gerarchia dei processi: sono tutti uguali, ma quando un processo è generato il padre possiede un token detto handle che gli permette di controllare il figlio. Questo tolen può essere passato ad altri process, annullando la gerarchia.

4.1.5 Stati dei processi

Nonostante ogni processo sia un'entità indipendente, con il proprio program counter e stato interno, deve spesso interagire con altri processi, ad esempio accettando come input l'output di altri. Quando un processo si blocca lo sa in quanto non può continuare logicamente, tipicamente quando sta aspettando per un input che non è disponibile. È inoltre possibile che sia bloccato in quanto il sistema operativo ha deciso di allocare la CPU per un altro processo. Queste due condizioni sono diverse in quanto la prima è inerente il problema, mentre nel secondo caso è una tecnicalità del sistema. Ci sono pertanto tre stati che il processo può avere:

- Running: il processo sta utilizzando la CPU.
- Ready: eseguibile, stoppato per far eseguire un altro processo. Stato logicamente simile al primo in quanto il processo in entrambi i casi è capace di essere eseguito.
- Blocked: incapace di essere eseguito fino a che un evento esterno accade. Questo stato è differente in quanto il processo non può essere eseguito anche con la CPU libera.

L'unica transizione non possibile è duella da ready a blocked. La transizione da running a blocked avviene quando il sistema scopre che un processo non può contineare al momento. In alcuni sistemi si può eseguire una system call come *pause* per entrare in uno stato bloccato, in altri sistemi come UNIX, quando un processo legge da una pipe o da un file speciale e non si trova input disponibile il processo è automaticamente bloccato. La transizione da running a ready e viceversa sono causate dallo scheduler dei processi in modo da permettere l'eventuale esecuzione di tutti i processi in stato ready. La transizione da blocked a running avviene quando un evento esterno elimina il blocco logico del processo. Il modello dei processi permette di semplificare gli eventi interni al sistema: alcuni processi eseguono comandi dell'utente, altri sono parte del sistema e gestiscono richieste di esso. Quando avviene un interrupt il sistema ferma il processo corrente ed esegue l'interrupt che si bloccano quando aspettano che accada qualcos altro. Il livello più basso del sistema operativo è lo scheduler, con un insieme di programmi al di sopra di esso. Tutti i dettagli della gestione dell'interrupt e di blocco e inizio dei processi sono nascosti e il resto del sistema operativo è stutturato in forma di processo.

4.1.6 Implementazione dei processi

Per implementare il modello dei processi il sistema operativo mantiene una tabella (array di strutture) chiamata la tabella dei processi con un entry per processo contenente informazioni riguardo lo stato del processo, come il program counter, lo stack pointer, la memoria allocata, lo stato dei file aperti e le informazioni di accounting e scheduling e ogni altra informazione che deve essere salvata quando il processo viene passato da running a ready o blocked in modo che possa essere fatto ripartire successivamente. In un tipico sistema la tabella presenta tre colonne con la prima dedicata alla gestione del processo, la seconda alla gestione della memoria e la terza alla gestione dei file. Associato con ogni classe di I/O si trova una locazione (tipicamente fissa al fondo della memoria) detta interrupt vector che contiene l'indirizzo dell procedura del servizio di interrupt. Quando accade un interrupt tutti i processi che stanno venendo eseguiti salvano il program coutner, lo stato e dei registri nello stack grazie all'hardware di interrupt e il computer salta all'indirizzo specificato nell'interrupt vector che fa partire la procedura. Tutti gli interrupt cominciano con il salvataggio dei registri nell'entri del processo corrente, dopo l'informazione è pushata sullo stack dall'interrupt è rimossa e il puntatore allo stack è settato a puntare ad uno stack temporaneo utilizzato dal gestore dei processi. Il salvataggio dei registri e il settaggio dello stack pointer devono essere svolti da una piccola routine in assembly, uguale per ogni interrupt. Quando la routine è finita chiama una procedura in *C* che svolge il resto del lavoro specifico al tipo di interrupt. Quando è finito viene chiamato lo scheduler per vedere quale processo deve essere eseguito. Dopo quello il controllo è passato al codice in assembly per ricaricare in memoria i registri e le mappe per il proceso corrente e comincia ad essere eseguito. Dopo ogni interrupt il processo ritorna precisamente nello stesso stato in cui era prima che accadesse l'interrupt. Riassumendo il processo di interrupt:

- L'hardware salva lo stato del processo sullo stack.
- L'hardware carica il nuovo program counter dall'interrupt vector.
- Una procedura in assembly salva i registri.
- Una procedura crea un nuovo stack.
- Una servizio di interrupt in C viene eseguito (tipicamente legge e buffera l'input).
- Lo scheduler decide il prossimo processo da eseguire.
- Una procedura in C ritorna il codice in assmbly.
- Una procedura in assembly ricomincia il nuovo processo corrente.

4.1.7 Modellare la multiprogrammazione

L'utilizzo della multiprogrammazione permette il miglioramento dell'utilizzo della CPU: se il processo medio computa il 20% del tempo che risiede in memoria, 5 processi alla volta dovrebbero rendere la CPU occupata tutti il tempo (irrealisticamente ottimistico). Un modello migliore consiste nel guardare l'utilizzo della CPU in modo probabilistico: suppondendo che un processo utilizzi una frazione p del suo tempo aspettando per il completamento dell'I/O. Con n processi in memoria alla volta, la probabilità che tutti gli n processi stiano aspettando per l'I/O è p^n, pertanto l'utilizzo della CPU è $1 - p^n$. È facile notare come siano richiesti molti processi per rendere efficiente l'utilizzo della CPU. Si devono naturalmente fare delle assunzioni: tutti i processi sono indipendenti mentre con una singola CPU non si possono avere processi che vengono svolti in contemporanea. Pur non essendo preciso, questo modello dà una buona approssimazione delle prestazioni della CPU.

4.2 Threads

Nei sistemi operativi tradizionali ogni processo possiede uno spazio di indirizzamento e un singolo thread di controllo e nonostante questo è desiderabile avere multipli thread di controllo nello stesso spazio di indirizzamento eseguiti quasi in parallelo, quasi come processi separati.

4.2.1 Utilizzo dei thread

La ragione principale alla base dell'utilizzo dei thread è che molte applicazioni possiedono multiple attività che avvengono insieme. Alcune di queste potrebbero bloccarsi. Una decomposizione in multipli thread sequenziali semplifica il modello di programmazione. Si nota l'analogia con la neccessità di avere i processi, con l'unica differenza che i thread permettono a entità parallele di condividere uno spazio di indirizzamento e tutti i dati tra di loro, un'abilità fondamentale per certe applicazioni. Oltre a questo i thread sono più leggeri e veloci da creare e distruggere dei processi, proprietà utile per un sistema dinamico e rapido. Permettono anche una certa sovrapposizione di attività non legate alla CPU. Invine sono utili in sistemi con multiple CPU, dove il parallelismo è possibile. Esiste un modello di progetatzione in cui lo stato di computazione deve essere salvato e ripristinato nella tabella ogni volta che si cambia tra le richieste, simulando i threads e i loro stack, chiamato macchine a stato-finito. I thread rendono pertanto possibile mantenere l'idea dei processi sequenziali che fanno chiamate bloccanti e ottenere parallelismo:

Modello	**Caratteristiche**
Threads	Parallelismo, system calls bloccanti
Processi a thread singolo	Assenza di parallelismo, system calls bloccanti
Macchine a stato finito	Parallelismo, system calls non bloccanti, interrupts

4.2.2 Il modello dei thread classico

Il modello dei processi si basa sui concetti indipendenti di raggruppamento delle risorse ed esecuzione: i thread permettono la loro separazione. Si può considerare un processo come un modo per raggruppare risorse imparentate insieme: possiede uno spazio di indirizzamento che contiene il programma e i dati e altre risorse come file aperti, processi figli, allarmi in attesa, gestori di segnali, informazioni di gestione che uniti in esso sono gestiti più facilmente. Il processo inoltre possiede un thread di esecuzione che possiede un program counter che traccia quale istruzione eseguire successivamente, registri che contengono le variabili correnti, uno stack con la storia dell'esecuzione con un frame per ogni procedura senza ritorno. Pertanto se i processi raggruppano risorse i thread sono le entità in programma per esecuzione sulla CPU. I thread permettono un esecuzione multipla sullo stesso ambiente di processo con grande livello di indipendenza. Il termine multithreading è utilizzato per descrivere una situazione in cui sono presenti mutlipli thread, detti processi leggeri. Quando un processo a muiltithread viene eseguito su un sistema a singola CPU i threads fanno a turno come i processi. Thread diversi in un processo non sono indipendenti come processi diversi: possiedono lo stesso spazio di archiviazione e pertanto devono condividere le stesse variabili globali e possono accedere e modificare i rispettivi stack, non necessaria in quanto i thread si generano da un processo generato da un utente e sono creati in modo da cooperare. Si nota come pertanto gli oggetti singoli di un processo sono lo spazio di archiviazione, le variabili globali, file aperti, processi figli, allarmi in attesa, segnali e loro gestori e informazioni di contabilità, mentre un thread possiede il program counter, i registri, lo stack e lo stato. Nello stesso modo dei processi possono essere eseguiti, bloccati, pronti o terminati. Ogni stack del thread contiene un frame per ogni procedura che è stata chiamata ma non ha ancora prodotto un valore di ritorno, distinta per ogni thread. Quando si trova

multithreading i processi cominciano solitamente con un unico thread presente che può crearne altri attraverso *thread_ create* in cui un parametro specifica il nome della procedura per il nuovo thread. In alcuni casi possono essere gerarchici. La creazione di un thread ritorna un identificatore del thread che lo nomina. Quando un thread ha finito esce chiamando *thread_ exit* che lo fa svanire. In alcuni sistemi può aspettare per l'uscita di un thread specifico attraverso *thread_ join* che blocca il thread chiamante fino a che il thread specificato non ha finito. Un'altra chiamata comune è *thread_ yeld* che permette ad un thread di abbandonare la CPU per lasciare l'esecuzione per un altro thread. I thread generano delle complicazioni: la creazione di un processo figlio genera problemi in quanto dovrebbe ereditare tutti i thread del padre in quanto potrebbero essere essenziale, ma cosa succede a questi ultimi se erano bloccati da un interrupt. Un'altra classe di problemi dipende dal fatto che i thread condividono molte strutture dati che rendono necessaria una progettazione accurata.

4.2.3 Thread POSIX

Per rendere possibile la creazione di programmi con thread l'IEEE ha definito uno standard chiamato Pthreads supportato dalla maggior parte dei sistemi UNIX che definisce 60 chiamate a funzioni come:

- *Phtread_ create* che crea un nuovo thread.
- *Phtread_ exit* che termina il thread chiamante.
- *Phtread_ join* che aspetta per l'uscita di un thread specifico.
- *Phtread_ yield* che rilascia la CPU per permettere l'esecuzione di un altro thread.
- *Phtread_ attr_ init* che crea e inizializza la struttura di attributi del thread.
- *Phtread_ attr_ destroy* che rimuove la struttura di attributi del thread.

Ogni Pthread possiede un identificatore, un insieme di registri e un insieme di attributi salvati in una struttura. Quando un nuovo thread è creato viene ritornato l'identificatore del thread, molto simile alla *fork* se non si considerano i parametri. L'identificatore viene utilizzato per riferirsi ad altre chiamate. Quando un thread termina la chiamata alla funzione lo ferma e rilascia lo stack. Gli ultimi due gestiscono la memoria degli attributi inizializzandoli ai valori di default che potranno essere modificati. L'eliminazione degli attributi non causa la terminazione del thread.

4.2.4 Implementazione dei thread nello spazio utente

I thread possono essere implementati nel kernel o nello spazio utente, con una soluzione ibrida possibile. Quando sono implementati nello spazio utente il kernel consiera processi a thread singolo. Un vantaggio è che possono essere implementati in un sistema operativo che non li supporta in una liberia. Vengono eseguiti in un sistema a run-time, che è un insieme di procedure che li gestiscono. Ogni processo possiede la propria tabella dei thread privata che tiene traccia dei thread nel processo e delle proprietà uniche del thread, gestita dal sistema a tun-time. Quando un thread deve essere cambiato di stato l'informazione necessaria a farlo ricominciare si trova nella tabella. Quando un thread deve essere bloccato localmente chiama una procedura che controlla se tale processo deve essere messo in uno stato bloccat e se lo è salva i registri del thread nella tabella, cerca in essa per un thread pronto e ricarica nei registri i valori salvati del nuovo thread che viene eseguito automaticamente. Se la macchina ha istruzioni per salvare tutti i registri e per cricarli l'intero cambio di thread richiede poche istruzioni, molto più veloce che coinvolgere il kernel. La differenza con i processi è quando un thread ferma la propria esecuzione si possono salvare le informazioni

del thread nella tabella e può dire allo scheduler di selezionare un altro thread per l'esecuzione come parametri locali. Permettono inoltre ad ogni processo di possedere un proprio algoritmo di scheduling e scalano meglio. I loro problemi riguardano come sono implementate le system calls bloccanti in quanto non possono svolgerle perchè bloccherebbero tutti i thread. Le system calls potrebbero essere cambiate a nonblocking ma richiedere cambi nel sistema operativo non è ottimale. Un'altra alternativa è che è possibile dire in anticipo se una chiamata causa un blocco con una chiamata tipo *exists* che permette al chiamante di dire se una chiamata blocchera. La procedura potrà essere cambiata con una nuova che prima fa una chiamata *select* e poi la bloccante solo se è sicuro farla. Se blocca non viene fatta e vieme eseguito un nuovo thread. Il codice richiede cambi alla libreria delle system calls ed è inefficiente ma necessario, tale codice è detto jacket o wrapper. Un altro problema nasce con i page faults: il computer può non mantenere tutti i problemi nella memoria principale contemporaneamente. Quando accade una page fault il sistema operativo cerca le istruzioni mancanti dal disco bloccando il processo. Se il kernel non ha la conoscenza dei thread blocca l'intero processo, anche se altri thread sono pronti. Un ulteriore problema nasce dal fatto che se un thread comincia ad essere eseguito non rilascerà mai volontariamente la CPU. Una soluzione è richiedere al sistema di run-time un segnale di clock interrupt che dà il controllo a altri thread.

4.2.5 Implementazione dei thread nel kernel

Quando i thread sono implementati nel kernel non è necessario un sistema a run-time e tabella dei thread per processo. Quando un thread vuole crearne uno nuovo o distruggerlo fa una chiamata al kernel che la svolge aggiornando la tabella dei thread che contiene le informazioni per tutti i thread, un sottoinsieme di quelle mantenute per i processi. Tutte le chiamate che potrebbero bloccare il thread sono implementate come system calls. Quando un thread blocca il kernel può eseguire un altro thread dallo stesso processo o uno da un altro. Con i thread a livellu utente il sistema di run-time continuava a eseguire thread dallo stesso processo fino a che il kernel non toglieva l'utilizzo della CPU. A causa del grande costo della creazione e distruzione dei thread alcuni sono riciclati: quando un thread è distrutto viene marcato come non eseguibile ma le strutture dati non vengono influenzate: quando un nuovo thread deve essere creato viene riattivato. I thread del kernel non richiedono nuove system calls non bloccanti e le page fault sono più semplici da controllare. Lo svantaggio principale è il costo delle system calls.

4.2.6 Implementazioni ibride

Un modo per combinare i vantaggi dei thread a livello utente e kernel è un'implementazione ibrida che usa thread a livello kernel che poi multiplexa thread a livello utente su alcuni o tutti i primi. In questo approccio il programmatore può determinare quanti thread kernel usare e quanti thread a livello utente multiplexare su ognuno di essi, dando al modello il maggior grado di flessibilità. Il kernel conosce solo i thread di livello kernel e programma solo quelli, che potrebbero avere su di essi thread a livello utente che sono creati, distrutti e gestiti come quelli a livellu utente. Ogni thread di livello kernel ha un insieme di thread a livello utente che fanno a turno per utilizzarlo.

4.2.7 Attivazioni dello scheduler

I thread a livello kernel sono più lenti e un modo per migliorare questa situazione è attrvareso le attivazioni dello scheduler il cui obiettivo è imitare le funzionalità dei thread del kernel con migliori prestazioni e flessibilità: quando un thread si blocca su una system call o su una page fauilt deve essere possibile eseguire altri thread sullo stesso processo se sono pronti. L'efficienza è raggiunta

evitando transizioni non necessarie tra lo spazio utente e del kernel lasciando al run-time utente la possibilità di bloccare i thread sincronizzanti e programmare il prossimo da solo. Quando si utilizzano le attivazioni dello scheduler il kernel assegna un numero di processori virtuali a ogni processo e permette al sistema di run-time dello spazio utente di allocare thread ai processori. Il numero di processori virtuali inizialmente allocati ad un processo è 1, ma possono aumentare e diminuire in base alle necessità del processo. Quando il kernel sa che un thread è stato bloccato notifica il sistema di run-time del processo passando come parametri sullo stack il numero del thread in questione e una descrizione dell'evento. Il kernel attiva il sistema di run-time con un meccanismo detto upcall. Una volta attivato il sistema riprogramma i thread marcando i bloccati e prendendo il prossimo dalla lista dei pronti, inizializzando i registri e rieseguendolo. Quando il thread originale può di nuovo essere eseguito avviene un'altra upcall al sistema di run-time che può decidere se reiniziare il thread o metterlo nella lista dei pronti. Quando accade un hardware interrupt la CPU passa nella modalità kernel. Se l'interrupt è causato da un evento non significativo al processo interrotto viene ripristinato nello stato precedente all'interrupt, altrimenti viene sospeso e il sistema di run-time iniziato su quella CPU virtuale con il thread interrotto sullo stack. Il sistema può poi decidere quale thread programmare su quella CPU. Le upcall non seguono il principio fondamentale del sistema a livelli: fanno chiamate ai livelli superiori.

4.2.8 Thread Pop-Up

I thread sono utili in sistemi distribuiti: quando arrivano messaggi nell'approccio tradizionale si ha un processo o thread che viene bloccato su una system call *receive* che aspetta il messaggio in arrivo. Quando arriva il messaggio lo scompattta, esamina e processa. Un altro approccio è possibile quando l'arrivo del messaggio causa la creazione di un nuovo thread Pop-up per la sua gestione. Un vantaggio chiave è che essendo nuovi non hanno memoria che deve essere ripristinata. Sono tutti identici e permettono una creazione rapida, riducendo la latenza dall'arrivo e l'inizio del processo. Si deve prestare cautela nella pianificazione della loro gestione.

4.2.9 Rendere codice a thread singolo multithreaded

Convertire processi a thread singolo a multithread è difficile: il codice di un thread consiste di procedure multiple. Le variabili locali non danno problemi come le variabili globali al thread ma locali al processo: molte procedure le utilizzano ma altri thread dovrebbero logicamente non usarle. Una soluzione per gestire la sovrascrizione delle variabili globale è eliminarle, ma l'idea entra in conflitto con troppo software già esistente. Un altra è di assegnare ad ogni thread le proprie variabili globali provate in modo da evitare conflitti, creando un nuovo livello di scoping, con variabili visibili unicamente a quelle interne al thread. L'accesso a una variabile globale privata è difficile. È possibile allocare una parte di memoria per le variabili globali e passarla a ogni procedura nel thread come parametro extra o si possono introdurre nuove librerie per creare, settare e leggere queste variabili del thread attraverso due chiamate: una per scriverle e una per leggerle: *set_global("bifptr, &buf)* che ritorna uiul valore di un puntatore nella locazione creata attraverso *create_global* e *bufptr = read_global("bufptr")* che ritorna l'indirizzo salvato nella variabile globale in modo che i propri dati possano essere acceduti. Il problema successivo è che molte procedure non sono reentrant, ovvero non sono progettate per avere una seconda chiamata fatta a una data procedura mentre un'altra non è finita, come nel caso di *malloc* che mantiene tabelle riguardo l'utilizzo di memoria: quando è occupato ad aggiornare le proprie liste potrebbero trovarsi in uno stato inconsistente e nuove chiamate potrebbero portare a puntatori invalidi e crash. Una soluzione è dare a ogni procedura un jacket che setta un bit che marca la libreria come in uno. Ogni tentativo di utilizzarla da parte di

un altro thread risulta in un blocco. Anche i segnali, presentano problemi: alcuni sono specifici al thread altri no: se i thread sono implementati nell'user space non è possibile sapere a quale thread inviare il messaggio, mentre altri, non specifici non si sa quale thread dovrebbe intercettarli. Un altro problema è la gestione dello stack: quando uno stack va in overflow viene aumentato lo spazio e quando un processo ha multipli thread possiede multipli stack. Se il kernel non ne è a conoscenza non può aumentare la loro dimensione in quanto non individua l'overflow.

4.3 Sincronizzazione di processi

Abbiamo visto che i processi possono essere eseguiti in modo concorrente o in parallelo. Abbiamo poi visto il ruolo dello scheduler nella selezione di quale processo eseguire e per quanto tempo eseguirlo. Dunque è chiaro che un processo può essere interrotto quando ancora non ha terminato completamente la sua esecuzione e se ciò avviene mentre sono in corso operazioni di modifica su dati condivisi si può cadere in condizioni di incosistenza e perdita di integrità dei dati.

4.3.1 Condizioni di competizione

Nel caso in cui diversi processi accedano o manipolino gli stessi dati in maniera concorrente e il risultato dipenda dall'ordine particolare in cui gli accessi vengano effettuati, si parla di condizioni di competizione. Bisogna quindi fare in modo che i processi accedano ai dati condivisi uno alla volta, ovvero che in qualche modo debbano essere sincronizzati.

4.3.2 Regioni critiche

Per evitare le condizioni di competizione si proibisce che più di un processo possa scrivere o leggere i dati condivisi nello stesso momento, in maniera mutualmente esclusiva. La scelta di quali operazioni primitive utilizzare per ottenere la mutua esclusione è un problema fondamentale nella progettazione di un sistema operativo. La zona condivisa a cui viene fatto accesso viene detta regione critica o sezione critica e per avere una buona soluzione si deve porre che:

- **Mutua esclusione**: due processi non possono trovarsi simultaneamente nelle loro regioni critiche.
- **Progresso**: solo i processi che stanno per entrare nella sezione critica possono decidere chi entra. La decisione non può essere rimandata all'infinito.
- **Attesa limitata**: nessun processo deve aspettare per sempre prima di entrare nella sua regione critica.

4.3.3 Soluzioni software

Variabili di lock

Si consideri una variabile condivisa di lock inizialmente a 0. Quando un processo vuole entrare nella propria regione critica testa il lock: se è a 0 lo setta a 1 ed entra nella regione critica, altrimenti resta in attesa fino a quando il lock torna a 0. Il problema non viene risolto in quanto la competizione viene spostata sul lock. Inoltre viene violato il criterio del progresso. Se P0 cede il turno e non ha più necessità di entrare nella sezione critica allora anche P1 non può più entrare nella sezione critica (vedere slide 15 lezione 07).

Strict alternation

Un ulteriore approccio è dato dall'esistenza di un'ulteriore variabile che tenga conto del turno per poter entrare nella regione critica. Questo test continuo è detto busy waiting e il lock è detto spin lock. In genere, tuttavia, lo spin lock viene evitato in quanto è uno spreco di risorse CPU. Questa soluzione richiede che due processi si alternino nell'entrare nella regione critica. Viene evitata la competizione sul lock ma c'è il rischio di deadlock. Il deadlock può essere evitato invertendo l'ordine di alcune istruzioni, tuttavia in questo modo viene violato il principio della mutua esclusione.

Soluzione strict alternation

Viene aggiunta una variabile *turn* a cui si assegna l'id dell'altro processo, in modo da dire "entra prima tu se vuoi". Il primo dei due che esegue questa istruzione sarà il primo ad entrare nella sezione critica. Con questa soluzione si risolve il problema nel caso di due soli processi.

Algoritmo del fornaio

Questo algoritmo risolve il problema con N processi. L'idea di base è la seguente:

- Ogni processo sceglie un numero;
- Il numero più basso verrà servito per primo;
- Per casi di numero identico verrà usato un confronto a più livelli (numero, i);

Per i dettagli del codice vedere slides.

4.3.4 Soluzioni hardware

Disattivazione degli interrupts

Su un sistema a singolo processore la soluzione più semplice è quella di avere un processo che disattiva tutti gli interrupt appena entra nella sua regione critica e li riattiva appena prima di uscire. Dato che non possono accadere clock interrupt, una volta che il processo ha disattivato gli interrupt esamina e aggiorna la memoria condivisa senza che nessun altro processo possa intervenire. È chiaro che dare questo potere ai processi non sia una mossa saggia, inoltre questa tecnica non funziona nel caso di sistemi a multiprocessore in quanto *disable* agisce solamente sulla CPU che la esegue. La disattivazione degli interrupt tuttavia è conveniente per il kernel nel caso debba proteggere poche istruzioni mentre aggiorna variabili o liste speciali.

Istruzioni Test and Set e Swap

Un ulteriore modo per risolvere il problema dell'accesso a una risorsa condivisa può essere quello di sfruttare istruzioni particolari implementate a basso livello dall'architettura del processore, in grado di effettuare le operazioni di accesso alla risorsa in un unico ciclo di istruzioni. Le soluzioni che analizzeremo sono le istruzioni *test and set* e *swap*. La prima prende come argomento un valore booleano *var* per riferimento, gli assegna *TRUE* e ritorna il vecchio valore di *var*. La seconda scambia semplicemente i valori dei booleani passati come parametro. Seppure il loro utilizzo permetta di ottenere in maniera immediata le tecnica dello spin lock, un'implementazione che rispetti il principio di attesa limitata è più complessa da realizzare (vedere slide 35 lezione 07 per il codice).
In conclusione le soluzioni hardware hanno come vantaggi una migliore scalabilità dovuta al fatto che

sono indipendenti dal numero di processi coinvolti e che l'estensione a N sezioni critiche è immediata, ma come svantaggi una maggiore complessità di implementazione per il programmatore e la necessità del busy waiting con conseguente spreco di CPU.

4.3.5 Semafori

Le soluzioni al problema della sezione critica e della sincronizzazione di processi che abbiamo visto fino ad ora hanno principalmente due problemi: non è semplice implementarle nei programmi e sono basate su busy waiting. Vediamo ora i semafori, una soluzione generica, relativamente semplice da implementare e in grado di funzionare per qualsiasi problema.
Un semaforo è una variabile intera s a cui si accede **solamente** attraverso due primitive atomiche:

- Signal: V(s) incrementa di 1 il valore di s;
- Wait: P(s) tenta di decrementare di 1 il valore di s, se $s = 0$ resta in attesa.

Esistono due tipi di semafori, i semafori binari e i semafori a valori interi, che si differenziano per implementazione e utilizzo. È importante sottolineare che i semafori a valori interi si basano sull'implementazione dei semafori binari.

Semafori binari - implementazione

Dato che le primitive P() e V() devono essere atomiche, nel caso del semaforo binario risulta necessario l'utilizzo delle istruzioni atomiche di cui avevamo parlato precedentemente come *swap* e *test and set*. In particolare nel caso della primitiva P(), il semaforo si comporta come uno spinlock (mantenendo il problema del busy waiting) utilizzando l'istruzione *swap*. La primitiva V() invece consiste semplicemente di un'assegnazione a *TRUE* del semaforo.

Semafori interi - implementazione con busy waiting

Nel caso di semafori interi si rende necessario l'utilizzo di due semafori binari interni *mutex* e *delay*, rispettivamente di mutua esclusione inizializzato a *TRUE* e di delay inizializzato a *FALSE*. Il primo ha il compito di proteggere s da altre modifiche, il secondo serve ad attendere che il semaforo abbia un valore maggiore di zero prima poterlo liberare.

Semafori interi - implementazione senza busy waiting

Le implementazioni appena viste chiaramente non risolvono il problema del busy waiting, entrambe le tipologie di semafori infatti possono dover restare in attesa di essere "liberati" da altri processi, sprecando cicli di CPU. Una soluzione per il semaforo intero è l'aggiunta all'interno del semaforo di una coda di puntatori a PCB, in sostituzione del semaforo di *delay*. In questo modo, invece di restare in attesa attiva durante l'esecuzione della primitiva P(), il processo si appende nella lista di PCB del semaforo per poi chiamare la procedura *sleep()*. Nella primitiva V(), invece di chiamare *V(delay)*, si estrae dalla coda in modo FIFO un processo e con la procedura *wakeup()* si termina l'attesa per quel semaforo. Questa soluzione risolve parzialmente il problema dell'attesa attiva in quanto il semaforo binario *mutex* ha un implementazione di tipo spinlock. Tuttavia le operazioni sul *mutex* sono veloci e implicano poca attesa, quindi lo spreco di risorse è accettabile.
Una possibile alternativa sarebbe la disabilitazione degli interrupt durante P() e V(). Dato che P() e V() sono funzioni con sezione critica breve e conosciuta a priori, non si presentano i rischi discussi in precedenza. In ogni caso questa alternativa non viene molto utilizzata dai moderni sistemi operativi.

Dunque si presentano due modalità distinte di implementazione. Nel caso del *busy waiting* abbiamo che la CPU controlla attivamente il verificarsi della condizione di accesso alla sezione critica. Nel caso di *sleep* il processo viene messo in attesa che si verifichi la condizione di accesso alla sezione critica.
Il primo è scalabile e più veloce ma CPU-intensive e quindi adatto per attese brevi come l'accesso alla memoria; il secondo più lento e quindi adatto per attese lunghe come I/O.

Utilizzo dei semafori

I semafori binari vengono utilizzati principalmente con la funzione di lock di mutua esclusione per l'accesso a una sezione critica se inizializzati a 1. Se inizializzati a 0 invece hanno una funzione di sincronizzazione (in attesa di un evento) tra diversi processi.
I semafori interi vengono utilizzati per controllare l'accesso a una risorsa composta da un numero finito di istanze. Il semaforo viene inizializzato al numero di risorse disponibili e ogni processo che desideri utilizzare una risorsa effettua un operazione P() sul semaforo, quando la libera effettua una V(). In questo modo quando il semaforo raggiunge il valore 0, significa che tutte le risorse sono occupate.

4.4 Monitor

Nonostante i semafori siano uno strumento molto potente, in grado di risolvere qualsiasi problema di sincronizzazione, non sempre per il programmatore è semplice capire come utilizzarli per risolvere il problema e risulta comunque difficile dimostrare la correttezza della soluzione implementata. Possono infatti esserci degli errori che si verificano solo in particolari condizioni di esecuzione e per questo difficili da individuare. Per questi motivi i linguaggi di programmazione di alto livello spesso forniscono altri strumenti come i **monitor**, le classi **syncronized** di Java e le **CCR** (Conditional Critical Region). I monitor sono un **tipo di dato astratto**, una sorta di "classe" che incapsula variabili e procedure che saranno **protette** in automatico con la mutua esclusione. In questo modo il programmatore non si deve preoccupare di utilizzare *lock* o *semafori mutex* per potervi accedere. Le variabili dichiarate in un monitor formano lo **stato** del monitor e le funzioni definite nel monitor possono accedere solamente alle variabili dello stato, oltre che a quelle dichiarate localmente alla funzione ovviamente. Il monitor garantisce inoltre che un solo processo alla volta possa essere attivo all'interno del monitor (mutua esclusione). Il monitor è provvisto di un ulteriore meccanismo di sincronizzazione dato dal costrutto *condition*. Le uniche operazioni che possono essere invocate su una variabile condition sono *wait()* e *signal()*, le quali si comportano in maniera simile alle operazioni *P()* e *V()* dei semafori. Tuttavia *wait()* blocca **sempre** il processo che la chiama e *signal()* svaglia esattamente un solo processo, se non c'è nessun processo in attesa non succede nulla. Nel caso di *signal* ci sono due comportamenti possibili:

- **Signal and wait**. Il processo che ha chiamato signal attende e l'esecuzione passa al processo che si è sbloccato.
- **Signal and continue**. Il processo che ha chiamato signal continua la sua esecuzione ed esce dal monitor (in questo caso signal deve essere l'ultima istruzione della procedura).

Per concludere la trattazione sui monitor bisogna evidenziare come, sebbene essi siano più semplici da utilizzare per il programmatore, pochi linguaggi li implementano e per funzionare necessitino di memoria condivisa.

```
monitor monitor name
{
   /* shared variable declarations */

   function P1 ( . . . ) {
      . . .
   }

   function P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   function Pn ( . . . ) {
      . . .
   }

   initialization_code ( . . . ) {
      . . .
   }
}
```

Figura 4.1: Struttura di un monitor in pseudocodice.

4.5 Classi syncronized in Java

Java come linguaggio ad alto livello fornisce un suo meccanismo per la sincronizzazione dei processi ovvero la keyword ***syncronized***. Un metodo *syncronized* è un metodo che può essere eseguito da una sola thread alla volta. Ciò viene realizzato mantendendo un singolo lock (detto monitor, ma da non confondere con i monitor trattati in precedenza) per oggetto. Se il metodo *syncronized* è statico allora il lock viene messo sulla classe. E' possibile inoltre definire una sezione critica su qualsiasi blocco utilizzando sempre la keyword *syncronized*. Sono disponibili inoltre altri metodi di sincronizzazione disponibili nella classe Object e quindi ereditati da tutti gli oggetti, ovvero ***wait()***, ***notify()***, ***notifyAll()***.