**Parte II - Capitolo 3 - Processi**

Una questione che sorge nell'analisi dei sistemi operativi è la definizione delle attività della CPU. Anche se l'utente esegue solo un programma alla volta, il sistema operativo deve svolgere le proprie attività interne. Queste attività sono dette processi. Un ***processo*** è un programma in esecuzione, non solo il codice del programma (*sezione di testo*), ma comprende l'attività corrente (rappresentata dal valore del ***contatore di programma***) e dal contenuto dei registri della CPU; normalmente comprende anche il proprio ***stack***, contenente a sua volta i dati temporanei, e una ***sezione di dati*** contenente le variabili globali. Un processo può includere anche uno ***heap***, cioè della memoria dinamicamente allocata durante l'esecuzione del processo. La differenza tra programma e processo sta nel fatto che il primo è un'entità passiva, mentre l'altro è attivo. Un programma diventa processo quando il file eseguibile che lo contiene è caricato in memoria. Un processo è soggetto a cambiamenti di stato durante l'esecuzione. I possibili ***stati*** sono: *nuovo***,** *esecuzione***,** *attesa***,** *pronto***,** *terminato*. Ogni processo è rappresentato nel sistema operativo da un ***blocco di controllo di un processo*** (***process control block***, ***PCB***) che contiene molte informazioni connesse a un processo specifico, tra cui: *stato del processo*, *contatore di programma*, *registri di CPU*, *informazioni sullo scheduling*, *informazioni sulla gestione della memoria*, *informazioni di contabilizzazione delle risorse*, *informazioni sullo stato dell'I/O*. Il modello a processi sottointende che un processo è un programma che si esegue seguendo un unico percorso d'esecuzione detto ***thread***.

L'obiettivo della multiprogrammazione consiste nel disporre dell'esecuzione contemporanea di alcuni processi in modo da massimizzare l'utilizzo della CPU; inoltre si vuole commutare l'uso della CPU tra i vari processi. Lo ***scheduler dei processi*** seleziona un processo da eseguire dall'insieme di quelli disponibili per raggiungere questi obiettivi. Ogni processo è inserito in una ***coda di processi*** composta da tutti i processi del sistema. I processi presenti in memoria centrale, che sono pronti e nell'attesa di essere eseguiti, si trovano in una lista detta ***coda dei processi pronti*** (***ready queue***) che è generalmente una lista concatenata. Il sistema operativo ha anche altre code. L'elenco dei processi che attendono la disponibilità di un particolare dispositivo di I/O si chiama ***coda del dispositivo*** ed è propria di ogni dispositivo. Un nuovo processo che si colloca inizialmente nella coda dei processi pronti, deve attendere finchè non è selezionato per essere eseguito (*dispatched*). Una volta selezionato e assegnato alla CPU, il processo può: emettere una richiesta di I/O, creare un nuovo processo, rimosso forzatamente dalla CPU a causa di un'interruzione. Durante la sua esistenza, un processo di trova in varie code di scheduling. Il sistema operativo compie la selezione per mezzo di un opportuno scheduler. Lo ***scheduler a lungo termine*** (***job scheduler***) sceglie i lavori dall'insieme di *spooling* e li carica in memoria affinchè siano eseguiti. Lo ***scheduler a breve termine***, o ***scheduler di CPU***, fa la selezione tra i lavori pronti per l'esecuzione e assegna la CPU a uno di loro. Questi due scheduler si differenziano principalmente per la frequenza con la quale sono eseguiti. Lo scheduler a breve termine seleziona frequentemente (pochi millisecondi) un nuovo processo per la CPU. Lo scheduler a lungo termine esegue la selezione con frequenza molto inferiore. Esso controlla il ***grado di multiprogrammazione***, cioè il numero di processi presenti in memoria. È importante che lo scheduler a lungo termine faccia un'accurata selezione dei processi. Un ***processo con prevalenza di I/O*** (*I/O bound*) impiega la maggior parte del proprio tempo nell'esecuzione di operazioni di I/O; un ***processo con prevalenza d'elaborazione*** (*CPU bound*) richiede poche operazioni di I/O e impiega la maggior parte del proprio tempo in elaborazioni. È importante che lo scheduler a lungo termine selezioni una buona combinazione dei processi visti. Esistono sistemi (UNIX e Microsoft Windows) in cui l'utilizzo dello scheduler a lungo termine è minimo o nullo. In alcuni sistemi operativi si può introdurre un livello di scheduling intermedio tramite lo ***scheduler a medio termine***, rappresentato dallo schema di ***avvicendamento dei processi in memoria*** o ***avvicendamento*** (*swapping*). In presenza di un'interruzione, il sistema deve salvare il ***contesto*** del processo corrente, per poterlo ripristinare in seguito. Il contesto è rappresentato all'interno del PCB e comprende i valori dei registri di CPU, lo stato del processo e informazioni relative alla gestione della memoria. In generale si esegue prima il salvataggio dello stato corrente della CPU, sia in modalità utente che in modalità di sistema, poi si attuerà un corrispondente ripristino dello stato per poter riprendere l'elaborazione dal punto in cui si era interrotta. Il passaggio della CPU a un nuovo processo implica la registrazione dello stato del processo vecchio e il caricamento dello stato precedentemente registrato del nuovo processo; questa procedura va sotto il nome di ***cambio di contesto*** (***context switch***) e comporta un calo delle prestazioni perchè il sistema esegue solo operazioni volte alla corretta gestione dei processi e non alla computazione.

Nella maggior parte dei sistemi , i processi si possono eseguire in modo concorrente e si devono creare e cancellare dinamicamente. Durante la propria esecuzione, un processo può creare numerosi nuovi processi tramite un'apposita chiamata di sistema (create\_process); il processo creante sarà *genitore* mentre il nuovo processo sarà  *figlio*. Ciascun nuovo processo può creare alri processi formando una struttura ad ***albero***. La maggior parte dei sistemi operativi identifica un processo per mezzo di un numero intero univoco detto ***identificatore del processo*** o ***pid*** (*process identifier*). Per eseguire il suo compito, un processo necessita di alcune risorse. Quando un processo viene creato, questo ottiene le proprie risorse direttamente dal sistema operativo, oppure può essere vincolato a un sottoinsieme di risorse del processo genitore. Quando un processo ne crea uno nuovo, l'esecuzione può avvenire in due modi diversi: o il genitore continua l'esecuzione in modo concorrente ai figli, o il processo genitore attende che alcuni o tutti i processi figli terminino. Ci sono due possibilità anche per quel che riguarda lo spazio d'indirizzi di un nuovo processo: o il processo figlio è un duplicato del genitore, o nel processo figlio si carica un nuovo programma. Un nuovo processo si crea per mezzo della chiamata di sistema fork() ed è composto di una copia dello spazio degli indirizzi del genitore. Generalmente dopo una chiamata di fork(), viene riportato il valore zero nel processo figlio e viene riportato il PID del figlio nel processo genitore. In questo modo i processi sanno qual'è il processo padre e qual'è il processo figlio. Dopo la creazione, uno dei due processi avvia una chiamata di sistema di exec() per sostutuire lo spazio di memoria del processo con un nuovo programma. Il processo genitore può generare figli, oppure, se durante l'esecuzione del processo figlio non ha altre funzioni da svolgere, può invocare la chiamata di sistema wait() per rimuovere se stesso dalla coda dei processi pronti fino alla terminazione del figlio. Un processo termina quando finisce l'esecuzione della sua ultima istruzione e inoltra la richiesta al sistema operativo di essere cancellato usando la chiamata di sistema exit(). Un processo (generalmente un genitore) può terminare l'esecuzione di un altro processo tramite la chiamata TerminateProcess(). Un genitore può porre fine all'esecuzione di un figlio per diversi motivi: il processo figlio ha ecceduto nell'uso di alcune risorse assegnategli, il compito assegnato al figlio non è più richiesto, il processo genitore termina e il sistema operativo non permette al figlio di continuare l'esecuzione. In alcuni sistemi, se il processo genitore termina, devono terminare anche i suoi figli: si parla di ***terminazione a cascata***. Un processo è detto ***zombie*** se ha terminato la propria esecuzione, ma non è ancora stato eliminato.

I processi eseguiti concorrentemente nel sistema operativo possono essere indipendenti o cooperanti. Un processo è ***indipendente*** se non può influire su altri processi del sistema o subirne l'influsso. Un processo è ***cooperante*** se influenza o può essere influenzato da altri processi in esecuzione nel sistema. Un ambiente che consente la cooperazione tra i processi è utile per diverse ragioni: condivisione delle informazioni, accelerazione del calcolo, modularità, convenienza. Per lo scambio dei dati e informazioni i processi cooperanti necessitano di un meccanismo di ***comunicazione tra processi*** (***IPC***, ***interprocess communication***). I modelli fondamentali della comunicazione tra processi sono due: a memoria condivisa e a scambio di messaggi. La comunicazione tra i processi basati sulla ***memoria condivisa*** richiede che i processi comunicanti allochino una zona di memoria condivisa. Normalmente il sistema operativo tenta di impedire a un processo l'accesso alla memoria di un altro processo, quindi è necessario che due o più processi raggiungano un accordo e rimangano fuori dal controllo del sistema operativo. Un paradigma molto usato per illustrare il concetto di cooperazione tra processi è il problema del ***produttore/consumatore***. Il processo che produce informazioni è detto produttore, mentre il processo che le consuma è detto consumatore. L'esecuzione concorrente dei due processi richiede la presenza di un buffer che possa essere riempito dal produttore e svuotato dal consumatore. Il buffer dovrà risiedere in una zona di memoria condivisa. I due processi devono essere sincronizzati in modo che il consumatore non tenti di consumare un'unità non ancora prodotta. Sono utilizzabili due tipi di buffer: il buffer ***illimitato*** (senza limiti di produzione da parte del produttore) o ***limitato*** (presuppone l'esistenza di una dimensione fissa). Con buffer limitato il consumatore deve attendere che il buffer sia vuoto, mentre il produttore deve attendere che sia pieno. Un altro modo in cui il sistema operativo può permettere la comunicazione tra i processi è quello di fornire strumenti appositi per lo ***scambio di messaggi***. Lo scambio di messaggi è un meccanismo che permette a due o più processi di comunicare e di sincronizzarsi senza condividere lo stesso spazio degli indirizzi. Questa tecnica è particolarmente utile nei sistemi distribuiti. Un meccanismo per lo scambio di messaggi deve prevedere almeno due operazioni: send e receive. I messaggi possono avere lunghezza fissa o variabile a seconda delle impostazioni. Se due processi vogliono comunicare deve esistere un ***canale di comunicazione*** (*communication link*) tra loro che può essere realizzato secondo diversi metodi logici: comunicazioni diretta o indiretta, sincrona o asincrona, gestione automatica o esplicita del buffer. Con la ***comunicazione diretta*** ogni processo che intende comunicare deve nominare esplicitamente il ricevente della comunicazione. Esiste quindi un solo canale tra ciascuna coppia di processi. Questo schema è simmetrico quando il trasmittente e il ricevente devono nominarsi a vicenda per poter comunicare. Se si vuole creare uno schema asimmetrico, solo il trasmittente deve nominare il ricevente. Con la ***comunicazione indiretta*** i messaggi vengono inviati a delle ***porte*** (*mail-box*) che li riceve. Due processi possono comunicare solo se condividono una porta. Il sistema operativo deve offrire i meccanismi per creare una nuova porta, inviare o ricevere messaggi tramite la porta, rimuovere una porta. Il processo che crea una porta è il proprietario predefinito, ma il diritto di proprietà e di ricezione si possono passare ad altri processi tramite opportune chiamate di sistema. Lo scambio di messaggi può essere ***sincrono*** (o ***bloccante***) oppure ***asincrono*** (o ***non bloccante***). Nel primo caso le primitive si bloccano in attesa dell'arrivo, mentre nell'altro caso continuano la loro esecuzione. Una comunicazione in cui sia l'invio che la ricezione sono bloccanti è detto ***lock step***. Se la comunicazione è diretta o indiretta, i messaggi scambiati tra i processi sono contenuti in code temporanee che possono essere: ***capacità zero***, cioè il canale non può avere messaggi in attesa al suo interno (scambio sincrono), ***capacità limitata***, cioè la coda ha lunghezza finita n e ogni nuovo messaggio viene posto in fondo alla coda se questa non è piena, ***capacità illimitata***, cioè la coda ha una lunghezza potenzialmente infinita e può contenere un numero infinito di messaggi. Il caso a capacità zero è talvolta chiamato sistema a scambio di messaggi senza memorizzazione transitoria (*no buffering*), mentre gli altri due sono sistemi con memorizzazione transitoria automatica (*automatic buffering*).

Analizziamo tre sistemi per la comunicazione dei processi. La ***API POSIX*** prevede meccanismi per la condivisione della memoria. Per prima cosa il processo deve allcare un segmento condiviso in memoria tramite la chiamata di sistema shmget (*shared memory get*). La chiamata comprende tre parametri: uno è la chiave o l'identificazione del segmento condiviso, il secondo è la dimensione in byte del segmento, il terzo stabilisce il modo di accesso al segmento (lettura o scrittura). Se la chiamata va a buon fine viene restituito un intero che identifica il segmento di memoria allocato: i processi che vorranno condividere questo segmento dovranno richiederlo per mezzo del suo identificatore. I processi che vogliono accedere a un segmento di memoria condivisa devono ricorrere alla chiamata di sistema shmat (*shared memory attach*). Anche questa chiamata richiede tre parametri: l'identificatore intero del segmento da annettere, un puntatore che indica in che punto della memoria va annesso il segmento condiviso e un selettore di modo d'accesso. Gli altri processi che condividono il segmento rileveranno una modifica della memoria se questa viene effettuata tramite un'accesso in scrittura. Un sistema operativo basato sullo scambio di messaggi, è il sistema operativo ***Mach***. I messaggi si inviano e si ricevono attraverso porte. Anche le chiamate di sistema s'invocano tramite messaggi. Al momento della creazione di un task si creano due porte speciali: la porta *Kernel* e la porta *Notify*. Il kernel utilizza la porta *Kernel* per comunicare con il task e notifica l'occorrenza di un evento alla porta *Notify*. Per il trasferimento di messaggi sono necessarie solo tre chiamate di sistema: msg\_send, che invia un messaggio a una porta, msg\_receive, per ricevere un messaggio, msg\_rpc, per le chiamate di procedure remote, che invia un messaggio e ne attende esattamente uno di risposta per il trasmittente. La chiamata di sistema port\_allocate crea una nuova porta e assegna lo spazio per la sua coda di messaggi. La dimensione massima predefinita è di otto messaggi. Mach garantisce chepiù messaggi in arrivo dallo stesso trasmittente siano accodati nell'ordine d'arrivo, ma non garantisce un ordinamento assoluto (i messaggi inviati da due trasmittenti diversi vengono accodati in un ordine qualsiasi). Gli stessi messaggi sono composti da un'intestazione fissa (lunghezza del messaggio e due nomi di porte) seguita da una porzione di dati variabile (composta da una lista di dati tipizzata). Nell'operazione di receive, un insieme di porte che deve ricevere un messaggio è trattato come una porta singola. I problemi più gravi dei sistemi a scambio di messaggi sono dovute alle scarse prestazioni dovute alla doppia operazione di copiatura dei messaggi dal tramisttente alla porta, quindi dalla porta al ricevente. Il sistema di messaggi del Mach evita di copiare i messaggi utilizzando tecniche di gestione della memoria virtuale. Il sistema operativo ***Windows XP*** è un esempio di moderno progetto che impiega la modularità per aumentare la funzionalità e diminuire il tempo necessario alla realizzazione di nuove caratteristiche. Esso gestisce diversi ambienti operativi con cui i programmi applicativi comunicano attraverso un meccanismo di scambio di messaggi. Tale funzione è detta *chiamata di procedura locale* e si usa per la comunicazione tra due processi presenti nello stesso calcolatore. Come in Mach si utilizza un oggetto porta per realizzare, stabilire e mantenere una connessione tra due processi. Se deve essere inviato un messaggio più lungo, il trasferimento avviene tramite un *oggetto sezione* che costituisce una regione di memoria condivisa.

Le tecniche di comunicazione tramite messaggi possono essere usate per la comunicazione tra sistemi client/server per cui si considerano tre categorie.

Una ***socket*** è definita come un'estremità di un canale di comunicazione. Una coppia di processi che comunica attraverso una rete usa una coppia di socket, una per ogni processo e ogni socket è identificata dall'indirizzo IP concatenato a un numero di porta. Il server attende le richieste del client, stando in ascolto a una porta specificata; quando il server riceve una richiesta e se viene accettata la connessione, si stabilisce una comunicazione con il client. Quando un processo client richiede una connessione, il calcolatore che lo esegue assegna una porta specifica (numero arbitrario maggiore di 1024). Tutte le connessioni devono essere uniche. Il linguaggio Java prevede tre tipi differenti di socket: quelle realizzate con la classe *Socket* sono dette ***orientate alla connessione*** (*TCP*), quelle che usano la classe *DatagramSocket* sono ***prive di connessione*** (UDP) e quelle basate sulla classe *MulticastSocket* che permette l'invio simultaneo dei dati a diversi destinatari. L'indirizzo 127.0.0.1 è noto come ***loopback*** è usato da una macchina per riferirsi a se stessa in modo da creare una comunicazione tramite il protocollo TCP/IP. La comunicazione tramite socket è diffusa ed efficiente, ma considerata una forma di comunicazione fra sistemi distribuiti di basso livello.

Uno tra i più diffusi tipi di servizio remoto è il paradigma delle ***chiamate di procedure remote*** (***RPC***, ***remote procedure call***). Questa tecnica è stata progettata per astrarre il meccanismo della chiamata di procedura affinchè si possa usare tra sistemi collegati tramite rete. Per molti aspetti è simile al meccanismo di IPC, ma i messaggi scambiati per la comunicazione RPC sono ben strutturati e non semplici pacchetti di dati. La semantica delle RPC permette al client di richiamare una procedura presente in un sistema remoto nello stesso modo in cui invocherebbe una procedura locale. Esso nasconde i dettagli necessari che consentono la comunicazione, assegnando un ***segmento di codice di riferimento*** (*stub*) alla parte client. Il segmento individua la porta del server e passa alla strutturazione dei parametri (*marshalling*) che implica l'assemblaggio dei parametri in una forma che si può trasmettere tramite una rete. Una questione da affrontare riguarda le differenze nella rappresentazione dei dati nel client e nel server: ad esempio, alcuni sistemi rappresentano gli interi tramite metodi diversi (*big endian* e *little endian*). Per risolvere questo problema molti sistemi RPC utilizzano una rappresentazione dei dati indipendente dalla macchina. Una di queste rappresentazioni è la ***rappresentazione esterna dei dati*** (***external data representation***, ***XDR***): nel client la strutturazione dei parametri riguarda la conversione dei dati, prima di inviarli al server, dal formato della specifica macchina nel formato XDR; nel server, i dati in formato XDR si convertono nel formato della macchina server. Un'altra questione importante riguarda la semantica delle chiamate. Per evitare dei fallimenti dovuti a comuni errori di rete, il sistema operativo deve agire esattemente una volta sui messaggi. Per implementare la semantica *esattamente una volta* occorre eliminare il rischio che il server non riceva mai la richiesta. Il server deve implementare la semantica *al massimo una volta* e integrarla con l'invio al client di una ricevuta che attesti l'avvenuta esecuzione della procedura. Esiste un problema di riconoscimento delle porte del server da parte dei client. Il problema è risolvibile in due modi: la corrispondenza tra client e server può essere predeterminata fissando i numeri delle porte, oppure la corrispondenza si può effettuare dinamicamente tramite un meccanismo di *rendezvous*. Generalmente il sistema operativo fornisce un demone di rendezvous detto ***matchmaker*** a una porta RPC fissata. Il client invia un messaggio contenente il nome della RPC al demone, che risponde col numero di porta e la richiesta di esecuzione della RPC si può inviare a quella porta al termine del processo.

Una ***pipe*** agisce come canale di comunicazione tra processi. Sono state uno dei primi meccanismi di comunicazione tra processi nei sistemi UNIX e generalmente forniscono ai processi uno dei metodi più semplici per comunicare l'uno con l'altro, sebbene con qualche limitazione. Esploriamo due tipi comuni di pipe utilizzate sia in UNIX che in Windows. Le ***pipe convenzionali*** permettono a due processi di comunicare secondo una modalità standard chiamata del produttore-consumatore. Il produttore scrive a un'estremità del canale (***estremità dedicata alla scrittura*** o *write-end*) mentre il consumatore legge dall'altra estremità (***estremità dedicata alla lettura*** o *read-end*). Questo tipo di pipe è unidirezionale, perchè permette la comunicazione in un'unica direzione (per essere a doppio senso devono essere utilizzate due pipe). Generalmente una pipe viene creata da un processo padre per mettersi in comunicazione con un figlio generato con il comando fork(). Nei sistemi Windows le pipe convenzionali sono dette ***pipe anonime*** e sono del tutto analoghe alle pipe in UNIX. Le ***named pipe*** costituiscono uno strumento di comunicazione molto più potente, in quanto la comunicazione può essere bidirezionale e non è necessaria la relazione di parentela padre-figlio. Una volta creata una named pipe, questa può essere utilizzata da diversi processi per comunicare. A differenza delle pipe convenzionali, le named pipe continuano a esistere anche dopo che i processi comunicanti sono terminati. In UNIX le named pipe sono messe a disposizione sotto il nome di FIFO. Nostante la comunicazione possa essere bidirezionale, il tipo di comunicazione per le FIFO è di tipo half duplex (nel caso il cui i dati debbano viaggiare in entrambe le direzioni vengono usate due FIFO). In Windows è permessa la comunicazione full duplex.