Dispense di
Sistemi operativi e reti

Emanuele Izzo, 0253052
Gianmarco Corsetti, 0123456

Corso di laurea triennale Informatica
Università Tor Vergata, Facoltà di Scienze MM.FF.NN.
04/08/2020

Documento realizzato in LaTeX

Indice

Parte I

Sistemi operativi

Capitolo 1

Introduzione

Il sistema operativo è un software che gestisce tutte le applicazioni e le funzioni hardware del computer, e si propone come interfaccia tra computer e utente.

Capitolo 2

Sistemi a singolo processore e multiprocessore

2.1 Sistemi a singolo processore

con sistemi a singolo processore si intendono computer a singolo processore, dove il processore presentava un solo core. Ciò comportava la possibilità di eseguire una sola istruzione alla volta.

2.2 Sistemi a multiprocessore

I sistemi a multiprocessore possono avere nelle stessa CPU più core, quindi più processori nello stesso processore, che condividono il bus della memoria e delle periferiche. Ciò porta ad un aumento della produttività (se si hanno n processori, l'incremento è poco meno di n, questo perché il sistema operativo deve comunque gestire il coretto funzionamento di questi processori, e quindi dovrà comunque eseguire delle istruzioni in più) e una resistenza al guasto. Esistono due tipi di sistemi a multiprocessore:

- Multiprocessori simmetrici (SMP), in cui ogni processore esegue i compiti assegnatoli del sistema operativo e dove ogni core è visto allo stesso livello;
- Multiprocessori asimmetrici (AMP), in cui si ha un core che fa da master, che definisce che dovrà fare cosa, e tutti gli altri che fanno da worker, che eseguono i compiti assegnatoli.

Capitolo 3

Struttura del sistema operativo

3.1 Batch multi programmati

Con i sistemi batch multi programmati, e in particolare con la multiprogrammazione, siamo in grado di utilizzare la CPU in maniera più efficiente, evitando che un programma in esecuzione tenga occupata la CPU e i dispositivi di I/O allo stesso tempo.

3.2 SO Time-Sharing

Questo tipo di sistema operativo va a condividere il tempo di utilizzo del computer a più utenti nello stesso istante. Il SO fa eseguire diversi programmi al processore così che si possa simulare il parallelismo tra i tanti utenti collegati al PC.

3.3 SO Real-Time

Questo tipo di SO utilizza la multi programmazione e si basa sulla esecuzione di processi in tempo reale, attraverso delle scadenze, chiamate deadline, che devono essere rispettate e che sono impostate dalle applicazioni (quindi, oltre ad essere corretto il risultato dell'esecuzione del programma, si deve avere questo risultato entro un certo limite di tempo). Questo di solito è usato per la gestione di un sistema fisico detto anche ambiente operativo. I SO Real Time si dividono in due sotto categorie:

- Hard Real Time, in cui se la deadline non viene rispettata può essere danneggiato il funzionamento dell'ambiente operativo (aumentando la Quality of Service, o QoS);
- Soft Real Time, che accetta anche ritardi (diminuendo la QoS).

3.4 Funzionamento Dual Mode e Multi-Mode

Il SO esegue i processi in due modalità distinte: Kernel ed Utente. Queste due modalità sono riconosciute dal processore e dal SO tramite il bit di modalità, che vale 0 per la quella Kernel e 1 per Utente.

3.5 Meccanismi e criteri

Per la progettazione di un SO devono essere ben distinti due aspetti di esso, e sono i meccanismi e i criteri. Il primo stabilisce il modo in cui eseguire delle azioni mentre il secondo stabilisce i modi diversi con cui usare i meccanismi (un esempio di meccanismo è il Contest Switch e lo Scheduling è il criterio che fa uso del Contest Switch).

3.6 Tipologie di struttura del SO

3.6.1 Struttura Monolitica

Questa struttura implementa un sei di istruzioni che sono le chiamate di sistema o SC che implementano un determinato servizio (di solito queste istruzioni sono scritte in Assemply per aumentare la velocità di esecuzione e diminuire lo spazio occupato nella RAM).

3.6.2 Struttura Stratificata

Questa struttura si basa sulla programmazione ad oggetti (OOP), e ogni modulo può utilizzare le funzionalità dei modelli inferiori, avendo quindi non più un solo strato tra l'hardware e le SC, bensì n, di cui uno è il Kernel, che si trova a diretto contatto con l'hardware. Questa struttura semplifica la fase di modifica del codice e rende ogni stato indipendente dall'altro, richiedendo però un'analisi profonda e corretta dell'implementazione. Da notare che, per passare dallo strato n allo strato del Kernel, sono necessarie n SC, e tutta la struttura del SO dovrà essere caricata in memoria.

3.6.3 Struttura Micro-Kernel

Questa struttura va a privilegiare lo spazio che va ad occupare lo stesso. Per gestire le risorse del sistema, si vanno a definire due componenti del SO: le tecniche, che consentono la gestione della memoria, e le strategie di gestione, che definiscono in che modo usare le tecniche. Nel caso del micro-kernel, esso è formato solo dalle tecniche, fatte girare in modalità privilegiata, mentre le strategie di gestione sono implementate nei programmi che girano in modalità utente. Quando un processo applicativo richiede una risorsa, interagisce con i relativo processo server mediante un insieme di chiamate di sistema detto IPC (Inter Process Communication). In questo modo abbiamo una diminuzione dello spazio occupato dal SO, a fronte della necessità di uso di una SC per ogni operazione di comunicazione.

3.6.4 Struttura Modulare

Questa struttura si basa sulla programmazione basata ad eventi e ad oggetti, costruendo il SO tramite dei moduli, ognuno con un scopo preciso. Ogni modulo è implementato da un'interfaccia che contiene le funzionalità contenute nel modulo e il corpo che costituisce le vere e proprie righe di codice. Ovviamente il corpo del modulo è "nascosto" per non potervi apportare modifiche in modo casuale, e si comunica so attraverso l'interfaccia che permette di accedere alle funzionalità offerte dal modulo. Il numero di nodi che il SO

deve avere non è statico, ma può variare in base alle operazioni che si devono fare e alle necessità.

3.6.5 Struttura Ibrida

Pochi SO adottano una struttura unica , infatti è possibile avere una struttura che prende i pregi di più strutture e riesce a sopperire a certi difetti (non tutti ovviamente). Per esempio: Linux e tutti i sistemi Unix prediligono una struttura monolitica, aumentando le prestazioni che può avere un computer, e implementano la modularità che gli permette di aggiungere nuove funzionalità in maniera dinamica; Windows invece usa una struttura monolitica e mantiene un comportamento da micro-kernel.

Capitolo 4

Gestione dei processi

Partiamo da due definizioni: un programma è una lista di istruzioni da eseguire, mentre un processo è l'insieme di azioni che si effettuano (pertanto un processo è l'esecuzione di un programma). I processi prendono il nome di *daemon* in ambienti Linux, e *service* in ambienti Windows. Un processo include una *sezione dati*, uno *stack* ed un *heap* (nel caso la miemoria allocata sia dinamica). Tuttel le proprietà del processo sono meorizzate sul suo descrittore che si chiama *PCB (Process Control Box)*

4.1 Stato di un processo

Il processo nella sua fase di vita può passare in più stati:

- *Nuovo* quanto è stato appena creato:
- *Pronto* se è pronto per l'esecuzione:
- *Esecuzione* se gli è stata assegnata la risorsa della CPU;
- *Bloccato* se aspetta un segnale esterno per poter ripartire;
- *Terminato*.

Quando un processo passa dallo stato di esecuzione a a quello di pronto, significa che il SO gli ha revocato la risorsa della CPU, e quindi si è verificata una *revoca* o prerilascio (da cui deriva il termine inglese preemptive). In molti SO è previsto anche che. per un processo che è situato nella memoria principale, nel caso la non ci sia più spazio nella RAM, si faccia uso della memoria virtuale, per cui si esegue un'operazione di *swap*.

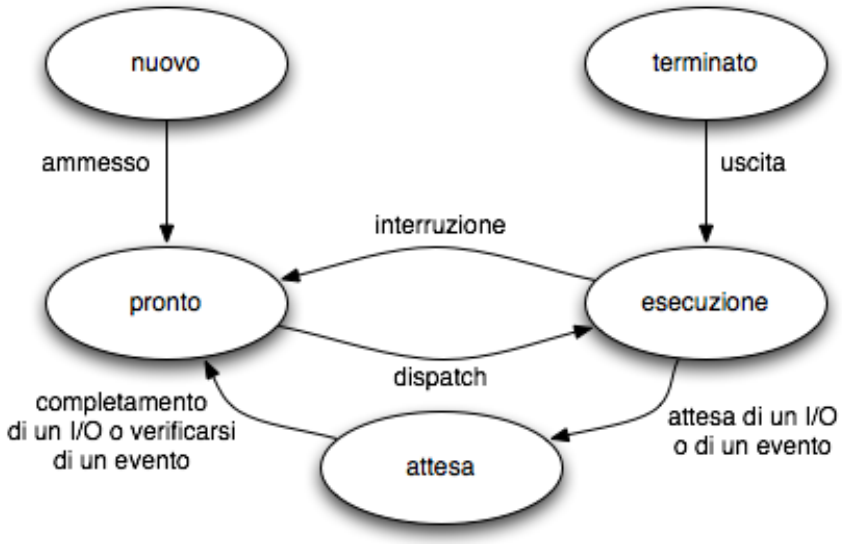

Figura 4.1: Diagramma degli stati

4.2 Descrittori del Processo PCB

Il PCB contiene tutte le proprietà del processo, ed è raggruppato in una tabella chiamata *tabella dei processi* (rappresentata come una lista concatenata). I campi del PCB sono:

- *ID di un processo*: è un numero usato per riconoscere univocamente un processo nel PC.
- *Stato di un processo*:
- *Info sullo scheduling della CPU*: quale algoritmo viene utilizzato per lo scheduling:
- *Informazioni sulla gestione della memoria*;
- *Contesto del processo*: informazioni dei registri del processore al momento della sospensione dell'esecuzione di un processo;
- *Utilizzo delle risorse*: informazioni riguardo la lista dei dispositivi di I/O allocati al processo, i file aperti, il tempo di esecuzione, etc.;
- *ID del processo successivo*;
- *Informazioni sulla sicurezza*;
- *Informazioni sulle variabili d'ambiente*;
- *Informazioni utente*.

4.3 Code di processi

Esiste una coda di processi per ogni stato che compare nella figura (nel caso di sistemi monoprocessore, lo stato di esecuzione ho ha una coda). Generalmente una CPU memorizza il PCB del processo in esecuzione in un registro che si chiama *registro del processo in esecuzione*. Per ogni coda si ha un *descrittore di coda*, che contiene il primo e l'ultimo elemento della coda, creando così una lista concatenata circolare. Nel caso in cui la lista sia vuota in terviene un processo che è fatto apposta per questa situazione, in modo da evitare di mantenere la CPU senza niente da fare. Uqesto processo si chiama *processo di inattività* o anche *idle*.

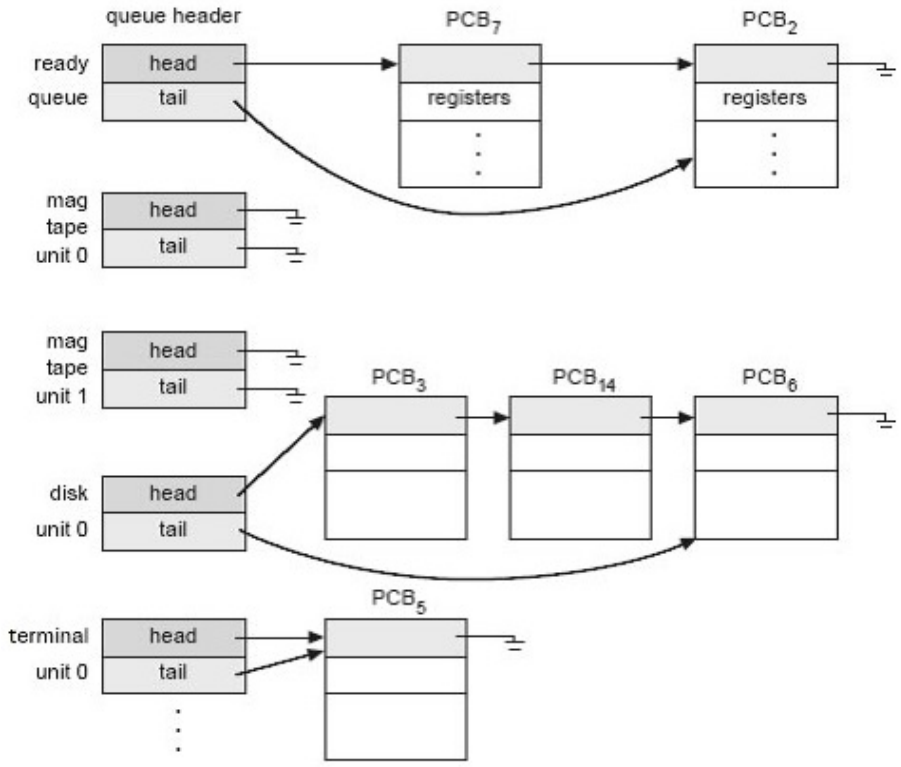

Figura 4.2: Code di processi

4.4 Scheduler ed area di spool

Più programmi sono salvati in una parte del File System sul disco che si chiama *area di spool*: in questa area i programmi sono messi solo se i processi sono in stato di pronto e sono in attesa di eseere eseguiti. Lo *scheduler a lungo termine*, o job scheduler, seleziona i programmi da questa area e li carica nella memoria, per poi eseguirli. Lo *scheduler a breve termine*, o CPu scheduler, seleziona dalla coda di pronto il prossimo processo da eseguire e gli assegna la CPU. Lo scheduler a breve termine sceglie il nuovo processo da eseguire in maniera molto frequente rispetto a quello a lungo termine. Inoltre lo scheduler a breve termine è più semplice dal punto di vista della scelta, in quanto deve solo scegliere il successivo, mentre lo scheduler a lungo termine deve essere molto più preciso.

4.5 Cambio di contesto

Il cambio di contesto è una procedura molto importante che si realizza attraverso mirate chiamate di sistema o SC. Le operazioni effettuate sono:

- Salvataggio del contesto del processo in esecuzione che si trova nel campo del PCB;
- Inserimento del PCB nella coda di pronto o di bloccato in base al motivo del cambio di contesto;
- Selezione di un altro processo dalla coda dei processi pronti e caricamento del PCB nel registro dei processi in esecuzione;
- Caricamento del contesto del nuovo processo nei registri del processore e ripristino dello stato;
- Aggiornamento di tutte le strutture dati che rappresentavano le risorse utilizzate dai processi.

Capitolo 5

Operazioni sui processi

Partiamo dal dire che ogni processo ha un proprio ID di tipo intero, chiamato *PID (Process ID)*. Un processo viene creato tramite la SC fork, la quale non prende nessun parametro in input e restituisce un intero come output:

- 1 per il processo padre;
- 0 per il processo figlio;
- -1 in caso di errore.

Essendo una SC, quando la si esegue, ovviamente ci sarà un interrupt, e quindi il processo padre subisce un context switch, nel quale il kernel dà in pasto al processore la funzione per creare un processo figlio. Alla fine della fork si avranno un processo padre ed un processo figlio, il quale condivide con il padre il segmento del codice e le variabili. È possibile cambiare il comportamento di un processo tramite la SC exec che sostituisce lo spazio di memoria del processo con un nuovo programma (in particolare, per cambiare il comportamento di un processo, dobbiamo modificare l'immagine del processo, sostituendola con un file eseguibile): per fare ciò la exec modifica il segmento del codice dei dati e dello stack, mentre gli altri valori nel PCB non vengono toccati.. Se la exec dovesse avere successo, allora non viene ritornato alcun valore, se invece dovesse fallire non è possibile capirlo, dato che verrà processato il codice che doveva essere rimpiazzato.

5.0.1 Execv ed execl

La exec nello specifico è una famiglia di funzioni, e le due sono usate di più sono due: la execv e la execl.

```
/**
 * path = Percorso/nome del file eseguibile
 * char1, char2, charn = Paramentri da passare al file eseguibile
 *
 * L'ultimo parametro da passare deve essere il carattere vuoto
 **/
execl(char *path, char *arg1, char *arg2,
  ..., char *argn, (char*) 0);

/**
 * path = Percorso/nome del file eseguibile
 * argv[] = Array di paramentri da passare al file eseguibile
 **/
execv(char *path, char *argv[]);
```

La funzione exit viene usata per indicare la fine di un processo, e come parametro prende una variabile stato, la quale consente al processo figlio di comunicare al processo padre un valore di tipo intero che rappresenta il valore di uscita (per errore o meno).

```
/**
 * stato = Valore ritornato al processo padre
 **/
void exit(int stato);
```

Per poter rilevare la notifica della exit del processo figlio, il processo padre deve sincronizzarsi con lui ed attendere la sua terminazione. Questo è possibile utilizzando le funzioni wait e waitpid: la prima mette il padre in attesa della terminazione del suo processo figlio (nel caso in cui ne abbia uno solo, o in generale del primo che termina), mentre la seconda mette il padre in attesa della terminazione di un certo processo figlio (entrambe le funzioni permettono di raccogliere inoltre lo stato di uscita).

```
/**
 * stato = Puntatore della variabile dove salvare lo stato
 **/
pid_t wait(int *stato);

/**
 * pid = PID del processo figlio
 * stato = Puntatore della variabile dove salvare lo stato
 * opzioni = Indica se il processo chiamante deve attendere la
 *           terminazione del figlio oppure continuare
 *           la propria esecuzione
 **/
pid_t waitpid(pid_t pid, int *stato, int opzioni);
```

In alcuni casi il processo figlio può terminare senza che il processo padre abbia rilevato il suo stato di uscita: in questo caso il processo prende il nome di zombie (una volta che il padre chiama la wait per questo processo, però, esso e il suo PCB vengono deallocati). Se un processo padre termina senza aver terminato i processi figli, i figli diventano "orfani" e vengono "adottati" dal processo *init*.

Capitolo 6

Comunicazione tra processi

In un SO i processi possono essere *cooperanti* o *indipendenti*, sia che siano eseguiti in modo concorrente o in parallelo. I processi sono *concorrenti* quando la loro esecuzione si sovrappone nel tempo, ed esistono due casi di processi concorrenti:

- L'*overlapping*, dove ongi processo è eseguito da un processore differente e la loro esecuzione si può sovrapporre nel tempo;
- L'*interleaving*, dove più processi possono condividere la risorsa della CPU, e quindi si spartiscono la risorsa nel tempo per essere eseguiti.

I processi si dicono paralleli se possono essere eseguiti in parallelo su un'unico processore (ciò è possibile solo se la CPU può lavorare su altri processi mentre quelli elaborati lavorano sui dati inseriti). Un processo si dice *indipendente* se non può influenzare o essere influenzato a sua volta da altri processi in esecuzione nel sistema. Un processo si dice *cooperante* se può influenzare o essere influenzato da altri processi in esecuzione. Avere un SO che offre la cooperazione dei processi fornisce i vantaggi della *modularità*, della *condivisione di informazioni* e di una *maggiore velocità di calcolo*.

6.1 Metodi di comunicazione

In generale ci sono due modi principali per far comunicare più processi: *shared memory (memoria condivisa)* e *message passing (passaggio di messaggi)*. il primo modello fa' sì che i processi allochino mediante delle SC delle vere e proprie zone di memoria condivisa in cui possono scambiarsi i messaggi. Nel secondo modello, invece, la comunicazione avviene tramite messaggi scambiati tra i processi cooperanti (ovviamente questa comunicazione viene gestita da delle SC che sono tutte della famiglia delle *IPC*). Seppur la memoria condivisa consente una velocità di trasferimento più elevata rispetto allo scambio di messaggi, nelle architetture multicore si ha che lo scambio di messaggi ha una velocità più alta della modalità a memoria condivisa, dato che nelle architetture a multiprocessore è presente una quantità di cache elevata nella quale i dati condivisi sono copiati, creando così problemi co coerenza delle informazioni.

6.2 Memoria condivisa

La memoria consivisa, nello standard POSIX, viene allocata creando un file tramite la funzione shm_open. Nel caso in cui la funzione ha successo, essa restituisce un intero che identifica il descrittore del segmento di memoria condivisa.

```
/**
 * name = Nome della sezione di memoria condivisa
 * flag = Specifica le modalita' di accesso al segmento
 *        (O_CREAT|O_READ|O_WRITE|O_RDWR)
 * mode = Autorizzazioni che il processo ha sul segmento
 *        condiviso
 **/
shm_fd = shm_open(const char *name, int oflag, mode_t mode);
```

Una volta creato, questo segmento di memoria condivisa ha dimensione nulla, e, per dargli una dimensione, si usa la funzione ftruncate, che dimensiona in byte il segmento di memoria.

```
/**
 * fd = Identificatore del file
 * length = Lunghezza (in byte)
 **/
int ftruncate(int fd, off_t length);
```

La funzione mmap permette di creare un file mappato in memoria che corrisponde al segmento di memoria condivisa, ritornando poi il puntato al file creato, che può essere utilizzato per accedere al segmento di memoria condivisa.

```
/**
 * add = Indirizzo di partenza
 * length = Lunghezza della mappatura
 *          (deve essere maggiore di 0)
 * prot = Protezione di memoria del mapping
 * flags = indica se gli updates al mapping devono
 *         essere visibili agli altri processi che
 *         mappano la stessa regione
 * fd = Identificativo del file da mappare
 * offset = Offset
 **/
void *mmap(void *addr, size_t length, int prot,
  int flags, int fd, off_t offset);
```

Per rimuovere il segmento di memoria condivisa si usa la funzione shm_unlink.

```
/**
 * name = Nome della memoria condivisa
 **/
int shm_unlink(const char *name);
```

6.3 Scambio di messaggi

Per usare la tecnica dello scambio di messaggi si devono avere almeno due funzioni basilari: una per inviare il messaggio, e l'altra per riceverlo. Queste due funzioni sono rispettivamente la send e la receive. Per poter comunicare, però, è necessario che tra i due processi esista un canale di comunicazione, e questo può essere realizzato sia a livello hardware, sia a livello software in vari modi. Di norma il messaggio è diviso in due parti: l'*intestazione* e il *contenuto*. La prima parte contiene informazioni importanti riguardanti:

- Da dove viene;
- Dove deve andare;
- Che tipo di messaggio trasporta;
- Quanto è lungo;
- Il CRC.

Mentre il contenuto ha al suo interno il vero e proprio messaggio.

6.3.1 Comunicazione diretta e indiretta

Esistono due modi per la comunicazione: *diretta* ed *indiretta*. La prima per comunicare richiede di conoscere l'altra estremità della comunicazione.

```
1 send(destinatario, messaggio);
2
3 receive(mittente, messaggio);
```

Mentre nella comunicazione indiretta solo il mittente andrà a specificare il destinatario, mentre il destinatario non è tenuto a conoscere il mittente.

```
1 send(destinatario, messaggio);
2
3 receive(id, messaggio);
```

Il parametro *id* nella receive è impostato sul nome del processo con il quale la comunicazione ha avuto luogo, ed è recuperabile da un campo dell'intestazione del messaggio ricevuto. Attraverso la comunicazione indiretta si usano i trasferimenti di messaggi tramite mailbox o porte. La mailbox è un'astrazione di un oggetto in cui i messaggi vengono messi e prelevati dai processi. Ognuna di esse ha un identificativo univoco, e nello standard POSIX si utilizza una stringa di caratteri.

```
1 send(mailbox, messaggio);
2
3 receive(mailbox, messaggio);
```

Queste mailbox possono appartenere o al processo, o al sistema operativo. Nel caso in cui è parte dello spazio degli indirizzi del processo, allora si distingue tra il proprietario che riceve solo messaggi e l'utente che può solo inviare messaggi. Pertanto, quando il processo che ha la mailbox termina, anche la mailbox viene cancellata, quindi, se un processo che ha ancora il nome della mailbox vuole mandargli un messaggio, è necessario che venga avvisato che quella mailbox non esiste più, e che quindi non è possibile svolgere l'operazione. Invece, quando la mailbox è del SO, allora non è collegata ad un particolare processo, ma è un po'di tutti.

6.3.2 Sincronizzazione

- *Send bloccante o sincrona*: il mittente invia un messaggio al destinatario e rimane in attesa;
- *Receive bloccante o sincrona*: il processo, fino a che non riceve un messaggio, è bloccato;

- *Send non bloccante o asincrona*: il processo mettente, dopo aver inviato il messaggio, continua con la sua esecuzione;
- *Receive non bloccante o asincrona*: consente, anche se aspetta un messaggio in arrivo, di far continuare l'esecuzione del processo.

6.3.3 Code di messaggi

I messaggi sono posti in code temporanee, e queste possono essere implementate in due modi principalmente: *code a capacità 1* e *code a capacità N*. La prima tecnologia di coda è una coda molto semplice in cui, se si immette un elemento, la send è bloccante come la receive. Nella seconda tipologia la coda può avere più elementi, e si possono fare inserimenti multipli, dato che qui possono essere implementate le send e receive non bloccanti. Un dettaglio importante è che il primo metodo è senza buffering, emntre il secondo è a buffering automatico. Le code sono identificate da un nome, e per poter aver accesso alla coda il processo deve avere il suo nome. In POSIX le librerie che permette di lavorare con le code di messaggi sono <sys/stat.h>, <fcntl.h> e <mqueue.h>, e la funzione che permette l'allocazione di una coda di messaggi è mq_open: in caso di successo, la funzione restituisce il descrittore della coda dei messaggi.

```
/**
* name = Nome della coda
* flag = Specifica le modalita' di accesso alla coda
*        (O_RDONLY|O_WRONLY|O_RDWR|O_CREAT)
* mode = Autorizzazioni sulla coda
* attr = Puntatore all struttura dati della coda
**/
mqd_t mq_open(const char *name, int oflag);

mqd_t mq_open(const char *name, int oflag, mode_t mode,
  struct mq_attr *attr);
```

```
/**
* mq_flags = 0 o O_NONBLOCK
* mq_maxmsg = Numero massimo di messaggi
* mq_msgsize = Massima dimesione del messaggio (in byte)
* mq_curmsgs = Numero di messaggi correntemente in coda
**/
struct mq_attr {
  long mq_flags;
  long mq_maxmsg;
  long mq_msgzise;
  long mq_curmsgs;
}
```

La SC mq_close non fa altro che prendere il descrittore della coda e chiuderla, per poi riciclare il descrittore nella coda dei descrittori.

```
/**
* mqdes = Descrittore del file
**/
int mq_close(mqd_t mqdes);
```

Le funzioni per scrivere e leggere sulla coda di messaggi sono, rispettivamente, mq_send e mq_receive.

```
/**
 * mqdes = Descrittore della coda
 * msg_prt = Puntatore al messaggio
 * msg_len = Lunghezza del messaggio
 * msg_prio = Priorita' del messaggio
 **/
int mq_send(mqd_t mqdes, const char *msg_prt,
  size_t msg_len, unsigned int msg_prio);

ssize_t mq_receive(mqd_t mqdes, const char *msg_prt,
  size_t msg_len, unsigned int msg_prio);
```

6.4 Comunicazione con pipe

Le *pipe* sono un'astrazione di un canale per consentire la comunicazione tra processi. Esistono due tipi di pipe: *pipe senza nome* e *pipe con nome*. Per quanto riguarda come accordare i messaggi, si utilizza la politica FIFO per l'immissione e l'emissione dei messaggi dalla pipe. La pipe è un tipo di comunicazione unidirezionale, quindi ci si può accede solo o in lettura o in scrittura, e sono dei canali di comunicazione del tipo *da-molti-a-molti*. Le pipe sono gestite come dei veri e propri file, infatti, anche loro hanno un file descriptor. In POSIX la libraria che permette di gestire le pipe è <unistd.h>, la SC usata per creare la pipe è pipe: se ha successo la funzione restituisce 0, in caso contrario restituisce un valore negativo. Da notare che l'array dei file descriptor contiene, rispettivamente, nella posizione filedes [0] il file descriptor per la lettura, e nella posizione filedes [1] quello di scrittura.

```
/**
 * fd = Array contenente i file descriptors della pipe
 **/
int pipe(int fd[2]);
```

Le SC usare per leggere e scrivere sulla pipe sono, rispettivamente, read e write.

```
/**
* fd = Array contenente i file descriptors della pipe
* buf = Puntatore al messaggio
* count = Numero di byte da leggere/scrivere
**/
ssize_t nbytes = read(int fildes[0], const void *buf, size_t count);

ssize_t write(int fildes[1], const void *buf, size_t count);
```

La SC che permette di chiudere la pipe è close.

```
/**
 * fd = Array contenente i file descriptors della pipe
 * buf = Puntatore al messaggio
 * count = Numero di byte da leggere/scrivere
 **/
ssize_t nbytes = read(int fildes[0], const void *buf, size_t count);

ssize_t write(int fildes[1], const void *buf, size_t count);
```

6.4.1 Comunicazione tramite pipe con nome

Le pipe con nome forniscono un tipo di comunicazione più forte, infatti con la named pipe è possibile far comunicare tutti i processi a prescindere dalla loro parentela. Ogni processo ha la possibilità di comunicare con chi vuole/ha necessità di farlo. Anche questo tipo di pipe sono trattate e gestite come veri e propri file (infatti sono viste come file nel File System). Le pipe con nome vengono create tramite la SC mkfifo. La pipe verrà eliminata solo se il file creato verrà esplicitamente cancellato dal File System, e il sistema di comunicazione rimane half-duplex.

```
/**
 * pathname = Nome della pipe
 * mode = Specifica i permessi della pipe
 **/
int mkfifo(const char *pathname, mode_t mode);
```

Capitolo 7

Thread

É possibile che l'esecuzione di più processi possa portare ad un alto tempo di esecuzione e quindi ad un *overhead*. É un bene avere lo spazio di indirizzamento separato per gli indirizzi, ma questo comporta una collaborazione tra processi quasi nulla o molto difficile da attuare. I *thread* rappresentano i diversi moduli che girano in parallelo all'interno di un processo, e sono definibili come *flussi di esecuzione*. All'interno di un processo ci possono essere più flussi di esecuzione, e quindi più thread, che possono essere creati e distrutti in maniera molto più semplificata rispetto ai processi (per indicare che un processo ha più thread al suo interno si usa il termine *multithreading*). Ogni thread ha un suo *descrittore*, uno *stato* (pronto, bloccato, in esecuzione), uno *spazio di memoria*, uno *stack* e un *contesto*. Le risorse appartengono al processo, e sono condivise tra tutti i thread contenuti nel processo. La gestione dei thread può avvenire sia a livello *utente* che a livello *kernel*.

7.0.1 Thread a livello utente

Esistono delle librerie apposite per la corretta gestione di questi thread, e queste librerie gestiscono tutto del thread, in modo tale che il SO non li considera neanche esistenti. Il parallelismo dei thread a livello utente non è possibile con architetture multiprocessore. Questo modello usato è anche detto *modello molti a uno*.

7.0.2 Thread a livello kernel

In questo caso la gestione dei thread viene fatta a livello kernel, e ogni funzione di gestione dei thread viene fatta tramite delle SC, che deve mantenere sia i descrittori dei processi che quelli dei thread. Questo modelli si chiama *modelli uno ad uno*, e viene supportato da praticamente ogni SO. Nel *modello molti a molti* il numero dei thread è sempre maggiore o uguale a quello dei processi.

7.1 Thread in POSIX

Per poter realizzare i thread in POSIX possiamo far uso della libreria <pthread.h>, che definisce il tipo di dato pthread_t, che viene usato per creare i thread all'interno del processo. Se la funzione ha successo , viene restituito 0 ,altrimenti restituisce un codice di errore.

```c
/**
 * thread = Variabile del nuovo thread da creare
 * attr = Attributi del nuovo thread da creare
 * start_routine = Funzione da far eseguire al nuovo thread
 * arg = Argomenti da passare alla funzione
 **/
int pthread_create(pthread_t *thread,
  const pthread_attr_t *attr,
  void *(*start_routine) (void *), void *arg);

/**
 * Struttura della funzione passata alla pthread_create
 **/
static void *thread_start(void *arg) {
  \\ Contenuto
}
```

La funzione pthread_exit permette di terminare l'esecuzione di un thread, prendendo come parametro un valore rappresentante lo stato di uscita.

```c
/**
* stato = Valore di ritorno
**/
static void pthread_exit(void *stato);
```

La funzione pthread_join mette in attesa della terminazione del thread passato in parametro, salvando il suo stato di uscita nella variabile passata come parametro. Nel caso in cui il thread sia già terminato, la funzione salva soltanto lo stato di ritorno nella variabile.

```c
/**
 * thread = Thread di cui si aspetta la terminazione
 * stato = Valore ritornato dal thread
 **/
int pthread_join(pthread_t thread, void *stato);
```

Capitolo 8

Sincronizzazione tra processi o thread

I processi e i thread possono interagire in due modi, tramite *cooperazione* o *competizione*. Il primo modello, come per i processi che per i thread, avviene attraverso *operazioni di sincronizzazione o di comunicazione*. Il secondo modello, invece, si ha quando dei *processi richiedono delle risorse comuni* che non possono essere usate contemporaneamente. In questo caso è necessario che le orapzioni eseguite su queste *risorse comuni* siano effettuate in *mutua esclusione*. Ci sono due modelli di interazione dei processi, e sono *modello a memoria condivisa* e *modello a scambi di messaggi*. Questi due modelli si possono chiamare rispettivamente modelli ad *ambiente globale* e ad *ambiente locale*. Il primo sfrutta la memoria condivisa, il secondo sfrutta lo scambio di messaggi. Tutte le operazioni che si svolgono in mutua esclusione, quindi che non si sovrappongono nel tempo, prendono il nome di *sezione critica*. Per attuare la mutua esclusione, la soluzione è quella di realizzare un protocollo che i processi devono rispettare ogni volta che entrano in un'area di memoria condivisa, per poter evitare errori. Predefiniamo una sezione di ingresso alla sezione di memoria condivisa chiamata *prologo*, e si memorizza l'accesso o meno di un qualsiasi processo: quindi, se la risorsa è libera allora il processo può accedere, in caso contrario viene bloccato. Se il processo entra nella sezione critica, dopo che ha eseguito le sue operazioni, il processo deve rilasciare la risorsa e consentire ad altri di poterne usufruire. L'insieme di operazioni che portano il processo fuori dalla sezione critica si chiama sezione di uscita o *epilogo*.

8.1 TSL

Affinché tutto possa funzionare, bisogna avere la certezza che il SO esegua le operazioni di lettura e scrittura in *maniera atomica*. Questa garanzia l'abbiamo tramite l'istruzione *TSL (Test and Set Lock)*: è un'istruzione Assembly che copia il contenuto di una variabile y in un registro x del processore, e scrive in y un valore diverso da 0. Per utilizzare in POSIX la TSL si usa la libreria <pthread.h>, che implementa il tipo di variabile pthread_mutex_t, rappresentante lo stato del mutex, ossia la variabile su cui si applica la TSL, e la coda di processi bloccati. L'allocazione di un mutex si effettua attraverso la funzione pthread_mutex_init.

```
1 /**
2  * M = Variabile mutex
3  * attr = Attributi del mutex (se omesso il mutex viene
```

```
 *        inzializzato con valori di default)
 **/
int pthread_mutex_init(pthread_mutex_t *M,
  const pthread_mutexattr_t *attr);
```

La funzione che permette di verificare se un mutex è libero è pthread_mutex_lock: se il mutex è occupato, il thread chiamante viene messo nella coda dei thread in stato di blocco, in caso contrario continua la sua esecuzione.

```
/**
 * M = Variabile mutex
 **/
int pthread_mutex_lock(pthread_mutex_t *M);
```

Una versione non bloccante della funzione appena definita è pthread_mutex_trylock: questa funzione, in caso il mutex sia occupato da un altro thread, ritorna l'errore *EBUSY*.

```
/**
 * M = Variabile mutex
 **/
int pthread_mutex_trylock(pthread_mutex_t *M);
```

Per liberare la variabile mutex e attivare, se presente, un thread nella coda dei thread in stato di blocco, viene chiamata la funzione pthread_mutex_unlock.

```
/**
 * M = Variabile mutex
 **/
int pthread_mutex_unlock(pthread_mutex_t *M);
```

Per deallocare le risorse assegnate ad una variabile mutex viene usata la funzione pthread_mutex_destroy .

```
/**
 * M = Variabile mutex
 **/
int pthread_mutex_destroy(pthread_mutex_t *M);
```

8.2 Semaforo

Il semaforo è una struttura dati composta da un *valore* intero non negativo e da una *coda* dei processi sospesi. Le SC che gestiscono il semaforo sono la *wait()* e la *signal()*. La wait() controlla se il valore del semaforo è uguale a 0: se sì il processo viene sospeso ed inserito nella coda dei processi sospesi, altrimenti decrementa il contatore di 1. La signal() controlla se la coda dei processi sospesi contiene almeno un processo sospeso: se sì, il processo viene rimosso dalla coda dei processi sospesi e viene riattivato, altrimenti viene incrementato il valore di 1. Possiamo notare che la signal() non è bloccante, al contrario della wait() nel caso in cui il valore è uguale a 0. Entrambe le SC devono essere realizzate in modo tale che siano atomiche. in POSIX la libreria che permette di lavorare con i semafori è <semaphore.h>, che implementa il tipo di variabile sem_t, rappresentante il semaforo, ossia il suo valore la coda di processi sospesi. L'allocazione di un semaforo si effettua tramite la funzione sem_init.

```
/**
 * sem = Variabile semaforo
 * pshared = Flag che indica se il semaforo deve essere
```

```
 *           condiviso tra processo o tra thread
 * value = Valore iniziale del semaforo
 **/
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

La funzione che implementa la SC wait() è sem_wait.

```
/**
 * sem = Variabile semaforo
 **/
int sem_wait(sem_t *sem);
```

Esiste una versione non bloccante della SC wait(), implementata dalla funzione sem_trywait.

```
/**
 * sem = Variabile semaforo
 **/
int sem_trywait(sem_t *sem);
```

La funzione sem_post implementa invece la SC signal().

```
/**
 * sem = Variabile semaforo
 **/
int sem_post(sem_t *sem);
```

Per deallocare il semaforo viene utilizzata la funzione sem_destroy.

```
/**
* sem = Variabile semaforo
**/
int sem_destroy(sem_t *sem);
```

La funzione sem_getvalue permette di salvare il valore numerico del semaforo all'interno di una variabile.

```
/**
* sem = Variabile semaforo
* sval = Variabile in cui salvare il valore del
*        semaforo
**/
int sem_getvalue(sem_t *sem, int *sval);
```

Capitolo 9

Scheduling

Lo *scheduler* è il componente del SO che si occupa di selezionare direttamente dalla coda di pronto il processo da assegnare alla CPU. É eseguito molto frequentemente, dato che viene chiamato ogni volta che finisce un processo, quindi per evitare overhead è stato realizzato in modo tale che sia efficiente in termini di velocità di esecuzione (in particolare, lo scheduler implementa le politiche, mentre il *dispatcher* implementa i meccanismi, ossia il cambio di contesto). Lo scheduler a lungo termine è utilizzato principalmente nei SO batch, dato che dobbiamo scegliere i processi all'inizio di ogni azione e non ne seleziona solo uno, ma bensì molti di più. Oltre a questo lo scheduler a lungo termine ha il compito di controllare il *grado di multiprogrammazione* nella macchina, quindi controlla il numero di processi che sono presenti in memoria principale nello stesso tempo. Lo scheduler a medio temrine si occupa di trasferire temporaneamente i processi dalla RAM alla memoria secondaria, e delle ooperazioni di swap-out e di swap-in.

9.1 Comportamento dei processi

Durante la sua attività il processo generalmente può passare in due fasi più volte:

- Fase di *CPU burst*, in cui viene usata soltanto la CPU;
- Fase di *I/O burst*, in cui il processo effettua solo operazioni di I/O.

Quando un processo esegue un I/O burst allora non usa la CPU, e questa nei sistemi multiprogrammati significa che è possibile assegnare la CPU ad un altro processo. Tra gli algoritmo di scheduling ci sono due sottoclassi: gli algoritmi *pre-emptive* e gli algoritmi *non pre-emptive*. Il primo insieme può revocare, secondo vari criteri, la CPU al processo a cui è stata assegnata la risorsa, mentre il secondo non può revocare la CPU, a meno che il processo non si sospenda volontariamente o perché è terminato.

9.2 Parametri di scheduling

- *Tempo di attesa*: quantità di tempo che un processo trascorre nella coda di pronto in attesa che gli venga assegnata la CPU;
- *Tempo di completamento (Turnaround time)*: intervallo di tempo che passa tra l'avvio del processo ed il suo completamento;

- *Tempo di risposta*: intervallo di tempo tra avvio del processo e l'inizio della prima risposta da esso;
- *Utilizzo della CPU*: percentuale media di utilizzo della CPU nell'unità di tempo di esecuzione;
- *Produttività (Throughput rate)*: numero di processi completati nell'unità di tempo definita dal SO;

In generale i parametri che gli algoritmi di scheduling devono massimizzare sono gli ultimi due, quindi l'utilizzo della CPU e la produttività.

9.3 Principali algoritmi di scheduling

9.3.1 FCFS

Il *FCFS (First Come First Serverd)* è un algoritmo di scheduling non pre-emptive che assegna la CPU al primo processo che ne fa richiesta, ossia al processo che è da più tempo in attesa della CU. Nel caso in cui ci siano molti processi che si sospendono di frequente, o nel caso di un sistema interattivo, si avranno dei tempi di risposta davvero lunghi.

9.3.2 SJF

L'*SJF (Shortest Time First)* è un insieme di algoritmi, basati sul tempo di esecuzione:

- *SNPF (Shortest Next Process First)*, algoritmo non pre-emptive, che assegna la CPU al processo con minor tempo di esecuzione tra quelli nella coda di pronto;
- *SRTF (Shortest Remaining Time First)*, algoritmo pre-emptive, che assegna la CPU al processo con minor tempo rimanente di esecuzione tra quelli nella coda di pronto (si attiva quando un nuovo processo entra nella coda di pronto).

9.3.3 RR

L'*RR (Round Robin)* è un algoritmo di scheduling pre-emptive, che assegna a rotazione la CPU a tutti i processi nella coda di pronto. L'idea è quella di trattare la coda di pronto come una coda circolare con politica FIFO. Ad ogni processo viene assegnata la CPU per quanto di tempo, che va dai 10 ai 100 ms, e, quando al processo che usa la CPU le viene revocata la risorsa, viene posto in fondo alla coda di pronto.

9.4 Algoritmi di scheduling basati sulle priorità

Esistono algoritmi di scheduling basati sulla priorità (importanza che il processo ha nel SO). L'algoritmo a priorità assegna la CPU al processo con priorità più alta e quelli con uguale priorità vengono gestiti con la politica FCFS. La priorità può essere gestita in due modi:

- *Priorità statica*, se viene scelta quando viene avviato il processo;
- *Priorità dinamica*, se la priorità può aumentare/diminuire in base a vari criteri.

Un aspetto positivo della priorità dinamica è che i processi non potranno mai andare in *starvation*.

9.5 Algoritmi si scheduling a code multiple

Non esistono solo processi CPU-Bound e I/O-Bound, ma anche quelli interattivi di tipo batch, che sono detti *foreground* o *background*. Data la grande diversità di processi, allora questi vengono posizionati in code differenti. Nei casi più complessi, arriviamo ad avere anche più di 4 code per cui i processi possono passare durante la loro esecuzione: questo fenomeno prende il nome di *multilevel feedback queue*.

9.6 Algoritmi di scheduling real-time

Gli algoritmi di scheduling real-time si basano di solito sulle priorità, e questi algoritmi possono essere *statici* o *dinamici*. Quelli statici assegnano le priorità ai processi in base alla conoscenza di alcuni parametri temporali dei processi noti all'inizio. Quelli dinamici, invece, possono cambiare priorità ai processi nel corso della loro esecuzione. Importanti da definire sono l'*istante di richiesta*, ossia l'instante in cui i processo entra in coda di pronto, la *deadline*, ossia l'instante entro il quale il processo deve essere terminato, e il *tempo di esecuzione*, ossia il tempo necessario al processo assegnato alla CPU per svolgere il suo lavoro. I processi real-time possono essere *periodici* o *aperiodici*: quelli periodici vengono attivati ciclicamente a periodo costante, che dipende anche dalla grandezza fisica che il processo deve controllare, mentre quelli non periodici vengono avviati in situazioni di imprevisti o di emergenze.

9.6.1 RM (Rate Monotonic)

Sia $\mathrm{E}_{\max}$ il tempo massimo di esecuzione di un processo in ciascun periodo T. Sia $\mathrm{E}_{\max}$ che T sono tempi noti a priori che sono imposti dall'applicazione real-time. *RM (Rate Monotonic)* è un algoritmo pre-emptive che assegna la priorità dei processi in base alla durata del loro periodo: quindi, se il periodo è grande, allora avranno una priorità minore. Se adottiamo RM è necessario ma non sufficiente che sia verificata questa condizione: $\mathrm{U} = \sum \frac{\mathrm{E}_i}{\mathrm{T}_i} < 1$, dove U è il coefficinete di utilizzo della CPU. Se usiamo RM, affinchè un insieme n di processi sia schedulabile, è sufficiente che il coefficiente di utilizzo sia: $\mathrm{U} \leq n(2^{\frac{1}{n}} - 1)$.

Capitolo 10

Blocco critico (stallo)

La situazione di *stallo*, o anche detto *deadlock*, si può verificare tra due o più processi quando ciascuno dei processi possiede almeno una risorsa e ne richiede altre. Per rappresentare lo stato di allocazione di un SO si usano due tipologie di rappresentazione: i *modelli basati su grafo* e i *modelli basati su matrici*. Nel primo tipo di modello i deadlock possono essere descritti con un grafo orientato detto *grafo di allocazione delle risorse*. Questo grafo $G := (V, E)$ è definito nel seguente modo:

- $V = P \cup R$, dove $P := p_1, p_2, ..., p_n$ sono i processi e $R := r_1, r_2, ..., r_m$ sono le risorse;
- $E := \{(p_i, r_j) : p_i \in P \wedge r_j \in R \wedge p_i$ richiede la risorsa $r_j\} \cup \{(r_i, p_j) : r_i \in R \wedge p_j \in P \wedge r_i$ è stata allocata per $p_j\}$.

Nel secondo tipo di modello le relazioni sono rappresentate tramite delle tabelle, e non da un grafo. Entrambe le rappresentazioni sono complementari tra loro.

10.1 Tipologie di rirorse

- Risorse *riusabili*, che possono essere quindi riutilizzate senza che vengano deallocate dopo l'uso;
- Risorse *consumabili*, che una volta usate non possono essere riutilizzate;
- Risorse *condivisibili*, che possono essere riutilizzate senza ricorrere alla mutua esclusione.

Le situazioni di stallo si possono avere con le risorse riusabili e consumabili.

10.2 Condizioni di stallo

Consideriamo un insieme di processi $P := p_1, p_2, ..., p_n$ e un insieme di risorse $R := r_1, r_2, ..., r_m$, si potrà verificare una situazione di stallo se risultano vere quattro condizioni:

- *Mutua esclusione*: le risorse possono essere utilizzate da un solo processo alla volta se la risorsa ha una sola singolarità;
- *Processo e attesa*: i processi non rilasciano le risorse che hanno acquisito, e per continuare la loro esecuzione ne richiedono altre;

- *Mancanza di pre-rilascio (pre-emptive)*: le risorse che sono sono state già assegnate ai processi non possono essere revocate insieme ai processi;
- *Attesa circolare*: Se esiste un insieme di processi $p_k, p_{k+1}, ..., p_{k+h}$ tali che $\forall i \in [k, k+h-1], p_i$ è in attesa di una risorsa acquisita da p_{i+1}, e p_{k++h} è in attesa di una risorsa acquisita da p_k.

Le prime tre condizioni sono necessarie ma non sufficienti affinchè avvenga un deadlock, mentre la quarta è una condizione sufficiente e necessaria solo nel caso di risorse con una singolarità 1.

10.3 Metodi per il trattamento dello stallo

Esistono due tecniche per la prevenzione dello stallo: *prevenzione statica* e *prevenzione dinamica*. La prima consiste nello scrivere adeguatamente i programmi, in modo tale che le quattro condizioni per il deadlock non vengano mai verificate, e possiamo intervenire sul possesso e attesa, la mancanza di pre-emptive e l'attesa circolare.

10.4 Prevenzione dinamica

La prevenzione dinamica si basa su algoritmi che, dato lo stato corrente di allocazione delle risorse e alle richieste dei processi, verificano se l'assegnazione di risorse dovute ad una nuova richiesta da parte di un processo può portare ad una situazione di stallo. Uno tra i più noti algortmi è quello ideato da Dijkstra, ed è detto *algoritmo del banchiere*. Questo algoritmo rispetta i seguenti vincoli:

- Il SO può gestire un numero fisso di processi ed un numero fisso di risorse;
- I processi devono dichiarare inizialmente il numero massimo di risorse di cui hanno bisogno durante la loro esecuzione;
- I processi possono richiedere nuove risorse mantenendo le unità già in loro possesso;
- Tutte le risorse assegnate ad un processo sono rilasciate quando il processo termina la sua esecuzione.

In generale possiamo dire che tutti gli algoritmi di prevenzione dinamica si basano sul concetto di *stato sicuro*, Lo stato del sistema si dice sicuro se è possibile trovare una sequenza di assegnazione delle risorse ai processi, in modo tale che tutti i processi possano usare le risorse che richiedono e terminare in seguito. Nel caso in cui non esiste una sequenza sicura in cui tutto quello prima non è garantito, allora ci troviamo in uno *stato non sicuro*. Questo stato può portare al deadlock, e ne segue quindi che l'algoritmo deve consentire l'allocazione delle risorse ai processi solo quando le allocazioni portano a stati sicuri.

10.5 Rilevamento dei blocchi critici

Quello che di solito si fa quando si rileva una situazione di stallo è eliminare il blocco critico senza usare nessuna tecnica di prevenzione, come si fa in Windows e in Unix.

L'algoritmo di rilevazione viene eseguito in maniera periodica, con una frequenza che dipende dal tipo di applicazione, o quando, ad esempio, il grado di uso della CPU scende sotto una soglia critica, dato che il blocco critico può portare inefficienza nelle prestazioni del sistema.

Capitolo 11

Gestione della memoria principale

Con gestione della memoria principale indichiamo le tecniche di gestione della RAM e della memoria che viene allocata ai processi (essendo la RAM un componente importante, deve essere gestita in maniera efficiente). Il *gestore della memoria* deve tener traccia di quali parti di memoria sono in uso, allocare la memoria ai processi in base alle loro richieste e liberarla quando i processi terminano.

11.1 Creazione di un file eseguibile

Di norma un programma è costituito da un insieme di moduli scritti tramite un linguaggio di programmazione, e l'insieme dei moduli è detto *codice sorgente*. Ciascun modulo viene compilato producendo un modulo oggetto, il quale viene collegato con eventuali altri moduli e/o librerie tramite il *linker*, facendo sì che si produca il codice eseguibile, che viene poi caricato in memoria.

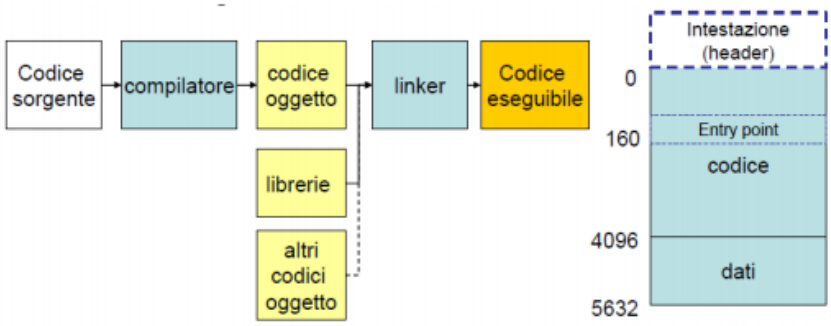

Figura 11.1: Creazione di un file eseguibile

11.2 Tecniche di gestione della memoria

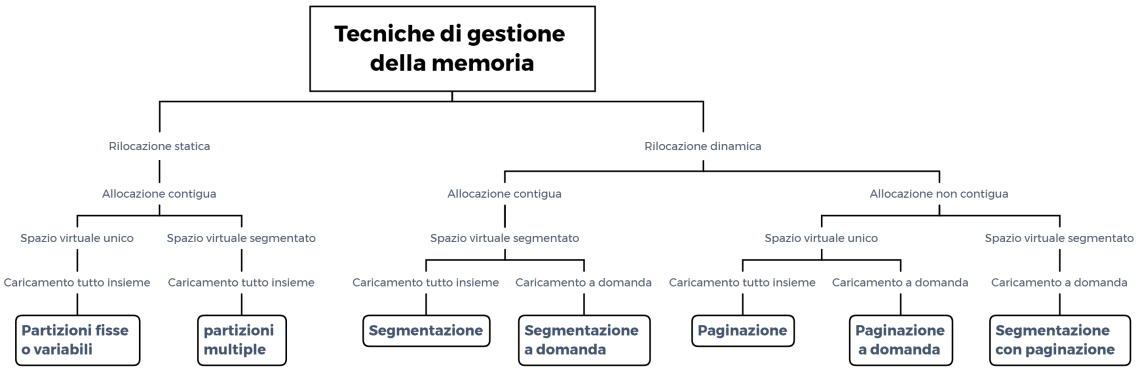

Figura 11.2: Tecniche di gestione della memoria

Lo *spazio virtuale di un processo* è la quantità di memoria necessaria per contenere il suo codice, i dati su cui opera e lo stack con l'heap. La struttura del processo sarà composta quindi da tre parti distinte, chiamati segmenti: il *segmento del codice*, il *segmento dati* e il *segmento dello stack*. L'insieme di tutti gli indirizzi virtuali è detto *spazio degli indirizzi virtuali*, mentre l'insieme di tutti gli indirizzi fisici, ossia degli indirizzi di memoria fisica, è detto *spazio degli indirizzi fisici*. I due spazi di indirizzi sono legati tra di loro da una *funzione di rilocazione*.

Rilocazione statica e dinamica

Le tecniche per effettuare la rilocazione degli indirizzi sono di solito due, *rilocazione statica* e *rilocazione dinamica*. La rilocazione statica è una tecnica semplice per cui tutti gli indirizzi virtuali sono rilocati prima che il processo inizi la sua esecuzione. La rilocazione dinamica esegue una traduzione degli indirizzi da virtuali a fisici nell'esecuzione del processo e non all'inizio, pertanto il caricatore trasferisce in memoria direttamente il contenuto del codice eseguibile che contiene gli indirizzi virtuali senza apportare modifiche. L'architettura di un un processore che consente di effettuare la rilocazione dinamica non può essere di un tipo qualsiasi, deve avere infatti un supporto hardware che è di solito chiamata *MMU (Memory Management Unit)*, che implementa la funzione di rilocazione.

Spazio virtuale unico o segmentato

Per organizzare lo spazio virtuale esistono vari modi: *spazio virtuale unico* e *spazio virtuale segmentato*. Il primo organizza un unico spazio di memoria, in cui vengono messi tutti gli indirizzi virtuali. Il secondo, invece organizza lo spazio di memoria virtuale rappresentandolo come un gruppo di indirizzi indipendenti, detti *segmenti*, dividendo di solito codice, dati e stack in tre diversi segmenti. Qui il linker può essere realizzato in maniera tale che generi il modulo eseguibile con tre moduli separati. Ad ogni segmento viene dato un indirizzo che ne identifica la posizione nell'immagine del processo. Se si ha un'architettura che può indirizzare più segmenti, allora questa prende il nome di *architettura con indirizzamento segmentato*.

11.2.1 Memoria partizionata

Le tecniche di memoria partizionata prevedono che la memoria sia suddivisa in più partizioni. Si utilizza la *rilocazione statica* operando su immagini di processo con *spazio virtuale unico*

Capitolo 12

Funzioni in POSIX

12.1 Librerie

La seguente è una lista di librerie usate per la gestione dei processi, dei thread e della loro mutua esclusione.

```
#include <stdlib.h>
#include <pthread.h>
#include <sys/types.h>//per i processi(definisce il tipo di dati per i processi e le funzioni come getpid()
#include <sys/stat.h>//per le code di messaggi
#include <sys/shm.h>
#include <sys/ipc.h>
#include <fcntl.h>//per le code di messaggi(serve a controllare i file)
#include <mqueue.h>//per le code di messaggi(per dare una regola alla struttura delle code di messaggi in memoria)
#include <unistd.h>//per i processi
#include <string.h>//per le stringhe
#include <semaphore.h>//per i semafori
#include <sys/wait.h>
```

12.2 Code di messaggi

12.2.1 Definizione della struttura

La seguente è la definizione di una coda di messaggi.

```
struct mq_attr{
  long mq_flags; //flag che indica se ha un comportamento bloccante o meno, quindi o 0 o O_NONBLOCK (bloccante o non bloccante)
  long mq_maxsize; //numero massimo di mesaggi che posso memorizzare
  long mq_msgsize; //la dimensione massima che un messaggio puo' avere
  long mq_curmsgs; //il numero di messaggi che sono ora in coda
};
```

12.2.2 Funzioni

Le funzioni principali che andremo a vedere sono *mq_open* e *mq_close* per l'apertura e la chiusura della coda di messaggi, e *mq_send* e *mq_receive* per inviare e ricevere messaggi. Da notare che il tipo di dato utilizzato è *mqd_t*.

```
mqd_t mq_open(const char *name, int oflag); //creazione di una coda di messaggi bloccante
mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr); //creazione di una coda di messaggi di cui si definisce il comportamento
//alla coda //nel campo oflag mettiamo O_RDONLY o O_WRONLY o O_RDWR, per tutte le altre caratteristiche sono passate dalla struct (vedi mq_attr)

ssize_t mq_recive(mqd_t mqdes, char *msm_ptr, size_t msg_len, unsigned int *msg_prio); //per poter usare la receive dobbiamo avere il descrittore della coda, il puntatore al messaggio, la sua lunghezza e a priority (numero negativo)

ssize_t mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int *msg_prio); //per poter usare la send dobbiamo avere il descrittore della coda, il puntatore al messaggio, la sua lunghezza, e la sua priority (numero negativo)

int mq_unlink(const char *nome_coda); //dobbiamo avere il nome della coda uguale a quello usato nella sua apertura
```

12.3 Semafori e mutex

Sia i semafori che i mutex sono degli strumenti di sincronizzazione nella mutua esclusione sia per processi che per thread. Per essere creati usano la funzione di *sem_init* e *pthread_mutex_init*, e *sem_t* è il tipo di dato che rappresenta il descrittore del semaforo e *pthread_mutex_t* è quello che rappresenta il descrittore del mutex.

12.3.1 Funzioni del mutex

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr); //inizializza il mutex con un puntatore alla struttura e ai suoi attributi
int pthread_mutex_lock(pthread_mutex_t *mutex);// blocca il mutex e per poterlo fare deve avere il suo descrittore
int pthread_mutex_trylock(pthread_mutex_t  *mutdesc); //versione non bloccante della funzione lock che prova a bloccare il mutex
int pthread_mutex_unlock(pthread_mutex_t *mutdesc);// sblocca il mutex e pe poterlo fare deve avere il suo descrittore
int pthread_mutex_destroy(pthread_mutex_t  *mutdesc); //dealloca il mutex rendendolo non piu' utilizzabile
```

12.3.2 Funzioni del semaforo

```
int sem_init(sem_t *semdescr, unsigned int pshared, unsigned int first_value); //per poter creare un semaforo dobbiamo avere il suo descrittore, un valore che ci dice che quel semaforo verra' condiviso con altri thread o altri processi (0 condiviso tra thread, 1 condiviso tra processi) e il valore iniziale del semaforo.
int sem_wait(sem_t *semdesc); //per poter mettermi in attesa come processo o thread devo sapere su quale semaforo mi sto  mettendo in attesa, quindi devo avere il descrittore(ooviamente nell'operazione di attesa io decremento il valore del semaforo)
int sem_trywait(sem_t *semdesc); //versione non bloccante della wait del semaforo
```

```
4 int sem_post(sem_t *semdesc); // incremento il valore del semaforo ed
   esco dalla sezione critica
5 int sem_destroy(sem_t *semdesc); //dealloco il semaforo (0 in caso di
   successo, 1 in caso di errore)
6 int sem_getvalue(sem_t *semdesc, int *sval); //ritorna sempre 0, ma se
   passo come parametro il puntatore di un intero nel parametro *sval
   allora potro' avere il valore del semaforo nella variabile passata.
```

12.4 Variabili conditon

Sono elementi di sincronizzazione per i thread, e queste variabili condition sono sempre associate ad un mutex. Sono inizializzati e distrutti dalle funzioni *pthread_cond_init* e *pthread_cond_destroy*, e queste variabili sono rappresentate da un tipo di dato *pthread_cond_t* che appunto rappresenta il loro descrittore.

```
1 int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *
   cond_attr) //si passa il puntatore alla variabile condition e uno alla
   struttura che contiene gli attributi della condizione (se settato a NULL
    si utilizzano gli attributi base); questa funzione alloca le risorse
   per la variabile condition
2 int pthread_cond_destroy(pthread_cond_t *cond) //si passa il puntatore
   alla variabile condition e dealloca tutte le risorse della variabile
   condition; questa funzione ritorna 0 in caso di successo o un numero
   diverso da 0 in caso di errore
3 int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex) //si
   passa un puntatore all'istanza di condition che rappresenta la
   condizione di sincronizzazione e il puntatore al mutex che gestisce la
   risorsa critica, questa funzione ritorna sempre 0; questa funzione fa da
    wait al thread in esecuzione
4 int pthread_cond_signal(pthread_cond_t *cond) //si passa il puntatore all
   'istanza di conditionche rappresenta la condizione di sincronizzazione,
   questa funzione ritorna sempre 0 e in questo caso non utilizziamo il
   mutex e questa funzione svolge l'operazione di signal sul primo thread
   in coda di pronto
5 int pthread_cond_broadcast(pthread_cond_t *cond) //si passa il puntatore
   all'istanza di condition che rappresenta la condizione di
   sincronizzazione, questa funzione ritorna sempre 0 e in come
   pthread_cond_signal, questa funzione svolge l'operazione di signal ma al
    contrario di sbloccare solo un thread, sblocca tutti i thread in coda
   di pronto
```

Parte II

Reti di calcolatori

Capitolo 13

Introduzione

Internet è una *rete di reti* che collega dispositivi diversi in tutto il mondo, detti *host*. Gli host sono connessi tra loro mediante *linee di comunicazione (link)* e dispositivi di commutazione, tra i quali *router* e *switch*. Un parametro che caratterizza le varie tecnologie di link è la *velocità di trasmissione* (o *larghezza di banda*). In internet di usa prevalentemente la *commutazione di pacchetto*. Un host, per inviare i dati ad un altro host, li frammenta in piccole parti e aggiunge a ciascuna parte un'intestazione: l'informazione ottenuta prende il nome di *pacchetto*. Un pacchetto compie un tragitto dall'host sorgente fino all'host ricevente: questo viene detto *percorso (path)* o *cammino (route)*. Gli host sono connessi ad Internet attraverso un *Internet Service provider (ISP)*. I diversi ISP si caratterizzano per le diverse tecnologie di accesso alla rete. I suddetti ISP, detti di livello inferiore, sono interconnessi attraverso ISP di livello superiore. Esistono anche reti i cui host sono isolati dalla rete pubblica (mediante *firewall* e tecnologie hardware/software). Queste reti sono chiamate *Intranet*.

13.1 Protocolli di rete

Gli host, i router e altri dispositivi usano molti protocolli per scambiarsi dati tra loro. I due protocolli principali di internet sono il *TCP (Transmission Control Protocol)* e l'*IP (internet Protocol)*, i quali sono spesso chiamati con l'unico termine *TCP/IP*. Un protocollo definisce:

- Le strutture dei messaggi e l'ordine di trasmissione;
- Le operazioni eseguite al momento della trasmissione/ricezione di messaggi.

13.2 Apllicazioni client e server

il modello *client/server* è il modello più usato per realizzare le applicazioni di rete. Un'applicazione di rete si realizza in due parti: il *client* (host che richiede servizi) e il *server* (host che fornisce servizi)

Capitolo 14

Serivizi orientati alla connessione e servizi senza connessione

Le applicazioni sviluppate per Internet utilizzano protocolli basati su due tipi di servizio:

- Il servizio *orientato alla connessione* è affidabile e garantisce che i dati trasmessi da un host mittente giungeranno al destinatario senza errori e nello stesso ordine con cui sono stati spediti,
- Il servizio *senza connessione* non è affidabile e non garantisce che tutti i dati inviati dall'host mittente giungeranno a destinazione.

Internet può consentire servizi in *soft real-time* ma non fornisce servizi in *hard real-time*, cioè servizi che garantiscano che i dati inviati dal mittente arrivino al destinatario entro una *deadline*.

14.1 Servizio orientato alla connessione

Le applicazioni client/server (prima id inviare i dati veri e propri) eseguono una procedura detta *handshaking* mediante la quale si scambiano informazioni di controllo. La connessione di considera instaurata al termine della procedura di handshaking. Il servizio orientato alla connessione è aloquanto complesso ed è realizzato in un insieme di servizi, come il *trasferimento di dati affidabile*, il *controllo di flusso* ed il *controllo della congestione*. Il trasferimento di dati si dice *affidabile* quando un'applicazione è in grado di trasmettere tutti i dati che invia senza errori e nell'ordine di partenza. L'affidabilità è ottenuta attraverso l'invio di *messaggi di riscontro* ed eventuali *ritrasmissioni*. Il servizio di *controllo del flusso* garantisce che il mittente non saturi il buffer di ricezioni del destinatario. Il servizio di *controllo della gestione* consente di prevenire che la rete entri in uno stato di congestione a causa del traffico di pacchetti eccessivo. Il TCP è il protocollo standard usato in Internet che implementa il servizio orientato alla connessione.

14.2 Servizio senza connessione

Il servizio senza connessione non esegue la procedura di handshake, il che porta ad una velocità di trasmissione più elevata, ma con una mancata garanzia dell'avvenuta ricezione

dei pacchetti, e privo dei controlli di flusso e di congestione. L'*UDP (User Datagram Protocol)* è il protocollo usato in Internet che implementa il servizio senza connessione.

Capitolo 15

La subnet della rete

La *subnet* della rete è costituita da un insieme di router e di link che li collegano. Esistono due principali tecnologie per realizzare le subnet: la commutazione di *circuito* e la commutazione di *pacchetto*.

15.1 Commutazione di circuito

Le risorse sono assegnate durante la fase iniziale della comunicazione (fase di *setup*), e restano allocate e riservate durante la comunicazione. Quando si stabilisce un circuito, esso sarà a velocità costante garantita. Quando la larghezza di banda di un link è superiore alla larghezza di banda necessaria si può usare la *multiplazione* (a divisione di *frequenza* o a divisione di *tempo*). La multiplazione a divisione di frequenza (*FDM*) divide la banda di frequenza B_L di un link fisico in N circuiti logici, in modo che sia $B_L \geq \sum_i B_{C_i}$. Le sotto-bande B_{C_i} dei circuiti logici possono avere misure diverse tra loro. Nella multiplazione a divisione di tempo (*TDM*), il tempo di utilizzo del link è suddiviso in intervalli di durata fissa detti *frame*, ciascuno dei quali è ulteriormente suddiviso in *slot*. Per l'FDM , la larghezza do banda B_L del link è suddivisa in N bande, ciascuna con larghezza B_C. Per il TDM, il dominio temporale è suddiviso tra N circuiti con N slot in ciascun frame.

15.2 Commutazione di pacchetto

Il mittente divide il messaggio in frammenti di piccole dimensioni, detta *pacchetti*. Inoltre, le risorse non sono riservate, ma sono *condivise*. Molti router utilizzano la trasmissione *store-and-forward*, la quale prevede che il router deve ricevere l'intero pacchetto prima di poterlo ritrasmettere su un link in uscita. Questi router introducono un *ritardo di trasmissione*. Altri router utilizzano la trasmissione *cut-throw* che consente di iniziare il rilancio del pacchetto appena sono stati elaborati solo alcuni campi. Un router è collegato a vari link. Per ogni link il router ha un *buffer di uscita*, che memorizza i pacchetti che devono essere rinviati su quel determinato link. Quindi, oltre al ritardo di trasmissione, i pacchetti subiscono il ritardo dovuto alla presenza di altri pacchetti nel buffer di uscita (*ritardo di coda*). Quando un messaggio viene suddiviso in pacchetti, si dice che la rete effettua in *pipeline* la trasmissione dei messaggi: ciò significa che parti del messaggio vengono trasmesse in parallelo.

15.3 Confronto tra commutazione di circuito e pacchetto

La commutazione di pacchetto è poco adatta per un servizio in tempo reale, a causa del suo ritardo variabile e non prevedibile, a differenza della commutazione di circuito. La commutazione di pacchetto offre però un miglior utilizzo della larghezza di banda rispetto alla commutazione di circuito: è più semplice ma anche più efficiente e meno costosa della commutazione di circuito.

15.4 Reti a commutazione di pacchetto datagram

Le reti a commutazione di pacchetto possono essere di tipo *datagram* o a *circolo virtuale*. In una rete datagram, ciascun pacchetto contiene nella sua intestazione l'*indirizzo IP* del destinatario e l'indirizzo del mittente. Quando un pacchetto arriva ad un router, questo analizza l'indirizzo del destinatario e invia un pacchetto ad un router adiacente. Ciascun router ha una *tabella di instradamento* che mette in corrispondenza l'indirizzo con un link di uscita.

Capitolo 16

Mezzi trasmissivi

16.1 Il Doppino

È il meezzo trasmissivo più economico, e consiste in due fili, isolati tra loro, avvolti a spirale (l'avvolgimento consente di ridurre le interferenze elettriche). Varie coppie di doppino sono usate per realizzare varie categorie di cavo: questo cavi sono chiamati *cavi ethernet*. Tutti i tipi di cavo ethernet usano il connettore RJ45, e si differenziano per le prestazioni e per il livello di immunità alle interferenze.

Tipo	Velocità massima di trasmissione	Larghezza di banda	Interferenze
CAT 5	100 Mbps	100 MHz	/////
CAT 5e	10 Gbps	100 MHz	/////
CAT 6	10 Gbps	250 MHz	/////
CAT 6a	10 Gbps	500 MHz	Ridotte
CAT 7	10 Gbps	600 MHz	Minime

Figura 16.1: Struttura interna del cavo ethernet e possibili schermature

16.2 Cavi coassiali

Consistono di due conduttori in rame concentrici, ed hanno una velocità di trasmissione maggiore rispetto ai doppini. Sono di due tipi: cavi coassiale in *banda base* e in *banda traslata*

16.3 Fibre ottiche

È un sottile mezzo di vetro o plastica che conduce impulsi di luce. Consente velocità di trasmissione superiori alle *centinaia di Gbps* e sono immuni alle interferenze elettromagnetiche.

16.4 Canali radio terrestri

I canali radio trasmettono segnali tramite onde elettromagnetiche. Possono essere classificati in due gruppi: canali per *reti in area locale* e canali in *area geografica*.

16.5 Canali radio satellitari

Un satellite collega due o più trasmettitori a microonde situati sulla Terra, detti *stazioni al suolo*. Il satellite riceve le trasmissioni su una banda di frequenza e le ritrasmette si un'altra banda, con una larghezza di banda dell'ordine dei Gbps. Esistono vari tipi di satelliti:

- *Satelliti geostazionari*: a 36000 km dalla Terra, hanno un ritardo di propagazione di circa 240 ms con una velocità di trasmissione di centinaia di Mbps;
- *Satelliti ad orbita bassa*: a 160/2000 km dalla terra, hanno un ritardo di propagazione di 20-25 km.

Capitolo 17

Accesso alla rete Internet

17.1 Accesso residenziale

Le tecnologia usate per l'accesso residenziale sono la *DSL* e l'*HFC*.

17.1.1 DSL

DSL è una famiglia di tecnologie progettata per funzionare con le reti telefoniche cablate esistenti. Le velocità di trasmissione sono asimmetriche nelle due direzioni (maggiore per il download e minore per l'upload). La DSL utilizza tecniche FDM, in quanto la larghezza di banda è divisa in tre bande di frequenza: canale *telefonico*, canale in *upstream* e canale in *downstream*. Le larghezze di banda dei singoli canali variano dal tipo di tecnologia DSL. I canali sono usati in parallelo, e ciò porta ad avere una velocità di trasmissione non costante. I termini *FTTS* (o *FTTC*) indicano collegamenti nei quali si usa la fibra ottica tra centrale e cabinet e il doppino tra cabinet e modem DSL, tratto che è di lunghezza inferiore ai 300 m). Con *FTTH* si indicano collegamenti in cui la fibra ottica è usata in entrambe le tratte.

17.2 Accesso aziendale

Una rete *LAN* viene utilizzata per collegare i computer ad Internet, e la tecnologia di accesso più diffusa è la rete *Ethernet*: la più recente (*Ethernet commutata*) usa gli *switch* per connettere tra loro i cumputer ed un *router di default* che si occupa di instradare i pacchetti all'esterno della LAN.

17.3 Accesso wireless

Esistono due ampie classi di accesso wireless ad Internet:

- Wireless *LAN*, in cui gli utenti con dispositivi mobili comunicano con una stazione base (*punto di accesso*) wireless entro un raggio di 100 m;
- Wireless *in area geografica*, in cui la stazione base serve gli utenti entro un raggio di 10 km.

17.4 ISP e reti dorsali di Internet

Le reti di accesso Internet sono connesse al resto di Internet tramite una gerarchia a livelli di *ISP*. Gli ISP di *livello 1* costituiscono la *rete dorsale* di Internet. Gli ISP di accesso residenziali e aziendali costituiscono i livelli più esterni di questa gerarchia. I punti nei quali un ISP si connette ad altri prendono il nome di *punti di presenza (POP)*. La tipologia di Internet è formata da decine di ISP di livello 1 e 2 e migliaia di ISP di livello inferiore.

Capitolo 18

Intensità del traffico

Ad ogni router è associata una *coda* (buffer di uscita) che contiene tutti i pacchetti ricevuti per poterli poi instradare. La coda è gestita con politica FIFO. Quando un pacchetto è trasmesso da un nodo all'altro subisce vari tipi di ritardo:

- Ritardo di *elaborazione* (trascurabile);
- Ritardo di *coda* o di *attesa* (variabile tra i microsecondi e i millisecondi);
- Ritardo di trasmissione (dai microsecondi ai millisecondi);
- Ritardo di propagazione (dell'ordine dei millisecondi).

La somma costituisce il *ritardo totale*. Tutti questi ritardi si verificano su un singolo router. Supponiamo di avere n link tra gli host sorgente e destinazione, i ritardi dei nodi si sommano forniscono il ritardo *punto-a-punto*: $r_{pp} = \sum r_{e_i} + r_{t_i} + r_{p_i} + r_{c_i}$ (solo nel primo link sono presente i ritardi r_t ed r_p). L'*intensità del traffico* è data dal rapporto: $I_t = l * \frac{p_{\text{avg}}}{s}$, dove l è la dimensione media dei pacchetti, p_{avg} è il numero medio di pacchetti al secondo posti nella coda e s è al velocità di trasmissione del link di uscita.

Capitolo 19

Strati protocollari e modelli di servizio

Per ridurre la complessità di progetto e di realizzazione, i protocolli sono organizzati a *stratio livelli*. Ogni livello fornisce servizi al livello sottostante: ogni livello è implementato in maniera indipendente. La pila protocollare di internet è costituita da 5 livelli: *applicazione*, *trasporto*, *rete*, *collegamento* e *fisico*.

Strati protocollari

Applicazione
Trasporto
Rete
Collegamento
Fisico

Messaggio

Messaggio
Segmento
Datagram
Frame
1-PDU

19.1 Il modello OSI

È basato su 7 livelli: i due livelli in più sono il livello di *sessione* e di *presentazione*. Il livello di presentazione fornisce servizi che consentono di interpretare il significato dei dati scambiati, mentre il livello di sessione fornisce la delimitazione e la sincronizzazione dello scambio dei dati. Questi due livelli, mancanti nella pila protocollare di Internet, sono implementati al livello applicazione.

Modello OSI

Applicazione
Presentazione
Sessione
Trasporto
Rete
Collegamento
Fisico

Strati protocollari

Applicazione
Trasporto
Rete
Collegamento
Fisico

19.2 livelli dello strato porotocollare

19.2.1 Livello di applicazione

Il livello di applicazione consiste in applicazioni di rete (esempio: HTTP, SMTP, FTP, etc.).

19.2.2 Livello di trasporto

Il livello di trasporto fornisce il servizio di trasporto dei messaggi del livello applicazione fra le estremità di un'applicazione. I protocolli di trasporto più usati in Internet sono il *TCP* e l'*UDP*. Il *TCP* fornisce alle app un servizio orientato alla connessione *affidabile*, fornisce il *controllo del flusso* e il *controllo della congestione*. Il TCP frammenta i messaggi superiori ad una determinata dimensione in segmenti più piccoli nel lato mittente e li riassembla nella destinazione. L'*UDP* fornisce alle app un servizio *senza connessione* e trasmette i dati senza alcuna garanzia. Se il messaggio è troppo grande, deve essere frammentato a livello di applicazione.

19.2.3 Livello di rete

Il livello di rete è responsabile dell'instradamento dei datagram. Ha due componenti principali:

- *Protocollo IP*: definisce i campi del datagram IP e le operazioni che i due host e router eseguono su questi campi;
- *Protocolli di instradamento*: decidono il percorso che i datagram devono seguire fra sorgente e destinazione.

19.2.4 Livello di collegamento

Il livello di rete utilizza i servizi del livello di collegamento per instradare i pacchetti

19.2.5 Livello fisico

Il compito dello strato fisico è quello di trasmettere realmente i segnali fisici corrispondenti al bit del frame da un nodo al successivo.

Capitolo 20

Livello di applicazione

20.1 Architetture e protocolli del livello di applicazione

L'architettura stratificata consente a processi su host diversi di comunicare tra loro *scambiandosi messaggi*. Le applicazione di rete, per comunicare, devono utilizzare *protocolli* che definiscono il *formato* dei messaggi, l'*ordine* in cui essi sono scambiati e le *operazioni* da svolgere nella fase di trasmissione e ricezione dei messaggi. Le principali architetture delle applicazioni di rete sono *client/server* e *P2P*. Nell'architettura i client non comunicano direttamente tra loro (si indica con *client* l'host che richiede l'instaurazione della comunicazione, e *server* l'host con cui il client vuole instaurare la comunicazione). Nell'architettura *P2P* gli host connessi ad una rete sono chiamati *peer* e possono scambiarsi dati direttamente tra di loro (ogni peer si comporta sia da client che da server).

20.2 Indirizzamento dei processi

Un processo è specificato da due informazioni: l'*IP* dell'host (32/128 bit) su cui il processo è attivo e un'identificatore del processo dell'host, ovvero il *numero di porta* (16 bit). Un processo comunica nella rete attraverso un'interfaccia software con il livello di trasporto detta *socket*.

20.3 Servizi forniti dai protocolli di trasporto

Il TCP fornisce alle applicazioni un servizio di trasferimento affidabile dei dati. La comunicazione è di tipo *full-duplex*, e fornisce anche un servizio di controllo della congestione: si deduce che il TCP non garantisce una velocità minima di trasmissione dei dati. L'UDP fornisce un servizio di trasferimento dati non affidabile, pertanto i messaggi che arrivano alla socket ricevente possono non arrivare in ordine o non arrivare completamente. L'UDP non implementa il servizio di controllo della congestione, quindi un processo può inviare dati fino alla massima velocità consentita, ma non si hanno garanzie sul ritardo.