CAPITOLO 1
Introduzione a Unix

Cos'è UNIX? Unix è il nome di una famiglia di sistemi operativi disponibili su diverse piattaforme hardware. Unix è un sistema **multi-utente** o multi-user (può essere utilizzato da più utenti contemporaneamente), **multi-processo** o multi-tasking (può lanciare ed eseguire più processi contemporaneamente, con le dovute accortezze) e **time-sharing** (il sistema selezione i processi da eseguire in un lasso di tempo uguale per tutti attorno ai 10-100 ms).

Struttura di UNIX. La struttura di Unix è a strati. *L'utente* interagisce col sistema e quindi con il kernel attraverso una **SHELL** (interprete dei comandi) e con le **chiamate a sistema** (system calls). Il **KERNEL** si occupa invece di gestire tutto il resto, ovvero il file system, la schedulazione dei processi (ovvero il "ragionamento" secondo il quale un processo viene candidato all'esecuzione rispetto ad altri), la memoria, gli interrupt, gli errori, il servizio di input ed output e l'accounting di sistema.

Linux. Linux appartiene alla famiglia dei sistemi Unix-Like. Creato da **Linus Torvald** nel 1991 non costituisce un sistema commerciale e viene distribuito sotto la GNU Public License. Linux possiede un vero kernel Unix ma non è un sistema Unix a tutti gli effetti, infatti non include tutte le applicazioni presenti in un sistema operativo commerciale. Esistono comunque versioni commerciali di Linux che si avvicinano al massimo al concetto di sistema di tipo Unix.
Il kernel è di tipo **monolitico** e prevede l'impiego di **moduli**; il **threading** a livello del kernel è molto limitato ma è supportato il multi-threading a livello applicativo; Linux non ha un **preentive kernel** e ciò fa sì che un processo avviato in modalità privilegiata non possa essere sospeso arbitrariamente; supporta diversi tipi di **file system**; è predisposto per supportare i **multiprocessori**.

CC di GNU sotto Linux. Il **gcc** di Linux (conosciuto anche come **cc** di GNU) costituisce un compilatore per tutti i programmi scritti in C, C++ e Objective C. Fa il "mestiere" di ogni buon compilatore facendo sì che il programmatore abbia sotto controllo ogni fase della compilazione (fase di preprocessing, compilazione, linking, assemblaggio). Gestisce ogni dialetto del C e applica un controllo di qualità e informazioni di debugging al codice che gli viene sottoposto, oltre ad ottimizzare il codice. Quando si scrive a riga di comando **gcc -o file.c file** dapprima il file.c viene maneggiato dal preprocessore che ne espande le macro e le predispone per l'integrazione con i file di libreria inclusi nel sorgente, dopodiché viene creato un file oggetto mediante compilazione e per ultima cosa il linker lo assembla fornendo un file eseguibile contenente il programma. Un programmatore produce sempre due tipi di errore:

- ✔ **Errori logici**: si realizzano in fase di esecuzione, procurando arresti inattesi e prematuri del programma o aberrazioni di risultati da lo stesso, o ancora possono impedire al programma di funzionare come dovrebbe;
- ✔ **Errori sintattici**: si presentano in fase di compilazione, quelli che impediscono al compilatore di compilare il codice, vengono segnalati con precisione e dopo la loro correzione sarà possibile ottenere un file eseguibile.

Scovare gli errori logici è molto difficile, è possibile allora servirsi di un programma di debug che possa facilitare tale procedura. In Linux esiste **GBD** (GNU Debugger).

Makefile. Spesso i programmi sono composti da più file sorgenti interlacciati tra loro. Attraverso la compilazione manuale di un **makefile** possiamo mandare in compilazione tutti i file in successione senza mandare manualmente il comando alla shell, in più il comando **make** ci aiuta a ricompilare solamente quei file che sono stati interessati a qualche modifica senza dover per forza ricompilare tutti i file collegati.
Un **makefile** costituisce un insieme di **regole** e **dipendenze**, dove le regole sono un *target* (ovvero l'elemento che il make deve creare), un *elenco di una o più dipendenze* da rispettare nella suddetta creazione, un *elenco di comandi* da eseguire per la creazione del file di target. Il

makefile indica inoltre l'ordine con cui i file di target devono essere creati ed in qualche modo tale creazione deve avvenire.

CAPITOLO 2

Il Sistema Operativo: il File System e System Calls

Risorse di sistema. Il kernel fornisce un accesso alle risorse di sistema (stando esso proprio tra l'interfaccia utente e la macchina fisica), tra cui il **processore** (*assegnato e tolto dal kernel ad ogni programma*); le **periferiche input/output** (*tutti i dati per o da un programma passano attraverso il kernel*); la **gestione dei processi** (*è il kernel che decide quando e quanti crearne*); la **memoria** (*che gestisce il kernel*); tutte le **altre periferiche**; il **timer**; la **rete**; la **comunicazione tra i processi**.

File System. La filosofia di Linux considera qualsiasi cosa alla stregua di un file. Per cui il file system contiene dati senza una specifica struttura (sono sequenze di byte), i file di dati sono protetti e tutti i dispositivi periferici sono trattati come file. I **tipi di file** presenti sono: **file di flusso** interpretato di volta in volta dai programmi (*sono sequenze di byte tutti i file tranne quelli di dispositivo/speciali*), **directory file** che contengono le informazioni dei file presenti in esse, **file speciali** che trattano i dispositivi come file. Ogni file inoltre possiede un **i-number**, ovvero un numero identificativo univoco (contenuto nell'**i-node**). Ogni file è descritto dal suo personale **i-node**: poiché il file non contiene nessun indicatore riguardo le sue caratteristiche reali a parte il suo contenuto, deve appoggiarsi ad una struttura esterna che ne descriva *numero di file*, *tipo di file*, *lunghezza del file*, *ID del gruppo e dell'utente proprietario*, *diritti d'accesso*, *tempo di creazione e modifica*, *file system proprietario* e tanto altro. Il file system è organizzato in una struttura gerarchica al capo (o radice, se la vediamo come un albero) sta la directory **root**.

- ✔ **Directory**. Sono sequenze di byte ma differentemente dai file normali non possono essere scritte da programmi ordinari, contengono una serie di directory entries che definiscono le associazioni tra i nomi usati dall'utente (=**file name**, ne possono esistere più per uno stesso file) per chiamare i file ed il loro i-number;
- ✔ **Pathname**. Ogni file ne possiede uno diverso che ne identifica univocamente la posizione partendo dalla cartella root;
- ✔ **Home directory**. Ogni utente ne ha una che le viene assegnata dal sistema. Ad essa possono sottostare tutte le cartelle definite dall'utente.
- ✔ **Working directory**. E' la cartella entro la quale stiamo lavorando, ogni file può essere specificato direttamente con il pathname relativo a tale cartella. Tale cartella è ovviamente intercambiabile.

Implementazione dei file. Ogni file possiede un descrittore che mantiene in esso le seguenti informazioni relative allo stesso: **tipo** di file, **posizione** nel file system, **dimensione** del file, numero di **links**, **proprietario**, **permessi**, tempo di **creazione**, tempo di **accesso** e tempo di **modifica**. Tutti i link ad uno stesso file hanno uguale importanza e non è possibile distinguere il file originale dai nuovi link. Tali link sono proibiti tra file system differenti.

System Calls. Per utilizzare tutte le funzionalità del sistema, è possibile dialogare con esso attraverso le **system call** (chiamate di sistema). Tali funzioni sono state standardizzate nel POSIX, documento che determina l'interfaccia in C di tali funzionalità e ne generalizza il nucleo di base. I sistemi POSIX do solito integrano il loro standard con funzioni extra in ausilio alle esistenti (ma che non fanno parte di tale standard, includendo anche primitive delle librerie C). Lo standard POSIX è portabile: una volta ricompilato un sorgente coerente alle specifiche POSIX, è possibile riutilizzarlo anche in altri ambienti POSIX differenti da quello natio dotato di un ambiente di programmazione C standard. Come nella tradizionale programmazione in C, all'inizio di ogni sorgente vanno incluse le librerie POSIX necessarie

all'utilizzo delle primitive scelte.

- ✔ **Gestione degli errori**. Quando una system call fallisce tende a tornare al chiamante il valore -1, assegnando di conseguenza un valore specifico alla variabile **extern int errno**, ovvero il codice corrispondente all'errore avvenuto. Per capire meglio in che errore si è incappati, è necessario leggere il valore interno alla variabile di cui sopra. Errno non viene mai azzerato, quindi è sbagliato utilizzarlo per capire a seconda del suo valore se la chiamata è fallita oppure no. Uso la system call **perror()** per leggere il valore di errno.
- ✔ **Gestione dei file**. Per utilizzare un file prima bisogna aprirlo, comportamento ottenibile con l'utilizzo della funzione **open()** che individua il file attraverso il suo pathname, copia in memoria il suo i-node e associa al file un numero intero che lo identifichi (file descriptor) nelle successive operazioni. Con **close()** libero il file descriptor e lo rendo disponibile per altri usi. La funzione open() crea un oggetto "open file" e restituisce il "file descriptor". L'oggetto "open file" contiene alcune strutture per gestire il file ed altro; il "file descriptor" rappresenta l'interazione tra il file ed il processo. Più processi possono aprire lo stesso file, associando ad esso diversi file descrpitor, ed il sistema non tutela le interazioni invasive tra questi diversi processi. Quando un processo termina tutti i file che ha aperto vengono chiusi automaticamente.

CAPITOLO 3

Il Sistema Operativo: il File System e la gestione dei file

Struttura di un disco. Unix divide lo spazio di un disco rigido in sezioni contigue chiamate volumi o partizioni (**dischi logici**). Ogni disco logico contiene un solo file system (ed un file system può stare in un solo volume) e la loro dimensione è decisa all'atto di creazione di tali partizioni. Il comando ***mount*** collega un file system esterno ad un'altro file system (***unmount*** fa la cosa opposta). I volumi sono suddivisi in blocchi di **dimensione fissa** (multiplo della dimensione reale di un settore fisico del disco) e tali blocchi sono **l'unità base di gestione dei file** (un file deve essere grande almeno un blocco e il sistema accede in lettura a uno o più blocchi alla volta).

Struttura di un volume. Un volume (o disco logico) si compone di:

- ✔ **Blocco di boot**. Si trova in testa al settore ed indica al sistema il codice di bootstrap da leggere quando avviene l'operazione di boot.
- ✔ **Superblocco**. Descrive lo stato di un file system, la sua dimensione, dove trovare spazio libero e tanto altro. I blocchi liberi si trovano nella loro apposita lista e quelli allocati possono non essere continui.
- ✔ **Lista degli i-node**. Lista a cui fa riferimento il kernel quando cerca e legge gli i-node dei dati contenuti nel file system (si fa riferimento ad ognuno tramite il suo i-number).
- ✔ **Blocchi di dati**. Presupposto che un blocco può essere allocato ad un solo file, qui si trovano i dati sui file e quelli relativi alla gestione del sistema.

Utenti e gruppi. Unix assegna ad ogni utente un nome (**user name**) ed un numero identificativo (**user-id**) entrambi pubblici, stessa cosa fa con i gruppi che possono essere creati dall'amministratore di sistema anch'essi corredati di **group name** e **group-id**. Ogni utente può appartenere ad uno o più gruppi.

Sicurezza in Unix. L'**accesso al sistema** è consentito agli utenti solamente tramite login/logout; l'**accesso ai file** è consentito solo a coloro che soddisfano i criteri dei requisiti d'accesso; l'**accesso ai processi** e consentito solo agli utenti autorizzati.

- ✔ **Accesso ai processi**. In genere, gli user-id effettivo e reale coincidono con lo user-id (spesso si riferiscono allo user-id che ha lanciato il processo), stesso discorso per lo group-id (che si riferisce al gruppo dell'utente che ha lanciato il processo). I file

hanno spesso due bit che corrispondo a **set-user-id** ed ad **set-group-id**. Se:

1. set-user-id=1: user-id reale=user-id; user-id effettivo=user-id;
2. set-group-id=1: group-id reale=group-id; group-id effettivo=group-id;

- ✔ **Accesso ai file**. Ad ogni file sono associati un **proprietario**, un **gruppo** (a cui appartiene in proprietario), dei **permessi** che determinano quali azioni possono compiere proprietario e membri del suo gruppo. Questi attributi sono assegnati dal sistema alla creazione del file, in seguito il proprietario li può modificare con ***chown*** (cambio di proprietario), ***chgrp*** (cambio di gruppo) e ***chmod*** (modifica permessi). I permessi si distinguono in r=readable, w=writeable, x=executable e vengono definiti per il proprietario, per il gruppo e per altri utenti.

Tipi di file. Esistono diversi tipi di file in Unix: **file regolari**, **directory**, **file speciali a caratteri** (file di dispositivi orientati al carattere), **file speciali a blocchi** (come prima ma orientati al blocco), **FIFO** o pipe (file utilizzato per la comunicazione tra processi), **socket** (stessa cosa di prima ma per la comunicazione in rete), **symbolic link** (file che punta ad un altro file). La struttura **stat** di ogni file contiene l'informazione riguardo la tipologia del file che si sta maneggiando, ed è possibile recuperare tale informazione utilizzando una macro.

CAPITOLO 4

Il Sistema Operativo: il File System, File Sharing, standard I/O

Virtual File System. Il **VFS** è un meccanismo implementato nel kernel che lo stesso impiega per accedere a tutti i file system supportati, mantenendo una interfaccia unica per tutti i programmi a livello utente. Quando un processo fa una chiamata di sistema su un file, il kernel opera una funzione sul VFS, manipolando alcune strutture generiche. Sarà poi il VFS a fare una ulteriore modifica sul file system reale a seconda della natura del suddetto file system. Le operazioni del VFS corrispondo a quelle che tutti i file system devono possedere, e riguardano la gestione dei file. Sono suddivise in operazioni sul **filesystem**, operazioni sui **file** ed operazioni sugli **i-node**.

File Sharing. Unix permette la condivisione dei file aperti tra più processi (**file sharing**). Ogni processo ha un **entry** nella tabella dei processi ed ad ogni entry corrisponde una tabella dei **file descriptor aperti** associati a quel processo. Ad ogni file descriptor corrispondono un **flag** ed un **puntatore** ad un elemento della tabella generale dei file aperti, tale tabella è gestita dal kernel e contiene per ogni elemento il **flag dello stato del file**, **l'offset** corrente del file e un **puntatore** ad un elemento della tabella dei **v-node** per il file. Ogni file aperto possiede una struttura v-node che ne contiene tutte le specifiche e le caratteristiche, compreso l'i-node.

- ✔ Ogni volta che una operazione di write è stata completata viene incrementato il **file offset** del numero di byte aggiunti, se supera il **file size** tale campo viene aggiornato.
- ✔ Se il file è aperto con O_APPEND, il flag corrispondente è messo a 1 e per ogni operazione di scrittura il **file offset** è importato al **file size** letto nel v-node.
- ✔ La funzione lseek modifica solo il **file offset** e non avvengono operazioni I/O.

Libreria standard I/O. E' descritta nello standard ANSI-C. L'operazione di apertura e creazione di un file associa a tale file uno **stream**, la funzione standard **fopen()** associa allo stream un puntatore ad un oggetto di tipo FILE (in genere si tratta di una struttura contenente le informazioni per gestire lo stream richieste dalla libreria standard). Gli stream disponibili sono lo **stdin**, lo **stdout** e lo **stderr**.

Buffering. Usare un buffer nella libreria standard I/O riduce la quantità di read() e write() da utilizzare. Esistono 3 tipi di buffering principali:

- ✔ **Full Buffered**. Le funzioni I/O avvengono solo quando il buffer è pieno. Sono full buffered tutti i file risiedenti su disco.

- ✔ **Line Buffered**. Le operazioni I/O sono consentite quando si incontra un carattere newline sia in imput che in output. Questo tipo di buffering si applica agli stream da e per terminale.
- ✔ **Unbuffered**. La librearia I/O non bufferizza i caratteri, tanto che quando li inseriamo desideriamo che siano disponibili per l'output il prima possibile. Lo stream stderr è unbuffered.

CAPITOLO 5

Il Sistema Operativo: I processi

I processi. Un processo consiste nell'esecuzione di un programma, la CPU riceve un flusso di byte che interpreta come istruzioni macchina da eseguire (M.J. Bach). Ogni processi in Unix possiede un proprio **Process ID** (**pid**) univoco maggiore di 0 che lo identifica. *Esistono processi speciali in Unix*: **swapper**, con pid=0 è un processo del kernel; **init**, con pid=1 è invocato dal kernel al termine del bootstrap e non termina mai, è un processo normale con super-privilegi; **pagedaemon**, con pid=0 è un processo del kernel che supporta il paging della memoria virtuale.
I possibili stati di un processo sono:

- ✔ **Esecuzione**. Processo in esecuzione in modo utente;
- ✔ **Esecuzione**. Processo in esecuzione in modo kernel;
- ✔ **Pronto in memoria**. processo non in esecuzione ma in attesa di essere schedulato (scelto) dal kernel per l'assegnamento di risorse e CPU;
- ✔ **Attesa in memoria**. Il processo attende e risiede in memoria;
- ✔ **Pronto in area swap**. Il processo è pronto ma lo swapper lo deve caricare in memoria prima di essere schedulato;
- ✔ **Attesa in area swap**. Processo in attesa ma lo swapper lo ha già caricato in memoria per far posto ad altri processi;
- ✔ **Prelazionato**. Processo ritornato dal modo kernel a quello utente;
- ✔ **Creato**. Processo creato;
- ✔ **Zombie**. Processo in exit ma anche in status zombie.

Un processo è composto da quattro settori fondamentali che costituiscono la sua immagine: nella **u-area** stanno le informazioni relative al processo di pertinenza del sistema operativo (in pratica, il contesto del processo); nell'area **dati** stanno i dati globali del processo; nello **stack** si trovano le variabili automatiche e le informazioni di ritorno dopo una chiamata a funzione; nel segmento di **testo** stanno le istruzioni macchina che la CPU deve eseguire.

La memoria nei processi. La memoria può essere intesa come "spazio" contente il kernel ed i processi oppure come vettore di pagine possibilmente associate a processi in esecuzione (le pagine sono fisicamente presenti sul chip della memoria!!).

La SHELL. La **SHELL** non è altro che un programma il cui compito è eseguire altri programmi. I sistemi Unix possiedono diversi tipi di shell a seconda dell'uso di destinazione, ma fondamentalmente tutte forniscono funzionalità di **esecuzione di programmi**, **gestione dell'input/output** e di **programmazione**. La shell segue i seguenti step: attende che l'utente le dia un comando, crea un nuovo processo per eseguire il comando, carica il programma da eseguire dal disco nel processo, esegue il programma nel suo processo fino alla fine.
La chiamata a sistema che invoca l'esecuzione di un nuovo programma è la **exec (pathname, argomenti)**, la quale sostituisce l'immagine del chiamante con il programma specificato nel suo pathname e lo esegue passandogli gli argomenti specificati. Ne esistono di 4 tipi a seconda della tipologia di struttura contenete gli argomenti vogliamo adoperare (vettore, lista, ecc..). Nello specifico, una shell esegue i seguenti passi per portare a termine il suo compito:

- ✔ **presentazione del prompt**;

- ✔ **Preleva la riga di comando**. La shell attende che l'utente abbia terminato di digitare il comando (che termina sempre con newline), poi lo legge e il comando viene restituito alla shell;
- ✔ **Analisi del comando**. Si analizza il comando da destra a sinistra, rendendosi conto che la prima parola a sinistra è il comando ed a seguire si trovano le opzioni;
- ✔ **Ricerca dei file**. La shell cerca il programma da lanciare;
- ✔ **Preparazione dei parametri**. La shell passa le opzioni del comando come parametri al programma da lanciare;
- ✔ **Esecuzione del comando.** La shell manda in esecuzione in programma eseguendo una serie di chiamate di sistema (fork(), exec(), wait()).

Creazione di un processo. In Unix TUTTI i processi (a parte init) sono creati da un processo "padre" mediante una chiamata a funzione **fork()**. La fork() duplica l'immagine del padre per creare un processo "figlio" identico al padre.
Il figlio eredita dal padre:

- ✔ tutti i file aperti ed eventuali flag close-on-exec impostati;
- ✔ tutti gli identificatori per il controllo dei processi;
- ✔ tutti gli identificatori per in controllo della sessione;
- ✔ directory di lavoro e directory radice;
- ✔ la maschera dei permessi;
- ✔ la maschera dei segnali bloccati e le azioni installate;
- ✔ i segmenti di memoria condivisa;
- ✔ i limiti sulle risorse;
- ✔ le variabili d'ambiente.

Non vengono ereditati:

- ✔ il valore di ritorno del fork;
- ✔ il pid;
- ✔ il parent process id (ppid);
- ✔ gli allarmi ed i segnali pendenti perchè per il figlio vengono azzerati.

Ambiente di un processo. Quando un processo "inizia" si entra nella sua funzione main(). Quando il kernel esegue il programma chiama una routine di inizializzazione prima della funzione main; è questa routine che si occupa di prelevare al kernel gli argomenti della linea di comando e le variabili d'ambiente da mettere a disposizione del main().
Per terminare un processo esistono 5 modi diversi suddivisi in 2 gruppi:

- ✔ terminazione normale (ritorno al **main()**; chiamata di **exit()**; chiamata di **_exit()**);
- ✔ terminazione anomala (chiamata di **abort()**; terminazione tramite **segnale**);

CAPITOLO 6

Comunicazione tra i processi: i segnali

Classi di primitive. Unix mette a disposizione dei processi diverse modalità di comunicazione, tra cui pipe, segnali, code di messaggi, semafori, memoria condivisa e sockets.

Segnali. I **segnali** sono utilizzati da kernel per gestire certi errori critici, per cui di solito si sceglie di interrompere l'esecuzione del programma. Unix offre un comando **kill** ed una chiamata **kill()** per lanciare un segnale, che può essere inteso come una forma molto limitata di comunicazione tra processi. I segnali, più in generale, servono a comunicare ad un programma l'occorrenza di qualche evento, poiché esistono eventi che possono generare un segnale, tra cui l'*errore in un programma*, la *terminazione di un processo figlio*, una *operazione illegale* e tanti altri. Il kernel interviene subito dopo l'occorrenza del evento e manda un segnale al processo interessato, di conseguenza il processo può eseguire la funzione di default oppure un'azione decisa dall'utente. Il comportamento di Unix è cambiato negli anni da un

comportamento inaffidabile (la routin scelta dall'utente non rimaneva attiva e dopo ogni segnale andava reimpostata) ad un **comportamento affidabile** (il gestore dei segnali una volta impostato rimane attivo) nella gestione dei segnali. Quando il comportamento è affidabile un segnale può essere **bloccato** e rimanere **pendente** finchè non viene **consegnato** od **ignorato**.

Tipi di segnali. Si possono classificare i segnali in 3 gruppi a seconda degli eventi che li possono scatenare:

- ✔ **Segnali generati da errori**. Durante l'esecuzione del programma si verifica un errore e il processo non è più in grado di continuare a lavorare (in maniera corretta). Gli errori che generano segnali sono la divisione per zero, uso di indirizzi non validi e le eccezioni hardware, in altri casi sono le condizioni software.
- ✔ **Segnali generati da eventi esterni**. Segnali scatenati dall'arrivo di in input, dalla terminazione di un figlio o dalla scadenza di un timer.
- ✔ **Segnali generati da richieste esplicite**. Sono segnali scatenati dall'intervento dell'utente come una combinazione da tastiera o l'utilizzo di una specifica chiamata di sistema.

Quando un processo riceve un segnale può scegliere di compiere tre azioni differenti:

- ✔ **Eseguire l'azione di default**. Si termina il processo.
- ✔ **Ignorare il segnale**. Se il programma che riceve un segnale è molto grosso è un problema ricevere un segnale inaspettato, così l'esecuzione continua.
- ✔ **Intraprendere un'azione specifica**. Solitamente, viene decisa dall'utente.

Chiamate "slow". Sono quelle chiamate di sistema che potrebbero bloccare un processo in maniera indefinita, ed includono operazioni di lettura da certi tipi di file, operazioni di scrittura, pause() e wait() e tanto altro.

Concorrenza di segnali. Cosa succede se un processo riceve due o più segnali in maniera concorrenziale? Lo stesso gestore dei segnali potrebbe essere chiamato ricorsivamente, oppure il secondo segnale arrivato viene ignorato, oppure il secondo segnale viene bloccato in attesa che venga gestito il primo.

Segali e funzioni rientranti. Quando si presenta un errore e si lancia un segnale, il processo si blocca e alla risoluzione del problema il processo riprende la sua esecuzione da dove "l'aveva lasciata". Delle volte però capita che il segnale si scateni in concomitanza con operazioni che interrompendosi possano compromettere l'integrità del processo. Bisogna stare molto attenti quando si ha a che fare con queste operazioni.

Stato dei segali.

- ✔ **Segnale generato (generated)**. Il segnale è da intendersi generato quando nel processo si verifica l'evento scatenante. Il kernel allora imposta un flag nella tabella del processo.
- ✔ **Segnale consegnato (delivered)**. Il segnale è consegnato quando l'azione conseguente alla ricezione del segnale è già stata intrapresa.
- ✔ **Segnale pendente (pending)**. È lo stato in cui si trova il segnale tra la generazione e la consegna.

Un segnale può essere bloccato e rimanere in pendenza. Se si generano più segnali uguali, ne verrà consegnato uno solo (se non si prevede una coda di segnali).

Programmazione del tempo. I programmi possono gestire il tempo in modi diversi. Si può ritardare il programma di un numero di secondi scelto impostando la chiamata a funzione sleep(). Con la chiamata alarm() imposto un timer allo scadere del quale lancio un segnale, ma nel mentre il programma può lavorare a qualcos'altro.

CAPITOLO 7

Interazioni con le periferiche

Programmazione dei dispositivi. Dopo i file e le directory un computer dispone di tanti altri "elementi" con i quali interagire per assumere ed emettere informazioni (stampanti, webcam, scanner, ecc...). I **dispositivi esterni** sono assunti da Unix come oggetti del tutto simili ai file (poiché anche i dispositivi possiedono TUTTE le qualità di un comune file). Ogni dispositivo possiede un **nome** suo specifico ed esclusivo, risiedente nella cartella home/dev. Inoltre, ogni dispositivo utilizza le stesse chiamate a sistema di un file, con l'eccezione che alcuni dispositivi non ammettono per natura certe funzioni (come ad esempio la write() per un mouse).

I-node dei dispositivi. Come i file anche i dispositivi possiedono il loro **i-node**. Mentre l'i-node di un file contiene il puntatore ai blocchi di dati costituenti il file, un dispositivo possiede un i-node che contiene i puntatori al driver del dispositivo risiedenti nel kernel.

Collegamento a un dispositivo. I collegamenti ai dispositivi sono diversi da quelli ai file. Il collegamento ad un file riguarda alcuni buffer nel kernel (BUFFERING), che vengono riempiti di dati quando un programma manda informazioni al file e poi svuotati scrivendo tutto su disco. Un collegamento ad un dispositivo invece non vuole ritardi nella consegna delle informazioni così un processo che manda dati ad un dispositivo non deve riempire nessun buffer (NO BUFFERING). Inoltre, un collegamento a dispositivo ha certe proprietà.

Software Tools e User Programs. I **Software Tools** sono quei programmi che non fanno nessuna distinzione tra file e dispositivi utilizzandoli alla stessa maniera (tra cui **who**, **ls**, **sort** e **uniq**). Gli **User Programs** invece sono programmi appositamente scritti per interagire con i dispositivi, come i programmi che controllano e gestiscono scanner e stampanti (ed ance **vi**, **pine**, **more** e altri).

CAPITOLO 8

About SHELL e PIPE

Tecniche di redirezione. Quando si opera in un ambiente di shell è possibile scegliere di redirigere l'output **su un file** anziché a schermo oppure fare in modo che **l'output di un processo diventi l'input di un secondo**. La qual cosa è possibile poiché la shell lavora sempre su **flussi di dati**, tra i quali il flusso di input (standard input) per prendere dati, il flusso di output (standard output) per emetterli ed il flusso di errore (standard error) per comunicare eventuali "intoppi". Ciascun flusso standard possiede il suo unico e specifico file descriptor (stdin=0, stdout=1, stderr=2).

Pipe. Una **Pipe** è un file di dimensione limitata gestito con una politica FIFO (il primo dato ad entrare è anche il primo ad uscire=coda): un processo produttore riempie il file ed attende se il file è pieno ed un processo consumatore legge il file ed attende se il file è vuoto. Esistono due tipi di pipe:

- ✔ **Pipe con nome.** Create da mknod (devono essere aperte con open());
- ✔ **Pipe senza nome.** Create ed aperte su un "pipe device" definito al momento della configurazione.

Dettagli sulla pipe. Generalmente una pipe serve ad agevolare la comunicazione tra processo padre e processo figlio. Ad un primo sguardo il file di pipe può essere inteso come un normalissimo file, eppure possiede caratteristiche specifiche:

- ✔ una read() da una pipe può bloccare il processo;
- ✔ se tutti i processi che scrivono chiudono la porta in scrittura della pipe ad una lettura

successiva read() ritornerà EOF;

- ✔ una pipe è un insieme di byte, se più processi la leggono contemporaneamente può avvenire asincronia e rendere incomplete le informazioni lette dai processi, a meno di una coordinazione tra loro. Inutile dire che quando un processo legge dalla pipe "cancella" i dati appena letti presenti nella pipe stessa;
- ✔ l'operazione write() può bloccare il processo se non c'è spazio dove scrivere e fallisce se non ci sono lettori.

Pipe con nome. Le **pipe con nome** possono essere utilizzate da processi che condividono lo stesso file system mettendo in comunicazione 2 o più processi contemporaneamente ed in maniera indipendente. La creazione di una pipe del genere avviene tramite la chiamata di sistema **mkfifo()**. Questo tipo di pipe può essere utilizzato da un sistema client/server o da una shell che vuole comunicare con una pipeline senza creare file temporanei. Le differenze tra **pipe** (pipe senza nome) e **FIFO** (pipe con nome) sono:

PIPE	FIFO
~ La pipe è utilizzabile solo da processi che hanno un "avo" in comune che abbia aperto per loro la pipeline.	~ Avendo un nome nel file system possono essere utilizzate anche da processi che non hanno nessuna gerarchia.
~ Pipe aperte con popen().	~ Sono create con mkfifo() e aperte con open().
~ Pipe chiuse con pclose().	~ Sono chiuse con close() ed eliminate con unlink().

Comunicazione client/server. I client inviano richieste al server tramite una FIFO comune mentre il server crea una FIFO specifica per ogni client.

CAPITOLO 9

Code di messaggi, semafori e memoria condivisa

System V IPC. Il sistema Unix-System V ha messo a disposizione 3 sistemi di intercomunicazione tra processi (InterProcess Comunication = IPC), i quali sono le **code di messaggi**, i **semafori** e la **memoria condivisa**. Ciascuna di queste strutture è identificata all'interno del kernel da un numero positivo elevato, la conoscenza del quale è fondamentale per l'utilizzo di tali strutture da parte dei processi. La lista degli identificatori è circolare, quando si crea e si distrugge una struttura il suo identificativo non viene riutilizzato ma si attende il raggiungimento della capacità massima del circolo per poi azzerarlo.

Identificativi e chiavi. Quando viene creata una struttura IPC è necessario specificare una **chiave**. Tale chiave è del tipo key_t ed è definito solitamente come un long int, sarà poi il kernel a tradurre la chiave nel suo relativo identificativo. Esistono diversi modi per un client ed un server per utilizzare la stessa struttura IPC:

- ✔ **Chiave IPC_PRIVATE.** Il server crea una struttura specificando una chiave IPC_PRIVATE e depositare l'identificativo in una struttura risiedente nel file system che il client possa utilizzare. Utile quando i processi in gioco possiedono una gerarchia condivisa.
- ✔ **Chiave condivisa.** Client e server si accordano su una chiave specifica condivisa in un header file. Il server crea una struttura con questa chiave ma sorgono dei problemi se la chiave è già stata utilizzata.
- ✔ **Utilizzo di ftok()**. La funzione ftok() genera una chiave a partire da un pathanme (di cui servono i 16 bit meno significativi dell'i-number associato al pathname) ed un numero tra 0 e 255 (del quale servono gli 8 bit meno significativi). Il comportamento del server è poi come al punto 2.

A questo punto si può creare la struttura desiderata con una chiamata a sistema del tipo **smgget()** e altre (vedi appendice B). Alla creazione viene associata ad ogni struttura IPC

anche una seconda struttura contenente i permessi.

Code di messaggi. Dette anche **Message Queue**, fanno in modo che due o più processi possano scambiarsi informazioni attraverso una comune coda di messaggi: il processo mittente utilizza un modulo di scambio di messaggi per mettere il suo messaggio su una coda in modo tale che il processo destinatario lo possa leggere. A ciascun messaggio è associato un **identificativo** o **tipo** in modo che i processi possano scegliere il messaggio appropriato. Innanzi tutto, i due processi devono condividere la chiave della coda messaggi per rendere possibile lo scambio delle informazioni.

Memoria condivisa. Detta anche **Shared Memory**, è una delle più efficienti tecniche di IPC. Consente a due processi di condividere lo stesso segmento di memoria fisica. Per fare ciò dobbiamo riservare il segmento di memoria con l'apposita chiamata **shmget()**, collegarlo ai due processi con **shmat()** e alla fine del suo utilizzo scollegare il segmento dai processi con **shmdt()**. Una memoria condivisa è rappresentata da una struttura riassuntiva delle sue qualità nella quale troviamo la sua dimensione, il numero di processi che la stanno utilizzando, come la memoria condivisa è collegato allo spazio di indirizzamento dei processi. Quando un processo vuole accedere alla memoria condivisa il sistema crea per lui una struttura dati che tiene conto delle qualità della memoria condivisa per quel processo. Se il collegamento avviene per la prima volta si creano delle pagine di memoria condivisa. Gli altri processi che vorranno collegarsi semplicemente rendono disponibile il loro spazio degli indirizzi che il sistema collegherà poi alla pagina della memoria condivisa.

I Semafori. I **semafori** non sono veramente delle IPC, ma servono come controllori del "traffico" in entrata ed uscite dalle strutture IPC, regolando gli accessi ad esse da parte dei processi. Infatti, per accedere ad una struttura dati condivisa, si deve:

- ✔ Controllare il semaforo che controlla la risorsa in questione;
- ✔ Se il valore è positivo il processo può utilizzare la risorsa decrementando di 1 il valore del semaforo per notificare che sta impiegando una istanza di quella risorsa;
- ✔ Se il valore del semaforo è 0 il processo attende che il valore diventi positivo mettendosi in coda, riprendendo dal punto 1.
- ✔ Quando il processo ha finito di utilizzare la risorsa incrementa di 1 il semaforo ed i processi in attesa vengono risvegliati.

E' importante che i semafori compiano le azioni di incremento e decremento in maniera atomica, per evitare incongruenze.
I **semafori IPC** definiti dal System V sono però più complessi di una banale variabile che si decrementa ed aumento all'occorrenza, tanto che sono formati da più valori messi assieme da specificare all'atto di creazione del semaforo. Anche creare un semaforo è indipendente dalla sua inizializzazione e queste forme di semaforo possono rimanere presenti anche se nessun processo ne fa uso.

CAPITOLO 10

Multithread

Il processo. Informalmente un processo è un programma in esecuzione. Il **processo** è un ambiente attivo di runtime che racchiude un programma in esecuzione, fornendogli uno stato di esecuzione assieme a certe altre risorse.

Multiprocesso. In Unix è possibile che più processi vengano mandati in esecuzione allo stesso istante, tale sistema è definito **multi-processo** o **multitasking**. L'esecuzione simultanea può essere **concorrente** (i processi che si trovano nello stato running possono essere mutuamente swapped in e swappoed out dal sistema operativo) o **parallela** (i processi sono in esecuzione allo stesso istante per la presenza di processori multipli).

Il thread. Un **thread** è l'unità base di utilizzo della CPU. A ciascun processo è affidato di default un singolo thread. Il thread è spesso chiamato **processo leggero** poiché è molto simile ad un processo possedendo un suo personale thread-ID, programm counter, stack, insieme di registri ed altro. Tutti i thread all'interno dello stesso processo condividono il segmento dei dati e del codice e le risorse fornite dal sistema operativo.
Ogni sistema operativo ha la propria libreria ed il proprio sistema di gestione dei thread. Ciò rende difficile scrivere programmi multi-thread perchè spesso le funzioni per gestire i thread non sono portabili da un sistema all'altro. POSIX ha fornito uno standard chiamato **Pthreads**.

Processo	Thread
~ *Process ID*	~ *Thread ID*
~ Programm Counter (PC)	~ Programm Counter (PC)
~ Tabella dei segnali	~ Tabella dei segnali
~ Registri	~ Registri
~ *Indice di priorità del processo*	~ *Indice della priorità del thread*
~ Stack e puntatore allo stack	~ Puntatore allo stack e stak
~ Heap	**Tutti i thread condividono la stessa memoria, lo heap e la tabella dei file descriptor.**
~ Mappa della memoria	
~ Tabella dei File Descriptor	

I thread possono essere creati e gestiti più velocemente dei processi per via del loro essere "più leggeri" dei processi stessi; siccome i thread condividono lo stesso heap e segmento di codice e dati hanno minore overhead; i thread possono vivere interamente nello spazio utente senza passare in quello kernel per la creazione di nuovi thread; i processi non devono subire swapped per creare thread.
Come un sistema operativo può avere più processi in esecuzione parallela, un processo può avere più thread in esecuzione parallela, gestiti da uno scheduler interno al processo oppure da più processori contemporaneamente.

Benefici del multithreading. Sono descritti nella legge di Amdahl:

$$speedup = 1/((1-p)+(p/n))$$

Ovvero, lo speed ottenuto dall'esecuzione parallela del codice è il tempo necessario per eseguire il lavoro parallelizzabile (p) diviso per il numero dei processori (n)+1 meno il lavoro parallelizzabile (1-p). Più codice può essere eseguito in parallelo più veloce sarà l'esecuzione dell'intero programma. **La legge tuttavia non è lineare**. Ottengo un aumento della produttività, della rapidità di risposta di una applicazione, semplifico i metodi di comunicazione tra i processi, si guadagna anche su sistemi a singolo processore.

Tipi di thread. L'implementazione del thread può avvenire sia a **livello utente** (è presente una librearia utente ed il kernel gestisce solo i processi ignorando l'esistenza dei thread) che a **livello kernel** (supportato a livello del sistema operativo).

- ✔ **Many to one.** I thread sono in esecuzione nello spazio utente che permette context switch veloci, il numero di thread permessi è illimitato, il parallellismo non è possibile, se si blocca un thread si blocca tutto il processo.
- ✔ **One to one.** E' supportata l'esecuzione parallela, context switch più lenti, il numero dei thread possibili è limitato.
- ✔ **Many to many.** Molto flessibile, supporta pienamente l'esecuzione parallela, implementato sia nello spazio utente che kernel, context switch lenti, il numero di thread utenti è illimitato.

CAPITOLO 11

I thread ed i meccanismi di sincronizzazione

Sincronia tra thread. Quando coesistono più thread contemporaneamente sorge il problema della comunicazione tra loro.

- ✔ **Mutua esclusione.** Riassunto nella parola **mutex**, la mutua esclusione è un tipo di dato che permette a più thread di sincronizzare il loro accesso a risorse condivise. E' some un **semaforo binario a due stati** (locked e unlocked). Una volta che il mutex è bloccato, gli altri thread che vorranno bloccare lo stesso semaforo verranno sospesi, ciò è vero finchè il thread bloccante non sblocca il mutex: quando succede uno degli altri thread può bloccare nuovamente il mutex e ricominciare il ciclo.
- ✔ **Variabile condizionale.** Si tratta di un meccanismo di sincronizzazione che fa in modo di far attendere i thread fino alla verifica di una data condizione che stavano aspettando. Le variabili condizionali non proteggono codice, ma procedure. La variabile viene manipolata da altri thread che quando "terminano il loro lavoro" segnalano ad altri thread di poter prendere il loro posto. Ogni variabile condizionale è associata ad un **mutex**.
- ✔ **Blocchi lettore-scrittore.** Anche se i mutex risolvono molti problemi, un largo impiego di questi semafori potrebbe rendere serializzata l'applicazione che ne fa uso. Una sezione critica necessita di protezione solo se più thread vogliono modificarne i dati; più letture o scritture contemporanee sono ammesse ma i mutex bloccano sena riguardo sia l'una che l'altra operazione. I **blocchi lettore-scrittore** permettono più accessi in lettura ma uno solo in scrittura nella sezione critica.

Barriere. Può capitare che si voglia far in modo di bloccare ad un certo punto della sezione critica l'azione di un thread facente parte di un gruppo di suoi simili fino a quando tutti gli altri thread appartenenti allo stesso gruppo saranno arrivati allo stesso punto, definito **barriera**. Una barriera è creata impostando il suo valore al numero di thread presenti nel gruppo: può essere implementato come un contatore che ogni qualvolta un thread raggiunge la barriera si decrementa fino ad arrivare a 0 quando giunge l'ultimo thread (ed ogni thread che lo raggiunge si sospende in attesa degli altri). L'ultimo thread che arriva sveglia gli altri lanciando loro un segnale che li riattiva. Una barriera quindi è costituita da un mutex ed una variabile condizionale.

Problemi di sincronizzazione. Ne esistono principalmente tre:

- ✔ **Deadlocks** (abbraccio mortale). Si presenta quando i thread sono bloccati in modo errato, tanto che il primo che ha bloccato il mutex non riesce a liberarlo per il secondo che allo stesso modo ha bloccato il primo. Nessun thread può continuare in modo corretto tanto da consenitre agli altri di proseguire. Per evitare tale situazione si devono usare meno mutex possibili.
- ✔ **Race condition.** Si verifica quando l'operazione di assegnamento di una variabile è indeterminata a causa di un possibile context switch o parallelizzazione.
- ✔ **Inversione delle priorità.** Può accadere che un thread a bassa priorità blocchi un mutex e poi venga prelazionato da uno ad alta priorità che però non può proseguire perchè il mutex è mantenuto dal precedente thread a bassa priorità.

Thread e segnali. Ogni thread possiede la sua maschera dei segnali ma la gestione dei segnali è condivisa da tutti i thread del processo. I segnali sono mandati la maggior parte dei casi ad un thread specifico ma se non è necessario viene scelto un thread arbitrario: se un thread blocca la ricezione di un segnale un altro thread può modificare tale blocco. Una cosa furba è mascherare tutti i thread ai segnali tranne uno che li gestisce e li smista.

Semafori POSIX. Sono **primitive** del sistema operativo che permettono la sincronizzazione tra processi e thread. Sono disponibili i semafori System V gestiti dal kernel

(come visto in precedenza), i semafori POSIX con nome identificati tramite un nome POSIX IPC ed i semafori POSIX allocati nella memoria condivisa. Le operazioni svolte su un semaforo sono solitamente tre: **create()** che crea il semaforo con un valore iniziale specificato, **wait()** che fa attendere un thread o un processo al semaforo se questo ha un valore minore o uguale a 0, **post()** che incrementa il valore del semaforo. Siccome un semaforo può essere usato come un mutex, la differenza tra l'uno e l'altro è che un mutex deve essere sempre sbloccato dal thread che l'ha bloccato, ma l'operazione post() su un semaforo non deve essere fatta dal thread che l'ha decrementato.

CAPITOLO 12

RPC ~ Remote Procedure Call

Progettazione di un programma distribuito. Ne esistono di due tipologie distinte:

- ✔ **Progettazione orientata alla comunicazione.** Per prima cosa si progetta il protocollo, poi si passa alla progettazione dei programmi che aderiscono a questo protocollo (stile socket).
- ✔ **Progettazione orientata alla applicazione.** Per prima cosa avviene la realizzazione dell'applicazione, dopo di chè si suddividono i programmi e si aggiunge il protocollo di comunicazione (stile RPC).

Remote Procedure Call. Avviene una chiamata ad una procedura che è già attiva su un'altra macchina (detta anche **subroutine**). I punti essenziali di tale procedimento sono l'identificazione e l'accesso alla procedura remota, il passaggio dei parametri alla procedura e il ritorno del valore di ritorno.
Come funziona:

- ✔ Il client chiama una procedura locale chiamata **stub del cliente**, il cui scopo è quello di preparare i parametri da passare alla procedura remota e dei messaggi di rete (*marshaling*) "simulandola" agli occhi del client che nonostante chiami una sua stessa procedura crede di aver già a che fare con una procedura del server.
- ✔ Lo stub del cliente invia i messaggi di rete preparati al sistema remoto tramite una chiamata di sistema.
- ✔ Tali messaggi sono trasferiti al sistema remoto attraverso un protocollo orientato alla trasmissione.
- ✔ Una procedura **stub del server** sta attendendo l'arrivo dei messaggi dal client da sistema remoto. Essa interpreta le informazioni contenute nei messaggi traducendole nel formato corretto.
- ✔ Lo stub del server chiama la procedura selezionata passandole gli argomenti del client.
- ✔ Una volta terminata la procedura essa torna un valore che viene restituito allo stub del server.
- ✔ Lo stub del server converte il valore e lo inserisce in un messaggio.
- ✔ Il messaggio viene rimandato al client tramite rete.
- ✔ Lo stub del client legge il messaggio attraverso il kernel locale.
- ✔ Lo stub del client restituisce il risultato al processo chiamante.

E' opportuno rendere il codice degli stub completamente indipendente dalla effettiva realizzazione della rete attraverso la quale avviene la comunicazione, che sia essa fatta di socket o di qualsiasi altra cosa.

Sun RPC. Esistono un certo numero di implementazioni RPC e Sun ne offre una, la Sun RPC, largamente utilizzata. Possiede un ricco insieme di strumenti di supporto (NFS, Network File System si basa su Sun RPC).
Per ridurre la complessità delle specifiche dell'interfaccia, Sun RPC supporta un singolo argomento per la procedura remota (negli ultimi tempi anche a più argomenti). Il singolo argomento contiene una struttura formata da un limitato numero di dati e valori.

Ciascuna procedura è identificata tramite:

- ✔ hostname (IP Address);
- ✔ Identificatore di programma (32 bit), ciascun programma remoto ne possiede univocamente uno;
- ✔ Identificatore di procedura (32 bit), numerati a partire da 1 e poi sequenzialmente;
- ✔ Identificatore della versione del programma (iniziano da 1 e a seguire).

Server iterativo. In un qualsiasi istante si può invocare al più una sola procedura remota, se viene chiamata una seconda procedura questa rimane congelata fino al termine della prima. Se necessario è possibile realizzare un sistema concorrente ed avere un server iterativo è utile quando si ha a che fare con programmi che condividono i dati (come i server) giusto per non creare collisioni tra le procedure.

Semantica delle chiamate remote. Chiamare una procedura locale implica che tale procedura verrà eseguita esattamente una sola volta, non vale lo stesso discorso per le procedure remote, che possono essere chiamate ma non è certo che vengano eseguite una sola volta. Per far sì che il comportamento delle procedure remote sia identico a quello delle procedure locali è necessario utilizzare un trasporto affidabile (TPC). Sun non supporta le chiamate affidabili. Il comportamento conforme indica che se otteniamo una risposta dalla routine remota essa è stata eseguita al più una volta, se invece non attendiamo la sua risposta o non è stata eseguita o lo è stata troppe volte.

Comunicazione di rete. Il protocollo utilizzato dalle RPC è il TPC/IP, infatti molte interazioni tra RPC sono state progettate sulle librerie delle socket. In questo modo l'API utilizzata è diversa ma la base è la stessa. I server non utilizzano porte di protocollo, i client conoscono l'ID del programma ed il suo indirizzo IP e RPC prevede la ricerca del numero identificativo della porta di un programma remoto.

Servizio di ricerca della porta. Un servizio di ricerca della porta è in funzione su ogni host che contiene uno o più server RPC, che si registrano con un servizio del tipo "sono il programma 1 ed attendo sulla posta 111". Ogni sistema che supporta i server TCP fornisce un servizio di "port mapper" che fornisce una registrazione centralizzata per i servizi TCP. I server comunicano al port mapper che servizi offrono, ed i client che cercano quel dato servizio si informano dal port mapper su quale porta fa al caso loro (dove il **port mapper** è esso stesso un server TCP).

RPCGEN. Esiste uno strumento che consente la creazione automatica di server e client RPC, questo strumento è RPCGEN che compie la maggior parte del lavoro al posto nostro. Come input è necessario fornire al programma la definizione di un protocollo nella forma di una lista di procedure remote e di parametri.

CAPITOLO 13

Modelli di programmazione e protocolli di rete

Modelli di programmazione. Ne esistono di diversi tipi:

- ✔ **Modello Client/Server.** È il modello più diffuso per la programmazione di rete, prevede un programma server che riceve le richieste di fornitura di un servizio da un lato client che richiede il servizio. I server possono essere iterativi o concorrenti.
- ✔ **Modello Peer-to-Peer.** Non esiste una divisione dei compiti come nel caso precedente, ogni programma funge da client e server mandando e ricevendo richieste indistintamente.
- ✔ **Modello Tree-Tier.** È una estensione del modello client/server ma su tre livelli: il primo livello è costituito dal client che invia le richieste, il secondo da un server

intermedio che analizza le richieste ed il terzo dal server vero e proprio che soddisfa le richieste del client.

Modello di riferimento OSI. Tale modello di riferimento (OSI = **Open System Interconnect**) ammette 7 livelli che definiscono le funzioni dei protocolli di trasmissione dati. Ogni livello del modelli OSI costituisce una funzione che viene seguita quando i dati sono scambiati tra applicazioni cooperanti attraverso il supporto di una rete di comunicazione. Può succedere che un livello contenga più di un protocollo che funge da integrazione alla funzione che deve attivarsi su quel dato livello. Ogni protocollo inoltre comunica con un suo pari (peer), ovvero un altro protocollo dello stesso livello ma sito su un sistema remoto, così ogni protocollo si preoccupa al più del suo livello sovrastante o sottostante.
Il sistema di scambio dei dati tra macchina locale e remota viene perpetrata tra i livelli di ciascuna macchina e funziona in modo tale da consentire ad un protocollo di mandare dati ai protocolli sottostanti fino a raggiungere il livello fisico che si occupa di mandare i dati alla macchina remota che li attende, dove le informazioni invece di scendere salgono nella gerarchia fino a raggiungere il livello destinatario. Ai protocolli di livello non interessa come è fatto il livello sottostante o sovrastante, l'importante è che sappiano come passare i loro dati. Questo supporta i cambiamenti tecnologici e l'installazione di nuovi server che non vanno ad intaccare la struttura portante del sistema dei protocolli.
Ecco come sono strutturati i 7 livelli del modello OSI:

- ✔ **Livello applicativo.** Qui risiedono i processi utente che accedono alla rete. Comprende sia i processi direttamente utilizzati dall'utente che quelli dei quali l'utente non si cura.
- ✔ **Livello presentazionale.** Le applicazioni che collaborano per lo scmabio dei dati si accordano su come i dati devono essere rappresentati. Il livello fornisce la routine di presentazione standard dei dati.
- ✔ **Livello di sessione.** Gestisce le sessioni (connessioni) tra le applicazioni cooperanti.
- ✔ **Livello di trasporto.** Tale livello si assicura che il ricevente riceva i dati esattamente come gli sono stati inviati.
- ✔ **Livello di rete.** Gestisce i collegamenti attraverso la rete ed isola i livelli sovrastanti dai dettagli implementativi della rete.
- ✔ **Livello del data link.** Questo livello gestisce la spedizione affidabile dei dati nella rete fisica gestita dal livello sottostante.
- ✔ **Livello fisico.** Si definiscono qui gli aspetti implementativi dell'hardware sottostante facente funzione di rete per il trasporto dei dati.

Protocollo TCP/IP. Questo protocollo viene rappresentato da un modello a quattro livelli (livello applicazione, livello trasporto, livello internet, livello di accesso alla rete). Funziona come il modello OSI nella trasmissione dei file, in aggiunta ogni livello allega ai dati una informazione di controllo specifica (detta **header** perchè è posizionata in testa al flusso dei ati) che ne assicura la corretta spedizione.

- ✔ **Livello di accesso alla rete.** Il protocollo di questo livello fornisce i meccanismi adeguati per inviare i dati agli accessi direttamente presenti sulla rete. Questo livello conosce in dettaglio come è fatta la rete e tutti i suoi meccanismi.
- ✔ **Livello internet.** Fornisce il protocollo base di spedizione dei dati sul quale si fonda tutto il modello applicativo. Include una definizione del **datagram**.

Il datagram. È l'unità base della comunicazione internet, è un insieme di dati che contiene tutte le informazioni necessarie alla sua spedizione. Una rete che manda e riceve pacchetti utilizza queste informazioni per instradare il pacchetto dal mittente al destinatario. Il protocollo guarda l'indirizzo di spedizione del pacchetto e se si trova nella rete locale lo manda direttamente al destinatario, sennò lo indirizza ad un **gateway**, un dispositivo che selezione la rete giusta sulla quale inviare il pacchetto (tale esericio è detto **routing**).

Gateway e router. Sono fondamentalmente due cose differenti, tradizionalmente i dispositivi che mandavano e ricevevano pacchetti sulla rete erano detti **gateway** (chi li mandava) e **host** (chi li riceveva). Però se un host era connesso a più reti fondamentalmente poteva mandare pacchetti e faceva la funzione del gateway. Attualmente un **gateway** muove dati tra protocolli differenti mentre un **router** lo fa tra reti differenti.

Socket. Una **socket** è un punto terminale della comunicazioni tra processi, individuata dal protocollo utilizzato, l'indirizzo IP della macchina e il numero della porta. Una coppia di socket identifica univocamente una rete.

CAPITOLO 14

Sistemi distribuiti e sistemi di reti

Sistema distribuito. Si tratta di un insieme di processori che non condividono memoria o clock possedendo ognuno una sua memoria personale. Ogni processore interagisce con gli altri attraverso linee di comunicazione. Un sistema distribuito rende disponibile agli utenti un certo numero di risorse (ES: file system distribuito) e naturalmente deve gestire tutte le interazioni e le comunicazioni tra processi in modo da prevenire situazioni di stallo.
Un sistema distribuito offre:

- ✔ **Condivisione delle risorse.** Condivide i file, l'hardware remoto specializzato, database distribuiti, funzionalità di stampa remota, ecc.
- ✔ **Accelerazione del calcolo.** L'elaborazione ed il calcolo possono essere distribuiti tra i vari processori agevolando così la routine dei processi.
- ✔ **Affidabilità.** Gestione delle situazioni di malfunzionamento di un processore.
- ✔ **Comunicazione.** Scambio di messaggi tra gli utenti, trasferimento di file, mail, ecc.

Tipologie di sistema distribuito. Esistono diverse tipologie di sistema distribuito:

- ✔ **Sistema operativo di rete (NOS).** Permette il login remoto ed il trasferimento di file remoto.
- ✔ **Sistema operativo distribuito.** L'utente accede alle risorse remote come se fossero risorse locali, spostare dati e processi è compito del sistema operativo. Per quanto riguarda la **migrazione dei dati** esistono due soluzioni: **Andrew File System**, che trasferisce il file del sito A al sito B e se modificato lo restituisce al sito A; **Sun Network File System**, che trasferisce solo la porzione di file necessaria ed al termine della modifica si aggiorna tutto il file interessato. Per la **migrazione del calcolo** il processo nel sito A può chiedere di utilizzare una routin remota alla cui conclusione viene restituito al sito A il suo risultato, e lo stesso processo può inviare un messaggio al sito B dove il sistema operativo crea un processo che compie la routine specificata dal primo processo, e quando termina il processo in sito B manda un messaggio al processo in sito A tramite RPC (protocollo UDP).

Quando si tratta di far migrare un processo di calcolo, può succedere che parte del processo o tutto il processo stesso venga trasferito per l'esecuzione in un altro sito diverso da quello di creazione. Il WEB è un chiaro esempio di sistema distribuito con trasferimento dei processi (si pensi alle applicazioni JAVA).

Tipologie di reti. Possiamo distinguere tra **LAN** (Local Area Network) e **WAN** (Wide Area Network).

- ✔ **Local Area Network.** Di solito copre piccole aree geografiche (< 1km) ed è una rete multiaccesso a cavo, a forma di rete o stella, consente una comunicazione veloce ed economica. Come nodi ha una serie di personal computer o server.
- ✔ **Wide Area Network.** Collega siti molto distanti tra di loro e presenta una connessione point-to-point su linee dedicate ed affittate dalle compagnie telefoniche. La comunicazione richiede l'invio multiplo di pacchetti e messaggi ed i nodi sono costituiti

da server e mainframe.

Comunicazione sulla rete. I problemi e le strategie sono molteplici.

- ✔ **Denominazione e risoluzione dei nomi.** La rete identifica i processi su un host remoto per mezzo di una configurazione del tipo <host-name, identifier>. Il **Domain Name Service (DNS)** specifica sia la struttura dei nomi degli host che la risoluzione degli indirizzi, associa un indirizzo IP ad un nome. I modi per risolvere i nomi sono molteplici, di solito i domini hanno un server dedicato a questo compito.
- ✔ **Strategie di instradamento.**
  - ✗ **Fixed Routing.** La strada tra A e B è specificata a priori e non cambia mai salvo eventi straordinari. E' possibile scegliere il cammino più corto per minimizzare i costi ma la rete non supporta sovraccarichi di lavoro ed i messaggi sono ricevuti nello stesso ordine in cui sono mandati.
  - ✗ **Virtual Routing.** La strada tra A e B esiste finchè esiste la sessione. Sessioni diverse possono instaurare cammini diversi. I messaggi sono ricevuti nell'ordine di emissione e la rete si adatta parzialmente a episodi di sovraccarico.
  - ✗ **Dynamic Routing.** Solo quando si spedisce il messaggio la strada tra A e B è specificata. Si adatta parzialmente alle situazioni di sovraccarico e utilizza le strade meno utilizzate. I messaggi possono essere ricevuti in sequenza diversa rispetto all'emissione.
- ✔ **Strategie di connessione.**
  - ✗ **Circuit switching.** E' un collegamento esclusivo permanente per la durata della connessione (ES. collegamento telefonico).
  - ✗ **Message switching.** E' un collegamento temporaneo della durata dell'invio di un messaggio (ES. collegamento postale).
  - ✗ **Packet switching.** Un messaggio logico viene suddiviso in pacchetti e ciascun pacchetto viene inviato separatamente.

  Il circuit switching richiede un certo tempo per l'impostazione ma ne risparmia durante la comunicazione, gli altri due richiedono meno tempo di configurazione ma più tempo per lo scambio dei messaggi.
- ✔ **Contesa.** Se più siti decidono di trasmettere nello stesso istante si crea una condizione di collisione, le tecniche per evitare una situazione simile sono:
  - ✗ **CSMA/CD.** Un sito ascolta la rete per vedere se qualcuno trasmette. Se la rete è libera la trasmissione inizia. Se malauguratamente due siti fanno la stessa cosa contemporaneamente avviene una collisione, la trasmissione si interrompe subito e riprende dopo un intervallo di tempo casuale. Quando la rete è molto utilizzata le sue prestazioni sono degradate a causa della grande frequenza di collisioni.
  - ✗ **Token Passing.** Un tipo di dato detto token circola continuamente sulla rete e se un sito vuole trasmettere lo deve attendere e prelevare. Allora può trasmettere e quando termina rilascia il token.
  - ✗ **Message slot.** Un numero di contenitori di grandezza prefissata circolano sulla rete, un host che vuole trasmettere riduce in pacchetti il suo messaggio ed affida ogni pacchetto al primo slot libero che capita.

Indirizzi di Internet. Un indirizzo internet occupa 32 bit e identifica sia reti che computer (codifica sia ID di rete che ID dell'host). L'ID dell'host è relativo a quello di rete. Gli indirizzi di internet sono affidati da un ente ed esistono 5 tipi diversi di indirizzi IP.

Maschera di sottorete. Tale meccanismo identifica un gruppo di indirizzi contigui che una interfaccia può agevolmente raggiungere. Costituisce un filtro che consente solo a certi messaggi di passare oltre, infatti quando un messaggio arriva il router utilizza la maschera sull'indirizzo di destinazione, se il risultato corrisponde allora lascia passare il messaggio.

Domain Name System. E' difficile per gli utenti ricordare gli indirizzi IP è sono impossibili da indovinare. Però gli indirizzi IP sono eccezionali per identificare un computer. Così il **DNS** traduce un hostname in un indirizzo IP. I nomi di dominio possono essere semplici o complicati, comunque rispettano una rigida gerarchia. Ogni **hostname** è costituito da una serie di label separati da punti. Un **dominio** è un sottoalbero dell'albero gerarchico mondiale dei nomi, ogni dominio possiede un **nome di dominio** che è costituito dai label che conducono all'host, dalla foglia alla radice dell'albero gerarchico.

CAPITOLO 15

Gestione dei segnali nei modelli TCP

I processi zombie. I server TCP delegano ad un processo figlio l'incombenza di gestire il lato client. Dopo aver soddisfatto il cliente ed aver subito la chiamata di sistema exit(), il processo figlio entra in modalità **zombie** in attesa che il padre prelevi da esso le informazioni sul suo stato di terminazione. Lasciare attivi dei processi zombie è controproducente perchè occupano spazio nel kernel e non rilasciano le risorse che potrebbero invece utilizzare altri processi in attesa. Purtroppo poiché il server deve assecondare diverse richieste concorrenti non può lanciare una chiamata wait() su ogni figlio prima di creare uno nuovo per gestire un nuovo client. Per semplificare le cose si fa in modo che il processo figlio alla sua terminazione lanci un segnale specifico al padre (SIGCHLD).

Il segnale SIGPIPE. Alle volte nei sistemi client/server concorrenti può capitare che il server si interrompa improvvisamente senza che il client lo venga a sapere, così il cliente proverà a mandare dati al server ignorando che la comunicazione tra i due è interrotta. Il segnale **SIGPIPE** fa un modo che il client si interrompa e non comunichi più con il server. Gestire il segnale previene l'interruzione improvvisa del client.

Web Server. Un **web server** è un programma che esegue programmi a sua volta, lista le directory e compie operazioni di cat sui file. Per fare ciò utilizza una socket stream e le operazioni che compie di solito si riassumono così: il client si connette ed il server si mette in ascolto, allora client manda una richiesta al server, il server l'analizza e la esegue, poi manda indietro la risposta, il client la riceve e la mostra a schermo, poi ripete la sequenza delle operazioni appena descritte.

HTTP. Acronimo di **HyperText Transfer Protocol**, si tratta di un protocollo che gestisce la comunicazione tra browser web e server web (in un modello client/server), dove un server web è inteso come un **server HTTP**. HTTP permette che il client invii una richiesta al server e che il server ne restituisca la risposta. HTTP può gestire transizioni multiple utilizzando una sola connessione TCP (la porta tradizionale impiegata è la 80). Una richiesta conforme al protocollo HTTP è formata da linee di testo in ASCII la prima delle quali è detta Request-Line. La linea di richiesta contatta il server attraverso la porta specificata e con un comando HTTP detto **metodo**, comunicando il nome del documento richiesto (il suo indirizzo) e la versione del protocollo utilizzato. La linea di richiesta contiene 3 token separati da spazi. I **metodi** della richiesta possono essere:

- ✔ **GET.** Riceve le informazioni identificate attraverso un URI.
- ✔ **HEAD.** Riceve le meta-informazioni riguardo l'URI.
- ✔ **POST.** Invia informazioni ad un URI e riceve i risultati.
- ✔ **PUT.** Conserva le informazioni un una locazione specificata da un URI.
- ✔ **DELETE.** Rimuove l'entità specificata dall'URI.
- ✔ **TRACE.** È inpiegato per eseguire il forwarding HTTP attraverso proxies, tunnels, ecc.
- ✔ **OPTIONS.** Impiegato per determinare la capacità del server, oppure le caratteristiche di una risorsa.

In seguito alla linea di richiesta seguono un numero anche pari a zero di **intestazioni HTTP**.

Ogni intestazione contiene un nome di un attributo seguito da ":" seguito a sua volta dal valore dell'attributo. Le **intestazioni di richiesta** forniscono al server delle informazioni riguardo al client come di quale tipo di client si tratta, quali contenuti saranno accettati e chi sta facendo la richiesta. Può anche capitare però che non siano fornite intestazioni. Ogni intestazione finisce con una linea vuota.
La risposta che arriva al client dal server HTTP è una serie di linee di caratteri ASCII contenenti una **linea di stato**, una **sezione di intestazioni** e un **contenuto** che può essere qualsiasi cosa ma che in generale è un file HTML. La **linea di stato** è formata dall'informazione della versione HTTP, da uno status code di 3 cifre (per la macchina) e da un messaggio in caratteri ASCII (per gli utenti). Le **intestazioni** forniscono al client informazioni riguardo l'entità del documento che sta per essere restituito come la sua tipologia, la sua dimensione, la sua codifica e la data dell'ultima modifica. Il **contenuto** può essere qualsiasi cosa e le sue prime righe riportano la sua tipologia e la sua lunghezza.

Impiego tipico dei metodi. Il metodo **GET** è impiegato per ottenere un documento HTML, **HEAD** è utilizzato per sapere se il documento è stato modificato e **POST** è impiegato per sottomettere un form.

Impiego di un URI. Nell'utilizzo di un server HTTP 1.1 è utilizzato solo un path. I server HTTP 1.1 dovrebbero essere in grado di gestire un URI, ma tanti non lo fanno. Quando utilizziamo un **server proxy** si impiega un URI assoluto. Il client deve dire al proxy dove deve prelevare il documento.

Collegamenti persistenti. Quando un client porge una richiesta al server, il server la soddisfa e poi chiude i socket, ma HTTP fa in modo che le richieste fatte dal client possano essere multiple impedendo al server di tagliare la comunicazione una volta eseguita una richiesta.

CAPITOLO 16

Client/Server con protocollo UDP e modelli di I/O

Client/Server con protocollo UDP. Alle volte può accadere che implementare un sistema di client e server attraverso un protocollo TCP orientato alla connessione risulti costoso in termini di overhead (ovvero in termini di quantità di pacchetti da scambiare tra client e server per instaurare la connessione). In questo caso è più agevole utilizzare un **protocollo UD**P ed implementare le poche operazioni di controllo del programma.
All'interno del server per la gestione delle licenze software, un utente U avvia un programma P sotto licenza, il programma P chiede al server S la licenza di esecuzione, il server S controlla il numero di utenti che stanno eseguendo il programma P, se esistono ancora licenze libere il server da il via libera al programma che si esegue, sennò nega l'esecuzione al programma ed il programma avvisa l'utente di provare più tardi.

Implementazione.

- ✔ **Ticket Model.** Il server fornisce dei **ticket digitali** che il client ed il server si possono scambiare, nella forma *pid.ticketnumber*.
- ✔ **Protocollo.**
  - ✗ **Per ottenere una chiave.** Il client invia un messaggio del tipo *HELO mypid* ed il server risponde *TICK ticketpid* oppure *FAIL* se nessun ticket è disponibile.
  - ✗ **Per restituire una chiave.** Il client invia un messaggio *GBYE ticketpid* ed il server risponde *THNX message*.
- ✔ **La struttura dati.** Possiamo utilizzare una struttura dati che associa ad ogni pid un ticket. Quando il ticket di libera si azzera l'informazione del pid.

Un server di licenze si adatta ad essere implementato con un protocollo UDP (quello con i datagram) data la sua semplicità.

Gestione dei problemi.

- ✔ **Crash di un client.** Quando un programma client si inceppa il ticket non viene più restituito, così viene perso e reso inutilizzabile. A lungo andare se tutti i programmi falliscono e si inceppano, si possono perdere tutti i ticket e rendere il server inutilizzabile. Per ovviare a questo problema è bene che il server mandi periodicamente una richiesta al client di restituzione del ticket per controllare che sia ancora attivo (è possibile tramite la gestione del segnale SIGALRM).
- ✔ **Crash del server.** Se il server si guasta si perdono tutte le liste delle licenze affidate ai client e loro non possono più avviarsi avendo perduto il loro meccanismo di innesco. Riavviare il server fa parte della risoluzione del problema ma non basta poiché è possibile la duplicazione delle chiavi. Serve allora che i client verifichino le loro chiavo continuamente.

Modelli di I/O. Nei sistemi Unix sono disponibili 5 modelli di input ed output:

- ✔ **I/O bloccante.** È il modello prevalente per le operazioni di input ed output, fino ad ora si è parlato di questo visto che le socket sono per natura bloccanti!
- ✔ **I/O non bloccante.** Quando la socket è in modalità non bloccante, equivale a richiedere al kernel che se il processo non può compiere l'operazione di input/output senza andare in sleep, non si sospenda il processo ma si torni un errore.
- ✔ **Multiplexing di I/O.** In questo modello si blocca il processo fino a quando non gli sarà possibile effettuare la sua funzione di input o output.
- ✔ **I/O signal driven (SIGIO).** Quando il descrittore è pronto per l'operazione si utilizzano i segnali che il kernel manda al processo per rendere nota la sua situazione.
- ✔ **I/O asincrono.** Il kernel inizia e ci segnala la fine dell'operazione di input/output.

I/O non bloccante. In Unix le chiamate di sistema che si occupano delle interazioni input/output possono essere del tipo **slow system call** se il processo che le chiama può essere bloccato su tali chiamate (*read()*, *write()*, *open()* e altre). Un processo bloccato su una system call può essere sbloccato da un segnale mandato allo stesso processo, che fa in mondo di forzare la chiamata a sistema tornando un errore; in questi casi la system call va gestita in modo tale da ripetersi se fallisce.
È possibile impostare le interazioni I/O di un processo in modo che non siano bloccanti, specificandolo direttamente nella chiamata open() con il flag O_NONBLOCK oppure durante il procedimento con la funzione fcnlt() e lo stesso flag di prima.

I/O multiplexing. Se un processo deve leggere da due file descriptor può succedere che utilizzando delle funzioni bloccanti si blocchi la lettura sul primo file compromettendo così la buona riuscita della lettura sul secondo file. Una soluzione può essere suddividere il processo in due processi distinti ognuno dei quali si occupa di leggere uno dei due file: nel caso della comunicazione con un modem, se si interrompe la comunicazione con il figlio il padre può venirlo a sapere tramite un segnale, idem se ad interrompersi è il padre, anche se in questa configurazione le cose si complicano.
Volendo risolvere il problema con un solo processo, mi basta impostare i file descriptor in modalità non bloccante, eseguo allora una read() sul primo file descrpitor e se trovo dati li processo altrimenti la chiamata rientra subito, dopodichè faccio la stessa cosa con il secondo file descriptor e dopo aver terminato aspetto pochi secondi prima di ricominciare a leggere il primo file desctriptor. Questa tecnica è detta **polling** e comporta un notevole spreco di tempo per la CPU, perchè quando non ho dati da leggere ho comunque un tempo d'attesa.
Un'altra soluzione è l'uso di **select()**, comunicando con essa al kernel quali file descriptor ci servono, quali condizioni devono verificarsi per ciascun file descriptor, quanto tempo vogliamo attendere. Le informazioni che otteniamo sono quanto file descriptor sono pronti e quali lo sono per read(), write() ed exception condition, in questa maniera riusciamo a compiere le funzioni I/O appropriate.

CAPITOLO 17
Memory Mapped I/O e Socket Unix Domain

Memory Mapped I/O. Il termine **Memory Mapped I/O** indica la possibilità di effettuare delle operazioni di input/output su un file solo in certe zone di memoria del processo. Tale meccanismo mappa il contenuto del file in un buffer in memoria, utilizza la tecnica della paginazione impiegata nella realizzazione della memoria virtuale così che tutte le operazioni compiute nella zona di memoria verranno trasferite al file tramite il meccanismo della memoria virtuale. Tutto ciò comporta la semplificazione delle operazioni di input/output, risparmiando sulla creazione di buffer intermedi, vengono caricate solo le parti del file che andranno modificate e le pagine che mappano il file si salvano automaticamente.
Sono due i segnali associati al memory mapped:

- ✔ **SIGSEGV.** Tentativo di accesso alla memoria non disponibile.
- ✔ **SIGBUS.** Tentativo di accesso alla memoria che non ha senso al momento dell'utilizzo.

Il sistema in genere fornisce porzioni di memoria la cui dimensione è il multiplo della dimensione della pagina. Se la memoria richiesta è più piccola la parte inutilizzata è riempita di zeri.

Socket Unix Domain. I protocolli Unix Domain rendono possibile un meccanismo di comunicazione client/server su un solo host senza l'utilizzo della rete ma con le stesse API che si usano su un sistema client/server su host differenti. Le **Socket Unix Domain** rappresentano una alternativa ai collegamenti IPC descritti in precedenza. Ne esistono di due tipi: **stream socket** (simili a TCP) e **datagram socket** (simili a UDP).

Passaggio di descrittori. Il passaggio di descrittori di file tra i processi che conosciamo sono quelli tra processi parenti padre e figlio oppure quelli in cui i descrittori rimangono aperti anche dopo la chiamata della exec(). Con le socket Unix Domain è possibile trasferire i descrittori anche tra processi che non sono parenti.

File Locking. Unix permette l'accesso simultaneo sia in lettura che in scrittura ad un file se questo ne possiede i privilegi. Tale politica è inaccettabile per i database che devono limitare l'accesso ai dati od ad una parte di essi (detto **record locking**). Il concetto di record locking su Unix prende il nome di **byte range locking**.
Nei sistemi operativi multi-tasking possono presentarsi le seguenti situazioni:

- ✔ Un server sta scrivendo un file di dati ed il client lo vuole leggere: il client leggerà un fle vuoto o incompleto. In questo caso il client dovrebbe sempre attendere che il server abbia finito di scrivere il file.
- ✔ Il client sta leggendo il file riga per riga quando il server si frappone ed incomincia a scrivere il file, così il client se lo vede modificare in tempo reale. Il server allora dovrebbe sempre attendere che il client finisca di leggere il file prima di modificarlo.

Per prevenire questi problemi sono stati individuati due tipi di file lock: il **file lock in lettura** ed il **file lock in scrittura**.
Esistono inoltre due tipi di file lock:

- ✔ **Locking file.** Il processo apre un "lock file" prima di scrivere sul file protetto. Se la creazione di quel file fallisce l'operazione sul file protetto si sospende per qualche secondo.
- ✔ **Locking region (locking record).** Il kernel Unix permette ad un processo di bloccare l'accesso ad una regione di un file (dove regione indica una serie di byte a partire da un dato punto di lunghezza finita). Se una richiesta di lock di regione confligge con un blocco già presente, il processo attende fino alla risoluzione del conflitto.

CAPITOLO 18
Shell Scripting

Tipi di SHELL. Esistono diversi tipi di shell, quali **Bourne Shell**, **Korn Shell**, **C Shell**, **Bash** (Bourne Again Shell). Scegliere tra le esistenti è una questione di gusti, ma la più diffusa in Unix è la Bash. Al momento del selezionamento dell'account in Unix viene riservata a tale account una shell. Una shell ha le seguenti caratteristiche:

- ✔ **Comandi built-in.**
  - ✗ **Esterni.** Quando si esplicità un comando esterno, il programma corrispondente a quel comando viene cercato in memoria, poi caricato ed eseguito.
  - ✗ **Interni (built-in).** Il comando viene riconosciuto ed eseguito direttamente dalla shell.
- ✔ **Caratteri speciali.** Detti anche **metacaratteri**, esistono caratteri speciali che la shell comprende come indicazione per l'azione che deve compiere.
- ✔ **Wildcards.** Si utilizzano per specificare i file pattern, la stringa contenente i wildcards viene sostituita con l'elenco dei file che soddisfano la condizione e con dei caratteri speciali.
- ✔ **Command substitution.** Gli apici '...' sono utilizzati per questo compito. Il comando specificato tra apici viene eseguito ed il suo output viene reimpiegato e sostituito al posto del comando.
- ✔ **Sequenze.**
  - ✗ **Non condizionali.** Il metacarattere viene utilizzato per eseguire due comandi in sequenza una dietro l'altro.
  - ✗ **Condizionali. ||** viene usato per permettere al secondo comando di essere eseguito solo se l'esecuzione del primo comando fallisce. **&&** invece esegue entrambi i comandi in successione solo se il primo non fallisce.
- ✔ **Raggruppamento di comandi.** Racchiudendo i comandi in parentesi, è possibile raggrupparli, in questo modo vengono eseguiti in una subshell e condividono gli stessi standard in/out/err.
- ✔ **Esecuzione in background.** Specificata dal carattere speciale **&**, viene creata una subshell che esegue il comando specificato in concorrenza con la shell che ha accettato il comando. Non prende in controllo della tastiera ed è utile per attività lunghe che non necessitano il controllo dell'utente.
- ✔ **Quoting.** I caratteri **'** inibiscono wildecards, command substitution, variable substitution mentre i caratteri **''** inibiscono solo le wildcards.
- ✔ **Subshell.** Quando si avvia un shell, questa può generare un processo figlio al quale affidare esecuzioni in background, esecuzioni di script o esecuzioni di comandi raggruppati. Le subshell possiedono una loro directory corrente e due distinte aree di variabili vengono gestite differentemente.
- ✔ **Variabili.** Ogni shell supporta due tipi di variabili:
  - ✗ **Locali.** Le subshell non le ereditano perchè servono a calcoli prettamente locali.
  - ✗ **Di ambiente.** Le subshell le ereditano perchè servono come ponte comunicativo tra processo padre e figlio.

  Entrambe le variabili contengono dei tipi di dato stringa ed ogni shell possiede le sue variabili di inizializzazione.
- ✔ **Utilizzo delle variabili.** Per accedere ad una variabile si usa **$**, mentre per assegnare un valore ad una variabile la sintassi cambia a seconda della shell. Esiste un comando per trasportare le variabili locali tra le variabili d'ambiente.
- ✔ **Script.** Qualsiasi sequenza di comandi può essere memorizzato su un file e poi mandato in esecuzione. Gli script sono utili per quelle sequenze lunghe e ripetitive da mandare in esecuzione. Per rendere eseguibile uno script lo si memorizza in un file, lo si abilita con il comando **chmod**, poi sulla shell si digita il nome del file. Lo script allora verrà preso in consegna da una subshell che lo eseguirà. La scelta della shell che lo

eseguirà è specificata dalla prima riga dello script, che può indicare la stessa shell o una differente tramite il suo pathname.

- ✔ **Here Document.** Viene copiato lo standard input fino ad una parola specifica, allora il comando viene eseguito con ciò che è stato scritto prima come input.

Espressioni. La shell non supporta direttamente la valutazione delle espressioni, infatti esiste per questo compito l'utility **expr expression** che valuta l'espressione e manda il risultato allo stdout. Per funzionare tutti gli elementi dell'espressione devono essere separati da spazi, tutti i metacaratteri devono essere introdotti da \, il risultato può una stringa o un numero.

Exit Status. Quando un processo termina torna sempre un **exit status**, dove per le convenzioni Unix se è 0 ha avuto successo e se non è 0 allora è fallito. Gli exit status non vengono usati come condizioni nei test nella struttura di controllo della shell.

Strutture di controllo.

- ✔ **If-Then-Elif-Else.** Se tutti i comandi in IF hanno successo sono eseguiti quelli in THEN, sennò viene eseguito ELIF, se l'ultimo comando in ELIF ha successo ok, sennò viene eseguito ELSE.
- ✔ **Case-In-Esac.** L'espressione in CASE viene confrontata e poi viene eseguita la funzione corrispondente a tale valore.
- ✔ **While-Do-Done.** Se la condizione in WHILE è verificata si esegue il comando in DO finchè la condizione sopra è verificata.
- ✔ **Until-Do-Done.** Ciò che è specificato in UNITL viene eseguito finchè non ha successo, sennò si esegue ciò che sta dentro DO.
- ✔ **For-In-Do-Done.** Simile al WHILE.

File System /proc. Possiamo considerare il **file system /proc** come una finestra sul kernel in esecuzione. I file qui presenti non corrispondono al dispositivo fisico, sono più oggetti "magici" che forniscono un accesso alle variabili ed alle strutture del kernel. Esiste una directory per ogni processo in esecuzione, identificata dal pid del processo stesso a cui appartiene. I file nella cartella /proc sono in parte formattati per essere leggibili.

APPENDICE A

I comandi di Unix

Il formato dei comandi in Unix. È del tipo **comando [argomento …]** dove l'argomento può essere inteso come un **opzione** o **flag** oppure come un **parametro** separati da almeno un separatore.
ESEMPIO: **ls -l -F file1 file2 file3** dove -l e -F sono opzioni e l'elenco dei file costituisce l'elenco dei parametri da utilizzare.

Nome	Descrizione	Formato
who	Stampa il nome, il terminale, data e ora del login di tutti gli utenti del sistema.	who [...]
who am I	Come sopra, ma mostra le informazioni solo di colui che eseue il comand.	who [am I]
date	Modifica o stampa la data corrente nel formato indicato	date [...]
man	Accede al manuale on-line di Linux, suddiviso a sua volta in pagine per ogni famiglia di comandi.	man [opzione] titolo

whatis	Mostra solo la sezione NOME della pagina del manuale.	whatis [comando]
apropos	Cerca i comandi nel manuale che contengono la parola specificata nella loro descrizione.	apropos [word]
pwd	Stampa il pathname completo della cartella corrente.	pwd [...]
cd	La directory specificata nel comando diventa la cartella corrente, senza specifiche si torna alla directory superiore.	cd [directory] [...]
ls	Mostra il contenuto della directory specificata in ordine alfabetico nel rispetto delle opzioni specificate. Se non è specificato nulla lista il contenuto della cartella corrente.	ls [options] [directory]
du	Stampa il numero di blocchi di ogni file e di ogni cartella in maniera ricorsiva. Se non è specificato NAME si intende la directory corrente.	du [options] [name]
mkdir	Crea una nuova cartella di nome specificato.	mkdir [directory]
rmdir	Rimuove le cartelle indicate solo se vuote.	rmdir [directory]
ln	Associa un nuovo nome (name2) al file name1 con un link, non succede se è una cartella.	ln [name1] [name2]
id	Comando che stampa lo user name e user-id, group name e group-id di tutti gli utenti.	id [-a] [user name]
last	Fornisce le informazioni riguardo l'ultimo login di un utente, se non specificato di tutti gli utenti.	last [name ...]
chmod	Modifica i permessi di un file con la seguente configurazione: [ugoa] [+ - =] [r w x]	chmod [permissions] [filename]
mv	Sposta un file o una cartella name sotto la directory target, se name e target non sono cartelle name sostituisce target.	mv [options] [name ... target]
cp	Come mv, ma name viene compiato.	cp [options] [name] … target
rm	Rimuove il file indicato, se si tratta di una cartella serve l'opzione -r (allora prima di eliminare la cartella tutti i suoi file verranno elminati ricorsivamente).	rm -r [name] ...
touch	Aggiorna la data dell'ultimo accesso (-a), dell'ultima modifica (-m) di filename. Se il file non esiste lo crea, senza time si tratta dell'ora corrente.	touch [option] [time] filename ...
find	Ricerca tutti i file in pathname che soddisfano l'espressione specificata.	find pathname ... [espression]
ps	Fornisce una lista completa dei processi in corso.	ps [options]
tty	Permette di conoscere il nome del file che rappresenta il nostro terminale.	tty [options]
write	Manda un messaggio ad un altro utente.	write [user] [ttyname]
stty	Consente all'utente di conoscere e modificare le impostazioni per un terminale.	stty [options]
ipcs	Ottiene lo stato di tutti gli oggetti IPC System V presenti nel sistema.	ipcs [options]
ipcrm	Permette di rimuovere gli oggetti IPC dal sistema.	ipcrm <msg|sem|shm> <IPC ID>

APPENDICE B

System Call POSIX.1

Lo standard POSIX. Per compilare un sorgente conforme alle specifiche POSIX è necessario includere all'inizio del file le librerie necessarie (quelle esclusivamente POSIX sono pochissime).

void **perror** (const char *str)

La funzione **perror()** legge il valore della variabile **errno**, convertendolo in una stringa in lingua inglese che poi verrà stampata con il contenuto di **str** anteposto al messaggio.

int **open** (const char *path, int oflag, ...)

La funzione **open()** apre o crea il file specificato dal **path** secondo le specifiche di **oflag** (che può assumere moltissimi valori). Restituisce poi il **file descriptor** associato al file in questione.

int **creat** (const char *path, mode_t mode)

Equivale alla funzione open() con i permessi in sola scrittura, la **creat()** crea un nuovo file aperto in scrittura. Se il file esiste già lo svuota. **Mode** definisce i permessi del file.

int **close** (int filedes)

La funzione **close()** chiude il flusso del file relativo al **file descriptor** specificato. Ritorna poi l'esito della chiusura.

ssize_t **read** (int filedes, void *buf, size_t nbyte)

La funzione **read()** legge in **buf** un numero pari a **nbyte** di byte dal file **filedes** a partire dalla posizione corrente, per poi aggiornarla. Restituisce poi il numero di byte effettivamente letti.

ssize_t **write** (int filedes, const void *buf, size_t nbyte)

Lavora inversamente alla funzione read(), scrivendo da **buf** una quantità **nbyte** di informazione a partire dalla posizione corrente che viene poi aggiornata, restituendo alla fine il numero effettivo di byte scritti in **filedes**.

off_t **lseek** (int filedes, off_t offset, int whence)

Questa funzione non offre nessun servizio di input/output, semplicemente sposta la posizione corrente di **filedes** di una quantità **offset** a partire dalla pozione indicata in **whence**, restituendo poi la posizione di arrivo dopo lo spostamento.

int **link** (const char *existing, const char *new)

Crea un nuovo (hard) link **new** al file **existing**, restituendo poi l'esito della funzione.
Per cambiare nome ad un file: int **rename** (const char *old, const char *new).

int **unlink** (const char *path)

Elimina il (hard) link **path**, se è l'ultimo dealloca il file. Se il file è in esecuzione si attende la fine dell'esecuzione per deallocarlo.

int **mkdir** (const char *path, mode_t mode)
int **rmdir** (const char *path)
int **chdir** (const char *path)

Per creare, rimuovere e cambiare una directory.

int **chmod** (const char *path, mode_t mode)
int **chown** (const char *path, uid_t owner, gid_t group)
int **utime** (const char *path, const struct utimbuf *times)

La prima funzione modifica i permessi come specificato in **mode** di un file **path**. La seconda modifica proprietario e gruppo del file **path**. La terza vuole modificare i tempi di ultimo accesso e di ultima modifica.

int **dup** (int filedes)
int **dup2** (int oldfd, int newfd)

La prima funzione associa al file di file descriptor **filedes** un file descriptor aggiuntivo e lo ritorna al chiamante. La seconda, sostituisce il nuovo file descriptor al vecchio e restituisce il muovo valore al chiamante.

int **stat** (const char *path, struct stat *buf)

Questa funzione preleva dalla struttura stat alcuni parametri relativi al file **path** e li memorizza nella struttura **buf**. E' la funzione che utilizziamo per leggere gli elementi della struttura stat.

int **ioctl** (int filedes, int request, ...)

E' una funzione che rende possibile tutte quelle operazioni sui file che non sono state espresse attraverso l'interfaccia standard.

pid_t **fork** (void)

Creo un nuovo processo del quale ritorno al padre solo il process ID.

pid_t **wait** (int *statloc)
pid_t **waitpid** (pid_t pid, int *statloc, int options)

La prima funzione sospende il processo chiamante fino al termine di un processo figlio e restituisce il pid del figlio terminato. La seconda invece permette di specificare di quale figlio attendere la terminazione.

void(***signal**(int signo, void (*func)(int)))(int)

Restituisce un puntatore al precedente gestore del segnale. **Signo** è il numero identificativo del segnale, mentre **func** o è un valore standard oppure è una funzione decisa dall'utente.

int **alarm** (unsigned int sec)
int **pause** (void)

La prima funzione torna 0 oppure il numero di secondi che le mancano, imposta una sveglia che invia un segnale allo scadere del tempo indicato. La seconda funzione sospende il processo chiamante finchè non riceve un segnale del tipo generato dalla alarm().

int **kill** (pid_t pid, int signo)

La funzione invia un segnale di tipo **signo** al processo indicato da **pid**.

int **raise** (int signo)

La funzione invia un segnale di tipo **signo** al processo che invoca la funzione.

int **sigaction** (int signo, const struct sigaction *act, struct sigaction *oact)

La funzione imposta un'azione per i segnali. **Signo** indica il segnale per il qualche vogliano determinare l'azione, **act** specifica la suddetta azione, **oact** registra le informazioni precedenti per signo.

int **tcgetattr** (int fd, struct termios* info)

Copia le impostazioni correnti per il driver da terminale associato al file aperto **fd** nella struttura **termios**.

int **tcsetattr** (int fd, int when, struct termios* info)

La funzione imposta gli attributi presenti nella struttura **termios** per il dirver di terminale associato al file **fd**.

int **ioctl** (int fd, int operation [, argoments])

La chiamata di sistema **ioctl()** permette di controllare un dispositivo.

int **pipe** (int filedes[2])

La funzione crea una pipe senza nome e la apre sia in lettura che in scrittura. Assegna a filedes[0] il file descriptor del lato aperto in lettura ed a filedes[1] il file descriptor del lato aperto in scrittura.

FILE* **popen** (const char *cmdstring, const char *type)
int **pclose** (FILE *fp)

La prima funzione restituisce un puntatore a file se va a buon fine, dove **cmdstring** è un comando da eseguire e **type** rappresenta il flag della tipologia del flusso stanard al quale il puntatore a file è legato. La seconda, chiude gli stream.

int **mkfifo** (const char *pathname, mode_t mode)

La funzione che un file nominato di tipo FIFO.

key_t **ftok** (const char *pathname, int proj_id)

La funzione genera un valore per una chiave a partire da un numero da 0 1 255 (**proj_id**) e dal nome di un file realmente esistente specificato in **pathname**.

int **msgget** (key_t key, int flag)
int **semget** (key_t key, int nsems, int flag)
int **shmget** (key_t key, int size, int flag)

Con una di queste funzioni si crea la struttura IPC desiderata (stesso principio di open()).

int **socket** (int domain, int type, int protocol)

La funzione crea un socket del tipo specificato e restituisce il socket descriptor.

int **connect** (int sockfd, struct sockaddr *servaddr, int addrlen)

La funzione è usata da un client per connettere la socket locale **sockfd** alla socket di indirizo specificato.

int **listen** (int sockfd, int backlog)

La funzione viene usata da un server orientato alle connessioni per segnalare che è in attesa di una connessione sulla socket locale specificata.

int **accept** (int sockfd, struct sockaddr *peer, int *addrlen)

Il server usa questa funzione per attendere una connessione sulla socket specificata.

int **bind** (int sockfd, struct sockaddr *addr, socklen_t len)

Assegna alla socket specificata un indirizzo oppure un nome.

int **send** (int sockfd, const void *msg, int len, int flag)
int **recv** (int sockfd, void *buf, int len, unsigned int flags)

Nella prima funzione **sockfd** è il file descriptor associato alla socket, **msg** è un puntatore ai dati da inviare, **len** è la lunghezza dei dati che devono essere inviati e **flag** per comodità è messo a 0. Nella seconda uguale.

int **recvfrom** (int sockfd, char *buf, int nbyte, int flag, struct sockaddr *from, int *addrlen)
int **sendto** (int sockfd, char *buf, int nbyte, int flag, struct sockaddr *to, int *addrlen)

Servono per ricevere ed inviare un datagramm.

int **fcntil** (int fd, int cmd, …/*int args*/)

La funzione viene impiegata per:

- ✔ duplicazione di un file descriptor esistente;
- ✔ impostare od ottenere i flag associati ad un file descriptor;
- ✔ impostare od ottenere i flag si stato di un file;
- ✔ impostare od ottenere la proprietà dell'I/O asincrono;
- ✔ impostare od ottenere un record lock.

void ***mmap** (void *start, size_t lenght, int prot, int flags, int fd, off_t offset)

La funzione esegue l'operazione di mapping in memoria del file associato al file descriptor **fd**, restituisce il puntatore alla porzione di memoria mappata.

int **munmap** (void *start, size_t lenght)

La funzione compie l'un-mapping della regione di lunghezza **lenght** che inizia a **start**.

Bibliografia e Crediti. Le informazioni ivi contenute sono il risultato di una "condensazione" delle dispense/slide proiettate a lezione dal professor Giovanni Di Domenico per il corso di Sistemi Operativi e Laboratorio tenuto presso l'Università degli Studi di Ferrara, laurea triennale in Informatica, AA 2009/2010.
Tale riassunto è finalizzato ad un'operazione di studio personale e non sostituisce in nessun modo le dispense del professore.
Autore di questo riassunto è Farinelli Agnese, disponibile per il download all'indirizzo http://www.thalionwen.altervista.org/.