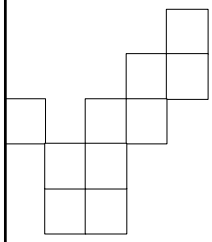


COMUNICAZIONE FRA PROCESSI

Segnale
Passaggio di dati fra processi

Comunicazione fra processi

- LINUX fornisce un ricco ambiente per la comunicazione fra processi.
- La comunicazione può limitarsi alla notifica del verificarsi di un evento, da parte di un processo ad un altro processo (*segnale*)
- Ma può anche coinvolgere il trasferimento di dati fra processi.

Segnale

- Meccanismo standard dello UNIX usato per comunicare ad un processo che un evento si è verificato.
- Può essere generato da un processo utente o dal kernel a seguito di un errore software o hardware.
- Ogni segnale ha un nome a cui viene associato una costante intera positiva definita in *signal.h*

Segnale

- Le azioni associate ad un segnale sono le seguenti:
  - **Ignorare il segnale** (tranne per SIGKILL e SIGSTOP).
  - **Catturare il segnale** (equivale ad associare una funzione quando il segnale occorre).
  - **Eseguire l'azione di default associata** (terminazione del processo per la maggior parte dei segnali).

Passaggio di dati fra processi

- Il passaggio di dati tra processi può avvenire:
  - Utilizzando le **pipe**
  - Utilizzando le **FIFO**
  - Utilizzando **IPC di System V**

Pipe

- Una pipe è un canale di comunicazione tra due processi.
  - Un processo scrive nella pipe.
  - L'altro processo legge dalla pipe.
- Una pipe può essere vista come un file perché le operazioni di scrittura e lettura sono come quelle dei file.
  - Un processo ha la pipe (file) aperta in scrittura
  - L'altro processo ha la pipe (file) aperto in lettura.

Caratteristiche delle Pipe

- Flusso di dati in una sola direzione (half-duplex).
- Possono essere utilizzate solo tra processi che hanno un antenato in comune (che ha creato la pipe).

7

System call: Pipe

#include <unistd.h>
int pipe(int *fd[2]*);

Restituisce 0 se OK, -1 in caso di errore

- fd[1] è il file descriptor del file (pipe) aperto in scrittura.
- fd[0] è il file descriptor del file (pipe) aperto in lettura.
- inoltre l'output di fd[1] corrisponde all'input di fd[0].

8

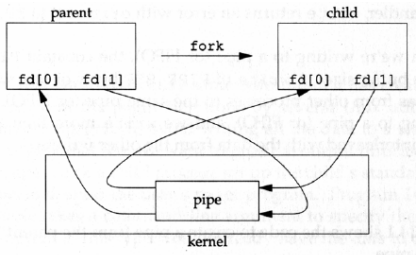
Utilizzo della Pipe (1)

- La funzione pipe ritorna i file descriptor allo stesso processo … mentre una pipe dovrebbe connettere due processi.
- fork : padre e figlio possono comunicare usando la pipe.
  - Chi deve scrivere chiude fd[0] e tiene aperto fd[1].
  - Chi deve leggere chiude fd[1] e tiene aperto fd[0].

```
int fd[2];
…
pipe(fd);
pid = fork();
…
```

9

Situazione dopo pipe + fork



10

Utilizzo della Pipe (2)

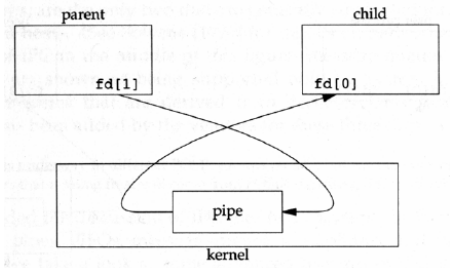
- Una delle possibilità dopo la fork è la seguente

```
if (pid>0) { // padre
close (fd[0]);}
else if (pid==0) { // figlio
close (fd[1]);}
```

- Questo crea un canale dal padre verso il figlio

11

Pipe da padre a figlio



12

Utilizzo della Pipe (3)

- Una volta che è stata creata la pipe e che è stato scelto il verso di comunicazione è possibile utilizzare le funzioni di I/O che lavorano con i file descriptor (tranne **open**, **creat**e **lseek**).
- Una pipe è un canale di comunicazione in cui i dati vengono letti nello stesso ordine in cui vengono scritti.

13

I/O su pipe (1)

- Funzione **write**
  - Quando la pipe si riempie (la costante PIPE_BUF specifica la dimensione), la **write** si blocca fino a che la **read** non ha rimosso un numero sufficiente di dati.
  - La scrittura è atomica se i dati sono <= PIPE_BUF.
  - Se il descrittore del file che legge dalla pipe è chiuso, una **write** genererà un errore (segnale SIGPIPE).

14

I/O su pipe (2)

- Funzione **read**
  - Legge i dati dalla pipe nell'ordine in cui sono scritti.
  - Non è possibile rileggere o rimandare indietro i dati letti.
  - Se la pipe è vuota la **read** si blocca fino a che non vi siano dei dati disponibili.
  - Se il descrittore del file in scrittura è chiuso, la **read** restituirà EOF dopo aver completato la lettura dei dati.

15

I/O su pipe (3)

- Funzione **close**
  - La funzione **close** sul descrittore del file in scrittura agisce come *end-of-file* per la **read**.
  - La chiusura del descrittore del file in lettura causa un errore nella **write**.

16

Considerazioni

- Più di un processo che scrive
  - è possibile avere più processi che scrivono in una pipe.
  - l'unico problema è che il processo che legge riceve l'input mescolato.
  - Se ogni messaggio è <= PIPE_BUF allora la cosa funziona.
- Più di un processo che legge
  - Nulla lo vieta ma non ha senso!

17

Pipe vs FIFO

- la *pipe* può essere usata solo tra processi "imparentati" (che hanno un antenato comune che ha creato la pipe).
- la *fifo* consente di scambiare dati tra processi qualsiasi.

18

FIFO (named pipe)

#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char **pathname*, mode_t *mode*);

Restituisce 0 se OK, -1 in caso di errore

19

Apertura di una FIFO

- Una volta creata, una FIFO può essere aperta con **open** oppure **fopen**.
- La FIFO è un tipo di file. Si può testare il campo **st_mode** della struttura **stat** con la macro **S_ISFIFO**.
- Anche se somiglia ad un file (si utilizzano le stesse funzioni di I/O, risiede sul filesystem) ha le caratteristiche di una **pipe**:
  - I dati scritti vengono letti in ordine **first-in-first-out.**
  - Le chiamate in lettura e scrittura sono atomiche se la quantità di dati è minore di **PIPE_BUF.**
  - Non è possibile rileggere i dati già letti né posizionarsi all'interno con **lseek.**

20

Sincronizzazione (1)

- Normalmente una **open** in lettura si blocca fino a che un processo non effettua una **open** in scrittura (e viceversa).
- Se viene utilizzato il flag O_NONBLOK
  - una **open** in lettura ritorna immediatamente anche se non c'è un processo che ha effettuato una **open** in scrittura.
  - una **open** in scrittura restituisce un errore se non viene effettuata una **open** in lettura.

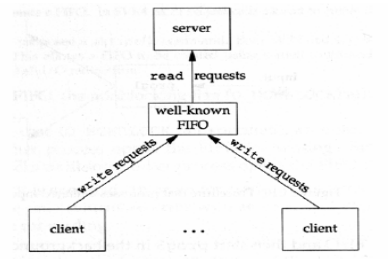
21

Sincronizzazione (2)

- Se si effettua una **write** su una FIFO che nessun processo ha ancora aperto in lettura, viene generato il segnale SIGPIPE.
- Quando l'ultimo degli scrittori chiude una FIFO, viene generato un **end-of-file** per il processo lettore.

22

Comunicazione Client-Server con FIFO



LIMITAZIONE : i messaggi dei clienti devono essere più piccoli di PIPE_BUF altrimenti si mescolano.

23

Comunicazione dal server al client

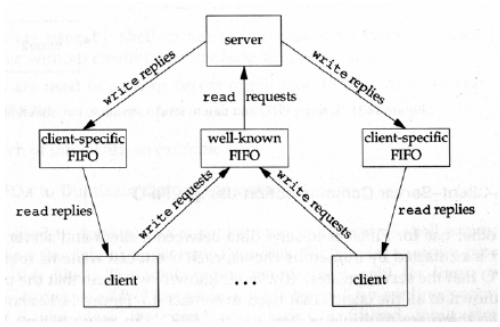
- Per rispondere il server deve usare una FIFO dedicata per ogni processo.
- Quindi client e server devono accordarsi sul nome della FIFO che il server userà per comunicare con quel particolare client.
- Il server può sfruttare informazioni fornite dal client (ad esempio il pid).
- Ogni processo quindi inserisce il pid nel messaggio verso il server.
- Il server può usare un nome del tipo

FIFO 3754

dove 3754 è il pid del processo client.

24

Client-server



25

IPC System V

- Code di Messaggi
- Semafori
- Memoria Condivisa

26

Identificatori e chiavi

- Ogni struttura IPC è indirizzata tramite un **identificatore** che viene assegnato dal kernel quando la struttura viene creata.
- Quando si crea una struttura IPC bisogna specificare una **chiave** (di tipo **key_t**) che sarà convertita in **identificatore** dal kernel.

27

Criteri di scelta della chiave

- Utilizzare la chiave speciale **IPC_PRIVATE** che garantisce la creazione di una struttura nuova e passare l'**identificatore** tra i vari processi che utilizzeranno la struttura.
- Fissare a priori tra client e server una chiave (ad esempio in un header file) che sarà utilizzata nel server per creare la struttura.
- Fissare a priori un **pathname** ed un **ID** ed utilizzare la funzione **ftok** per ottenere una chiave.

28

Vantaggi e svantaggi

- **IPC_PRIVATE** garantisce che la struttura creata sia nuova ma ha lo svantaggio di dover passare l'identificatore tra server e client spesso tramite file (accesso al file system).
- Definire una **chiave** a priori (oppure un **path** ed un **ID**) è più veloce ma non garantisce che la **chiave** non sia già stata utilizzata da altri processi.
- Le strutture IPC devono essere esplicitamente eliminate.

29

Creazione/Riferimento

- per creare/riferirsi ad una struttura IPC
  - **msgget**
  - **semget**
  - **shmget**
- hanno 2 argomenti in comune
  - **key_t** *key*
  - **int** *flag*

30

Flag

- contengono i permessi di uso della struttura (tranne quelli di esecuzione, che sono ignorati).
  - usate per esempio: 660, 420, etc ...
  - se si vuole creare una struttura si fa l'**OR** con il bit IPC_CREAT.
  - se si vuole essere sicuri che non esiste già la struttura si fa l'**OR** anche con il bit **IPC_EXCL** (come nella open …).

31

Chiave

- **IPC_PRIVATE** se si vuole creare una struttura nuova.
- una chiave (ad esempio scambiata tramite un header file) che non è associata a nessuna altra struttura esistente (si spera …).
- se ci si riferisce ad una struttura esistente (per es. in un client) bisogna usare la chiave usata dal server
  - se è stata creata con **IPC_PRIVATE** si usa direttamente l'**identificatore** generato nel server (e passato eventualmente tramite file) nelle chiamate **msgsnd**, **msgrcv**.

32

Permessi

- quando è creata una struttura IPC viene associata ad essa una struttura **ipc_perm** che contiene i permessi, il proprietario, etc...
- possono essere modificati solo dal creatore o da root (alcuni campi) con le funzioni di controllo
  - **msgctl**
  - **semctl**
  - **shmctl**

33

IPC vs pipe/FIFO

- le strutture IPC sono systemwide
- Non vengono rimosse quando il processo che le ha create termina la sua esecuzione.
  - si possono cancellare con **ipcrm** da shell
- una pipe è rimossa automaticamente.
- una fifo resta nel sistema, ma è svuotata quando l'ultimo processo che vi si riferisce termina la sua esecuzione.

34

Code di Messaggi

- sono liste linkate di messaggi.
- ci si riferisce tramite l'identificatore di coda.
- i nuovi messaggi sono inclusi alla fine con **msgsnd.**
- i messaggi sono prelevati dalla coda con **msgrcv** (non necessariamente con ordine fifo, ma in base al loro **tipo**).
- ogni msg ha due campi
  - un **tipo** (long integer).
  - i **dati** effettivi (di qualunque tipo, in genere stringhe).

35

Code di Messaggi

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int flag); /* apre una coda esistente o ne crea una nuova */
```

- Restituisce ID della coda se OK, -1 in caso di errore.

36

Struttura associata ad una coda

```
struct msqid_ds {
  struct ipc_perm  msg_perm; /* see Section 14.6.2 */
  struct msg  *msg_first;  /* ptr to first message on queue */
  struct msg  *msg_last;   /* ptr to last message on queue */
  ulong        msg_cbytes; /* current # bytes on queue */
  ulong        msg_qnum;   /* # of messages on queue */
  ulong        msg_qbytes; /* max # of bytes on queue */
  pid_t        msg_lspid;  /* pid of last msgsnd() */
  pid_t        msg_lrpid;  /* pid of last msgrcv() */
  time_t       msg_stime;  /* last-msgsnd() time */
  time_t       msg_rtime;  /* last-msgrcv() time */
  time_t       msg_ctime;  /* last-change time */
};
```

37

Funzione msgctl

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

int msgctl(int *msqid*, int *cmd*, struct msqid_ds **buf*); */* effettua varie operazioni sulla coda */*

- Restituisce 0 se OK, -1 in caso di errore

38

Argomento cmd

- IPC_STAT
  - Copia la struttura **msqid_ds** in **buf**
- IPC_SET
  - Copia alcuni campi di **buf** in **msg_perm**
- IPC_RMID
  - Rimuove la coda di messaggi

39

Funzione msgsnd

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

int msgsnd(int *msqid*, cons void **ptr*, size_t *nbytes*, int *flag*); */* funzione per accodare messaggi */*

- Restituisce 0 se OK, -1 in caso di errore

40

Struttura del messaggio

struct mesg {

long mtype;

char mtext[512]; }

41

Flag

- Il flag può assumere valore IPC_NOWAIT (simile al flag O_NONBLOCK).
- Se il flag **non** è settato, msgsnd si blocca se la coda è piena.
- Se il flag è settato a IPC_NOWAIT e se la coda è piena, allora una chiamata a msgsnd restituisce un errore.

42

Funzione msgrcv

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

int msgrcv(int *msqid*, void **ptr*, size_t *nbytes*, long *type*, int *flag*); */* funzione per prelevare messaggi dalla coda */*

- Restituisce la dimensione del campo mtext se OK, -1in caso di errore.

43

Argomento type

- **type**== 0
  - Viene restituito il primo messaggio nella coda
- **type**> 0
  - Viene restituito il primo messaggio con **mtype** = **type**
- **type**< 0
  - Viene restituito il primo messaggio il cui **mtype** sia il più piccolo tra tutti gli **mtype** <= di abs(**type**).

44

Semafori

- sono differenti dalle altre forme di IPC.
- sono contatori usati per controllare l'accesso a risorse condivise da più processi.
- il protocollo per accedere alla risorsa è il seguente
  - 1) testare il semaforo
  - 2) se > 0, allora si può usare la risorsa (e viene decrementato il semaforo).
  - 3) se = 0, processo --> sleep finché > 0 e goto step 1
- quando ha terminato con la risorsa, incrementa il semaforo.

45

Semafori

- Forma intuitiva:
  - semaforo binario
- Semaforo System V:
  - insieme di semafori... bisogna specificarne il numero
  - Creazione indipendente dall' inizializzazione

46

Semafori

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

int semget(key_t *key*, int *nsems*, int *flag*);

/ funzione per la creazione di un semaforo */*

- Restituisce ID del semaforo se OK, -1 in caso di errore.

47

Esempio

- semid = semget (key, 1, IPC_CREAT|IPC_EXCL|0666);
  - crea un nuovo semaforo (in un server)
  - key: regole identiche a quelle delle code di messaggi
- semid = semget (key, 0, 0);
  - apre un semaforo (non lo crea !) in un client

48

Strutture associate ai semafori

```
struct semid_ds {
  struct ipc_perm  sem_perm;  /* see Section 14.6.2 */
  struct sem  *sem_base;  /* ptr to first semaphore in set */
  ushort       sem_nsems; /* # of semaphores in set */
  time_t       sem_otime; /* last-semop() time */
  time_t       sem_ctime; /* last-change time */
};
```

```
struct sem {
  ushort  semval;   /* semaphore value, always >= 0 */
  pid_t   sempid;   /* pid for last operation */
  ushort  semncnt;  /* # processes awaiting semval > currval */
  ushort  semzcnt;  /* # processes awaiting semval = 0 */
};
```


Funzione semctl

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

- int semctl(int *semid*, int *semnum*, int *cmd*, union semnum *arg*);

/ utilizzata per varie operazioni sui semafori */*

Argomento cmd

- IPC_STAT, IPC_SET, IPC_RMID
  - Come per le code di messaggi
- GETVAL
  - Restituisce il valore del semaforo **semnum**
- SETVAL
  - Setta il semaforo **semnum** con il valore in **arg.val**
- GETALL
  - Restituisce i valori di tutti i semafori nell'array **arg.array**
- SETALL
  - Setta tutti i semafori con i valori di **arg.array**

Funzione semop

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

int semop(int *semid*, struct sembuf *semoparray[ ]*, size_t *nops*); **/ effettua una serie di operazioni su un set di semafori */*

- Restituisce 0 se OK, -1 in caso di errore

Struttura sembuf

```
struct sembuf {
  ushort  sem_num;  /* member # in set (0, 1, ..., nsems-1) */
  short   sem_op;   /* operation (negative, 0, or positive) */
  short   sem_flg;  /* IPC_NOWAIT, SEM_UNDO */
};
```

- *sem_num* : numero del semaforo
- *sem_op*:
- < 0 : richiesta. Viene sottratto il valore di *sem_op* da *sem_val*. Se abs(*sem_op*) > *sem_val* il processo dorme finché non è disponibile la risorsa
- > 0 : rilascio. Il valore viene sommato a *sem_val*
- = 0 : controllo. Serve a controllare se tutte le risorse sono allocate

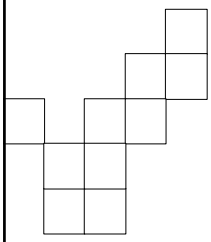
Memoria condivisa

- Permette a due o più processi di condividere una data regione di memoria.
  - L'unico trucco da utilizzare con la memoria condivisa è quello di sincronizzare l'accesso alla memoria trai vari processi.
  - Se il server sta ponendo i dati nella memoria condivisa, il client non deve accedervi fino a quando il server non ha terminato.
    - Spesso i semafori sono utilizzati per la sincronizzazione.

Memoria condivisa

- Svantaggi
  - Non offre alcun metodo di sincronizzazione
    - Un processo non può chiedere al sistema operativo se qualche dato sia stato scritto in una regione di memoria condivisa.
    - Né può sospendere l'esecuzione fino a che una tale operazione di scrittura si sia verificata.

55



FILE SYSTEM

Struttura
FHS
ext2
Mount di un filesystem

Struttura del filesystem di Linux

- Il filesystem di LINUX è molto simile al filesystem standard di UNIX ma naturalmente con alcune piccole differenze.
- La chiave di lettura consiste nel comprendere la sua struttura sottostante...(vedremo il funzionamento interno e la metodologia impiegata per la gestione dei file).
- La struttura di directory è definita come una gerarchia ad albero:
  - Al livello più alto c'è la directory principale (**root**) ed è l'unica directory che si incontra a questo livello.
  - Tutte le altre directory sono referenziate in rapporto alla directory root che è indicata come la directory /
    - La directory root contiene un piccolo gruppo di sottodirectory e di file. Da notare che la directory /root è una sottodirectory della directory ed è la home directory dell'utente root.

57

FHS - Storia

- 1993: inizia un'opera diretta alla revisione della disposizione dei file e delle directory in LINUX. Questo progetto è noto con il nome *Filesystem Standards* o *FSSTND*.
- Dopo un certo periodo di tempo, la sua portata si è ampliata fino ad includere problemi che erano di ordine generale anche per altri sistemi operativi di tipo UNIX.
  - In considerazione di questa focalizzazione espansa, il progetto è stato ribattezzato *Filesystem Hierarchy Standard* (o *FHS*).

58

FHS – Caratteristiche generali

- Se da un lato FHS contiene una grande quantità di dettagli relativi a ciò che il filesystem di LINUX dovrebbe o non dovrebbe essere, il suo obiettivo principale è quello di fornire un filesystem coerente e standardizzato. Questo filesystem standardizzato può essere definito in due categorie ortogonali:
  - **File condivisibili e file non condivisibili**: questa categoria consiste di file che possono essere condivisi fra numerosi host e di file che sono specifici di un particolare host.
  - **File statici e variabili**: questa categoria include i file statici, come la documentazione, i file binari delle applicazioni e le librerie, che non cambiano senza l'intervento dell'amministratore di sistema. Qualsiasi file che cambia senza l'intervento dell'amministratore di sistema viene considerato un file di dati variabile.

59

Specifiche FHS per la directory root

- Come anticipato in precedenza, la directory root è la prima e unica directory al sommo livello della gerarchia ad albero. Come tale, per essa sono previste speciali considerazioni. Dal documento FHS: **"… il contenuto del filesystem root dovrebbe essere in grado di effettuare il boot, il ripristino, il recupero e/o la riparazione del sistema …"**
- Al fine di realizzare questi scopi, il filesystem root deve contenere i componenti essenziali per il boot del sistema, gli strumenti essenziali per la riparazione del sistema e le utility essenziali per il backup o il ripristino del sistema pur mantenendo il filesystem root il più piccolo possibile.
  - Un filesystem root piccolo è meno esposto ad errori e si presta a una più facile manutenzione qualora qualcosa dovesse andare per il verso storto.

60

Filesystem di Linux – ext2

- Una volta che un nuovo drive appena aggiunto viene riconosciuto dal sistema, non può essere utilizzato fino a quando non è stato partizionato e su esso non è stato creato un filesystem.
- Il filesystem di default per Linux è il filesystem esteso di tipo 2 (ext2).
  - La struttura del filesystem viene costruita quando viene eseguita la utility *make filesystem (mkfs)*. Questa utility è in effetti soltanto un programma che chiama routine subordinate per la creazione del filesystem. Quando si crea un filesystem *ext2*, la utility mkfs in effetti chiama *mke2fs* per creare l'esatta struttura del filesystem. Una volta creata, la struttura del filesystem non è più possibile modificarla senza riformattare la partizione o meglio, in altra parole, senza ricreare un nuovo filesystem con l'utility *mkfs*.
  - Quando viene creato, un filesystem ext2 genera nella partizione del disco rigido una serie di aree dette comunemente gruppi di blocchi. Ogni gruppo di blocchi viene segmentato in numerose sezioni più piccole.

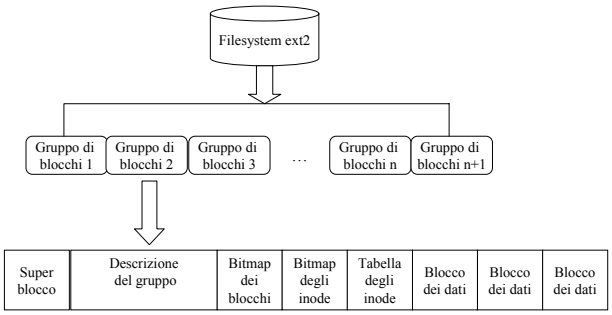
61

Filesystem di Linux – ext2

Sezione del gruppo di blocchi	descrizione
Superblocco	In esso sono memorizzate informazioni relative all'intero filesystem. Ogni gruppo di blocchi contiene un superblocco, ma si tratta soltanto di una copia di back up del superblocco presente nel primo gruppo di blocchi.
Descrizione del gruppo	Sono memorizzate informazioni su ciascun gruppo di blocchi. Ogni gruppo di blocchi contiene un descrittore di gruppo, ma si tratta di duplicati del descrittore del gruppo di blocchi presente nel primo gruppo di blocchi. Qui risiedono i puntatori alla tabella inode.
Bitmap dei blocchi	È un bitmap che indica quali blocchi sono in uso.
Bitmap degli inode	È un bitmap che indica quali inode sono in uso.
Tabella degli inode	È una tabella di inode allocati su questo gruppo di blocchi.
Blocco dei dati	È dove sono memorizzati i dati.

62

Filesystem di Linux – ext2



63

Filesystem di Linux – ext2

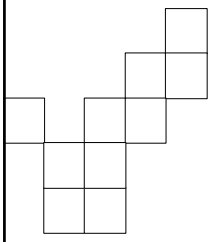
- Un *inode* è fondamentalmente un puntatore a un file che però contiene informazioni riguardanti il file stesso, come permessi, proprietà, dati dell'ultima modifica e puntatori agli effettivi blocchi dei dati. Un inode non contiene alcun file di dati. Esiste esattamente un inode per ogni file. Un filesystem *ext2* viene costruito con un gruppo fisso di inode al momento della creazione.

64

Mount di un filesystem

- Perché un filesystem appena creato sia accessibile in Linux, occorre effettuarne il mount.
  - Risale ai giorni in cui i file venivano mantenuti sui nastri che dovevano essere appunto "montati" affinché il filesystem potesse accedere ad essi.
  - Questo metodo di trattare con le partizioni rende più facile la manutenzione del filesystem.
  - I filesystem di Linux possono essere raggruppati in modo logico, in maniera da sembrare un solo filesystem quando in effetti non lo sono.
    - Questo filesystem a partizioni multiple appare all'utente come una gerarchia di directory.
    - Quando si effettua il mount di un device in una directory *mount*, tutto ciò che si trovava in quella directory non è più accessibile fino a quando non si effettua l'unmount mediante il comando *unmount*. Dopo averlo eseguito, il contenuto precedente risulta nuovamente intatto all'interno della directoy stessa.

65



GESTIONE DELLA MEMORIA

Intoduzione
Gestione della memoria fisica
Memoria virtuale
Caricamento ed esecuzione dei programmi utente

Gestione della memoria

- Ogni processo in Linux
  - è dotato di uno spazio di indirizzamento virtuale che può raggiungere i 3GB.
  - i restanti 1GB sono riservati per il kernel.
  - in user mode, lo spazio dedicato al kernel non è visibile; lo diventa non appena il processo passa in kernel mode.

67

Gestione della memoria

- Lo spazio di indirizzamento di ogni processo
  - è costituito da un insieme di regioni omogenee e contigue.
  - queste regioni sono costituite da una sequenza di pagine con le stesse proprietà di protezione e di paginazione.
  - ogni pagina ha una dimensione costante (4KB su architettura x86)

68

Gestione della memoria

- Il sistema di gestione della memoria ha due componenti :
  - il primo si occupa dell'assegnazione e del rilascio di pagine, gruppi di pagine e piccoli blocchi di memoria (memoria fisica);
  - il secondo si occupa della gestione della memoria virtuale, la memoria indirizzabile dai processi in esecuzione.

69

Gestione della memoria fisica

- Assegnatore delle pagine
  - principale strumento della gestione della memoria fisica nel nucleo di Linux.
  - responsabile dell'assegnazione e del rilascio delle pagine fisiche.
  - usa un algoritmo detto *buddy-list* per tenere traccia delle pagine fisiche disponibili.

70

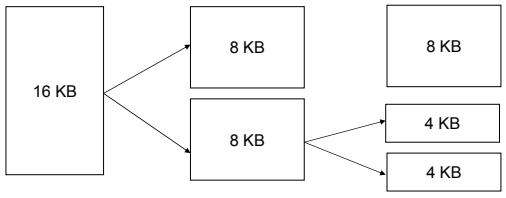
Algoritmo buddy-list

- Descrizione
  - associa coppie di unità adiacenti di memoria assegnabile
    - ogni regione di memoria assegnabile ha una compagna adiacente (buddy)
    - ogni volta che due regioni compagne assegnate si liberano entrambe, vengono combinate per formare una regione più ampia.
    - anche questa regione più ampia ha una compagna con la quale potenzialmente potrà formare una regione assegnabile ancora più ampia.
    - alternativamente se una richiesta per una piccola regione di memoria non può essere soddisfatta assegnando una regione libera esistente, una regione libera più ampia sarà suddivisa in due regioni compagne al fine di soddisfare la richiesta.

71

Esempio

- richieste di assegnazione di una regione di 4 KB;



- la più piccola regione disponibile è di 16 KB.
- la regione è ricorsivamente divisa fino ad ottenere la dimensione richiesta.

72

Algoritmo *buddy-list*

- Vantaggi
  - allocazione della memoria molto veloce (basta guardare nella lista corrispondente, o in quelle immediatamente superiori)
- Svantaggi
  - frammentazione interna ed esterna.
  - esistono dei meccanismi per "riutilizzare" la memoria sprecata dalla frammentazione interna

73

Organizzazione della memoria fisica

- Page cache
  - pagine non più utilizzate che attendono di essere rimpiazzate; gestite a parte nel caso vengano riutilizzate.
- Buffer cache
  - utilizzata per la gestione dell'I/O su dispositivi a blocchi.
- Sistema per la memoria virtuale
  - gestisce i contenuti dello spazio d'indirizzi virtuali di ciascun processo.

74

Memoria virtuale

- Il sistema per la memoria virtuale di Linux
  - si occupa dello spazio d'indirizzi visibile ad ogni processo.
  - crea pagine di memoria virtuale su richiesta.
  - gestisce il loro caricamento da disco o il loro trasferimento nell'area d'avvicendamento su disco quando è richiesto.
- Il gestore della memoria virtuale
  - considera lo spazio d'indirizzi di un processo da due punti di vista diversi: come insieme di regioni distinte e come insieme di pagine.

75

Spazio d'indirizzi di un processo

- Dal primo punto di vista :
  - riflette le istruzioni che il sistema per la memoria virtuale ha ricevuto riguardo all'organizzazione dello spazio d'indirizzi.
  - insieme di regioni non intersecantesi dove ogni regione rappresenta un sottoinsieme continuo e allineato alle pagine dello spazio d'indirizzi.
  - ogni regione è descritta da un'unica struttura dati *vm_area_struct*.
  - le regioni riguardanti un dato spazio d'indirizzi sono organizzate in una struttura ad albero binario bilanciato che permette una rapida ricerca della regione corrispondente a un indirizzo virtuale.

76

Spazio d'indirizzi di un processo

- Dal secondo punto di vista :
  - è di natura fisica.
  - realizzato grazie alle tabelle delle pagine fisiche del processo.
  - gli elementi della tabella delle pagine determinano l'esatta posizione corrente di ogni pagina della memoria virtuale.
  - la gestione della memoria è realizzata per mezzo di un insieme di procedure invocate dai gestori dei segnali di eccezione del nucleo, ogniqualvolta un processo tenta di accedere a una pagina che non è in quel momento puntata da alcun elemento della tabella delle pagine.
  - ogni struttura *vm_area_struct* nella descrizione dello spazio d'indirizzi contiene un campo che punta a una tabella di funzioni che realizzano i servizi chiave di gestione delle pagine per ogni data regione di memoria virtuale.

77

Regioni di memoria virtuale

- Linux opera con diversi tipi di regioni di memoria virtuale.
- Proprietà di un tipo di memoria virtuale (1):
  - memoria ausiliaria
    - descrive l'origine delle pagine.
    - nella maggior parte dei casi si tratta di un file o non è presente.
  - una regione priva di memoria ausiliaria è il tipo più semplice di memoria virtuale.
    - Rappresenta memoria a *valori nulli,* nel senso che quando un processo tenta di leggere una pagina di questa regione ottiene come risposta semplicemente una pagina di memoria riempita di zeri.

78

Regioni di memoria virtuale

- una regione la cui memoria ausiliaria sia un file agisce come una finestra sui contenuti di quel file:
  - quando un processo tenta di accedere a una pagina della regione, nella tabella delle pagine è scritto l'indirizzo di una pagina della cache delle pagine del nucleo corrispondente allo scostamento appropriato all'interno del file.
  - la stessa pagina di memoria fisica è usata sia dalla cache delle pagine sia dalle tabelle delle pagine del processo, cosicché ogni cambiamento apportato al file dal file system è immediatamente visibile a ogni processo che ha la locazione di quel file associata a una regione del suo spazio d'indirizzi.

Regioni di memoria virtuale

- Proprietà di un tipo di memoria virtuale (2):
  - reazione alle operazioni di scrittura.
  - la visibilità di una regione di memoria dello spazio d'indirizzi di un processo può essere *privata* o *condivisa*.
    - nel primo caso, se il processo scrive in quella regione, la procedura di paginazione rileva la necessità di una copiatura su scrittura per garantire l'effetto locale dei cambiamenti.
    - nel secondo caso, d'altra parte, l'operazione comporta l'aggiornamento dell'oggetto associato a quella regione, in modo che i cambiamenti siano immediatamente visibili a ogni altro processo che condivide la regione.

Durata di uno spazio d'indirizzi virtuale

- Il nucleo crea un nuovo spazio d'indirizzi virtuale in due situazioni:
  - avviamento di un nuovo programma tramite la chiamata del sistema *exec*.
  - creazione di un nuovo processo per mezzo della chiamata del sistema *fork*.

Durata di uno spazio d'indirizzi virtuale

- nel primo caso si assegna al nuovo programma un nuovo spazio d'indirizzi completamente vuoto; è compito delle procedure che caricano il programma di preparare lo spazio d'indirizzi con regioni di memoria virtuale.
- nel secondo caso, la creazione di un nuovo processo tramite la *fork* comporta la creazione di una copia completa dello spazio d'indirizzi virtuale esistente del processo genitore.
  - il nucleo copia i descrittori *vm_area_struct* del processo genitore, e crea poi un nuovo insieme di tabelle delle pagine per il figlio.
  - le tabelle del genitore sono copiate direttamente in quelle del figlio, incrementando solo il conteggio dei riferimenti a ogni pagina coinvolta.
  - Ne segue che dopo la *fork* i processi genitore e figlio condividono le stesse pagine di memoria fisica nei loro spazi d'indirizzi.

Durata di uno spazio d'indirizzi virtuale

- un caso particolare si ha quando durante l'operazione di copiatura s'incontra una regione di memoria virtuale privata.
  - Tutte le pagine appartenenti a questa regione sulle quali il processo ha scritto qualcosa sono private.
  - Gli eventuali successivi cambiamenti apportati dal genitore o dal figlio non devono ripercuotersi sulla pagina corrispondente nello spazio d'indirizzi dell'altro processo.
  - Durante la copiatura delle tabelle si stabilisce che le pagine in questione siano soltanto leggibili e le si contrassegna per la copiatura su scrittura.
  - Fino a che nessuno dei due processi modifica queste pagine, i due processi condividono le stesse pagine di memoria fisica, ma quando un processo tenta di scrivere in una di esse, si controlla il conteggio dei riferimenti alla pagina, e se essa è ancora condivisa il processo ne copia i contenuti in una nuova pagina di memoria fisica, usando poi questa copia in luogo dell'originale.
  - Si tratta di un meccanismo che assicura il più a lungo possibile la condivisione tra processi di pagine di dati privati; le copie si creano solo se è assolutamente necessario.

Paginazione ed avvicendamento dei processi

- uno dei compiti importanti che il sistema per la memoria virtuale deve assolvere è spostare pagine di memoria dalla memoria fisica al disco quando la memoria fisica in questione è richiesta per altri scopi.
- il sistema Linux non usa l'avvicendamento di interi processi, ma adotta esclusivamente i più recenti meccanismi di paginazione.

Paginazione ed avvicendamento dei processi

- Il sistema di paginazione si può dividere in due parti:
  - *L'algoritmo di scelta* decide quali pagine trasferire su disco, e quando farlo.
  - *Il meccanismo di paginazione* compie il trasferimento ed esegue anche l'operazione inversa non appena ve ne è bisogno.

85

Paginazione ed avvicendamento dei processi

- Algoritmo di scelta
  - prima cerca nella paging cache.
  - poi cerca nella buffer cache.
  - in entrambi i casi utili a un algoritmo simile a quello dell'orologio.
  - altrimenti cerca nelle pagine allocate ai processi
    - si cerca il processo che ha più pagine in memoria.
    - si anali ano tutte le sue aree vm_area_struct (a partire dall'ultima area analizzata nella ricerca precedente).
    - non vengono considerate le pagine condivise,utilizzate dai canali DMA, locked, assenti dalla memoria.
    - se la pagina ha il bit di riferimento acceso,questo viene spento.
    - se la pagina ha il bit di riferimento spento,viene selezionato.

86

Paginazione ed avvicendamento dei processi

- **Meccanismo di paginazione**
  - in grado di paginare sia in specifici dispositivi e in partizioni, sia in normali file, anche se quest' ultima operazione è assai più lenta a causa dei rallentamenti indotti dal file system.
  - l'assegnazione di blocchi che risiedono in dispositivi di avvicendamento è eseguita attraverso una mappa di bit dei blocchi usati che si trova sempre nella memoria fisica.
  - l'assegnatore adotta un algoritmo che tenta di scrivere le pagine secondo successioni ininterrotte di blocchi del disco al fine di migliorare le prestazioni.
  - l'assegnatore registra che una pagina è stata trasferita nel disco usando una peculiarità delle tabelle delle pagine delle moderne CPU: il bit di pagina assente dell'elemento della tabella relativo alla pagina trasferita è posto a uno, il che permette al resto dell'elemento della tabella di essere sovrascritto con un indice che identifica il luogo in cui la pagina è stata trasferita.

87

Memoria virtuale del nucleo

- Linux riserva per usi interni una regione costante e dipendente dall'architettura dello spazio d'indirizzi virtuali di ogni processo.
- Gli elementi della tabella delle pagine che si riferiscono a queste pagine del nucleo sono contrassegnati come protetti, cosicché le pagine non sono né visibili né modificabili quando la CPU è in esecuzione nel modo utente.

88

Memoria virtuale del nucleo

- quest'area di memoria virtuale del nucleo è costituita di due regioni.
  - La prima è statica e contiene riferimenti a ogni pagina di memoria fisica disponibile nel sistema, fornendo un semplice modo di tradurre indirizzi fisici in indirizzi virtuali durante l'esecuzione di codice di nucleo.
    - La parte centrale del nucleo insieme con tutte le pagine assegnate dall'ordinario assegnatore delle pagine risiedono in questa regione.

89

Memoria virtuale del nucleo

- Il resto della parte riservata al nucleo dello spazio d'indirizzi non è dedicato ad alcun scopo specifico.
  - Gli elementi della tabella delle pagine che si riferiscono a questa regione possono essere modificati dal nucleo in modo da puntare a qualunque altra area di memoria desiderata.

90

Memoria virtuale del nucleo

- Il nucleo fornisce due funzioni che permettono ai processi di usare questa memoria virtuale:
  - *vmalloc* assegna un numero arbitrario di pagine di memoria fisica e le associa a un'unica regione della memoria virtuale del nucleo, dando la possibilità di assegnare grandi sezioni di memoria contigua anche quando non vi sia sufficiente spazio fisico contiguo per soddisfare la richiesta.
  - *vremap* associa una sequenza d'indirizzi virtuali a un'area di memoria usato da un driver di dispositivo per compiere l'I/O associato alla memoria.

Caricamento ed esecuzione dei programmi utente

- L'esecuzione dei programmi utente da parte del nucleo di Linux si attiva per mezzo della chiamata del sistema *exec*.
  - Chiede al nucleo di eseguire un nuovo programma all'interno del processo corrente in modo tale da sovrascrivere integralmente il contesto d'esecuzione attuale con il contesto iniziale del nuovo programma.
- Obiettivo
  - Verificare che il processo chiamante abbia diritto d'esecuzione sul file interessato.
- Risolta tale questione, il nucleo invoca una procedura di caricamento per avviare l'esecuzione del programma
  - Il caricatore non trasferisce necessariamente i contenuti del file da eseguire nella memoria fisica, ma per lo meno associa il programma alla memoria virtuale.

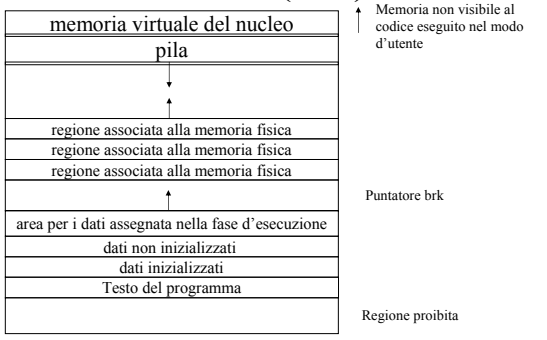
Caricamento ed esecuzione dei programmi utente

- In Linux non c'è una singola procedura per il caricamento dei programmi:
  - Il sistema operativo mantiene invece una tabella dei possibili caricatori, e dà a ognuno di essi l'opportunità di tentare di caricare il file in questione al momento dell'esecuzione di una *exec*.
- Motivazione
  - Nel passaggio dalla versione 1.0 alla versione 1.2 del nucleo il formato dei file binari di Linux è stato modificato.
    - I primi nuclei adottavano il formato *a.out*.
    - I più recenti sistemi Linux usano il più moderno formato *ELF*.
- Vantaggi di ELF
  - Flessibilità.
  - Estendibilità.

Associazione dei programmi alla memoria (ELF)

- Il caricamento di un file binario nella memoria fisica non è eseguito dal caricatore binario.
  - Le pagine del file binario sono invece associate a regioni della memoria virtuale.
  - Solo quando il programma tenterà di accedere a una data pagina, la conseguente eccezione di pagina mancante causerà il caricamento di quella pagina nella memoria fisica.
- Inizialmente
  - L'associazione del file a regioni di memoria virtuale è compito del caricatore binario del nucleo.
  - Un file in formato ELF consiste in un'intestazione seguita da diverse sezioni allineate alle pagine.
  - Il caricatore ELF lavora leggendo l'intestazione e associando le sezioni del file a regioni distinte della memoria virtuale.

Associazione dei programmi alla memoria (ELF)



Organizzazione della memoria per i programmi ELF

Associazione dei programmi alla memoria (ELF)

- Il nucleo si trova nella regione riservata a un estremo dello spazio d'indirizzi. (regione privilegiata di memoria virtuale inaccessibile agli ordinati programmi eseguiti nel modo d'utente)
- Il resto della memoria virtuale è disponibile per le applicazioni che possono usare funzioni del nucleo che creano regioni associate a porzioni di un file o disponibili per i dati delle applicazioni.
- Il compito del caricatore è di instaurare le associazioni iniziali per permettere l'avvio dell'esecuzione del programma: fra le regioni da inizializzare ci sono la pila e i segmenti di dati del programma.
- La pila è posta in cima alla memoria virtuale d'utente e cresce verso il basso impiegando indirizzi via via inferiori.
  - Include copie delle variabili d'ambiente che costituiscono gli argomenti passati al programma dalla chiamata del sistema *fork*.

Associazione dei programmi alla memoria (ELF)

- Le altre regioni sono poste vicino all'estremo inferiore della memoria virtuale.
  - Le sezioni del file binario che contengono il testo del programma o dati solamente leggibili sono assegnate a regioni protette da scrittura.
  - I dati scrivibili inizializzati sono quindi associati alla memoria virtuale, e infine i dati non inizializzati sono associati a regioni private a valori nulli.
- Appena sotto queste regioni di dimensione fissata si trova una regione a dimensione variabile che i programmi possono espandere secondo la necessità al fine di memorizzare dati assegnati nella fase d'esecuzione.
  - Ogni processo ha un puntatore *brk* che punta all'estensione attuale di questa regione di dati, e con una singola chiamata del sistema un processo può ampliare o contrarre la sua regione relativa a *brk*.
- Una volta che queste operazioni sono state completate il caricatore inizializza il contatore di programma del processo secondo il punto d'inizio registrato nell'intestazione ELF, e il processo è pronto per lo scheduling.

Collegamento statico e dinamico

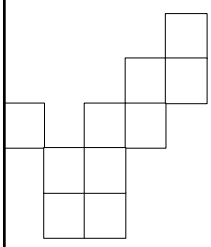
- Una volta che il programma è stato caricato e avviato, tutti i dati necessari estratti dal file binario sono stati trasferiti nello spazio d'indirizzi virtuale del processo.
  - La maggior parte dei programmi deve eseguire funzioni delle librerie di sistema, e anche queste devono essere caricate.
- **Collegamento statico**
  - Quando un programmatore scrive un'applicazione, le funzioni di libreria necessarie sono direttamente incorporate nel file binario eseguibile del programma.
  - I programmi di questo tipo possono cominciare l'esecuzione non appena siano stati caricati.
  - Svantaggi
    - Ogni programma deve contenere copie diverse delle stesse funzioni di libreria.

Collegamento statico e dinamico

- Collegamento dinamico
  - sia in termini di memoria fisica sia di spazio di disco è più efficiente caricare nella memoria le librerie di sistema una sola volta. (collegamento dinamico)
  - Linux realizza il collegamento dinamico nel modo d'utente grazie ad una speciale libreria di collegamento.
  - Ogni programma il cui collegamento alle librerie sia stato eseguito dinamicamente contiene una piccola funzione collegata staticamente che è chiamata all'avviamento del programma.
  - Questa funzione non fa altro che associare la libreria di collegamento alla memoria virtuale ed eseguire il codice contenuto nella funzione.

Collegamento statico e dinamico

- La libreria di collegamento legge l'elenco delle librerie dinamiche richieste dal programma e i nomi delle variabili e delle funzioni di libreria di cui il programma si vuole avvalere.
- Queste informazioni sono contenute nelle sezioni del file binario ELF.
- La libreria di collegamento associa poi le librerie richieste alla memoria virtuale, e risolve i riferimenti ai singoli contenuti in queste librerie.
- Non è importante l'esatto punto della memoria dove risiedono queste librerie condivise.
  - Si tratta di codice compilato in modo d'essere *codice indipendente dalla posizione* (PIC) che si può eseguire a partire da qualunque indirizzo di memoria.



I/O

Dispositivi a blocchi
Dispositivi a caratteri

I/O

- Tutti i driver dei dispositivi sono rappresentati come file ordinari.
  - L'utente apre un canale d'accesso a un dispositivo nello stesso modo in cui apre un file qualunque.
- LINUX suddivide i dispositivi in tre classi:
  - **Dispositivi a blocchi**: comprendono tutti i dispositivi capaci di fornire l'accesso diretto a blocchi di dati di dimensione fissa completamente indipendenti (dischi, dischetti, dischi ottici)
  - **Dispositivi a caratteri**: comprendono la maggior parte degli altri dispositivi ad eccezione dei dispositivi di rete e non devono necessariamente fornire le piene funzioni dei file ordinari.
  - **Dispositivi di rete**: sono trattati diversamente dagli altri dispositivi; gli utenti non possono trasferire direttamente dati a dispositivi di rete, ma devono comunicare indirettamente tramite il sottosistema di rete del nucleo.

Dispositivi a blocchi

- Costituiscono l'interfaccia principale a tutte le unità a disco del sistema.
  - Deve fornire servizi che permettano di accedere ai dischi il più rapidamente possibile.
- I due componenti principali sono:
  - *block buffer cache.*
  - *gestore delle richieste.*

103

Block buffer cache

- Funge da insieme di memorie di transito (buffer) per l'I/O in esecuzione.
- Funge da cache per l'I/O già completato.
- Buffer cache è costituita da:
  - Aree di memoria per l'I/O
    - Insieme di pagine di dimensione variabile assegnato dal gruppo di pagine della memoria centrale del nucleo.
  - Insieme corrispondente di descrittori di tali aree (*buffer_head*) uno per ogni area di memoria della cache.
- Buffer_head:
  - Contengono tutte le informazioni che il nucleo mantiene riguardo alle aree di memoria per l'I/O.
    - Identificazione: ogni area di memoria per l'I/O è identificata da una tripla (dispositivo a blocchi associato, scostamento di dati all'interno del dispositivo a blocchi, dimensione dell'area di memoria)

104

Block buffer cache

- la gestione delle aree di memoria per l'I/O del nucleo risolve automaticamente il problema della scrittura nei dischi del contenuto di tali aree di memoria che hanno subito modifiche.
- Utilizza due demoni:
  - Il primo si attiva semplicemente a intervalli regolari per chiedere che siano scritti nei dischi tutti i dati le cui modifiche, avvenute da un certo intervallo di tempo, non vi siano ancora state riportate.
  - Il secondo è un thread del nucleo che si attiva ogniqualvolta la funzione *refill_freelist(chiamata ogniqualvolta il nucleo necessita di altre aree di memoria per l'I/O)* trova che una parte troppo estesa dei dati della *buffer cache* non è ancora stata riportata nei dischi.

105

Gestore delle richieste

- Strato di programmi che gestisce la lettura e la scrittura del contenuto di un'area di memoria per l'I/O da e su un driver di dispositivo a blocchi.
- funzione *ll_rw_block*
  - Esegue le letture e le scritture a basso livello relativamente ai dispositivi a blocchi.
  - Argomenti:
    - Lista di descrittori *buffer_head*.
    - Lista di indicatori *(flag) di lettura e di scrittura.*
  - Approntа le operazioni di I/O per tutte queste aree, senza attendere il loro completamento.

106

Dispositivi a caratteri

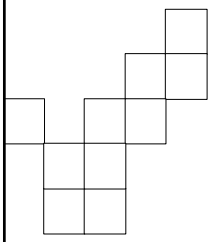
- Un driver di un dispositivo a caratteri può essere un qualsiasi driver che non offra l'accesso diretto a blocchi di dati di dimensioni fissate.
- Ogni driver di un dispositivo a caratteri registrato dal nucleo del LINUX deve anche registrare un insieme di funzioni che realizzano le operazioni di I/O su file gestite dal driver.
- Il nucleo non esegue quasi alcuna operazione preliminare su una richiesta di lettura o scrittura sul file relativa a un dispositivo a caratteri ma passa semplicemente la richiesta al dispositivo in questione lasciandogli il compito di servirla.

107

Dispositivi a caratteri

- l'eccezione principale a questa regola è costituita dai driver dei dispositivi a carattere relativi ai terminali
  - il nucleo mantiene un'interfaccia uniforme a questi driver per mezzo di un insieme di strutture *tty_struct:*
    - Ognuna di esse fornisce il controllo del flusso e la memorizzazione transitoria dei dati provenienti dal terminale e passa questi dati ad un interprete.
  - L'interprete delle informazioni provenienti da un terminale segue una determinata disciplina, la più comune delle quali è la *tty*, che incolla il flusso dei dati del terminale ai flussi di I/O standard dei processi utenti in esecuzione, permettendo a questi processi di comunicare direttamente con il terminale dell'utente.

108



Strutture di rete

Interfaccia a socket
Driver dei protocolli
Driver dei dispositivi di rete

Strutture di rete

- Punti di forza del sistema operativo LINUX
  - Gestisce i protocolli standard della rete Internet per la comunicazione fra sistemi UNIX.
- Il nucleo di LINUX realizza i servizi di rete per mezzo di tre strati di programmi:
  - L'interfaccia a socket.
  - I driver dei protocolli.
  - I driver dei dispositivi di rete.

110

L'interfaccia a socket

- Socket:
  - Estremità di un canale di comunicazione.
  - Una coppia di processi che comunica attraverso una rete usa una coppia di socket.
    - Una per ogni processo
    - Ogni socket è identificata da un indirizzo IP concatenato a un numero di porta.
  - In generale le socket impiegano un'architettura client-server:
    - Il server attende le richieste dei client, stando in ascolto a una porta specificata.
    - Quando il server riceve una richiesta, se accetta la connessione proveniente dalla socket del client, si stabilisce la comunicazione

111

I driver dei protocolli

- Si richiede che tutti i dati che arrivino a questo livello siano etichettati da un identificatore che specifichi secondo quale protocollo devono essere trattati.
- Lo strato dei protocolli può riscrivere i pacchetti, crearne di nuovi, dividerli in frammenti o riassemblarli da frammenti, o anche semplicemente scartare i dati in arrivo.
- Quando ha terminato l'elaborazione di un gruppo di pacchetti, li passa all'interfaccia socket se la destinazione dei dati è locale, oppure a un driver di un dispositivo di rete, se i pacchetti devono essere inviati lungo la rete.
- Lo strato di protocolli decide a quale socket o dispositivo inviare il pacchetto.

112

Strutture di rete

- Tutta la comunicazione che avviene fra gli strati di programmi che realizzano i servizi di rete è eseguita passando singole strutture dette *skbuff*.
  - Contengono un insieme di puntatori ad un'unica regione contigua di memoria all'interno della quale i pacchetti di rete possono essere assemblati.
- L'insieme dei protocolli più importante in LINUX è la serie di protocolli IP, composta di un certo numero di protocolli distinti.
  - Il protocollo IP realizza l'instradamento da un calcolatore ad un altro nella rete.

113

Strutture di rete

- Su questo protocollo d'instradamento sono costruiti i protocolli:
  - UDP: trasferisce i singoli datagram arbitrari fra i calcolatori.
  - TCP: instaura connessioni affidabili con consegna garantita nell'ordine originario, e ritrasmissione automatica dei dati persi.
  - ICMP: si usa per la trasmissione di messaggi di stato e di vari tipi di messaggi d'errori.

114

Driver dei dispositivi di rete

- Si assume che i pacchetti (skbuff) che raggiungono la pila di protocolli siano già etichettati da un identificatore interno che indica a quale protocollo è attinente il pacchetto.
- Driver dei dispositivi di rete diversi codificano il tipo di protocollo in modo differente sul loro mezzo di trasmissione.
  - L'identificazione del protocollo va eseguita all'interno dei driver stessi, i quali usano per questo fine una tabella hash degli identificatori di protocollo noti e passano poi il pacchetto al protocollo appropriato.

115

Driver dei dispositivi di rete

- I pacchetti IP che giungono al sistema sono consegnati al driver IP, il cui compito è quello di eseguire l'instradamento:
  - Determina la destinazione del pacchetto e lo inoltra al driver del protocollo interno appropriato per la consegna locale;
  - Oppure lo reinserisce nella coda del driver di un dispositivo di rete se deve essere inoltrato a un altro calcolatore.
- La decisione riguardante l'instradamento viene presa sulla base di due tabelle:
  - FIB(forwading information base)
    - Contiene informazioni sulle configurazioni d'instradamento e può specificare percorsi basati su indirizzi di destinazione determinati o su parametri che rappresentano più destinazioni.
    - È organizzata come un insieme ti tabelle hash indicizzate dagli indirizzi di destinazione
    - La ricerca parte sempre dalle tabelle che contengono i percorsi più precisi.
    - Quando una ricerca di questo tipo ha successo, il percorso individuato è posto nella cache dei percorsi, la quale contiene solo destinazioni specifiche prive di parametri, in modo che le ricerche siano più rapide.
  - cache delle più recenti decisioni d'instradamento.

116

Driver dei dispositivi di rete

- In diverse fasi il protocollo IP passa i pacchetti a una sezione distinta di codice per la **gestione di barriere di sicurezza**, cioè il filtraggio selettivo dei pacchetti secondo criteri arbitrari ma di solito relative alle strategie di sicurezza.
  - Mantiene un certo numero di **catene di barriere di sicurezza** distinte.
  - Permette la comparazione di una struttura *skbuff* con una qualunque di esse
  - Catene distinte assolvono funzioni distinte:
    - Una si usa per inoltrare i pacchetti;
    - Una per la ricezione dei pacchetti;
    - Una per i dati generati dal calcolatore.

117

Driver dei dispositivi di rete

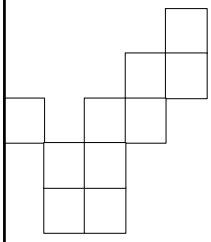
- Ulteriori funzioni del driver IP:
  - Scomposizione dei pacchetti
    - Un pacchetto da spedire troppo grande per essere posto nella coda di un dispositivo viene semplicemente diviso in **frammenti** più piccoli che possono essere posti in coda.
  - Riassemblaggio dei pacchetti di grandi dimensioni
    - Il calcolatore ricevente si occuperà di riassemblare i frammenti.
    - Il driver IP mantiene un oggetto *ipfrag* per ogni frammento che attende il riassemblaggio ed un oggetto *ipq* per ogni datagram in corso d'assemblaggio.
    - I frammenti in arrivo sono confrontati con ogni *ipq* e nel caso di riscontro positivo il frammento è aggiunto all'oggetto, altrimenti si crea un nuovo *ipq*.
    - Quando l'ultimo frammento di un *ipq* è arrivato si costruisce una struttura *skbuff* interamente nuova per alloggiare il pacchetto e si passa di nuovo il pacchetto al driver IP.

118

Driver dei dispositivi di rete

- I pacchetti che l'IP identifica come destinati al calcolatore locale sono passati a uno degli altri driver di protocollo.
  - I protocolli TCP e UDP adottano lo stesso metodo per associare ogni pacchetto alle relative socket mittenti e destinatarie:
    - Ogni coppia di socket collegate è identificata in modo unico dagli indirizzi del mittente e del destinatario e dai corrispondenti numeri di porta.
  - Il protocollo TCP deve anche occuparsi delle connessioni non affidabili; a tal fine mantiene:
    - Liste ordinate dei pacchetti trasmessi, ma privi di una ricevuta di ritorno, che saranno ritrasmessi dopo un certo tempo.
    - Liste dei pacchetti ricevuti in modo disordinato, che saranno presentati alla socket una volta ricevuti i dati mancanti.

119



SICUREZZA

Autenticazione
Controllo degli accessi

Introduzione

- Problemi
  - **Autenticazione**
    - Assicurare che nessuno possa accedere al sistema senza prima dimostrare di averne diritto.
  - **Controllo dell'accesso**
    - Fornire un meccanismo che permetta di controllare se un utente abbia diritto d'accesso a un certo oggetto, e che impedisca l'accesso se l'esito del controllo è negativo.

121

Autenticazione

- Basato su un file di parole d'ordine leggibile da tutti.
  - la parola d'ordine di un utente è combinata con un valore casuale.
  - il risultato è codificato tramite una funzione di codifica non invertibile.
  - infine registrato nel file delle parole d'ordine.
- Quando un utente presenta una parola d'ordine al sistema
  - Essa viene ricombinata con il valore casuale memorizzato nel file delle parole d'ordine.
  - Al risultato si applica la funzione di codifica non invertibile.
  - Se ciò che si ottiene coincide con il contenuto del file delle parole d'ordine l'utente è ammesso al sistema.

122

Autenticazione

- Problema associato a tale meccanismo:
  - Le parole d'ordine spesso limitate alla lunghezza di otto caratteri.
  - Numero dei possibili valori casuali così basso che si potevano facilmente combinare le parole d'ordine più comuni con ogni possibile valore casuale e avere buone probabilità d'individuare una o più parole d'ordine contenute nel file delle parole d'ordine, ottenendo così l'accesso non autorizzato alle utenze corrispondenti.

123

Autenticazione

- Soluzioni proposte
  - Estensioni del meccanismo delle parole d'ordine al fine di mantenerle segrete
    - Memorizzazione di tali parole in un file non accessibile al pubblico.
  - Utilizzo di parole d'ordine più lunghe.
  - Adozione di metodi codifica più sicuri.
  - Introduzione di meccanismi di autenticazione che limitano il tempo di collegamento al sistema.
  - Ecc …

124

Autenticazione

- Il sistema PAM (*pluggable authentication modules*)
  - Basato su una libreria condivisa che può essere usata da ogni componente del sistema che abbia bisogno di autenticare utenti.
  - Permette di caricare su richiesta moduli di autenticazione secondo le indicazione contenute in un file di configurazione valido per tutto il sistema.
  - Un nuovo meccanismo di autenticazione aggiunto in seguito si può registrare nel file di configurazione in modo che tutti i componenti del sistema possano immediatamente usufruirne.

125

Autenticazione

- I sistemi PAM sono in grado di specificare
  - Metodi di autenticazione.
  - Limitazioni alle attività degli utenti.
  - Funzioni di avvio di una sessione di lavoro
  - Funzioni di cambio di una parola d'ordine.
    - Quando un utente cambia la propria parola d'ordine, tutti i meccanismi necessari di autenticazione possono essere aggiornati in un'unica soluzione.

126

Controllo degli accessi

- Realizzato per mezzo d'identificatori numerici unici
  - Identificatore d'utente *(uid)* individua un singolo utente o un singolo insieme di diritti d'accesso.
  - Identificatore di gruppo *(gid) aggiuntivo* che si può usare per determinare i diritti di più utenti.
- Si applica a vari oggetti del sistema
  - Ogni file disponibile nel sistema è protetto dal meccanismo standard del controllo degli accessi
  - Lo stesso vale per le regioni di memoria condivisa e i semafori.

127

Controllo degli accessi

- Ogni oggetto in un sistema UNIX sottoposto al controllo degli accessi di singoli utenti o gruppi
  - Dispone di un unico uid associato.
  - Dispone di un unico gid associato.
- Anche i processi utenti hanno un unico uid, ma possono avere più gid.
  - Se l'uid di un processo coincide con l'uid di un oggetto, quel processo gode dei **diritti d'utente** o dei **diritti di proprietà** di quell'oggetto.
  - Se gli uid non corrispondono ma uno dei gid di un processo corrisponde al gid di un oggetto, il processo gode dei **diritti di gruppo** su quell'oggetto.
  - altrimenti il processo ha i **diritti generici** sull'oggetto.

128

Controllo degli accessi

- Eseguito dal sistema Linux assegnando agli oggetti una **maschera di protezione:**
  - Specifica quali modi d'accesso (lettura, scrittura o esecuzione) si devono concedere ai processi con diritto di accesso proprietari, di gruppo o generici.
- **Eccezione**
  - uid privilegiato **root**
    - un processo dotato di questo speciale uid gode automaticamente dei diritti d'accesso a qualunque oggetto del sistema.
    - meccanismo che permette al nucleo di impedire agli utenti ordinari di accedere a determinate risorse del sistema.

129

Controllo degli accessi

- Setuid
  - Meccanismo standard che permette ad un programma di godere, durante una certa esecuzione, di privilegi diversi da quelli dell'utente che esegue il programma.
    - ad esempio, il programma *lpr* che pone un file in coda di stampa ha accesso alle code di stampa del sistema, anche se l'utente che ne richiede l'esecuzione non lo ha.
  - La realizzazione distingue fra uid reale e uid effettivo di un processo
    - Il primo è quello dell'utente che richiede l'esecuzione del programma.
    - Il secondo è quello del proprietario del file.

130

Controllo degli accessi

- saved user-id
  - Permette ad un processo di perdere e riacquistare ripetutamente il suo uid effettivo.
    - Le versioni standard dello UNIX riescono a fornire questo servizio solo scambiando gli uid reali ed effettivi
      - Lo uid precedente è ricordato, ma lo uid reale del programma non sempre corrisponde allo uid dell'utente che ne richiede l'esecuzione.
  - Permette ad un processo di rendere il suo uid effettivo uguale al suo uid reale e ritornare poi al valore precedente del suo uid effettivo senza dover mai modificare il valore dello uid reale.

131

Controllo degli accessi

- Miglioramenti
  - Aggiunta di una caratteristica dei processi che permette di usufruire di un sottoinsieme dei diritti conferiti dallo uid effettivo:
    - Le proprietà *fsuid* e *fsgid* di un processo sono usate quando è consentito l'accesso ad un file, e sono attive ogniqualvolta lo uid effettivo o il gid lo sono.
      - *fsuid* e *fsgid* possono essere attivate indipendentemente dagli identificatori effettivi, cosa che permette ad un processo di accedere al file per conto di un altro utente senza dover assumere in alcun senso l'identità di quell'utente.

132

Controllo degli accessi

- LINUX offre un altro metodo per il passaggio flessibile dei diritti da un programma a un altro.
  - Una volta che un collegamento fra due processi del sistema sia stato istituito per mezzo di una socket locale, ognuno dei due processi può inviare all'altro un descrittore di file relativo a uno dei suoi file aperti.
  - L'altro processo riceve quindi un descrittore duplicato dello stesso file.
  - Ciò permette a un client di offrire a un processo server l'accesso selettivo a un solo file senza conferirgli alcun altro privilegio.

133