Sistemi di Elaborazione

Sistemi Operativi

Gianluca Amato
Dipartimento di Scienze
Ufficio n. 12
Tel: 085-4546425
Email: amato@sci.unich.it
`http://www.sci.unich.it/~amato`

Ricevimento: Giovedì 16-18

Presentazione del corso

Obiettivi del corso

Gli obiettivi del corso sono la conoscenza dei sistemi di elaborazione dal punto di vista hardware (architetture di elaboratori) e software (sistemi operativi).

Ricevimento: Giovedì 16.00-18.00 presso il mio studio, Ufficio n. 12 del Dipartimento di Scienze, nuovo edificio.

Programma d'esame

Nella prima parte del corso saranno illustrati gli aspetti architetturali fondamentali dei moderni calcolatori elettronici. Nella seconda parte del corso saranno introdotti i concetti fondamentali dei moderni sistemi operativi, con esempi dedicati in particolare ai sistemi operativi Unix/Linux e Windows.

Architetture degli elaboratori elettronici (Andrea Roli)

- Classificazione dei calcolatori elettronici. Progresso tecnologico e architetturale.
- Architettura di Von Neumann: CPU, memoria, I/O. Fasi di esecuzione delle istruzioni.
- Architettura della CPU. CISC vs. RISC.
- Gestione delle interruzioni.
- Tipologie di memorie e loro caratteristiche. Gerarchie di memorie.
- La memoria virtuale.
- Cenni sulle caratteristiche delle architetture avanzate.

Sistemi operativi (Gianluca Amato)

- Introduzione ai sistemi operativi: ruolo, funzionalità e concetti di base. Storia dei sistemi operativi: sistemi batch, multiprogrammazione, time-sharing, l'avvento dei personal computer. Classificazione dei sistemi operativi: per server, per personal computer, real-time, embedded. Struttura di un sistema operativo: sistemi monolitici e stratificati, macchine virtuali, modello client-server. Chiamate di sistema.

- Il concetto di processo e sua rappresentazione nel sistema operativo: processi, stati di un processo, gerarchie di processi, thread. Comunicazione tra processi: sezioni critiche, mutua esclusione, semafori, mutex, scambio di messaggi. Problemi classici di comunicazione tra processi: filosofi a cena, lettori/scrittori, barbiere sonnolento, scambio di messaggio su buffer circolare. Schedulazione: schedulazione nei sistemi batch, interattivi e real-time.
- Gestione delle risorse: il concetto di risorsa, deadlock, l'algoritmo dello struzzo, metodi per identificare, risolvere, evitare o prevenire i deadlock.
- Gestione dell'I/O: I/O mappato in memoria, I/O programmato, I/O con interruzioni, I/O con DMA. I dischi magnetici: struttura di un disco fisso, formattazione di un disco, i sistemi RAID.
- File system: i file, le directory, sistemi gerarchici. Implementazione dei file system: allocazione contigua, lista di allocazione, i-node, implementazione delle directory. Gestione dello spazio su disco.
- Sicurezza: minacce, intrusi e perdita accidentale dei dati. Principi di crittografia: crittografia a chiave simmetrica e a chiave pubblica, firma digitale. Autenticazione dell'utente. Attacchi dall'interno del sistema: cavallo di troia, login spoofing, bombe logiche, backdoor, buffer overflow. Attacchi dall'esterno: i virus. Meccanismi di protezione: liste di controllo degli accessi e capability.

Testi di riferimento

- A.Tanenbaum. Architettura dei computer : un approccio strutturato. UTET libreria, 2000;
- A.Tanenbaum. I Moderni Sistemi Operativi (II edizione). Jackson, 2002.

Oppure in lingua originale:

- A.Tanenbaum. Structured Computer Organization (4th edition). Prentice Hall.
- A.Tanenbaum. Modern Operating Systems (II edition). Prentice Hall.

I testi consigliati saranno utilizzati solo parzialmente, in quanto coprono argomenti piu' ampi rispetto a quelli trattati nel corso.

Sistemi operativi: programma dettagliato

I riferimenti che seguono sono relativi al libro di A. Tanenbaum "I Moderni Sistemi Operativi".

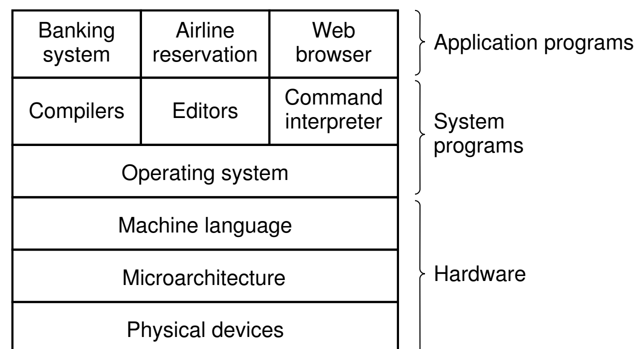
- capitolo 1: tutto escluso la sezione 1.8. La sezione 1.4, benchè non trattata a lezione, è un utile riassunto della prima parte del corso sulle architetture dei sistemi di elaborazione. Della sezione 1.6 non si richiede la conoscenza dei nomi delle varie system call: una conoscenza generale delle loro funzioni è sufficiente.
- capitolo 2: sezioni 2.1, 2.2 (fino alla sottosezione 2.2.1 inclusa), 2.3 (escluso la soluzione di Peterson e le sottosezioni 2.3.7 e 2.3.9), 2.4, 2.5 (escluso Schedulazione a tre livelli, "Code multiple", "Shortest Process Next", "Schedulazione garantita", "Schedulazione a lotteria", "Schedulazione fair-share", sottosezione 2.5.6), 2.7.
- capitolo 3: tutto tranne sezioni 3.4.2, 3.5.2, 3.5.3, 3.5.4, 3.7, 3.8.
- capitolo 5: sezioni 5.1 (escluso 5.1.5), 5.2, 5.3 (molto superficialmente), 5.4.1 (fino alla sottosezione riguardante i CDROM esclusa).
- capitolo 6: sezioni 6.1 (fino a 6.1.7 incluso), 6.2, 6.3 (escluso 6.3.6, 6.3.8).
- capitolo 9: sezioni 9.1, 9.2, 9.3 (fino alla sottosezione "Come penatrano i cracker" esclusa), 9.3.1 sottosezione "Migliorare la sicurezza delle password", 9.3.2, 9.3.3, 9.4 (fino a 9.4.5 incluso), 9.5 (fino a 9.5.4 incluso), 9.6.

Segue una traccia molto dettagliata del contenuto del corso. **Questa traccia non è una dispensa sulla quale studiare**, in quanto non è per nulla esaustiva. Può essere usata come guida generale per capire cosa studiare, ma non può sostituire il libro di testo e le lezioni.

Capitolo 1

Introduzione

Un sistema di elaborazione è composto da vari dispositivi. Gestirli in maniera ottimale è difficile: servono i **sistemi operativi**.



Cosa fa parte del sistema operativo? Di solito si parla di S.O. per la parte che gira in modo **supervisore**. Ma non sempre è così: per essere sicuri, si parla di **nucleo**.

Due modi di vedere un sistemi operativo:

- **macchina virtuale**: nasconde le complicazioni dell'hardware. Fornisce un insieme di servizi che un programma può utilizzare tramite **chiamate di sistema**.
- **gestore risorse**: assegna le risorse a chi le richiede, condivisione in **tempo** (CPU) e in **spazio** (memoria, disco).

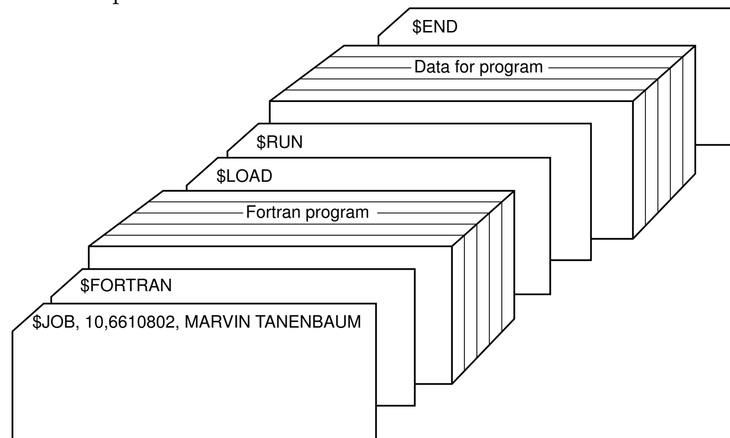
1.1 Storia dei sistemi operativi

Parallela alla storia dei sistemi di elaborazione:

- **prima generazione**, 1945-55
  - computer costruiti con relay e valvole: poco affidabili
  - il progettista, programmatore e utente sono la stessa persona

- programmazione in linguaggio macchina, tramite pannelli con contatti fisici prima e schede perforate dopo
- quasi nessun sistema operativo: solo driver dispositivi e bootstrap

- **seconda generazione**, 1955-65
  - nasce il transistor: affidabili, possono essere venduti a terzi: mainframes
  - progettista, programmatore ed operatore sono figure diverse
  - programmazione in assembler o fortran tramite schede perforate
  - sistemi operativi **batch**



  - un "monitor" controlla l'avvicendamento dei job

- **terza generazione**, 1965-80
  - circuiti integrati: diffusione negli ambienti commerciali, **minicomputer**.
  - **spooling** (prima) e **multiprogrammazione** (poi) per ridurre gli sprechi di tempo nelle operazioni di I/O
  - **timesharing** per avere più utenti contemporaneamente: MULTICS

- **quarta generazione**, 1980-
  - circuiti integrati LSI (large scale integration): microcomputer
  - interfacce utente grafiche (GUI)
  - gestione di reti di elaboratori (network operating systems)

1.2 I sistemi operativi ai giorni nostri

- sistemi operativi per **mainframe**: IBM OS/390
  - gestione di più processori **fortemente accoppiati**
  - come personal computer ma con immense capacità in termini di I/O (migliaia di dischi);

  - elaborazioni batch, timesharing e **transazioni**.

- sistemi operativi per **server**: Unix/Linux, Windows NT/2000
  - girano su personal computer ma anche su mainframe
  - gestione di più processori **fortemente accoppiati**
  - compiti primari: file server, print server, web server
  - elaborazioni batch, timesharing
- sistemi operativi per personal computer: Linux, Windows 98/XP
  - interfaccia utente
- sistemi operativi real-time:
  - requisiti temporali che devono essere rispettati (conflitto con time-sharing)
  - **hard real-time** vs **soft real-time** (QNX)
- sistemi operativi distribuiti
  - distribuzione della computazione su più macchine **debolmente accoppiate**
  - visione unitaria del sistema di calcolo
  - tolleranza ai guasti, condivisione risorse, aumento della velocità
- sistemi operativi per palmtop (PalmOS, Windows CE)
- sistemi operativi embedded
- sistemi operativi per smart-card

1.3 Concetti di Base dei Sistemi Operativi

Processo: un programma in esecuzione

- timesharing: necessità di cambiare da un processo a un altro
- **address space**
- stati interni registri e file aperti: **process table**
- creazione e terminazione processi: **process tree**
- comunicazione tra processi correlati: sincrona e asincrona (**segnali**)
- protezione: **UID**

Scheduler: parte del S.O. che si occupa di passare da un processo ad un altro.

- attivato da interrupt periodici
- politiche di scheduling

Deadloc

- esempio: incrocio a 4 vie
- esempio concreto: 2 processi e 2 risorse

Memoria

- primo aspetto: proteggere la memoria dei processi
- secondo aspetto: **memoria virtuale**

Input/Output

- parte dipende dal dispositivo (device driver) parte no (API)
- ottimizzare le prestazioni: **cache**

File

- file come comodo metodo per condividere spazio disco
- organizzare i file con l'uso di directory: **directory hierarchy**
- operazioni sui file: apertura, creazione, scrittura, lettura, ...
- operazioni su directory

Sicurezza

- impedire l'accesso a informazioni confidenziali
- impedire il malfunzionamento del sistema (ad esempio virus)

Shell

- formalmente non è parte del S.O.
- esempio di sessione
- unix e windows

Chiamate di sistema

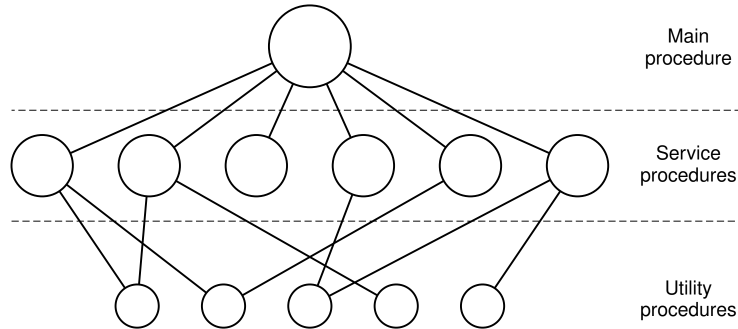
- istruzione specifica per le chiamate di sistema
- chiamate di sistema da linguaggi ad alto livello

```
count = read(fd,buffer,nbytes)
```

1.4 Struttura dei Sistemi Operativi

- sistemi monolitici

- sistemi monolitici con stratificazione interna



- sistemi stratificati: THE

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

e anche MULTICS

- macchine virtuali
  - VM/370 per aggiungere il time-sharing a OS/360
  - emulazione DOS
  - vmWare
  - JVM
- sistemi client-server
  - microkernel: MACH
  - importanza dei meccanismi per la comunicazione
  - Hurd, Mac OS/X
  - meccanismi per l'accesso hardware ai processi
  - separazione dei meccanismi dalla politica

1.5 Riciclaggio dei concetti

- dipendenza dalla tecnologia
- importanza dello studio di sistemi "obsoleti"
- esempio: allocazione contigua su CDROM, terminali grafici

Capitolo 2

Processi

Il modello dei processi sequenziali nasce per trattare in maniera semplice il parallelismo.

2.1 Processi

Un processo è un programma in esecuzione. La CPU passa da uno all'altro salvando tutti i dati necessari per riprenderne l'esecuzione

Esempio del cuoco.

2.1.1 Creazione Processo

A regime in un sistema ci sono vari processi: come si creano?

- inizializzazione: **demoni**
- un processo ne crea un altro
- un utente richiede di creare un altro processo
- esecuzione di un job

In tutti i casi, la creazione avviene con una system call: **fork** e **CreateProcess**.

2.1.2 Terminazione Processo

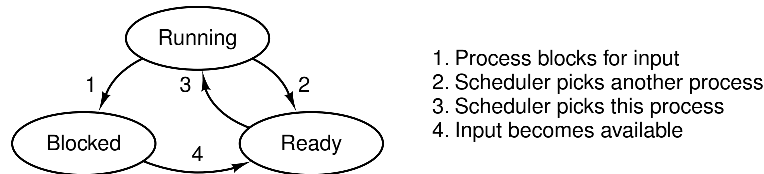
- uscite normali (volontarie): **exit** e **ExitProcess**
- uscite a causa di errore (volontarie)
- uscite a causa di errori fatali (involontarie)
- richieste di terminazione dall'esterno (involontario): **kill** e **KillProcess**

2.1.3 Gerarchia di Processi

I processi formano una gerarchia padre-figlio. In Unix la gerarchia ha un significato semantico (**wait**), in Windows no.

2.1.4 Cambiamenti di Stato

Due motivi diversi per cui un processo in esecuzione è interrotto: scadenza del tempo concesso o attesa di input.



2.1.5 Implementazione dei Processi

tabella dei processi con una riga per ogni processo contenente PC, stack, spazio indirizzi, file aperti, etc...

La gestione del parallelismo avviene con gli interrupt: **interrupt vector**, modifica di stato e **scheduler**.

2.1.6 Thread

Talvolta serve un flusso di esecuzione multiplo sullo stesso spazio degli indirizzi. Un processo, in effetti, fa due funzioni:

- raggruppa risorse: spazi indirizzi, file, gestori segnali;
- rappresenta un flusso di esecuzione: stato CPU, stack

Separando il secondo dal primo si ottengono i **thread**.

2.2 Comunicazione tra Processi

I processi devono comunicare. Per esempio in una pipe:

```
ls | sort
```

Le problematiche che nascono sono

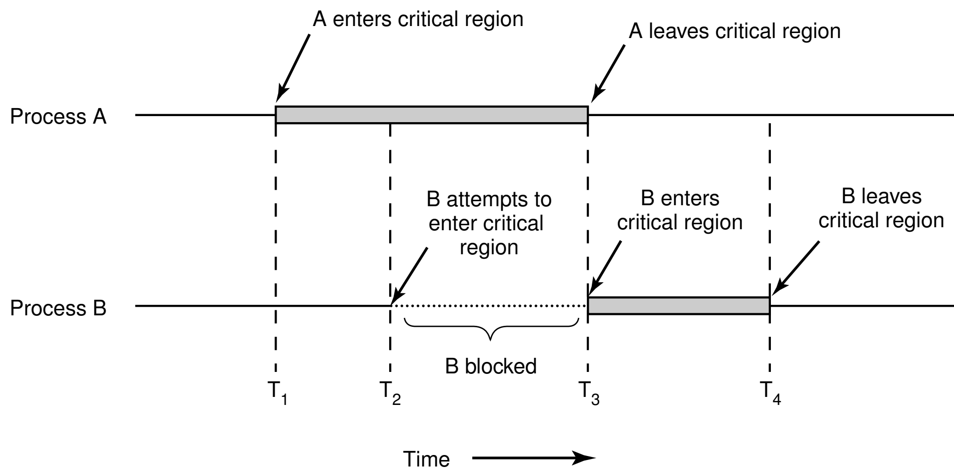
- il meccanismo di comunicazione
- i conflitti
- la sincronizzazione

2.2.1 Condivisione e Mutua Esclusione

Esempi di meccanismi di comunicazione: memoria condivisa e file.

Una corsa critica si verifica quando si ha accesso a dati condivisi: esempio dello **spooler**. Bisogna proibire l'accesso in contemporanea ai dati condivisi: **mutua esclusione**.

La zone di un programma dove si accede a dati condivisi è detta **regione critica**. Bisogna impedire che due processi siano contemporaneamente nella zone critica. Nessuna assunzione sulla velocità della macchina.



Alcuni meccanismi per la mutua esclusione

- disabilitare gli **interrupt**: utile per il S.O., ma poco sicuro se fatto dagli utenti.
- variabile di blocco (0 libero, 1 occupato) non funziona in condizioni normali.

```
enter_region:
   cmp LOCK,0
   jne enter_region
   mov LOCK,1

exit_region:
   mov LOCK,0
```

- variabile di blocco con istruzione speciale TSL

```
TSL RX,LOCK
```

Nessun altro processore può accedere alla memoria.

```
enter_region:
   TSL RX,LOCK
   cmp RX,0
   jnz enter_region

exit_region:
   mov LOCK,0
```

Purtroppo c'è attesa attiva! Spreco di CPU e problemi con processi a priorità.

- mutex: lock + coda. È una chiamata di sistema

```
lock/wait(LOCK)
<sezione critica>
unlock/signal(LOCK)
```

LOCK è una struttura interna del S.O. che contiene lo stato una lista di processi in attesa.

2.2.2 Sincronizzazione

Talvolta i meccanismi di mutua esclusione non sono sufficienti, e sono necessari meccanismi di sincronizzazione più complessi: il problema del **produttore-consumatore**.

Supponiamo che due processi condividono un buffer comune di lunghezza fissata. Uno processo scrive, l'altro legge. Soluzione sbagliata con primitive **sleep** e **wakeup**.

```
#define N 100                              /* number of slots in the buffer */
int count = 0;                             /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {                         /* repeat forever */
        item = produce_item( );            /* generate next item */
        if (count == N) sleep( );          /* if buffer is full, go to sleep */
        insert_item(item);                 /* put item in buffer */
        count = count + 1;                 /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);  /* was buffer empty? */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                         /* repeat forever */
        if (count == 0) sleep( );          /* if buffer is empty, got to sleep */
        item = remove_item( );             /* take item out of buffer */
        count = count – 1;                 /* decrement count of items in buffer */
        if (count == N – 1) wakeup(producer); /* was buffer full? */
        consume_item(item);                /* print item */
    }
}
```

Anche aggiungere la mutua esclusione su count non aiuta. Ecco allora i semafori di Dijkstra (1965). Operazione **down** e **up** come lock e unlock ma ora il semaforo ha un valore intero!

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                     /* TRUE is the constant 1 */
          item = produce_item( );       /* generate something to put in buffer */
          down(&empty);                 /* decrement empty count */
          down(&mutex);                 /* enter critical region */
          insert_item(item);            /* put new item in buffer */
          up(&mutex);                   /* leave critical region */
          up(&full);                    /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                     /* infinite loop */
          down(&full);                  /* decrement full count */
          down(&mutex);                 /* enter critical region */
          item = remove_item( );        /* take item from buffer */
          up(&mutex);                   /* leave critical region */
          up(&empty);                   /* increment count of empty slots */
          consume_item(item);           /* do something with the item */
     }
}
```

I semafori sono difficili da utilizzare: si pensi se si inverte il mutex con il semaforo empty! C'è una sorta di conflitto tra la mutua esclusione e la sincronizzazione.

2.2.3 Messaggi

La condivisione non funziona su macchine distribuite. Occorre ricorrere al passaggio di messaggi. Basato su due chiamate di sistema:

```
send(destination,message);
receive(source,message);
```

Implementare i messaggi può essere più complicato dei semafori:

- perdita di messaggi sulla rete: **acknowledgment**
- perdita dell'acknowledgment: **id dei messaggi**
- metodo per identificare i processi

Classificazione dei sistemi di comunicazione

- bloccante e non-bloccante
- sincrona e asincrona (receive bloccante e send bloccante/non bloccante)
- mailbox

Vediamo il modello produttore-consumatore con messaggi

```
#define N 100                          /* number of slots in the buffer */

void producer(void)
{
      int item;
      message m;                       /* message buffer */

      while (TRUE) {
            item = produce_item( );    /* generate something to put in buffer */
            receive(consumer, &m);     /* wait for an empty to arrive */
            build_message(&m, item);   /* construct a message to send */
            send(consumer, &m);        /* send item to consumer */
      }
}

void consumer(void)
{
      int item, i;
      message m;

      for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
      while (TRUE) {
            receive(producer, &m);     /* get message containing item */
            item = extract_item(&m);   /* extract item from message */
            send(producer, &m);        /* send back empty reply */
            consume_item(item);        /* do something with the item */
      }
}
```

2.3 Scheduler

Lo **scheduler** si occupa di scegliere, tra i processi in stato di pronto, quale portare in esecuzione. Ma quando vanno prese decisioni sullo scheduling dei processi?

- quando un processo viene creato: va eseguito il padre o il figlio?
- quando un processo termina
- quando un processo si blocca su un I/O o su un semaforo
- quando arriva un interrupt di I/O
- quando arriva un interrupt periodico: scheduling **preempitivo**.

La politica che adotta lo scheduler dipende dal tipo di sistema. In generale si cerca

- massima utilizzazione delle risorse;
- bilanciamento tra i vari processi.

tenendo conto che il context-switching è costoso. Più in particolare:

- **sistemi batch**: preemption non fondamentale
  - throughput: massimizzare il numero di job per ora
  - turnaround time: minimizzare il tempo tra sottomissione e terminazione dei job

sono talvolta in contrasto

- **sistemi interattivi**: fondamentale la preemption
  - tempo di risposta
  - proporzionalità: soddisfare le aspettative dell'utente

2.3.1 Scheduling in Sistemi Batch

First Come First Served

Una lista per i processi pronti, eseguiti in ordine di arrivo. Un processo rimane in esecuzione finchè vuole (no-preemption) e non fa operazioni di I/O.

Problema: un processo compute-bound che esegue per un secondo alla volta e tanto I/O bound che usano poca cpu ma devono eseguire 1000 letture da disco. Ogni processo I/O bound legge 1 blocco al secondo e ci mette 1000 secondi per terminare!! Se ci fosse la preemption ogni 10 msec, ci impiegherebbe solo 10 sec.

In generale è meglio schedulare il prima possibile i processi I/O bound.

Shortest Job First

Supponiamo che abbiamo una idea di quanto impiega un job a terminare. Scegliamo tra i job pronti quello che termina prima.

SJF è ottimale, quando i job arrivano contemporaneamente!

2.3.2 Scheduling in Sistemi Interattivi

Round Robin

A ogni processo è associato un **quanto** di tempo. Un processo viene eseguito per questo quanto, a meno che non si blocca prima. Semplice da implementare, basta una lista.

Problema: che valore scegliere per il quanto?

Scheduling con Priorità

A ogni processo è assegnata una priorità, e il processo con più alta priorità è sempre quello in esecuzione. Un demone che manda posta in background avrà priorità più bassa di un processo che visualizza un video.

- come evitare che un processo ad alta priorità consumi tutto il tempo macchina? Diminuire periodicamente la priorità
- allocazione delle priorità: statica / dinamica
- priorità dinamica per processi I/O: 1/f dove f è la frazione dell'ultimo quanto usato.

- come schedulare processi della stessa priorità?

2.3.3 Scheduling in Sistemi Real-Time

In un S.O. real-time, lo scheduling è gestito da eventi. Ad ogni "evento" (temporizzatore, dispositivo) viene eseguito un processo che è molto piccolo. Gli eventi si distinguono in **periodici** e **aperiodici**.

Occupiamo solo di quelli periodici. Supponiamo m eventi periodici e l'evento i occorre ogni P_i secondi e impiega C_i secondi. Può darsi che il sistema non sia schedulabile. La condizione di schedulabilità è

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Si distinguono algoritmi **statici** o **dinamici**. I primi funzionano solo se so in anticipo il tipo di eventi.

2.4 Deadlock

I computer sono pieni di risorse che possono essere usate da un solo processo alla volta: stampanti, scanner. Quando i processi richiedono accesso a più di una risorsa, si possono verificare dei deadlock. Altro caso è quello dei database.

In generale il termine **risorsa** indica sia dispositivi hardware che software, che possono essere utilizzate da al più un processo alla volta. Di una certa risorsa, una macchina può avere varie istanze: ad esempio può avere due stampanti, o più nastri magnetici. Quando un processo richiede una risorsa, una di queste istanze può essere utilizzata e le altre rimangono disponibili.

In generale, il pattern di uso delle risorse è il seguente:

1. richiedi risorsa
2. usa risorsa
3. rilascia risorsa

Le fasi di richiesta e rilascio sono di solito gestite da apposite chiamate di sistema. In generale, ogni processo avrà avuto accesso ad alcune risorse e sarà eventualmente bloccato in attesa di una risorsa. Possiamo disegnare la situazione in un **grafico di allocazione risorse**. Abbiamo deadlock quando c'è un **ciclo**.

Due tipi di risorse:

- **preemptable**: può essere tolta ad un processo, senza effetti collaterali (memoria)
- **non-preemptable**: non può essere tolta ad un processo (masterizzatore)

In generale, ci sono 4 strategie per i deadlock:

- ignorare il problema;
- riconoscere il deadlock e ripristinare una situazione non bloccata;
- evitare i deadlock a tempo di esecuzione;
- prevenire i deadlock vincolando le richieste di risorse.

2.4.1 Ignorare il Problema

Può non essere così bizzarra, se la prob. di deadlock è molto bassa. Ad esempio, in molti sist. operativi abbiamo situazioni di deadlock dovuto al fatto che le risorse interne (come la tabella processi) sono finite.

2.4.2 Riconoscimento del deadlock

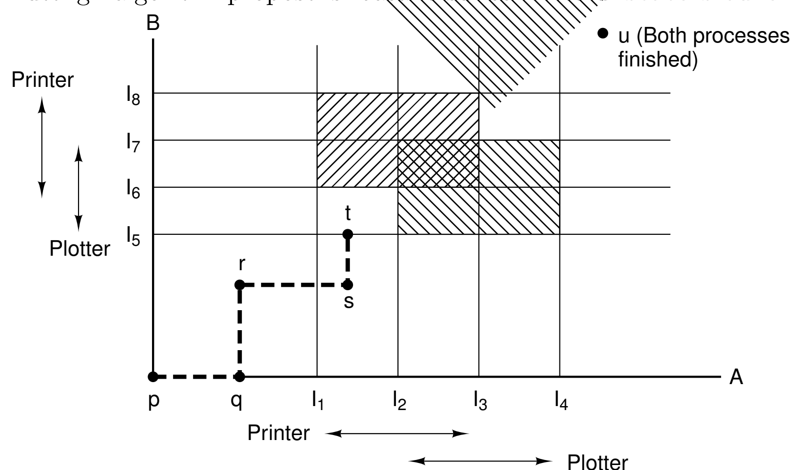
Il sistema non tenta di impedire il deadlock, ma riconosce quando si verifica. Il problema è che una volta riconosciuto, non c'è una buona soluzione:

- preemption (stampante)
- **checkpoint**. Vengono salvati dei checkpoint a intervalli regolari. Quando c'è un deadlock e una risorsa X è necessaria, viene ucciso un processo e ripristinato in uno stato precedente al momento in cui ha richiesto X.
- uccidere un processo: va posto una scelta accurata su quale uccidere.

2.4.3 Evitare i deadlock a tempo di esecuzione

Quando un processo richiede una risorsa il sistema operativo potrebbe anche decidere di non concederla. In questo modo si possono eliminare deadlock potenziali. C'è un algoritmo per fare questo?

Tutti gli algoritmi proposti si basano sul concetto di **stato sicuro**.



Gli algoritmi per evitare i deadlock operano mantenendosi sempre in stati sicuri.. lo svantaggio è che bisogna sapere in anticipo le risorse che un processo.

2.4.4 Prevenire i deadlock

Varie tecniche

- evitare che più processi possano richiedere la stessa risorsa: spooling;
- evitare che i processi possano possedere una risorsa e richiederne un altra. Tutte le richieste devono avvenire contemporaneamente, ma ovviamente non sempre i processi sanno quante risorse utilizzeranno, e inoltre occupo una risorsa per molto tempo.

- in alternativa, rilascio tutte le risorse quando me ne serve una nuova.
- assegnare un ordine alle risorse, e permettere le richieste solo in questo ordine.

Capitolo 3

Problemi di comunicazione tra processi

3.1 Filosofi a cena

- soluzione con deadlock
- soluzione con starvation
- soluzione con attesa non deterministica
- soluzione ottimale
  - vettore AFFAMATO: per ogni filosofo, contiene lo stato corrente tra "pensante", "affamato", "mangiante".
  - vettore di semafori: un filosofo si sospende sul suo semaforo se non ha le forchette.

3.2 Il barbiere che dorme

- semaforo CLIENTI: numero di clienti in attesa
- semaforo BARBIERI: numero di barbieri in attesa
- semaforo MUTEX: mutua esclusione per l'accesso a in_attesa
- variabile IN_ATTESA: conta numero clienti in attesa, ha lo stesso valore di CLIENTI.

Il problema della **attesa indefinita**.

3.3 Lettori/scrittori

Soluzione con attesa indefinita:

- variabile rc: contiene il numero di lettori correnti

- semaforo MUTEX: protegge accesso a rc
- semaforo DB: separa gli accessi dei lettori e dello scrittore

Soluzione senza attesa indefinita:

```
int lettori_attesa=0;
int scrittori_attesa=0;
int lettori_attivi=0;

semaforo sem_lettori=0;
semaforo sem_scrittori=0;
semaforo mutex=1;

void lettore(int i)
{
  while(TRUE) {
    down(&mutex);
    if ((scrittori_attesa==0) && !ScritturaInCorso) {
       lettori_attivi++;
       up(&mutex);
    } else {
       lettori_attesa++;
       up(&mutex);
       down(&sem_lettori);
    }

    leggi_base_di_dati();

    down(&mutex);
    lettori_attivi--;
    if ((lettori_attivi==0) &&  (scrittori_attesa!=0)) {
       scrittori_attesa--;
       ScritturaInCorso=TRUE;
       up(&sem_scrittori);
    }
    up(&mutex);

    elabora_dati();
}

void scrittore(int i)
{
  while(TRUE) {
    pensa_ai_dati();

    down(&mutex);
    if ((lettori_attivi==0) && !ScritturaInCorso) {
        ScritturaInCorso=TRUE;
        up(&mutex);
    } else {
```

```
            scrittori_attesa++;
            up(&mutex);
            down(&sem_scrittori);
        }

        scrivi_base_di_dati();

        down(&mutex);
        ScritturaInCorso=FALSE;
        if ((lettori_attesa==0) && (scrittori_attesa!=0)) {
           ScritturaInCorso=TRUE;
           up(&sem_scrittori);
        } else
           while (lettori_attesa!=0) {
              lettori_attesa--;
              up(&sem_lettori);
           }
        up(&mutex);
      }
    }
```

Capitolo 4

I/O

4.1 Concetti hardware

Il SO deve controllare i dispositivi di I/O e fornire una interfaccia al resto del sistema semplice da usare e omogenea. A questo scopo i dispositivi di I/O possono essere divisi in due categorie: dispositivi **a blocchi** (dischi) e **a caratteri** (stampanti). Ci sono anche dispositivi che non cadono in queste due categorie: orologi o dispositivi video mappati in memoria.

Una unità di I/O è composta da due parti: il **controller** o **adattatore** e il dispositivo vero e proprio. Compito del controller è realizzare in hardware quello che sarebbe troppo faticoso realizzare via software (es. winmodem).

Il controller ha dei **registri** a cui il processore può accedere e dei **buffer** per i dati. Si può optare per il **Memory-mapped i/o** che ha vantaggi e svantaggi:

- le stesse istruzioni per manipolare dispositivi di I/O e variabili
- nessun meccanismo speciale per la protezione
- caching più problematico
- problemi con bus specializzati per l'accesso in ram

Il trasferimento dei dati dai registri alla memoria centrale può avvenire sotto controllo della CPU o tramite **DMA**.

4.2 Concetti software

Alcuni concetti fondamentali dell'I/O dal punto di vista software:

- device independence
- uniform naming
- gestione degli errori: il più a basso livello possibile
- i/o sincrono (bloccante) e asincrono (non bloccante)
- buffer: perchè non si conosce il destinatario o per problemi di temporizzazioni.

- dispositivi dedicati o condivisibili

Ci sono tre modi per implementare le operazioni di I/O in un SO:

- I/O programmato: attesa attiva

```
for (i=0; i<count; i++) {
   while (printer_status_reg != READY) ;
   printer_data_register = buffer[i];
}
return_to_user();
```

- I/O con interruzioni

```
semaphore print_ready=1;

for (i=0; i<count; i++) {
   down(printer_ready);
   printer_data_register = buffer[i];
}

interrupt_handler() {
   up(printer_ready);
}
```

- I/O con DMA: riduzione del numero di interrupt

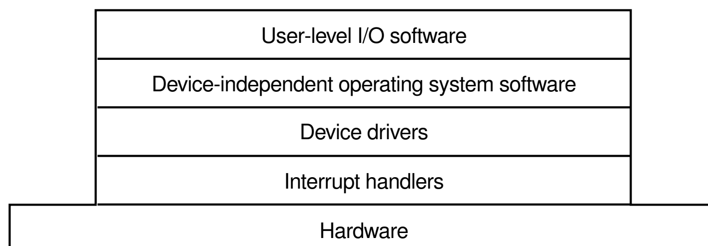
```
semaphore DMA_is_over=0;

set_up_DMA_controller();
down(DMA_is_over);

interrupt_handler() {
   up(DMA_is_over);
}
```

4.3 Organizzazione del software di I/O

Il software di I/O è strutturato a livelli:



Gli interrupt sono necessari se non si vuole attesa attiva. Sono tuttavia difficili da trattare. La soluzione migliore è far si che siano i più trasparenti possibili: trattarli come semafori.

I **device driver** sono progettati per una periferica specifica o periferiche molto simili. È l'unica parte del S.O. che sa come funziona i registri dell'adattatore. Interfacce comuni per dispositivi a **blocchi** o **caratteri**.

Le parti superiori della gerarchia sono indipendenti dal dispositivo. Si occupano di queste funzioni:

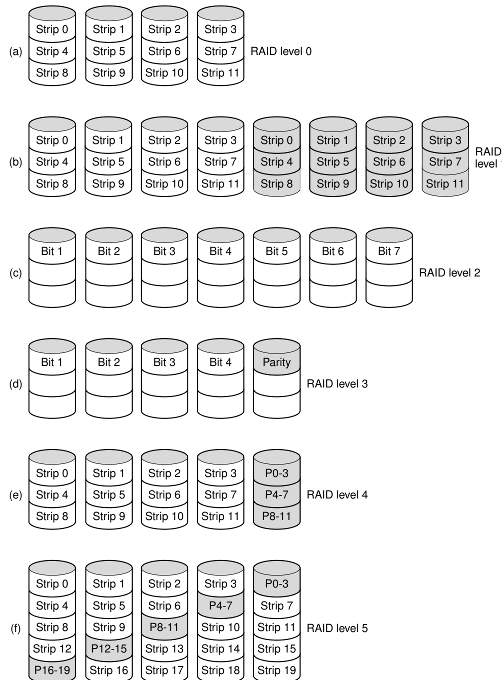
Uniform interfacing for device drivers
Buffering
Error reporting
Allocating and releasing dedicated devices
Providing a device-independent block size

4.4 Dischi magnetici

Sono divisi in cilindri, testine e settori. Ormai il numero di settori per cilindro non è costante, per cui la geometria che il SO vede non è detto sia quella effettiva. Si preferisce spesso il *logical block addressing*.

4.4.1 RAID

Nasce per velocizzare gli HD, che si evolvono più lentamente della CPU.



Vantaggi e svantaggi:

- level 0: ottime prestazioni, bassa affidabilià
- level 1: la scrittura ha le stesse prestazioni di un disco singolo, la letture è più veloce, buona affidabilità
- level 2: ottimo data rate, ottima affidabilità, stesse prestazioni di un disco singolo su operazioni separate.
- level 4: come level 0 con ottima affidabilità ma con un collo di bottiglia nelle operazioni di scrittura a causa del disco di parità.
- level 5: come level 4 ma senza collo di bottiglia perchè la parità è divisa tra i dischi.

4.4.2 Formattazione

I dischi devo essere formattati a **basso livello** prima di essere utilizzati.

Preamble	Data	ECC

Sopra questa formattazione c'è quella ad **alto livello**: file systems

Capitolo 5

File system

Data dalla necessità di memorizzazione:

- grandi quantità di dati
- memorizzazione **permanente**
- accesso concorrente da più processi
- rintracciabilità

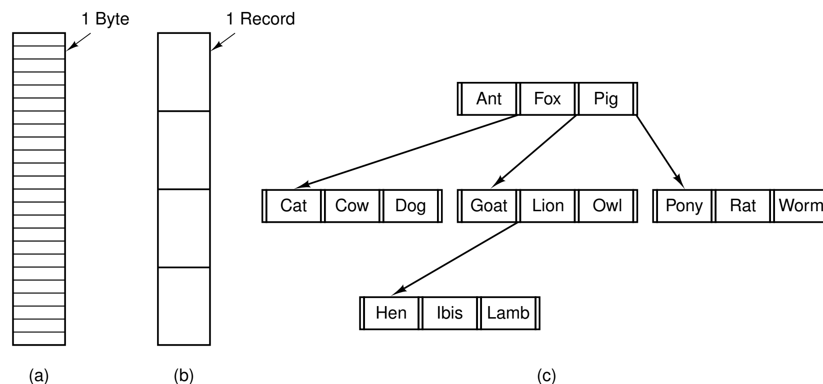
Soluzione: memorizzare le informazioni su supporti di memoria permanente in unità chiamate **file**.

5.1 I file

Serve un meccanismo per recuperare un file memorizzato su disco (o su un altro dispositivo): il **nome**. Alcune differenze nei file systems:

- sensibilità maiuscole/minuscole
- trattamento dell'estensione
- codifica dei caratteri

Struttura dei file:



Tipi di file: file speciali (es. **dispositivi**), file di **testo** e **binari**, **magic number**.

Accesso ai file: sequenziale e casuale.

Attributi dei file: informazioni di protezione, dimensione, date.

Chiamate di sistema: creazione, cancellazione, apertura, lettura, scrittura, seek, rinominazione, operazioni su attributi.

5.1.1 Esempio

```
#define DIM_BUF 4096
#define OUTPUT_MODE 0700

void copia(char* origine, char* destinazione) {
   int in_fd, out_fd, letti, scritti;

   in_fd=open(origine,O_RDONLY);
   if (in_fd < 0) exit(-1);

   out_fd=creat(destinazione,OUTPUT_MODE);
   if (out_fd < 0) exit(-1);

   letti=1;
   while(letti!=0) {
      letti=read(in_fd,buffer,DIM_BUF);
      if (letti < 0) exit(-1);
      scritti=write(out_fd,buffer,letti);
      if (scritti < 0) exit(-1);
   }

   close(in_fd);
   close(out_fd);
}
```

5.2 Le directory

Conflitto nel nominare i file: dalle directory a un livello alle directory gerarchiche.
Specificare i nomi dei file: pathname assoluto, directory corrente, pathname relativo.

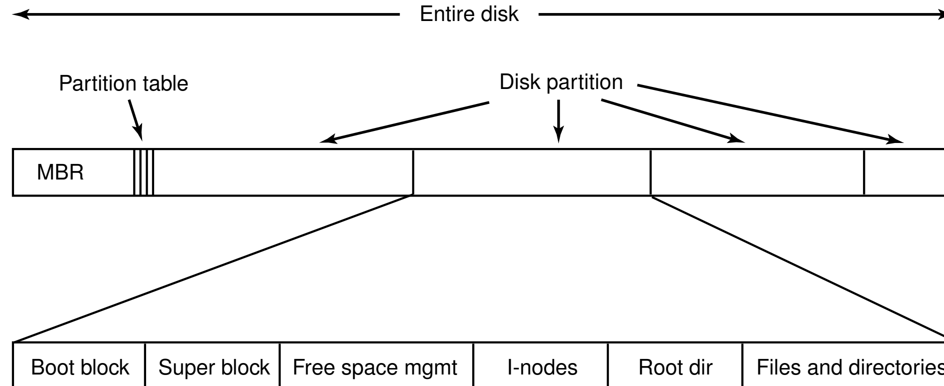
Le directory speciali: directory punto "." e punto punto ".."

Collegamenti

Le chiamate di sistema: creazione, cancellazione, lettura, crea collegamenti, rimuovi collegamento.

5.3 Implementazione

I file system sono memorizzati in dischi, che possono essere divisi in entità logiche chiamate **partizioni**. L'organizzazione generale è la seguente:



5.3.1 Allocazione dei file

Fattore **chiave** nell'implementazione di un file system: metodo di allocazione dei file.

allocazione contigua

- vantaggio: semplice da implementare (posizione iniziale e lunghezza dei file unici dati necessari)
- vantaggio: efficiente in lettura
- svantaggio: frammentazione, spazio sprecato, **deframmentazione**.
- variante: scrittura negli spazi liberi.

allocazione a lista concatenata

- vantaggio: non si perde spazio per la frammentazione
- svantaggio: accesso casuale poco efficiente
- svantaggio: dimensione di un blocco che non è una potenza di due

allocazione con FAT

- vantaggio: non si perde spazio per la frammentazione
- svantaggio: la FAT occupa molto spazio su memoria e disco

allocazione con i-node

- vantaggio: non si perde spazio per la frammentazione
- vantaggio: solo gli inode dei file aperti devono stare in memoria

5.3.2 Le directory

Mappano i nomi del file alla posizione del corrispondente i-nodo o al primo settore del file.

- attributi nelle directory o negli i-nodi
- lunghezza dei nomi fissi o variabile
- ricerca lineare,tabelle hash o alberi binari

La struttura delle directory può essere ad albero o un dag: link **fisici** o **simbolici**.

5.3.3 Spazio Libero

Il problema della scelta della dimensione dei blocchi: **spazio disco** vs **velocità di accesso**. Influenza il numero di bit necessari per accedere a un blocco.

Come tenere traccia dei blocchi liberi: **lista dei blocchi liberi** e **bitmap**. Le **quote** disco.

5.4 Prestazioni del file system

caching

- algoritmi di caching
- tecnica LRU modificata per vari tipi di dato
- cache write-through contro write-back

lettura anticipata

- letture sequenziali e casuali

posizionamento dati sul disco:

- posizionamento inode
- deframmentazione

Capitolo 6

Sicurezza

La **sicurezza** e i **meccanismi di protezione**. La sicurezza ha tre obiettivi.

- confidenzialià dei dati
- integrità dei dati
- disponbilità del sistema

Chi tenta di violare il sistema sono gli **intrusi**.

- utenti casuali non tecnici
- curiosi
- violazione per vantaggio personale
- spionaggio commerciale o militare

Altri danni possono venire da **virus** e per **perdita accidentale**.

6.1 Crittografia

Mai usare la crittografia basata su algoritmi nascosti. Ciò che è segreto deve essere la **chiave**. Si distinguono algoritmi a chiave **simmetrica** o a chiave **pubblica**. La crittografia si usa anche per **firme digitali**.

6.2 Autenticazione

Basata su tre cose:

- qualcosa che l'utente conosce (password)
- qualcosa che l'utente ha (tessere magnetiche, carte con EPROM, smart-card)
- qualcosa che l'utente è (modello retinico)

Chi tenta di violare l'autenticazione è il **cracker**.

6.3 Attacchi

- dall'interno del sistema
  - cavalli di Troia
  - login spoofing
  - bombe logiche
  - porte nascoste. **verifica del software** come metodo di protezione.
  - buffer overflow
- dall'esterno del sistema
  - virus e worm
  - virus: replicazione e **payload**
  - classificazione dei virus: per eseguibili, per MBR, residenti in memoria, macro-virus, etc.
  - tecniche anti-virus: riconoscimento della firma, virus polimorfi, controlli di integrità e di comportamento

6.4 Meccanismi di Protezione

Un sistema contiene oggetti che hanno bisogno di essere protetti. Ogni oggetto ha un nome unico e delle operazioni che è possibile svolgere su di esso (es. `read/write` per file, `up/down` per semafori).

Si introduce il concetto di **dominio**: insieme di coppie (oggetto, diritti). Ogni processo opera in dominio di protezione. Su Unix dipende dalla copia UID (user identifier) e GID (group identifier). I domini si possono memorizzare in una **matrice di protezione**.

La matrice è sparsa, per cui si usano metodi di memorizzazione più efficienti, per riga o colonna.

Per colonna

Si ottengono le **liste di controllo degli accessi** (ACL). Ad esempio il sistema di protezione file di Unix o Windows.

Per riga

Si ottengono le **liste di capacità (capability)** associate ad ogni processo. Servono metodi per proteggere le liste da manipolazioni: **architettura etichettata**, **capability nel kernel**, **criptazione**.

Il vantaggio delle capability è un controllo più granulare che consente di applicare il **principio del minimo privilegio**. È però più difficile da implementare e non consente una revoca selettiva dei diritti.