Riassuntini

giovedì 14 gennaio 2021 10:15

Capitolo 1

1.1

Che cos'è un sistema operativo?

Si potrebbe dire che un sistema operativo è il software su una macchina che viene eseguito in kernel mode, ma sarebbe un po' riduttivo. In generale SO ha due funzioni: fornire astrazioni dell'hardware per gli applicativi e gestire le risorse hardware. L'astrazione serve perché l'hardware a livello basso è complicato, ad esempio il disk driver e sopra ancora i file, e la gestione risorse serve perché il sistema operativo deve poter fare diverse cose. Ci sono risorse multiplessate in tempo, come il processore e la stampante, e risorse multiplessate in spazio, come il disco e la memoria.

1.2

Come si sono sviluppati i sistemi operativi?

I tipi di sistemi operativi si possono crudamente abbinare alla generazione di macchina su cui si sono sviluppati.

La prima generazione di macchine, i calcolatori meccanici e con valvole, non aveva sistemi operativi e neanche un linguaggio macchina. I calcoli venivano programmati direttamente sulla macchina spesso modificando una plug board.

La seconda generazione di macchine è quella dei sistemi batch, che inizialmente non avevano un sistema operativo. Bisognava inserire le punch card, in caso inserire un compilatore, e poi passarlo ad un sistema che faceva i calcoli. Una operazione alla volta. Si è poi automatizzato questo processo avendo una macchina più piccola che raccoglieva gli input, e una macchina più grande che eseguiva i calcoli e poi mandava ad una terza macchina i dati da stampare in output. C'era un rudimentale sistema operativo che in caso caricava il compilatore FORTRAN. Questi sistemi operativi erano FMS e IBSYS.

La terza generazione ha introdotto la multiprogrammazione, con la possibilità di eseguire calcoli durante le operazioni di IO per non lasciare in idle queste grosse macchine, quindi c'erano più processi in diversi segmenti di memoria. Con l'arrivo dei circuiti integrati e l'aumento delle prestazioni, venne introdotto anche il timesharing, una variante della multiprogrammazione che permetteva di avere più utenti collegati in contemporanea, con una allocazione proporzionale di risorse, con il CTSS di IBM. Venne inoltre sviluppato il multix, un sistema di timesharing che avrebbe permesso di avere centinaia di utenti collegati. Ken Thompson sviluppa una versione di unix per un solo utente che ha diverse diramazioni sviluppate anche da persone diverse come BSD e system V. Poi si standardizzano le interfacce di unix con posix. A partire da minix si sviluppa linux.

La quarta generazione dei personal computer, con CPM e MSDOS e MacOs e le UI e così via.

E poi gli smartphone.

1.3 (esclusi 1.3.3 e 1.3.6)

Hardware dei computer

1.5 (fino a 1.5.3 incluso)

Concetti parte del sistema operativo

I processi

I processi sono come dei contenitori per tutte le informazioni necessarie ad eseguire un programma, che non sono memorizzate nell'address space ad esso associato. Queste informazioni sono memorizzate in molti sistemi operativi in un array di strutture chiamato process table.

Address space

Insieme di indirizzi di memoria a cui un processo può accedere. Può essere fisico o virtuale.

In memoria principale può esserci un processo per volta, e quindi in generale un solo address space, o più processi contemporaneamente, ma nel secondo caso bisogna implementare dei meccanismi di protezione. Su macchine moderne gli indirizzi però sono 2^32 o 2^64 byte, quindi si usa una tecnica chiamata memoria virtuale, che è una astrazione dell'address space fisico.

File

I file sono un'astrazione del disco, organizzati in modo gerarchico all'interno di cartelle. La struttura gerarchica si può descrivere con un path, che può essere assoluto o locale.

Aprendo un file bisogna controllare i suoi permessi, dopodichè viene ritornato un file descriptor da usare nelle operazioni successive.

I dispositivi di IO su unix si affidano ai file per le operazioni di IO, con i block special files e i character special files.

Un pipe è un file che viene usato per la comunicazione tra due processi.

1.6 (escluso 1.6.5)

System Calls

Le system call sono il modo che i processi hanno per accedere a certe funzionalità della macchina, come le operazioni di IO, la creazione e rimozione di processi, la richiesta di memoria aggiuntiva, etc.

Per le system call spesso ci sono delle librerie che semplificano le chiamate. Queste mettono i parametri della nostra chiamata (tipo identificativo file, byte da leggere e un buffer su cui metterli) in un posto in cui il sistema operativo se le aspetta (lo stack) e poi trappano il kernel, che leggerà le informazioni relative alla system call e da una tabella selezionerà l'handler adatto che farà le proprie operazioni e poi ritornerà alla procedura della libreria, che ritornerà al programma principale.

La chiamata fork genera un nuovo processo identico al padre. Sarà poi compito del figlio determinare di essere un figlio (il pid ritornato da fork è zero per i figli) e caricare al proprio interno un programma differente tramite la system call exec.

La system call link crea una nuova entry in una directory (i-node, nome) che ha lo stesso i-node di un file già esistente ma possibilmente un nome diverso. Nell'i-node c'è un contatore ai file che ci fanno riferimento (quando è zero ci si può scrivere sopra).

1.7 (escluso 1.7.6)

Struttura dei sistemi operativi

Nei sistemi operativi monolitici tutto il sistema operativo viene eseguito a livello kernel in un solo file binario. Quindi tutte le procedure sono accessibili tra di loro, fornendo sì una certa comodità ma anche complessità strutturale, problemi di sicurezza e il rischio che un errore in una singola procedura possa buttare giù tutto il sistema.

Nei sistemi operativi stratificati non c'è un solo binario. Al livello 0 si gettano le basi per la multiprogrammazione, quindi ai livelli successivi si possono fare più cose in contemporanea (eliminando anche il rischio che una butti giù tutte le altre). Layer 1 si occupa della paginazione, layer 2 della comunicazione tra processi, Layer 3 IO, Layer 4 con tutti i programmi utente.

Nei sistemi operativi a microkernel si riduce il kernel al minimo, facendogli fare solo interrupt, scheduling e comunicazione tra processi. Anche i device driver vengono eseguiti in usermode, e devono fare chiamate al kernel per funzionare.

Un sistema operativo di gestione delle virtual machine, nella sua versione più basilare (come quella dei vecchi framework ibm), fa da layer intermediario tra sistemi operativi completi e l'hardware sottostante, con un sistema di timesharing per gestire le risorse. In pratica i sistemi mandano richieste all'hardware, di cui la vm fa una trap e gestisce come vuole.

La versione disponibile al pubblico di questo sistema si chiama hypervisor, che può essere di tipo 1 se ci si installano direttamente sopra tutti i sistemi operativi della macchina, e di tipo 2 se fa uso di un sistema operativo host.

Capitolo 2

2.1

Un processore in un determinato istante esegue un'operazione per volta. Il sistema operativo si occupa di dare l'illusione di parallellismo, una sorta di pseudoparallelismo, alternando diversi processi velocemente. Questo sistema si chiama multiprogrammazione.

In unix c'è un solo modo per creare un processo, la chiamata fork. In generale i processi possono essere creati da altri processi (che siano controllati da un utente come la shell o lavori batch) o all'inizializzazione, direttamente dal sistema operativo.

I processi possono terminare perché hanno terminato il loro compito o per volontà dell'utente. Oppure se fanno qualcosa di illegale per cui arriva un fatal error per cui non hanno un handler, oppure possono essere terminati da un altro processo con le giuste autorizzazioni.

I processi si possono essenzialmente trovare in tre stati: running e ready (che si alternano in base alle decisioni dello scheduler, e blocked (in cui il processo va quando sta aspettando qualcosa, tipo un input o una risorsa). Quando un processo riceve la risorsa che aspettava, va da blocked a ready.

Tutte le informazioni sul processo che non sono nel suo address space sono nella sua entry nella process table, queste sono:

Registri, program counter, psw, stack pointer, priorità, parametri di scheduling, ID processo, parent ID, process group, segnali, tempo a cui il processo è iniziato, tempo di cpu utilizzato, tempo cpu dei figli, tempo del prossimo alarm, puntatori a alle info di text segment, data segment e stack segment, root directory, working directory, descrittori del file, urer ID, group ID.

Quando arriva un interrupt, il program counter, il puntatore dello stack, la psw e alcuni dei registri sono buttati nello stack corrente dall'hardware, che poi seleziona un interrupt handler e passa il controllo a una procedura assembly che salva il resto dei registri e carica un suo stack temporaneo per poi passare il controllo alla procedura C. Lo scheduler a questo punto decide che processo deve essere eseguito e una procedura assembly carica le informazioni richieste.

Con n processi in memoria, p che è la probabilità che sia in attesa di IO in ogni determinato istante, allora l'utilizzo della cpu è dato dalla euristica 1-p^n.

2.2 (fino a 2.2.6 incluso tranne 2.2.3)

In molte situazioni è preferibile avere più thread di controllo per processo, che condividono lo stesso address space. Ci sono 3 ragioni per cui avere i thread è una buona idea, la prima è che

i processi svolgono diverse funzioni ed ha senso strutturalmente dividerle in filoni di controllo diversi, la seconda è che sono molto più leggeri dei processi per quanto riguarda la creazione e distruzione, quindi il numero di operazioni da eseguire varia molto e quindi anche il numero di thread, ha senso usare i thread. La terza ragione è la performance, infatti i thread sono molto più utili quando vengono usati su un processo che non è interamente CPU bound, così che l'IO e il calcolo possano essere eseguiti in parallelo.
I thread utilizzati in un server possono essere un dispatcher e una serie di worker thread. Quando arriva una richiesta il dispatcher la carica da qualche parte e sveglia un worker thread per fargliela elaborare, passando il filo di controllo. È importante cosiderare se le system call sono bloccanti per tutti i thread nel processo o soltanto per il chiamante. Nel primo caso si perde lo pseudo parallelismo ma si mantiene la maggior semplicità strutturale.
Ogni thread ha il suo program counter, i suoi registri e un suo stack, ed uno stato per la schedulazione. Alcuni processori hanno supporto hardware diretto per il multithreading, permettendo switch che richiedono soltanto nanosecondi.
Un thread si può chiamare con una procedura thread_create, in cui si specifica la procedura da eseguire. Un thread può terminarsi chiamando thread_exit o può attendere il completamento di un altro thread con thread_join. È importante la chiamate thread_yeald, che permette ad un thread di essere messo volontariamente in ready dallo scheduler, perché non ci sono interrupt di clock nei thread come avviene per i processi.
I thread possono essere implementati in user space o nel kernel, ma anche approcci ibridi sono possibili.
I thread user space non sono soggetti ad interrupt di clock, quindi quando fanno qualcosa che potrebbe bloccare devono chiamare una procedura runtime (c'è un sistema runtime che li gestisce tutti, con una thread table per processo) che controlli se si blocca. Il vantaggio grosso è che chiamando thread yeald lo switch si può fare molto velocemente perché non c'è bisogno di fare un context switch. Inoltre nel sistema runtime si può implementare un algoritmo di scheduling a scelta. Un problema che si presenta è quello delle chiamate bloccanti (tipo l'attesa di un input da tastiera), che dovrebbero essere rese non bloccanti (ma non si può riscrivere l'os) o avvolte in un wrapper che testa se la chiamata bloccherà, schedulando in caso un altro thread.
Nei thread implementati a livello kernel la thread table sta nel kernel, e per crearli e distruggerli si usano chiamate di sistema. Dato che le chiamate aggiungono overhead, i thread terminati non vengono eliminati ma vengono messi in stato "terminato" quindi quando un nuovo thread deve essere creato si scrivono semplicemente i dati nel thread già esistente. Possono essere schedulati per processo o globalmente.
Con un implementazione ibrida si possono avere thread userspace che vanno sopra i thread kernel space.
In tutti i casi i casi rimane il problema del fork, si duplicano anche i thread? Se poi si fa un exec non conviene ma altrimenti forse sì. C'è inoltre il problema dei segnali, a quale thread mandarlo e come?

2.3 (tranne 2.3.8, 2.3.10, le parti in pseudo-assembler, implementazione dei mutex e mutex applicati ai pthread)

Nella comunicazione tra processi, quindi nella condivisione di risorse, sorge il problema delle race condition, situazioni in cui due o più processi leggono e scrivono su risorse condivise, e producono risultati inconsistenti se lo scheduler decide di andre di mezzo, ad esempio, tra la lettura di un puntatore ad una casella libera e la scrittura di un valore a tale indice (nel frattempo la casella potrebbe non essere più libera).
La soluzione è l'introduzione di regioni critiche di mutua esclusione, in cui un solo processo per volta può entrare. Per avere una regione critica corretta bisogna anche far sì che: non ci siano assunzioni sul numero di cpu, nessun processo in esecuzione al di fuori della propria regione critica può bloccare un altro processo, e nessun processo dovrebbe attendere per sempre per entrare nella propria regione critica.
Su processori single core, la soluzione più semplice è di disabilitare gli interrupt ma ci sono mille ragioni per cui non è una buona idea.
Si può avere uno spinlock, che testa continuamente una variabile per vedere se si può entrare in regione critica. Ma avere un while sempre attivo effettivamente viola la terza condizione perché nessun processo può essere eseguito quando un processo esegue un check a ripetizione.
La soluzione di Peterson è migliore ma pur sempre uno spinlock. Si aggiunge un vettore di interessi, e delle funzioni per entrare ed uscire dalla zona critica.

```
Enter_CriticalZone
    Interested[self] = true
    turn = self
    While turn == self AND Interested[other] == true
Leave_CriticalZone
    Interested[self] = false
```

Esistono istruzioni apposite per creare i LOCK come TSL (Test and Set Lock) che vengono svolte atomicamente e bloccano il memory bus per evitare che si creino race condition, e richiedono hardware apposito. Sui processori Intel c'è XCHG.
Il problema del priority inversion si presenta quando un processo di alta priorità aspetta in busy waiting un processo in regione critica a priorità più bassa che non verrà mai schedulato. Quindi è meglio usare le chiamate SLEEP e WAKEUP(processo da svegliare) così da evitare gli spinlock.
Nel problema del produttore-consumatore due processi condividono un buffer di dimensione fissa in cui il produttore inserisce elementi e il consumatore ne toglie. La soluzione più semplice produce race condition perché se ad esempio il consumatore legge N, che è 0, e poi viene deschedulato, il produttore lo mette a uno, assume che prima fosse zero, sveglia il consumatore (che non sta dormendo), prima o poi andrà a dormire e appena il consumatore viene schedulato andrà a dormire anche lui. La soluzione è di avere un WAKEUP bit che viene settato quando si fa un WAKEUP, e che quindi può essere controllato prima di fare uno SLEEP (e resettato ad ogni iterazione).
Dijkstra ha introdotto i semafori, dei contatori di WAKEUP accumulati, con operazioni DOWN (che diminuisce e usa un wakeup) e UP (che aggiunge un wakeup) atomiche. Quando un processo vuole svegliarne altri, fa un UP e, se il semaforo era 0 vuol dire che c'erano processi in attesa, uno di questi si sveglierà e completerà il suo down effettivamente lasciando il semaforo a zero ma con un processo in meno in attesa.
La soluzione del produttore-consumatore con i semafori è:

```
Producer
    Down(&empty)
    Down(&mutex)
    Produce(buffer)
    Up(&mutex)
    Up(&full)
Consumer
    Down(&full)
    Down(&mutex)
    Produce(buffer)
    Up(&mutex)
    Up(&empty)
```

Il semaforo mutex è usato per la mutua esclusione mentre full e empty per la sincronizzazione. Il mutex può essere usato per gestire il blocco di processi che fanno richieste a risorse condivise. Esiste anche una primitiva apposta per il mutex. C'è l'assunzione che ci sia della memoria condivisa (come in Unix e Windows) o che i semafori funzionino attraverso system call al kernel.
Futex Fast User Space Mutex per linux ha una libreria kernel con una coda di processi in attesa, ma che viene usata solo se il test al lock fatto in user space ci dice che è già preso.
Un monitor è una collezione di procedure e dati che si occupa automaticamente di gestire la mutua esclusione, le attese e i segnali. In pratica è un concetto di linguaggio, che può essere implementato dal programmatore. Si possono usare delle condition, che sono variabili usate per attese e segnali a cui viene allacciato in compilazione un semaforo di qualche tipo. L'idea è che il compilatore sia meno prono ad errori del programmatore nell'implementare queste cose. Per il produttore consumatore è il seguente:

```
MonitorProducerConsumer:
    Condition empty, full
    Integer count
    Insert(Item):
        If count == N wait on full
        insertItem(item)
        count++
        If count == 1 signal empty
    Remove(item):
        If count == 0 wait on empty
        removeItem(item)
        count--
        If count == N-1 signal full
Producer:
    While true
        Produce(item)
        MonitorProducerConsumer.Insert(item)
```

Una barriera è un meccanismo di sincronizzazione pensato per applicazioni divise in fasi, con più processi. Quando tutti i processi verificano una condizione, possono passare alla fase successiva. È utile per esempio in grosse matrici.
Il problema dei dining philosofers è stato introdotto da Dijkstra. Ci sono 5 filosofi, che possono mangiare o pensare. Davanti a ciascuno di loro c'è un piatto, e una forchetta alla destra di

ciascuno. Per mangiare devono prendere la forchetta a destra e la forchetta a sinistra, quindi non possono mangiare tutti insieme. Prendendo semplicemente una forchetta dopo l'alta si rischia di incorrere in un deadlock, quindi si potrebbe proteggere questa operazione con un mutex, ma poi un solo filosofo per volta potrebbe mangiare. La vera soluzione consiste nell'avere un semaforo per ciascun filosofo in modo da potersi bloccare o svegliare, e un array di stato che deve essere controllato per il vicino di destra e sinistra (nessuno dei due deve mangiare) prima di mettere un filosofo a mangiare.

```
Philosofer(i)
    While true
        Think()
        Takeforks(i)
        Eat()
        Putforks(i)
Test(i)
    If state[i] == hungry AND state[left] != eating AND state[right] != eating
        State[i] = eating
        Up(s[i])
Takeforks(i)
    Down(mutex)
    State[i] = hungry
    Test(i)
    Up(mutex)
    Down(s[i])
Putforks(i)
    Down(mutex)
    State[i] = thinking
    Test(LEFT)
    TEST(RIGHT)
    Up(mutex)
```

2.4

Che cos'è lo scheduling?
Lo scheduler è la parte di sistema operativo che si occupa di scegliere quale processo deve essere messo in esecuzione, quando, e con quale criterio sulla base di un algoritmo di scheduling.
Lo scheduling era importante nei sistemi programmati ma meno importante nei personal computer di oggi. Ad oggi è importante nei server di rete. C'è da considerare che cambiare il processo in esecuzione comporta un po' di overhead perché bisogna salvare registri, cache, memory map in alcuni casi, e ricaricare tutto del nuovo processo. Quindi il numero di switch per secondo è una metrica importante.
I processi si dividono in IO bound e CPU bound, in base alla percentuale di tempo spesa in operazioni di IO.
Esistono algoritmi di scheduling con senza prelazione che semplicemente eseguono un processo finchè non si blocca o non rinuncia alla cpu di sua spontanea volontà, e altri con prelazione che possono eseguire un processo per un quanto di tempo e per poi togliergli il controllo e darlo a qulcun altro.
Gli algoritmi possono essere per: sistemi batch (che possono tranquillamente essere senza prelazione perché non ci sono utenti che aspettano risposta immediatamente), interattivi (in cui la prelazione è essenziale per far sì che all'utente arrivi qualcosa in tempi brevi) o realtime (in cui la prelazione non serve perché tutti i programmi sono pensati per un solo scopo).
Gli obiettivi degli algoritmi sono:

- Per tutti i sistemi, giustizia (dare a tutti una giusta quantità di cpu), applicazione delle polizze, bilanciamento (tenere occupate tutte le parti del sistema).
- Per i sistemi batch, throughput massimizzato, turnaround time minimizzato, utilizzo cpu massimizzato
- Per sistemi interattivi, response time minimizzato, proporzionalità (bisogna rispettare le aspettative dell'utente più dell'effettivo carico di lavoro coinvolto nel considerare la giusta suddivisione).
- Per sistemi realtime, rispettare scadente e funzionare in modo prevedibile.

Gli algoritmi di scheduling per sistemi batch sono:

- First Come First Served
  - Senza prelazione, i lavori vengono eseguiti nell'ordine in cui arrivano, con una coda.
- Shortest Job First
  - Senza prelazione, ogni volta che lo scheduler viene chiamato, verrà eseguito il lavoro più breve. Massimizza il throughput, ma il turnaround time può tendere ad infinito.
- Shortest Job First Con Prelazione
  - Se arriva un lavoro più breve, lo scheduler viene chiamato all'interrupt di clock ed eseguirà quello.
- Shortest Remaining Time Next
  - Con prelazione, esegue ad ogni interrupt di clock il processo che ha meno tempo di burst rimanente.

Per sistemi interattivi gli algoritmi di scheduling sono:

- Round Robin
  - Ha una lista di processi da eseguire con un puntatore circolare che esegue operazioni per il processo puntato per un quanto di tempo (polizza), per poi passare al successivo. Nel sistema del prof i processi in arrivo durante l'esecuzione di X, sono messi a sinistra di X spingendo tutti quelli che ci sono già a sinistra.
- Priority Scheduling
  - Lo scheduling con priorità divide i processi in famiglie di priorità e all'interno di esse schedula con round robin. Per far sì che anche i processi meno prioritari vengano schedulati ogni tanto, si può far sì che scalino di priorità con il tempo.
- Guaranteed scheduling
  - Ogni processo deve avere 1/n di cpu time. Ad quanto si controlla il rateo di tutti i processi, e si eseguono quelli con il rateo basso per bilanciare il tutto.
- Lottery scheduling
  - Pseudorandomico con biglietti, processi con priorità più alta possono avere più biglietti.
- Fair Sharing
  - Divide la cpu per utente, non per processo.

I sistemi realtime possono essere di tipo hard, per cui devono rispettare tutte quante le deadline, per forza, o soft (tipo un sistema di multimedia) che in caso possono sgarrare un pochettino.

Capitolo 3

La parte del sistema operativo che si occupa della gestione della gerarchia di memoria è il memory manager. Sa che processi sono allocati, dove, e come gestire i rimpiazzi.

3.1

Per memoria senza astrazioni si intende un sistema in cui c'è un solo programma in memoria, insieme al sistema operativo (che può essere interamente o parzialmente in una rom). Quindi il processo ha accesso diretto agli indirizzi di tutto l'address space. Per avere più programmi in memoria (senza swappare) bisognava evitare di usare indirizzi assoluti, naturalmente. È meglio un sistema in cui ogni programma ha accesso ad un proprio address space privato.

3.2

Il concetto di address space è relativamente semplice. Ogni processo ha una sezione di memoria indicata da un registro base (che viene sommato in runtime ad ogni riferimento assoluto alla memoria) e un registro limite (da confrontare per vedere che non si stia cercando di scrivere fuori dal proprio address space. Naturalmente non c'è spazio per tutti i processi e non arrivano tutti in memoria allo stesso momento, quindi ci sono due tecniche utilizzate per gestire la memoria: lo swapping per cui ogni processo viene portato interamente in memoria in uno spazio libero, e la memoria virtuale che permette ad un processo di essere eseguito anche se solo parte dei suoi dati è in memoria.
Nello swapping appunto, i processi vengono infilati nei posti liberi. Bisogna considerare che la memoria utilizzata dai processi non è fissa, quindi solitamente si riempie l'inizio dell'address space con il text segment che è fisso, e la parte successiva con il data segment che cresce in alto, e la parte finale dello spazio con lo stack che cresce in basso. Se si ha soltanto un data segment allora questo può crescere finchè non arriva al base address del processo successivo.
Quando non c'è spazio disponibile, si può attendere che si liberi uno slot grande a sufficienza o uccidere qualche processo in memoria.
La memoria libera va mappata in qualche modo, per sapere dove sono gli spazi. In generale ci sono due modi per farlo, bitmap e free list. La bitmap è letteralmente un array in cui ogni casella indica la libertà o meno della word a cui si riferisce (1 occupato, 0 no). La free list è lista linkata in cui ogni casella indica uno l'inizio dello spazio libero, la sua lunghezza, e un puntatore alla casella successiva.
Ci sono diversi algoritmi per scegliere in che buco mettere un programma:

- First fit, autoesplicativo e veloce
- Next fit, è come first fit, ma ha un puntatore per proseguire dall'ultima casella libera riempita.
- Best fit, cerca l'intera lista per trovare lo slot più piccolo che può accogliere il programma
- Quick fit, ha diverse free list, in base alle dimensioni dei buchi, quindi effettivamente molto veloce perché deve solo scegliere un buco appropriato prendendolo a caso dalla lista con

dimensione interessata.
Worst fit, prende il buco più grande, con l'idea di creare meno frammentazione esterna lasciando buchi più grandi possibile.

3.3

Il problema dello swapping è che non si può usare più memoria di quella che si ha a disposizione, quindi è stata introdotta la memoria virtuale. Inizialmente sotto forma di overlay, piccoli moduli in cui i programmatori dividevano i programmi, che venivano caricati in memoria da un overlay manager. Questo sistema si è poi evoluto nella paginazione, per i processi sono divisi in pagine di dimensione fissa e la memoria in frame che le possono accogliere. In questo modo ogni processo ha a disposizione un address space virtuale, grande quanto l'intero spazio di indirizzamento. Quando si usa la memoria virtuale, gli indirizzi virtuali non vanno direttamente alla memoria ma passano da una Memory Management Unit che li converte a indirizzi fisici.
La prima parte dell'indirizzo virtuale fa riferimento alla pagina, e la seconda fa riferimento all'offset, ovvero a dove all'interno della pagina bisogna andare a puntare. La MMU prende la prima parte dell'indirizzo e va a controllare la page table per vedere se è disponibile in memoria o va recuperata dal disco, e in caso lancia una trap page fault. In sostanza la page table è una funzione che mappa un indirizzo virtuale ad un indirizzo fisico.
Gli elementi di ciascuna entry della page table sono:

Page frame number, il risultato della funzione in sostanza
Absentee bit, che dice se la pagina manca in memoria
Bit di permessi
Referenced bit, indica se la pagina è stata referenziata, quindi letta
Modified bit, settato in automatico dall'hardware quando la pagina viene modificata. È anche chiamato dirty bit, una pagina dirty deve essere scritta su disco prima di essere swappata via.
Caching disabled, utile per dispositivi di IO perché non vogliamo ad esempio leggere sempre la stessa versione cachata dell'input da tastiera.

Sorge il problema della velocità, perché con uno spazio di indirizzamento molto ampio la page table deve anch'essa essere ampia. Avere una grande tabella hardware fatta di registri sarebbe ideale, ma non pratico causa costo. Avere tutta la tabella in memoria principale funziona, ma richiede un accesso in più per vedere dove si trova la tabella.
La soluzione è stata di equipaggiare la MMU di un set di registri hardware (da 8 a 256) in cui mettere alcune entry della page table, chiamato Translation Lookaside Buffer (TLB). Questo può controllare in contemporanea ciascun registro, quindi dandogli in input il numero di pagina ci dice subito il pageframe o ci restituisce un errore, per cui la MMU va a fare un table lookup normale. Se trova la pagina nella page table, sceglie una pagina nella TLB da ricopiare in page table (bit modified incluso) e ci mette la pagina appena trovata. In architetture RISC, la TLB è un gestita a livello software fornendo alcuni vantaggi, come la possibilità di precaricare pagine che si pensa saranno referenziate a breve. In generale una soft missa indica che la pagina non era in TLB, ma nella page table, mentre una hard miss indica che la pagina era soltanto su disco.
Rimane il problema di come gestire address space molto larghi. Come primo approccio possiamo considerare una page table su più livelli, in cui abbiamo nei bit più significativi un numero di pagina, poi un secondo numero di pagina che ci indica in sostanza la posizione nella page table indicata dal primo numero, e infine un offset. Possiamo naturalmente aggiungere livelli, ma per ogni livello si aggiunge un accesso in memoria.
Un altro approccio consiste nell'utilizzare la struttura dati dizionario, con una funzione di hash e delle liste di collisione. Si hasha il numero di pagina, e da lì si scorre una lista in cui dobbiamo vedere se c'è un frame number per la nostra pagina. È un sistema lento, si usava su PowerPC.

3.4

Algoritmi di rimpiazzo pagine.
Algoritmo ottimale, rimpiazza la pagina che verrà usata più in là nel tempo (o una che non verrà più usata).
Least Recently Used, tiene una lista con l'ordine dell'utilizzo di tutte le pagine, e rimpiazza quella che è stata usata per ultima (si può approssimare con un contatore di memory reference che viene scritto sulla entry della pagina referenziata).
Not Recently Used, divide le pagine in classi secondo i seguenti criteri, e ne rimuove una a caso dalla classa più bassa disponibile.

0. Non referenziata, non modificata
1. Non referenziata, modificata
2. Referenziata, non modificata
3. Referenziata, modificata

First In First Out, autoesplicativo, tiene una lista linkata dell'ordine di aggiunta delle pagine.
Second Chance, variante di fifo che controlla il bit referenced, se lo trova ad 1 lo pulisce e riaggiunge la pagina in lista, effettivamente dandole una seconda possibilità.
Clock, che è letteralmente second chance con un puntatore circolare, e il solito bit R usato allo stesso modo. Si gira finchè non si trova una pagina rimpiazzabile, la si cambia con una nuova e si porta avanti il puntatore.
Not Frequently Used, ha un contatore associato ad ogni pagina (a livello software), a cui ad ogni clock aggiunge 1 se R = 1, altrimenti niente. È pesante e non pratico. La versione modificata AGING ha un set di bit per ogni entry, a cui aggiunge a sinistra il bit R ad ogni ciclo, poi la pagina con più zero a sinistra verrà eliminata perché sarà quella usata meno recentemente.
Working Set PRA, fa uso del concetto del working set (le pagine utilizzate nelle k memory reference più recenti).

La CPU ha un current virtual time sempre a disposizione, il bit R è sempre relativo al ciclo corrente. Ad ogni page fault, tutta la page table viene iterata, e le pagine con bit R a 1 hanno il loro Time of Last use aggiornato al tempo CPU corrente. Se R = 0, la pagina è candidata alla rimozione, quindi si confronta la sua età (CPU time - Time of last use) con una soglia prestabilita, se è maggiore la si rimuove, altrimenti ci si segna a parte la pagina di età maggiore per velocizzare il rimpiazzo successivo.

Wsclock, funzina come il working set ma è molto più efficiente. Ad ogni page fault gira e se trova una pagina dirty la schedula per scrittura su disco in modo da non farlo subito, proseguendo con il puntatore fino a trovare una pagina non in working set che sia clean.

3.5 (fino a 3.5.7 incluso)

Nell'implementazione della memoria virtuale sorgono alcuni problemi, primo tra questi è se usare gli algoritmi di rimpiazzamento pagine a livello locale, ovvero per processo, o a livello globale. Bisogna tenere conto di una metrica chiamata PPR Page Fault Rate che indica i page fault in un dato quanto di tempo. Si usa un grafico a doppia soglia, superata quella maggiore si switcha a rimpiazzo globale, e superata quella minore a rimpiazzo locale. Nel mezzo si continua con la politica corrente, per evitare switch di politica costanti.
A volte non si riesce in alcun modo a ridurre il page fault rate di un processo, e quindi bisogna ridurre il carico complessivo sulla macchina swappando a disco alcuni processi.
Un altro problema importante è la dimensione ottimale delle pagine. In generale metà dell'interno di ciascuna pagina viene sprecato, quindi la frammentazione media interna è np/2 in cui p dimensione pagine e n frame utilizzati. Un programma che ha bisogno di 4kb per volta spreca spazio con pagine maggiori di 4kb, ma avere pagine troppo piccole aumenta lo spazio della page table e riduce l'efficacia del TLB.
L'overhead medio per processo è dato da se / p + p/2 in cui s è dimensione media del processore, p sempre la dimensione pagina, e lo spazio occupato da un elemento in page table. Quindi e * s/p è spazio in page table * numero pagine, da sommarsi allo spazio sprecato per frammentazione interna.
La dimensione ottimale delle pagine è data dall'euristica: p = sqrt(2se).
Quando lo spazio di indirizzamento era più piccolo, si era pensato di dividerlo in due, effettivamente duplicandolo, in Instruction Space e Data Space. Al giorno d'oggi in generale non serve a molto, ma torna utile nella cache L1, in modo da avere soltanto le cose che si usano in cache (che è piccola).
È importante avere la possibilità di condividere alcune pagine. Se I space e D space sono separati, basta rendere le pagine I o D accessibili da più processi e avere puntatori appositi nella process table.
Quando un processo forka su unix, le pagine non sono duplicate ma rese read only, con successivi copy on write.
È possibile avere librerie dinamicamente linkate, che vengono portate in memoria su richiesta dopo l'esecuzione di un processo che ci fa riferimento. Così si riduce spazio su disco e in memoria evitando di includerle ogni volta nel binario, permettendo anche aggiornamenti del modulo senza ricompilare tutto.
Avere un file mappato in memoria consiste nell'avere una porzione di file riportata in memoria centrale, in cui fare operazioni di IO senza interpellare il disco. In questo modo si possono spostare le operazioni di IO effettiva ad un momento meno caotico.

3.6

Cosa succede ad un page fault?

1. Hadware trappa il kernel salvando il program counter sullo stack
2. Una routine assembly salva il resto dei registri e le informazioni volatili
3. Il sistema operativo si informa su cosa è successo, scopre che è un page fault, e cerca quale pagina mancava
4. Quando conosce l'indirizzo virtuale controlla la validità e la protezione
    i. Se c'è un segmentation fault si segnala o uccide il processo
    ii. Se è valida si controlla se c'è un page frame libero facendo andare il PRA
5. Se il frame selezionato è dirti si schedula la pagina per il trasferimento e si marca il frame come busy. Si fa un context switch per far andare lo scheduler e si blocca il processo finchè non è completato l'IO.
6. Quando il frame è pulito, l'SO controlla dove su disco si trova la pagina e inizia a portarla in memoria, lasciando bloccato il processo che ha faultato.
7. Un interrupt del disco segnala che la pagina è arrivata, quindi la page table si aggiorna e si marca il frame come normale.
8. Si ricarica l'istruzione che aveva faultato
9. Si rischedula il processo
10. Routine di caricamento

Spesso alcune istruzioni possono essere completate solo parzialmente, quindi a volte i processori mettono dei registri a disposizione per capire a che punto si è arrivati, altre volte no e ci si arrangia.
A volte si possono pinnare delle pagine in memoria per farci scrivere i dispositivi di IO.
Per la scrittura pagine sul disco si può avere una corrispondenza diretta che occupa un sacco di spazio o passare da una diskmap che sta in memoria.

È buona cosa separare politica e meccanismi, avendo per esempio un sistema per settare le priorità a livello user, o a volte di usare addirittura un pager a livello user.

3.7 (fino a 3.7.1 incluso)

La segmentazione si basa sul paging ma fornisce in pratica un numero arbitrario di address space. Questo aiuta quando si hanno segmenti che variano molto di dimensione, e aiuta a condividere e proteggere meglio la condivisione di segmenti di memoria. Il problema è che il programmatore deve attivamente dividere i propri programmi.

Capitolo 4

Nelle macchine c'è bisogno di un supporto di memorizzazione che sopravviva allo spegnimento della macchina. Deve essere largo, e accessibile da più processi allo stesso momento. In un disco abbiamo la possibilità, essenzialmente, di fare operazioni di lettura e scrittura di un blocco. Naturalmente sorgono diverse problematiche, in cui il sistema operativo ci deve aiutare. Come trovare le informazioni, come proteggerle, come sapere quali blocchi sono liberi. La parte del sistema operativo che si occupa di queste cose è il file system.

4.1 (fino a 4.1.6 incluso)

La nomenclatura segue delle regole dettate dall'implementazione degli header. Può essere case sensitive o meno, supportare caratteri speciali, diverse lunghezze di stringa, etc. In generale c'è sempre un'estensione, ma non è obbligatoria. In generale al sistema operativo non importa dell'estensione, è solo parte della nomenclatura, ed è più un qualcosa che riguarda i programmi.

Un file può essere strutturato in diversi modi: come sequenza di byte (unix, windows), per cui dar senso a quello che c'è dentro è premura dei programmi e non dell'OS, o come sequenza di record come avveniva nell'era delle punch card. Si può anche avere una struttura linkata ad albero, che può essere presente su framework usati per analisi dati.

Ci sono diversi tipi di file, quelli normali che contengono informazioni, le directory che sono file di sistema per mantenere la struttura del file system, e i character e block special files per modellare dispositivi di IO.

In generale i file normali possono essere ASCII o binari. Un eseguibile binario ha 5 sezioni, header, text, data, relocation bits, symbol table. L'header inizia con un magic number, che dice al sistema operativo che si tratta di un eseguibile, poi ci sono le dimensioni delle varie sezioni del file. Dopo ci sono text e data del programma, che vengono caricati in memoria e rilocati con i relocation bits. La symbol table è usata per il debugging.

Nei dischi magnetici i file vengono letti in modo sequenziale. Ad oggi esistono anche sistemi ad accesso randomico, che permettono di leggere direttamente da una posizione. Windows e linux usano seek, per settare la posizione e poi leggere in modo sequenziale.

I file possono contenere degli attributi o metadata, spesso uno di questi è protezione, poi ci sono propietario, lock flags, creation time, time of last change, current size, ed altri.

Le operazioni possibili nei file sono: create, delete, open, close, read, write, seek, get attribute, set attribute, rename.

4.2

La forma più semplice di sistema di directory è averne una sola che contiene tutti i file, ma di solito c'è una organizzazione gerarchica, descritta da un path. Il path può essere globale o locale, ovvero che parte da una working directory predeterminata. All'interno di ogni directory ci sono due puntatori "." e ".." che indicano rispettivamente la directory corrente e il genitore.

Le operazioni possibili sulle directory sono analoghe a quelle dei file: create, delete, open, close, read dir, rename, link e unlink (che lavorano con i riferimenti agli inode), get e set attribute. Esistono anche link simbolici, che indicano il path di un file.

4.3

Il layout del disco di un PC contiene prima di tutto un MBR Master Boot Record, che viene letto dal BIOS all'avvio e indica la partizione attiva. Nel secondo settore c'è la partition table che indica gli indirizzi di inizio e partenza delle partizioni. Le partizioni contengono prima di tutto un boot block che viene letto dal bios per caricare l'OS. Ci sarà poi un superblock che contiene informazioni sul file system, un sistema di gestione dello spazio libero, un array di Inode e la struttura gerarchica delle directory.

In unix, dopo il superblock ci sono dei cylinder group, che contengono una copia del super blocco, una inode map, una block bitmap, i blocchi di inode e i blocchi di dati.

Ci sono diversi metodi che si possono usare per implementare i file. L'allocazione contigua li mette uno dopo l'altro, dividendoli in blocchi di dimensione fissa. È veloce e comoda per medium sequenziali come dischi magnetici o cd, ma in generale crea frammentazione interna, esterna e se non ci sono blocchi di dimensione sufficiente si rischia di non poter inserire certi file.

Si può avere un'allocazione con lista linkata, in cui si indica in una tabella il blocco di inizio del file, e ogni blocco punta al successivo. Per leggere tutto il file non ci sono problemi, ma per leggere il blocco n bisogna scorrere n-1 blocchi. Crea frammentazione interna e overhead di spazio a causa dei puntatori.

La FAT File Allocation Table funziona come una lista linkata ma tutto sta in una sola tabella, quindi si possono seguire i puntatori (che puntano ad altri elementi in tabella) senza andare a vedere indirizzi lontani sul disco. Occupa comunque tanto spazio.

L'ultima implementazione è con gli inode, index node, che sono dei blocchi contenenti gli attributi del file e 8 blocchi che puntano a disk block contenenti i dati. In generale l'ultimo punta ad un altro inode che ci dà la possibilità di estendere la dimensione del file. Nello specifico in unix, gli ultimi 3 blocchi sono speciali. Il terzultimo contiene un singolo blocco indiretto, che è tutto composto da puntatori a blocchi. Il penultimo punta ad un inode che è riempito di puntatori ad inode che puntano a blocchi di dati, e l'ultimo come il penultimo ma con una tripla indirezione.

Per implementare una directory, dobbiamo in sostanza mappare il nome del file al blocco di dati o all'inode corrispondente. Le entry della directory possono essere tutte della stessa dimensione fornendo un limite al nome del file, oppure si può estendere in due modi: specificando la lunghezza prima del campo (ma causa problemi alla rimozione perché lascia un buco di dimensione variabile) o inserendo un puntatore ad un heap con tutti gli attributi.

Per condividere file fra directory e utenti vanno implementati i link, per cui due o più entry nelle directory possono puntare allo stesso Inode (che conterrà un contatore delle entry che lo puntano, e potrà essere sovrascritto soltanto quando questo sarà 0).

In UFS, ogni entry nella cartella contiene semplicemente il puntatore all'inode e un nome di dimensione fissa. Per scorrere la gerarchia di cartelle, si segue il puntatore di una cartella che ci porta ad un Inode che contiene le informazioni sulla cartella, e l'inode che contiene le varie entry.

Ci sono diverse varianti possibili a questo file system. Il Log Structured File System mantiene in memoria un log di tutte le operazioni eseguite, e ogni tot le scrive tutte insieme sul disco.

Un Journaled File System tiene traccia di tutte le operazioni che sta per fare prima di farle, così se c'è un crash del sistema si può andare a riapplicare ciò che non si è riuscito a fare.

In generale si hanno diversi file system presenti sulla stessa macchina. In windows sono indicati da una lettera di disco diversa, ma nei sistemi unix possono essere montati uno all'interno dell'altro, e si può passare da uno all'altro senza mai sapere che sono file system o dispositivi diversi. Questo perché c'è un virtual file system. Questo sistema ha una interfaccia superiore che accetta chiamate posix dai processi utente, mentre al disotto ci sono interfacce VFS che mandano effettivamente i comandi ai vari file system per eseguire le operazioni richieste. È una specie di layer di traduzione.

4.4 (fino alla parte di caching di 4.4.4 escluso 4.4.2 e "disk quota")

Ottimizzare l'utilizzo di spazio sul disco è importante. Un fattore da considerare è la dimensione dei blocchi di dati. Maggiore la dimensione, minore il numero di chiamate che si devono effettuare per operare su un file, e quindi maggiore la velocità di lettura/scrittura. Al contrario però, minore la dimensione dei blocchi e maggiore sarà lo spazio occupato in essi (quindi meno frammentazione interna).

I blocchi liberi si possono mappare con una free list o con una bitmap (come nello swapping o nei segmenti). La freelist è divisa in blocchi di blocchi in sostanza, per poterne portare in memoria un pezzo per volta. Di solito, si cerca di portare metà dei blocchetti che si potrebbero mettere teoricamente in memoria, così che quando cancelliamo dei file possiamo andare ad aggiungerli alla nostra mezza lista e minimizzare le volte in cui dobbiamo passare dal disco.

Il sistema operativo si deve occupare in alcuni casi di fare un'analisi di consistenza. Genera una bitmap dei free blocks, e poi partendo dall'Inode della root va a generare una bitmap dei blocchi in uso ricorsivamente. Quindi confronta questi due array e se c'è 1 in uno, e zero nell'altro per ogni casella, allora siamo a posto.

- 0 in entrambi: il blocco è mancante in seguito ad un crash, lo aggiungiamo alla freelist
- 2 nella freelist: sarà sicuramente libero, mettiamo 1 nella freelist
- 2 negli occupati: bisogna ricopiarne uno in un blocco libero e informare l'utente che sceglierà cosa farne.

La cache è forse l'aspetto più importante da considerare per avere performance decenti. Si tiene una lista doppiamente linkata dei blocchi utilizzati, che saranno anche puntati tra di loro come collision list di una hash table (per controllare che un file sia o meno in cache velocemente). Quando un file non è in cache, lo si mette in cache (utilizzando algoritmi come quelli di page replacement) e si fanno le successive operazioni da lì.

Capitolo 5

5.1

I device di input possono essere a blocchi, leggono e scrivono un insieme di byte alla volta e hanno un buffer, o a caratteri che leggono e scrivono un solo carattere per volta (tipo una tastiera).

Il device controller è un pezzo di hardware che si occupa di operare i dispositivi di IO, convertendo le stream seriali in qualcosa di comprensibile per la cpu, e arbitrando la comunicazione tra la cpu - bus e perifierica.

Il primo tipo di IO che viene in mente è quello programmato, nel senso che è gestito dal software. In pratica il processo utente passa una stringa al kernel e gli dice di stampare. Il kernel blocca tutto e manda un carattere per volta in stampa. Era così sul vecchio DOS, ma adesso non è più desiderabile.

Oggi è più comune il memory mapped IO, per cui tutto l'IO è mappato in memoria, a volte nello stesso spazio di indirizzamento del processo e a volte in uno spazio diverso, accessibile tramite delle porte. Al giorno d'oggi, in entrambi i modi.

In pratica il sistema operativo mette a disposizione delle zone di memoria come registri o buffer delle periferiche. Ad esempio, la zona 1000 può essere adibita a buffer della stampante, e tutti i dati che ci vengono scritti passano in output.

Per non occupare il bus principale con operazioni di IO tra memoria e periferiche, le operazioni tra CPU e memoria hanno un bus specializzato nei sistemi moderni che si chiama Primary Memory Bus

Un DMA controller (Direct Memory Access) si occupa svolgere le richieste di IO per conto della cpu. Quindi la cpu manda una richiesta alla DMA, dicendo dove sono i dati e quante e quali operazioni ci sono da fare. Il DMA controller manda le istruzioni al controller della periferica, dicendogli di mettere i risultati nella memoria principale. Il controller della periferica, una volta terminate le sue operazioni, manda un ACK alla DMA, che manderà un'interrupt alla CPU.

Ci sono due modalità di prendere controllo del bus, in burst mandando tutte le informazioni insieme, e cycle stealing che aspetta che ci siano cicli liberi.
Gli interrupt che sono diretti alla CPU non vanno mai direttamente in cpu, ma passano per un interrupt controller. Questo raccoglie i segnali delle periferiche in un piccolo buffer interno, e li manda alla cpu in ordine senza sovrapporli, attendendo poi un ack dalla cpu prima di rimandarlo in caso.
È anche in grado di aspettare che la cpu sia pronta ad accettare un interrupt, ovvero quando ha completato la fase di execute della istruzione corrente in modo da non interromperla a metà. Il problema principale è che ci sono diversi flussi nelle cpu moderne, per cui più istruzioni potrebbero essere a diverse percentuali di completamento allo stesso momento. Si dice che la cpu si trova in uno stato preciso quando il PC è noto e quando tutte le istruzioni correnti sono eseguite e quelle future non ancora fetchate. In alcuni casi si può segnalare la cpu dicendo di mettersi in uno stato preciso, o si hanno dei registri che indicano il completamento delle varie istruzioni, ma nella maggior parte dei casi il sistema operativo riceve l'interrupt e poi si deve arrangiare per ricostruire quello che era prima in esecuzione.

5.2 (escluse le parti in pseudo-codice)

Il software di IO deve essere indipendente dal dispositivo, e avere nomenclatura uniforme. Il sistema di gestione dell'IO è stratificato, con al livello più basso l'hardware che al più raccoglie tutto in un buffer e solleva un interrupt. L'SO esegue dall'interrupt vector una procedura apposita per la classe di dispositivi, che si interfaccia con il device driver della periferica. I driver parlano con le interfaccie di sistema, che sono device indipendant.

5.3 (fino a 5.3.3 incluso)

Ma cosa succede nello specifico quando arriva un'interrupt?

1. L'hardware salva il PC e qualche registro della cpu, poi parte una procedura assembly per salvare il resto sullo stack corrente.
2. Una procedura apposita costruisce il contesto per l'interrupt handler e un nuovo stack.
3. Si manda un ACK all'interrupt controller
4. Si capisce dal bus quale periferica ha causato un'interrupt e si esegue la ISR appropriata dal vettore di interrupt.
5. Si sceglie il prossimo processo da eseguire, ma soltanto se ce ne è un altro che in risposta all'interrupt è passato a pronto e ha una priorità superiore a quello che era precedentemente in esecuzione.
6. Si crea il contesto, si caricano i registri, PC, psw e si esegue il processo.

I device driver risiedono nel kernel, ma sono spesso modulari.
Solitamente si ha sempre qualche tipo di buffer per i dispositivi di IO, e solitamente anche più di 1, a diversi livelli. Bisogna evitare che un buffer si riempia facendoci perdere informazioni, quindi a di solito ci sono due buffer nel kernel, quando uno si riempie si usa l'altro e nel frattempo quello pieno manda i dati ad un buffer user space.

Capitolo 6

Un deadlock è una situazione in cui due o più processi hanno acquisito delle risorse che non sono disposti a rilasciare prima di acquisire le risorse in possesso degli altri processi (che a loro volta le tengono strette).

6.1

Le risorse possono essere non prelazionabili o prelazionabili. Nel primo caso non possiamo toglierle al processo che le sta usando (una di queste potrebbe essere un masterizzatore blue ray in corso di scrittura), nel secondo invece possiamo toglierle anche forzatamente, come per esempio la memoria.
L'accesso ad una risorsa può essere inteso come un down su un semaforo (o mutex) associato alla risorsa in questione. Finchè ogni processo a una sua risorsa non ci sono problemi, ma quando se ne prendono più per volta bisogna stare attenti anche all'ordine di acquisizione.
Se processo 1 prende in questo modo

```
Down risorsa1
Down risorsa2
Up risorsa2
Up risorsa1
```

E processo 2 in questo

```
Down risorsa2
Down risorsa1
Up risorsa1
Up risorsa2
```

È chiaro che si possono creare dei deadlock, e che quindi bisogna prestare attenzione all'ordine di acquisizione.

6.2

La definizione formale di deadlock è *Un set di processi è in deadlock se ciascun processo nel set è bloccato aspettando un evento che solo un altro processo nel set può causare.*
Ci sono 4 condizioni che si devono verificare perché ci sia un deadlock:

1. Ci deve essere mutua esclusione
2. Ci deve essere hold and wait, ovvero che si può prendere una risorsa, e richiederne un'altra senza rilasciare la corrente.
3. Nessuna prelazione, se ci fosse potremmo togliere le risorse ai processi che non le rilasciano volontariamente
4. Attesa circolare

La quarta è visibile bene modellando la richiesta e il possedimento delle risorse con un grafo orientato. In questo le risorse sono quadrati e i processi cerchi. Un arco da processo a risorsa indica che il processo è bloccato in attesa di tale risorsa, mentre un arco da risorsa a processo indica che la risorsa è in possesso del processo. Quando si vanno a creare dei cicli si ha un deadlock.
Ci sono 4 modi per occuparsi dei deadlock:

1. Ignorare il problema
2. Identificazione e risoluzione
3. Evitarli dinamicamente
4. Prevenzione, negando una delle 4 condizioni precedenti.

6.3

Ignorare il problema è un'opzione accettabile se ce ne sono pochi, tipo uno ogni 5 anni, allora ci si può permettere di fare un reset della macchina e andare avanti. Tanto in 5 anni crasherà per molte altre ragioni.

6.4

Identificazione e risoluzione. È Semplice per noi umani vedere un ciclo in un grafo, ma non è sempre così immediato. Innanzi tutto il metodo con il grafo è utile solo se abbiamo una risorsa per ogni tipo di risorsa, e poi ci serve un algoritmo.
C'è un algoritmo di dijkstra per trovare cicli nei grafi, che è in pratica un DFS con una coda controllata ad ogni aggiunta.

1. Per ogni nodo in N ripetere i passaggi successivi
2. Sia L una lista vuota, e tutti gli archi non marcati
3. Aggiungere il nodo corrente in fondo alla lista L, controllare che non sia in lista due volte. Se c'è il grafo contiene un ciclo e si termina.
4. Dal nodo corrente controllare se ci sono archi non marcati in uscita, in caso andare al 5 e altrimenti al 6
5. Prendere un arco non marcato a caso, seguirlo, settare il nodo in cui si è come corrente e andare a 3
6. Se è il nodo iniziale, non ci sono cicli e possiamo terminare. Altrimenti siamo in una strada senza uscita, si rimuove il nodo corrente dalla lista, si marca il nodo precedente come corrente e si torna allo step 4.

Quando abbiamo risorse multiple, le possiamo organizzare in due vettori, uno in cui ci sono tutte le risorse esistenti e uno in cui abbiamo solo quelle disponibili, quindi Ai ci dice quante risorse disponibili ci sono del tipo i.
A questo punto costruiamo due matrici in cui la riga indica il processo e la colonna la risorsa, la prima C indica quante risorse e di quale tipo sono attualmente in possesso di ciascun processo, mentre la matrice R ci indica quante sono richieste.
L'algoritmo per l'identificazione ci dice:

1. Controlla se esiste una riga di R <= di A
2. Se esiste, esegui le richieste del processo e quando termina aggiungi tutte le risorse che erano nella corrispondente riga C in A e cancella le richieste del processo.
3. Se non esiste una riga del genere siamo in un deadlock e terminiamo.

Per occuparci di un deadlock identificato possiamo prelazionare temporaneamente delle risorse, ma spesso non è possibile. Oppure possiamo fare un rollback, ma bisogna implementare un sistema di checkpoint periodici dei processi per cui il loro stato corrente viene scritto su un file. La terza opzione consiste nell'uccidere processi finchè non si sblocca la macchina.

6.5

Come si evita un deadlock? Gli algoritmi per farlo si basano sul concetto di safe state, per cui esiste una sequenza di allocazione delle risorse che permette a ciascun processo nel set di terminare.
L'algoritmo del banchiere considera se ci sono abbastanza risorse libere per completare l'esecuzione di un processo, prima di assegnargliele. Altrimenti ne cerca un altro in modo da liberare qualche risorsa.
La generalizzazione di questo algoritmo su risorse multiple è uguale all'algoritmo di identificazione su risorse multiple in pratica.

1. Cerca una riga di R <= A, se non esiste andremo a finire in un deadlock.
2. Esegui il processo della riga trovata, aggiungi le risorse di C ad A
3. Ripeti i passi 1 e 2 finchè tutti i processi non sono terminati.

6.6

Per prevenire un deadlock si possono attaccare le 4 condizioni che lo causano.
La mutua esclusione si può eliminare con un processo di buffer, tipo uno spooler di stampa. Gli si mandano tutte le richieste e non c'è bisogno accedere esclusivamente alla stampante perché tanto lo fa solo il buffer.
L'hold and wait si risolve rilasciando le risorse già allocate quando non si possono acquisire quelle che si hanno già, oppure chiedendole tutte insieme.
Per quanto riguarda la prelazione, basta rendere le risorse prelazionabili ma appunto non è sempre possibile.
Per la circular wait si potrebbero numerare le risorse su scala globale, e quindi obbligare i processi ad allocarle in ordine numerico.

6.7 (escluso 6.7.2)

Two Phase Locking è una tecnica usata sui database per cui in una prima fase si richiedono i lock ad una serie di risorse, ma se non si può averli tutti le si rilascia e si ritenta in un secondo momento. Nella seconda fase le risorse sono effettivamente lockate e si fanno le operazioni.
Un livelock è un deadlock che non è effettivamente bloccato, in cui si usano dei trylock e si ritenta ogni tot ma non si va da nessuna parte.
La starvation è quella situazione in cui si usa un sistema di priorità che causa ad alcuni processi un'attesa infinita (si risolve con fifo).

Thread e mutua esclusione in go (su slide)

Anomalia di Belady e stack algorithms (su slide)

La proprietà di inclusione dice che l'insieme delle pagine in memoria all'istante t con m frame è sempre un sottoinsieme delle pagine in memoria all'istante t con m+1 frame. Gli algoritmi di rimpiazzo che soddisfano questa proprietà non sono soggetti all'anomalia di belady.
L'anomalia di belady è una particolare situazione in cui algoritmi di rimpiazzo causano più page fault con un numero maggiore di frame disponibili. Accade spesso con richieste di pagine più o meno circolari che hanno un "periodo" più grande del numero di frame.

Logical Volume manager (su slide + documento esterno)

Un LVM prende le partizioni dei volumi fisici, divise in blocchi chiamati Physical extent, e li raggruppa in gruppi chiamati Volume groups. L'unità di un volume group è il group extent. Dai group extent si costruiscono dei volumi logici, di unità logical extent. Sui volumi logici si possono costruire dei file system.
Si possono creare degli snapshot dei volumi logici, scrivere tutte le operazioni successive in un journal e poi usarlo per fare un backup.
I group extent possono essere mappati in concatenazione sui vari dischi disponibili, oppure stripati. Questo sistema è utile per i NAS. Si può anche costruire un file system sulla rete.

Cloud computing (su slide + documento esterno)

Il cloud funziona essenzialmente collassando più server virtualizzati all'interno di una sola macchina fisica. Questo comporta che si debba mantenere i sistemi operativi virtualizzati ma anche quello ospite, e crea un single point of failure.
L'idea generale è che un calcolatore consuma energia, va mantenuto e raffreddato e prima o poi andrà cambiato. Quindi a qualcuno è venuto in mente di vendere un "equivalente" on demand.
Prima del cloud si faceva housing o hosting.
Considerando la dinamica temporale di un servizio, ci si rende conto che la maggior parte delle risorse vengono sprecate una volta che l'interesse iniziale scema. Un sistema cloud può adattarsi e fornire soltanto le risorse utili in un determinato momento.
In generale il cloud computing è un modello di accesso tramite rete, on demand, ad una serie di risorse computazionali condivise, che possono essere configurate, create e gestite velocemente e semplicemente.
Software as a Service indica i servizi tipo g suite, in cui si vende solo un pacchetto di applicativi da usare con un abbonamento.
Platform as a service indica il noleggio di un webserver o qualcosa di equivalente, on demand
Mentre infrastructure as a service offre proprio una macchina virtualizzata completa.
Il cloud è quindi scalabile, elastico, richiede poca manutenzione, è sempre disponibile e accessibile ovunque.
Può essere pubblico, privato, ibrido, o comunitario.
Il problema è che non sempre taglia i costi e avere tutto in rete è un overhead considerevole, specialmente se si fanno trasferimenti di dati ingenti. In italia poi non abbiamo proprio connessioni Gigabit ovunque ecco. Alle banche piace perché taglia i costi dell'acquisto delle macchine.

Supporto a sistemi multimediali (su slide)

Il dato multimediale è soft real time, e segue delle dinamiche tutte sue (le perdite video vengono tollerate molto meglio delle perdite audio). Nasce come analogico e occupa un sacco di spazio.
In generale le reti non sono perfette, ed è complicato passarci dati multimediali, grandi come sono.
Le informazioni multimediali in un sistema operativo sono digitali (si trovano su un file), occupano tanto spazio ma possono essere compresse. Il sistema operativo deve poter garantire la fruizione real time, e le classiche funzionalità di un registratore.
La codifica consiste nel stabilire una corrispondenza tra un'informazione analogica e una digitale. Ha anche il vantaggio di proteggere il contenuto per cifratura o con l'inserimento di watermark. Inoltre può risolvere errori di trasmissione a volte. Non vuol dire compressione, che è soltanto un effetto collaterale di una codifica ben strutturata.
Rappresentando ogni frame di un video come un raster ci accorgiamo subito che occupiamo tantissimo spazio.
La codifica di huffman permette di creare una corrispondenza tra informazioni da rappresentare e stringhe più brevi, riducendo lo spazio occupato in maniera lossless.
Lo scartare invece informazioni inutili è una compressione lossy.
La ridondanza spaziale è ad esempio una zona di colore omogeneo. La ridondanza temporale indica ad esempio una zona che non si evolve enormemente nel tempo.
La compressione deve anche considerare la fisiologia e psicologia degli esseri umani, infatti dopo una certa soglia ci accorgiamo di più dei cambiamenti.
Anche il mezzo di fruizione è importante, è inutile mandare un video a colori su una tv in bianco e nero.
L'mpeg è uno standard con delle regole di compressione spaziale e temporale pensate in maniera tale da non disturbare troppo la percezione dell'utente medio.
Ci sono diversi tipi di standard mpeg.
È una codifica asimmetrica, per cui lo standard detta solo le linee guida per la creazione del file, e per cui il codificatore svolgerà la maggior parte del lavoro, lasciando solo operazioni semplici nella parte di decodifica. Con lo stesso standard si possono avere codificatori e decodificatori più efficienti.
L'mpeg costruisce un video codificando una sequenza di fotogrammi in modo da ridurre il più possibile la ridondanza.
Per la ridondanza spaziale in pratica si mantengono i contorni e si esprimono in differenza dal primo elemento della matrice, per poi linearizzare il tutto.
Per la ridondanza temporale dividiamo i frame in Indipendenti e P che predicono. Si creano poi dei vettori di movimento nello schermo rappresentati in matrice.
Ci sono anche frame B che fanno riferimento sia avanti che indietro, prendendo in prestito dei rettangoli.
I P frame sono codificati come la differenza con il frame I che li precede, con vettori di movimento, mentre i B hanno anche differenza con il frame successivo.
La codifica e il contenitore sono due concetti differenti.

Si parla di streaming ogni volta che del contenuto viene utilizzato prima che sia arrivato per intero. Il contenuto è trasmesso in maniera continua e asincrona.
Il video viene ricevuto a velocità variabile, quindi c'è bisogno di un buffer per poterlo riprodurre ad un bitrate costante.
- Il buffer non deve essere troppo piccolo per non perdersi dati, overrun.
- Se è troppo grande si sprecano risorse e si introducono ritardi, underrun.

Se perdiamo dei dati possiamo introdurre rumore.
Per andare avanti e indietro dobbiamo saltare ad un frame i vicino e buttare via il buffer. Per farlo in modo efficiente servono uno scheduler e un file system specializzati.
I processi realtime sono segnati come periodici, quindi tipo 20 volte al secondo utilizzo sempre lo stesso burst di cpu e il SO può venirmi incontro. Le scadenze sono ricorrenti e note a priori.
RMS Rate Monotonic Scheduling è una politica di scheduling apposita per processi periodici. È a priorità statica, assegnata in modo inversamente proporzionale al periodo.
EDS Earliest Deadline First Prevede priorità inversamente proporzionale al tempo rimanente al deadline.
La lettura sequenziale è il modo migliore per fruire contenuti multimediali. Si può fare un'indicizzazione per trovare facilmente i frame I e l'audio corrispondente.