

Appunti di

Programmazione di Sistema

G. Cabodi

Sommario

1. Gestione Memoria	3
2. Memoria Virtuale	11
3. Introduzione a OS161	29
4. OS161 – Address Space & Memory Management	47
5. OS161 – Sincronizzazione di primitive	58
6. Memorie di Massa	70
7. Interfaccia File System	77
8. Implementazione File System	85
9. OS161 – UserProcess	98
10. Sistemi di I/O	110
11. OS161 – I/O	128
12. OS161 – File	130

1. Gestione Memoria

Il sistema di elaborazione non è limitato a un solo processo, ma può avere più processi attivi.

Un programma viene mandato in esecuzione portando l'eseguibile dal disco in memoria (memoria RAM, memoria volatile, memoria che perde informazione quando si spegne il PC).

La CPU come memorie su cui lavorare ha i registri, che sono dentro la CPU ma sono pochi, e la memoria RAM che sono fuori ma hanno misure ragionevoli.

Per accedere alla memoria un microprocessore ha bisogno di utilizzare indirizzi e dati, mentre l'accesso ai registri è rapido (all'interno di un colpo di clock), l'accesso alla memoria RAM spesso ha altre tempistiche.

Cache è una memoria intermedia che cerca di avere tempi di accesso molto più vicini al microprocessore e dimensioni intermedie tra RAM e registri.

Sia in modo volontario che involontario, si affronta il problema di fare in modo che un processo non possa interferire sulla memoria di un altro processo e sulla memoria del sistema.

Meccanismo di creazione di un programma eseguibile: a partire da file sorgenti (file .c) è compito del compilatore, che si distingue in due fasi quella del preprocessore e quella del compilatore vero e proprio, di generare per ogni file sorgente un file oggetto. I tre file oggetto vengono messi insieme per creare un unico eseguibile, il linker oltre a unire i file oggetto, aggiunge all'eseguibile la possibilità di chiamare funzioni di libreria. (grafico slide)

In questo contesto, un'eseguibile generato in questo modo verrà utilizzato per far partire un processo e in un sistema con multiprogrammazione succede che nella RAM, ad un certo momento, saranno presenti più processi insieme al sistema operativo, o meglio al kernel.

Nel momento in cui in un sistema in operazione fossero presenti il sistema operativo e 3 processi e una parte della RAM libera, ciò che bisogna garantire in termini di protezione, che il singolo processo acceda solo alla RAM che gli è riservata e non vada a sovrascrivere o leggere RAM che non gli compete. Mentre un processo è in esecuzione è importante che ci siano nella CPU due registri che stanno a indicare l'indirizzo di base e il limite. Nell'esempio `base= 300040` e `limit=420940-300040`. (immagine slide)

Collocare un processo a indirizzi arbitrari nella RAM non è indolore, perché se io posizionassi i due processi in modo consecutivo nella RAM, il primo processo non avrebbe problemi dal momento che i registri corrisponderebbero, il secondo invece il comando `JMP 28` non lo porterebbe alla sua istruzione `CMP` ma all'`ADD` del primo processo.

Quindi il `base register` e il `limit register` oltre a indicare dove inizia e finisce un processo, si possono anche usare per ricalcolare gli indirizzi effettivi (nell'esempio `JMP 28` è da considerare come il 28 esimo indirizzo a partire dal `base register`).

Bind -> associare indirizzo effettivo a uno logico, connettere due indirizzi facendo in modo che si corrispondano.

Il problema è che se un eseguibile che viene portato da disco in memoria non ha un meccanismo di gestione automatizzata di associazione tra indirizzi, non può che richiedere di conoscere già a che indirizzo sarà caricato, la cosa più semplice è pensare che un programma debba iniziare all'indirizzo 0, quindi nei sistemi più semplici l'unico sistema è usare indirizzi assoluti.

Il che implica che in un sistema un processo utente non possa che partire dall'indirizzo 0, questo è un vincolo, si potrebbe cambiare però è un sistema fisso. Sui PC non vanno andare molto bene, può andare bene in sistemi embedded semplici dove girano pochi programmi e gli schemi sono pressoché fissi.

Binding significa dire a quale indirizzo vero corrisponde indirizzo che era ancora provvisorio.

In un codice sorgente gli indirizzi sono solitamente simbolici, una volta che il compilatore traduce il programma in codice assembler metterà degli indirizzi, se mette degli indirizzi rilocabili (come per esempio 14 byte dall'inizio di questo modulo). Questo non è ancora l'indirizzo, il completamento verrà fatto dal Linker o dal Loader, se è fatto dal primo nell'eseguibile ci saranno già gli indirizzi finali, nel secondo caso mentre il programma viene caricato in memoria si decidono gli indirizzi. Si passa da indirizzo rilocabile a indirizzo assoluto.

Ogni operazione di bind mappa un indirizzo provvisorio/incompleto in un indirizzo definito.

Nel momento in cui si parla di istruzioni e dati da caricare in RAM possiamo identificare 3 fasi in cui l'operazione di Binding avviene:

- Fase di Compilazione (generazione eseguibile): dire che il binding viene fatto in compile time significa dire che bisogna conoscere a quale indirizzo sarà eseguito un certo programma e nell'eseguibile verranno inseriti indirizzi assoluti;
- Fase di Load (caricamento eseguibile in memoria): binding in load time significa che il codice eseguibile dev'essere rilocabile, significa dire che mentre viene collocato in RAM l'eseguibile viene completato e si mettono gli indirizzi giusti perché si conosce dove vengono collocate le varie parti del programma.
- Fase di Esecuzione: binding fatto in esecuzione non significa un binding fatto una tantum all'inizio dell'esecuzione della prima istruzione, ma significa binding differito fino all'esecuzione di un certo modulo/di una certa funzione.

(schema slide)

L'unica soluzione per permettere di differire l'associazione di un indirizzo mediante binding all'indirizzo in memoria è quello di far sì che in fase di esecuzione ci sia una traduzione tra quello che è un indirizzo logico (quello che vuole vedere il programma) e un indirizzo fisico (andrebbe usato da CPU per accedere alla RAM). INDIRIZZO LOGICO -> indirizzo generato dalla CPU, ovvero che è scritto dentro al codice assembler, al codice eseguibile. Questo indirizzo è un numero, ma non è ancora quello vero che verrà scritto nell'address bus.

INDIRIZZO FISICO -> indirizzo visto dalla memoria

Quindi la CPU genera l'indirizzo logico e la memoria riceve l'indirizzo fisico, questo significa che tra la CPU e la memoria, nel passaggio sull' address bus c'è un meccanismo di traduzione, chiamato traduzione da indirizzo logico a indirizzo fisico.

Quindi la soluzione per fare in modo che si possa fare binding in esecuzione, è che questo binding sia fatto in modo automatico e sia di fatto realizzato in modo automatico dalla Memory-Management Unit (MMU), dispositivo hardware, che sta a bordo della CPU, che è stato realizzato per tradurre l'indirizzo logico in indirizzo fisico. Il meccanismo più semplice per realizzare questo è quello di usare il relocation register, che prima chiamavamo base register, che verrà sommato in modo automatico a livello hardware all'indirizzo logico. Il valore che si trova nel relocation register, ad ogni accesso in memoria, verrà sommato tramite un addizionatore che sta dentro alla MMU all'indirizzo logico.

Se nel programma c'è un indirizzo logico questo diventerà fisico dopo aver sommato il relocation register.

Queste operazioni di binding e di traduzione da indirizzo logico a fisico fatta automaticamente, possono essere usate per realizzare vari schemi di caricamento ed esecuzione del programma che possiamo definire Dynamic Loading e Dynamic Linking.

Questi due termini vengono spesso confusi, sono usati in modo quasi interscambiabile ma hanno un significato diverso.

È possibile che un programma abbia solo uno dei due, nessuno dei due o entrambi.

Dynamic Loading -> significa caricamento dinamico, significa che un programma intero possa essere caricato in memoria a pezzettini uno alla volta. Quindi se un programma ha 10 funzioni, supponiamo che in un'esecuzione di queste funzioni ne vengano usate solo due. Loading dinamico -> l'idea di portare in memoria solo ciò che serve. Una routine non viene caricata fino a quando non viene chiamata.

Questo ha un unico obiettivo: risparmiare spazio in memoria.

Per fare ciò il binding dev'essere fatto in modo dinamico e in fase di esecuzione, e ogni modulo sarà caricato e collegato agli indirizzi veri soltanto quando serve.

Il concetto di dynamic loading significa: il programmatore ha a disposizione delle funzioni per caricare dinamicamente una funzione. Quindi se la funzione è già stata caricata vai avanti e chiamala, se no caricala e poi eseguila. È un problema gestito dal programmatore che scriverà istruzioni esplicite di caricamento di funzioni.

Dynamic Linking si contrappone allo Static Linking.

Link significa collega, collegare pezzi di un programma decidendo gli indirizzi. Il linking è l'operazione in cui si decidono gli indirizzi che vengono usati da un modulo e che sono di oggetti non interni a quel modulo.

Static Linking se il linker e il loader decidono tutti gli indirizzi, quando un programma parte gli indirizzi sono a posto.

Dynamic Linking quando si associa un indirizzo a un'entità, tipicamente una funzione, solo quando questa viene eseguita. Questo viene fatto mediante un automatismo, detto stub, piccolo pezzo di codice messo nell'eseguibile che sostituisce la funzione vera.

Talvolta il dynamic linking viene usato per funzioni di libreria che sono caricate dinamicamente e l'indirizzo viene riconosciuto e associato alle istruzioni opportune solo nella fase di load. Questo viene fatto nelle librerie shared.

Prima strategia per la collocazione in memoria di più processi: Allocazione Contigua -> la memoria viene partizionata in intervalli di indirizzi in cui uno viene assegnato al sistema operativo e gli altri vengono assegnati a processi utenti contemporaneamente attivi (non significa tutti in esecuzione ma significa pronti all'esecuzione).

Per fare questo è necessario che ci siano meccanismi di riallocazione, quindi è necessario fare in modo che il processo possa non conoscere all'inizio gli indirizzi in cui sarà collocato, ma possa essere spostato a una parte di memoria che dipende dall'esecuzione.

Per far questo il supporto più semplice è avere base register e limit register gestiti direttamente dalla MMU. Tra l'indirizzo logico e fisico c'è un pezzo della MMU che fa un paio di operazioni. La prima operazione è una verifica di correttezza e una protezione sull'indirizzo per evitare che sia troppo grande, questo blocco dice che se l'indirizzo va bene si esegue, altrimenti si attiva una trap di gestione dell'errore, la vera operazione di traduzione è la somma con il relocated register.

La configurazione finale è quella seguente (foto slide), che illustra la possibilità che dinamicamente, durante l'esecuzione, si passi da una soluzione in cui in memoria ci possano essere, oltre al sistema operativo, tre processi e la memoria sia effettivamente piena, a una seconda fase in cui il processo 8 termina lasciando una partizione libera, in questa partizione può essere allocato il processo 9 che richiede meno spazio del processo 8, e poi termina anche il processo 5.

In questa fase dinamica il grado di multiprogrammazione (quanti processi sono contemporaneamente attivi) è limitato dal numero di partizioni.

Tuttavia, le partizioni possono avere dimensioni variabili e questo ha una conseguenza -> i buchi (hole) possono essere riutilizzati ma non necessariamente in modo completo. L'alternanza di allocazioni e deallocazioni può portare a un'alternanza di pieno/vuoto -> frammentazione.

Con quale strategia si va a cercare memoria disponibile quando bisogna allocare un nuovo processo?

In caso di un'unica partizione il processo potrebbe essere allocato all'inizio, alla fine o in una posizione intermedia, la scelta è di posizionarlo all'inizio.

Ci sono 3 possibili soluzioni:

- First-fit: se ci fossero più partizioni libere, prenderei la prima che va bene, la prima partizione che è grande abbastanza. Non sarà ottimale in termini di politica di allocazione, residui che lascerà in termini di residui di partizione, ma sarà ottimale in termini di velocità
- Best-fit: spende più tempo a cercare la partizione perché vuole trovare la migliore, quindi sceglierà la partizione più piccola tra quelle che sono abbastanza grandi. L'obiettivo è di lasciare un residuo di partizione il più piccolo possibile.
- Worst-fit: allocare la partizione più grande, scegliendola il residuo di partizione potrebbe essere ancora sufficientemente grande da poter essere allocato a un altro processo.

Qual è la soluzione migliore? First-fit e Best-fit sono le soluzioni migliori, rispetto alla worst-fit, in termini di velocità e utilizzo della memoria.

Siccome dinamicamente vengono creati e distrutti dei processi e alcuni di essi occupa e rilascia partizioni contigue di memoria di dimensione variabile, quello che succede in memoria è che si andranno a creare alternanze pieno/vuoto di dimensione diverse -> FRAMMENTAZIONE

Frammentazione Esterna: i frammenti di memoria libera stanno fuori rispetto alla memoria allocata ai processi.

Frammentazione Interna: quando si decide quanta memoria assegnare a un processo, è possibile che a un processo venga allocata un po' più di memoria di quella necessaria, la differenza di memoria viene detta frammentazione interna.

Statisticamente parlando, se a un certo punto abbiamo N blocchi allocati, è realistico che ci siano $0,5N$ blocchi persi per frammentazione, questo dice che circa 1/3 della memoria possa essere inutilizzabile.

La frammentazione ha due fondamentali problemi:

- Una parte della memoria è inutilizzabile
- La memoria è inutilizzabile perché fatta da pezzettini piccoli, che potrebbero essere troppo piccoli per soddisfare la dimensione minima richiesta dal processo. Quando una memoria è tanto frammentata sarebbe opportuno riuscire a deframmentare o compattare i frammenti in qualche modo, quindi sarebbe opportuno fare un'operazione di spostamento dei processi in memoria per far sì che le cose diventassero più gestibili. Per far ciò è necessario che la rilocazione non sia solo una cosa risolta in fase di Load, ma dev'essere possibile spostare un processo mentre è in esecuzione (non significa che CPU stia eseguendo un'istruzione del processo, non in stato di READY). Compattazione possibile solo se la relocazione è dinamica.

Questo genera il problema I/O: un processo potrebbe non essere in stato di READY, ma di WAIT (non è in esecuzione, è in memoria e sta aspettando il completamento di una I/O, che significa che una struttura dati del processo è sorgente o destinazione di una I/O, quindi se spostiamo in quel momento il processo questa variabile da cui un'operazione di I/O sta prendendo/mandando dati avrebbe dei problemi).

Le soluzioni sono 2:

- o Accettare che esistano operazioni di I/O che lavorano su dati di processi in stato di wait ma bloccare questi processi rispetto allo spostamento, vietare lo spostamento quando è in corso le operazioni di I/O
- o Fare in modo che non ci siano operazioni di I/O che sono svolte con sorgente/destinazione dati di processi. Per far questo si usano buffer di kernel, si fa in modo che passaggio da una struttura dati di un processo a un dispositivo di I/O sia fatto con un'intermediazione in aree dati di kernel.

Il problema della frammentazione non è solo della RAM ma anche del backing store (-> consideriamolo come un disco, disco del file system).

La compattazione ha un costo elevato e non è detto che sia accettabile o realistico.

La paginazione è motivata non solo dal costo della compattazione, la frammentazione è solo una delle motivazioni per cui si è cercato uno schema di allocazione più complicato ma che permette più flessibilità ed efficienza.

L'idea di allocare ai processi spazio non contiguo (vera novità!), a un processo vengono allocati indirizzi spezzettati nelle cosiddette pagine. Quello che si vuole ottenere è che non ci sarà più frammentazione esterna, non ci sarà più il problema delle dimensioni variabili delle partizioni allocate.

La soluzione è quella di dividere la memoria fisica in blocchi di dimensione fissa, detti frames, dimensione che sarà potenza di 2. Memoria vista come vettore di frames.

PAGINA-> parte della memoria logica

FRAME-> parte della memoria fisica

La paginazione si basa sull'idea di suddividere la RAM in frame di dimensione fissa.

Con l'allocazione contigua la memoria logica e fisica coincideva come rappresentazione.

Ora lo spazio di indirizzamento logico resta contiguo e viene suddiviso in pagine, che verranno ognuna allocata in un frame della memoria fisica, per fare questo occorrerà gestione dei free frame.

Per far funzionare un programma che ha bisogno di N pagine, serviranno N frames liberi.

Il vero problema è che siccome lo spazio degli indirizzi logici è contiguo mentre lo spazio degli indirizzi fisici non è contiguo non è possibile basare la traduzione logico fisica in una semplice somma, ma sarà necessaria una tabella.

A questo punto sarà più rilevante il problema della frammentazione interna perché quello che succede è che a un processo sarà sempre allocata come dimensione complessiva di RAM un multiplo di 1 frame.

La traduzione da indirizzo logico a fisico viene fatta suddividendo l'indirizzo in due parti:

- Numero di pagina (p)
- Offset (d)

L'utilizzare una dimensione di frame (potenza di 2) permette di avere un risultato simile a quello che con numeri in basi 10 si ottiene considerando centinaia o migliaia e giocare su cifre basse e cifre alte. cioè se prendiamo un numero in base 10 e cerchiamo di raggruppare i numeri possibili sulle migliaia, le tre cifre basse rappresentano l'offset, ovvero lo spostamento del numero all'interno del migliaio a cui appartiene, le cifre alte servono a numerare le migliaia. Lo stesso vale per gli indirizzi logici. Se abbiamo un indirizzo logico di m bit di cui n bit sono dedicati al page offset e m-n sono dedicati al page number, quindi p indicheranno il numero di pagina e d indicherà l'offset all'interno di quella pagina.

Per sapere la pagina in quale frame è basta guardare la tabella delle pagine. (foto slide)

Un processo aveva bisogno di una certa dimensione, e questa è stata suddivisa in pagine con arrotondamento a un multiplo di pagine. (foto slide).

La frammentazione interna è il pezzo dell'ultima pagina non utilizzato.

Nel caso peggiore l'ultima pagina è stata allocata perché serviva 1 byte -> frammentazione = 1frame – 1 byte.

In media, la frammentazione interna è circa ½ dimensione del frame.

Quindi qual è la dimensione del frame desiderabile? Se guardiamo dal punto di vista della frammentazione, sono meglio frame piccoli. Ma se noi diminuiamo le dimensioni dei frame aumenterà la dimensione della tabella delle pagine. Normalmente le dimensioni delle pagine possono oscillare da pochi kbyte ai Mbyte.

In una riga della tabella delle pagine ci possono essere più informazioni:

- Page frame number: valore più importante, che occupa più spazio
- Informazione di protezione: una certa pagina può essere sola lettura o anche scritta
- Modifica rispetto all'ultimo caricamento in memoria
- Abilitazione o no al caching

Oltre a dire dove sta in RAM quella pagina possiamo associare informazioni comuni a ciò che è scritto in quella pagina.

La page table non sta nella CPU, sta nella memoria RAM e anche la tabella delle pagine che serve per accedere in RAM sta nella RAM, nella CPU stanno dei registri che dicono come accedere a un vettore, nella CPU ci sono due registri:

- Page-table base register (PTBR): puntatore alla page table, c'è l'indirizzo di inizio della page table
- Page-table length register (PTLR): indica la dimensione della page table

Quindi se per tradurre da logico a fisico bastava un passaggio interno alla CPU, in questo caso bisogna fare una lettura in RAM. Quindi per usare un indirizzo fisico, bisogna fare preventivamente una lettura in RAM per passare da indirizzo logico a indirizzo fisico. Per ogni accesso in memoria servono due accessi e questo crea un rallentamento di un fattore 2 della memoria.

Per risolvere questo problema si usa uno schema simile a quello usato più volte nelle RAM-> caching (si cerca di fare una replica più veloce di quella cosa).

Si fa in modo che la traduzione logico-fisica, prima di dover arrivare alla page table, passi da un'altra tabella che si trova nella CPU, la translation look-aside buffers (TLBs), anche chiamata memoria associativa, tabella che permette di evitare, se va bene, l'accesso in RAM.

Le TLBs è un vettore compatibile con quello che può stare nella CPU in cui non ci saranno molte corrispondenze pagina-frame ma ce ne stanno quante basta per fare in modo che quasi sempre andrà bene. La TLBs è un vettore di righe dove l'importante è la pagina e frame.

La differenza rispetto alla tabella delle pagine è che in quest'ultima il valore del frame viene scritto nella riga n (per esempio il frame 31 pagina 140, il frame 31 verrebbe scritto all'indice 140 della tabella), la TLBs è più compatta.

Il tutto si basa sul fatto che si spera che gli accessi in memoria abbiano notevole ripetitività.

Senza un supporto hardware specializzato, prima di poter dire che, per esempio, alla pagina 19 corrisponde il frame 50, occorrerebbe fare una scansione lineare costosa in questa tabella (fuori dalle scelte possibili). (scheda slide)

la TLB è una memoria associativa: se ci sono n righe, ognuna di queste ha un comparatore hardware interno. P viene inviato in parallelo a tutte le righe, ognuno di questi comparatori fa un confronto e al più uno di questi comparatori risponde trovato. Se uno di questi risponde trovato, c'è un TLB hit e via hardware viene inviata sull'uscita TLB il numero di frame f e si ha la traduzione logico fisica. Il trucco è che tutto questo è fatto all'interno della CPU, è come se fosse fatto in un circuito combinatorio, non si sprecano tempi di clock o si spreca pochissimo tempo.

Se non trovo corrispondenze, ho un TLB miss e si fa un accesso in memoria alla page table.

Il ragionamento da fare in termini di prestazioni è il seguente: occorre poter stimare la frequenza di successo della TLB.

Assumiamo che la TLB abbia una probabilità di successo dell'80%.

Supponiamo che l'accesso in memoria abbia un costo di 10 nanosecondi.

Supponiamo che la TLB risponda hit e il tempo di accesso alla TLB sia trascurabile e la corrispondenza logico fisica sia trovata in un unico accesso e quindi l'accesso in memoria ha un costo di 10 nanosecondi. Invece se la TLB non ha successo la traduzione logico fisica deve usare la tabella delle pagine e deve usare 2 accessi -> 20 nanosecondi.

Facendo la media dei vari accessi si può dire che l'Effective Access Time (EAT), ovvero il tempo effettivo di accesso, è dato da:

$$EAT = 0.80 \times 10 + 0.2 \times 20 = 12 \text{ nanosecondi}$$

Quindi con l'80% di successo della TLB abbiamo un rallentamento del 20% della memoria.

Se la percentuale di successo fosse del 99%, avremmo evitato di rallentare la memoria.

Un vantaggio della tabella delle pagine consiste nel fatto che, si può dal momento in cui per ogni pagina si mette una riga nella tabella delle pagine si può aggiungere al numero del frame altre informazioni, come per esempio informazioni di protezione, pagina valida/non valida, si potrà anche dire che una riga può essere protetta in lettura.

Il bit di validità può essere gestito inserendo di fianco al numero di frame un bit, chiamato valid-invalid bit e potrebbe servire nel momento in cui su 8 pagine solo 6 pagine fossero usate e 2 no.

Un altro vantaggio consiste nella possibilità di condividere pagine tra processi diversi. Questo è un vantaggio che indica risparmio di memoria, o meno occupazione di RAM.

Supponiamo che più processi usino delle funzioni di libreria, invece di replicare queste funzioni nello spazio virtuale e quindi poi nei frame assegnati ad ogni processo, vi si può condividere facendo in modo che ogni processo veda virtualmente delle pagine come fossero sue, ma dal punto di vista fisico queste pagine siano mappate sugli stessi frame.

Per far questo le pagine devono essere condivisibili, quindi devono essere in sola lettura. Funzioni di questo tipo si dicono rientranti.

Il fatto di condividere pagine tra processi diversi è molto simile alla prassi usata per far sì che dei threads di un processo condividano dei dati, solitamente mediante variabili globali. L'unica vera differenza è che i threads nel condividere variabili globali le usano per interagire, nelle pagine di codice condivise tra processi queste funzioni non sono fatte per far sì che i processi si parlino, vengono usate per occupare meno memoria.

Dei processi potrebbero usare pagine condivise per comunicare -> interprocess communication

Laddove processi diversi vogliono condividere funzioni di libreria, se queste funzioni hanno bisogno di dati anche globali, è bene che questi dati restino privati e non pensati per condivisione con altri processi. Per far questo si fa in modo che la condivisione non sia piena ma sia limitata alle pagine di codice rientrante, e le pagine di dati siano di fatto private (un processo non può modificare una variabile di un altro processo).

Supponiamo di avere indirizzi da 32 bit, dimensioni della pagina di 4 KB (2^{12} byte, questo significa i 12 bit bassi dell'indirizzo sono usati per l'offset o displacement all'interno della pagina, i rimanenti 20 bit sono il numero di pagina che fa parte dell'indirizzo logico). Quindi la tabella delle pagine dovrà avere un numero di righe pari al numero possibile di pagine (2^{20} circa 1 milione). Ogni riga dovrà avere un numero di bit sufficienti a contenere un numero di frame, che può essere rappresentato in un intervallo tra 16 e 32 bit.

Quindi ogni riga occupa 4 bytes -> la tabella delle righe occupa circa 4 MB per ogni processo.

La tabella delle pagine deve essere contigua in memoria, non deve essere paginata perché bisogna usare il numero di pagina come indice.

Proviamo a ipotizzare che 4MB siano molti, quindi proponiamo soluzioni nel caso in cui la tabella sia troppo grande:

- Hierarchical Page Tables: è un'idea che vuole ovviare il problema di troppa memoria contigua richiesta, nel senso che se mi servono 4MB, la tabella continua a richiederne 4 MB ma la chiede partizionata. L'obiettivo è realizzare la tabella delle pagine a 2 livelli: nel senso che quella che prima era la tabella delle pagine viene spezzata in sezioni e viene messa davanti un ulteriore tabella di pagine di primo livello che serve solo per individuare in quale delle tabelle delle pagine di 2 livello si va a cercare l'effettivo numero di frame. Un'indirizzo logico su 32 bit può essere visto come p (indice di pagina) e d (offset/displacement). Consideriamo p=22 e d=10. Supponiamo che p sia troppo grande, quindi scomponiamolo in p1=10 e p2=12. Con questa soluzione è possibile fare la traduzione da indirizzo logico a indirizzo fisico usando p1 per selezionare una riga all'interno di una tabella delle pagine esterna, in questa riga si prende l'indirizzo della tabella delle pagine di secondo livello, in questa si usa p2 per individuare il numero di frame e all'interno si usa il displacement per trovare l'indirizzo (anche la RAM non visualizza tutto). Il problema è che questa volta per fare un accesso in RAM non si fanno più due accessi ma 3.

Su indirizzi da 64 bit, e pagine di dimensione di 4KB (tabella delle pagine 2^{52} righe, ma non si usa). Troppo grande p=52 d=12, quindi divido p1=42, p2=10, ancora troppo grande p1=32 (2nd outer page), p2=10 (outer page), p3=10, con questo ultimo accesso invece di 3 accessi ne servono 4.

- Hashed Page Tables: come alternative alle tabelle gerarchiche, soprattutto nella situazione di tabelle grandi per affrontare il problema del tempo di accesso (quando TLB da miss), usare una tecnica di hashing potrebbe essere vantaggiosa perché riuscirebbe a raggiungere in un tempo ragionevole la pagina. E poi il motivo è che la tabella di hash potrebbe avere dimensioni più compatte. Possono esserci due tipi di tabella di hashing:
 - A ogni entry corrisponde un frame
 - A ogni entry corrisponde un cluster

Nella tabella di hash si rappresentano solo le pagine effettivamente utilizzate.

- Inverted Page Tables: cerca di rappresentare una tabella in cui invece di usare il numero di pagina come indice e il frame come contenuto, usa il frame come indice e il numero di pagina come contenuto. Quindi avendo questa strategia, invece di avere una tabella grande quanto lo spazio di indirizzamento del processo, avremo una tabella delle pagine proporzionale alla dimensione della RAM. Questo diminuisce la dimensione della tabella e ne crea una sola, ma pone problemi di velocità che possono eventualmente essere affrontati avendo una tabella di Hash e poi usando efficacemente la TLB. Il vettore è parallelo alla RAM. Mentre nella versione precedente ogni processo aveva la sua tabella delle pagine, in questo caso nella stessa tabella parallela alla RAM ci sono le pagine di più processi e quindi potrebbe esserci p 10 del processo 1 e un p 10 per un processo 5. Per distinguerle mettiamo pid per identificare le pagine dei vari processi. Lo svantaggio è che purtroppo non si ha accesso diretto alla riga che cerchiamo. Per rendere più veloce la ricerca

si aggiunge una tabella di hash in cui ci sono liste di adiacenza che hanno corrispondenza pagina virtuale-frame.

Oracole SPARC Solaris -> microprocessore su cui si basavano work station-> sistemi di elaborazione più potenti dei pc attuali

Microprocessore che è dotato del sistema operativo Solaris (simile a Unix), sistema a 64 bit, ci sono due tabelle di hash, una per il kernel e una per il processo utente. C'è una TLB che contiene entry di tabelle di traduzione dette TTE e c'è una cache di TTE che sta dentro un translation store buffer (TSB).

Swapping è una tecnica per la quale un processo può essere buttato fuori dalla memoria RAM e salvato su disco perché per un po' si decide di non farlo partecipare all'esecuzione sulla CPU. Ha a che vedere con lo scheduling della CPU.

Supponiamo di avere troppi processi, tali da saturare la memoria RAM, e vogliamo far posto per un nuovo processo, quindi prendiamo uno di quelli esistenti e lo buttiamo fuori quindi si fa roll out o roll in per quello che entra, spostando i frame di quel processo dalla RAM al backing store (-> partizione di disco adibita a questo tipo di lavoro).

Quando un processo che era stato spostato fuori deve essere rimesso dentro, deve essere rimesso esattamente dove stava? Dipende dal fatto che gli indirizzi siano o meno rilocabili. Se si fa Load Link dinamico è possibile che un processo venga messo dove c'è spazio.

Unix, Linux e Windows hanno varianti di questa tecnica.

Fare swapping implica che la memoria è troppo carica, quindi è un po' un'ultima ratio.

Un context switch (cambiare il processo in esecuzione) ha un costo non trascurabile, in quanto si tratta di processi che hanno bisogno per avvicinarsi di salvare i registri e di vedere se i registri ripristinati sono adeguati all'esecuzione. Per far ciò bisogna aver salvato tutto il contesto di un processo e bisogna ripristinare i registri più altri valori per configurare la CPU per l'esecuzione del nuovo processo. A questo vanno aggiunti i tempi necessari per portare i frame da disco a RAM e viceversa, se contiamo un processo di 100 MB che devono essere trasferiti su disco con una velocità di 50 MB/s, vuol dire che lo swap out time è di 2 secondi, più i 2 secondi di swap in per un totale di 4 secondi che non sono pochi.

Problema dello swapping molto simile al problema che avevamo visto nel considerare lo spostamento di un processo da una zona a un'altra della memoria quando si voleva fare compattazione di zone contigue.

Quando si fa swap out di un processo bisogna fare attenzione che se questo processo è in stato di WAIT I/O non si può buttarlo fuori perché eventualmente ci sono pagine che contengono dati che stanno transitando verso l'output o contengono dati che sono in fase di riempimento da dati che provengono in input.

Ci sono due soluzioni:

- Non fare swap out di quel processo
- Fare in modo che un processo in WAIT FOT I/O non abbia mai dati di memoria user, dati della memoria appartenenti al processo coinvolti in I/O. Questo significa che si usa una tecnica detta doppio buffer, cioè se devo fare un output faccio velocemente un trasferimento dei dati da area utente a un buffer kernel e durante questo non posso essere swap out, dopo posso farlo.

Nei sistemi operativi moderni si cercano varianti di swapping.

Nel sistema mobile lo swapping mobile non si fa, i meccanismi adottati sono del tipo, piuttosto di salvare un processo per ripristinarlo in modo semi automatico, si chiude l'applicazione salvandone dati intermedi in modo che si possa successivamente far ripartire l'app e continuare dal punto in cui si era lasciato.

Swapping con paginazione -> un processo potrebbe salvare su backing store solo alcune pagine, tendenzialmente quelle che non servono per fare in modo che il processo vada avanti.

Intel 32 e 64 bit Architectures -> in questi sistemi la gestione delle page table coinvolge un altro tipo di entità, chiamato di segmento. Il segmento è come se fosse una pagina di dimensione variabile. Un segmento può arrivare fino a 4 GB quindi ci possono essere fino a 16K segmenti per processo. Quindi un processo avrà una tabella dei segmenti e avrà un numero di 16 bit divisi in 13 1 2 che servono a indicare il selettore di segmento. La segmentation unit è il primo stadio di traduzione logico fisica a cui segue la paging

unit il secondo stadio di traduzione logico fisica. Alla paging unit arriva un indirizzo su 32 bit diviso in page number (a sua volta diviso in due) e offset.

L'indirizzo logico è quindi fatto da un selettore e da un offset, dove il selettore seleziona un segmento e l'offset è l'offset all'interno del segmento (la grandezza dipende dalla dimensione del segmento, abbiamo visto che il selettore è 16 bit). Il selettore viene usato per individuare in una descriptor table un segment descriptor che viene sommato all'offset per generare l'indirizzo lineare. Questo viene inviato allo stadio di paginazione che può funzionare come tabella delle pagine a 2 livelli (tabelle piccole da 4 KB) o 1 (tabelle grandi da 4MB).

Segmentazione e paginazione messe in cascata può essere uno schema a 3 livelli di traduzione.

L'architettura ARM è quella di un microcontrollore (microcontrollore vs microprocessore-> il primo è un microprocessore dedicato all'I/O, deve avere caratteristiche di basso consumo), a 32 bit. Può avere due soluzioni in termini di paginazione: la prima pagine da 4 KB a 16 KB, l'altra basata sulle section (pagine da 1MB a 16MB). Due livelli di TLB, importante per garantire di fare pochi accessi alle tabelle delle pagine ma che la traduzione sia fatta all'interno della CPU.

2. Memoria Virtuale

La paginazione a richiesta ha come obiettivo quello di definire uno spazio di memoria di indirizzamento virtuale che sia realizzato solo parzialmente in memoria in un certo tempo in modo dinamico, così detto a richiesta.

Un programma per essere eseguito non ha bisogno, di solito, tutte le sue parti e questo può essere una motivazione per il fatto che non tutto un programma è necessario in un determinato tempo. Questo può essere declinato in due modi:

- Una parte del programma non è mai usata in una data esecuzione
- Una parte del programma potrebbe sostituire un'altra che era usata in precedenza ma non più ora.

Per far ciò deve esistere un supporto per eseguire un programma solo parzialmente in memoria.

Il risultato che si può raggiungere è quello di fare in modo che un programma richieda meno memoria e quindi ci siano più processi possibili in contemporanea esecuzione.

Memoria virtuale -> la netta separazione tra lo spazio di memoria logica e quella fisica che ha nascosto l'aspetto che una parte degli indirizzi logici sono mappati a indirizzi fisici e una parte no. Virtualmente c'è tutto lo spazio logico, realmente no.

Questo ha come conseguenza il fatto che se la RAM avesse una certa dimensione, lo spazio di indirizzamento logico potrebbe essere molto più grande della RAM stessa. Questo permette di fare condivisione, di creazione più efficiente di processi, di aumentare il numero di programmi eseguiti contemporaneamente, di fare più efficientemente operazioni di swap e caricamento dei processi stessi.

Lo spazio di indirizzamento virtuale, l'insieme degli indirizzi che un programma in esecuzione vede, di solito parte dall'indirizzo 0 ed è uno spazio di indirizzamento contiguo, mentre gli indirizzi fisici sono realizzati mediante frame e non tutte le pagine devono stare in un frame. La traduzione logico fisica è fatta dalla MMU.

La memory map è l'insieme delle pagine che serve per tradurre da logico fisico. Alcune delle pagine possono essere localizzate direttamente in memoria fisica, altre non stanno in memoria fisica ma stanno in backing store.

Siccome non tutte le pagine devono stare in un frame è possibile vedere uno spazio di indirizzamento virtuale, anche grande, con un buco in mezzo. Questo è molto comodo perché l'heap (memoria in cui si fanno allocazioni dinamiche) e lo stack possono avere crescita/descrescita dinamica (codice e dati hanno dimensione fissa). Tra heap e stack si possono inserire le librerie shared. Lo spazio di indirizzamento logico è fatto a pezzettini con aree inutilizzate.

Paginazione a richiesta -> una pagina entra in un frame quando ce n'è un'effettiva esigenza.

Questo sistema è molto simile al paging con lo swapping, però ha un meccanismo leggermente diverso, qui è centrale la pagina e non l'intero processo.

Se una pagina è necessaria la riferiamo, se questo riferimento non è valido questo significa che non è in un frame, questo in linea di massima dovrebbe significare che sta sul disco. A questo punto si può fare il Lazy Swapper, che non porta in memoria nessuna pagina a meno che questa non venga richiesta.

In definitiva, i processi di caricamento parziale di programmi in memoria si basano sulla capacità di indovinare in futuro le pagine di cui ci sarà bisogno. Per fare operazioni di swapping e paging serve il supporto della MMU.

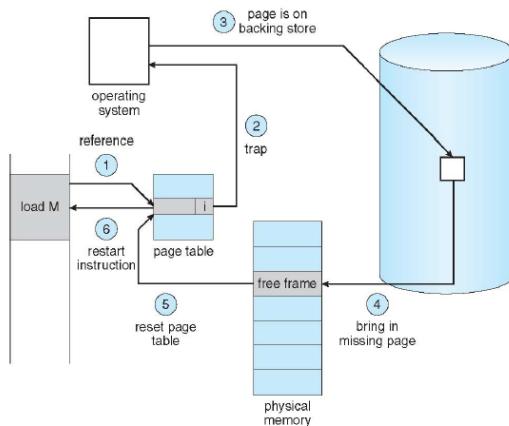
Se la pagina è già in memoria non serve il demand paging, altrimenti bisogna caricarla.

È possibile che una pagina a cui un processo vuol fare accesso non sia ancora mappata su un suo frame ma sia già in memoria, per gestire ciò di fianco al numero di frame c'è un bit di validità o non validità (valido se è su un frame).

Su backing store potrebbero stare sia pagine che non stanno in frame ma anche una copia da aggiornare che stanno in frame.

Un Page Fault costa, bisogna fare delle operazioni per gestirlo: si parte con un riferimento a pagina, il primo riferimento a una pagina che non è in un frame (quindi ha il bit di validità a 0) genera un page fault, fatto via hardware, ovvero la MMU dev'essere in grado di gestire questo bit di validità e scatenare una trap. Una volta attivata questa trap, la funzione del sistema operativo del kernel che gestisce questa trap deve andare a guardare un'altra tabella per capire se questo bit non valido significa:

- Questo riferimento è un errore -> abortisce programma
- Pagina che potrebbe essere valida ma non è in RAM:
 - Si ricerca un frame libero
 - Si porta la pagina da disco a RAM (operazione su disco)
 - Si sistema la tabella delle pagine per indicare che ora la pagina è in memoria settando il bit di validità a 1
 - Si fa ripartire l'istruzione che ha causato page fault



Come caso estremo di paginazione richiesta, si potrebbe partire con un processo che non ha nessuna pagina in memoria, quindi la prima istruzione genererà un page fault e tutto potrà partire con una tecnica di pura paginazione richiesta.

In questo contesto vanno tenute in conto alcune complicazioni.

Per esempio, un'istruzione potrebbe richiedere fetch e decodifica di istruzioni che sommano due numeri in memoria e portano il risultato ancora in memoria (4 accessi in memoria). Per far ciò bisogna verificare il fatto che un'operazione di page fault che porti in memoria una sola pagina potrebbe essere poco.

La gestione della page fault dovrebbe potenzialmente portare più pagine in memoria-> più complicato

La ricerca di un frame libero può essere fatta rapidamente avendo a disposizione una lista dei frame liberi (free frame list) -> ogni frame libero contiene un puntatore al successivo frame libero e quando è necessario un frame è sufficiente che il gestore del page fault (kernel) abbia un puntatore al primo frame libero per ottenere direttamente un frame disponibile.

Ci sono delle complicazioni: in alcuni casi il kernel può mettere a disposizione dei frame liberi e preazzerati. La free frame list all'inizio contiene tutta la memoria disponibile.

Gestione Page Fault:

- Trap al sistema operativo: il processo utente in esecuzione viene fermato, va in stato di READY, il gestore della trap deve entrare in gioco
- Vengono salvati i registri del processo
- Bisogna determinare che l'interrupt era dovuta a una page fault e bisogna chiamare il gestore della page fault
- Il gestore deve verificare che il riferimento era legale e determinare dove si trova la pagina sul disco (altrimenti abort)
- Deve far partire una lettura su disco in un free frame
- Mentre questa operazione è in corso, anche il kernel si mette in attesa e da la CPU a un altro processo nella coda READY
- Ci sarà un interrupt che dice che I/O è stato completato
- Il processo in esecuzione viene interrotto, si salvano i registri e si fa partire la parte di kernel che deve gestire interruzione
- Si determina che l'interruzione è dovuta al termine dell'I/O
- Si corregge la tabella delle pagine e si cerca di far partire il processo che aveva scatenato page fault
- Non parte subito ma viene inserito nella coda READY e prima o poi verrà attivato

Ci sono 3 principali attività per gestire un page fault:

- Servizio di interrupt
- Lettura della pagina da disco a memoria (più costoso)
- Rifar partire il processo (meno costoso)

La probabilità di page fault p compresa tra 0 e 1 (0 no page fault, 1 sempre page fault)

Chiamiamo il tempo effettivo di accesso

EAT=(1-p) x memory access time + p (page fault overhead + swap page out + swap page in)

Per esempio:

Memory access time = 200 nanosecondi

Average page fault service time = 8 millisecondi

$$EAT = (1-p) \times 200 + p \times (8 \text{ millisecondi}) =$$

$$= (1-p) \times 200 + p \times 8000000 =$$

$$= 200 + p \times 7999800$$

Se p tende a 0 abbiamo 200 ns se p tende a 1 abbiamo 8 millisecondi.

Se p è 1/1000 (1 page fault ogni 1000 accessi) -> EAT = 8.2 microsecondi (peggiamento del 40%)

Per avere un peggioramento inferiore al 10% dobbiamo avere $p < 2,5 \times 10^{-6}$ -> un page fault ogni 400000 accessi.

Trattiamo della valutazione del costo di un accesso in memoria, tenendo conto della possibilità di avere un page fault: per certi versi, è un ragionamento simile a quello fatto per l'uso della TLB, ma là si trattava di risparmiare/accorciare il tempo di due accessi ad un accesso. Qui la differenza tra i due tempi no-page fault e page fault è decisamente più alta.

Ottimizzazioni della paginazione a richiesta

Come si può cercare di ridurre il costo complessivo dei page fault, scatto che va pagato per poter effettuare paginazione a richiesta, cioè senza tutte le pagine in memoria? Cioè, vogliamo risparmiare memoria, non portando tutte le pagine in frame? Allora dovremo spendere del tempo per gestire i page fault.

Ogni volta che c'è un'azione, una parte di un algoritmo costosa, ci possono essere due strade per migliorare le prestazioni:

- rendere quella cosa meno costosa
- diminuirne il numero di chiamate, di attivazioni

Dal punto di vista dei page fault, i tentativi che si possono effettuare sono:

- cercare di fare in modo che costi meno **tempo** il trasferimento dei dati tra disco e memoria: l'I/O legato allo spazio di swap (alle pagine che devono andare su disco, o viceversa) può essere migliorato **rendendo più efficiente l'accesso a disco** → si può allocare in modo migliore la parte del disco dedicata a swap per blocchi più grandi, e avere una gestione del disco più efficiente di un file system standard
- una volta deciso che la partizione di swap permette accessi più efficienti che non al file system, allora teniamo conto che, almeno all'inizio, le pagine sono reperite nel file system dal file eseguibile, che se viene caricato in modo incrementale sarà anch'esso soggetto a swap in. Tutto sommato, se la parte di disco dedicata a swap è più efficiente, tanto vale copiare tutto il file all'inizio, nella partizione di swap, e poi paginare di lì: uno scatto pagato all'inizio, comunque, ma più uniforme/regolare, SE la partizione di swap è più efficace. In Unix PSD, si usava questa strategia.
- In Solaris e BSD attuale si usa una strategia quasi opposta: si fa comunque paginazione dal file eseguibile su disco, (quindi non si trasferisce nella partizione di swap), ma si butta via la pagina senza restituirla alla partizione di swap quando si decide che non ce n'è bisogno. Siccome il file su disco c'è ancora, e le pagine di codice non vengono cambiate, allora si fa solo swap in, e non swap out delle pagine su disco.

Ci sono altre piccole ottimizzazioni: ae, la **anonymous memory**, dei frame allocati per contenere delle pagine, non per contenere pezzi dell'eseguibile, cioè codice o dati globali, ma ad esempio per stack e heap. Stack e heap partono senza dati, e vengono riempiti progressivamente; ae, supponiamo che per l'heap si faccia una malloc sufficientemente grande, e come conseguenza una malloc di un vettore molto grande che in realtà non si usa tutto, ma solo una parte: di fatto, una parte di queste pagine non sarà utilizzata, non ci si scrive mai. Allora, questa anonymous memory può essere vista come dei frame che vengono semplicemente presi, associati a pagine in cui all'inizio non c'è niente di significativo e quindi non vengono caricate da disco: no swap in. Di una parte di questa, la parte non utilizzata, non ci sarà neppure swap out se dovessero poi anche uscire dai frame. Questa strategia dell'anonymous memory si basa sull'idea "quando si può, evitiamo di copiare sia da disco a memoria, che da memoria a disco".

Un'altra ottimizzazione è che ci sono pagine che sono state modificate in memoria, ma non ancora scritte su disco: queste vanno salvate; le pagine non modificate in memoria non vanno riscritte su disco.

In sostanza: si può cercare di essere più efficienti nei trasferimenti relativi a disco, o di risparmiare le copie avanti-indietro che non servono, o perché non ce n'è bisogno, o perché su disco c'è già una copia, inutile sovrascriverla.

Nei sistemi mobile, non si usano queste complicazioni perché costano troppo

Copy-on-Write

Un altro tipo di problematica di ottimizzazione legata alla paginazione a richiesta è la Copy-on-Write: è una strategia legata al caso di processi che condividono pagine. È frequente, ae, nel caso di un processo che faccia una fork di un altro processo: nel momento in cui, in un sistema tipo Unix, si genera un child di un processo parent, il parent e il child, che fanno riferimento allo stesso eseguibile, subito dopo la fork, sono

identici.

Si immaginino i processi 1 e 2 con pagine logiche distinte, tuttavia mappate sugli stessi frame fisici (subito dopo una fork).

Dopo un po', supponiamo che p1 modifichi la pagina C, che NON è condivisa: queste sono pagine shared nel senso che entrambi i processi vedono la stessa informazione. Nel momento in cui una di queste pagine (ae, C) diverge, chi modifica la pagina sarà costretto (in automatico dal SO) a farne una copia.

Copy-on-Write significa dunque: "se scrivi in quella pagina, prima ti fai una tua copia privata".

Quando servono, in generale, pagine libere o frame liberi per ottenere spazio in memoria, vengono anche gestiti dei pool, degli insiemi di pagine già azzerate (non sono free list). La semplice free list (= quando serve un frame libero, peschiamo dalla free list), può essere ulteriormente complicata da un gestore di frame liberi che permette di ottenere pagine "pulite".

Cosa succede quando non ci sono free frames?

Prima si era detto: carichiamo un po' alla volta, non tutte le pagine all'inizio e se ad un certo punto, perché c'è poco spazio, perché il processo vuole troppa memoria, ecc, ci accorgiamo che c'è una pagina che non sta in un frame e va caricata, ma non ci sono più frame liberi?

Se non ci sono frame liberi, occorre adottare una politica di replacement, o rimpiazzamento, o sostituzione: cioè, la conseguenza diretta di paginazione a richiesta (= una pagina va in un frame solo quando c'è una richiesta) è quella che non basta la richiesta, ma questa richiesta potenzialmente deve scatenare una restituzione di un'altra pagina su cui ci sarà un rimpiazzo.

Una nuova pagina deve entrare: ne facciamo uscire una per far posto. Il problema è: quale pagina? Serviranno degli algoritmi sufficientemente furbi per far sì che la pagina che deve uscire per essere rimpiazzata in un dato frame, non sia una pagina a cui si fa accesso subito dopo: se questo succede, si palleggia una pagina dentro e fuori dalla memoria più volte, ed è la cosa peggiore che si possa fare. L'impatto dello sbagliare la pagina da far uscire è di alzare il numero di page faults, con un costo in tempo non indifferente.

Page replacement

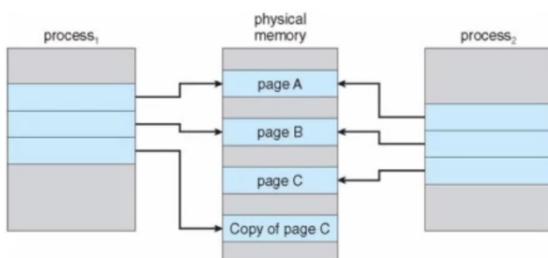
Serve a prevenire la sovra-allocazione di memoria, cioè doverne allocare troppa, oppure non poterne allocare abbastanza. Quando si fa servizio di un page fault, dentro la page fault service routine (la funzione che gestisce un pf), e questa funzione ha deciso che occorre un frame libero per una pagina che deve entrare, occorre trovarne una da far uscire, per fare page replacement.

Per fare page replacement, oltre ad altre cose che vedremo, è opportuno usare il **modify (o dirty) bit**, (presente anche in TLB e page table oltre a un bit di validità). Il modify bit indica che la pagina è stata modificata (= scritta) rispetto all'ultima volta che è stata portata da disco in memoria. Il vantaggio del dirty bit è che se una pagina ha il bit non settato (= 1), vuol dire che ce n'è già una copia su disco, inutile copiarla indietro.

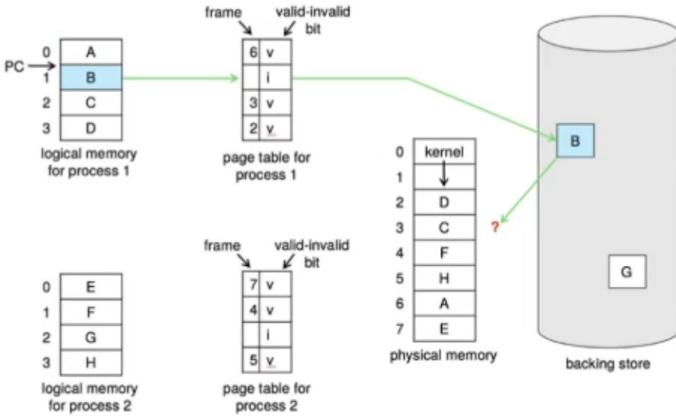
Con le politiche di sostituzione di pagina, completiamo la trattazione sulla gestione della memoria virtuale: se la paginazione a richiesta era la novità per gestire in modo più dinamico l'insieme di pagine di un processo, non può essere effettuata correttamente senza parlare della sostituzione di pagine.

Es. di sostituzione pagina:

After Process 1 Modifies Page C



Need For Page Replacement



Ci sono due processi, 1 e 2. Entrambi hanno una tabella delle pagine, di cui 3 mappate su frame e una no (la 1 nel p1, e la 2 nel p2). In memoria c'è il kernel che occupa i primi due frame, e poi ci sono 6 frame occupati da 3 pagine per processo: la RAM è piena. Ad un certo punto, serve la pagina 1 del processo 1: dal backing store, si cerca posto per B ma non c'è, bisogna trovare qualcuno che faccia posto, la cosiddetta vittima (pagina che deve uscire) del processo 1 o globalmente.

Basic Page Replacement

Politica di sostituzione pagine base: il problema si attiva nel momento in cui nella gestione di un page fault.

1. si è trovata la pagina su disco
2. si deve trovare un frame libero in cui copiarla:
 - se c'è nella free list, usalo
 - se no, attiva una politica di sostituzione pagina che si concretizza con l'individuare il frame "vittima" = frame che diventerà libero per la pagina che deve entrare, mandando la pagina che contiene sul backing store. Sarebbe opportuno prenderlo con dirty bit a 0, perché più conveniente: la pagina non è stata modificata, quindi non deve essere copiata su disco. Se il dirty bit è a 1, bisogna fare due trasferimenti: la vittima torna su disco, la nuova pagina entra nel frame.
3. Il resto è quello che si è già visto parlando di page fault.

Una volta individuata la vittima, la prima operazione sta nel salvare eventualmente la vittima. Se si gestisce il dirty bit, e questo è uguale a 0, questa operazione non è necessaria. Poi si mette la nuova pagina nel frame e si aggiorna la tabella delle pagine. Nella page table si modifica sia il bit della pagina che esce, sia di quella che entra: un valid diventerà invalid, e un invalid, valid.

Frame and Page Replacement Algorithms

=Algoritmi di sostituzione pagine che vengono attivati dalla gestione di un page fault quando si tratta di individuare la vittima.

La prima osservazione è questa: quanti frame assegniamo ad un processo? Il numero di frame assegnati ha un impatto non indifferente sul problema di individuare le vittime, perché è abbastanza ovvio che se ad un processo assegniamo molti frame, prima di riempirli tutti passerà del tempo, e quindi sarà meno critico il processo di sostituzione pagine; se diamo pochi frame, abbastanza presto saranno tutti pieni e bisognerà attivare la sostituzione delle pagine. Un secondo aspetto è: quale frame selezionare come vittima quando

sono tutti pieni? L'obiettivo per determinare l'ottimalità dell'algoritmo è ottenere la frequenza dei page fault più basso possibile: vogliamo **ridurre il numero di page fault**. Per far ciò, usiamo una strategia che viene spesso usata per fare benchmarking di algoritmi: usiamo una stringa di riferimenti (=una sequenza di riferimenti in memoria della quale scriviamo in successione gli indirizzi o i numeri di pagina). Ogni numero della stringa dei riferimenti è un riferimento a una pagina (una R/W, o Load/Store (Assembler), o una Fetch di un'istruzione). Non mettiamo indirizzi 'veri', ma solo il pezzo di indirizzo che rappresenta la pagina. Poi proviamo questi accessi con varie tipologie di algoritmi che possono determinare qual è la vittima dato il numero di frame a disposizione.

Se noi rappresentassimo in un grafico cartesiano sulle ascisse il numero di frame disponibile, e sulle ordinate il numero di page fault che vengono generati, è plausibile che l'andamento sia di questo tipo: più il numero di frame è elevato, meno ci saranno page fault. Un frame è il minimo: è ovvio che con un frame avremo un certo numero di page fault. Supponiamo che la stringa di riferimenti faccia 22 accessi circa, e con un solo frame ci siano 13 page fault: secondo il grafico, se avessimo 2 frame, ne avremmo solo 7, se avessimo 3 frame, 5, 4 frame, circa 4, e via via diminuiranno i page fault. Più frame mettiamo, meno page fault avremo, perché sarà più facile trovare la pagina già in memoria.

Page Replacement Strategy

È data la formula della frequenza di page fault (frequenza e probabilità sono quasi la stessa cosa: la frequenza è il numero di casi rappresentativi sul totale, la probabilità è l'attesa per il futuro), la frequenza di page fault di un dato algoritmo A, con m numero di frame. Dato un algoritmo di sostituzione pagine, e dato il numero di frame disponibili fisso, non variabile nel tempo e dedicato a un certo processo, la frequenza di page fault sarà il numero di page fault sul numero totale di accessi. Data una stringa di riferimento, la frequenza di page fault relativa a quella stringa sarà al denominatore, quanti sono i riferimenti, al numeratore, il numero di page fault.

$F(A, m, w)$ è il numero di page fault generati dalla stringa di riferimenti w , usando l'algoritmo A in un sistema con m frame. Se ad esempio, su 22 accessi (w), abbiamo 15 page fault, la frequenza di page fault sarà $15/22$. Date varie stringhe di riferimenti, in realtà la frequenza complessiva sarà una media pesata delle singole frequenze relative.

Se noi supponessimo di fare un plot su un grafico in cui, sulle ascisse mettiamo il numero di frame, e sulle ordinate mettiamo la frequenza di page fault, allora noi ci aspetteremo che se il numero di frame disponibili è basso (ad esempio 1) avremo il massimo numero di page fault, quindi una frequenza di page fault che tende a 1; se invece arrivassimo ad avere tanti frame quante le pagine del processo, tenderemmo ad avere page fault nulli o quasi (bisognerà decidere se per ogni pagina, conta come page fault la prima copia in memoria). Questo grafico ha due casi estremi: ogni frame è in grado di ospitare una pagina diversa dalle altre, e questo vuol dire che non ci saranno mai page fault, eccetto il primo page fault per portare la pagina nel frame corrispondente; l'altro caso estremo è di avere sempre page fault perché c'è sostanzialmente un solo frame disponibile dove devono entrare tutte le pagine, dunque è realistico avere la frequenza di page fault 1 o quasi 1.

Se si sceglie un algoritmo casuale che quando deve determinare una vittima, la prende a caso, il grafico di f è una retta. Più l'algoritmo è furbo, più tende ad andare verso l'origine con una curva schiacciata e "migliore"; la curva predittiva, quella in cui si conosce il futuro (algoritmo ottimale: prendo la vittima che per più tempo non userò nel futuro), tende a minimizzare il numero di page fault. Per tutti questi algoritmi valutiamo i page fault in funzione dei frame assegnati; se aumentiamo il numero di frame, a parità di algoritmo, diminuirà la frequenza di page fault.

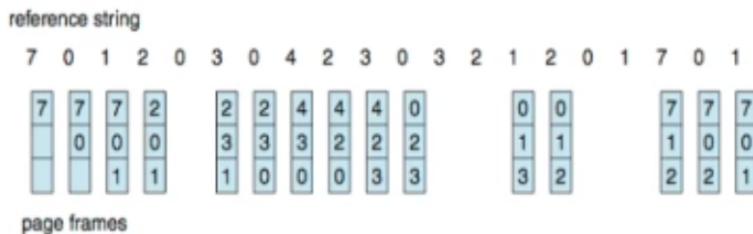
Nella selezione delle vittime, sarebbe opportuno tenere conto dei bit di riferimento e modifica, nel senso che tendenzialmente se noi abbiamo assegnato a una pagina nella page table o TLB un dirty/modified bit, che dice se la pagina è stata modificata o no, ci può aiutare per determinare la vittima; così come se usiamo il reference bit, =1 nel momento in cui si accede alla pagina.

First-In-First-Out Algorithm

Vediamo il primo algoritmo di sostituzione pagina, per certi versi il più facile: il FIFO. È un algoritmo nel quale si fa la classica coda dove chi arriva prima, passa prima; in questo caso, la pagina che è entrata per prima in un frame, sarà anche la prima ad uscire. Non è la politica migliore, ma è la più semplice.

Supponiamo di avere una certa stringa di riferimento e 3 frame: in questo caso, A=FIFO, m=3.

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



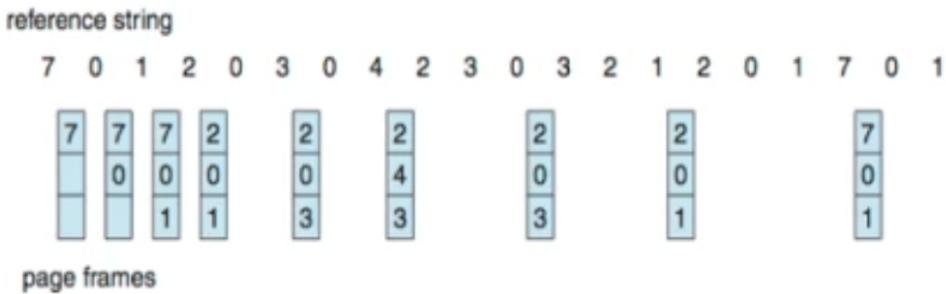
15 page faults

I vari fotogrammi rappresentano i 3 frame dopo ogni accesso. C'erano 3 frame vuoti (non rappresentati). Dopo l'accesso alla pagina 7, questa viene portata nel primo frame, e gli altri 2 sono liberi; nella coda FIFO, in testa c'è 7. Si fa accesso a 0, che entra nel secondo frame disponibile, e ce n'è un terzo libero. Nella coda, in testa c'è 7, poi 0. Arriva la 1, che si mette nel 3o frame disponibile; la coda è 7 0 1. Ora, pensando alla coda FIFO come un buffer circolare con head e tail, nella prima configurazione, l'head è 7 e il tail è la casella vuota dopo il 7; nella seconda, l'head è 7 e il tail è la casella dopo 0; quando c'è, 7 0 1, caso di FIFO pieno, head e tail = 7: ogni volta che ci sarà richiesta di una nuova pagina (ae, 2) bisogna individuare una vittima, e la vittima è **head**. La vittima esce, viene copiata su disco, e viene rimpiazzata (qui, da 2). La testa del FIFO passa allo 0. Successivamente c'è un accesso a 0 di nuovo: è già in un frame => non c'è page fault (fotogramma vuoto). Per ora abbiamo fatto 4 page fault (i primi accessi a 7, 0, e 1). Il sesto accesso a pagina 3 genera page fault; la testa del FIFO è 0, dunque esce 0, entra 3, la head passa a 1. Nuovo accesso allo 0, ahimè, appena buttato fuori, fa uscire l'1 e entrare lo 0 (testa su 2); il prossimo accesso il 4, esce il 2... finendo la stringa, ho 15 page fault.

La politica FIFO è poco predittiva, e si basa sull'idea che se una pagina è in memoria da più tempo, sarà la vittima, indipendentemente dal fatto che io voglia o no farne accesso a breve. Un caso particolare dell'algoritmo FIFO è l'anomalia di Belady: con una certa stringa di riferimenti, con 3 frame disponibili il numero di page fault sarebbe 9, e con 4 (più frame), il numero di page fault aumenta. L'anomalia è legata al fatto che questo grafico a 'scalino' non è conforme ad una curva che scende, ma è una curva che talvolta pur aumentando m potrebbe salire.

Optimal Algorithm

Ora vediamo un esempio di algoritmo diametralmente opposto, cioè che conosce già il futuro. In un caso pratico, non è possibile conoscere il futuro, ma se noi simuliamo una stringa di riferimento già rilevata, e facciamo sì che l'algoritmo che vede già la stringa, veda anche i riferimenti futuri, possiamo immaginarlo. Definiamo l'algoritmo ottimo: la vittima è la pagina che per più tempo nel futuro NON mi servirà, cioè la pagina che o non sarà più usata, o la prossima volta che sarà usata è più avanti nel tempo rispetto alle altre. Si può dimostrare che questo algoritmo è ottimo, ovvero genera il minor numero di page fault.



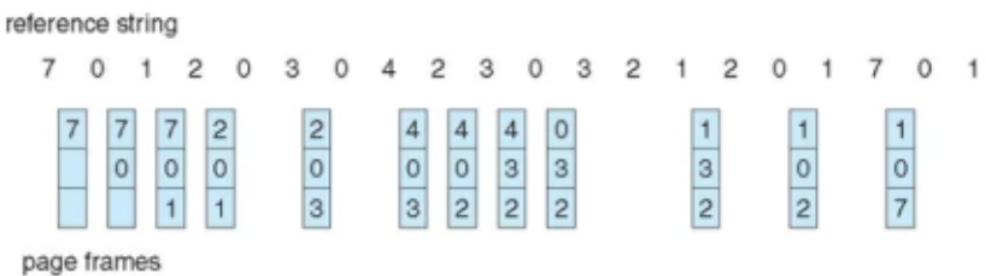
Riprendendo la stringa e i frame di esempio del FIFO, abbiamo un primo page fault inevitabile con il 7, supponendo un algoritmo di pure paging (tutti i frame liberi inizialmente), quindi 7, 0, 1 sono page fault che riempiono i 3 frame. Arriva 2, che genera page fault; nel FIFO si buttava il 7, il più vecchio. Qui si guarda la stringa e si controlla tra 7, 0 e 1 qual è quello che verrà usato più nel futuro (lo 0 viene usato quasi subito, l'1 poco dopo, il 7 quasi alla fine): si butta fuori il 7. Poi arriva lo 0, no page fault; poi il 3: si sceglie una vittima (nel FIFO, si buttava lo 0, che poi veniva usato subito dopo) tra 2, 0 e 1. Quello che compare più tardi nella stringa è l'1, quindi il 3 sostituisce l'1... e così via. Alla fine, ho 9 page fault (anziché 15), e con questa stringa NON si può fare meglio.

Questo algoritmo non è realizzabile perché in un sistema in esecuzione si può solo conoscere il passato, non le pagine future: di un programma non si sa quali saranno le pagine usate in futuro, se non per programmi estremamente semplici.

Least Recently Used Algorithm

Un modo per cercare di avvicinarsi all'algoritmo predittivo, guardando al passato è il LRU. Supponiamo che la storia passata si replichi nel futuro: è probabile che la pagina a cui farò accesso più lontano nel futuro sarà, tra i frame, quella a cui ho fatto accesso più lontano nel passato. Si assume una certa ripetitività negli accessi.

La vittima sarà la pagina usata MENO di recente, quella non usata da più tempo. Si usa la storia passata, per fare una previsione. Per far ciò, bisogna conoscere di ogni pagina l'ultimo istante in cui è stata usata, e quando bisogna scegliere la vittima, confrontare tra le pagine quella con il tempo più lungo.



Con questo algoritmo, si arriva a 12 page fault. Nel dettaglio: i primi tre accessi sono identici; il terzo è uguale al FIFO, perché quando ad una pagina ho fatto solo un accesso, FIFO = LRU. Dunque il 2 prende il posto del 7. 0 non genera page fault. Arrivo al 3 e il 3 sfrutta il fatto che allo 0 ho fatto un nuovo accesso più di recente: quindi non è più il più "vecchio", invece la pagina a cui non ho fatto accesso da più tempo tra le 3 è l'1. Arrivo al 4, che rimpiazza il 2, essendo quello con l'accesso più lontano, e così via.

Time	1	2	3	4	5	6	7	8	9	10	11	12
String	4	3	2	1	4	3	5	4	3	2	1	5
Fault	*	*	*	*	*	*	*		*	*	*	
Frame 0	4	4	4	1	1	1	5	5	5	2	2	2
Frame 1	-	3	3	3	4	4	4	4	4	4	1	1
Frame 2	-	-	2	2	2	3	3	3	3	3	3	5

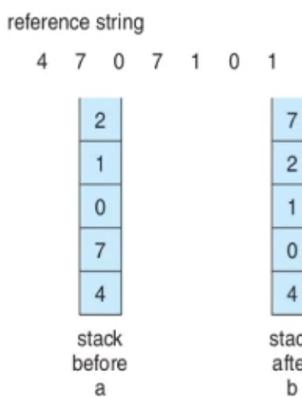
victim

$$f = 10 / 12 = 0.83 \Rightarrow 83\%$$

In un esempio più dettagliato, con i tempi di accesso a partire da 0 o da 1, fino a 12, la stringa 4321... e 3 frame. La pagina 4 va nel frame 0, la pagina 3 nell'1, la 2 nel 2; la pagina 1 genera un page fault e deve scegliere la vittima, la pagina 4, perché si è fatto accesso più lontano nel tempo. La prossima vittima quando arriverà di nuovo il 4 è la 3; quando arriva il 3, prende il posto del 2 e così via. Si è indicata in rosso la vittima che viene scelta in caso di page fault, e con l'* la verifica di un page fault. La frequenza è numero totale di page fault diviso numero di riferimenti = 0,83.

Ora, due considerazioni: LRU è bello, ma poco pratico. Purtroppo l'LRU è un algoritmo che costa eccessivamente: se ogni pagina ha un contatore, una specie di variabile dove si salva il tempo dell'ultimo accesso, ogni volta che accediamo ad una pagina modifichiamo in un vettore di contatori il tempo dell'ultimo accesso. Questo è fattibile, con un costo in memoria, ma quando c'è un page fault bisogna fare una ricerca del minimo, la pagina con il tempo di accesso più piccolo tra tutti. Senza ottimizzazioni particolari questo ha un costo lineare nel numero di frame, mentre la politica FIFO ha costo unitario (si ricorda solo la head del FIFO). Il problema è dunque la ricerca, che rende la counter implementation poco pratica. Si può fare un'implementazione a stack fatta con lista doppio linkata, un po' più efficiente a tempo costante, anche se comunque costosa. Le politiche LRU e OPT non hanno l'anomalia di Belady, quindi aumentare il numero di frame è sempre vincente.

Use of a Stack to Record Most Recent Page Reference



Ad esempio, prima dell'istante a, prima di fare riferimento a 7, le pagine ordinate in base agli ultimi accessi era 21074. La pagina più recente è 2, la più vecchia è 4 (la vittima nel caso di page fault). Quando accedo a 7, non genera page fault ma diventa la più recente, quindi deve essere spostata nella lista, in testa allo stack. Questo stack visualizzato come vettore, non può essere realizzato con vettore perché richiede uno slittamento lineare dei dati: quindi è una lista doppio linkata, perché questo tipo di lista, se c'è un puntatore al 7, permette in tempo costante, pur mettendo a posto 6 puntatori, di spostare il 7 in testa.

LRU Approximation Algorithm

L'algoritmo LRU fatto così è ancora costoso, quindi si tendono a fare LRU approssimati, i quali si accontentano di non scegliere la vittima come la pagina usata meno di recente, ma una pagina usata abbastanza poco recentemente; si determina una vittima sufficientemente indietro nel tempo, anche se non quella più vecchia in assoluto. Per far ciò serve un supporto hardware, il reference bit, nella page table

e nella TLB: si fa in modo che ci siano un po' di pagine con reference bit a 1 (=si è fatto accesso di recente) e un po' di pagine con reference bit a 0 (=non si è fatto accesso di recente). Se siamo in grado periodicamente di azzerare tutti i reference bit dei frame e delle pagine in frame, e di far sì che ogni volta che si accede ad una pagina, il reference bit = 1 automaticamente, allora esiste un modo per determinare se una pagina è abbastanza recente, o no, a seconda del reference bit =1 o 0.

Un algoritmo che sfrutta questo reference bit e approssima LRU, essendo ragionevolmente efficiente, è il **Second Chance**: è un algoritmo di base FIFO (quindi costo $O(1)$), ma un po' più costoso. La potenziale vittima viene guardata; se è con ref bit=1 (accesso recente), viene salvata, il suo bit va a 0 (=al prossimo giro gli tocca, a meno che ci sia un nuovo accesso che rimette il ref bit a 1), e il puntatore alla testa del FIFO passa alla prossima pagina, con lo stesso ragionamento.

Supponiamo di avere le pagine che vengono guardate in un buffer circolare del FIFO, e la vittima abbia reference bit = 1; viene salvata, non è la vittima, il suo reference bit posto =0, e si guarda la pagina successiva. Anche questa ha reference bit = 1, viene messo a 0 e si guarda la successiva. Questa ha reference bit = 0, è la vittima.

Enhanced Second Chance Algorithm

Di questo algoritmo Second Chance si possono fare delle varianti, dove si tiene conto non solo del reference bit ma anche del modify bit. Quando si cerca la prossima vittima, cerco prima una pagina non recente (reference bit = 0) e che abbia anche un modify bit = 0, così non si deve ricopiare su disco. Se non trovo questa pagina, vado a una con reference bit = 0 e modify bit = 1, perché comunque la posso ritenere vecchia, ma va salvata. Se non trovo nemmeno questa, trovo una pagina con reference bit = 1, ma con modify bit a 0. L'ultima scelta è una pagina con entrambi i bit a 1.

Questi algoritmi approssimano la strategia LRU (= la previsione sul futuro la si fa guardando la pagina con l'ultimo accesso più lontano nel passato); ci sono algoritmi alternativi che invece di guardare l'ultimo accesso nel passato, guardano la frequenza di accessi in un certo intervallo, invece di guardare un istante particolare nel futuro: una pagina mi serve se nel tempo di riferimento l'ho usata tante volte, non mi serve se l'ho usata poco. Da questo ragionamento posso creare vari algoritmi:

- scelgo la pagina usata meno frequentemente (LFU): se è stata usata poche volte nel passato, mi servirà lontano nel futuro
- scelgo la pagina usata più frequentemente nel passato (MFU): se è già stata molto usata, nel futuro lo sarà poco

Page-buffering algorithms

Ora esaminiamo alcune altre problematiche legate all'efficienza della paginazione a richiesta, non tanto nell'indovinare la vittima migliore, ma nell'esplorare tentativi di migliorare l'efficienza globale della paginazione. Sono i cosiddetti algoritmi di page buffering, che cercano di gestire un pool di pagine libere, una zona intermedia di pagine vittime non ancora spostate su backing store, e cercano come strategia di ovviare al problema di "sbagliare" la vittima, cioè scegliere come vittima una pagina che serve poco dopo. Teniamo un pool di frame liberi che sarà sempre disponibile: l'idea è di non andare mai nella situazione di dover trovare, durante un page fault, nessun frame libero e dover quindi determinare la vittima: si fanno le cose con tempistiche leggermente diverse.

Ipotizzando che c'è sempre un pool di frame liberi, allora il frame sarà disponibile quando ce n'è bisogno, durante un page fault, e quindi la gestione del page fault non deve anche cercare una vittima; a questo punto si legge la pagina da disco nel free frame e si seleziona comunque una vittima, che però invece di essere trattata come vittima, viene solo selezionata e aggiunta al free frame. In un altro momento opportuno, allora eventualmente i frame vittime determinate in precedenza, aggiunte al pool dei free frame, potranno essere salvate su disco. Quando e come questo salvataggio venga fatto, sono aspetti interni che determinano alla fine le difficoltà di realizzazione dell'algoritmo e quindi l'efficienza globale.

Ae, si può provare a tenere traccia di una lista di pagine modificate, tramite il dirty bit, e cercare poi quando ci sono più pagine nel free frame pool, di usare questi bit di modifica per determinare poi quali pagine prendere. Un'idea è quella, quando una pagina è scelta come vittima e aggiunta al pool dei free frame, di mantenerne intatte le informazioni: non viene ancora buttata su disco, ma viene messa nel pool così com'è. Se a questa pagina faccio nuovamente riferimento, allora la ripristino senza averla dovuta salvare su disco e poi riprenderla. Questo ovvia alla scelta "sbagliata" della vittima. L'efficienza di questi algoritmi si vede sperimentalmente; molto dipende dalla dimensione e gestione del free frame pool.

Applications and Page Replacement

Osserviamo un altro aspetto che può determinare le prestazioni complessive del sistema: se anziché solo il SO, anche l'applicazione avesse parte nel determinare la politica di sostituzione e allocazione di pagine, le cose potrebbero andare meglio. L'applicazione, il programma utente, potrebbe avere più chiaro cosa sta facendo e quindi cosa farà nel futuro, mentre il SO non può che adottare politiche generali, ha meno conoscenza diretta dell'applicazione.

Un esempio sono i database, dove si fanno molte ricerche in memoria, ci potrebbero essere delle strategie dettate dalla conoscenza degli algoritmi di ricerca, per aiutare o sostituirsi al SO per decidere cosa sta in memoria e cosa no. A questo punto, esiste un potenziale conflitto di azioni, di duplicati, ae, nel tenere troppe copie di pagine in buffer qualora sia il SO che le applicazione tentassero di fare lo stesso lavoro: se l'applicazione, in certi sistemi dedicati, si assume la responsabilità di partecipare al lavoro di rendere efficiente l'allocazione di dati in memoria, allora il SO passa la palla e non fa certe azioni, se no si tende a duplicare l'utilizzo di memoria.

Allocation of frames

Un altro importante aspetto nel determinare le prestazioni è quanti frame vengono allocati ad un processo, qual è il minimo. È sufficiente un frame ad un processo? No, ci sono istruzioni assembler che hanno bisogno di più di una pagina in parallelo. Ae un IBM 370, un main frame con linguaggio assembler molto evoluto, aveva un'istruzione SS MOVE, una copia di un dato da una locazione di memoria ad un'altra: nell'ipotesi che l'istruzione SS MOVE, fosse sul bordo finale di una pagina, e iniziasse dalla prossima, cioè che stesse su due pagine, e che sia la sorgente che la destinazione della MOVE fossero a metà, sul bordo, tra due pagine, sia la sorgente che la destinazione occupano due pagine, l'istruzione due pagine => per eseguirla, servivano almeno 6 pagine. Tutto ciò per dire che il minimo non è 1, ma qui è 6.

Fixed Allocation

Quanti frame associamo ad un processo? Ne affidiamo un numero fisso e costante a tutti i processi, o variabile? E se variabile, tra processi, ossia una volta assegnato è costante, o varia anche nel tempo? Un primo schema di allocazione è "a tutti uguali": decidiamo quanti ne vanno al sistema operativo, e i rimanenti sono allocati divisi per ogni processo, pariteticamente. Si sa a priori quanta RAM è assegnata ad un processo, ma questo schema non è molto lungimirante nel valutare se il processo ha proprio bisogno di tutti quei frame. Alcuni processi potrebbero volerne di più, altri di meno.

L'allocazione proporzionale è più equilibrata in questo senso; se conosciamo la dimensione di un processo, del suo spazio di indirizzamento, possiamo attribuire RAM in modo proporzionale alla dimensione del processo. Se chiamiamo $s(i)$ la dimensione del processo $p(i)$ e S la sommatoria di tutti gli $s(i)$, dove gli $s(i)$ sono i processi attivi in memoria nello stesso tempo, se m è il numero totale di frame, ovviamente diamo al processo i -esimo un numero di frame $a(i)$ proporzionale al suo $s(i)$ quindi uguale a $[s(i)/S] \times m$.

Nell'esempio numerico, $m = 64$, prendiamo due processi, $s1 = 10$ pagine, $s2 = 127$ pagine; $S = 137$ pagine => $a1 = 10/137 \times 64 \approx 4$ frame per il processo 1, e $a2 = 127/137 \times 6 \approx 57$ per il processo 2. Si può anche decidere se alla fine si tronca, si arrotonda... l'importante è che alla fine ci stiamo realisticamente.

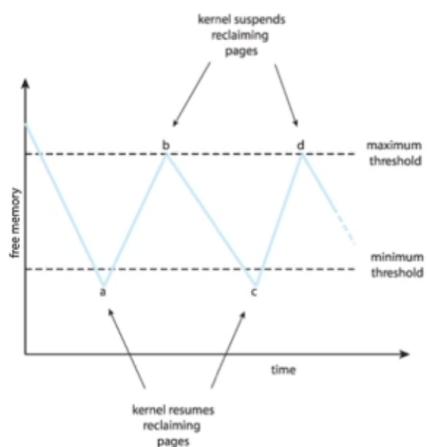
Global vs Local Allocation

Ulteriore problematica: quando cerchiamo la vittima per fare sostituzione, la cerchiamo a livello globale, o locale in un singolo processo? Possono essere realizzate entrambe. Una politica globale è quella in cui quando c'è bisogno di una sostituzione pagina, bufferizzata o no, si cerca una vittima tra tutti i processi, e quindi di fatto un processo può rubare un frame da un altro processo. Questo può avere un impatto non indifferente nei tempi di esecuzione dei processi, nel senso che in un sistema più o meno carico, un certo programma può essere più o meno lento perché dipende anche dall'esecuzione degli altri processi. Tuttavia questa strategia può essere buona dal punto di vista globale per lo sfruttamento del sistema: il throughput finale può essere decoroso. È una strategia che cerca di mantenere il sistema equilibrato e può portare a termine tanti task.

La politica di sostituzione locale, è quella dove, quando un processo ha bisogno di una sostituzione pagina, la vittima è presa tra le pagine del processo stesso. È molto più consistente e tende ad avere prestazioni ripetibili, però può portare a un sottoutilizzo della memoria, nel senso che, se abbiamo dato a un processo troppi frame che lui non utilizza, un altro che ne avrebbe bisogno non riesce ad andare avanti.

Reclaiming Pages

Per implementare una strategia di rimpiazzo globale, cioè nella quale un processo che ha bisogno di un frame potrebbe ottenerlo da un altro processo, tutte le richieste di memoria sono soddisfatte da una lista globale di free frame. Non si aspetta mai che questa lista si svuoti, e quindi ci sono sempre dei frame liberi; quando un processo ha bisogno di un free frame ha garanzia che questo frame ci sia. Il vero rimpiazzo, cioè la ricerca della vittima, viene attivato quando la lista scende sotto una soglia: non abbiamo ancora finito i free frame, ma sono troppo pochi, quindi attiva la ricerca di una vittima, e la strategia è di fare in modo che ci sia sempre un numero di frame liberi equilibrati.



Supponiamo di rappresentare in un grafico, sulle ascisse il tempo, sulle ordinate i frame liberi, quindi la dimensione della free frame list. Se questa nel tempo, a causa di molte richieste di pagine, arriva ad 'a' = il numero di frame liberi è diventato troppo piccolo, allora attiviamo la richiesta di pagine (=la ricerca di vittime, e di sostituzione pagine). Poi, come vengono fatte (più vittime insieme, o una alla volta) è una strategia da decidere. Quando la free list diventa sufficientemente piena, cioè ci sono più pagine, caso 'b', sospendiamo la richiesta di pagine. Ogni volta che la free list diventa troppo piccola, la riattiviamo.

Non-Uniform Memory Access

Questo ha pesantemente a che vedere con l'hardware dei sistemi attuali. Con Non Uniform Memory Access, NUMA, si intende rappresentare il fatto che nei sistemi moderni multicore tendenzialmente più core, più microprocessori sullo stesso sistema hardware, hanno una memoria condivisa, e quindi alla RAM di fatto si accede quasi come se fosse una rete locale (con un protocollo di accesso, in cui la lettura/scrittura in memoria sono più una richiesta di un dato ad un certo indirizzo; è meno predibile il tempo di esecuzione). In questo contesto è plausibile che globalmente la RAM sia vista come se fosse divisa in partizioni di RAM locali a ogni processore. È come se ogni processore avesse la sua RAM, a cui accede velocemente il processore relativo; gli altri processori possono accedere alle altre RAM, ma più lentamente. I processori hanno sì accesso a tutta la RAM, ma ad alcune parti più velocemente, ad altre più lentamente. In questo contesto, sarebbe opportuno che quando un processo che gira su una CPU ha bisogno di frame, li prendesse dalla RAM più efficiente: esiste un concetto di località, di vicinanza di cui tenere conto. Nel sistema Solaris si creano questi Igrops, delle strategie per far sì che quando un processo su una CPU ha bisogno di frame,

tendenzialmente li cerchi in una certa area di memoria, e che si crei un legame tra processo e CPU su cui viene eseguito. Questo meccanismo interagisce anche con la schedulazione dei processi.

Thrashing

Di quanti frame ho effettivamente bisogno? Per capirlo, introduciamo il concetto di thrashing: se il processo non ha abbastanza frame, allora continuerà a paginare e a generare page fault. Se il processo non ha abbastanza pagine in frame, è ovvio che la frequenza di page fault sarà alta. Allora si innesca il thrashing: non ci sono abbastanza pagine in frame, si cerca una nuova pagina, bisogna allocare un frame per questa pagina, e ne buttiamo una con cui rimpiazzarla, che però serve poco dopo, quindi la riportiamo in frame... si innesca un meccanismo che porta ad un degrado di prestazioni molto forte. Questo significa che stiamo cercando di risparmiare memoria, di tenere un alto numero di processi, ma questi processi non riusciranno ad essere eseguiti sulla CPU. Stiamo paradossalmente facendo tutte le azioni per usare di più la CPU: cioè, la multiprogrammazione, eseguire più processi insieme, serve a riempire i buchi di utilizzo della CPU mentre i processi, ae, chiedono I/O. Avere molti processi è positivo dal punto di vista di uso della CPU, ma se si arriva a una certa soglia, un certo grado di multiprogrammazione, succede invece che la CPU diventa inutilizzata perché i processi non riescono ad eseguire non avendo delle pagine: questo è il thrashing.

Demand paging and Thrashing

Se noi cerchiamo di mettere a punto un sistema di paginazione a richiesta che tenga conto del thrashing, dobbiamo tenere conto di quali sono le pagine su cui stanno lavorando i processi. Questo ha portato a osservare che i processi lavorano con una certa **località**. Ad un certo istante tendenzialmente, un processo lavora su certe pagine. Se noi tentassimo di rappresentare su quali pagine sta lavorando in memoria, si ottiene un grafico. Sulle x ho il tempo di esecuzione e sulle y gli indirizzi di memoria o i numeri di pagina, ovvero lo spazio di indirizzamento logico; ogni tacca è un'istruzione (circa, non riusciamo ad avere una grande risoluzione). La figura mostra come il programma lavora con certe località: ae, comincia prevalentemente sulle pagine dal 22 – 26 e qualcosa sul 32, poi si concentra sulle pagine 18 – 24/29 – 30, poi cambia di nuovo; e ae, le pagine dal 26 al 29 non vengono usate fino quasi alla fine del programma. Dunque, un programma in un certo momento dell'esecuzione tende a lavorare su certe pagine: questo perché, ae, in un certo istante sta facendo un loop, o sta chiamando ricorsivamente una funzione, quindi sta costantemente richiamando le stesse istruzioni, una località di istruzioni, e anche sui dati, magari sta lavorando nello stack su certe strutture dati o variabili globali. Un programma migra per località di accessi, non farà mai accessi casuali con distribuzione di probabilità uniforme su tutte le sue pagine. Le località possono sovrapporsi (ae, le località a cui accede al t 7, possono essere un sottoinsieme o un sovrainsieme di quelle a cui accede al t 50).

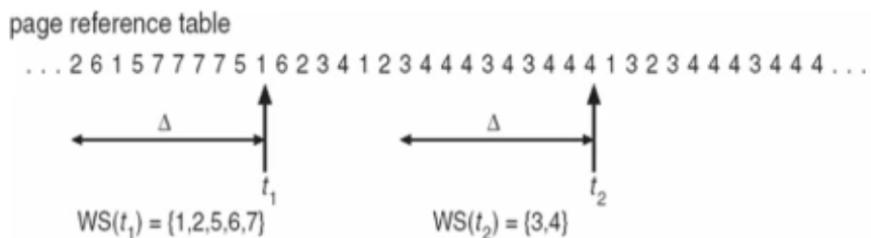
Andiamo sul thrashing quando la sommatoria delle località di tutti processi, cioè le pagine su cui stanno lavorando, sono più grandi della memoria disponibile. Se ci sono un certo numero di processi e la loro località è di 5 GB, e la memoria è di 3 GB, saremo in condizione di thrashing. Cercare di tenere conto della località di accessi sia a livello di politiche di assegnazione di frame che di assegnazione pagine, può essere una strategia vincente.

Working Set Model

Un primo modello molto semplice e rudimentale che cerca di usare il concetto di località è quello del working set. Usiamo una strategia che più o meno abbiamo già usato nelle politiche di scelta della vittima; la località di accessi, è, vagamente parlando, l'insieme di pagine a cui sta lavorando un processo ad un certo istante, un punto nel tempo o un intervallo. Le pagine su cui lavoro ora e su cui lavorerò nell'immediato futuro sono la mia località. Un modo di decidere qual è la località è di dire, le pagine su cui un processo ha lavorato negli ultimi Δ istanti sono la località, sperando che siano quelle su cui continuerò a lavorare per un po' = proiezione del passato verso il futuro.

Chiamiamo working set window, finestra di working set, un intervallo di tempo, e chiamiamo working set le pagine su cui si è lavorato in questo intervallo. Ae, le ultime 10000 istruzioni siano Δ ; la dimensione del

working set del processo p(i) (WSS(i)) è il numero totale di pagine in cui si è fatto riferimento negli ultimi Δ istanti. Nelle ultime 10000 istruzioni un processo potrebbe aver lavorato a 10000 pagine diverse (non troppo probabile), su 1000, su 500, su una sola pagina... il WSS(i) varia in un intervallo compreso tra 1 e Δ , dipende dalla località. Se questo Δ è troppo piccolo, questo WWS non comprenderà la località; se guardo una finestra temporale all'indietro, e questa è troppo piccola, non è detto che le pagine a cui ho fatto accesso negli ultimi Δ istanti comprendano la mia località. Se la finestra è troppo grande, la località la comprende sicuramente, ma fin troppo: se Δ è infinito prende tutto il programma. Dunque, cerchiamo di prendere una finestra temporale Δ adeguata a comprendere la località, non troppo piccola se non ce la facciamo, non troppo grossa se no Δ è più grande della località. Questa finestra temporale Δ viene usata come strumento per identificare il working set che assumiamo essere la località; teniamo come frame D = numero dei demand frame, frame che vengono richiesti = sommatoria dei WSS di tutti i processi: approssimiamo la località con i working set.



Supponiamo di avere una sequenza di riferimenti a pagine, e di avere $\Delta = 10$. Ad un certo istante t_1 , il Δ vede come working set $WS(t_1) = \{1,2,5,6,7\}$; il WSS è 5. Se andiamo avanti un po' di istanti di tempo, con Δ sempre uguale a 10, all'istante t_2 , il WS = {3,4} e il WSS = 2. Se riusciamo a fare in modo che D (i frame allocati, e le pagine sui frame) coincidano con il working set, abbiamo realizzato un modello di sostituzione pagine a working set. Tutto va bene se D riesce a stare in memoria. Questo tipo di politica ha un difetto: a t_1 , il ws è fatto da 5 pagine, a t_2 , il ws è fatto da 2; dopo ogni istante il ws potrebbe rimanere uguale, aumentare o diminuire. Se noi passassimo da t_1 a t_1+1 , alla pagina 6, già nel ws, non ci sarebbe page fault: ma mantenendo $\Delta=10$, devo buttare via 2, dunque il ws perde la pagina 2. Questo è il difetto di una politica di working set esatto.

Keeping Track of the Working Set

Tenere traccia del working set è costoso, perché occorre dopo ogni accesso allineare il working set al cosiddetto resident set, insieme dei set residenti. Si tende ad approssimare questo tipo di strategia. Un modo di approssimare il modello di funzionamento a working set è di avere una specie di campionamento dopo un certo intervallo di istanti. Supponiamo di avere $\Delta = 10000$; invece di guardare ad ogni accesso, quali sono le pagine a cui si è fatto accesso negli ultimi 10000 istanti con una sliding window, facciamo un'osservazione ogni 5000 istanti. Ogni 5000 istanti dunque attiviamo un interrupt che, per ogni pagina, memorizza il valore del reference bit e poi lo azzera. Durante i prossimi 5000 istanti, ogni accesso a pagina mette a 1 il reference bit. In breve, supponiamo che ci sia un istante in cui tutti i reference bit sono a 0; nei prossimi 5000 istanti le pagine a cui faccio accesso avranno il reference bit a 1. Alla fine dei 5000 istanti ci saranno un po' di pagine con reference bit a 1, e un po' a 0; guardando solo i reference bit posso discriminare le pagine a cui ho fatto accesso o no negli ultimi 5000 istanti. Volevamo 10000: se noi ci ricordiamo anche il reference bit nel penultimo intervallo di 5000, potremmo osservare in un certo istante le pagine a cui ho fatto accesso negli ultimi 5000 istanti, quelle a cui ho fatto accesso nei 5000 precedenti, quelle con solo un accesso, quelle con nessuno. Con due bit, tengo traccia degli accessi nei 5000 e precedenti 5000 istanti. Se questa informazione la uso andando avanti nei page fault che farò nella prossima time window di 5000 istanti, notiamo come sto in realtà sovrapprossimando il working set, il Δ . Questo è una finestra a campionamento fisso.

Page Fault Frequency

Un'altra strategia di approssimazione, la page fault frequency, che invece di tenere l'idea del window Δ , si approccia alla località in modo diverso: cioè, invece di andare a ricercare la causa, si basa sull'effetto. In sostanza consiste che se hai troppi page fault, allarga il resident set = numero di frame, se nei hai pochi puoi restringerlo. La dinamicità non si basa sull'indovinare la località, ma nell'adattare il numero di pagine su cui lavori. Siccome l'obiettivo della località degli accessi è quello di minimizzare i page fault, guardiamo alla fine i page fault.

Se rappresentiamo su un grafico il numero di frame disponibili e la frequenza di page fault (numero totale di page fault diviso numero di accessi), se aumentiamo i frame diminuiranno i page fault, e viceversa. Teniamo conto che nella politica di page fault frequency ci sono due soglie: una soglia oltre la quale i page fault sono troppi, e aumentiamo il numero di frame, e una sotto cui diminuiamo il numero di frame. Un modo per realizzare in modo semplice questo algoritmo è quello di avere una sola soglia, anziché due: l'idea è di non mettere un interrupt, un time out, ma lavoriamo solo quando dobbiamo lavorare "per forza" perché c'è un page fault. Ad ogni page fault, e quando c'è un page fault bisogna comunque perdere tempo, misuriamo la distanza in tempo dall'ultimo page fault: questo è semplice, basta tenere una variabile da qualche parte che se lo ricordi. Chiamando T la distanza dall'ultimo page fault, se T è grande \rightarrow l'ultimo page fault è lontano nel tempo \rightarrow la frequenza di page fault è bassa, va bene. Se T è piccolo \rightarrow la frequenza di page fault è alta, non va bene. Definiamo una costante c , determinata anche sperimentalmente, e diciamo che se $T < c$, i page fault sono vicini, la frequenza è più alta di ciò che vogliamo \Rightarrow allora aggiungiamo un frame al resident set, senza sostituzione pagina. Se invece $T \geq c$, stiamo andando bene, la frequenza di page fault è adeguata all'obiettivo, quindi possiamo usare la sostituzione pagina, rimuovendo una pagina dal resident set. La vittima la determiniamo tra le pagine a cui non abbiamo fatto accesso dall'ultimo page fault: usiamo come finestra di osservazione il tempo intercorso dall'ultimo page fault. Per far ciò, abbiamo bisogno di un reference bit settato a ogni riferimento a pagina, e resettato l'ultima volta che abbiamo fatto una sostituzione pagina. Ogni volta che T è maggiore di c , cerchiamo una vittima tra le pagine con reference bit a 0, e intanto resettiamo il bit in tutte le pagine del resident set.

Tempo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Stringa	6	4	4	3	2	4	4	4	1	3	3	2	4	4	5	1	2	2	2	6	1	2	5	4
Frame 0	6	6	6	6	6	6	6	6	6	6	6	6	6	5	5	5	5	5	5	6	6	6	6	6
Frame 1	-	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	2	2	2	2	2
Frame 2	-	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	1	1	1	1	1
Frame 3	-	-	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	-	-	-	5	5
Frame 4	-	-	-	-	-	-	1	1	1	1	1	1	1	1	1	1	1	1	1	-	-	-	-	4
Fault	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
Ref Bit										0		0		0										

C=3
Page referenced
victim

Ae, decidiamo che $c = 3$, e partiamo con riferimento alla pagina 6; la mettiamo nel resident set, è la prima. Pagina 4, un altro page fault, aggiungiamo la pagina 4 al resident set, e notiamo che sotto ci sono 2 page fault a distanza 1: $1 < 3$, significa che continuiamo ad aggiungere frame. Terzo riferimento, di nuovo pagina 4, no page fault. Quarto riferimento, pagina 3, un page fault che dista 2 dal precedente page fault: $2 < 3$, continuiamo ad aggiungere. Facciamo accesso a 2, che genera un page fault, a distanza 1 dal precedente: troppo piccolo di nuovo, continuiamo ad aggiungere. Poi abbiamo riferimento a 4 tre volte, nessun page fault; il prossimo page fault è scatenato dalla pagina 1, ed è a distanza 4 dal precedente: questo aggiunge 1 al resident set, e azzera i reference bit. Prima di questo, tutti i reference bit erano a 1. Una volta qui, partiamo con il nostro algoritmo: ogni volta che da qui in avanti sarà usato il verde per indicare il reference bit = 1. Facciamo accesso a 3, e poniamo reference bit a 1 = diventa verde. Etc etc fino al secondo 4. Per ora abbiamo 3 pagine con reference bit a 1 e 2 con reference bit a 0. Facendo accesso alla pagina 5, abbiamo un

page fault sufficientemente lontano dal precedente, sicuro $>3 \Rightarrow$ sostituzione pagina: dobbiamo decidere chi è la vittima a cui sostituiremo il 5. Le vittime sono le pagine con reference bit a 0, che vengono TUTTE buttate fuori: la 6 viene rimpiazzata, la 1 buttata. Andando avanti, riferimento a 1, page fault troppo vicino, aggiungiamo la pagina, con reference bit a 1; poi il 2, reference bit a 1. quando arriva il prossimo page fault, pagina 6, la distanza è maggiore di 3: salviamo le pagine con reference bit a 1, buttiamo via le altre. Qui sono traslati i frame per questioni di compattazione; possiamo lasciarli sia così, sia dov'erano. Con questo algoritmo, è dunque possibile avere un resident set dinamico; sui tempi lunghi, se noi cerchiamo di mantenere una distanza c tra un page fault e l'altro, la frequenza di page fault dovrebbe essere $1/c$. Il confronto è tra T e c , tendenzialmente la frequenza di page fault potrà essere minore di $1/c$ o maggiore a seconda di come usiamo il T e di conseguenza il resident set.

In realtà il metodo di page fault frequency ha due soglie, una oltre cui dobbiamo cominciare ad attivare la sostituzione pagine, e l'altra oltre cui dobbiamo smettere di attivarla, ma l'algoritmo è più complicato.

Working set e page fault rates

Notiamo che il working set, cioè le pagine su cui ha lavorato un programma, la località, e le frequenze di page fault hanno un andamento non uniforme. Un programma per un po' continua a lavorare su una certa località, e farà pochi page fault; ad un certo punto tenderà a farne tanti vicini, perché sta trasferendo la sua località, e quindi cambia le pagine. Poi di nuovo i page fault diminuiranno mentre il programma lavora sulla nuova località. L'andamento della frequenza è a picchi e a intervalli bassi. Ovviamente, sarebbe bene che il working set coprisse almeno la distanza tra due picchi.

Allocating Kernel Memory

Per ora ci siamo occupati della memoria da allocare ai processi utente, e l'obiettivo era di fare in modo che in un sistema girassero molti processi, usando meno memoria possibile. E il kernel, e il SO? Allora, la prima osservazione da fare è: anche SO può paginare però ha bisogno di alcune strutture dati allocate in modo contiguo, almeno in parte. Ae, la tabella delle pagine DEVE essere allocata contigua, perché è un vettore ad accesso diretto. Riguardo al kernel, spesso le sue strutture dati sono allocate in una memory pool dedicata al kernel, e alcune strutture dati hanno bisogno di memoria contigua.

Buddy System

Qui citiamo due tipi di allocator per il kernel. Il primo dei due, il buddy system è un allocatore per kernel che cerca di allocare in modo contiguo, ma cercando di ridurre il problema della frammentazione esterna. L'allocatore contiguo puro a partizioni variabili ha troppa frammentazione esterna: quindi per il kernel si tende di evitare la paginazione pura, e si cerca di allocare contiguo in tagli abbastanza uniformi per ridurre l'effetto della frammentazione. Il buddy system alloca per potenze di 2: si cerca di allocare per una potenza di 2, arrotondando alla potenza di 2 più grande di quella che serve. Ae, il buddy allocator tiene delle pagine contigue di una certa dimensione a potenza di 2, e ritaglia delle partizioni allocate di dimensione più piccola ma sempre multipli di potenze di 2. Supponiamo che ci sia una partizione libera di 256 KB, e il kernel richieda 21 KB, questa partizione sarà divisa in due parti da 128 KB; una delle due sottopartite sarà ulteriormente divisa in due parti da 64 KB, e ulteriormente da 32 KB. Una di queste da 32 KB sarà poi usata per l'allocazione.

Slab Allocator

L'alternativa è lo slab allocator, che non usa il concetto di potenza di 2, ma cerca comunque di avere un doppio livello di allocazione cercando di mantenere delle dimensioni un po' uniformi. Si parla di slab e di cache: uno slab, è una o più pagine contigue, una cache è fatta da uno o più slab. Si cerca di allocare per un tipo di struttura dati del kernel una singola cache, fatta di slab. Quando si crea una cache, quindi, la si riempie di oggetti liberi che vengono allocati quando vengono usati per una struttura dati del sistema. C'è meno frammentazione.

Supponiamo di avere nel kernel struct di dimensione 3 KB e struct o vettori di dimensione 7 KB. Le strutture

dati da 3KB vengono piazzate in una cache composta da slab, pagine contigue in memoria, mentre quelle da 7 in un'altra cache che fa riferimento ad altri slab.

Slab Allocator in Linux

In Linux, ad esempio, si usa lo slab allocator, più complicato di quello che ho detto, ma l'idea di partenza è questa. Supponiamo, ae, che per una struct task (=descrittore che serve per una particolare azione del SO) ci sia bisogno di 1,7KB di memoria. Ogni task creata prende e ritaglia una struct da una cache opportunamente allocata con degli slab. Ogni slab nella cache può essere pieno, vuoto o parzialmente pieno/vuoto. Quando c'è una richiesta uno slab allocator usa una struct in uno slab parziale, e se non ce n'è prende un'altra empty slab, e via così.

Anche Solaris ha uno slab allocator; Linux 2.2 ha varianti di uno slab, che si chiamano SLOB e SLUB, due varianti di questa politica di base. Quasi sempre si parte con strategie semplice, poi le si complica per casi particolari o ottimizzazione.

Other Considerations

Arriviamo ad alcune considerazioni finali sulla paginazione. Nell'ordine:

- **Prepaging:** per diminuire il numero di page fault che succedono all'inizio (i primi accessi a memoria generavano tutti page fault, quindi la frequenza di page fault sarà elevata allo start up di un processo). Per ovviare a questo può essere conveniente prepaginare, cioè cominciare a portare in frame alcune delle pagine di cui il processo avrà bisogno all'inizio. Questa però può essere un'arma a doppio taglio: possiamo prepaginare qualcosa che non serve. Si assume che prepaginiamo s pagine, e sia a la percentuale delle pagine effettivamente usate (compresa tra 0 e 1). $s*a$ sono le pagine usate, e $s(1-a)$ quelle non usate. Se a tende a 0 non va bene: sarebbe meglio se tendesse a 1, prepaginando solo le pagine che servono.
- **Page Size:** abbiamo già visto, quando abbiamo osservato le architetture dei microprocessori, che tendenzialmente i microprocessori mettono a disposizione più dimensioni di pagine. Dal punto di vista della frammentazione interna, sarebbero opportune pagine piccole; ma pagine piccole aumentano la dimensione delle tabelle delle pagine, che vorrebbero invece pagine grandi. La risoluzione sta a indicare in quanto frammentiamo lo spazio di indirizzamento. Parlando dell'overhead di I/O (copia di pagine da disco verso disco), convengono pagine grandi perché ammortizzano meglio il costo dell'I/O: le pagine vengono almeno dimensionate a KB perché sono unità di trasferimento adeguate per i dischi, di meno no. Il numero di page fault tenderebbe a diminuire con pagine grosse. Per la località di accesso, meno pagine, quindi pagine più grosse, fanno rappresentare la località con meno pagine: una pagina grossa comprende un po' di località un po' no. Per la TLB, le pagine grosse vanno meglio. Ci sono un po' di aspetti per cui van bene le pagine piccole, e un po' per cui vanno meglio le grosse; le dimensioni delle pagine sono aumentate progressivamente con l'aumento della dimensione della RAM e degli spazi di indirizzamento. Di solito, comunque, sono dimensionate a multipli di potenze di 2, con un intervallo che va da 4 KB a 4 MB, circa.
- **TLB reach:** la copertura della TLB, le pagine raggiungibili tramite TLB, ha molto a che vedere con quante righe ci sono in tabella. Quanto copre la TLB è dato dal numero di righe, quindi di coppie pagina-frame, rappresentabili in tabella, e dalla dimensione della pagina, o frame (più è grande, più la TLB copre, ma questo porta ad un aumento della frammentazione). Avere pagine di dimensione multipla potrebbe avere un senso, ma può creare complicazioni nella gestione delle pagine.
- **Program structure:** diamo un'occhiata al concetto di località, e cerchiamo di capire come la scrittura di un programma può impattare la località; il software può determinare più o meno page fault. Supponiamo di avere due programmi con una matrice con 128*128 interi, con due iterazioni diverse:

```

1)   for (j = 0; j < 128; j++)
      for (i = 0; i < 128; i++)
          data[i,j] = 0;
2)   for (i = 0; i < 128; i++)
      for (j = 0; j < 128; j++)
          data[i,j] = 0;

```

dove i è l'indice di riga e j di colonna. In sostanza, uno dei due programmi percorre la matrice per colonne e l'altro per righe; nel secondo programma, quello che va meglio, siccome generalmente i linguaggi di programmazione e i compilatori che stanno dietro allocano le matrici per righe, (cioè i 128 interi della prima riga stanno tutti vicini l'uno all'altro e poi iniziano quelli della seconda, e così via), la località del secondo programma tende ad essere più uniforme, prima percorre tutta la prima riga, poi la seconda, e così via. Supponiamo che ogni riga stia in una pagina (supponendo un intero su 4B). Il secondo programma avrà quindi 128 page fault, uno per ogni cambio di pagina/riga. Il primo programma, invece, andando per colonne, accederà alla prima casella della prima riga, poi la prima casella della seconda e così via, ad ogni accesso farà page fault, perché cambia pagina. Dunque, supponendo che quando arrivi alla seconda colonna, la prima pagina sia già uscita dal resident set, farà 128*128 page fault. Come è stato scritto un programma può avere impatto; è probabile che un ottimizzatore potrebbe riuscire a salvare qualcosa, però il programmatore ha le sue responsabilità nello scrivere programmi efficienti in rapporto alla paginazione.

- **I/O interlock:** quando una pagina è coinvolta in un'operazione di I/O, destinazione o sorgente dei dati in trasferimento da disco, la pagina va bloccata e non si può rimpiazzarla: si chiama pinning l'azione di bloccare la pagina.

Operating System Examples

- **Windows:** usa paginazione a richiesta con una forma di clustering, che porta in memoria, quando c'è bisogno di una pagina, anche pagine vicine sperando che appartengano alla località. Ai processi è assegnato un working set minimo e uno massimo, dimensioni minime e massime dell'insieme di pagine a cui si fa accesso, dunque il resident set può variare. A un processo si assegnano un numero di pagine che va fino al massimo; se si scende sotto il minimo si fa una rivisitazione, una "messa a punto" del working set per rimettere a posto la memoria libera. L'**automatic working set trimming** toglie pagine ai processi che ne hanno in eccesso e le dà a chi ne ha bisogno. I working set lavorano tra due soglie: è una via di mezzo del working set esatto e il page fault frequency.
- **Solaris:** mantiene liste di pagine libere da assegnare ai processi che fanno page fault, poi ha varie soglie che non vediamo nel dettaglio (ae, una soglia per iniziare a paginare, una per aumentare la paginazione e per fare swapping su un processo, uno che fa page-out che usa una variante dell'algoritmo clock, simile al FIFO...). In definitiva, ci sono queste soglie che cercano di tenere conto di quanta memoria libera c'è e di qual è la frequenza di scan che viene attivata. Anche il Solaris usa varianti di queste politiche di cui abbiamo parlato.

Tutti i SO attuali o non fanno paginazione perché costosa, o usano gli algoritmi di cui abbiamo parlato, adattati alle esigenze dei sistemi e eventuale sperimentazione fatta.

3. Introduzione a OS161

Iniziamo con un ripasso sui concetti di thread e processo, già affrontati precedentemente.

Threads

Il thread e il processo rappresentano in qualche modo lo stato di controllo di un programma in esecuzione; l'idea alla base è quella di una serie di istruzioni mandate in esecuzione. Lo stato, o contesto, è tipicamente rappresentato dallo stato della CPU e da uno stack. La memoria RAM associata a un processo è costituita da codice (*program code*), dati e stack (dati dinamici). Il codice e i dati NON fanno parte del contesto, mentre lo stack sì.

Uno user thread è un thread messo a disposizione dei programmi; questo thread è gestito a livello di programma utente, ovvero in user space, quindi non si tratta di “attivare” quelli che sono i kernel threads. Le librerie principali per la creazione di user threads sono POSIX Pthreads, Windows threads e Java threads.

Un kernel thread viene generato a livello kernel, e schedula i singoli thread come entità separate; sono supportati da tutti i principali sistemi operativi, sia fissi che mobile.

Da un certo punto di vista, gli user threads sono più “leggeri”; tuttavia, deve esistere un mapping tra user e kernel threads, e può essere:

- *one-to-one*: per ogni user thread corrisponde esattamente un kernel thread;
- *many-to-one*: più user threads corrispondono a un kernel thread;
- *many-to-many*: più user threads mappati su più kernel threads.

Processi

Un processo è un programma in esecuzione. Oltre alle istruzioni, dette talvolta *text sections*, un processo è costituito da un program counter, da uno stack, da una sezione di dati (variabili globali) ed eventualmente da un heap, dove vi sono le allocazioni dinamiche. Il programma è quindi un’entità passiva che risiede su disco, che diventa processo quando entra in esecuzione: ciò significa che lo stesso processo può generare diversi processi.

In memoria, dal punto di vista logico, un processo è caratterizzato da un intervallo di indirizzi contigui, detto logic address space, con di solito testo, dati e heap da una parte e stack dall’altra, generando una sorta di “buco” dove generalmente finiscono le librerie condivise. Nel caso di un semplice programma C, ad esempio, abbiamo il codice nella sezione di testo, le variabili globali suddivise per inizializzate e non inizializzate nella sezione dei dati, i parametri al main verosimilmente alla cima dello stack, e le variabili locali nello stack; se si allocano delle variabili (es. malloc), si va a incidere sull’heap.

Un processo può essere single threaded o multi threaded. I thread condividono il codice, dati globali e files, mentre il contesto (quindi registri, PC e stack) è caratteristico di ciascun thread.

Threads in OS161

Scendiamo nello specifico per OS161. La memoria è strutturata in maniera analoga a come descritta precedentemente, con stack da una parte, dati e codice dall’altra.

Una libreria consente al programmatore di scrivere codice per creare, gestire e distruggere thread. In OS161 c’è una libreria che è in grado di gestire thread e il loro contesto. La struttura dati su cui si appoggia questa libreria è una struct chiamata *thread*:

```
struct thread {
    char *t_name;                      /* Name of this thread */
    const char *t_wchan_name;           /* Name of wait channel, if
                                         sleeping */
    threadstate_t t_state;              /* State this thread is in */

    /* Thread subsystem internal fields. */
    struct thread_machdep t_machdep;
    struct threadlistnode t_listnode;
    void *t_stack;                     /* Kernel-level stack */
    struct switchframe *t_context;     /* Saved register context (on
                                         stack) */
```

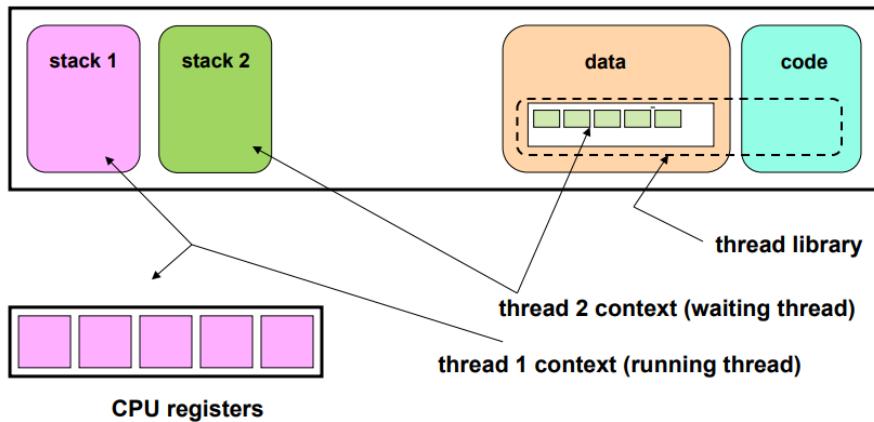
```

    struct cpu *t_cpu;           /* CPU thread runs on */
    struct proc *t_proc;         /* Process thread belongs to */
    ...
};

}

```

Di seguito si riporta una possibile configurazione della memoria in OS161 con 2 thread kernel, uno running e uno waiting (notare che per il secondo lo stack esiste, mentre i registri sono stati salvati nella struttura dati della thread library):



Lo switchframe, che di fatto indica il contesto e serve per salvare dati in context switch, risiede nello stack; come visto prima, la struct thread tiene rispettivamente un puntatore allo stack e alla struct switchframe.

Vediamo ora una lista di funzioni della libreria thread:

- Funzioni eseguite una tantum, al bootstrap/shutdown

`thread_bootstrap`
`thread_start_cpus`
`thread_panic`
`thread_shutdown`

- Funzioni esterne per la gestione dei threads

`thread_fork`
`thread_exit`
`thread_yield` `thread_consider_migration`

- Funzioni interne (chiamate dalle esterne)

`thread_create`
`thread_destroy`
`thread_make_runnable`
`thread_switch`

Di queste, scendiamo più nel dettaglio di alcune:

- `int thread_fork (const char *name, struct proc *proc, void (*entrypoint)(void *, unsigned long), void *data1, unsigned long data2);`

Crea un nuovo thread, che inizierà l'esecuzione dalla funzione "entrypoint", che riceve 2 parametri, un puntatore e un intero, i quali sono passati anche alla funzione `thread_fork` stessa. Il thread apparterrà al processo "proc", o al processo del thread corrente se "proc" è null.

```

thread_fork(..., void (*entrypoint)(void *, unsigned long), void
*data1, unsigned long data2) {
...
newthread = thread_create(...);
...
switchframe_init(newthread, entrypoint, data1, data2);
thread_make_runnable(newthread, false);
}

thread_create(...) {
thread = kmalloc(sizeof(*thread));
thread->... = ...;

return thread;
}

switchframe_init(...) {
/* setup switchframe in stack */
}

thread_make_runnable(struct thread *target) {
...
target->t_state = S_READY;
threadlist_addtail(&targetcpu->c_runqueue, target);
...
}

```

Entrando nel corpo della funzione, troviamo una `thread_create`, la quale effettua l'allocazione della struct `thread`, la `switchframe_init`, che “mette a posto” uno switchframe nello stack (ovvero inizializza il contesto), e la `thread_make_runnable`, la quale setta lo stato del thread a READY e lo accoda tra i thread pronti per l'esecuzione.

```

thread_switch(threadstate_t newstate, ...) {
    struct thread *cur, *next;

    cur = curthread;

    /* Put the thread in the right place. */
    switch (newstate) {
        case S_RUN:
            panic("Illegal S_RUN in thread_switch\n");
            case S_READY:
                thread_make_runnable(cur, true /*have lock*/);
                break;
    }
    next = threadlist_remhead(&curcpu->c_runqueue);

    /* do the switch (in assembler in switch.S) */
    switchframe_switch(&cur->t_context, &next->t_context);
}

```

```
    ...
}
```

Ad un certo punto verrà chiamata la funzione `thread_switch`, che sostanzialmente gestisce un avvicendamento sulla CPU, cioè ci sarà una chiamata di un cambio di stato: essa riceve come parametro `newstate`, ovvero il nuovo stato, e se il nuovo stato è `READY` il thread diventa `RUNNABLE`. La `threadlist_remhead` determina il thread successivo da mandare in esecuzione. Infine, la `switchframe_switch` esegue il context switch salvando il processo corrente e ripristinando il successivo, mandando in esecuzione il prossimo thread. Più nel dettaglio, quest'ultima funzione è scritta in assembler, perché esegue operazioni di basso livello: nel contesto del thread corrente va a salvare uno a uno i registri nello stack, e nei registri va a mettere il contesto del thread successivo. Per farlo, inizializza lo stack pointer (`sp`) decrementando il valore attuale dello `sp` stesso di 44 (ossia 4 byte * 10 registri, + altri 4 byte), e facendo store word (`sw`) dei vari registri (e ovviamente per ciascuno incrementando di 4 lo `sp`). Poi, si mette il puntatore allo `sp` a 0 e si salva nel contesto. Successivamente, si carica nello `sp` il puntatore del nuovo processo entrante e si effettua load word (`lw`) per ciascun registro. Le operazioni di store e load sullo `sp` sono quelle che determinano l'effettivo cambio di stack tra i 2 processi. Alla fine di tutto, si aggiunge 40 al valore dello `sp`.

- `__DEAD void thread_exit(void);`

Termina il thread corrente. Gli interrupts non devono essere disabilitati.

- `void thread_yield(void);`

Il thread corrente termina, e darà la precedenza al successivo thread nello stato `RUNNABLE`, rimanendo anch'esso `RUNNABLE`. Gli interrupts non devono essere disabilitati.

Vediamo ora cosa succede quando, oltre a creare un thread di kernel, si crea un processo utente. In memoria ci sarà sempre il kernel, ma troveremo anche un processo (che possiamo chiamare applicazione) che avrà un suo stack, suoi dati e suo codice; in un certo istante, possono essere caricati nei registri perché in esecuzione sulla CPU. Tralasciamo per ora l'heap; i processi saranno in grado di usare solo variabili globali.

In OS161, la PBC (Process Control Block) è identificata dalla struct `proc`, qua riportata:

```
struct proc {
    char *p_name;           /* Name of this process */
    struct spinlock p_lock; /* Lock for this structure */
    unsigned p_numthreads; /* Number of threads in this process */

    /* VM */
    struct addrspace *p_addrspace; /* virtual address space */

    /* VFS (Virtual File System) */
    struct vnode *p_cwd; /* current working directory */

    /* add more material here as needed */
    ...
};
```

Si noti che un processo sa quanti sono i suoi thread, ma non ha la lista esplicita dei thread (mentre il thread ha il puntatore al processo a cui appartiene).

Quando c'è un processo, questo viene generato a partire da un thread di kernel, e avrà, oltre a stack, dati e codice lato applicazione, uno stack lato kernel usato quando il processo farà delle system call e

sostanzialmente richiede dei servizi al kernel: in generale, in OS161 un thread ha 2 stack, uno usato quando sta eseguendo codice non privilegiato (ovvero il programma user), e uno quando usa codice di kernel privilegiato (essenzialmente system calls). Vedremo che la creazione di un processo parte dalla creazione di un kernel thread, il quale diventerà un processo dopo aver allocato la memoria per codice, dati e stack user. Ricordiamo che la struct thread a ha un puntatore al kernel stack, mentre la struct proc ha un puntatore all'address space (memoria usata dal processo quando è in modalità user), la quale contiene uno user stack (e altri 2 segmenti per ora non trattati).

Quando si vuole eseguire uno user program, dal menu OS161, si scrive p seguito dal nome del file ELF ed eventuali argomenti. In OS161, ci sono 3 programmi che possono essere eseguiti che sono dei test per kernel threads:

- tt1: chiama threadtest->runthreads(1/*loud*/) che genera NTHREADS (8) threads eseguendo loudthread. L'output di questo programma sono caratteri generati dagli 8 threads;
- tt2: chiama threadtest2->runthreads(0/*quiet*/) che genera NTHREADS (8) threads eseguendo quiertthread. I threads effettuano busy wait (200000 per ciascuna iterazione) seguiti dall'output di un carattere;
- tt3: chiama threadtest3->runtest3 che genera un certo numero di thread per fare lavori sostanzialmente di sincronizzazione.

Tracciando l'esecuzione di uno user program, si noterà che c'è una funzione detta cmd_prog che chiama common_prog che chiama proc_create_runprogram, la quale crea un processo utente col compito di eseguire un certo file eseguibile. Per creare questo processo utente, si chiama una thread_fork che esegue cmd_proghread che esegue runprogram, la quale fa il lavoro di creazione del processo, ossia genera l'address space, legge e carica il file ELF e infine fa partire il processo. Si riporta la runprogram:

```
int runprogram(char *progname) {
    struct addrspace *as;
    struct vnode *v;
    vaddr_t entrypoint, stackptr;
    int result;

    /* Open the file. */
    result = vfs_open(progname, O_RDONLY, 0, &v);
    ...
    /* Create a new address space. */
    as = as_create();
    ...
    /* Switch to it and activate it. */
    proc_setas(as);
    as_activate();

    /* Load the executable. */
    result = load_elf(v, &entrypoint);
    ...

    /* Done with the file now. */
    vfs_close(v);
```

```

/* Define the user stack in the address space */
result = as_define_stack(as, &stackptr);
...
/* Warp to user mode. */
enter_new_process(0/*argc*/, NULL/*userspace addr of argv*/,
                  NULL/*userspace addr of environment*/,
                  stackptr, entrypoint);

/* enter_new_process does not return. */
panic("enter_new_process returned\n");

return EINVAL;
}

```

Running a user program

Esaminiamo come si fa a mandare in esecuzione un programma utente in OS161, avevamo visto come a partire da un comando P, nome dell'eseguibile con tutta una catena di chiamate si arriva ad eseguire **Runprogram** nel contesto di un thread di kernel.

Runprogram

Questo Runprogram ha come obiettivo, mettere a punto l'Address Space di un processo caricarci l'eseguibile e farlo partire.

```

/* see kern/syscall/runprogram.c */
int runprogram(char *progname) {
    struct addrspace *as;
    struct vnode *v;
    vaddr_t entrypoint, stackptr;
    int result;

    /* Open the file. */
    result = vfs_open(progname, O_RDONLY, 0, &v);
    ...
    /* Create a new address space. */
    as = as_create();
    ...
    /* Switch to it and activate it. */
    proc_setas(as);
    as_activate();
}

```

La partenza dopo aver fatto, **load_elf** viene demandata alla funzione **enter_new_Process** che riceve come parametri, per il momento Nulli, sarebbero poi da mettere a posto nel momento in cui se li volesse gestire, **argc*** **argv***, un ambiente anche questo nullo, ma quello che conta un puntatore allo stack, **stackptr**, per il processo che deve essere avviato e un **entrypoint** che rappresenta l'istruzione che va eseguita per prima.

```

/* Load the executable. */
result = load_elf(v, &entrypoint);
...
/* Done with the file now. */
vfs_close(v);

/* Define the user stack in the address space */
result = as_define_stack(as, &stackptr);
...
/* Warp to user mode. */
enter_new_process(0/*argc*/, NULL/*userspace addr of argv*/,
                  NULL /*userspace addr of environment*/,
                  stackptr, entrypoint);

/* enter_new_process does not return. */
panic("enter_new_process returned\n");
return EINVAL;
}

```

Enter_new_process

è una funzione che trucca un avvio di un processo utente, facendo finta che ci sia un processo utente che ritorna da una trap o da un interrupt, cioè mette a posto il cosiddetto **trapframe** che è una struttura dati, nella quale normalmente un processo vede salvare il suo contesto, quando viene interrotto.

Ci sono due strutture dati in cui viene salvato il contesto di un processo per poi essere ripristinato, lo switch frame è il salvataggio in corrispondenza di un context switch.

Cioè un processo piuttosto che non un thread

viene salvato il suo contesto quando la CPU schedula. Un altro tipo di salvataggio è quello in cui un processo o un qualunque thread di Kernel viene interrotto, da una chiamata di interrupt o di trap e alla fine ci sarà una return from interrupt o una return from trap.

```
/* see kern/arch/mips/locore/trap.c */
void enter_new_process(int argc, userptr_t argv, userptr_t env,
                      vaddr_t stack, vaddr_t entry) {
    struct trapframe tf;
    bzero(&tf, sizeof(tf));
    tf.tf_status = CST_IRQMASK | CST_IEP | CST_KUP;
    tf.tf_epc = entry;
    tf.tf_a0 = argc;
    tf.tf_a1 = (vaddr_t)argv;
    tf.tf_a2 = (vaddr_t)env;
    tf.tf_sp = stack;
    mips_usermode(&tf);           I
}
void mips_usermode(struct trapframe *tf) {
    ...
    /* This actually does it. See exception-.S. */
    asm_usermode(tf);
}
```

Per far partire un processo utente lo si sistema, come se la prima istruzione da eseguire fosse successiva a una return from interrupt o from trap. La `enter_new_process` ha una variabile `struct trapframe`, che è compatibile con quello che è il ritorno da un trap, in MIPS lo si azzerà e si mettono a posto alcuni campi, qui si mettono a posto uno stato, ma soprattutto entry program counter il `program_counter` iniziale che è l'entry e poi ci sono `argc*` e `argv*` che in questo caso non sono significative, lo sarebbero poi in un altro momento, poi l'ambiente e lo stack quindi lo stack pointer è nel trapframe. `mips_usermode` usa lo stack frame e fa un ulteriore livello di chiamata, cioè `enter_new_process` fa da wrapper a `mips_usermode`, avendo predisposto il `thread_frame`.

mips_usermode dopo un pò di istruzioni fa una chiamata a `asm_usermode` ricevendo come paramtro il `trap_frame`, o in questo caso il puntatore a `trap_frame`. `mips_usermode` è una funzione assembler che se andassimo a vederla:

```
/* see kern/arch/mips/locore/exception-mips1.S */
asm_usermode:
/* a0 is the address of a trapframe to use for exception "return".
 * It's allocated on our stack.
 * Move it to the stack pointer - we don't need the actual stack
 * position any more. (When we come back from usermode, cpustacks[]
 * will be used to reinitialize our stack pointer, and that was
 * set by mips_usermode.)
 * Then just jump to the exception return code above.
 */
j exception_return
addiu sp, a0, -16                         /* in delay slot */
.end asm_usermode

exception_return:
/* restore registers from trapframe */
```

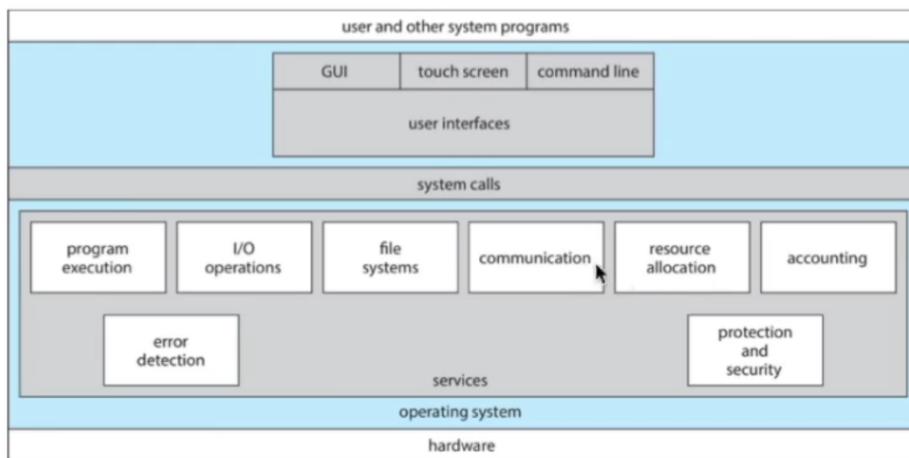
L'ultima parte di questo file è quella che gestisce l'entrata in user_mode di un processo con un re direzione Jump a exception Return a uscita dalla return.

Ricordatevi che l'Assembler MIPS ha un delay slot perché è un processore con pipeline e penso che alcuni di voi conoscano l'Assembler mips altri no non è così fondamentale in questo corso conoscere Assembler, però capire di cosa si parla quindi vuol dire semplicemente che questo calcolo di Stack Pointer come il contenuto di a0 dove si è sottratto 16 è un'istruzione che viene eseguita prima di arrivare alla Jump exception Return. Exception return è una parte significativa del codice che fa un ripristino dei Registri prendendo le tread_frame, troviamo i 3 puntini (...) per dire che ci sono più istruzioni che potete vedere nel codice della funzione, e alla fine sostanzialmente fa un return from exception.

Ripeto i dettagli possono essere visti potete provare anche a fare una esecuzione in debug mettendo un breakpoint su Mips_usermode e vedere che cosa viene effettuato.

A view of Operating System Services

Vediamo quali sono i servizi forniti da un sistema operativo perché vogliamo rivedere le System Call. Le system call sono l'interfaccia del sistema operativo verso tutta una serie di servizi che sono forniti per l'esecuzione dei programmi, per le operazioni di I-O, il file Systems, le comunicazioni, l'allocazione di risorse ed altro.

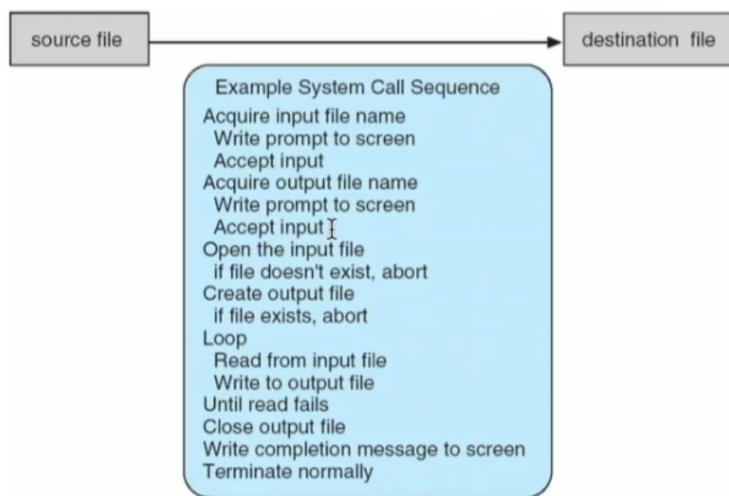


In sostanza un programma utente, User, anche altri programmi di kernel sistema ovviamente ma un programma utente, può interfacciarsi con delle System Call che sono il livello intermedio che permette al lato utente di ricevere servizi da parte delle varie componenti del sistema operativo.

System Call

Le System Call sono tipicamente scritte in un linguaggio di alto livello C/C++ dipende dal contesto, le parti che vedremo saranno scritte in C se dovete interfacciarsi in un contesto Windows con la grafica user interface avreste il C++.

Ci sono alcune interfacce standardizzate ad esempio Win32 è un'interfaccia verso servizi nel sistema Windows, le API posix sono disponibili ad esempio in Unix, Linux, MacOS e altro ci sono anche interfacce Java verso java virtual machine. Ci occuperemo di capire come sono realizzate le System Call:



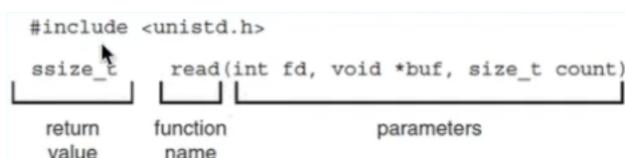
Ad esempio sopra vediamo una sequenza di chiamate a system Call per copiare i contenuti di un file a un altro file, è interessante perché si potrebbero tentare di tracciare le varie chiamate la System Call, per far vedere che per passare il contenuto di un file in un altro, per fare una filecopy, ci sono tutta una serie di chiamate di sistema che possono gestire, sia l'acquisizione del nome del file da tastiera, quindi ci sono:

- System call che gestiscono la I/O
- System call che gestiscono l'I/O per acquisire il nome del file di uscita
- System Call per fare una Open del file sorgente.
- Poi una system call che crea il file di uscita
- Delle system call che gestiscono le singole read le singole write
- Delle system call per fare la Close del file letto e la Close del file scritto

Quindi probabilmente non bastano una decina di System Call, senza contare che poi ci sono quelle ripetute iterativamente un po' di volte.

Esempio di system call: la funzione read

Un esempio di System Call è quella della funzione Read, che in un contesto Unix/Linux serve ad effettuare una lettura da file, non parliamo di lettura del testo in C con la scanf o fscanf, ma di una lettura di un dato binario da file.



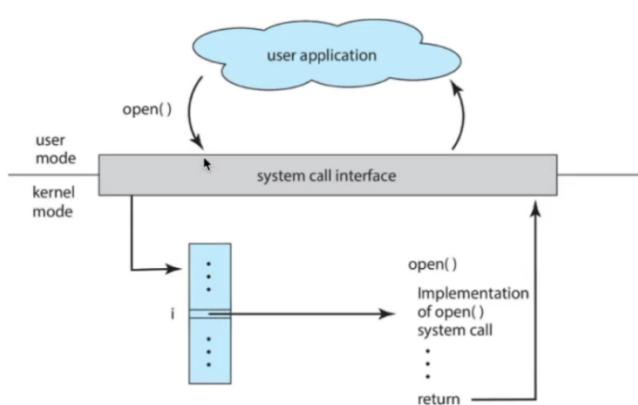
La funzione Read, di cui potete vedere la documentazione con man read, è una funzione che dal punto di vista del programma chiamante ha un prototipo che ritorna un valore intero di tipo ssize_t e riceve 3 parametri che sono: un intero che è il file descriptor, un puntatore Void * all'area di memoria nella quale bisogna andare a collocare il dato letto e la size_t count, che dice quanti byte bisogna leggere.

Un programma che voglia usare la read deve includere il prototipo, o meglio includere il file.h che contiene il prototipo e dovrà aspettarsi di passare i parametri corretti di ricevere come ritorno un dato intero senza segno che dice quanti byte sono stati effettivamente letti. Ma questa è una cosiddetta API quindi una abstract program interface, un'interfaccia verso una funzione che è visibile al limite dal programma utente.

Come viene implementata una system call

La System Call rimanda, ridirige il lavoro verso un'attività del kernel che si basa sull'avere assegnato un numero al lavoro che viene fatto, cioè viene associato ogni System Call un numero diverso dalle altre System Call e le interfacce verso le System Call mantiene una tabella indicizzata a seconda di questi numeri. All'interno la System Call chiama un componente del sistema operativo (del kernel) che esegue un lavoro dedicato alle System call.

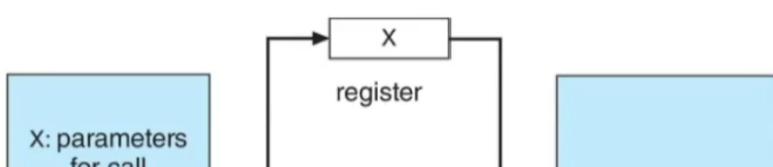
Normalmente il programma chiamante non deve sapere nulla di Come viene fatta la System Call al di dentro, chiama solo un'interfaccia che di fatto assegna un numero o riconosce il numero dalla system call e dice al kernel, per favore esegui il servizio legato a questa System call:



Cioè l'applicazione utente ad esempio chiama la Open, che è una System Call a livello utente, l'user application non vede assolutamente la Open interna che non viene neanche linkata con l'applicazione ma quello che viene eseguito è un passaggio da user Mode a kernel Mode attivando di fatto una Trap che qualcuno la chiama interrupt software ma chiamando una specie di interruzione che eseguirà un lavoro prendendo le informazioni da una tabella.

Il programma applicativo non è linkato con la Open ma dice, per favore sganciamo una trap al sistema operativo e diciamo al kernel di eseguire il numero dopo che io ho messo un numerino in una tabella, ho passato un numero a questa Trap che sceglierà il lavoro da fare una tabella.

Quindi normalmente serve che alla system call vengono passati dei parametri, non solo un numero, e avete visto che la Read aveva 3 parametri, la Open ne potrebbe avere i suoi cioè Quanti sono i parametri? dipende, allora il modo più semplice i modi in modo più semplice che ci potrebbe essere per passare i parametri al sistema operativo al kernel che seguono system call è quello di mettere i dati nei registri, ma non è detto che i parametri stiano tutti, i parametri potrebbero a questo punto stare in un blocco, una tabella in memoria e questo è il meccanismo scelto da Linux e Solaris, ad esempio quello che abbiamo visto essere un trap frame cioè un struttura in cui si passano dei dati, possono essere messi nello stack i parametri e poi il riferimento a questi dati nello stack è messo opportunamente nel trapframe.



Cioè user program che ha dei parametri per la chiamata, carica in pratica l'indirizzo di dove stanno i parametri in un registro decide di chiamare una System Call di un certo numero;

A questo punto Open avrà un numero e in definitiva il sistema operativo quando fa un servizio di una trap per una System Call dovrà localizzare identificare il numero della System Call in questo caso 13, usare un registro per andare a pescare dalla tabella i parametri e quindi eseguire il lavoro.

Tipi di System Call

Ci sono molte system call e possono servire a vari scopi e quindi li vediamo in modo molto rapido e ci sono System Call per controllare i processi crearli, terminarli, caricare, aspettare, debuggare e così via... ce ne sono per gestire i file e sono quelle forse un pochino più semplici da vedere, come aprire, creare e cancellare file leggere o scrivere e gestire dispositivi di I/O.

Poi acquisire o fornire semplicemente informazioni su data e ora, gestire comunicazioni gestire protezione delle informazioni, ovviamente queste è più ma abbiamo un'idea di del fatto che possono essere alcune decine.

Ci sono alcuni esempi di chiamate di sistema con il nome in Windows 32 e in Unix:

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS		
	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communications	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Ad esempio un processo in win32 viene creato con CreateProcess() e in Unix con Fork; La exit si chiama ExitProcess() in Windows, per aspettare la fine di un processo si fa wait() da una parte e WaitForSingleObject() dall'altra parte.

Vediamo che la Open, la Read, la Write e la Close si chiamano Createfile(), Readfile() Writefile() e CloseHandle(), avete sostanzialmente un elenco di nomi di chiamate di sistema.

Standard C Library Example

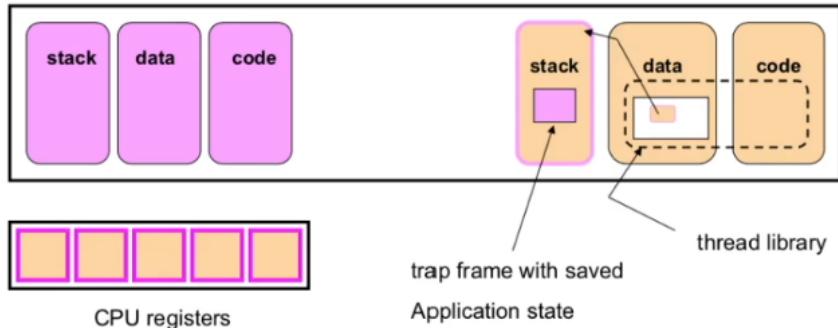
Quando un programma invoca una funzione tipo Printf, si tratta di una funzione di libreria, che dentro chiama la System Call Write(), quindi avete un paio di livelli, la Printf non è direttamente una System Call. La Printf gestisce a livello user il formato, il passaggio da stringhe a informazioni binarie che vanno spedite poi a video/standard output.

Mips trap: Handling System Calls, Exceptions and Interrupts.

In OS161 c'è un unico gestore delle eccezioni, un exception handler, che viene richiamato sia per le system call sia per le eccezioni vere e proprie che per l'interruzione, a secondo della causa ci sarà dal punto di vista hardware, un codice che indica la ragione se è una System call, un'eccezione o un'interruzione. La prima cosa che farà exception handler è quella di riconoscere qual era la causa. In OS161 ad ogni funzione di gestione dell'eccezione corrisponderà un diverso handler quindi ci sarà una funzione che gestisce le system call, una che si occupa delle eccezioni e una gli Interrupt.

OS161 Trap Frame

Alla base della gestione di un Trap, che serve sia per le eccezioni, che per entrare in user_mode e anche per la gestione delle System Call c'è il cosiddetto Trapframe:



Il Trapframe è una struttura che serve per salvare, il contesto cioè i registri del Task in esecuzione quando si attiva una trap; In altri termini quando c'è un programma utente in esecuzione e arriva una trap, viene salvato lo stato della CPU, i registri, in uno Stack, ce ne sono due di stack, per il processo utente1 dal lato user, e un'altro del lato Kernel, quest'ultimo si salva lo stato, cioè i registri nello Stack (lato kernel) a questo punto i registri sono disponibili e possono essere utilizzati per il servizio della trap.

Una Trap di fatto non cambia processo, ma passa dal processo lato user a processo lato kernel, quindi si salva semplicemente lo stato dei registri che sono in uso dal processo in lato user e si passa ad eseguire il kernel nel contesto però del processo diciamo attivo.

Os161 MIPS System Call Handler

La funzione che viene alla fin fine attivata quando il gestore di eccezioni si accorge che si tratta di una System Call:

```
void
syscall(struct trapframe *tf) {
    ..
    callno = tf->tf_v0; retval = 0;
    switch (callno) {
        case SYS_reboot:
            err = sys_reboot(tf->tf_a0); /* in kern/main/main.c */
            break;

        /* Add stuff here */
        default:
            kprintf("Unknown syscall %d\n", callno);
            err = ENOSYS;
            break;
    }
}
```

Si chiama syscall che riceve come unico parametro il puntatore al trapframe all'interno questa è solo una visualizzazione parziale, all'interno la funzione, in sostanza è un costrutto switch case nel quale per ora ci sono pochissime parti, cioè poi ne vedremo un'altra. Si tratta di guardare uno dei contenuti del trapframe che il campo tf_v0 che è di fatto il call number, cioè il numero della System Call, cioè il gestore di eccezioni riconosce l'eccezione syscall la syscall riceve poi un ulteriore argomento, dentro il trapframe che dice quale è la syscall.

il numero syscall a livello di costanti come eccezioni, di numero di eccezioni è la numero 8:

```

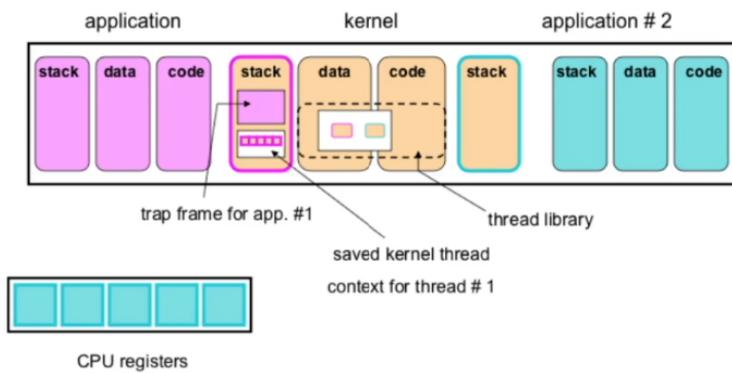
EX_IRQ      0    /* Interrupt */
EX_MOD      1    /* TLB Modify (write to read-only page) */
EX_TLBL     2    /* TLB miss on load */
EX_TLBS     3    /* TLB miss on store */
EX_ADEL     4    /* Address error on load */
EX_ADES     5    /* Address error on store */
EX_IBE      6    /* Bus error on instruction fetch */
EX_DBE      7    /* Bus error on data load *or* store */
EX_SYS    8    /* Syscall */
EX_BP      9    /* Breakpoint */
EX_RI     10    /* Reserved (illegal) instruction */
EX_CPU     11    /* Coprocessor unusable */
EX_OVF     12    /* Arithmetic overflow */

```

Mentre gli altri numeri sono Interrupt, piuttosto che non trap per la tlb. Questo non è altro che un particolare codice di eccezione, che viene interpretato come si syscall.

Two Process in OS161

Vediamo come possono essere visti due processi contemporaneamente attivi all'interno di os161:



Per visualizzare i due processi, si è disegnato, il kernel in mezzo e le due applicazioni ai lati vedete sempre dati e codice del kernel, il kernel ha due thread attivi, che sono diventati due processi. Ognuno di questi due processi ha uno Stack di kernel e ha Stack, dati e codice a livello user; Una delle due applicazioni è quella attiva nella CPU, l'application #2 mentre il L'application #1 ha il trap frame nello stack del kernel.

System Calls for process Management

OS161 implementa un sottoinsieme di system call di Linux:

	linux	OS161
Creation	fork,execve	* fork,execv
Destruction	_exit,kill	_exit
Synchronization	wait,waitpid,pause,...	waitpid
Attribute Mgmt	getpid,getuid,nice,getusage,...	getpid

Per esempio, a livello di creazione dei processi ci sono sia le fork che è la execv, che è una variante della execve in Linux; poi os161 prevede la exit, non prevede la Kill, c'è una Waitpid che è una delle primitive di sincronizzazione tra processi prevista in Linux e c'è la getpid.

OS161 Memory Management

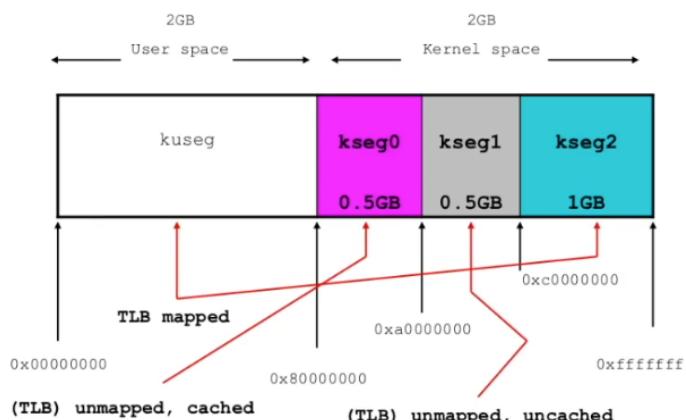
Vedremo come funziona la gestione della memoria sul processore MIPS, vedremo la tlb e vedremo la gestione della DUMBVM e di come viene creato il address Space di un processo per poter caricare un file eseguibile.

Per il kernel ci sono certe regole, per i processi ce ne sono altre, cioè ci sarebbero due possibilità a livello di memoria virtuale o fare in modo che il kernel non faccia traduzione logico fisica, ma usi solo indirizzi fisici, che vorrebbe dire quando un istruzione è privilegiata non usare la MMU e usare direttamente gli indirizzi che sono presenti nelle istruzioni come indirizzi fisici; l'altra possibilità è quella di abilitare la traduzione logica fisica, sia per il kernel che per i processi user, ma sostanzialmente far sì che lo spazio di indirizzamento del kernel e lo spazio indirizzamento dello user logici, siano diversi e che però implicherebbe quando si passa da user e kernel e viceversa cambiare ad esempio tabelle delle pagine, se fossimo in gestione paginata. Diciamo che os161 come Linux e altri sistemi operativi, in realtà non usano né l'una né l'altra ma in qualche modo una via di mezzo, nel senso che non usano un doppio spazio di indirizzamento quindi non usano la seconda ma non usano neanche la traduzione logico fisica disabilitata nel kernel cioè abilitano la traduzione logico fisiche sia per processi user che per processi kernel ma usano una Address Space unificato. Cioè il kernel è mappato in una porzione degli spazi di indirizzamento virtuale di ogni processo. C'è una protezione attiva per far sì che quando si lavora in modo non privilegiato si possa vedere utilizzare solo una parte dello spazio indirizzamento.

Il vantaggio di questo approccio non è tanto nel far sì che il processo utente vede il kernel ma è per far sì che il kernel veda il processo utente in modo più facile.

Address Translation on the MIPS R3000

Vediamo come viene realizzata questa mappatura di spazio di indirizzamento comune a processo utente e a kernel:



cosiddetto Kuseg, cioè il segmento di user.

La traduzione di questi indirizzi usa la TLB, i restanti 2 Gigabyte virtuali che quindi comprendono 0x80000000 a 0xFFFFFFF sono riservati al kernel e in particolare sono suddivisi a loro volta in tre parti: una parte da mezzo gigabyte, una seconda da mezzo gigabyte è una terza da 1 gigabyte.

- Il primo mezzo gigabyte non è mappato in TLB ciò vuol dire che non si prevede di fare paginazione su questo, cioè non si prevede la traduzione logico fisica mediante una tabella dovrà esserci un altro meccanismo e utilizza la cache.
- Il secondo mezzo gigabyte non è mappato in TLB ma non è neppure mappata nella cache.

Ricordiamo che il MIPS usato in questo contesto è un MIPS a 32-bit, il che vuol dire che ci sono complessivamente indirizzi che possono arrivare fino a 4 gigabyte, di questi due sono dedicati allo user space, e due al kernel space quindi si fa un partizionamento a priori metà al kernel e metà alla parte user.

Gli indirizzi bassi da 0 fino a 0x80000000 e vi ricordo che in 32 bit sono presentati da 8 cifre esadecimale, riguardano la parte utente il

- la terza parte da 1 gigabyte è mappato in TLB.

Allora il significato è sostanzialmente questo: il primo mezzo Gigabyte non è mappato in TLB Per far sì che ci sia di fatto una traduzione veloce da indirizzi logici e fisici che si baserà di fatto solo su una re location su una somma, è cached, cioè usa la cache perché qui ci mettiamo della Ram per il kernel. Il secondo mezzo Gigabyte non è mappato in TLB, di nuovo per avere una traduzione veloce che non faccia paginazione e non è cached perché qui tendenzialmente ci andranno a finire dei dispositivi di I/O Memory Map, per i quali non ha senso metterci davanti una memoria cache. Il terzo segmento è gestito in modo un po' diverso perché sarebbe un segmento di kernel paginato e quindi che usa memoria diciamo con paginazione verosimilmente perché il MIPS prevede una eventuale espansione di strutture dati per il kernel. In OS161 i processi utente sono in kseg quindi dei primi 2gigabyte il codice e i dati del kernel sono in kseg0, mentre i dispositivi di I/O sono in kseg1.

The MIPS R3000 TLB

La TLB del processore MIPS R3000 è una tlb Non completamente gestita via hardware ma in parte gestita via software, può contenere fino a 64 elementi:

- ogni riga contiene: un numero di pagina, un numero di frame fisico, un identificatore di address Space che potrebbe essere ad esempio l'identificatore del processo di cui quella pagina fa parte, è un po' di flag, flag di validità e altro.

Os161 fornisce funzioni di basso livello per gestire via software la TLB, con la possibilità di:

- Scrivere una riga la tlb: **tlb_write()**
- Modificare una di un entry casuale nella tlb: **tlb_random()**
- Leggere una data riga della tlb: **tlb_read()**
- Cercare una pagina un numero di pagina nella tlb: **tlb_probe()**

Quindi sono delle funzioni per chiedere dei risultati alla TLB. Se dal punto di vista hardware la Memory Management unit non è in grado di fare la traduzione cioè di trovare la corrispondenza di pagina da cui poi bisogna trovare il frame allora scatenerà un'eccezione che sarà gestita direttamente da una Trap apposita da OS161.

OS161 Address Space: dumbvm

Il gestore di traduzione nella versione attuale si chiama dumbvm, esso gestisce in modo automatico la traduzione logico fisica per il kernel e logico fisica per i processi utente usando la TLB:

Lo spazio di indirizzamento virtuale viene descritto per un dato processo da una cosiddetta struct address Space, contiene la descrizione delle tre aree del processo utente: il segmento di dati, segmento di codice e uno stack;

```

struct addrspace {
#if OPT_DUMBVM
    vaddr_t as_vbase1; /* base virtual address of code segment */
    paddr_t as_pbase1; /* base physical address of code segment */
    size_t as_npages1; /* size (in pages) of code segment */
    vaddr_t as_vbase2; /* base virtual address of data segment */
    paddr_t as_pbase2; /* base physical address of data segment */
    size_t as_npages2; /* size (in pages) of data segment */
    paddr_t as_stackpbase; /* base physical address of stack */
#else
    /* Put stuff here for your VM system */
#endif
};

```

Usa per ognuno di questi segmenti, i tipi vaddr_t che stanno per Virtual address e padd_t sta per physical address. Allora si tratta quindi di un indirizzo virtuale e di un indirizzo fisico quindi c'è per il segmento di codice, l'indirizzo virtuale di base, l'indirizzo fisico di base e la dimensione il numero di pagine. Si tratta di un segmento con allocazione contigua ma per multipli di una pagina. La dimensione di una pagina è data, come 4KB.

Per il primo segmento indirizzo logico virtuale di partenza, indirizzo fisico di partenza devono essere allineati e terminare con un numero sufficiente di zeri e poi il numero di pagine. Stessa cosa per il segmento due, segmento di dati, indirizzo virtuale di base indirizzo fisico di base e numero di pagine. Poi c'è il puntatore allo stack che non ha bisogno della dimensione dello Stack, semplicemente perché in dumbvm la dimensione dello stack è fissa ed è determinata da una costante, quindi non va in magazzinato. Tale struttura è stata definita con uno switch di compilazione if OPT_DUMBVM è vera, allora la struttura è definita così, altrimenti andrebbe definito in un altro modo.

Ad esempio, se si vuole utilizzare un meccanismo di paginazione dovremmo mettere, il puntatore della page table e la dimensione della page table e ho altre informazioni.

[Address Translation Under dumbvm](#)

La traduzione logico fisica viene fatta nella MMU, via hardware tentando di usare la TLB; Si prende l'indirizzo su 32 bit, si isola il pezzo che indica la pagina, si fa una ricerca nella TLB, se funziona si arriva direttamente a generare un indirizzo fisico. Se non viene trovato il numero di pagine nella TLB viene generato un page fault o un'eccezione di indirizzo che genera di fatto una trap; Questo Page Fault viene gestito da una funzione che si chiama vmfault e il mips/vm/dumbvm genera un'eccezione per il kernel e nella gestione di questa eccezione si userà la traduzione da logica fisica che anziché usare una tabella delle pagine userà direttamente L'address Space per riuscire a cercare, se l'indirizzo logico a cui si fa riferimento il numero di pagina, appartiene al segmento 1, se appartiene al segmento 2 o se appartiene allo stack. Una volta determinato a quale dei tre appartiene, si tenta di fare una traduzione logico fisica mediante un'applicazione di una relocation, quindi di un passaggio da logico a fisico in uno spazio di indirizzamento contiguo. Sono tre gli intervalli contigui segmento 1, segmento 2 e segmento 3.

[Loading a Program into an Address Space](#)

Dato un processo per il quale sia stato creato uno spazio di indirizzamento, Il passo successivo è cominciare a caricare in questo spazio di indirizzamento, load un eseguibile in nel formato Elf, executables linking format.

L'ELF è un formato di file eseguibile, per il quale si ha una corrispondenza tra la chiamata alla creazione di un processo, normalmente nella System Call execv, che è la System Call che crea un processo da associare un eseguibile. In realtà anche la runprogram fa la stessa cosa, però è un eseguibile lanciato da un menu di kernel anziché un eseguibile lanciato da un processo utente. La execv è quella che permette a un processo utente di eseguire un altro processo utente. C'è un'altra possibilità per creare un processo utente, che è la Fork ma la fork, non carica un nuovo eseguibile, semplicemente duplica che sta già eseguendo un eseguibile.

ELF Files

Il file Elf è un file che all'interno contiene: una intestazione e quantomeno due sezioni, una sezione di codice, una sezione di dati, che andranno rispettivamente caricati nello spazio di indirizzamento virtuale del processo, più altre informazioni. All'inizio il file contiene una descrizione dei segmenti, un Header che descrive questi due segmenti e poi contiene i segmenti stessi.

Quindi ogni sezione delle due contenute nel Elf conterrà, la descrizione di una regione della Address Space di un processo. Dal punto di vista pratico la load Elf fa un'apertura del file e caricamento dei due segmenti che vengono presi da file. Per ogni segmento nel file Elf c'è l'immagine del segmento e un Header che descrive dove va a finire questo segmento nello spazio indirizzamento virtuale, cioè c'è scritto di quel segmento: deve andare all'indirizzo virtuale A00000, la lunghezza del segmento, quanto deve essere grande, dove sta quel segmento nel file Elf e quanto è grande l'immagine del segmento nel file Elf, sembra strano che sia duplicata la lunghezza, in realtà sarebbe possibile avere un segmento che occupa che 10 megabyte nel file elf e ne devo occupare 16 megabyte in memoria virtuale, perché ad esempio una parte è caricata con tutti gli zeri o non è significativa, quindi non è detto che il contenuto del file Elf, corrisponde a quello che deve stare in memoria.

ELF Files e OS161

L'implementazione da dumbvm prevede nel file elf solo due segmenti: un segmento detto text che è il segmento di codice, un segmento detto data che contiene diciamo così le variabili costanti globali, non è descritto lo stack che viene creato di dimensione fissa 12 pagine e inizia andando all'indietro un byte prima del 0x80000000 che è l'indirizzo logico di inizio dello spazio del kernel.

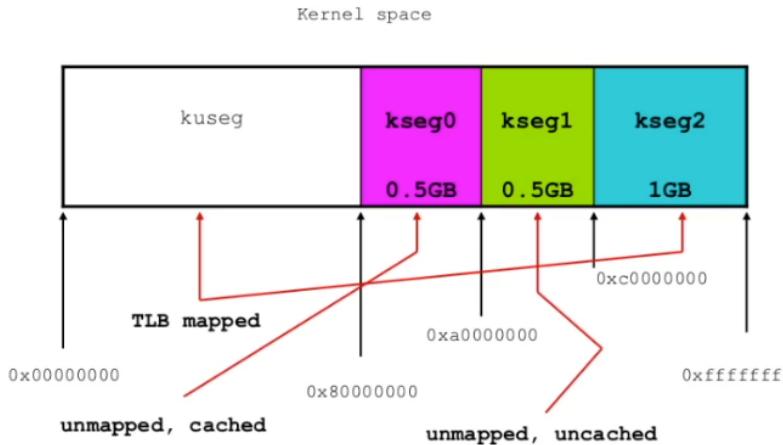
4. OS161 – Address Space & Memory Management

La Gestione della memoria per lo spazio user e per il kernel in Os161. Dumbvm è il gestore di memoria, si basa su allocazione contigua, in cui per un processo utente si allocherà dei multipli di pagina.

Le funzioni che vedremo sono getppages, una funzione che ottiene un intervallo contiguo di pagine e ram_stealmem, una funzione che riserva dei frame nella RAM contigui. Faremo un locatore che è in grado di servire sia lo spazio utente, che il kernel.

MIPS Virtual Address Space

Lo spazio di indirizzamento nell'architettura MIPS e quindi in OS161 è comune al kernel e allo user. cioè quando si è nel lato kernel si vede nello stesso spazio di indirizzamento la memoria user. Non è vero il viceversa perché un processo user non è autorizzato, neanche se potendo usare gli indirizzi non è autorizzato ad usare quelli di kernel.



Dei 4 Gigabyte disponibili per un'architettura a 32 bit, i primi due logici, quindi da 0 a 0 x80000000 sono riservati a indirizzi user, di lì in avanti da 0x80000000 in su c'è il kernel e il segmento due non viene usato in un OS161. Nel segmento Zero (Kseg0) c'è il kernel mentre in (Kseg1) ci sono i dispositivi di I/O Memory Maps.

Il Kernel parte da un indirizzo: 1 seguiti seguito da 31zeri, ora lo spazio di indirizzamento dell'user mappato in TLB, subirà una traduzione appunto basata su TLB, quindi su una tabella di mappatura di pagine anche se allocate contiguamente, quindi allocheremo contiguo, ma ci sarà una traduzione per pagine. Mentre il kernel agli indirizzi logici sarà sottratto 0x80000000 per riportare il segmento zero, da indirizzi logici che partono con 1 seguito da altri 31 bit, a indirizzi logici che partono da 1 seguiti gli stessi bit quindi si ha una traslazione di 2 gigabyte in basso, dal passaggio da logico a fisico per il kernel.

Kernel Loader (sys161: start.s)

Supponiamo di avere un kernel che viene piazzato in una memoria da 1MB

Logical addr. (KSEG0)	Physical addr.
0x80000000	0x0
0x80000200	0x200
0x80039d54 (_end)	0x39d54
0x8003a000 (P)	0x3a000
0x8003b000 (P+1000)	0x3b000
0x80100000	0x100000

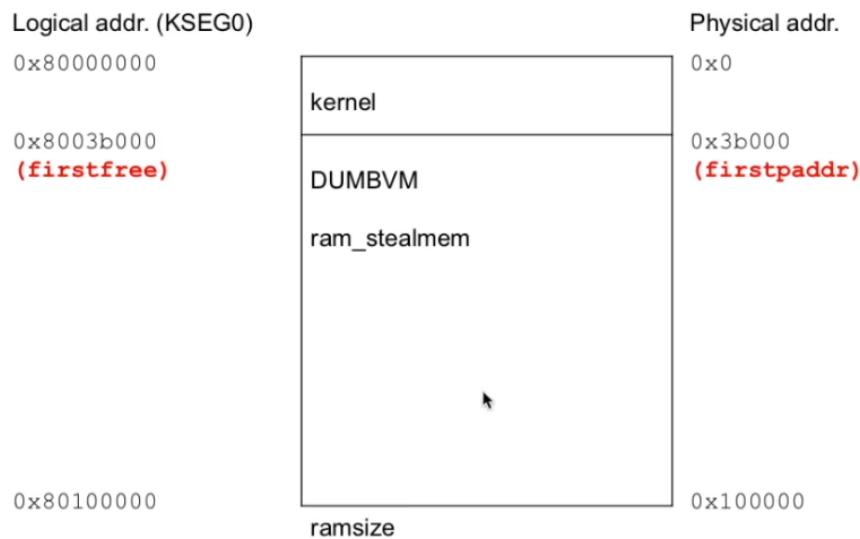
exception handlers
kernel
arg string for boot +
Page align
Stack for first thread
(1 page = 4096 B)
FREE MEMORY
ramsize (es. 1MB: sys161.conf)

In 1MB di ram, quando si carica il kernel si vanno a riempire gli indirizzi fisici che vanno da 0x0 all'indirizzo 0x100000. Gli indirizzi logici che corrispondono a questi indirizzi fisici sono sempre gli stessi cambia semplicemente che si aggiunge 0x80000000 quindi c'è una corrispondenza molto semplice per gli indirizzi (Kseg0).

Quando si fa Bootstrap del Kernel, in ram vanno a finire nei primi 0x200 byte i gestori delle eccezioni, poi a seguire il kernel, seguono le stringhe di bootstrap e un po' di allineamento che porta ad allineare ad un

multiplo di pagina. Subito dopo troviamo lo stack per il primo thread, e poi la memoria libera. Questa configurazione potrebbe cambiare nel momento in cui andiamo a scrivere delle parti nuove nel kernel che andrebbero a aumentare la dimensione del kernel.

Notiamo che finito il bootstrap a partire dall'indirizzo fisico 0x3b000 (logico di kernel 0x80003b00) potremmo dire che la memoria è libera, a partire dall'indirizzo che dal alto fisico prende il nome di **firstpaddr** mentre da quello logico **firstfree**, rappresentiamola quindi in un altro modo:



Facendo notare che la memoria liberà è la parte più significativa della ram disponibile. Questa è la memoria che utilizza DUMBVM quando si chiede di allocare.

Ram_bootstrap

Dove riveliamo questo firstpaddr? La funzione ram_bootstrap è una delle funzioni richiamate al bootstrap, che chiede al mainbus (a MIPS) la dimensione della ram, poi fa un controllo, e poi assegna all'ultimo indirizzo disponibile (ramsize) il lastpaddr.

```

void
ram_bootstrap(void) {
    /* Get size of RAM. */
    size_t ramsize = mainbus_ramsiz();
    if (ramsize > 512*1024*1024) {
        ramsize = 512*1024*1024;
    }
    lastpaddr = ramsize;
    /* Get first free virtual address from where
       start.S saved it. Convert to physical address. */
    firstpaddr = firstfree - MIPS_KSEG0;
}

```

A questo punto si ottiene il primo indirizzo virtuale, come la conversione da logico a fisico di firstfree, firstfree comincia con 0x800. Sottraiamo MIPS_KSEG0 e passiamo da logico a fisico.

Quindi la traduzione da logico a fisico per il kernel sottraendo MIPS_KSEG0 mentre si passa da fisico a logico comando MIPS_KSEG0. La definizione di MIPS_KSEG0 è di fatto 2GB.

Ram_stealmem

È l'allocatore vero e proprio, scritto in (kern/arch/mips/vm/ram.c) questa funziona alla lettera ruba memoria alla ram, cioè l'idea è che vogliamo ottenere npages pagine, il gestore prende il primo indirizzo fisico libero e ruba n-pagine e fa andare avanti il firstpaddr.

```
paddr_t ram_stalmem(unsigned long npages) {
    paddr_t paddr;
    size_t size = npages * PAGE_SIZE;

    if (firstpaddr + size > lastpaddr) {
        return 0;
    }

    paddr = firstpaddr;
    firstpaddr += size;

    return paddr;
}
```

Se il firstpaddr + quello che si richiede (size), se ci stiamo nella Ram allora andiamo avanti se invece non ci stiamo altrimenti return 0, quindi l'indirizzo fisico ritornato è zero, se ram_stalmem ritorna 0 vuol dire che non sono riuscito ad allocare. Invece se la dimensione è compatibile con lo spazio residuo allora si assegna a paddr il firstpaddr e si sposta firstpaddr in avanti di uno spazio pari a size. L'unico vero problema è che non si mantiene traccia delle allocazioni cioè non viene mantenuta una lista o una tabella di partizioni allocate e quindi non si potrà deallocate questo è il vero motivo per cui si chiama dumbvm.

Getppages

È una funziona che fa da wrapper alla ram_stalmem è tuttavia è scritta in dumbvm.c non in ram nel senso che anche questa funziona ritorna un physical Address e non fa altro che chiamare ram_stalmem in mutua esclusione, con spinlock e lock primitive di mutua esclusione.

```

static paddr_t
getppages(unsigned long npages) {
    paddr_t addr;

    spinlock_acquire(&stealmem_lock);

    addr = ram_stalmem(npages);

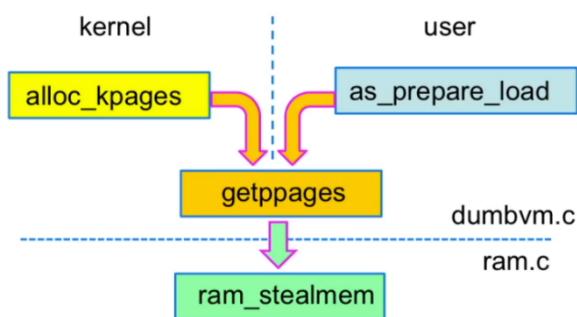
    spinlock_release(&stealmem_lock);

    return addr;
}

```

Architettura del software: dumbvm.c (alloc)

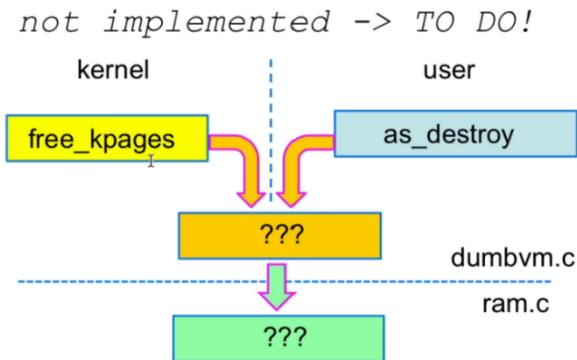
In ram.c esiste unicamente la funzione ram_stalmem che viene chiamata da getppages che è l'interfaccia di basso livello verso ram_stalmem e questa funzione viene usata a più alto livello sia dal kernel che dai processi user.



La funzione `as_prepare_load` crea lo spazio di indirizzamento di un processo, creando delle partizioni logiche contigue, mappate a partizioni fisiche contigue chiamati segmenti; Due segmenti indirizzi logici contigui e uno stack, ma anche la `alloc_kpages`, l'allocatore dinamico del kernel che ritaglia e genera strutture dati.

Nella versione attuale non modificata di OS161, entrambi questi allocator per i processi utenti e per il kernel chiamano la `getppages` per ottenere di fatto un intervallo di pagine fisiche (sono frame ma vengono chiamate pagine) di ram.

Quello che manca è l'implementazione per il supporto a `free_kpages` (free)



E per la `as_destroy` cioè per distruggere un processo utente e ritornare la memoria che era stata precedentemente allocata, vediamo allora come possiamo completare il gestore di memoria.

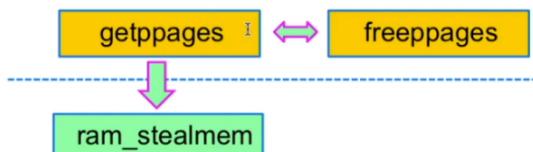
De-alloc (free) in ram.c – Solution A1

La prima soluzione per supportare la de-alloc in memoria, è quella di fare delle funzioni parallele a quelle esistenti, esistono `ram_stalmem` e `getppages`, realizziamo `ram_freetmem` e `freepages`, potremmo fare una `freepages` che è solo un interfaccia verso `ram_freetmem`, dovremmo fare così una struttura dati che sia una free-list una bitmap, per rappresentare e poter ritornare delle partizioni alla ram in memory management cioè in `ram.c`.

Questa è un'ipotesi che in linea di massima potrebbe andare bene però tenete conto che questa `ram.c` per il MIPS potrebbe cambiare qualcosa nel momento in cui volessimo cambiare il microprocessore, in sostanza la scelta è implementiamo l'allocatore e il deallocator nel gestore di memoria oppure passiamo alla soluzione A2

Solution A2

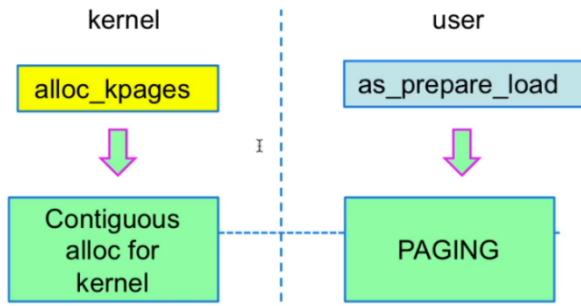
Facciamo una allocatore e deallocator a livello di `dumbvbm.c` sostanzialmente non tocando il gestore della ram, chiediamo alla ram memoria e non la restituiamo alla ram ma la gestiamo a livello più alto.



Ci sono due ipotesi: la prima in cui chiediamo alla ram tutto lo spazio disponibile e di fatto lo allochiamo e giochiamo con questo spazio per allocare e far restituire spazio, l'altra possibilità è chiediamo alla ram memoria un po' alla volta quando non ci basta quella che avevamo ottenuto prima ma a livello di `getppages` e `freepages` cerchiamo di palleggiare, cioè di riesumare memoria precedentemente allocata poi allocarla e poi liberarla nuovamente.

Paging in user space (solution B)

C'è un'altra possibilità cioè non usare una `dumbvbm` cioè fare un allocatore basato su paginazione fatto in modo serio, abbiamo paginazione per il processo user allocazione contigua per il kernel:



Proposed Solution for de-alloc in dumbvm (sol. A2)

- Si alloca per multipli di pagine, e si fa un allocatore comune sia al kernel che all'user
- È fatto in dumbvm, si usa per tenere traccia delle pagine allocate (frame allocati) una bitmap, una mappa di bit, che significa un vettore di bit che è parallelo alle pagine (i frame) in ram e servirà a dire allocato o non allocato.
 - Quando bisogna allocare si fa un tentativo di trovare frame disponibili, tra quelli precedentemente allocati, se non ce ne sono si chiama la `ram_stealmem` che chiede memoria alla ram.
 - Come alternativa si poteva fare una richiesta di tutta la ram all'inizio per poi giocare direttamente in questo modo.
- La bitmap è implementata qui non come una vera bitmap, in motivo vero è che in C un vettore di bit non esiste, il vettore di bit viene gestito come vettore di byte in cui poi a livello di singola word o singolo byte, si devono andare impacchettare i singolo bit. Il motivo è che le memoria non indirizzano il bit.
 - Per semplicità un vettore di 0 e 1 viene implementato come vettore di char
 - `freeRamFrames[i] = 1/0 (free/allocated): free=FREED! (by freeppages)`
 - In questo contesto 1 significa libero o liberato, mentre 0 occupato. Questa è semplicemente una convenzione.
- Oltre all'allocatore, serve il deallocator che restituisce una pagina precedentemente allocata. Ed è chiaro che è caratterizzata da:
 - Un puntatore all'inizio della partizione
 - La dimensione della partizione
- Quindi per poter gestire il puntatore e la dimensione, serve una tabella in cui l'allocatore quando restituisce il puntatore ad una partizione allocata si ricorda che a quel puntatore era stata associata una certa dimensione (in realtà non è necessario sempre, almeno per la parte user non sarebbe necessaria questa tabella)
- Le due funzioni che chiameranno il deallocator sono:
 - `void free_kpages(vaddr_t addr): table needed as only pointer passed`
 - `void as_destroy(struct addrspace *as): table not needed ad size is stored in address space`

`As_destroy`: distrugge uno spazio di indirizzamento utente, che restituisce due partizioni rappresentate nella struct `addrspace`, tale struct di ogni segmento conosce l'inizio e la dimensione e dello stack si conosce solo l'inizio perché la dimensione è una costante. Quindi questa funzione ha tutte le informazioni che le servono per descrivere una partizione deallocated cioè dove inizia e quanto è grande e quindi non serve ricordare in una tabella un indirizzo a quale dimensione fosse associato. Tuttavia per l'allocatore di kernel `free_kpages`, compatibilmente con quello che sono le

free degli deallocatori, non ritorna due parametri ma solo uno ritorna l'indirizzo di ciò che si vuole liberare, la free non dice quanto è grande ciò che si vuole liberare, a questo punto l'unica cosa che può fare il deallocator e prendere è prendere questo indirizzo (addr) andarlo a cercare in un tabella in cui andiamo a ritrovare la dimensione. Nel nostro contesto questa tabella si chiama allocSize[i].

Global variables (and test function)

Alcune indicazioni su quali variabili globali utilizzare:

```
static struct spinlock freemem_lock = SPINLOCK_INITIALIZER;

static unsigned char *freeRamFrames = NULL;
static unsigned long *allocSize = NULL;
static int nRamFrames = 0;

static int allocTableActive = 0;

static int isTableActive () {
    int active;
    spinlock_acquire(&freemem_lock);
    active = allocTableActive;
    spinlock_release(&freemem_lock);
    return active;
}
```

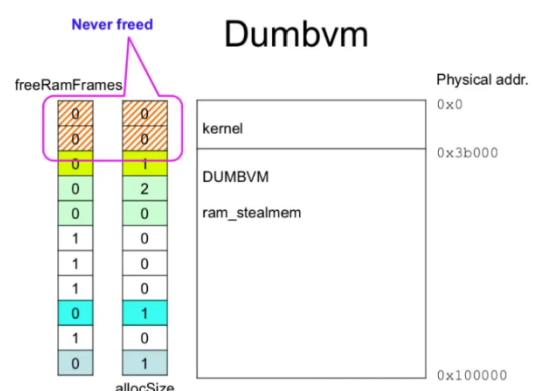
Uno spinlock che serva a gestire in mutua esclusione le operazioni di free e la tabella di allocazione. Poi un vettore che poi è una bitmap *freeRamFrames che serve ad associare un bit (in questo caso un char) a ogni pagina/frame allocabile. Un secondo vettore di interi unsigned long *allocSize, che servirà per ogni partizione allocata a ricordare quanto è grande questa partizione allocata; per comodità facciamo la allocSize parallela alla ram e quindi alle pagine.

Notate che ci potrebbe essere anche un altro allocatore che potreste realizzare voi che anziché usare vettori paralleli alle pagine, facesse una freelist, una tabella di partizioni occupate. nRamFrames indicherà la dimensione di questi vettori, partiamo da 0, qui è stata fatta la scelta di allocare dinamicamente questi vettori e settare dinamicamente i frame usando la dimensione vera della ram rilevata al bootstrap;

Si sarebbe potuto fare dei vettori con dimensioni costanti e quindi evitare i due puntatori tuttavia il problema sarebbe stato che potessero essere sovradimensionati e avremmo dovuto cambiare questa costante ogni volta che il kernel cambia sys161.conf e quindi non molto pratico. Poi facciamo una variabile allocTableActive che servirà a differenziare due fasi: una prima fase in cui l'allocatore di pagine non è ancora attivo, e una seconda in cui è attivo. Per fare questo ho fornito qui una funzione di testing che guarda in mutua esclusione la variabile allocTableActive, che ritorna 0 o 1 a seconda che l'allocatore sia attivo oppure no.

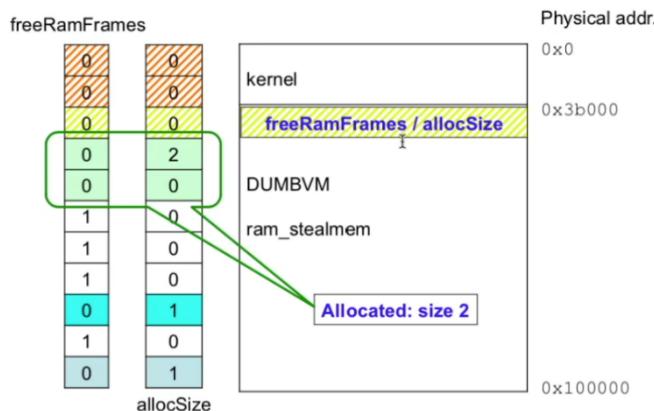
Dumbvm: in azione

Vediamo qual è la funzione dei due vettori: freeRamFrames e allocSize con un esempio:



Questi due vettori sono dimensionati con un minimo di spreco perché mappano anche una parte della ram che non è disponibile per allocazione e deallocazione; la parte in cui a indirizzi fisici di kernel non si può associare nulla in termini di locazione, le prime pagine (quella zona) non verrà mai liberata perché si tratta di una parte di kernel, a rigore ci sarebbe anche la parte freeRamFrames e allocSize, questi due vettori a sua volta vengono allocati per fornire un servizio di allocazione, la ram in cui stanno loro non sarà più utilizzabile.

In questa situazione intermedia ho cercato di rappresentare una situazione intermedia ad allocatore già avviato:



In cui ad esempio qui è stato allocata una partizione di due pagine/frame, guardando la freeRamFrames lo 0 dice partizione occupata/non libera, nella prima casella del vettore allocSize c'è scritto 2 per indicare che qui comincia un intervallo di 2 pagine; successivamente c'è un intervallo di 3 pagine libere. In sostanza l'allocatore usa questi due vettori indicando con 0 una locazione occupata e con 1 una libera.

Free_kpages & as_destroy

Vediamo queste due funzioni in OS161:

```

void free_kpages(vaddr_t addr) {
    if (isTableActive()) {
        paddr_t paddr = addr - MIPS_KSEG0;
        long first = paddr/PAGE_SIZE;
        KASSERT(nRamFrames>first);
        freeppages(paddr, allocSize[first]);
    }
}
void as_destroy(struct addrspace *as) {
    dumbvm_can_sleep();
    freeppages(as->as_pbase1, as->as_npages1);
    freeppages(as->as_pbase2, as->as_npages2);
    freeppages(as->as_stackpbase, DUMBVM_STACKPAGES);
    kfree(as);
}

```

La free_kpages esiste ma è vuota all'interno, qui sopra vediamo l'implementazione proposta, questa funzione è una funzione di ballo livello che viene chiamata dalla kfree. Fin quando la tabella non è attiva, un

eventuale chiamata a kfree e quindi a free_kpages non ha effetto, non restituisce niente, altrimenti quando si libera qualcosa, si restituisce un puntatore logico, vediamo che un indirizzo logico vaddr_t contrapposto a paddr_t sono rispettivamente un indirizzo nello spazio logico virtuale e un indirizzo fisico; Quindi a partire dall'indirizzo logico che va restituito convertiamo questo indirizzo da logico a fisico sottraendo MIPS_KSEG0, una volta diventato indirizzo fisico lo riconduciamo ad un numero di pagine long first.

Poi abbiamo una KASSERT, ne vedremo tanto di asserzioni che indica in questo caso che se la condizione fallisce piantati perché c'è qualcosa da sistemare, non sono dei test per aggiustare un programma e rimetterlo apposto servono semplicemente per dire che se questa condizione è falsa, c'è qualcosa che devi rivedere.

Se il primo indirizzo è più grande della Ram fisica c'è davvero qualcosa che non va. A questo punto chiamiamo la freeppages la funzione che vogliamo realizzare che abbiamo indicato prima, con l'obiettivo di restituire delle pagine alla Ram, restituiamo paddr e la dimensione che prendiamo dalla AllocSize.

Per quanto riguardo as_destroy la distruzione della spazio di indirizzamento, riceve l'address space e andando a vedere la struct address space, vediamo che possiamo restituire il primo segmento, di cui c'è sia il puntatore ovvero l'indirizzo fisico di partenza che la dimensione, restituiamo lo stackuser, la cui dimensione è DUMBVM_STACKPAGES una costante, quindi qui non c'è bisogno di allocSize, a rigore si potrebbe verificare con un KASSERT che le dimensione corrispondono a quelle di AllocSize. Fatto questo fa la kfree dell'address space. Queste due funzioni sono quelle che chiamano la liberazione, non vediamo quelle che chiamano l'allocazione perché tanto quelle sono già complete.

Inizialization: vm_bootstrap

Vediamo il cosiddetto bootstrap, in cui vogliamo andare a piazzare l'inizializzazione delle tabelle per gestire l'allocatore, questa è una funzione che viene chiamata al bootstrap, dopo il ram_bootstrap:

```
void vm_bootstrap(void) {
    int i;
    nRamFrames = ((int)ram_getsize()) / PAGE_SIZE;
    /* alloc freeRamFrame and allocSize */
    freeRamFrames = kmalloc(sizeof(unsigned char) * nRamFrames);
    allocSize = kmalloc(sizeof(unsigned long) * nRamFrames);
    if (freeRamFrames == NULL || allocSize == NULL) {
        /* reset to disable this vm management */
        freeRamFrames = allocSize = NULL; return;
    }
    for (i = 0; i < nRamFrames; i++) {
        freeRamFrames[i] = (unsigned char)0; allocSize[i] = 0;
    }
    spinlock_acquire(&freemem_lock);
    allocTableActive = 1;
    spinlock_release(&freemem_lock);
}
```

La prima cosa che può essere fatta è recuperare nella variabile nRamFrames il numero totale di frames. A questo punto allochiamo i due vettori freeRamFrames e allocSize. Qui poi si suppone che se almeno uno dei due è andato male, lasciamo perdere, e restiamo con l'allocatore che c'è possibile; Normalmente dovrebbe

funzionare; diciamo così se non funzionasse vorrebbe dire che probabilmente avete una ram troppo piccola, e dovremmo tornare in sys161.conf e ingrandire la dimensione della ram.

Fatto questo facciamo un'inizializzazione dei due vettori e mettiamo in tutti i bit (che poi sono char) di freeRam Frames 0 per dire che sono non liberato, cioè non ancora restituito e quindi non disponibile per essere riciclato in qualche modo e mettiamo allocSize a 0 semplicemente per inizializzarlo, probabilmente quest'ultima non serve più di tanto ma comunque lo facciamo per pulizia.

Poi mettiamo allocTableActive = 1 in modo protetto da spinlock, perché questa è la variabile che viene guardata in mutua esclusione per verificare se la tabella è già disponibile o no.

Getppages

Vediamo una versione modificata di questa funzione:

```
static paddr_t getppages(unsigned long npages) {
    paddr_t addr;

    /* try freed pages first */
    addr = getfreepages(npages);
    if (addr == 0) {/* call stealmem */
        spinlock_acquire(&stealmem_lock);
        addr = ram_stealmem(npages);
        spinlock_release(&stealmem_lock);
    }
    if (addr != 0 && isTableActive()) {
        spinlock_acquire(&freemem_lock);
        allocSize[addr/PAGE_SIZE] = npages;
        spinlock_release(&freemem_lock);
    }
    return addr;
}
```

Quando vogliamo npagine contigue (nframe) per prima cosa si tenta di chiamare la getfreepages, cioè ottenere delle pagine che non state precedentemente liberate, se l'indirizzo ritornato è zero cioè se non ce ne sono di queste pagine precedentemente liberate allora si ricorre a ram_stealmem, la ram_stealmem potrebbe anche essa rispondere 0 se non ci sono pagine disponibili, in caso contrario se la tabella è attiva allora ci segniamo in allocSize, l'indirizzo ritornato diviso PAGE_SIZE ci segniamo npages, cioè quante pagine abbiamo allocato sempre in mutua esclusione.

Getfreepages

La funzione a cui chiediamo se ci sono delle pagine contigue e libere:

```

static paddr_t getfreepages(unsigned long npages) {
    paddr_t addr;
    long i, first, found, np = (long)npages;

    if (!isTableActive()) return 0;
    spinlock_acquire(&freemem_lock);
    // Linear search of free interval
    for (i=0,first=found=-1; i<nRamFrames; i++) {
        if (freeRamFrames[i]) {
            if (i==0 || !freeRamFrames[i-1])
                first = i; /* set first free in an interval */
            if (i-first+1 >= np)
                found = first;
        }
    }
}

```

Andiamo a cercare le pagine che hanno 1 nel vettore bitmap, freeRamframes. Ci sono più modi per effettuare la ricerca di pagine libere, qui ne viene proposto uno che fa una ricerca in un tempo lineare e non quadratico.

La versione banale sarebbe: per ogni casella di freeRamFrames, se c'è 1 vai a cercare quanti 1 ci sono, quindi si vanno a cercare un intervallo di 1 contigui di dimensione almeno la dimensione data. Nel codice sopra vediamo un primo controllo che dice se la tabella non è attiva lasciamo satre e ritorniamo 0. Altrimenti in mutua esclusione con freemem_lock decide di lavorare da sola sulla tabella, e si fa una ricerca che ha come obiettivo ricercare un intervallo contiguo di uni in cui in ogni casella se c'è un uno, ci ricordiamo dove stava il primo 1 tra gli ultimi visti, e misuriamo se questo intervallo rilevato è sufficiente a soddisfare la richiesta npages. Ogni volta che troviamo uno 0 resettiamo la dimensione dell'intervallo di 1.

Mediane le variabili i, first, found e np facciamo questo: i parte da 0 quindi guardiamo tutti i bit nella freeRamFrames, first e found partono da -1 e andiamo avanti fino a nRamFrames, se troviamo un 1 e i==0 oppure nella casella precedente c'era uno zero, sappiamo di aver localizzato un primo intervallo di 1 contigui, poi diciamo che la distanza tra i e first +1 perché se l'inizio e la fine coincidono dobbiamo contare 1 è sufficiente >=np allora ci ricordiamo found = first; e a rigore ci andrebbe un break.

Alla fine, se found è maggiore uguale a 0 allora dobbiamo andare a modificare gli 1 nella bitmap e farli diventare zeri perché li abbiamo allocati:

```

if (found>=0) {
    for (i=found; i<found+np; i++) {
        freeRamFrames[i] = (unsigned char)0;
    }
    allocSize[found] = np;
    addr = (paddr_t) found*PAGE_SIZE;
}
else {
    addr = 0;
}

spinlock_release(&freemem_lock);

return addr;

```

Poi mettiamo allocSize di found uguale a np. Alla fine, ricordiamo di rilasciare questo lock prima di ritornare l'indirizzo valido.

Freepages

Riceve l'indirizzo fisico della partizione che deve restituire e il numero di pagine:

```
static int freepages(paddr_t addr, unsigned long npages) {
    long i, first, np=(long)npages;
    if (!isTableActive()) return 0;
    first = addr/PAGE_SIZE;
    KASSERT(allocSize!=NULL);
    KASSERT(nRamFrames>first);

    spinlock_acquire(&freemem_lock);
    for (i=first; i<first+np; i++) {
        freeRamFrames[i] = (unsigned char)1;
    }
    spinlock_release(&freemem_lock);

    return 1;
}
```

Npages dal lato kernel viene rilevato dal vettore allocSize se siamo dal lato user quando si libera un address space può essere rilevato direttamente dalle informazioni disponibili dell'address space. Al solito se la tabella non è attiva lasciamo stare, si calcola l'indice di inizio first da deallocare, facciamo un paio di asserzioni, ma quello che conta è che in mutua esclusione si va sul vettore freeRamFrames a mettere a 1 i frame restituiti. Non c'è bisogno di azzerare il vettore AllocSize perché viene visto solo nel caso in freeRamFrames sia a zero.

5. OS161 – Sincronizzazione di primitive

Affrontiamo in questa e nella prossima lezione aspetti di sincronizzazione e in particolare di realizzazione in OS161 di primitive di sincronizzazione.

Background

Questo capitolo è orientato a OS161 ma è al tempo stesso una rivisitazione di concetti di sincronizzazione in cui vogliamo fornire supporto di sincronizzazione per programmazione concorrente realizzandola dal di dentro e comunque analizzandone le problematiche.

In generale si tratta di esaminare cosa s'intenda e come si può gestire l'accesso coordinato a risorse condivise. Normalmente in programmazione concorrente bisognerebbe affrontare cosa succede di buono o meglio come si coordinano i lavori ma spesso e volentieri uno degli aspetti su cui ci si concentra maggiormente sono *i problemi di concorrenza*, in parte a torto perché occorrerebbe vederli solo a valle di aver fatto o esaminato tecniche di progetto.

I problemi i problemi di concorrenza possono essere classificati in problemi delle cosiddette *race conditions* dove una corsa critica è una situazione in cui il risultato di un'elaborazione concorrente può dipendere dall'ordine con cui sono state effettuate le variazioni. Attenzione! Stiamo parlando di un'esecuzione concorrente in cui il risultato dovrebbe essere indipendente dall'ordine che invece lo diviene.

Abbiamo poi i *deadlock* che sono i casi in cui ci sono più attori in circolo e ognuno aspetta qualcosa da qualcun altro e alla fine ognuno è in attesa e non si va avanti, mentre la *starvation* è il caso in cui i dati un certo numero di attori o di thread/processi in concorrenza ce n'è almeno uno che "fa la fame" nel senso che è in attesa, non riesce a conquistare la risorsa che vorrebbe e non riesce ad essere eseguito perché ci sono gli altri che occupano sempre la CPU, questo si chiama *resource starvation*.

Le soluzioni ai problemi di cui sopra sono un mix tra primitive di sincronizzazione e tecniche piuttosto che non raccomandazioni su come fare le cose bene.

Sezioni Critiche

Iniziamo dal definire la cosiddetta *sezione critica* o *critical section*, cioè la parte di un programma (istruzioni e dati) che deve essere eseguita in modo tale che non più di un attore (non più di un processo o thread) la stia eseguendo in parallelo agli altri. Per fare questo le soluzioni sono date da: *mutua esclusione* cioè escludere più di un accesso alla volta. Ciò però non basta: la mutua esclusione garantisce solo che non succeda qualcosa e può tranquillamente essere realizzata spegnendo tutto. Occorre anche garantire che ci sia un *progresso* quindi bisogna garantire che se un processo Pi vuole eseguire una sua sezione critica e nessun altro la sta occupando, il processo non aspetterà in modo indefinito ma prima o poi gli arriverà la risorsa. Inoltre, l'attesa dev'essere *limitata* nel senso che se il processo Pi sta aspettando che la sua sezione critica diventi disponibile ci dev'essere un limite sul numero di volte che altri processi entrino in questa sezione critica.

Queste tre soluzioni sono in parte le risposte ai problemi evidenziati in precedenza. Per ottenere questi risultati molto dipende dall'*hardware*, cioè dal microprocessore e dall'architettura (in particolare single core o multicore), e dal *kernel*, cioè da come il kernel supporti eventuali primitive di sincronizzazione.

Ad esempio, possiamo essere su un sistema single core, quindi con un solo processore, e preemptive, dove si intende la possibilità di interrompere e fermare un processo o un thread di kernel attivo. Potremmo essere su un sistema single core non-preemptive invece, quando c'è un task in esecuzione sulla CPU e lo si deve lasciar andare, oppure può essere un'architettura multicore che si differenzia dal single core perché su un quest'ultimo è sempre in esecuzione un solo programma o una sola parte di codice alla volta, mentre nel multicore è possibile che ci siano più programmi in esecuzione nello stesso momento.

Se ci sono n processi e ogni processo ha la sua sezione critica, cioè la parte di codice su cui vorrebbe lavorare in mutua esclusione, questi processi potrebbero usare questa sezione critica per scambiarsi dati, ad esempio in variabili comuni.

Bisognerà definire un protocollo con cui si accede alla sezione critica e anche delle modalità per entrare o uscire ed eventualmente poi realizzare una parte residua o finale della sezione critica.

Molto spesso lo schema utilizzato è quello di avere, come vedete in questa figura, una sezione critica che è racchiusa in mezzo a un punto d'ingresso e un punto di uscita e poi c'è in questa sezione una parte non critica ovvero un residuo la quale non va eseguita con problemi di conflitto con altri thread o processi.

Possiamo anche dire che la mutua esclusione è un modo di realizzare questa sezione critica ma non solo, l'esempio propone una ripetizione iterativa della sezione critica.

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

Peterson's Solution

Vediamo soluzioni che cercano di illustrare come si può arrivare a gestire il problema della sincronizzazione a basso livello (a livello di istruzioni macchina) e/o di architettura hardware. Diamo un'occhiata per prima alla soluzione di Peterson.

Supponiamo di essere in un caso in cui ci siano due processi che utilizzano una sezione critica in comune. Assumiamo poi che ci siano due operazioni disponibili di load e store a livello di assembler e che siano istruzioni atomiche (dove atomiche significa che possono essere interrotte). I due processi hanno in comune due variabili una variabile intera *turn* che indica a chi tocca entrare nella risorsa critica e questa variabile *turn* potrà essere zero o uno per indicare che tocca al processo zero oppure al processo uno, ovviamente non ne potrà indicare due contemporaneamente, ne potrà indicare solo uno. Poi c'è un vettore di due flag booleani che potrà essere un vettore di qualsiasi tipo (char, interi, booleani), l'importante è che ci sia il vettore di flag che, per ognuno dei due processi attivi, indicherà se il processo è pronto a entrare nella sezione critica e come potete immaginare sono possibili quattro configurazioni: nessuno dei due vuole entrare, solo uno vuole entrare e l'altro no, oppure entrambi vogliono entrare (situazione di conflitto).

L'algoritmo è riportato a destra ed è scritto per il processo Pi, quindi con i che può valere sia zero che uno, questo codice viene replicato due volte.

Può essere una specie di loop infinito in cui quello che si vuol fare è eseguire la sezione critica e poi eseguire la sezione remainder, la sezione critica dev'essere eseguita in mutua esclusione garantendo comunque il progresso.

```

while (true){
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;
    /* critical section */
    flag[i] = false;
    /* remainder section */
}

```

Per fare questo si realizza il seguente inizio di protocollo: } innanzitutto l'i-esimo processo dichiara di essere pronto a entrare nella sezione critica, poco prima mette il suo flag a vero e lo rimette a falso appena smesso di usare la sezione critica. Poi dice sostanzialmente *turn* uguale *j* (l'altro processo) cioè se *i* è zero *j* sarà uno e viceversa, questa è la chiave del funzionamento del protocollo: ognuno dei due processi dirà "tocco all'altro". Siccome questa variabile non è una variabile comune, succederà che l'ultimo dei due processi che darà il turno all'altro condizionerà le azioni.

Poi si aspetta fintanto che tocca all'altro (*turn==j*) e finché tocca all'altro e l'altro ha chiesto di poter entrare (*flag[j]* è vero); cioè se tocca all'altro e l'altro processo vuole usare la sezione critica, aspetto questo while.

Si passa per due possibili condizioni: toccherebbe all'altro ma l'altro non vuole la risorsa, oppure l'altro vuole la risorsa ma tocca a me. L'unico modo per essere bloccati è che tocchi proprio all'altro e l'altro voglia usare la sezione critica. Quando si esce da questo loop è garantito che si vuole entrare nella sezione critica in qualche modo l'altro non entra.

Si potrebbe dimostrare che questa soluzione garantisce la *mutua esclusione*, garantisce il *progresso* e il *bounded-waiting*.

Tuttavia, non è detto che funzioni sulle architetture. Il motivo principale è che su un'architettura moderna si possono fare più cose in parallelo, il fatto che un processo sia in esecuzione non esclude l'altro, è possibile che entrambi i processi siano in esecuzione contemporaneamente.

Semafori

Introduciamo adesso i semafori e vediamo come possono essere utilizzati per “proteggere” una sezione critica. Innanzitutto, bisogna dire che il semaforo è una primitiva di sincronizzazione molto versatile che può essere utilizzato sia per creare un lock che altri meccanismi di sincronizzazione.

È costituito da due funzioni:

- **P** (anche detta wait)
- **V** (anche detta signal)

Queste sono utilizzate come fossero dei mutex locks.

In termini pratici, un semaforo non è altro che una variabile di tipo int che può essere acceduta e modificata utilizzando le funzioni wait e signal.

Facciamo un esempio:

```
wait(S) {  
    while (S <= 0)  
        : // busy wait  
    S--;  
}  
  
signal(S) {  
    S++;  
}
```

In questa particolare implementazione del semaforo, si aspetta solo quando il contatore intero (S), è minore o uguale a 0. Questo che abbiamo visto, è anche detto **counting semaphore**, oltre questo, esiste anche un'altra versione detta **binary semaphore**. La differenza è che in quest'ultimo il valore del semaforo può essere solo 0 e 1 (è come un mutex lock).

Tuttavia, vale la pena notare che resta un'importante differenza tra un mutex lock e un semaforo binario. Mentre nel primo caso chi acquisisce il mutex lo deve anche rilasciare (esiste un concetto di ownership del mutex), nel semaforo c'è libertà e si può fare signal e wait da processi diversi.

Proprio per questo motivo I semafori possono essere utilizzati per realizzare svariati schemi di sincronizzazione. Ad esempio, supponiamo la presenza di due processi P1 e P2 che portano a compimento rispettivamente I task S1 e S2:

```
P1:  
    S1;  
    signal(synch);  
  
P2:  
    wait(synch);  
    S2;
```

In questo caso non si è realizzata una mutua esclusione, bensì I semafori sono stati utilizzati per creare uno schema di precedenza (S1 viene eseguito prima di S2).

Vediamo adesso come può essere implementato un semaforo, ossia come vengono sincronizzate le Wait e le Signal, in modo tale che non possano essere effettuate sullo stesso semaforo allo stesso tempo. Esistono due modi di farlo:

- Implementazione con Busy Wait
- Implementazione senza Busy Wait

Nel primo caso, il semaforo viene realizzato utilizzando una **variabile condivisa** in mutua esclusione. Il tutto si riconduce quindi alla gestione della sezione critica con una problematica in particolare: uno dei due processi si troverà in busy waiting, il che non è ottimale a livello di prestazioni.

L'alternativa, che permette di prevenire il busy waiting, prevede che il processo che deve attendere venga messo in attesa in una coda d'attesa (ogni semaforo avrà la sua).

Il processo in coda, effettua la wait in modo da rendersi conto se può (effettuare la wait, quindi se il semaforo ha un valore consono, che non sia negativo o uguale a zero).

Si utilizzeranno due operazioni:

- Block: blocca il processo perchè la wait “scopre” che non può andare avanti
- Wakeup: risveglia il processo quando la signal cambia il valore del semaforo e quindi magari la wait questa volta può essere effettuata.

A livello realizzativo, questa volta il semaforo non sarà più una variabile intera ma una struct, contenente il valore del semaforo e la lista di processi in attesa.

Sincronizzazione in OS161

Vediamo in OS161 come si realizzano i semafori e le altre primitive di sincronizzazione.

La prima idea è quella di lavorare in un sistema uniprocessore(una cpu), quindi si sfrutta il concetto che ci può essere un solo thread alla volta in esecuzione, quindi se il thread attualmente in esecuzione è in una sezione critica, allora la mutua esclusione su questa sezione critica può essere violata solamente se:

- 1) Il thread attuale è pre-empted, oppure lascia volontariamente la CPU mentre è nella sezione critica
- 2) Lo scheduler sceglie un altro processo da mandare in esecuzione e questo nuovo thread entra nella sezione critica in cui c'era il thread pre-empted.

Siccome la pre-emption può essere generata soltanto dall'interrupt di un timer, per ottenere la mutua esclusione, è sufficiente disabilitare gli interrupt dei timer prima che un thread entri in una sezione critica e riabilitarli una volta che il thread lasci la sezione critica. In OS161 è questo il modo in cui si ottiene la mutua esclusione, utilizzando le funzioni splhigh(), spl0(), splx() per abilitare e disabilitare le interruzioni.

Questo è un modo semplice per realizzare un semaforo nella versione 1.9x di OS161:

```
struct semaphore {  
    char *name;  
    volatile int count;  
};  
  
struct semaphore *sem_create(const char *name,  
    int initial count);  
void P(struct semaphore *);  
void V(struct semaphore *);  
void sem_destroy(struct semaphore *);
```

La funzione **wait** viene realizzata in questo modo:

```
void P(struct semaphore *sem) {
    int spl;
    assert(sem != NULL);

    /* May not block in an interrupt handler.
     * For robustness, always check, even if we can actually
     * complete the P without blocking. */
    assert(in_interrupt==0);

    spl = splhigh();
    while (sem->count==0) {
        thread_sleep(sem);
    }
    assert(sem->count>0);
    sem->count--;
    splx(spl);
}
```

Notiamo che con `splx(spl)` finale, si riabilita il livello di interruzione solo qualora prima fosse già abilitato. Questo schema potrebbe essere errato qualora la `wait` venisse chiamata con le interruzioni già disabilitate. Ricordiamo che questa operazione viene effettuata per usare in mutua esclusione il contatore del semaforo.

Per quanto riguarda la **signal** invece:

```
void V(struct semaphore *sem)
{
    int spl;
    assert(sem != NULL);
    spl = splhigh();
    sem->count++;
    assert(sem->count>0);
    thread_wakeup(sem);
    splx(spl);
}
```

Vediamo adesso un'altra implementazione del semaforo, questa volta senza busy waiting:

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

Rispetto a prima la semantica è leggermente diversa perché il valore del semaforo può essere negativo.

Thread blocking in OS161

Capiamo adesso come vengono realizzate il block e il wakeup.

In OS161, queste due operazioni vengono effettuate rispettivamente con:

- **void thread_sleep(const void *addr)** blocca il thread chiamante sul semaforo
- **void thread_wakeup(const void *addr)** sblocca i thread in attesa sul semaforo (indirizzo)

La **thread_sleep()** è molto simile alla **thread_yield()**, il thread chiamante rinuncia volontariamente alla CPU, lo scheduler sceglie un nuovo thread da eseguire e invia il nuovo thread in esecuzione. Tuttavia:

- Dopo una **thread_yield()**, il thread chiamante è pronto ad andare subito in esecuzione non appena viene scelto dallo scheduler.
- Dopo una **thread_sleep()**, il thread chiamante è bloccato e non potrà essere schedulato di nuovo finché non sarà sbloccato da una chiamata a **thread_wakeup()**.

OS161 locks

Sono delle primitive di sincronizzazione che sono “impostate” ma vanno implementate, nel senso che esistono le interfacce e le funzioni, ma vanno completate a seconda delle necessità.

Un Lock è molto simile ad un semaforo binario che parte con un valore iniziale di 1 e che può essere decrementato con **lock_acquire(mylock)** e re-incrementato con **lock_release(mylock)**.

Tuttavia, i Locks hanno un vincolo aggiuntivo: il thread che rilascia un lock deve essere lo stesso che l'ha acquisito più recentemente, nel semaforo questo non succede (la release può essere fatta da qualsiasi processo)

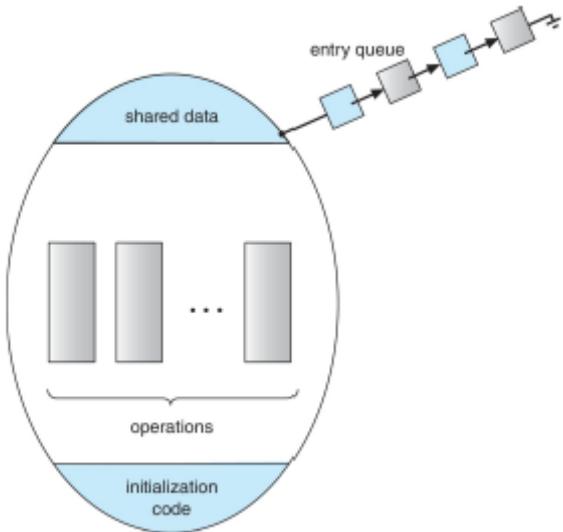
Limitazioni sui semafori

Per varie motivazioni, scrivere correttamente programmi che utilizzano i semafori come primitiva di sincronizzazione può risultare difficile:

- Sotto alcuni aspetti sono contro-intuitivi.
- I motivi per cui si attende ad un semaforo sono limitati esclusivamente alla P (wait):
 - * Il semaforo è limitato ad una condizione per attendere e questa è connessa ad un contatore, si aspetta se un certo contatore non rispetta una determinata condizione e viceversa.
 - * Non è possibile realizzare attese su condizioni più complesse (ad esempio su 3 variabili)
- Possibili deadlock

Per queste motivazioni, si è pensato di realizzare dei costrutti di più alto livello come i **monitor**, questo permette di realizzare uno schema di sincronizzazione molto simile a quello che si potrebbe utilizzare in un linguaggio ad oggetti.

A livello schematico, un monitor può essere visto come un contenitore in cui ci sono dati condivisi, un codice di inizializzazione e delle operazioni da poter effettuare. E' inoltre presente una coda d'accesso.



Vediamo uno schema di Wait on condition che potrebbe essere utilizzato per realizzare un monitor:

```
/* shared state vars with some initial value */
int x,y,z;
/* mutual exclusion for shared vars */
struct lock *mylock = lock_create("Mutex");
/* semaphore to wait if necessary */
struct semaphore *no_go = sem_create("MySem", 0);

compute_a_thing {
    lock_acquire(mylock); /* lock out others */
    /* compute new x, y, z */
    x = f1(x); y = f2(y); z = f3(z);
    if (x != 0 || (y <= 0 && z <= 0)) V(no_go);
    lock_release(mylock); /* enable others */
}
```

```

use_a_thing {
    lock_acquire(mylock); /* lock out others */
    if(x == 0 && (y > 0 || z > 0))
        P(no_go);
    /* Now either x is non-zero or y and z are
       non-positive. In this state, it is safe to run
       "work" on x,y,z, which may also change them */
    work(x,y,z);
    lock_release(mylock); /* enable others */
}

```

compute_a_thing e use_a_thing rappresentano una sorta di schema produttore-consumatore.

In mutua esclusione, il produttore effettua un task che modifica x, y, z ed eventualmente sveglia mediante una signal il consumatore.

Quest'ultimo va in mutua esclusione, controlla il valore di x, y, z e se non sono “in valori che piacciono”, si mette in attesa.

Tuttavia, questo schema presenta un problema di deadlock: mentre il processo consumatore è in attesa e ha ownership sul lock, non è possibile che un eventuale produttore lavori in mutua esclusione.

La **SOLUZIONE** per evitare un problema di questo tipo è far rilasciare il lock al consumatore mentre è in attesa e farglielo riprendere non appena il thread viene risvegliato.

Questa soluzione tuttavia, lascia una finestra temporale in cui il lock non è posseduto e in cui le variabili condivise possono essere modificate. Per questo motivo, quando il thread viene risvegliato, è necessario testare di nuovo le condizioni.

Vediamo quindi lo schema fixato:

```

use_a_thing {
    lock_acquire(mylock); /* lock out others */
    while (x == 0 && (y > 0 || z > 0)) {
        lock_release(mylock); /* no deadlock */
        P(no_go);
        lock_acquire(mylock); /* lock for next test */
    }
    /* Now either x is non-zero or y and z are
       non-positive. In this state, it is safe to run
       "work" on x,y,z, which may also change them */
    work(x,y,z);
    lock_release(mylock); /* enable others */
}

```

Il momento critico intercorre tra P(no_go) e lock_acquire(mylock), in questo lasso di tempo intercorre la possibilità che altri processi acquisiscono il lock e modificano la condizione per cui si è stati svegliati.

Per fare in modo che la condizione venga ri-testata una volta che il thread viene svegliato, l’if è stato sostituito da un while.

Condition variables (cv)

Fanno un lavoro simile ai semafori ma sono più automatizzate ad alto livello, nascondendo i dettagli dell’acquisizione/rilascio del lock.

Su una condition variable sono possibili due operazioni (per altro molto simili a quelle realizzate dai semafori):

- **x.wait()** - un processo che invoca questa funzione è sospeso fino a x.signal()
- **x.signal()** - risveglia uno dei processi che hanno invocato x.wait() e qualora non ci fossero, la funzione non ha effetto sulla variabile. Questa è una differenza importante rispetto a un semaforo, infatti, nel caso in cui stessimo utilizzando un semaforo, la signal modificherebbe comunque il contatore e l'effetto della signal sarà comunque visibile a chi farà wait dopo.

Una CV è un Abstract data type che incapsulano pattern di sincronizzazione e dal punto di vista interno costituisce un meccanismo che gestisce code, quindi attesa/riresa dell'esecuzione. Una CV funziona solo se associata ad un lock che garantisce la mutua esclusione su alcuni dati.

Le operazioni(atomiche) sono:

- **cv_wait(struct cv *cv, struct lock *lock)** - rilascia un lock, aspetta e riaccidisce il lock prima di tornare.
- **cv_signal(struct cv *cv)** - sveglia un thread in coda.
- **cv_broadcast(struct vb *cv)** - sveglia tutti i thread in coda

Come già detto precedentemente, se nessun thread è in attesa, cv_signal e broadcast non hanno effetto.

In OS161, la signal e la broadcast ricevono come ulteriore parametro il lock (anche se non servirebbe) in modo da controllare l'ownership su di esso.

Uno schema di sincronizzazione basato su CV (in questo caso utilizzando cv_signal) dovrà essere utilizzato con dei lock e potrà seguire uno schema di questo tipo:

P_i	P_j
lock_acquire(lock);	lock_acquire(lock);
while (condition not true)	... // modify condition
{	cv_signal(cond);
cv_wait(cond, lock);	lock_release(lock);
}	
... // do stuff	
lock_release(lock);	

È opportuno notare che è meglio mettere il while quando ci sono più processi che potrebbero modificare le variabili di condizione.

Un altro schema è quello della cv_broadcast:

P_i

```
lock_acquire(lock);
while (condition_i not true) {
    cv_wait(cond, lock);
}
... // do stuff
lock_release(lock);
```

P_j

```
lock_acquire(lock);
...
// modify conditions
// either for Pi or Pk
cv_broadcast(cond);
lock_release(lock);
```

P_k

```
lock_acquire(lock);
while (condition_k not true) {
    cv_wait(cond, lock);
}
... // do stuff
lock_release(lock);
```

P_j modifica le variabili condivise e invoca la cv_broadcast, chi era in cv_wait viene svegliato e verifica le proprie condizioni per vedere se si deve rimettere in wait o può ricontinuare.

Vediamo quindi come si può realizzare uno schema di Wait on Condition utilizzando le CV invece dei semafori:

```
/* shared state vars with some initial value */
int x,y,z;
/* mutual exclusion for shared vars */
struct lock *mylock = lock_create("Mutex");
/* condition variable to wait if necessary */
struct cv *no_go = cv_create("CondV");

compute_a_thing {
    lock_acquire(mylock); /* lock out others */
    /* compute new x, y, z */
    x = f1(x); y = f2(y); z = f3(z);
    if (x != 0 || (y <= 0 && z <= 0)) cv_signal(no_go);
    lock_release(mylock); /* enable others */
}
```

```

use_a_thing {
    lock_acquire(mylock); /* lock out others */
    while (x == 0 && (y > 0 || z > 0)) {
        cv_wait(no_go, mylock);
    }
    /* Now either x is non-zero or y and z are
       non-positive. In this state, it is safe to run
       "work" on x,y,z, which may also change them */
    work(x,y,z);
    lock_release(mylock); /* enable others */
}

```

Lo schema è simile a quello utilizzato nei semafori ma c'è la gestione implicita del lock.

Per quanto riguarda OS161, oltre agli spinlock, è presente una seconda primitiva di sincronizzazione detta **Wait Channel**.

Siccome in OS161 sono presenti sia le CV che i Wait Channel, per evitare confusione, si può pensare che i Wait Channel stanno ai CV come gli Spinlock stanno ai Lock.

Quindi non sono altro che delle condition variables utilizzabili nel kernel e che non hanno bisogno di essere scritte.

Il difetto delle Wait Channel rispetto alle CV, è che utilizzano gli spinlock che portano con se la problematica del busy waiting. Per questo motivo, in OS161, gli spinlock innestati o multipli non sono permessi.

Per quanto riguarda le operazioni che si possono utilizzare:

- **wchan_sleep(struct wchan *wc, struct spinlock *lk)** - corrisponde alla cv_wait.
- **wchan_wakeone(struct wchan *wc, struct spinlock *lk)** - corrisponde alla cv_signal.
- **wchan_wakeall(struct wchan *wc, struct spinlock *lk)** - corrisponde alla cv_broadcast.

Per quanto riguarda l'implementazione della sleep:

Guardando le KASSERT, si può supporre che una sleep sia fatta quando:

- non si è in un'interruzione
- si possegga lo spinlock ricevuto
- non si può essere owner di uno spinlock se in quel momento si ha la ownership su un altro spinlock.

Se tutte queste condizioni sono rispettate, il thread viene messo in stato d'attesa con l'operazione **thread_switch**.

```

wchan_sleep(struct wchan *wc, struct spinlock *lk) {
    /* may not sleep in an interrupt handler */
    KASSERT(!curthread->t_in_interrupt);

    /* must hold the spinlock */
    KASSERT(spinlock_do_i_hold(lk));

    /* must not hold other spinlocks */
    KASSERT(curcpu->c_spinlocks == 1);

    thread_switch(S_SLEEP, wc, lk);
    spinlock_acquire(lk);
}

```

Per quanto riguarda la wakeone:

```
wchan_wakeone(struct wchan *wc, struct spinlock *lk) {
    struct thread *target;
    KASSERT(spinlock_do_i_hold(lk));

    /* Grab a thread from the channel */
    target = threadlist_remhead(&wc->wc_threads);

    /* Note that thread_make_runnable acquires a runqueue
       lock while we're holding LK. This is ok; all
       spinlocks associated with wchans must come before the
       runqueue locks, as we also bridge from the wchan lock
       to the runqueue lock in thread_switch. */

    thread_make_runnable(target, false);
}
```

Terminiamo notando che nella versione multicore di OS161 i semafori non sono fatti come in precedenza, cioè sfruttando i livelli di interruzione.

In realtà, la vera differenza rispetto alle versioni < 2.0 è nella realizzazione interna della struct semaphore. In questo caso, oltre al contatore, c'è anche lo spinlock e il Wait Channel.

```
struct semaphore {
    char *name;
    struct wchan *sem_wchan;
    struct spinlock sem_lock;
    volatile int count;
};
```

Per quanto riguarda la P():

```
void P(struct semaphore *sem) {
    /* May not block in an interrupt handler. For robustness,
       always check, even if we can actually complete the
       without blocking. */
    KASSERT(curthread->t_in_interrupt == false);

    /* Use the semaphore spinlock to protect the wchan as well */
    spinlock_acquire(&sem->sem_lock);
    while (sem->sem_count == 0) {
        /* Note that we don't maintain strict FIFO ordering of
           threads going through the semaphore; */
        wchan_sleep(sem->sem_wchan, &sem->sem_lock);
    }
    KASSERT(sem->sem_count > 0);
    sem->sem_count--;
    spinlock_release(&sem->sem_lock);
}
```

Per quanto riguarda la V():

```
void V(struct semaphore *sem)
{
    KASSERT(sem != NULL);

    spinlock_acquire(&sem->sem_lock);

    sem->sem_count++;
    KASSERT(sem->sem_count > 0);
    wchan_wakeone(sem->sem_wchan, &sem->sem_lock);

    spinlock_release(&sem->sem_lock);
}
```

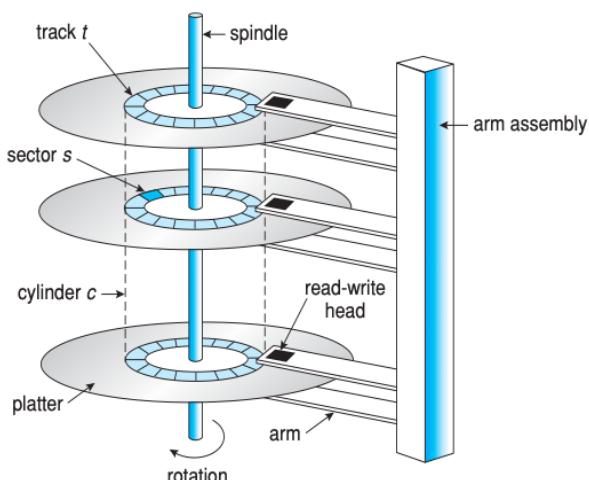
6. Memorie di Massa

Dispositivi di memoria di massa

Per prima cosa diciamo subito che mentre tradizionalmente i dispositivi di memoria di massa erano dischi basati su tecnologie magnetiche, con componenti meccaniche non trascurabili, ultimamente si affiancano le memorie non volatili nvm oppure Solid disk Drive che vuol dire sostanzialmente tecnologie molto più simili a quelle delle RAM ma di fatto da adottare per dispositivi simili ai dischi. Per quanto riguarda i dischi tradizionali qui detti HDD, si tratta quindi di dispositivi nei quali c'è una rotazione con accelerazioni e decelerazioni, c'è un tempo di trasferimento, c'è un tempo di posizionamento che può essere distinto in posizionamento di una testina in senso radiale a cercare il cilindro o traccia desiderata detto seek time e ha anche una rotazione del disco per andare a trovare il settore su disco desiderato, tutto questo implica che i dischi a disk non erano/non sono velocissimo o comunque sono molto meno veloci delle memorie RAM, e sono anche soggetti ad errori nel caso in cui la testina vada a toccare la superficie del disco. Qui vedete una rappresentazione semplificata, del fatto che a tutti gli effetti si tratta di dischi che possono avere similitudini con quelle che sono stati CD piuttosto che non DVD, piuttosto che non ho neanche i classici dischi in vinile, l'idea sostanzialmente è tuttavia quella di avere più dischi in parallelo sui quali sono mediante tecniche di orientamento dei dipoli magnetici su una spalmata con materiale ferromagnetico, sono rappresentate delle tracce che insieme formano i cosiddetti cilindri, ogni traccia è divisa in settori, nei quali diciamo gruppi di dipoli magnetici riescono a essere visti come un bit o dei Byte, e lo zero è l'uno sono di fatto visti a seconda che il dipolo magnetico sia orientato in una direzione piuttosto che non nell'altra. Delle testine diciamo sorrette da un braccio porta testine, servono a leggere o scrivere, che vuol dire rilevare l'orientamento dei dipoli o modificare l'orientamento dei dipoli magnetici e queste testine sono più d'una di solito, sulle due facce di un disco con più testine in parallelo e che servono a leggere o scrivere bit e se ne vediamo tanti in parallelo Byte o anche di più su dischi diversi normalmente. Supponendo che ad esempio queste testine siano 8, normalmente si potrebbe pensare che 8 testine su 8 facce di 4 dischi rappresentati con materiale ferromagnetico su entrambe le facce, queste 8 testine in parallelo leggono scrivono in parallelo 8 bit che possono essere visti come un byte che potrebbe essere visto come rappresentato in modo distribuito su 8 facce di 4 dischi in parallelo. Un disco è rappresentato da caratteristiche tecnologiche. Le dimensioni storicamente possono essere da una frazione di pollice a 14 pollici, le velocità si può pensare di arrivare a

veloce trasferimento teorico di gigabit al secondo, nelle trasmissioni dati solitamente si rappresentano le velocità in bit al secondo non in Byte al secondo. La velocità di trasferimento effettiva reale potrebbe essere anche abbastanza minore fino a quasi un ordine di grandezza inferiore a quella teorica. Si può notare che i tempi di seek e poi anche la latenza rotazionale sono molto peggiori, nel senso che per andare a cercare una traccia o cilindro ci possiamo mettere dei millisecondi che rispetto alla ai nanosecondi (che sono l'inverso dei gigabit al secondo) sono molto elevati. La latenza potete vedere che la latenza rotazionale, cioè quanto ci mettiamo a far girare il disco per raggiungere il settore che si cerca, potrebbe essere misurata in base a dei quanti giri al secondo o al minuto, e di solito si parla di giri al minuto, rpm/ 60 è il numero di giri al secondo, e l'inverso è il tempo misurato il secondo per fare un intero giro.

Hard disk performance



Latenza di accesso=tempo medio di accesso = tempo medio seek + tempo medio di latenza rotazionale (ordine di ms).

il tempo per effettuare un I/O è dato da tempo medio di accesso (latenza di accesso) + tempo effettivo per trasferire i dati, quantità ti vogliamo trasferire, moltiplicato quanto ci va trasferire un singolo dato, oppure diviso per la velocità, + l'eventuale overhead.

I dischi magnetici hanno dei limiti nelle prestazioni, pur essendo migliorati notevolmente con la tecnologia negli ultimi anni.

Memoria non volatile

Solid State state SSD, oppure hai altre forme di dischi USB basati sempre su tecnologie elettroniche. Le memorie Solid State sono tendenzialmente più piccole a parità di prezzo (sono più care), perché sono decisamente più veloci, e possono avere un tempo di vita più basso, in genere sono un po meno capaci ma sono più veloci. Una memoria non volatile e una scheda con dei chip, assomiglia abbastanza le schede di ram, è una tecnologia simile ma un po' più complicata. Sono scritte e lette non per Byte o per Word, come le memorie RAM, ma per pagine settori o comunque dei blocchi di dati; Per essere scritte devono essere prima cancellato e poi riscritte, e esiste un limite sul numero massimo di riscrittura che possono essere fatte, e questo è in qualche modo un limite il vero limite tecnologico e che impone delle strategie per rendere questi dispositivi efficaci. Bisogna fare in modo che tutte le parti del disco tutti i Byte tutti i blocchi siano più o meno riscritti in quantità equilibrata, in modo che invecchiano tutte le parti insieme. La vita media quindi viene misurata, non in numero totale di riscritture, ma in numero di riscrittura del disco per giorno **Drive Write per Day**, ad esempio un un disco di un terabyte con 5 riscrittura giornaliere massime, può essere in qualche modo inteso come: si possono riscrivere 5 terabyte al giorno al massimo, vuol dire riscriviamo tutto il disco 5 volte al giorno, a patto di riscriverlo tutto. Per passare dal numero di riscrittura giornaliere a quello totale, bisogna andare a leggere la garanzia del disco cioè capire per quanti giorni/mesi/anni è garantito che quel disco funzioni.

NAND Flash Controller Algorithms

Se non si fanno riscrittura le pagine, conterranno un po di dati validi e un po di dati non validi, il gestore di uno di queste di questi dispositivi dovrà avere una tabella che dice quali sono le pagine che hanno dei dati validi e quelli che non hanno dati validi.

Memoria volatile

I dispositivi di memoria usati come memoria di massa possono essere complementati anche dalle cosiddette RAM, cioè dalle Memorie di dalle memorie volatili, che vuol dire che quando si spegne il dispositivo perde il contenuto, per essere utilizzati come dischi temporanei più veloci si chiamano RAM drivers, cioè dei dischi RAM, che possono servire laddove di fatto servono di veloci purché si accetti che questi dischi vanno alla fine salvati e poi ripristinati momenti in cui si spegne il l'elaboratore.

Nastro magnetico e dischi ottici

I dischi magnetici e dischi SSD non sono necessariamente gli unici a che possono essere realizzati ci sono memorie di massa basate su nastro magnetico.

Esistono poi anche i dischi ottici CD DVD.

Strutture dischi (magnetici)

Un disco magnetico può essere visto come un vettore di blocchi, dove il blocco è una unità di memorizzazione immagazzinamento dei dati che può essere conveniente leggere scrivere in un colpo solo, quindi dal punto di vista logico il disco può essere visto come vettore di blocchi logici. Il vettore monodimensionale di blocchi logici, deve essere mappato con dei blocchi fisici, che invece hanno tutta un'altra architettura, si utilizza il termine “geometria” nel senso che questi blocchi vanno rimappati su quello che sono più dischi con più facce con delle tracce circolari che formano dei cilindri e con tanti settori su ognuna delle tracce, normalmente si fanno coincidere dei blocchi con dei settori, oppure ogni settore contiene un po' di blocchi e i settori sono numerati da 0 in avanti dal primo settore sulla prima traccia più periferica andando avanti fino a che si raggiungono le tracce più centrali. Su un disco si possono anche definire o riscontrare dei cosiddetti “Bad Sector” settori guasti con i quali per un po' si può sopravvivere, fino a che un disco non comincia ad avere molti settori guasti.

Disk Attachment

Si chiama poi disk attachment una struttura in cui si discute anche di come un disco viene agganciato, attaccato e quindi interagisce scambia dati con la CPU, la memoria con un con un sistema, e quindi a che vedere con i Bus che vengono usati per far interagire il disco con il resto del sistema.

HDD schedulazione

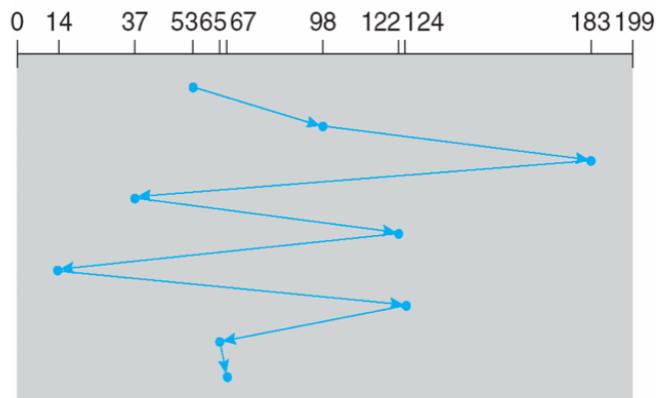
Il termine schedulazione di un hard disk di un hard Drive, cioè di un disco magnetico, sta indicare una componente del sistema operativo, che deve decidere dove andare a scrivere o leggere su un disco, per minimizzare ottimizzare i tempi, basandosi su una politica è un problema che assomiglia moltissimo al problema di gestione di un ascensore in un edificio piuttosto alto e con molti piani, cioè il problema è quello che siccome, in un sistema operativo ci possono essere più processi/thread che chiedono in contemporanea accessi a parti diverse del disco, anziché servire queste richieste semplicemente con una politica del tipo “ordiniamo le richieste in modo cronologico e riserviamo strettamente secondo questo ordine”, si può cercare di fare in modo che le richieste siano riordinate quindi schedulato e opportunamente per ottimizzare alcuni criteri.

Esempio:8

Supponiamo ad un certo punto su un certo disco caratterizzato da una numerazione che siano blocchi o che siano settori, e con una disponibilità ad esempio 0-199, ad un certo istante ci sono più processi che vorrebbero accedere in lettura o scrittura al numero 98 183 37122 14 e così via...

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



le richieste secondo l'ordine cronologico, si fa uno zig-zag.

SCAN

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

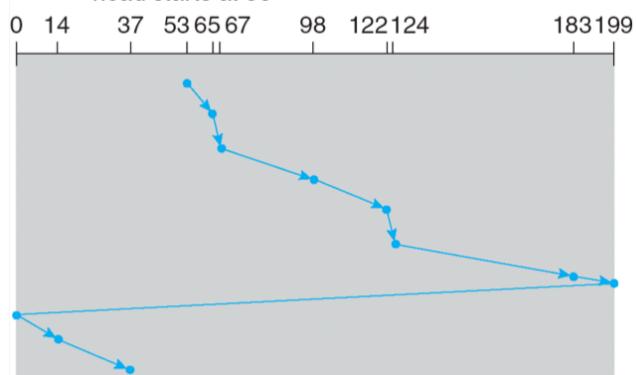


Supponiamo che questi numeri per semplicità siano visti ora come numeri di cilindri (differenza tra cilindro una traccia che una traccia e su una superficie di un disco il cilindro sono tante tracce parallele), è che il puntatore della testina è che la testina di lettura scrittura sia in questo momento sul cilindro numero 53. Adottando una politica detta fcfs che sta per first come, first serve (significa Fifo), la classica coda. Supponendo di partire vedete in figura dal cilindro 53 e servendo

Una politica alternativa, che è ispirata all'ascensore: si adotta una direzione del percorso e mentre si sale o si scende si servono le varie richieste che si trovano sul percorso, in questo modo si percorre meno strada.

C-SCAN

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53



Rispetto all'altra si eseguono le richieste solo in una direzione, poi si effettua una discesa rapida e poi si comincia a risalire. Quindi in pratica come se si trattassero i cilindri con una lista o vettore circolare.

Scelta algoritmo di Schedulazione

Lo shortest seek Time first, si prende la richiesta più vicina e un'eventuale alternativo. Uno dei possibili problemi da affrontare quello della starvation, che vuol dire c'è un utente che deve aspettare molto/troppi. In Linux c'è uno scheduler detto a **deadline** che mantiene code separate per lettura e scrittura dando maggiore priorità alla lettura perché è un po' più efficiente veloce e

tendenzialmente i processi si fermano devono aspettare di più da lettura che non da scrittura. In Linux ci sono quattro code due per la lettura due per la scrittura la differenza è che a seconda delle code si usano algoritmo di scheduling.

NVM Scheduling

Se non ci sono problemi di latenza e spostamenti le politiche appena viste sono inutili. Una memoria non volatile è più veloce per accesso casuale, mentre un disco con tecnologia classica ha prestazioni migliori per lettura sequenziale. Le operazioni di input/output sono molto più veloci con dischi NVM.

Error Detection and Correction

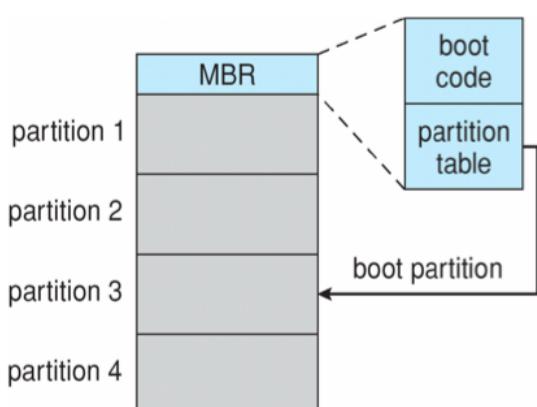
Nel momento in cui su un disco ci sono informazioni sbagliate, è opportuno individuare l'errore ed eventualmente correggerlo. Individuare l'errore in un file equivale a dire che il file non è affidabile e si spera di avere una copia non sbagliata, diverso è se c'è un errore, lo rivelò e sono anche in grado di correggerlo. Ci sono svariate tecniche, vicine alle problematiche per la compressione delle informazioni per la crittografia, quindi si tratta di problemi di codifica dell'informazione, tipicamente con una ridondanza che è in grado di gestire opportunamente queste cose.

Storage Device Management

Dal punto di vista della gestione del disco, un disco va formattato, può avvenire a livello fisico o a livello logico, ma significa che nel passaggio da un disco a essere visto come semplice sequenza di blocchi settori, cilindri o altro a un Disco organizzato in modo logico in un filesystem ci sono svariate posizioni intermedie. Diciamo semplicemente con una **formattazione fisica** significa dividere un disco in settori che possono essere trattati come parti da leggere o scrivere in modo individuale, e ogni settore si possono aggiungere delle informazioni per rilevamento e gestione e correzione dell'errore. Un Filesystem è necessario quando si vogliono immagazzinare file, cartelle etc.., per arrivare a un filesystem si deve partizionare il disco in gruppi di cilindri che vanno visti ognuno come se fosse un sottodisco, un disco fisico

come più dischi logici, e su ognuno di questi dischi logici si potrà fare una formattazione logica che si chiamerà filesystem. La **Root Partition**, partizione radice può essere quella che contiene il sistema operativo, mentre altre partizioni possono contenere altri sistemi operativi, oppure dei file system, oppure essere partizioni raw, nel senso che non hanno un filesystem logico. Le partizioni possono essere **mounted**, cioè agganciate a un sistema operativo in esecuzione alla fase di bootstrap, oppure in modo manuale o automatico. Quando si fa Mount della partizione o di un filesystem, c'è una fase di verifica di consistenza, cioè che i dati siano corretti, poi nella partizione root oppure in una partizione che di fatto estende la partizione di Boot normalmente c'è un blocco, che viene detto blocco di Boot, cioè il blocco che va

copiato direttamente in RAM per far partire il sistema.



Una parte della RAM detta Rom con a bordo il cosiddetto Basic I/O System, cioè il Bios, deve dare la possibilità di far partire le operazioni alla partenza. Il Bios va a leggere e caricare il Master Boot record, quello iniziale, nel quale c'è un po' di codice di Boot o in un pezzo del codice del cosiddetto bootloader, c'è la cosiddetta Partition table oppure le informazioni su dove trovare questa Partition table, ma in sostanza ci sono le informazioni necessarie direttamente o indirettamente per conoscere come viene partizionato il disco e dove andare a prendere il kernel da far strappare, cioè dove andare a trovare le informazioni da caricare in RAM perché vi sia il kernel in RAM operativo. Detta fase di bootstrap.

Swap-Space Management

La partizione di swap è un pezzo del Disco, o potrebbe essere anche un disco intero, sul quale devono essere informazioni che vanno mappate in qualche modo con le pagine in memoria RAM e quindi abbastanza comodo tutto sommato mappare le pagine o i frame con dei blocchi su disco, e vedere questa partizione, anziché come collezione di file organizzati in un certo modo, come un vettore di blocchi, che può essere visto tramite una bitmap dirà quali sono le pagine i blocchi. L'informazione che deve stare su disco è un insieme di blocchi, alcuni usati alcuni liberi, per ognuno dei quali è necessario avere una informazione gestione su che cosa c'è e come viene usato quel blocco.

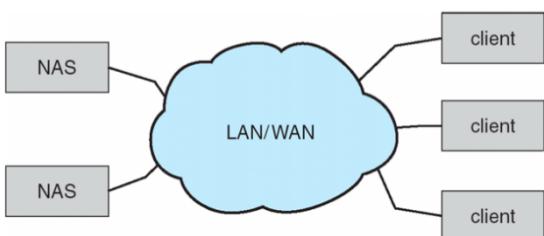
Storage Attachment

La disponibilità di reti di calcolatori, reti locali, reti geografiche, di applicazioni Cloud client-server, o altro fa sì che si possano vedere dei dischi virtuali sia su quello che sono dischi effettivamente fisicamente locali alla piattaforma al computer che si utilizza sia disponibili dalla rete. Ci possibilità di accedere a dispositivi di memoria a tre livelli:

- host-attached: dischi a bordo del computer che si utilizza
- network-attached: forniti dalla rete
- cloud: architettura software più moderna

Storage Array

Uno storage Array potrebbe essere di fatto una collezione di vari tipi di dispositivi, che contribuiscono a fornire un servizio complessivo di disponibilità di immagazzinamento di memoria. Quindi una storage area network è un insieme di storie che sono disponibili in varie modalità.



Struttura RAID

Servono per migliorare l'affidabilità di un disco aggiungendo della ridondanza, e diminuendo la probabilità di perdere tutte le informazioni presenti su un disco, quando si rompesse. È un vettore di dischi che complessivamente si possono vedere come un disco solo. Le strategie utilizzate sono molteplici, il cui obiettivo è quello di aumentare il tempo medio tra due fallimenti, cioè guasti, passerà rispetto a un altro disco molto più tempo prima che statisticamente ci sia un guasto sul disco. Due aspetti importanti quanto tempo passa prima che ci sia un errore, un'una rottura, un guasto sul disco, e quanto tempo ci va a riparare questo guasto, e quanto tempo ci va affinché invece io possa affermare che ho perso delle informazioni, dove perdere informazioni si basa sul fatto che quando c'è semplicemente un guasto singolo non si perde un'informazione perché il Raid garantisce una ridondanza, cioè quella informazione è reperibile da una specie di cosiddette di mirroring, cioè garantisce quelli di quell'informazione ce n'è un'altra copia, si perde un dato sostanzialmente quando si è guastata la prima copia e si guasta la seconda copia mentre non si sa ancora riparata la prima ripristinando i dati. Se in un raid con dischi in mirrored, quindi speculari cioè

due copie della stessa informazione, i due dischi falliscono cioè si guastano indipendentemente l'uno dall'altro allora si perde dato.

Mirrored RAID: 2 disks

$$\begin{aligned} \text{MTTF}_{(1\text{st_fail})} &= \text{MTTF}/2 = 100,000/2 = 50,000 \text{ hours} \\ \text{Prob}_{(2\text{nd_fail_during_repair})} &= \text{MTTR}/\text{MTTF} = 10/100,000 = 10^{-4} \end{aligned}$$

$$\text{Mean time to data loss} = \text{MTTF}_{(1\text{st_fail+2nd_fail_during_repair})}$$

$$\begin{aligned} \text{MTTF}_{(\text{fail+fail_during_repair})} &= \text{MTTF}_{(1\text{st_fail})} / \text{Prob}_{(2\text{nd_fail_during_repair})} \\ &= \text{MTTF}^2/(2*\text{MTTR}) = 10^{10}/(2*10) \\ &= 5*10^8 \text{ hours} \end{aligned}$$

Ci sono svariate tecniche tecniche che servono a gestire l'affidabilità e la possibilità di individuare errori, minimizzare i danni laddove qualcosa si guasti così via.

Altre informazioni

Ad esempio ridondanza nel rappresentare quello che si intende l'operazione che si intende fare o che si fa su disco, il fatto che l'operazione siano sincroni asincroni asincroni e quant'altro per gestire anche l'eventuale perdita di informazioni nel momento di eventuale rottura di qualche cosa, quindi ci sono l'affidabilità e la gestione situazione anomale è qualche cosa di cui occorre tenere in conto quando si pianifica quando si realizza una struttura di immagazzinamento dei dati specialmente quando questi dati sono importanti.

Estensioni

Ci sono estensioni, che oltre a quello che può essere il Raid vanno nell'ottica di rappresentare delle checksum, o aggiunta i dati cosa che in parte già fatta anche in ride e si può arrivare mediante organizzazioni basato su metadati piuttosto che non gruppi di dischi e di filesystem volumi pool di storage o altro di nuovo concetti.

7. Interfaccia File System

File concept

Serve in pratica gestire dei file. Un file è una sequenza di dati, che può essere vista visto a un livello di astrazione molto pragmatico, come uno spazio di indirizzamento contiguo, un file è una sequenza di Byte, può essere associato a un tipo di dato. Possiamo cercare di capire di che tipo di informazioni contiene e potremmo grossolanamente cominciare a suddividere file di dati, da file di istruzioni programmi non si tratta di una classificazione assolutamente rigida, e un file può essere di due o tre tipologie diverse. Il contenuto di un file viene tipicamente definito da chi crea il file chi lo costruisce

Attributi del File

Con attributi di un file, intendiamo le svariate caratteristiche che questo può avere:

- Nome, una stringa che lo identifica
- Identificatore un qualche cosa che sia univoco e non condiviso con qualcun altro e quindi di identificarlo univocamente.

- Tipo: indica qual è l'applicazione, il programma che può leggere o scrivere il file.
- Collocazione: dove sta su un file system
- Dimensione
- Protezione: cosa posso e non posso fare
- Informazioni temporali: data
- utente che è incaricato di agire sul file

Operazioni sui file

Ciò che interessa sono le operazioni che si possono fare su un file. Visto in qualche modo un dato astratto, cioè un collettore di informazioni su cui si possano fare delle operazioni di :

- Creazione
- Cancellazione
- Scrittura
- Lettura
- Riposizionamento nel file
- Troncamento un file: rimuovere l'informazione ad a un certo punto in avanti
- Open: apertura di un file esistente per lettura modifica
- Close: chiusura significa semplicemente smettere di lavorare su un certo file.

Per quanto riguarda il file si intende che ci sia un puntatore all'interno del lettore di locazioni o di indirizzi logici in un che si possa avere una un accesso diretto.

Apertura File

Per agire su un file, è che necessario aprirlo, fare l'operazione di Open, per cui serve una tabella di file aperti. Open file table è una tabella che dovrà gestire sistema operativo per garantire che ci siano informazioni sufficienti in modo da gestire organizzare e altre operazioni su un insieme di file aperti dopo che sono stati aperti e questi file potranno essere chiusi. All'interno di un file aperto ci sarà un puntatore all'ultima posizione scritto letta in un file, perché è opportuno ricordare si sta leggendo o dove si è fatta l'ultima lettura dove si è fatta l'ultima scrittura. Un file può essere condiviso, si può tenere un contatore delle operazioni di Open che sono fatte cioè di quanti stanno accendendo in parallelo a quel file. Si può tenere traccia della collocazione del file su un disco, nonché eventualmente dei diritti di accesso. Nel momento in cui un file potesse essere condiviso tra più utenti, più processi, che vogliono lavorarci contemporaneamente, sarebbe opportuno gestire eventuali sincronizzazioni, piuttosto che non mutua esclusione nell'accesso a un file in modo da garantire la consistenza dei dati.

Apertura File Locking

Il lock o file locking è un'operazione che viene tipicamente supportata e significa in più programmi più processi potrebbero tranquillamente gestire la muta inclusione-esclusione su File, gestendo delle variabili, o

meglio gestendo in modo esplicito semafori, lock, mutex, primitive di sincronizzazione, tali da considerare il file come una risorsa condivisa, e quindi una sezione critica. Essendo i file importanti, quasi tutti i sistemi operativi forniscono dei locali appositi associate ai file, quindi non gestiti in modo esplicito e manuale da dal programmatore ma associato automaticamente un file.

Si possono avere dei lock, e un processo può chiedere di acquisire lock per accedere a un file. Ci sono due tipi lock

- Lock shared: è una dichiarazione di intenti, significa che si dice di voler accedere in lettura un file e si accetta di farlo in condivisione con altri che vogliono leggere il file quindi si mette un like sull'Inter file che non blocca nessun altro intenzionato solo a leggere
- Lock esclusivi: significa Voglio modificare, il file e quindi è blocco eventuali altri accessi concorrenti.

Un'altra caratteristiche ci possono essere dei lock sono obbligatori, specialmente quando si volesse avere un accesso esclusivo per modificare un file, e ci sono dei lock che invece sono semplicemente un consiglio, sono dei lock solo per scrivere esplicitamente "sto accendendo in lettura questo file lo faccio in contemporanea con altri perché tanto nessuno sta modificando il file".

è possibile acquisire e rilasciare un lock esclusivo, oppure un lock shared e lo si può fare su un pezzo di file non sul su tutto il file, e questo è la novità.

Struttura dati interna

Nella forma più semplice un file non ha una struttura, nel senso che può essere visto come una pura sequenza di byte o di word, il significato di ognuno dei Byte va interpretato sia da chi legge sia da chi scrive. Tuttavia, si possono identificare delle strutture a record, nel senso che il file può essere visto come una sequenza o una collezione, organizzazione un po' più complicate che non il singolo Byte Word, è un termine generico.

Ci possono essere file con strutture più complesse, formattate e magari rilocabili che sono ad esempio quelle dei file eseguibili, che contengono sezioni diverse, e contengono delle parti che in qualche modo vanno interpretate o riadattate in modo opportuno quando venissero lette per essere ad esempio caricato in memoria. Queste tre tipologie non sono necessariamente mutua-esclusione, un record a struttura fissa può esser visto come una sequenza variabile. Si possono interpretare o simulare i primi, quelle più semplici pensando di avere dei file che contengono sia informazioni di dato che informazioni di controllo, chi decide qual è la struttura e il tipo di immagazzinamento dei dati in un file, può essere sistema operativo può essere l'applicazione. Ci potrebbe essere un file che dal punto di vista del sistema operativo viene visto come una sequenza di Byte e basta, ma l'applicazione sa che i primi 10 Byte rappresentano una cosa, e gli altri cinque nella presentano un'altra.

File ad accesso sequenziale

Un file accesso sequenziale viene tipicamente caratterizzato da una posizione iniziale, una posizione finale, una posizione corrente, perché le uniche operazioni possibili sono andare avanti dalla posizione corrente, oppure riposizionare tornare all'inizio.

Metodi di accesso

Confrontando accesso sequenziale con accesso diretto, detto anche random-access, si può spesso tentare di rappresentare

■ Sequential Access

```
read next  
write next  
reset  
no read after last write  
(rewrite)
```

■ Direct Access – file is fixed length logical records

```
read n  
write n  
position to n  
read next  
write next  
rewrite n
```

l'operazione di lettura e scrittura sequenziale come, "leggi il prossimo", "scrivi il prossimo" oppure "reset torna all'inizio". Mentre se si vuole rappresentare accesso diretto si dice "leggi l'ennesimo record".

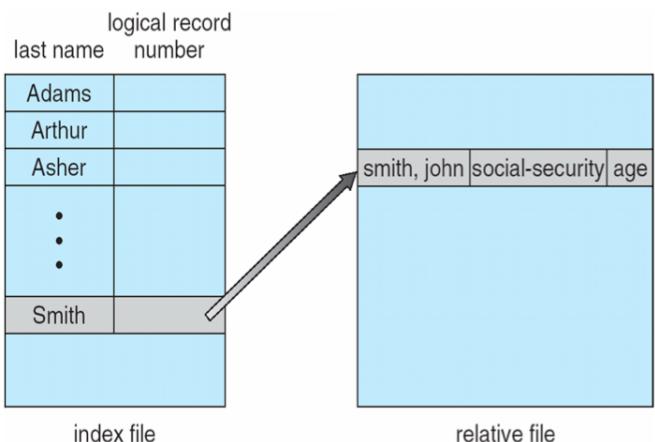
Simulazione di accesso sequenziale in un accesso diretto

Se abbiamo l'operazione di accesso sequenziale reset, che vuol dire riposizionare il current Pointer, il puntatore corrente, a zero e poi Read Next, Right Next, si legge alla posizione current pointer e si incrementa il current in Pointer di 1.

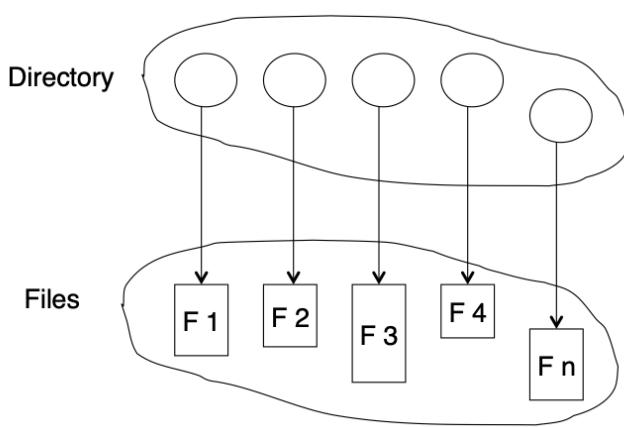
sequential access	implementation for direct access
reset	$cp = 0;$
read next	$read cp;$ $cp = cp + 1;$
write next	$write cp;$ $cp = cp + 1;$

Altri metodi di accesso

Partendo dai due accessi, diretto e sequenziale, si possono realizzare tipologia di accesso misto, o più complicato, basati sulla creazione di un indice per il file, oppure un file che fa da indice, e un file relativo in cui ci sono i dati, quindi nel primo potete fare una ricerca magari anche sequenziale dalla quale in base a In pratica una chiave di ricerca il cognome potete raggiungere il secondo.



Struttura Direttori



Si intende una modalità per poter collezionare più file invece che tutti a livello piatto su un disco in una struttura gerarchica che può essere a più livelli.

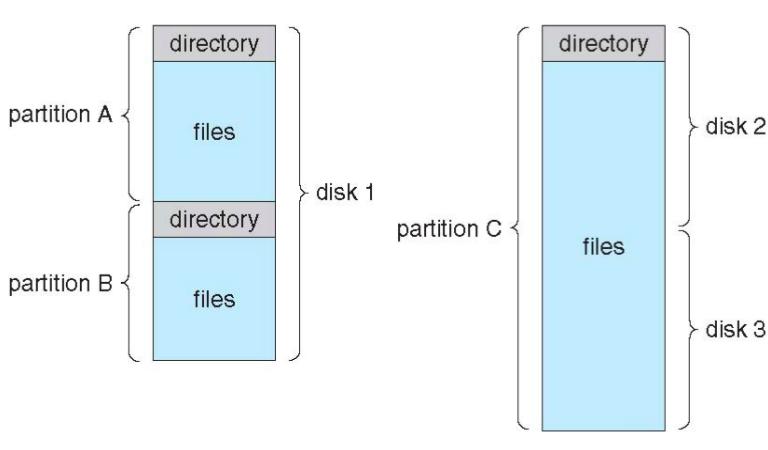
Qui vediamo una rappresentazione di direttore a un solo livello che significa che in un disco voi avete dei nodi che sono le informazioni a livello direttorio e ognuno di questi nodi corrisponde un file struttura piatta.

Struttura Disco

Un disco può essere suddiviso in partizioni, ogni disco o partizione può essere organizzata utilizzate in organizzazione di tipo RAID per avere mirroring per avere striping per avere shadowing o altre tipologie di ridondanza, che servono a migliorare le prestazioni in termini di affidabilità durata o tolleranza eventuali guasti. Una partizione su disco può essere usata in modalità Raw, che vuol dire in pratica andare direttamente ai blocchi poi mettendo in corrispondenza ad esempio con pagine in memoria cioè senza fare System oppure una partizione può essere formattata, per corrispondere se l'è visibile come un filesystem. I volumi che sono un'altra tipologia di un altro termine utilizzato frequentemente per identificare un contenitore di file system, differenza tra un volume è una partizione che in realtà un volume può contenere più partizioni addirittura più dischi. Ogni volume contiene può contenere un file system che caratterizzato tendenzialmente da direttori, tabelle è semplicemente da informazioni che servono a mantenere un file system di tipo generale. Qui vediamo due tipologie di organizzazioni, in cui vedete un disco che è stato diviso in due partizioni in ognuna delle quali ci sono c'è un filesystem con direttori e file direttori sono i contenitori file sono i contenuti o i veri contenitore di informazioni mentre i direttori sono organizzazioni.

Tipi di File System

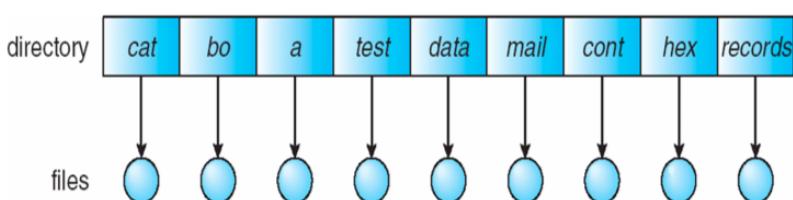
In generale si parla di file system di tipo generale **General purpose**, ma ci sono dei sistemi nei quali si possono identificare file system diversi a seconda del tipo di informazioni che si vogliono memorizzare. L'operazione principali che si fa su una cartella e cercare un file a partire dal nome o da altre informazioni che servono per cercare uno più file. L'albero del directory è l'insieme dei direttori in un file system, è una specie di tabella di ricerca ed è questa operazione che va fatto velocemente, poi una volta che si è localizzato un file, una volta che si deciso dove deve essere creato un file, c'è la creazione di un file, c'è la cancellazione di un file, c'è elenco di file in un directory, c'è la rinominazione di un file oppure c'è un'operazione attraversamento del filesystem, che vuol di' semplicemente enumerare passare su tutti i file e tutte le cartelle tutti i direttori di un filesystem.



Organizzazione File

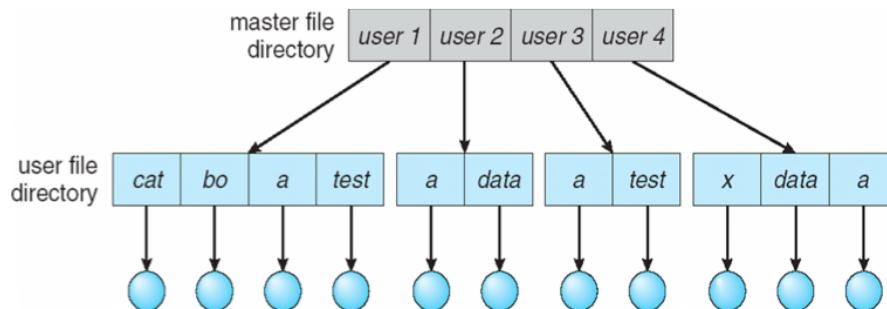
I direttori sono organizzati in modo logico, per poter trovare, cercare, localizzare, rapidamente un file quindi ciò che conta è **l'efficienza**, per fare questo ci si basa su strategie per dare i nomi ai file. È possibile che si dia lo stesso nome a fine diversi, o nomi diversi agli stessi file. Raggruppamenti di file in base a determinate proprietà. Ci sono varie strategie per localizzare accedere, identificare i file.

Direttori a singolo livello



Direttori a doppio livello

è più facile raggruppare i file, ma è necessario il pathname, cioè del nome composto in cui c'è una prima parte una seconda parte terza eventualmente sentiamo più livelli e poi ci sarà il nome del file. Nelle strutture più grandi invece non è così semplice trovare il file.



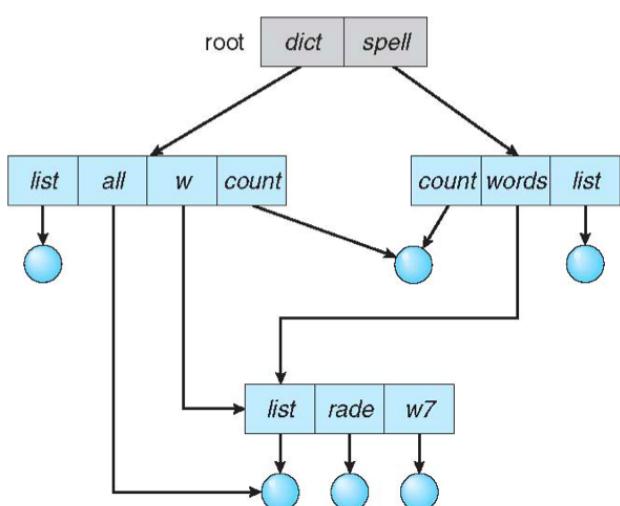
Struttura ad albero

- Si possono realizzare meccanismi di ricerca efficiente;
- si possono raggruppare
- può essere identificato il directory corrente

Nel momento in cui si identifica la posizione di riferimento il current directory, emerge la possibilità di far riferimento, cioè identificare altri direttori o anche direttore corrente ma in generale un altro direttorio e altri file, sia mediante un riferimento assoluto che relativo, il riferimento assoluto partì sempre dalla radice dell'albero il riferimento relativo va fatto rispetto al current directory e poi implicare risalire ridiscendere e cammini all'interno dell'albero.

Grafo Aciclico

Tuttavia, in realtà un albero di direttori non è necessariamente un albero, perché di solito i file system consentono di condividere dei nodi dell'albero e quindi di realizzare in realtà non un albero ma un grafo aciclico, il che significa che è possibile che un file appartenga contemporaneamente a due direttori, due cartelle diverse, così come un sotto directory appartenga a due direttori diversi. Ci sono due possibilità condividere file singoli e condividere direttori, un file singolo è sempre una foglia nell'albero/grafico aciclico, mentre un direttorio non è necessariamente una foglia può essere un nodo intermedio. Ci sono diciamo così strategie diverse, seconda del sistema operativo un modo è il Aliasing che consiste nel dare nomi diversi a un file, cioè in fare in modo che un file sia raggiungibile da due strade diverse; il problema sta nel momento



in cui un file vuole essere cancellato, perché se uno dei due direttori cancella un file, viene cancellato anche dell'altra parte, generando il dangling point, ovvero puntatori a niente, possibili soluzioni a questi puntatori sono: back pointer, cioè un file contiene anche i puntatori indietro, il che significa che quando un file è cancellato si può andare, a passando da un certo direttorio, si può andare aggiornare la cancellazione anche degli altri direttori che lo contengono. Questi puntatori all'indietro potrebbero essere realizzati con:

- un vettore
- daisy-chain una specie di lista
- c'è anche un'altra possibilità che quella di dire un file per essere cancellato deve essere cancellato in modo esplicito da tutti i direttori che lo contengono.

Ci possono essere modi diversi di rappresentare il fatto che un file sia condiviso tra direttori.

General Graph Directory

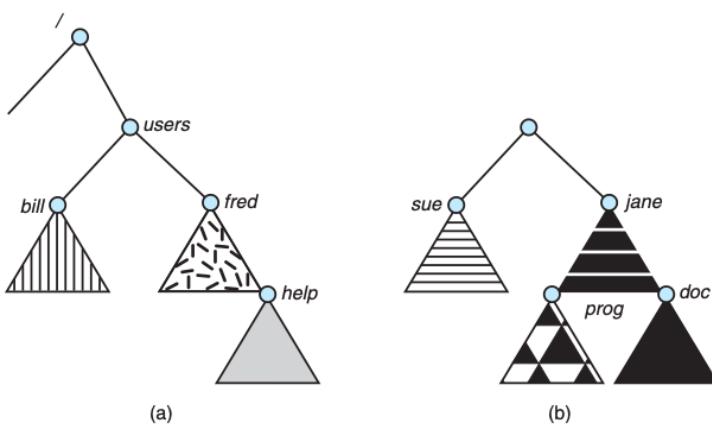
Se non sia visibilità globale sulla topologia dell'albero dei direttori, è possibile innescare dei cicli il che non è ammissibile il grafo deve essere aciclico.

Ci sono alcune strategie per evitarle:

- evitare i cicli per costruzione: non si possono condividere direttori ma si condividono solo foglio solo dei file, e siccome l'unico è l'unica entità che può essere scelta condivisa e il file e il file non può contenere altri file perché non è un direttorio non sono possibili cicli per costruzione.
- Si accettano i cicli ma si fa ogni tanto una passata per individuare i cicli rimuoverli;
- Ogni volta che si aggiunge un direttorio o un link si fa girare un algoritmo di individuazione di eventuali cicli, con l'algoritmo che è di fatto una specie di visita in profondità che parte dal link che sto generato, tuttavia Il problema è che è un algoritmo di individuazione di cicli e potenzialmente lineare nel numero di nodi e archi nel grafo, il che non ho mai detto che sia così efficace.

File system Mounting

L'operazione di mounting di un file system significa che quell'insieme di al di direttori e file diventa visibile agganciato in un sistema, se non si fa Mount, cioè se le informazioni rimangono solo sul disco ma non sono visibili dal kernel e dal sistema operativo è come se il filesystem fosse sganciato, non si tratta solo di una come dire operazione necessaria perché bisogna farla ma semplicemente perché in un sistema di lavorazione si possono essere più dischi più file system e comunque l'albero di direttori, o il grafo aciclico di direttori e file di un disco, va comunque agganciato a un entry point, cioè un punto di ingresso.



Qui abbiamo una rappresentazione in cui la radice, per ora non ne ha mount. A seconda di dove fate l'operazione di Mount potete decidere di vedere sotto una radice del vostro filesystem pezzi diciamo agganciati mounted da uno o più dischi. l'idea è di poter vedere in un unico albero unico grafo aciclico più dischi.

File Sharing

È possibile poi fare file-sharing, nel momento in cui se un sistema supporta la multiutenza più utenti ci possono essere file visibili da più utenti contemporaneamente, momento in cui si fa scegliere in condivisione di file servono meccanismi di protezione per sapere chi può fare che cosa è per definire delle priorità. In un sistema multi-user le protezioni sono spesso determinate da una gerarchia, in cui un utente viene identificato sia da un identificatore di utente sia da un identificatore di gruppo, perché questo serve a suddividere gli utenti secondo una gerarchia che vede l'utente, magari o un al possessore creatore del file quello che può fare tutto su quel file, il gruppo diciamo così una cerchia ristretta di utenti che sono parenti o che sono affini a Lower e gli altri.

Remote file system

Nel momento in cui si lavora a livello di rete Si possono condividere informazioni mediante programmi qua applicazioni quali FTP File Transfer protocol, per trasferimento e copia, file system distribuiti, fanno vedere in modo paritetico direttori e file in un sistema remoto come se fosse internet, applicazioni client-server che permettono di identificare una gerarchia nel senso che il sistema locale può essere client di un server remoto o viceversa, in cui sono chiare delle operazioni non paritetiche tra client-server. ci sono poi applicazioni client-server proprio come file sharing c'è un protocollo per poter condividere file.

Nel momento in cui c'è di mezzo alla rete è importantissimo tener conto di problematiche di errori di fallimenti di cadute del sistema che però hanno una natura molto diversa dagli errori sul file system locale, normalmente un errore su un disco locale significa una rottura, mentre un errore su un file system di rete il più delle volte significa semplicemente che è caduta la rete, ma questa viene ripristinata successivamente, quindi è diverso interpretare un errore di rete che tendenzialmente è temporaneo e quindi va gestito in modo da gestire un'attesa e quindi attendere un ripristino cosa tutt'altro che scontata invece si trattasse di un di un filesystem locale.

Consistenza

Per quanto riguarda quindi la consistenza o la semantica della consistenza di una di un file occorre occorre specificare come più utenti possono contemporaneamente accedere a un file e quindi gestire in pratica delle sincronizzazioni su File. Si tratta di gestire la possibilità di avere accessi multipli contemporaneamente su file in scrittura e ho lettura e quello che conta fondamentalmente è che se più processi user agiscono su un file il puntatore al File condiviso oppure ognuno se leggono ognuno legge e continua a leggere da dove aveva terminato un altro oppure ognuno va per conto suo questa importante per capire quel tipo di accesso che si può fare a file.

Protezione

Normalmente chi crea il file decide che cosa possono fare gli altri sul file, nel senso che può decidere che cosa si può fare sul file e da chi può essere fatto, su che cosa si può fare su un file dev'essere chiaro che le operazioni principali sono file:

- lettura
- scrittura
- append : scrivere ma aggiungendo
- execute

- delete
- list

Per decidere poi chi può fare che cosa ci sono varie possibilità: che però sono gestire i gruppi e le Access control list. Per creare i gruppi di utenti bisogna avere i privilegi bisogna essere System manager normalmente, perché il raggruppamento di utenti gruppi viene creato da chi ha il privilegio di gestire creare gli utenti. Per assegnare poi ha un file a determinati privilegi si usano dei comandi appositi.

8. Implementazione File System

Introduzione

Il file system è sicuramente tra le componenti più importanti e visibili all'utente, che consente di vedere le memorie di massa, i file ed i dispositivi di I/O.

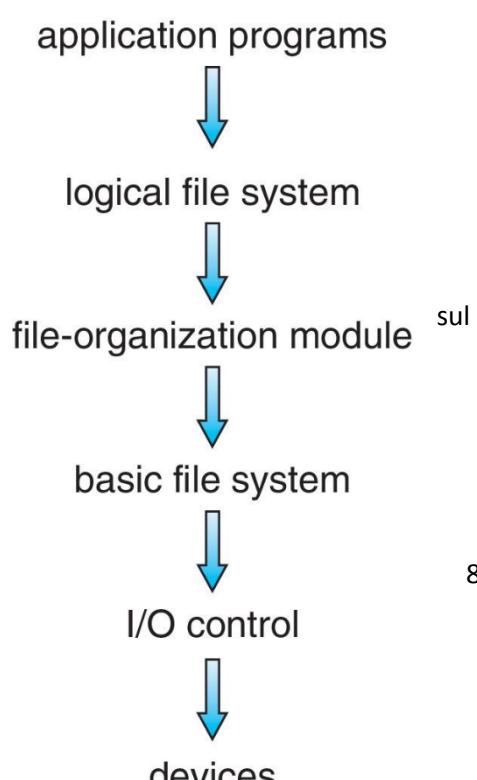
Dal punto di vista fisico, un file è un insieme di blocchi su disco.

Il file system:

- è l'insieme dei file memorizzati su memoria di massa/secondaria non volatile (mentre i dati in memoria RAM/primaria vengono cancellati allo spegnimento del dispositivo)
- fornisce un'interfaccia verso le informazioni salvate nelle memorie di massa
- fornisce una traduzione degli indirizzi da logico a fisico
- deve garantire un accesso efficiente

Il disco lavora su blocchi di dato con accesso diretto, ma con prestazioni ben inferiori rispetto a quelle della RAM. Un file viene trattato dal File Control Block (che contiene le informazioni necessarie per accedere ad un file) come se fosse un ADT (Abstract Data Type). I device driver sono le componenti che gestiscono l'I/O. Sia il SO che il file system sono organizzati a strati: essi sono, dal più astratto/vicino al SW al più concreto/vicino all'HW (sono evidenziati in grassetto quelli del file system):

- application programs (programmi utente)
- **logical file system**
 - gestisce le informazioni più astratte e vicine all'utente
 - comprende i metadati
 - traduce il nome del file nel numero di file
- **file-organization module**
 - capisce che ci sono più blocchi
 - gestisce la collocazione dei file su blocchi logici disco
 - traduce da numero di blocco logico a fisico
- **basic file system**
 - è il livello del file system più vicino all'HW



- riesce già a vedere informazioni indipendenti dal dispositivo fisico e dipendenti dall'organizzazione logica, come “*prendi il blocco 123*” (all'interno di un vettore di blocchi)
- dato che i dischi sono lenti, si gestiscono strategie di buffering e caching, come ad esempio buffer che conservano dati in transito per evitare letture e scritture multiple
- device drivers
 - gestiscono l'accesso ai dispositivi di I/O
 - dal punto di vista del file system, non ricevono mai informazioni del tipo “*leggi il file a.txt al blocco x*”, bensì qualcosa del tipo “*leggi il drive1, cilindro 72, traccia 2, settore 10 e scrivi nella locazione di memoria 1060*” (nel caso di un disco magnetico)

La gestione a strati consente di ridurre la complessità delle operazioni e fornire interfacce chiare. Ci sono vari tipi di file system, diversi per tipo di codifica.

Implementazione e operazioni sul file system

Per effettuare operazioni sul file system, è necessario che il kernel gestisca correttamente alcune strutture dati sia su disco che memoria.

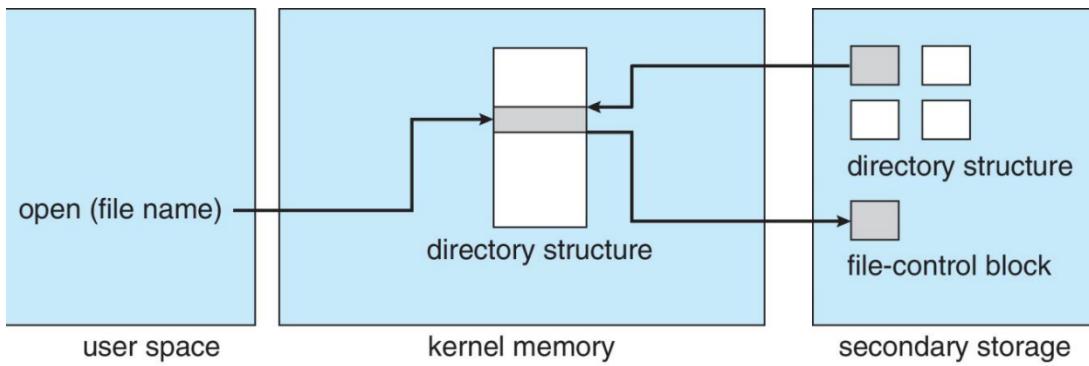
Strutture dati su disco

- **boot control block**: gestisce, all'accensione del calcolatore, le informazioni residenti su disco relative al bootstrap
- **volume control block**: gestisce un superblock e la master file table per memorizzare le informazioni sulle partizioni del disco
- **strutture a direttori**: permettono di collezionare i file e consentirne una ricerca agevole. Possono essere implementate come
 - liste lineari di filenames: ognuno ha puntatori ai blocchi di dato o ai FCB. Sono più semplici da programmare, ma si possono ottenere solo tempi di ricerca lineari
 - tabelle di hash: se fatte con chaining, permettono tempi più rapidi combinando accesso diretto ed eventuali liste per le collisioni, ma non è banale realizzarle
- **File Control Block (FCB)**: serve a memorizzare le informazioni principali di un file (permessi, date, numero di inodes...)

Strutture dati in memoria

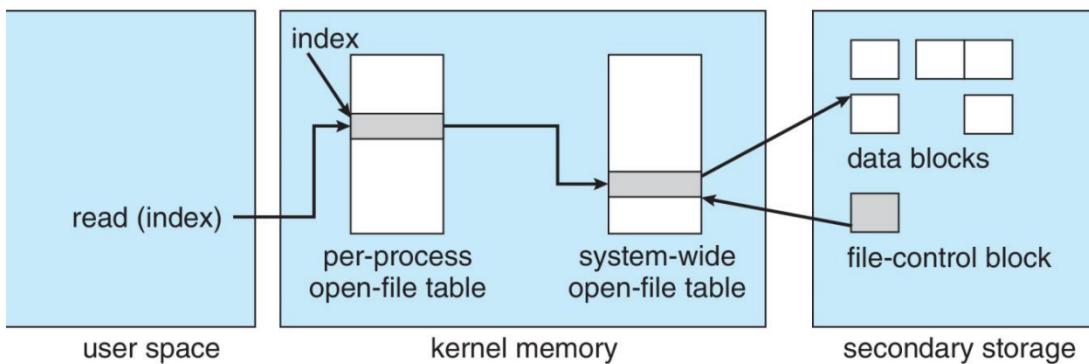
- **Mount table**: replica il contenuto del disco (o di una sua porzione) in memoria quando il SO monta il disco
- **System-wide open-file table**: tiene traccia di tutti i file aperti (eventualmente da più utenti) sul file system
- **Per-process open-file table**: è legata al singolo file

Vediamo come viene effettuata la open di un file (la close è analoga).



La open è fatta una tantum ed è una sorta di inizializzazione. A destra sono mostrati la struttura a direttori ed il FCB. La open parte dal nome del file e lo cerca nella copia locale della struttura a direttori nel kernel: o è già presente o è assente, localizza il FCB su disco. Il risultato sarà anche quello di copiare il FCB nella system-wide open-file table.

Vediamo come viene effettuata la read di un file (la write è analoga).



La read è fatta tante volte e su un risultato prodotto dalla open. La read parte dal numero (= descriptor) del file, lo cerca nella per-process open-file table, che punta alla system-wide open-file table, che contiene la copia del FCB. Il SO dovrà cercare di convertire il puntatore logico interno al file in un blocco su disco, reperire il blocco su disco ed effettuare le operazioni desiderate.

Nella per-process open-file table, ogni processo vede il proprio file con il rispettivo file descriptor:

- fd = 0: stdin
- fd = 1: stdout
- fd = 2: stderr
- fd > 2: altri file

Inoltre, La per-process open-file table e la system-wide open-file table devono consentire a due processi diversi di poter avere puntatori diversi allo stesso file, oppure di condividere puntatori. Se un file viene aperto contemporaneamente da 2 processi, ci sarà una sola copia del FCB nella system-wide open-file table, mentre ogni processo avrà una entry nella propria per-process open-file table.

Concettualmente, i direttori sono tabelle di simboli speciali, dato che si cercano i pezzi che compongono il filename all'interno di un grafo aciclico.

Metodi di allocazione

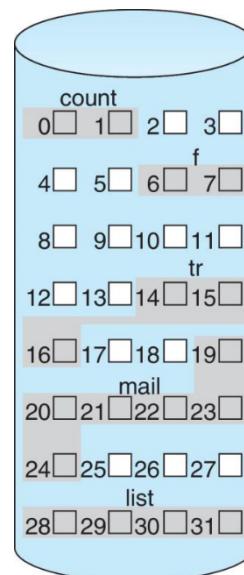
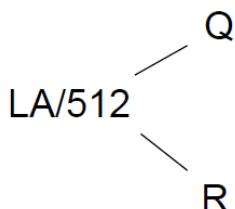
Vediamo ora come vengono allocati i file su disco.

Allocazione contigua

Ogni file occupa un insieme contiguo di blocchi su disco, a seconda della propria dimensione. È uno schema molto semplice, ma presenta il problema della frammentazione.

directory è una tabella contenente 5 file, tra count, che inizia al blocco 0 ed occupa 2 blocchi contigui.

Questo schema serve a capire a quale blocco appartiene il byte x all'interno di un file composto più blocchi. Ho un Logic Address e voglio tradurlo blocco logico.



directory		
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

da
in

Nella divisione:

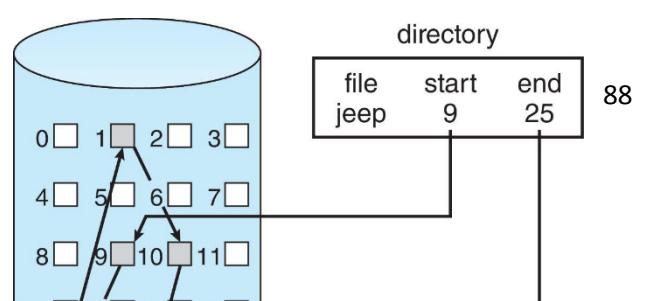
- 512 byte/word/informazioni (l'unità di misura non è specificata) è la dimensione del blocco
- Q è il quoziente e rappresenta il numero d'ordine del blocco (Q parte da 0)
- R è il resto e rappresenta il displacement all'interno del blocco

Molti file system moderni adottano uno schema di allocazione contigua extent-based (che assomiglia leggermente al concetto di slab allocator), secondo cui un extent è una parte contigua di blocchi su disco e un file consiste di più extent.

Allocazione a lista linkata(concatenata)

Con i file prevale l'utilizzo di strutture che garantiscono l'accesso sequenziale. Un file è una lista di blocchi che permette di allocare ogni blocco in un punto diverso del disco, senza problemi di contiguità e quindi di frammentazione esterna.

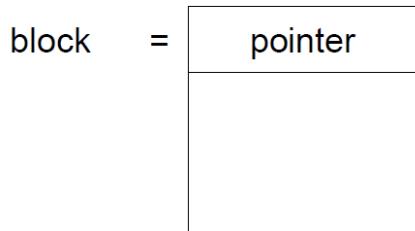
Mentre la memoria di un processo non può essere vista come una lista, bensì come un vettore logico rimappato su delle pagine in RAM, un file può essere visto come lista di blocchi, in cui ogni blocco punta al successivo, perché viene letto sequenzialmente. Lo spazio libero è gestito tramite una free-list, e si potrebbero migliorare le prestazioni tramite clustering dei blocchi.



Qui directory contiene, per ogni file, il numero del blocco di inizio (da cui si raggiungono quelli intermedi) e quello di fine.

Esistono dei problemi di affidabilità: se si danneggia un blocco, si perdono tutti quelli successivi (è come una caccia al tesoro: se a un certo punto perdo un indizio, non sono più in grado di trovare quelli successivi). Siccome le liste sono strutture dati lineari, dobbiamo scandire tutti i blocchi che precedono quello interessato.

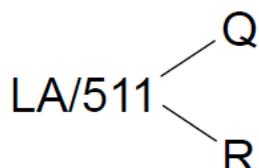
Ogni blocco contiene un puntatore (in genere di 32-64 bit) al blocco successivo e i dati.



In questo caso, se si vogliono sapere Q e R bisogna dividere per 511 anziché per 512, dato che 1 è riservato al puntatore.

Se il puntatore

- è all'inizio del blocco, displacement = R+1
- se è alla fine del blocco, displacement = R



FAT

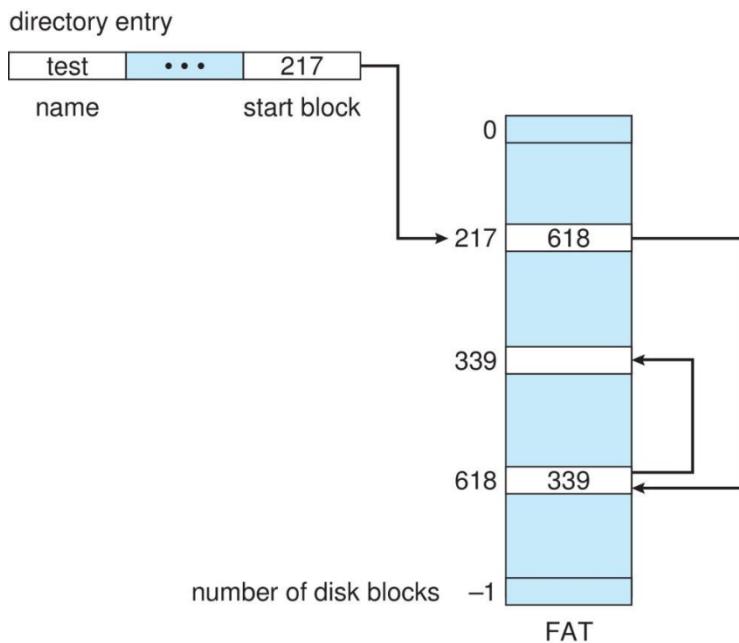
La FAT (File Allocation Table) è un particolare file system ad allocazione a lista linkata molto utilizzato nei sistemi Microsoft Windows. Risponde al problema principale dell'affidabilità delle liste linkate (cioè la perdita di un blocco e di quelli successivi).

La motivazione delle FAT è questa: se i puntatori sono critici (cioè, se perdo un blocco, perdo non solo le informazioni del blocco di dato ma anche i puntatori ai blocchi successivi), allora perché non memorizzarli in un vettore/lista (la FAT, appunto) e fare in modo che i blocchi contengano solo dati?

La FAT è da qualche parte nel disco.

- Se ho 1M blocchi da 1KB, occupo 1GB di blocchi dato
- La FAT avrà un puntatore per ognuno di questi blocchi, e quindi ne avrà 1M

- Se un puntatore è grande 4B, la FAT sarà grande 4MB (circa 3 ordini di grandezza in meno rispetto al disco)



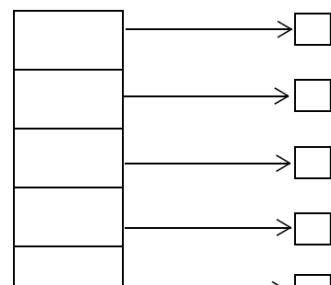
Facendo riferimento a **directory**, **test** comincia al blocco 217 (il cui contenuto è nel vettore dei blocchi di dato). Se si vuole leggere il contenuto di un altro blocco, si procede nella lista dei puntatori: il puntatore (del blocco) 217 ha come successore il puntatore 618, che ha come successore il 339.

Le FAT sono più affidabili delle altre tabelle a lista linkata: dato che i puntatori si trovano nella FAT, se si rompe un blocco di dato non si perdono quelli successivi, bensì solo i dati relativi al blocco danneggiato.

Allocazione indicizzata

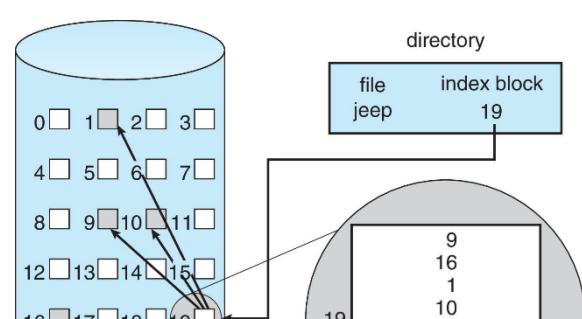
Rappresenta un'alternativa più veloce rispetto alle liste, dato che cercano di usare delle tabelle ad accesso diretto (cioè dei vettori).

Ogni file ha un **index block** (o più d'uno), e gli index blocks sono organizzati nell'**index table** mostrata in figura: a sinistra vi sono i puntatori a blocchi, e a destra i blocchi puntati.



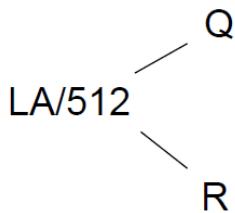
index table

L'**index block** di **jeep** è un'informazione aggiuntiva che racchiude, in un unico blocco, tutti i



puntatori ai blocchi di dato occupati dal file in modo da garantire accesso diretto.

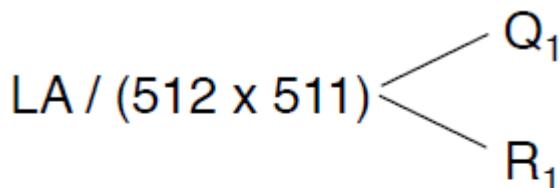
Il calcolo di Q ed R è uguale a quello visto nell'allocazione contigua:



Nel caso di `jeep`, sono stati utilizzati 5 degli 8 indici disponibili (quelli liberi hanno valore -1). E se il numero di indici contenuti in un blocco non bastasse per il file? Sono disponibili 3 soluzioni:

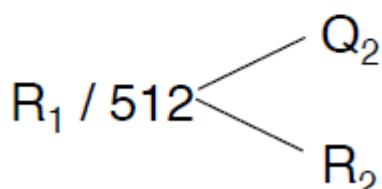
- **lista di blocchi di indici:** pago comunque un po' di ricerca sequenziale
- **tabella di blocchi di indici a doppio/triplo livello:** ogni blocco nella tabella di indici contiene 511 indici + 1 puntatore alla prossima tabella di indici.

Bisogna prima sapere a quale blocco di indici si fa riferimento, cioè



dove

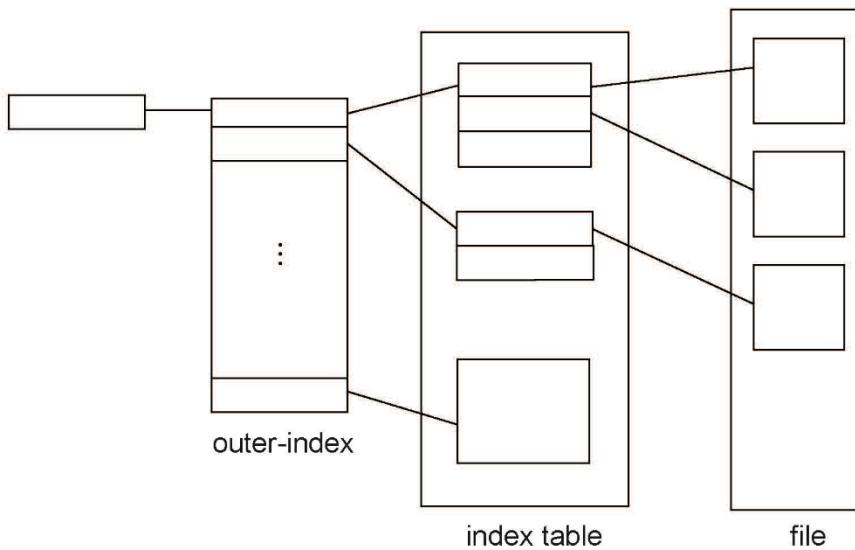
- 512 = dimensione del blocco di dato
- 511 = dimensione del blocco di indice
- Q1 = numero d'ordine della tabella di indici
- R1 viene usato come segue:



dove

- Q2 = displacement all'interno della tabella di indici

- R2 = displacement all'interno del blocco del file



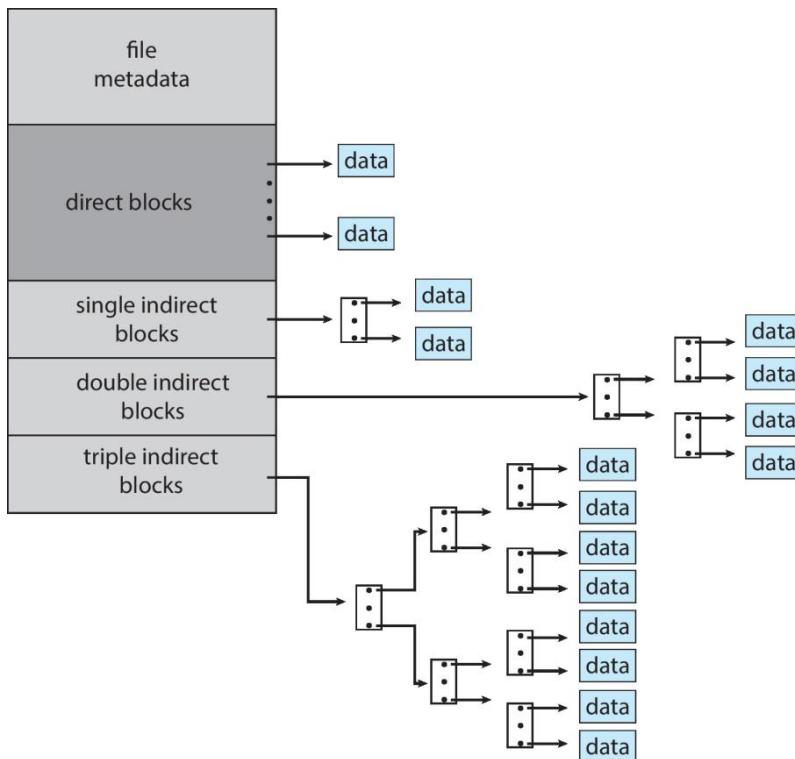
Se si vuole assolutamente garantire l'accesso diretto, si organizzano i blocchi indice a 2 livelli. L'unica vera differenza è che quando devo determinare il blocco indici non ho bisogno di scandire una lista, ma ci arrivo direttamente.

Analogamente alla Page Table a 2 livelli, si ha

- una tabella di indici outer/di 1° livello
 - una tabella di indici inner/di 2° livello
 - il file diviso in blocchi
-
- **inode:** un approccio misto, adottato nei sistemi UFS e Unix/Linux, che consiste in una tabella di indici a livello multiplo sbilanciata.

La tabella di indici gerarchica è di fatto un albero (lo si nota meglio se viene ruotata di 90° in senso orario) avente come radice il FCB, qui chiamato inode. Il concetto alla base del funzionamento è:

- se il file è piccolo, non c'è bisogno di molti livelli di indici
- se il file è grande, c'è bisogno di molti livelli di indici



Oltre ai metadati dei file sono presenti i puntatori ad alcuni blocchi diretti (10 nell'inode classico): i primi 10 blocchi di dato sono accessibili direttamente perché i puntatori a questi blocchi sono nel FCB, e se il file è piccolo (cioè ha al massimo 10 blocchi) non serve niente del resto.

Se il file è più grande di 10 blocchi, i primi 10 saranno comunque ad accesso diretto, e quello dall'11° in poi saranno contenuti in un livello di blocchi di indici di tipo indiretto singolo.

Supponiamo ora che:

- un blocco di indici contenga 512 puntatori
- sono già stati occupati 522 blocchi del file system, di cui
 - 10 diretti
 - 512 indiretti singoli

Se il file è grande almeno 523 blocchi, ci sarà un ulteriore puntatore a un blocco di indice (detto indiretto doppio) che punta ad una tabella di indici che però è gerarchica:

- il 1° livello è composto da blocchi di indici indiretti (indici indiretti di 1° livello)
- il 2° livello è composto da blocchi di indici indiretti (indici indiretti di 2° livello)
- il 3° livello è composto da blocchi di dato

Se non basta ancora, si impiega il blocco indice indiretto triplo.

Globalmente, la capienza dei blocchi di dato indirizzabili è data dalla somma dei vari contributi, ma è chiaro che quello fornito dal blocco di indice indiretto triplo è dominante: la seguente tabella fa riferimento al caso di un inode classico con 13 puntatori (10 diretti, 1 indiretto singolo, 1 doppio e 1 triplo) e di blocchi indice contenente 512 puntatori.

Tipo di puntatore	N_{max} di blocchi dato indirizzabili
diretto	10
indiretto singolo	512
indiretto doppio	512^2
indiretto triplo	512^3

Performance

Non è facile determinare quale sia il meccanismo migliore, perché dipende dal tipo di accesso desiderato per i file.

Metodo di allocazione	Tipo di accesso	
	diretto	sequenziale
contigua	✓	✓
a lista linkata		✓
indicizzata	✓	

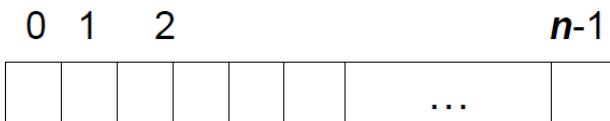
Si potrebbe, alla creazione del file, chiedere all'utente il tipo di accesso desiderato. L'allocazione indicizzata è più complessa e potrebbe richiedere 2 letture di blocchi di indice prima di quella del blocco di dato (se è a 2 livelli, come nella Page Table).

Spesso si scelgono soluzioni intermedie basate su clustering. Con le SSD (NVM) potrebbero esserci altre strategie. Se ci si convince del fatto che ci sono gap di prestazioni enormi tra i dischi e le istruzioni in memoria, si potrebbe sfruttare questa conoscenza scrivendo SW che sfrutti parallelismo/pipelining per ridurre l'I/O.

Gestione dello spazio libero

Vengono presentate di seguito alcune strategie per tracciare i blocchi disponibili:

- **free-list**
- **bitmap**



$$\text{bit}[I] = \begin{cases} 1 \Rightarrow \text{block}[I] \text{ free} \\ 0 \Rightarrow \text{block}[I] \text{ occupied} \end{cases}$$

La bitmap richiede spazio aggiuntivo e, se si fa corrispondere un solo bit anziché un byte ad ogni blocco, se ne può risparmiare ulteriormente ed usare gli algoritmi in maniera efficiente (in realtà non si possono avere vettore di bit, bensì dei vettori di byte/interi/unsigned con all'interno dei bit). Si veda il seguente esempio.

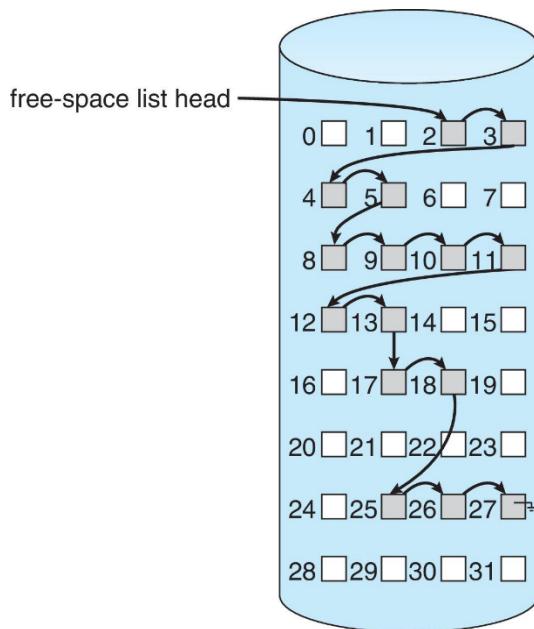
block size = 4KB = 2^{12} bytes

disk size = 2^{40} bytes (1 terabyte)

$n = 2^{40}/2^{12} = 2^{28}$ bits (or 32MB)

if clusters of 4 blocks -> 8MB of memory

- **free list linkata:** non ha bisogno dell'overhead della bitmap: basta il puntatore al primo ed i blocchi possono essere linkati tra di loro. Se si vuole realizzare una FAT, la free list linkata può essere implementata direttamente dentro la FAT, che è già un vettore di puntatori.



- **raggruppamento:** si può mantenere, in un blocco libero, l'insieme degli $n-1$ blocchi liberi successivi

- **conteggio:** si possono rappresentare delle free-list che siano un po' più efficienti nella ricerca di un intervallo di blocchi liberi (come viene fatto in OS161) tenendo l'indirizzo del primo blocco libero ed il numero di blocchi liberi successivi
- **Space Maps:** usano i metadati

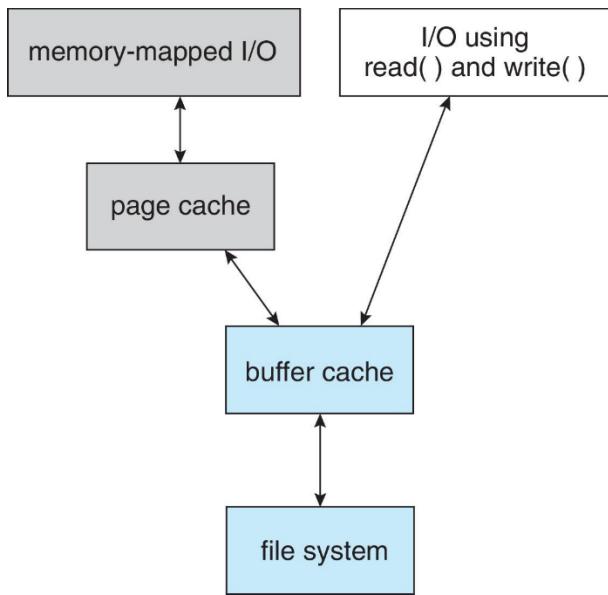
Consideriamo il problema della sovrascrittura:

- negli HDD non sussiste, in quanto basta posizionarsi sul blocco desiderato e scriverci
- negli SSD ciò non si può fare: bisogna prima cancellare il blocco e poi scriverci, e questo può essere problematico per le politiche di gestione dei blocchi liberi.

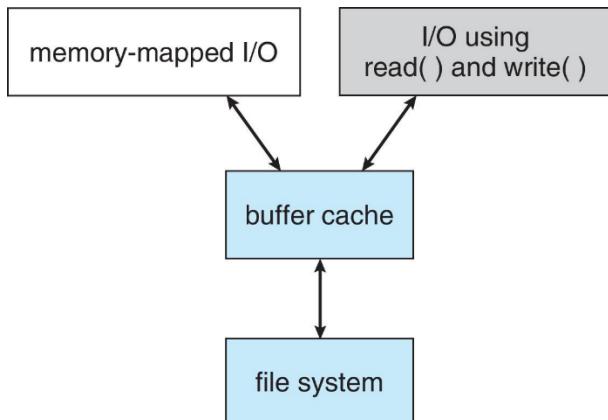
Efficienza

I principali fattori che incidono sull'efficienza e sulle prestazioni di un file system sono:

- **strategia di allocazione** (contigua, a lista linkata, indicizzata) e algoritmi scelti per i direttori (qui non trattati)
- **come vengono salvati i dati** all'interno delle tabelle per i direttori
- **come vengono gestiti i blocchi liberi** (bitmap, free linked-list...)
- gestione di **strutture dati a dimensione fissa o variabili**
- **bufferizzazione:** mantenere dati e metadati vicini può servire, dato che l'I/O è costoso. Buffer cache: si tiene una parte della RAM per utilizzarla efficacemente in termini di tempo
- **I/O sincrono/asincrono**
 - I/O sincrono: faccio partire la lettura/scrittura e aspetto che sia terminata per andare avanti
 - I/O asincrono: faccio partire la lettura/scrittura ma nel mentre faccio altro; può velocizzare le operazioni
- **free-behind e read-ahead**
 - read-ahead: quando si legge un blocco da disco si potrebbe già in anticipo leggere il successivo perché potrebbe servire dopo, sfruttando il fatto che la lettura sequenziale di più blocchi vicini costa quanto la lettura di un blocco solo
 - free-behind: è un'estremizzazione: quando passo dal blocco 5 al 6, si cerca di anticipare i tempi liberando lo spazio occupato dal blocco 5 per far posto al blocco 6
- il fatto che normalmente **le letture sono più lente delle scritture**
- impiego di **page cache**: è una cache utilizzata per il memory-mapped I/O (in cui si mappano i blocchi su disco in pagine di memoria). Avendo stabilito una corrispondenza blocco su disco - pagina in memoria, il modulo che gestisce il memory mapping crea una cache per sé dove da un lato c'è il blocco su disco e dall'altra la pagina in memoria.



Anche il disco ha il buffer cache per velocizzare la corrispondenza tra lettura/scrittura su disco e risultati che vanno in memoria. Se si realizzano 2 moduli indipendenti, uno che gestisce l'I/O su disco con buffer cache e l'altro il memory-mapping con un'altra cache, allora abbiamo un doppio livello di caching nella catena di sinistra, che però è inefficiente. Dunque, nei sistemi Unix-like si realizza lo Unified Buffer Cache per unificare i 2 moduli:



Ripristino dei dati

Per evitare perdite di dati si usano strategie come il checking della consistenza dei dischi per rivelare errori, e la gestione di backup e restoring: l'unico vero modo per non perdere informazioni è quello di avere dati duplicati.

Quando si rompe qualcosa, cosa si fa? Se c'è un crash e mi limito a ripristinare l'ultimo backup, rischio di perdere le ultime informazioni che ho gestito (come quando il computer si pianta mentre si sta lavorando su Word). Con un file system log structured (e.g. NTFS) si ha il concetto di transazione: prima di fare qualsiasi cosa, si dichiara e si salva l'intenzione di fare quella cosa, e solo una volta che quella cosa è committed si

può distruggere l'intenzione di fare. Se non si riesce a fare quella cosa, l'intenzione è salvata e potrà essere riusata in futuro.

9. OS161 – UserProcess

Running a user process

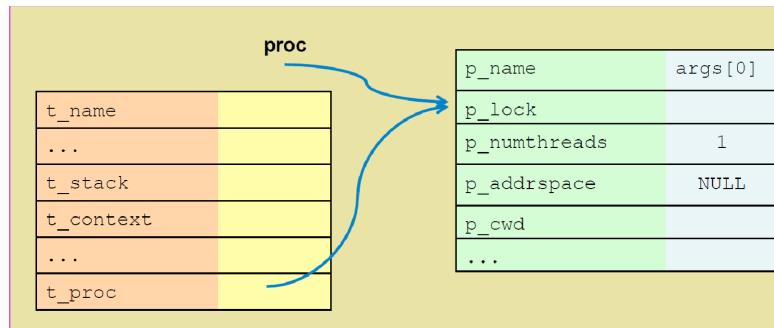
Affrontiamo nel contesto OS161 una parte conclusiva che coinvolge molti aspetti che vanno dagli aspetti di gestione della memoria alla sincronizzazione e gestione dei processi, fino al file system. Il motivo per cui viene presentato in questo momento, effettivamente, è legato al fatto che c'è una parte significativa di questo insieme di lucidi che riguarda il file system, almeno la parte conclusiva, ed è premessa del Laboratorio 5.

Chiamarlo *User Process* mette l'enfasi su cosa significa caricare un programma nell'address space: significa sia creare un address space, ma anche copiare il programma da disco (da file system) all'address space. Quando il kernel crea un processo per far girare un programma in OS161 deve preventivamente creare l'address space. Questa era stata una parte che per alcuni di voi era stata non chiara nei primi due laboratori, in cui si sono fatte determinate operazioni sull'address space senza che fosse stato spiegato a fondo. Il problema è che essenzialmente per capire l'address space bisogna leggere il file elf, cioè leggere il formato di un file. Infatti, il codice e i dati di un programma sono in un file eseguibile che è creato quando il programma è compilato e linkato (quando si fa *bmake* e *bmake install*). Ad esempio, quando fate *bmake* e *bmake install* di *testbin palin* create l'eseguibile di palin, il file elf contiene un programma user. In OS161 i file sono elf perché questo è il formato tipico del mondo UNIX Linux. Per creare un programma utente, senza pensare alle shell, per ora avete la funzione *common_prog* attivata tramite il menu che, di fatto, attiva la *runprogram* (cioè riesce a far girare un programma utente). Un'alternativa per creare un programma user (indicata nel Laboratorio 4) è la system call *fork*, che di fatto consente a un processo user di generare un altro processo user che svolga lo stesso eseguibile, lo stesso programma, quindi senza cambiare eseguibile. **PROTOTIPO:** *pid fork(void)*. La system call che permette a un processo user di generare un altro processo user che esegua un altro programma è la *execv*, che riceve il nome di un programma da eseguire, con gli argomenti. È quella che permette a un processo user di fare quello che fa la *common_prog* nel menù quando fate *testbin palin*, cioè permetterebbe a un processo utente di creare un altro processo utente che esegue un altro programma.

runprogram

Quello che rivediamo ora è la catena per arrivare alla *runprogram*. La *runprogram* è la funzione che internamente a OS161 permette al kernel di creare un programma utente. In parte sono concetti che abbiamo già visto e sono concetti che avete affrontato quanto avete svolto il laboratorio intitolato *waitpid*. La *common_prog* è la funzione attivata in qualche modo dal menu che attiva la *proc_create_runprogram* che alloca, crea e inizializza la struttura dati per un processo ma il processo non è ancora finito. Cioè la *proc_create_runprogram* genera una *struct proc* con un po' di informazioni dentro. Quella che conta è la *thread_fork* che è quella che genererà un thread incaricato di creare il thread.

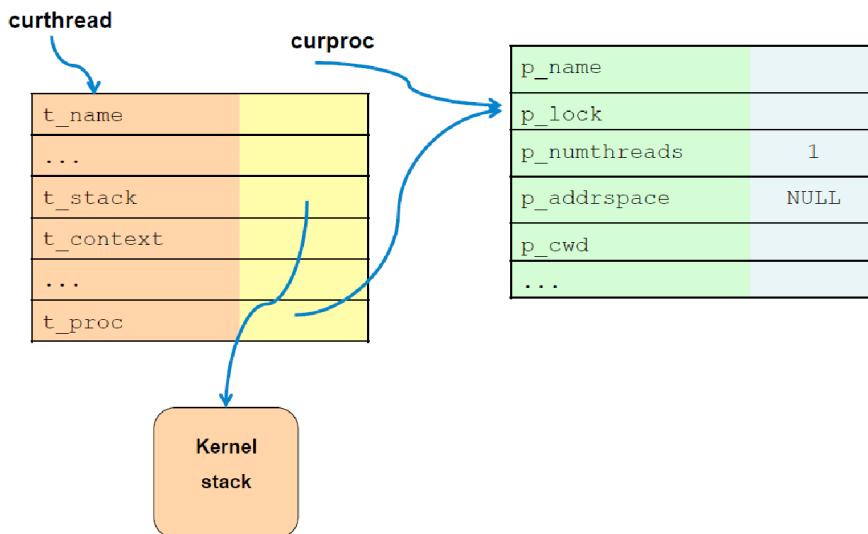
Figura 1 Struct Thread che punta al processo. Quel processo eseguirà la cmd_progthread



Perciò c'è una creazione di una *struct proc* e una creazione di una *struct thread* che però è associata alla creazione del thread stesso.

La *thread fork* in realtà è quella che deve sganciare la funzione *cmd_progthread* su un altro thread. Dal lato di chi esegue la *thread fork* abbiamo un puntatore *proc* che punta a una *struct process* e genereremo un nuovo thread internamente che avrà un suo kernel stack ma questo sarà un altro thread e un altro processo (perché noi siamo nell'ambito del kernel). Se guardiamo la stessa cosa dal lato del thread appena creato (*cmd_progthread*) sia il processo che il thread diventano *curproc* e *curthread*, quindi se entriamo nella *cmd_progthread* (la funzione che viene attivata in un nuovo thread) ci troviamo in kernel con una struttura dati *curproc* che non è ancora finita, cioè è un processo con semplicemente una *struct* vuota, e un thread che punta a quella *struct* e ha il suo kernel stack, quindi un thread di kernel.

Figura 2 cmd_progthread



La *runprogram*, che viene chiamata dalla *cmd_progthread*, è quella che ha il compito di fabbricare il processo user e di farlo partire. In particolare, ha il compito di creare la parte di memoria user, cioè Code / Data / User Stack. Come vedete il processo ha il puntatore all'address space, che avete già usato più volte quando avete chiamato la *as_destroy* o altre funzioni, però di fatto quell'address space, per ora, (*dumbvm*) è fatto con allocazione contigua. **Per il progetto di PAGING** occorre modificare questo address page facendo in modo che invece di puntare a tre partizioni contigue, lo stack si supporti la tabella delle pagine di un processo. Per ora questa è la versione *dumbvm* in cui voi avete lavorato a modificare il supporto di questo address space soprattutto supportando il riciclo, la liberazione di Code / Data / User Stack. Questo è il

runprogram che ha solo creato l'address space, non si vede ancora come viene attivato il processo. La differenza tra *cmd_proghread* e *runprogram* si percepisce dal fatto che *runprogram* svolge questo incarico, ma soprattutto chiamerà il caricamento del file ELF in codice e dati. Il file ELF conterrà le informazioni da piazzare dentro il segmento di codice e il segmento di dati dell'address space.

Versione ridotta di runprogram <pre>runprogram(char *progname) { struct addrspace *as; struct vnode *v; vaddr_t entrypoint, stackptr; int result; /* Open the file. */ result = vfs_open(progname, O_RDONLY, 0, &v); ... /* Create a new address space. */ as = as_create(); ... /* Switch to it and activate it. */ proc_setas(as); as_activate();</pre>	<pre>/* Load the executable.*/ result = load_elf(v, &entrypoint); ... /* Done with the file now.*/ vfs_close(v); /* Define the user stack in the address space */ result = as_define_stack(as, &stackptr); ... /* Warp to user mode.*/ enter_new_process(0/*argc*/, NULL/*userspace addr of argv*/, NULL /*userspace addr of environment*/, stackptr, entrypoint); /* enter_new_process does not return.*/ panic("enter_new_process returned\n"); return EINVAL; }</pre>
---	--

In questa versione si fa una *vfs_open*. Diciamo subito che in ambito OS161, seguendo una strategia UNIX Linux di astrazione o di implementazione di file system a livelli il file system a livello più alto è realizzato da *vfs* (virtual file system) sotto al quale stanno i file system veri: c'è un layer indipendente dal file system scelto. Come vedremo in OS161 sono supportati due file system che sono *EMU file system* e *SIMPLE file system*. L'*EMU* sarà quello che non fa niente e si appoggia al file system della macchina host. *SIMPLE file system* è invece un vero file system molto semplice. *Vfs_open* significa dato il nome dell'eseguibile fai la open, ovvero vai a cercarlo nel file system e ritornami un risultato: booleano intero per dire successo o insuccesso. Il vero risultato by reference però va in *&v* che è una *struct vnode*. Il *vnode* assomiglia all' inode e comunque è l'equivalente del file control block a livello generale visto nel capitolo 14 del Silberschatz.

La prima cosa che fa *runprogram* è aprire il file (equivalente della *open*) e ritornare un puntatore al file control block detto *vnode* in *v*. Come vedremo questo *v* sarà passato come primo parametro alla *load_elf* che farà effettivamente la lettura. Perciò: *vfs_open* / creazione dell'address space / Messa a punto dell'address space (*proc_setas(as)*, *as_activate()*). I dettagli della messa a punto preferisco evitarli, i commenti sono molto significativi e potete vedere di ogni funzione che cosa realizza; per esempio *proc_setas(as)* è soltanto un'assegnazione al campo corretto nella *struct proc*.

Una volta che abbiamo attivato l'address space, cioè facciamo in modo che il processo riesca a fare traduzione in indirizzi logici/fisici. Da questo punto in avanti si può supporre che il processo possa sfruttare lo spazio di indirizzamento e lavorarci sopra perché è stato attivato, o meglio, predisposto. Nella *Load_elf* vedremo i dettagli perché in parte l'utilizzo del address space va completato dalla lettura del file eseguibile, perché è lì che ce scritto quanto sono grandi e dove devono stare i segmenti di codice e di dato.

Il risultato della *Load_elf* sarà: ho fatto tutto, cioè ho caricato in memoria fisica e corrispondente memoria logica tutto ciò che serve per far partire il processo e ti ritorno l'*entry point*, si chiude il file (*fclose* partendo dal file control block) si aggiunge all'address space lo stack, perché non era necessario che fosse predisposto

dalla *load_elf* e poi si fa la *enter_new_process* cioè si fa partire il processo dall'entry point, quest'ultimo è stato definito dalla *load_elf*. È ovvio che ora dobbiamo vedere che cosa realizza la *load_elf* ed era questo che abbiamo lasciato in sospeso quando avevamo analizzato la catena di partenza di un processo alcune lezioni fa.

Per prima cosa vediamo che la *Load_elf* riceve un puntatore a *vnode*, a un file control block, e ritornerà un entry point. Quello che potete vedere nel momento in cui lavorate nella *load_elf* è che c'è una prima parte in cui prende l'address space (*proc_getas()*), non c'è bisogno di ricevere l'address space come parametro perché l'address space è già stato agganciato al processo. Quindi la *proc_getas* prenderà il puntatore alla address space dalla *struct curproc*. A questo punto ci sarà nel *Load_elf* la classica doppia lettura di un file, cioè una prima lettura, non completa, per prendere le misure, e una seconda lettura per leggere i dati. Non sarà la prima volta che vi è successo di dover allocare dinamicamente qualcosa in funzione dei dati in un file e di fare prima un percorso sul file per prendere le misure per sapere di quanta memoria c'è bisogno per caricare i dati, fare l'allocazione e poi leggere effettivamente i dati. Il file ELF, tra i file di cui abbiamo discusso nel capitolo 13 (formati diversi del file / sequenza di byte / record a lunghezza fissa, variabile) è qualcosa che assomiglia molto di più a un file con indici o comunque ad un file in cui c'è un intestazione, un header e ci sono dei sotto-header e ci sono delle parti del file che descrivono cosa sono altre parti del file.

Struct vnode

Quindi qui per prima cosa c'è una parte su come si fa a leggere un file: si deve arrivare ad una Virtual Operation Read (*VOP_READ*). Per capire questo vediamo che cosa è una *struct vnode*: è un file control block di tipo virtual file system, cioè contiene un *reference count*, uno *spinlock* per accedere in mutua esclusione al reference count, contiene un puntatore al descrittore del file system e un puntatore ad una struct che contiene puntatori a funzioni in quel file system. Non posso entrare molto nei dettagli: vi dico semplicemente che la struct file system è un interfaccia che punta al file system effettivo che verrà usato che può essere *EMUfs_ops* (file system EMULato che si appoggia al file system della macchina host, ovvero scrivere *testbin/palin* significa che state usando la cartella *testbin* sulla macchina host e il file *palin* sulla macchina host) o *SIMPLEfs_ops* (se voi controllate in *sys161.conf* vedete come si possono specificare dei dischi in questo formato di cui si può fare eventualmente mount, sono dei dischi interni a OS161). In realtà non si punta a entrambi ma ad una di queste due struct (*emufs_ops* o *sfs_ops*). Che cosa sono queste *ops*? Sono puntatori a funzione: se voi date un'occhiata mediante il codice browsable a queste struct vedrete che si tratta sempre di puntatori a funzione. Ora quel che mi preme dirvi è che quando la *load_elf* fa *VOP_READ*, in realtà questa *VOP_READ* è una macro che fa riferimento al campo *READ* della struct puntata dal campo *vn_ops* della struct *vn* e questa *VOP_READ* sarà un puntatore ad una funzione che corrisponderà o a *emufs_read* o *sfs_read*. Nel caso della *load_elf*, se la *load_elf* legge nel file system della macchina host e quindi usa *emufs_ops* alla *VOP_READ* corrisponderà una *emufs_read* (se mettete un break point su questa funzione intercetterete la lettura del file elf in questo modo) e così per la write. Ce ne sono molte altre di funzioni, questa è una versione semplificata. La *load_elf* una volta che ha fatto questa prima *VOP_READ* ha iniziato a leggere qualche cosa che è un primo pezzo, ma questo primo pezzo sarà stato l'header del file elf. Una volta letto l'header del file elf, nella *struct elf header eh*, questa *struct elf header* che è stata caricata da un file con lettura da file binario in elf header conterrà un numero di segmenti nel file che corrisponde anche ad un numero di sotto-header quindi per i che va da 0 a *eh.e_phnum* si vanno a leggere delle ulteriori informazioni nell'header del file elf. Nell'intestazione della *load_elf* ci sono due variabili che si chiamano *Executable header* e *Program Header*. In realtà il Program Header è un Segment Header, ovvero ci sarà un Header per il segmento di dati e un Header per il segmento di codice. Quindi l'elf consiste di un Executable Header che è rappresentabile nella struct *eh* e due Program Header che saranno rappresentati nella struct *ph*. Una volta che abbiamo letto l'Executable Header e prendiamo il numero dei Program Header, iterando

su questi leggiamo di ogni segmento il suo Program Header e una volta letto questo Program Header o Segment Header noi sappiamo quali sono il Virtual Address, l'offset nel file, la dimensione in memoria e la dimensione nel file, cioè di ogni Segmento noi sappiamo dove dovrà stare in memoria, a quale indirizzo virtuale e di quale dimensione, sappiamo anche dove sta nel file e di quale dimensione. La dimensione del file potrebbe essere diversa dalla dimensione in memoria perché semplicemente potrebbe essere un po' più piccolo il file, ci potrebbe essere dell'informazione, ad esempio un pezzo di tutti zeri, che mancano nel file. Tuttavia, quello che notate è che nell'header c'è scritto, perché di fatto è come se fosse un indice nel file, dove vado a leggere nel file questo segmento, quanto è grande questo segmento, dove e a che indirizzo logico lo devo mettere in memoria. Una volta che ho fatto questa doppia lettura, questa iterazione, sono due i segmenti che contano (segmento di codice e segmento di dato). Una volta che ho fatto questo per ognuno di questi segmenti ho letto il *Virtual Address*, il *mem_size* e così via e posso fare *as_define_region*, cioè una volta che ho acquisito il program header di un segmento, le informazioni di questo program header, che sono *ph.p_vaddr* e *ph.p_memsz*, mi servono per generare lo spazio di indirizzamento di questo segmento, cioè per allocare la memoria che servirà per caricare il segmento. Rifaccio un po' d'ordine: una volta che noi abbiamo letto l'header *elf_header* dall'elf header ricaviamo le informazioni sui program header e per ogni program header prendiamo l'indirizzo di partenza in memoria logica e la dimensione in memoria logica e questi ci servono per definire la regione. *As_define_region* è quella che allocherà e metterà di fatto in corrispondenza indirizzi logici e indirizzi fisici.

Fatto questo possiamo ripetere il secondo giro sul file, rileggiamo i vari program header e per ognuno dei program header utilizziamo non solo più l'indirizzo virtuale e la dimensione ma anche dove sta nel file questo segmento e quanto è grande nel file questo segmento per fare effettivamente la *load_segment* per leggerlo. Quindi prima si legge l'elf header, poi si leggono i due program header solo per prendere le misure, poi si rifà il giro rileggendo i due program header e si leggono anche i segmenti. A questo punto l'elf è in memoria e possiamo completare il load con *as_complete_load*. Mi pare che questa funzione non faccia quasi niente, potrebbe essere necessaria se ci fosse un altro tipo di allocatore un po' più complesso della dumbvm. Fatto questo si prende sempre dall'elf header, perché c'è scritto dentro al file, l'entry point e lo si ritorna tramite il parametro by pointer *entrypoint*. Questo chiude la *load_elf*.

Read from file

Quello che ci resta da vedere sono gli aspetti legati al file system. In poche parole, come si fa a leggere e scrivere su un file? Qui siamo prevalentemente sull'operazione di lettura perché la *load_elf* sono operazioni di lettura, però quello che dobbiamo vedere qua nella *load_elf* è come si fanno le letture perché imparando a fare le letture da file nella *load_elf* possiamo capire come andranno fatte letture e scritture per implementare correttamente la *sysread* e la *syswrite*, che sono quelle che non abbiamo ancora realizzato completamente.

Vediamo quindi un po' più in dettaglio come viene effettuata una read. Supponiamo ad esempio di vedere la read che carica un *program header ph*. Abbiamo una struct che è compatibile, perché si sa quale è la struttura con la rappresentazione binaria su file, quello che vogliamo fare è leggere da file un program header sapendo dove incomincia nel file e acquisirlo in *ph*. Si lavora in questo modo: si fa una *uio_kinit* che serve a agganciare *ph*, il puntatore a *ph*, come destinazione e *sizeof(ph)* dimensione di questa destinazione dell'I/O più altre informazioni che sono state già messe a punto per quanto riguarda la lettura, cioè c'è questo *iov*, che è di tipo *struct iovec iov*, c'è questo puntatore *ku* che è un'altra *struct ku* che vengono passati alla *uio_kinit* oltre a dire alla *uio_kinit* guarda per favore fai una lettura. Quindi questa *uio_kinit* è: predispongo la struttura dati per fare la lettura e successivamente faccio la lettura usando come parametri soltanto il *vnode* (il puntatore al file) e il puntatore alla *struct ku* che alla fine è quella che contiene tutto ciò che è necessario. Quindi, per riassumere, per effettuare una read si deve predisporre opportunamente il

puntatore a una struct di tipo *struct uio* (*uio* sta per user IO ma in realtà è anche kernel IO), si passa un puntatore a una struct che descrive il tipo di io da effettuare e il puntatore al file control block.

Diamo un'occhiata a questa *ku*. Questa *struct uio* di cui vedete qui sia la descrizione che una rappresentazione, in realtà è una struct che contiene tutto ciò che è necessario per sapere che tipo di IO va fatto e come va fatto. Ora, qui vedete che abbiamo come prima informazione un puntatore al cosiddetto *iovec* e la dimensione dell' *iovec*, (in realtà questa *struct uio* *ku* può predisporre un input di un vettore di cose da leggere, in questo caso questo vettore *iovec* è un vettore da una sola casella, da una sola struct, ma potrebbero essere più di una) contiene (dell'unica lettura da fare qui) *iov_base* e *iov_len*, cioè vogliamo leggere dal file e vogliamo piazzare in *buff* per una dimensione *size*. Quindi *iovec* conterrà nella prima casella, in questo caso l'unica di questo vettore, il puntatore e la dimensione di ciò che va letto.

Ora questo è generale. Che cos'è questo *buff*? Veniamo al caso particolare nostro, che è una *struct uio* che serve per la lettura in kernel space. Partiamo dal fondo e vediamo che *iovec* dovrà puntare al *program header ph* e avrà la dimensione del program header, quindi l'obiettivo è chiaro: vogliamo leggere un program header e quindi il risultato dell'IO dovrà andare a finire nella memoria puntata da *ph* e per la dimensione della *struct ph*. Tenete conto che questa è una destinazione in memoria kernel (*ph* è una variabile locale del kernel). Bene, la *struct uio* saprà questo avendo nel *uio_segflag* la variabile *UIO_SYSSPACE* per dire: guarda che questa è una lettura non in user ma in memoria kernel quindi per favore usa la traduzione logico/fisica che va bene per il kernel (voi sapere bene che la traduzione logico/fisica che va bene per il kernel è semplicemente quella di fare somma o differenza per *KSEG0*). È una READ e non c'è un puntatore a uno spazio di indirizzamento. Quindi per fare la lettura in kernel space alla fine questa *struct uio* sarà predisposta in modo da contenere un puntatore al *ph* (indirettamente), un puntatore a program header, dimensione del program header e dire che si tratta di una lettura in kernel. Mancano ancora da analizzare i campi della *struct uio_offset* e *uio_resid* che sono un pochino da vedere ma in pratica avranno a che fare con dove bisogna leggere, quanto bisogna leggere, quanto si è letto, ma per ora lo lascio quasi sufficientemente vago. La *VOP_READ* quindi è sufficiente che riceva *v* che indica il file e *ku* che indica dove piazzare i dati in memoria, quindi la *VOP_READ* ha tutte le informazioni che servono in quanto *v* rappresenta dove dobbiamo leggere dal file system e *ku* che indica in pratica dove vanno a finire i dati in memoria, in questo caso in memoria kernel (non è esattamente così, *ku* contiene qualcosa in più). Se invece la destinazione di una lettura fosse *USERSPACE* allora *ku* (non è più *ku* ma lo chiamiamo *u*) conterrebbe in *iovec* ancora base e len però normalmente, supponiamo di leggere ad esempio un vero segmento user, avrebbe il puntatore allo user segment e la dimensione dello user segment, ad esempio 8192. Questo puntatore allo user_segment sarebbe lo stesso puntatore, ad esempio, che avete nell'address space come segmento 2. Ora, ricordate che, ad esempio, in address space voi avete di ogni segmento il puntatore all'indirizzo logico e il puntatore all'indirizzo fisico e la dimensione. Tenete conto che in questo caso *iov_base* e *iov_len* sono il puntatore all'indirizzo fisico e la dimensione della memoria in cui fare la lettura. Per fare una lettura in *USERSPACE*, e qui vedete che invece di *SYSSPACE* è indicato *USERSPACE*, la *VOP_READ*, la funzione di lettura, dovrà anche sapere come fare le traduzioni da indirizzo logico a indirizzo fisico e quindi ci sarà nel campo *uio_space* ci sarà il puntatore all'address space del processo. Questo puntatore non c'era quando abbiamo fatto un'acquisizione in kernel space. Mi correggo in parte, queste due frecce (l'indicazione del puntatore *as_pbase2* e *iovec_base*) sono una un indirizzo fisico e una un indirizzo logico allo user segment: quello che riceve la *struct uio* è il puntatore a una descrizione della destinazione in memoria di tipo logico, quindi *iov_base* è un indirizzo logico. Dall'address space voi reperite un indirizzo fisico per essere in grado di effettuare la traduzione logico-fisica corretta, quindi sono entrambe le informazioni. Una volta che abbiamo sistemato una *struct uio* in questo modo la *VOP_READ* è in grado di effettuare la lettura da *v* e di caricare i dati nello user segment che potrà essere o un segmento di dato o un segmento di codice. Ora internamente la *VOP_READ* può redirigere a seconda del file system usato o

all'*emufs_read* o al *sfs_read*. Qui ho cercato di darvi una visione molto di alto livello di quali sono le operazioni e non si può andare più di tanto in dettaglio di tanto. Vi dico solo che se voi guardate la *emufs_read* vedrete che di fatto fa da wrapper e chiama la *emu_read* e poi la *emu_doread*. La *emu_doread* internamente, al di là di acquisire un lock e di rilasciarlo alla fine, scrive in registri virtuali (fa finta di scrivere a un dispositivo scrivendo in dei registri) e dietro questi registri, una volta che ha scritto in questi registri l'intenzione, la programmazione dei registri per attivare un'azione di io si mette in attesa che l'io sia terminato. Dietro a questi registri di un file system, di un dispositivo emulato, ci saranno delle operazioni che non guardano un hardware simulato ma guardano il file system della macchina host. A questo punto c'è questa memory barrier *membar_load_load()* che aspetta che sia terminato tutto e una volta che a questo punto è tutto a posto ci saranno due operazioni, ma questa operazione sarà una *uiomove*, ovvero uno spostamento ma sono operazioni di io che vengono fatte, ma in pratica quello che voi potrete vedere è che qua c'è una gestione di: mando una richiesta di IO a un dispositivo virtuale, questo fa tutto e poi devo semplicemente trasferire a destinazione e la destinazione è presa da *uio*. Questo *sc->e_iobuf* è il buffer su cui sono stati acquisiti i dati dal dispositivo virtuale.

Per quanto riguarda il Simple File System *sfs* si vede un po' meglio il fatto che qui si tratta di un *blockio*: è estremamente sintetizzato questo però quello che si può vedere è che la *sfs_read* di fatto chiama una *sfs_io* che serve sia per input che per output che traduce un *sfs_blockio* perché il disco è un dispositivo a blocchi. Ora, siccome non abbiamo ancora visto il capitolo sull' IO, la distinzione tra dispositivi a carattere e a blocchi non è ancora chiara, ma sappiamo già che il disco contiene dei blocchi. Quello che si vede in questa *sfs_blockio* è che c'è una parte in cui si cerca di generare il numero logico del blocco sul file, perché il file è di fatto un vettore di byte e anche un vettore di blocchi. Poi c'è questa *sfs_bmap* che serve a fare la traduzione da numero di blocco logico a numero di blocco fisico. La traduzione da blocco logico a fisico è semplice se il file è contiguo, se il file è lista richiede una scansione in lista, se il file è rappresentato internamente con inode è un po' meno banale e assomiglia un po' all'esercizio su calcolo del numero di blocchi di dato di un file che abbiamo visto: dobbiamo andare a capire dove è piazzato, per esempio, il blocco logico 100 in un file rappresentato con inode. In questo caso fa tutto la funzione *sfs_bmap* che una volta ricevuto il numero di blocco su disco farà quest'operazione *Do the I/O*, che qui è estremamente sintetizzato, se voi guardaste la funzione corretta vedreste che c'è molto di più, ma alla fine viene fatta la lettura di un blocco, la *sfs_rwblock*.

In sostanza abbiamo visto come si può leggere un file elf e acquisirlo in memoria sfruttando operazioni sul file system che per quanto fumose ci fanno vedere come per lavorare su un file system serve un *vnode*, bisogna mettere a posto un *uio* e chiamare delle operazioni che sono *VOP_READ* e *VOP_WRITE*: questi saranno in sostanza gli ingredienti che vi serviranno per effettuare le operazioni di lettura e scrittura, o meglio *sysread* e *syswrite*, cioè le system call che fanno lettura e scrittura.

File ELF, formato file ELF e loro contenuto. Accento su aspetti principali, anche elementi collaterali.

File ELF contiene la descrizione dell'*address space* di un processo contenendo di fatto il segmento di codice e segmento di dato; più altre informazioni: ci sono i contenuti di quello che dovrà diventare l'*address space* e la descrizione.

Dal punto di vista di file, è un file che contiene header con indici o con puntatori ai dati. Ogni segmento del file ELF descrive una regione contigua nello spazio di indirizzamento virtuale. E per ogni segmento il file ELF contiene anche un'immagine del segmento che è semplicemente una copia di quello che deve finire in memoria. Oltre alla copia di ciò che deve finire in memoria, contiene un header, composto da ELF header e program header. L'header descrive l'indirizzo virtuale di partenza del segmento, lunghezza del segmento nel

virtual address space (come segmento va caricato in memoria), partenza e lunghezza del segmento su file. Poi l'immagine che è una copia esatta dei dati binari che devono finire nell'address space. L'immagine può essere un po' più piccola di quello che deve finire in memoria, perché può contenere parti da riempire da zeri.

ELF Files and OS/161

Per quanto riguarda DUMBVM l'implementazione assume che il file ELF contenga solo due segmenti (versione riduttiva ma sufficiente per eseguibili che dobbiamo utilizzare).

- 1) Un segmento detto text (che è quello di codice).
- 2) Segmento data segment che contiene dati globali e costanti.

Il file ELF non descrive lo stack perché lo stack parte vuoto. Non ha bisogno di una descrizione. DUMBVM creerà uno stack per ogni processo di dimensione fissa, 12 pagine; e terminante all'indirizzo virtuale (cioè l'ultimo indirizzo buono per indirizzi logici di user). Ci potrebbero essere modelli più complicati in cui si potrebbe pensare di creare un eseguibile consentendo di modulare la dimensione dello stack. In questo caso no.

ELF sections and segments

Cosa c'è dentro file ELF.

Perché dobbiamo capire cosa c'è dentro all'eseguibile che viene caricato in memoria. Nel file ELF, program code e program data, raggruppati in sezioni che iniziano con degli acronimi, iniziali che indicano cosa rappresentano.

Program code -> .text

Dati globali -> program data

.rodata -> read only data

.data -> dati globali inizializzati a 0 o ad altro

.bss block start by symbol

.sbss small uninitialized global data

- In the ELF file, a program's code and data are grouped together into sections, based on their properties. Some sections:
 - .text: program code
 - .rodata: read-only global data
 - .data: initialized global data
 - .bss: uninitialized global data (Block Started by Symbol)
 - .sbss: small uninitialized global data

Ci sono delle parti che rappresentano i dati globali. Non tutto c'è sempre nei file ELF. Di solito c'è .text e .data.

Si supponga di prendere programma C che ha volutamente una #define di una costante 200. Variabile x di cui si vede il contenuto in esadecimale.

Prendo file ELF e lo guardo direttamente in binario o in esadecimale cercando di capire ogni byte ogni word che cosa rappresenta. Si può utilizzare mips_harvard_os161_readeelf (software) che utilizzato in maniera opportuna permette di guardare quello che c'è dentro un file ELF.

ELF Sections for the Example Program

```

Section Headers:
[Nr] Name      Type      Addr      Off       Size      ES Flg
[ 0]          NULL      00000000 0000000 00000000 00
[ 1] .reginfo MIPS_REGINFO 00400094 000094 000018 18 A
[ 2] .text      PROGBITS 004000b0 0000b0 000200 00 AX
[ 3] .rodata    PROGBITS 004002b0 0002b0 000020 00 A
[ 4] .data      PROGBITS 10000000 001000 000010 00 WA
[ 5] .sbss     NOBITS   10000010 001010 000014 00 WAp
[ 6] .bss      NOBITS   10000030 00101c 004000 00 WA
[ 7] .comment   PROGBITS 00000000 00101c 000036 00
...
Flags: W (write), A (alloc), X (execute), p (processor specific)

## Size = number of bytes (e.g., .text is 0x200 = 512 bytes
## Off  = offset into the ELF file
## Addr = virtual address

The mips-harvard-os161-readeelf program can be used to inspect OS161 MIPS
ELF files: mips-harvard-os161-readeelf -a segments

```

Section header (REGINFO) deve stare all'indirizzo 1212 dimensione 18 offset 094.

REGINFO nel program headers ta al virtual address 094. MemSize e FileSize coincidono, cioè quello che ho trovato nel file è esattamente quello che deve corrispondere in memoria.

Program header sta al virtual address 0x094.

```

Program Headers:
Type      Offset      VirtAddr      PhysAddr      FileSiz MemSiz Flg Align
REGINFO  0x0000094 0x00400094 0x00400094 0x00018 0x00018 R 0x4
LOAD     0x0000000 0x004000000 0x004000000 0x002d0 0x002d0 R E 0x1000
LOAD     0x001000 0x10000000 0x10000000 0x00010 0x04030 RW 0x1000

```

MIPS_harvard_os161_objdump (software) può servire per vedere le istruzioni, decodificare istruzioni vedere un pezzo che significa niente nella sez. .text -> tradotte in istruzioni che hanno senso

Si può vedere anche contenuto della parte READ_ONLY data.

Si può visualizzare sia parte di testo e dati globali, programma per decodificare contenuti.

In conclusione, l'ELF contiene un eseguibile.

SYSTEM CALLS

Le system calls per la gestione di un processo, partenza processo utente, mancano la read() e la write() fatte decorosamente. System call sono l'interfaccia per programmi utente. Programma utente vede funzioni open(), read(), close() che hanno un loro prototipo.

Dal lato kernel quando viene chiamata system call, viene scatenata una software interrupt (trap), che chiama la funzione syscall passando un trap frame e nel trap frame si guarda il call number (callno) che serve a pilotare uno switch case.

Syscall() ha uno switchcase che inizialmente aveva solo la sysreboot e la systime, e noi abbiamo riempito questo e abbiamo gestito gli errori.

L'aggiunta di una nuova syscall deve prevedere un protocollo, con cui la sys_func, sysread() scritta così è una convenzione. Non è obbligatorio, ma è una convenzione nel mondo unix BSD. Se vogliamo rispondere alla system call read ci sarà una sysread().

E' opportuno che parametri della syscall() siano uguali a quelli della read() lato utente e che parametri siano spacchettati e ripresi dal trap frame.

L'importante è poter gestire i due valori di ritorno, cioè il risultato vero e proprio e il flag di stato o di errore. Devono essere gestiti.

Sys_read

-> primo param-> file descriptor

- Secondo param -> puntatore in memoria al buffer destinazione della read
- Terzo param -> numero byte che vanno letti

Sys write

Parametri uguali alla sys_read

Usr_ptr conferma che il kernel dovrà gestire indirizzo logico di user.

In OS161 il kernel vede nello spazio indirizzamento indirizzi user e kernel.

User_ptr può essere usato per vedere se realmente è indirizzo user.

SYS_READ()

Sys_read versione ridotta; se si tratta di una read() che corrisponde a stdin file number cioè al numero dello standard input che di solito è 0. E leggere un solo byte si potrebbe provare di implementare una kgets

perché conosco kgets. Ma per kgets andavano usati due buffer locali di due caratteri perché la kgets costruisce una stringa con terminatore di stringa. Quindi si sarebbe dovuto copiare il primo di questi caratteri in buff (sarebbe stato complicato).

Non veniva gestito caso di numero di byte superiore a 1. (versione semplificata)

Getch ha il vantaggio di non dover costruire una stringa come risultato, cioè non mettere un terminatore, ma prendere direttamente. E può mandare a destinazione (*buf) risultato della getch ma stiamo gestendo solo casi di acquisizione da std input di un solo carattere.

SYS_READ() più ragionevole

Non c'è il vincolo di numero di byte che vengono acquisiti. Sulla sysread non c'è un vero e proprio controllo. Dipende molto dall'input che fa il programma -> gets o lettura carattere su console o su altro.

Se file descriptor è stdin proviamo a supportare size>1, for i che va da 0 a size acquisiamo carattere getch

P[i] buffer finale.

Iterazione che acquisisce carattere alla volta.

Sys_write()

Si potrebbe adottare strategia analoga, kprint %c -> è semplice e permette di essere più flessibile; non c'è kscanf.

Kprintf permette stampa di un solo carattere. Anche putch.

Per supportare for si mette putch con i da 0 a size.

Sys_exit()

Si ricorda che una variante della sys_exit quella che ha as_destroy() direttamente nella sys_exit(), fa thread exit mette a posto lo stato e non si preoccupa di sincronizzarsi (waitpid) e della distruzione del processo.

Il difetto è che non fa sincronizzazione. La sincronizzazione può essere fatta così:

- 1) proc 1 con PID 7 e fa exit.
- 2) proc2 waitpid(7).
- 3) proc 1 dovrebbe fare syscall che chiama exit.
- 4) proc 2 syswaitpid con 7

(Versione ridotta)

Sys_exit with synch

Lato kernel, la syscall sysexit corrisponde a una sys_exit che salva lo status (semplificata senza spinlock).

Poi signal_end che nasconde signal e poi thread exit.

WaitPID avrebbe chiamato syswaitpid -> 1 trova puntatore processo guardando tabella di processi -> proc_wait(proc trovato).

Proc_wait(struc proc*) dovrebbe aspettare fine processo con semaforo e poi fare la proc destroy.

Signal e wait si implementano usando semafori o condition variable (V/P o wait/signal(wakeone)).

C'è un problema. Si supponga processo che fa sysexit e segnala fine processo su un semaforo e poi chiama thread_exit.

Dall'altra parte processo che fa wait (P) e fa proc destroy.

Problema: non è garantito che proc_destroy e thread_exit chi tra i due parta prima e venga eseguito prima.

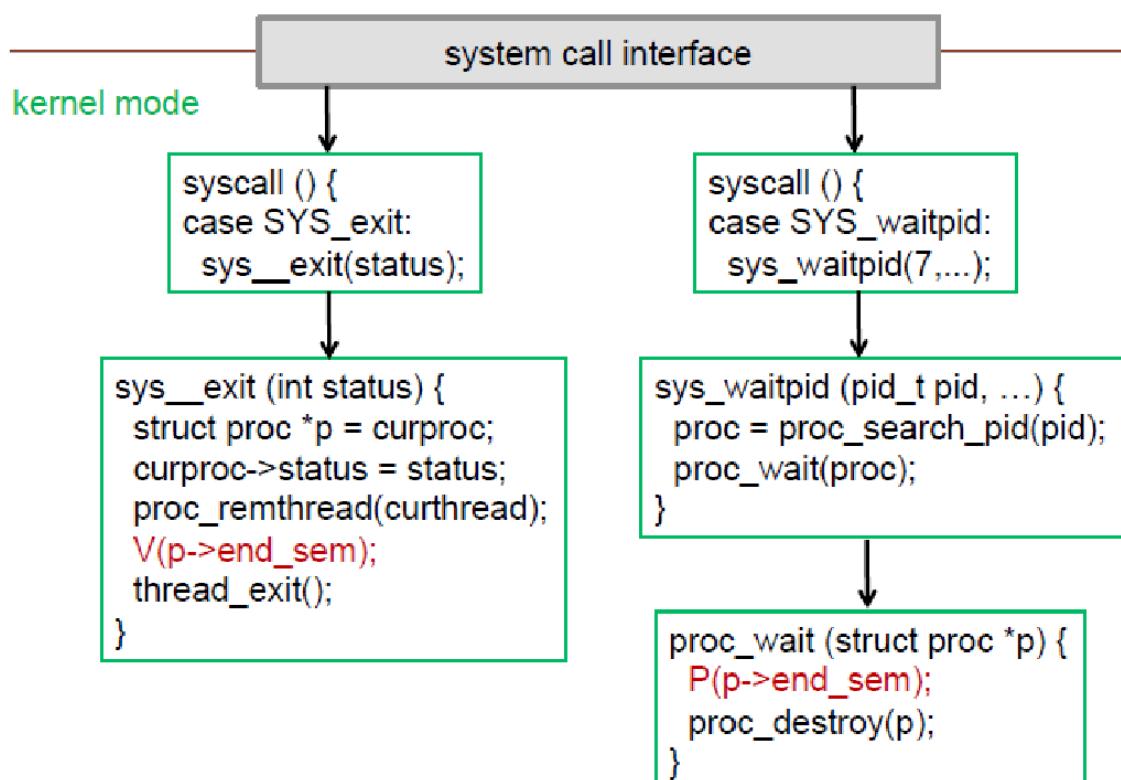
È interessante fare therad_exit e poi proc_destroy -> stacco thread dal processo -> quando faccio proc destroy posso distruggere processo perché non ci sono più thread.

Se proc destroy parte prima, proc ha ancora thread e non ci riesce.

Per costringere proc_destroy di essere invocato dopo della thread_exit, si evita di costringerlo ma si cerca di anticipare -> 1) aspetto a chiamare la proc_destroy (sincronizzazione aggiuntiva, pilotata all'interno della thread_exit) per avvisare che ho terminato oppure spostare la P dentro la thread_exit ma facendo attenzione che la thread exit non viene chiamata soltanto dentro alla sysexit ma anche alla fine della vita di un thrad.

2) più semplice staccare thread prima di fare la signal, farlo in anticipo -> thread_exit e proc_destroy viaggiano in maniera separata.

sys_exit (with synch)



Proc_search_pid

Gestire pid di un processo, tabella/vettore di processi, vettore di puntatore a struc proc.

Versione semplice vettore di puntatori

Dimensione vettore: PID_MAX+1

Versione ridotta os161 non è necessario attivare tanti processi.

Allocazione statica ma anche dinamica.

Ha senso allocazione dinamica se si pensa di avere una dimensione decisa in fase di configurazione (sys161.conf).

Se si decide di estenderla -> kmalloc.

Tabella dei processi è il secondo tipo di struttura dati che rappresentiamo, la prima era la bitmap, allocazione contigua delle pagine -> tabella dei file associati ai processi.

Manca presentazione implementazione file system nella versione finale, si dovrà ancora realizzare la read e la write e anche la open e la close.

Ispirarsi nella load_ELF per la gestione di vnode piuttosto che delle vop read e vop write.

10. Sistemi di I/O

Mettiamo l'enfasi sulla parte del S.O., tenendo conto che in un kernel la parte di gestione dell'I/O è quella più versatile e variabile, e più dipendente spesso da pezzi di S.O. scritti da terze parti (produttori HW di I/O).

- Riassunto e panoramica dell'HW di cui stiamo parlando
- Interfaccia verso le applicazioni
- Quello che c'è nel kernel
- Procedura per far diventare in HW qualcosa che è richiesto dal SO
- Riassunto di alcune implementazioni, alcuni casi particolari

Gli obiettivi saranno un po' un mix tra panoramica, discussione su principi generali e spiegazione nel dettaglio di alcune parti. Questa lezione serve a descrivere in modo abbastanza di alto livello dei principi generali di cui vedremo una concretizzazione più dettagliata in ambito os161.

I/O è importante in un S.O. perché non ci sono programmi che non lo fanno, e più si va avanti e più il sistema di elaborazione è un sistema intelligente che esegue programmi che interagiscono con dell'HW. L'HW non è ovviamente qualunque tipo di dispositivo ma tendenzialmente l'elaboratore avrà interazione con attuatori, sensori o dispositivi che servono a pilotare l'HW. Cioè un computer non è un'auto ma può stare a bordo di essa e controllare e monitorare i dispositivi che stanno a bordo. I dispositivi di I/O sono molto variabili, e ci sono strategie/metodi diversi per pilotarli e controllarli. Ci sono aspetti prestazionali, cioè come si fa a gestire dispositivi che hanno tempi di operazione molto diversi da quelli tipici della CPU? C'è poi una notevole variabilità in quanto è un settore molto dinamico in cui di anno in anno (mese in mese) saltano fuori nuovi dispositivi.

Dal punto di vista della terminologia: per parlare di HW dovremmo parlare di porte, bus, controllori. Dal punto di vista S.O, il livello più basso sarà detto **device driver**: è una componente del S.O, quella di più basso livello, che capisce e si interfaccia con l'HW ed è quella che direttamente lo pilota sia dal punto di vista di

programmazione che di gestione dei dati, e fornisce a più alto livello un'interfaccia tutto sommato indipendente dall'HW.

I/O Hardware

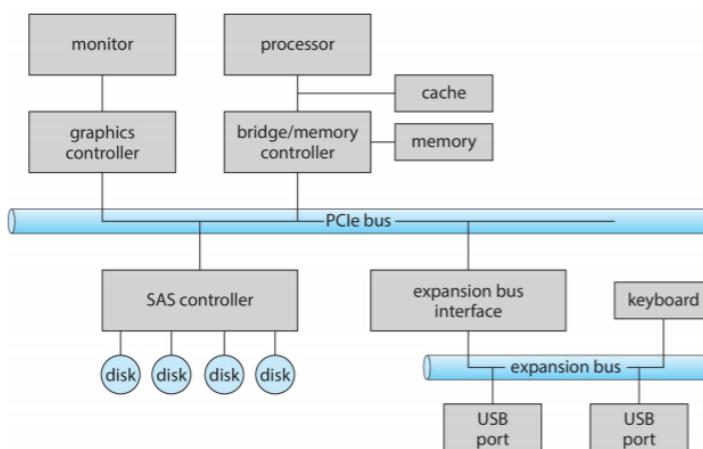
Che cos'è l'HW di I/O? C'è una grande variabilità ma tendenzialmente possiamo distinguerlo in:

- storage dei dati
- trasmissione dei dati che sia di rete o locale tra dispositivi
- interfaccia uomo macchina che si tratti di video tastiere mouse dispositivi touch ecc. che comprendono anche dei sensori di tipo medicale o attuatori a bordo di auto. Bisognerebbe distinguere tra sistemi embedded (prevalentemente centrati all'I/O e al controllo di altri dispositivi) e dispositivi di elaborazione di tipo generale.

Comunque, è chiarissimo che ci sono queste macrocategorie.

In questo contesto ci sono alcuni concetti comuni che fanno riferimento a come i dispositivi di I/O si interfaccino con la CPU ed in particolare col sistema di elaborazione. In particolare, questi concetti comuni possono essere:

- **porta**: si intende una vera connessione fisica, cioè il punto in cui il dispositivo si attacca fisicamente, elettronicamente al sistema.
- **bus** (daisy chain e accesso diretto condiviso al controllore). Con il termine bus si intende più come il dispositivo parla in termini di dati, come comunica con il sistema di elaborazione. Il bus è una delle strategie più adottate, cioè costituisce un canale comune su cui in linea di massima ad un dato istante qualcuno scrive e qualcuno legge. Il bus è anche il canale di comunicazione privilegiato tra computer e memoria ram; tuttavia di base ne esistono di vari tipi che si differenziano non solo per la dimensione, ma soprattutto per il tipo di indirizzamento e il tipo di controllo che viene effettuato sui bus. Mentre sul bus molti accedono (bus condiviso, uno scrive ed almeno uno legge) i meccanismi a daisy chain intendono invece delle organizzazioni a catena in cui ognuno è connesso solo ad altri due: e se un dispositivo A deve parlare con B allora c'è una specie di passaparola. Invece accesso diretto condiviso è di nuovo basato su un dispositivo comune (inteso più come un dispositivo di memoria comune), cioè se A vuole parlare con B scrive in memoria qualche indirizzo e B legge in memoria l'indirizzo opportuno. Ora, tra questi dispositivi possono esistere differenze enormi, vengono citati i bus PCI che è uno degli standard per i pc, poi i bus di espansione che sono bus aggiuntivi rispetto a quelli di memoria o anche rispetto ai PCI, e poi strategie più a daisy chain quali SCSI che sono meccanismi previsti per comunicazione con dischi veloci o dispositivi veloci connessi al computer.
- **controller** o host adapter che è un'entità attiva. Con attiva si intende che c'è un microcontrollore che sta dal lato del dispositivo e si interfaccia con i bus per parlare con la cpu.



Ora questo è uno dei possibili schemi, si basa appunto sullo schema del PCI bus con bus di espansione per ad esempio porte USB tastiera e eventualmente altro; c'è un bus SCSI per i dischi; c'è un PCI bus su cui si interfaccia direttamente il monitor tramite il controllore grafico e c'è il processore con la sua memoria cache che si interfaccia direttamente con la RAM (il bus con la RAM sta in questo controller o è nascosto da

questo controller (bridge) e arriva anche questo sul PCI bus). Quello che si vede qua è che la comunicazione processore memoria dev'essere veloce invece la comunicazione con il monitor passa dal PCI bus ma c'è il controllore grafico che permette di gestire localmente al sottosistema del monitor una parte significativa delle operazioni grafiche. Poi c'è una sezione che è pensata per i dischi che devono avere delle prestazioni per conto loro e l'expansion bus che è praticamente un bus per dispositivi lenti (le porte USB odioggi sono comunque abbastanza veloci). Questo è comunque uno schema significativo di un'architettura hardware vista a un livello estremamente alto.

Lasciamo stare Fibre Channel che è un altro dispositivo. In questa slide vediamo un qualcosa in più, non solo come sono connessi ma come fanno la CPU e i dispositivi di I/O ad interagire: di solito ci sono delle istruzioni di I/O per controllare i dispositivi ma soprattutto i dispositivi di I/O sono visti dalla CPU come se fossero delle locazioni di memoria. In realtà i dispositivi di I/O hanno dei registri su cui il software (in questo caso il device driver) essenzialmente piazzano informazioni di controllo o informazioni di dato; o meglio su questi registri possono passare comandi, indirizzi o anche dati da scrivere o da leggere. Quindi normalmente un dispositivo di I/O potrà avere un Data-in Register e un Data-out Register, uno status register, un control register: lo status è quello che viene letto dalla CPU per capire lo stato del dispositivo e il control è quello che viene scritto per comandare delle azioni. Normalmente sono dei registri da 1 a 4 byte, quindi roba piccola. Possono eventualmente essere dei buffer fifo intendendo che deve esserci la possibilità da una delle due direzioni di accumulare più dati disaccoppiamento un pochino le azioni l'uno dall'altro, cioè la CPU potrebbe scrivere quattro interi da spedire a un dispositivo e non aspettare necessariamente che il primo sia stato letto per cominciare a spedire il secondo. Quindi i buffer fifo sono tendenzialmente dei dispositivi molto utilizzati come canali di comunicazione per dati, poi normalmente i registri a bordo di un dispositivo non sono visti dalla CPU come se fossero dei normali registri con un nome (anche perché è abbastanza variabile, non si capirebbe bene quali nomi dare ai vari dispositivi). Normalmente i registri dei vari dispositivi di I/O rispondono a degli indirizzi di memoria cioè è come se fossero delle locazioni di memoria. Quindi i dispositivi hanno degli indirizzi e dal punto di vista assembler, quindi dal punto di vista della CPU, ci sono due possibilità:

- **Direct I/O instructions:** ci sono delle istruzioni speciali di I/O (in alcuni microprocessori sono proprio in out cioè leggi o scrivi dal dispositivo che corrisponde a questo indirizzo, ma si sa che è un dispositivo di I/O)
- **memory mapped I/O** che significa semplicemente load/store. Potrebbe essere memoria RAM oppure potrebbe essere un dispositivo di I/O: questo dipende semplicemente da quale è l'indirizzo che scrivete ed è ovvio che nel momento in cui si fa la mappatura degli indirizzi fisici in RAM bisognerà capire a quali indirizzi risponde effettivamente la RAM, a quali indirizzi invece risponde qualcos'altro.

Qui vedete una possibile configurazione, a bordo di un pc, di quali sono i dispositivi (DMA controller, interrupt controller, timer e via dicendo) che rispondono agli indirizzi in I/O. Normalmente gli indirizzi di I/O non hanno bisogno di altrettanti bit quanti gli indirizzi di RAM: il motivo è che è inutile usare i gigabyte per i dispositivi di I/O, bastano poche decine o qualche centinaia di indirizzi per ogni dispositivo. Qui si vede come su un PC siano sufficienti indirizzi su tre cifre decimali che sono ad esempio da zero a quattro kilobyte; qui vedete che addirittura si arriva solo fino a tre FF, sarebbe soltanto il primo kilobyte anziché 4 kilobyte.

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Polling

E ora un'occhiata a come si fa a interagire con un dispositivo di I/O, ne abbiamo già parlato più volte in altri contesti. Ci sono delle modalità basate su polling, che sono essenzialmente una forma di busy waiting, e un altro tipo di interazione che si basa sull'interrupt.

Cosa vuol dire fare polling? Vuol dire che per ogni byte di dato che vogliamo leggere o scrivere in I/O dobbiamo aspettare che il dato sia pronto prima di leggerlo o aspettare che il dispositivo sia pronto a ricevere un dato prima di scriverlo.

- 1) Per fare questo normalmente il dispositivo ha nel suo status Register un bit, per cui la prima cosa che si può fare è leggere il busy bit dallo status register: fin quando questo bit è a zero noi andiamo avanti;
- 2) una volta che il dispositivo è pronto mettiamo a 1 il bit cioè decidiamo qual è l'operazione che si vuole effettuare (lettura/scrittura). Se l'operazione è di scrittura scriviamo il dato e dovremo solo aspettare che il dato sia stato mandato in uscita e quindi lo copiamo nel registro data-out. Se invece l'operazione è di lettura bisognerà fare un trasferimento dal lato opposto cioè bisognerà andare a prendere un dato dal data-in (nelle slide si rappresenta l'operazione di scrittura per esempio).
- 3) A questo punto l'host (ricordiamo che l'host qui è la CPU che comanda, il controller è il microcontrollore a bordo del dispositivo, quindi l'host parla col controller o interagisce col controller e quando agisce uno non agisce l'altro) setta il command ready cioè vuol dire che il comando è pronto (ho deciso che bisogna scrivere o leggere) e l'operazione è fatta (lato host).
- 4) A questo punto la palla passa al controller che setta il busy bit ed esegue transfer: se il controller deve fare un'operazione di scrittura allora il dato è già stato impostato, prende quel dato e lo manda in I/O; se deve fare un'operazione di lettura il controller acquisirà il dato lo metterà sul data-in e toccherà poi successivamente al host andare a leggere il dato (è un po' più complicato pensare all'alternanza di operazioni in input perché comando l'input, l'input viene eseguito e poi il dato verrà preso dal l'host solo dopo).
- 5) A questo punto una volta che il trasferimento è stato eseguito, il controller settará il busy per segnalare che è libero. Mette a posto l'eventuale error bit e tutto il resto ma sostanzialmente completa l'operazione.

Tutto ciò ha una parte critica, il punto (1), nel senso che è un busy wait, cioè potrebbe succedere che l'attesa sia lunga. Se il dispositivo è veloce ha senso fare busy wait, se è lento non ha molto senso. La CPU potrebbe fare busy wait e dire vabbè ma io faccio altro. Il problema che potrebbe far altro e non risvegliarsi al momento giusto. Il problema è nel che pensare di continuare a fare busy wait essendo schedulato e mandato in stato ready, o a fare altro, potrebbe decisamente perdere il momento buono o l'attimo buono per acquisire il dato o perdersi addirittura dei dati; per cui la vera alternativa al polling è l'interrupt.

Interrupts

Fare polling significa controllare ogni minuto l'ora per vedere se sono arrivate le 8 di mattina per svegliarsi. Interrupt significa dormire e aspettare perché avete impostato una sveglia che vi sveglia una certa ora. Interrupt significa che a un certo punto la CPU riceverà sulla interrupt request line, cioè su un filo di ingresso, una richiesta di interruzione. C'è un protocollo apposito per dire che in una certa fase dell'esecuzione di una istruzione macchina è possibile che la CPU venga interrotta, abbandoni o lasci sospeso quello che sta facendo e attivi un protocollo di servizio dell'interrupt. L' interrupt handler è quello che riceve le richieste di interrupt che possono essere disattivabili o meno. L' interrupt vector è semplicemente un vettore che serve ad associare a dei numeri interi ben codificati delle azioni da fare cioè dovrà dire in parole povere che cosa bisogna fare, cosa bisogna attivare come risposta all'interrupt. Il motivo è che sostanzialmente c'è un solo filo per dire, ma sono tanti che possono interrompere e ognuno deve essere associato ad un numero da ricevere da parte della CPU in parallelo insieme alla richiesta di interruzione.

Ci sarà una forma di context switch, gestito tramite il trapframe, che permette di fermare e di sospendere chi è in esecuzione per attivare il servizio dell'interrupt. Ci sono degli schemi basati su priorità per decidere

a chi tocca nel caso vi siano interrupt multipli. Ci sono interrupt mascherabili e altri non. Ci sono delle strategie per mettere in catena o organizzare più richieste di più dispositivi che possano richiedere interrupt. Tutto questo è in qualche modo una visione hardware.

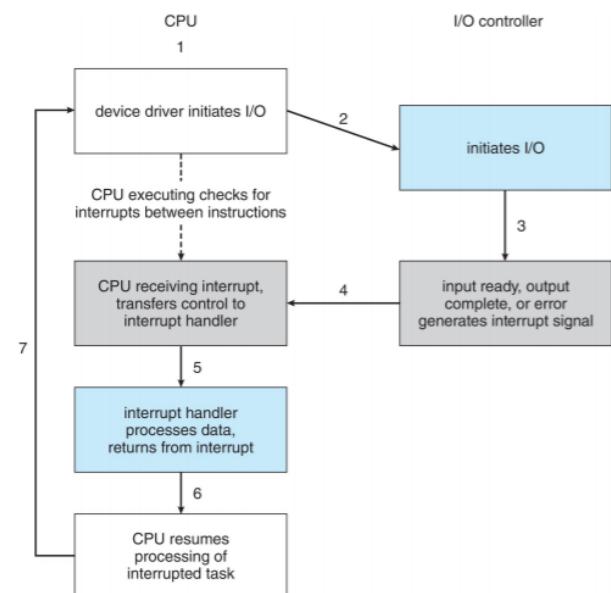
- Interrupt-Driven I/O Cycle

Dal punto di vista del software come possiamo pensare di organizzare un I/O basato su interrupt? Ora qui vediamo che da un lato abbiamo la CPU e dall'altro abbiamo l'I/O controller.

C'è il device driver, che è il livello basso del software, che riceverà una richiesta di operazione. Il device driver è quello che comanda l'I/O (potrebbe anche lavorare in modalità polling e cominciare facendo il busy wait). In questo caso non fa alcun busy wait ma inizia l'I/O.

L'operazione dal lato hardware sostanzialmente può essere vista, qui a destra numero 2 3 4, come effettuare un'operazione: bisogna iniziare l'I/O comandarla e poi eseguirla. Vedete che la CPU sostanzialmente comanda e inizia l'I/O e poi il device driver software aspetta che l'I/O sia completato. Cioè invece di dire faccio un loop in cui aspetto che il dispositivo sia pronto e comando le azioni, la CPU esegue l'inizializzazione (tipicamente si programma o si scrive qualche registro di controllo) lo faccio partire e aspetto che venga eseguito. Ma questo aspetto che venga eseguito dalla parte del device bianco si concretizza nella buona sostanza da un servizio di interruzione.

Ritorniamo a destra operazione 3: dopo essere stato inizializzato l'I/O viene eseguito (che vuol dire se un input ci sarà ormai il dato pronto per essere preso, viceversa il dispositivo ha ricevuto i dati) e quindi siamo pronti a ricevere un nuovo output. Detto ciò il dispositivo (se volete I/O controller) una volta che sa di aver effettuato l'operazione manda un interrupt che interrompe qualunque cosa sia in esecuzione. Vi faccio notare la freccia tratteggiata: sostanzialmente il device driver è in stato di wait e c'è qualcun altro in esecuzione nella CPU. Questo qualcun altro è un terzo o un qualcuno che è un altro processo che sta sfruttando la disponibilità della CPU. Ma come tutti i microprocessori, campiona in una certa fase di ogni istruzione l'eventuale interrupt. Quando arriva l'interrupt, pilotato dall'azione grigia (se volete qua dal I/O controller) parte il cosiddetto interrupt service routine, cioè la routine di servizio dell'interruzione che tecnicamente deve svolgere il compito dell'interruzione. Vuol dire "vai a piazzare il dato dove deve essere messo, fai dei conti, aggiorna un contatore, bufferizza, insomma fai qualche cosa" (c'è una piccola differenza per il fatto che sia un I/O a blocchi o a caratteri). Questo qualche cosa potrebbe essere troppo lungo per ciò che tipicamente sta in un interrupt service routine: una routine di servizio dell'interruzione deve normalmente essere breve perché dev'essere corto, in quanto quando stiamo servendo un'interruzione normalmente gli interrupt sono disabilitati. Quindi la CPU è dedicata al cosiddetto interrupt service routine che deve essere rapida. Se il lavoro è un po' lungo normalmente lo si suddivide in interrupt service routine e in interrupt handler che vedete qui (operazione 5 fino alla 6). Ora, questo risultato dell'operazione 5 e 6 porta alla fine,



se i dati sono finiti cioè se l'operazione di I/O è finita, a risvegliare di nuovo il device driver che "dice operazione completata".

Ora potremmo anche ricordarci che gli interrupt sono una forma di eccezione e che il meccanismo di interruzione è costoso ed è comunque qualche cosa che implica un cambio di contesto e un costo in generale. Le eccezioni in qualche modo stoppano, fermano, congelano quello che si sta facendo, ne salvano lo stato e fanno partire una exception handler che deve decidere cosa fare. Tenete conto che su un sistema multi-CPU, ogni CPU potrebbe ricevere un interrupt indipendentemente dalle altre e il software dovrebbe essere in grado di gestirlo. Ma ci accontentiamo qui di dire che di interruzioni ce ne possono essere molte e che costano.

Latency

	SCHEDULER	INTERRUPTS	0:00:10
total_samples	13	22998	
delays < 10 usecs	12	16243	
delays < 20 usecs	1	5312	
delays < 30 usecs	0	473	
delays < 40 usecs	0	590	
delays < 50 usecs	0	61	
delays < 60 usecs	0	317	
delays < 70 usecs	0	2	
delays < 80 usecs	0	0	
delays < 90 usecs	0	0	
delays < 100 usecs	0	0	
total < 100 usecs	13	22998	

Tanto per avere un'idea, qui vedete l'output di un comando disponibile sui sistemi macOS, latency. Con questo comando vi farà vedere in sostanza quanti sono gli interrupt che avvengono; per dire che su un sistema operativo reale non sono solo due o tre ma sono molto più frequenti.

Qui vedete una vector table che fa vedere come i numeri delle interruzioni siano mappati sulle potenziali cause di interrupt: fa vedere come per errore di divisione avete il numero zero nel vector number. Per un'eccezione di debug il numero 1 e così via.

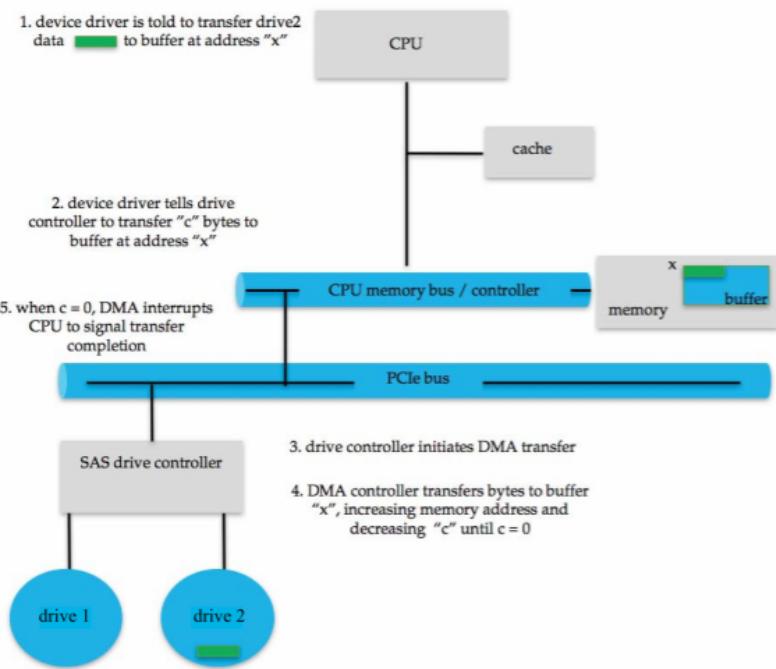
vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Direct Memory Access

Ulteriore nota dal contesto Hardware che va tenuto in conto qui e che può avere significato nel contesto di un device driver e/o di un'operazione di I/O è che non tutto dev'essere fatto dalla CPU: cioè quando diciamo "fai partire l'I/O" e poi ci sarà un meccanismo di interruzione che avviene, dovreste tenere presente il fatto che ci sono dei trasferimenti che possono essere di fatto completamente demandati all'hardware anche se sono trasferimenti che impattano sulla memoria o che sono da memoria a dispositivo ma comunque che interessano grossi blocchi di memoria. Il cosiddetto DMA direct memory access è una tecnica che implica un hardware adeguato e serve sostanzialmente ad evitare alla CPU di fare troppo. Richiede il cosiddetto DMA controller, cioè richiede che dall'altra parte (dalla parte del dispositivo di I/O), tipicamente un disco o un dispositivo che fa I/O per blocchi, ci sia un dispositivo in grado di gestire il DMA. Serve a evitare il cosiddetto I/O programmato, dove I/O programmato significa "se devo trasferire 1000 byte o 1000 word a 1000 interi la CPU dovrà eseguire 1000 volte delle load o delle store". Col DMA invece consiste in questo: la CPU dice "per favore DMA controller trasferisci 1000 byte 1000 interi 1000 Word 1000 float per me" e la CPU non fa mille operazioni di load o store ma le fa qualcun altro cioè il DMA controller. Esso di fatto bypassa la CPU e

pilota i bus dati, bus di indirizzi e bus di controllo, e pilota i trasferimenti tra memoria e dispositivo. Quindi ci sono dei momenti in cui la CPU non ha il controllo sui bus di accesso alla memoria. Per fare questo il sistema operativo ha l'incarico di dire al DMA controller "fai il trasferimento di quanti dati e a partire da dove in memoria": quindi servono gli indirizzi di partenza e di arrivo, serve sapere se si tratta di lettura o scrittura (quindi la direzione), quanti bit (cioè la dimensione di ciò che viene trasferito) e un opportuno comando al DMA controller. Esso diventerà un attore in più nel controllo sui bus (in un sistema multicore sono più di una le CPU che controllano i bus quindi è normale che ci possa essere la memoria già condivisa; qui la memoria sarà ulteriormente condivisa nel senso che ci sarà anche un controller che compete con le CPU). Quindi il controller agisce al posto della CPU sui bus operando quello che si chiama cycle stealing che vuol dire "nel tempo i bus di accesso alla RAM avranno dei momenti rubati alla CPU nel senso che la CPU non li sta utilizzando o non sta utilizzando i bus ma è il DMA controller che pilota questi bus per attivare delle operazioni di load/store nella RAM". Quando ha finito, il DMA controller segnalerà il completamento delle operazioni. Ci sono versioni del DMA che possono lavorare su indirizzi fisici e altri che possono lavorare anche su indirizzi logici, ma non mi addentro ora su questo problema.

Schematicamente parlando qui vedete una possibile rappresentazione quantomeno dal punto di vista di dove stanno i vari pezzi. Vedete in alto (numero 1) la CPU su cui gira il device driver che ha sostanzialmente lo scopo di far partire le operazioni cioè "trasferisci dei dati dal drive (cioè da un dispositivo) a un buffer cioè a un'area di memoria che parte all'indirizzo x". Vedete a destra (memory) dove c'è questo buffer all'indirizzo x e vedete che drive 2 è uno dei dispositivi (ad esempio dischi SCSI) da cui ci si aspetta che questi dati (rettangolo verde) vengano letti o scritti a seconda di quale l'operazione in memoria. Poi la sequenza di operazioni pilotata dal DMA parte da quella che vedete numerata qui con 2, il comando DMA controller è "trasferisci per favore c byte al buffer che sta all'indirizzo x" (operazione numero 2). Il device controller, cioè il lato hardware, fa partire l'operazione, itererà trasferendo i byte e prendendo il controllo della memoria all'indirizzo x decrementando ogni volta c (quindi facendo un loop); quando c è arrivato a zero il DMA dice "dati trasferiti, ritorniamo al controllo alla CPU".



Application I/O Interface

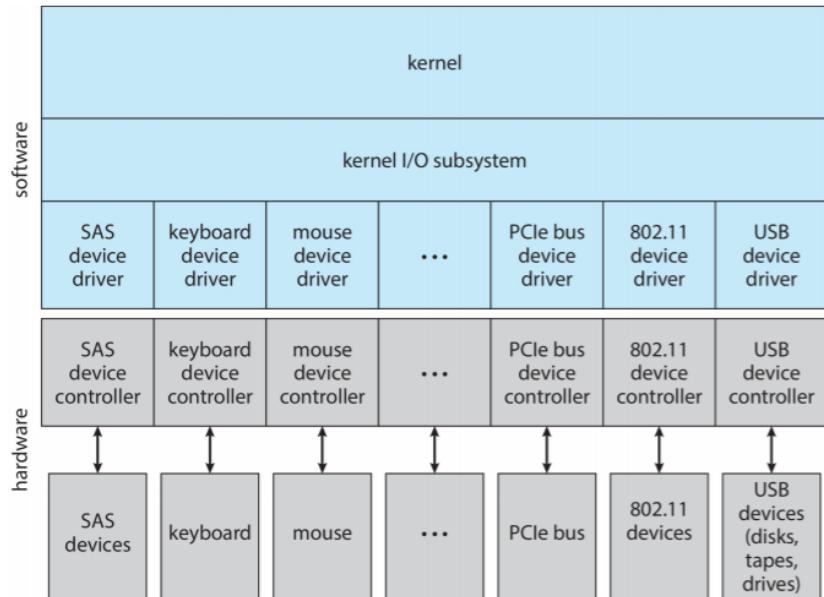
Saliamo di un livello dal device driver all'interfaccia verso il l'applicazione (verso il software applicativo). Le applicazioni si interfacciano con i device drivers tramite delle system calls cioè per pilotare degli I/O si passa tramite esse. I device drivers sono il pezzo che verrà attivato da una system call che nasconderà dettagli interni, ci saranno poi delle parti invece che sono di più alto livello. Ad esempio le read e le write che scrivete (os161) sono in parte gestite a livello di system call e in parte scenderanno al livello più vicino al dispositivo con VOP read e VOP write. I dettagli che vengono nascosti dai device driver e che impattano sui dispositivi (il tipo di dispositivo) riguardano quali tipi di dispositivi possiamo usare e come avvengono le operazioni in questi dispositivi. Notate come i dispositivi possono variare secondo più criteri:

- dispositivi a cui di fatto trasferiamo un carattere alla volta tipo per esempio la tastiera o il video in modalità testuale o dispositivi a blocchi quali i dischi normalmente.
- I/O di tipo sequenziale o di tipo casuale dove casuale vuol dire diretto
- abbiamo un I/O di tipo sincrono o asincrono
- abbiamo un dispositivo condivisibile o dedicato
- veloce o lento. Conta anche la velocità del dispositivo: è un dispositivo di rete veloce o un mouse?
- le operazioni sono di tipo scrittura e lettura, solo scrittura o solo lettura?

Cerchiamo di capire di cosa stiamo parlando. Quali sono le operazioni.

Ora questa classificazione viene in qualche modo ritrovata in quella che è **l'architettura del sottosistema di I/O del kernel**.

Come vedete dal basso ci sono i dispositivi di I/O e in corrispondenza i device controller che sono comunque una parte ancora hardware. Nel software voi vedete sostanzialmente che il kernel a livello software ha un sottosistema di I/O indipendente dal dispositivo; ha invece una parte dipendente dal dispositivo con una mappatura 1:1 con quei dispositivi che sono tendenzialmente i device driver (o la parte bassa dei device driver) (quanto meno possiamo pensare a un livello di astrazione sufficientemente approssimato che per ogni dispositivo ci sia un device driver).



Characteristics of I/O Devices

Dal punto di vista delle caratteristiche dei dispositivi di I/O in questa tabella possiamo notare qual è la caratteristica e quali sono i possibili valori di quella caratteristica. Ad esempio, per il tipo di trasferimento (I/O a caratteri o a blocchi) vediamo che il terminale (la tastiera + il video terminale) è un esempio di trasferimento a caratteri perché voi scrivete un carattere alla volta; oppure il disco. Con metodo di

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read-write	CD-ROM graphics controller disk

accesso sequenziale c'è il modem mentre il cd rom accesso casuale. Poi potrete avere un meccanismo di scrittura di tipo sincrono o asincrono (sincrono = prima di passare al prossimo aspetto che sia finito il precedente I/O; asincrono = tipo la tastiera in cui schiaccio un tasto e schiaccio il prossimo anche se non so ancora se il primo tasto sia arrivato no). Poi un dispositivo può essere dedicato come un nastro magnetico o condiviso come la tastiera che può essere condiviso tra più processi per esempio. Poi riguardo la velocità vedete elencate alcune caratteristiche che impattano sulla velocità del dispositivo che abbiamo visto con i dischi. Riguardo la direzione di I/O c'è quello in sola lettura, ad esempio il cd rom, in sola scrittura ad esempio controllore, sia lettura che scrittura il disco.

Mettendo in evidenza alcune di queste caratteristiche particolarmente importanti si potrebbe dire che di solito dal punto di vista di un sistema operativo si tendono a distinguere dispositivi di I/O a blocchi (tendenzialmente i dischi), a caratteri (normalmente tastiera e video testuale), ad accesso a file di tipo memory mapped e i network socket (sono dispositivi software che permettono di interagire in rete). Questi sono dispositivi diversi che fanno riferimento a dischi, interazione uomo macchina con tastiera, file system e/o rete (non è una classificazione esaustiva). Poi dal punto di vista di dispositivi di I/O ci sono anche delle possibilità in Unix, tramite la funzione ioctl(), di interagire direttamente col dispositivo; quindi questa è una funzione di tipo generale che permette varie tipologie di interazione con il dispositivo compreso vederne lo stato o pilotarlo o fare altre cose. Dal punto di vista della classificazione ai dispositivi, almeno in un sistema Unix, si attribuiscono due numeri invece di uno solo, che sono major e minor: il primo numero è per adottare un primo criterio di classificazione e il secondo un altro livello. Ad esempio il disco 8, in questo caso major 8, può vedere dei sotto dischi o sotto volumi che hanno dei minors da 0 a 3. Vedete poi altre informazioni nei sistemi Unix e Unix-like, i dispositivi sono visti di fatto come se fossero dei direttori file nel filesystem; cioè voi vedete questo dev/sda 1 2 3 che sono dei dischi ma li vediamo come se fossero dei file. Quindi il filesystem è come fosse un livello di astrazione che permette di nascondere anche dei dispositivi.

Block and Character Devices

Ora parliamo un attimo di dispositivi a blocchi (normalmente i dischi) e a caratteri. Le operazioni che si fanno sui dischi e sui dispositivi a blocchi sono le principali ma anche seek (spostati e vai direttamente a quel blocco, a quel cilindro, a quel settore). Possiamo andare in modalità raw che significa niente file system, si va direttamente ai settori su disco. I/O diretto o accesso tipo filesystem possono essere visti come due modalità alternative o due aspetti della stessa cosa: tenete conto che I/O diretto è spesso anche solo contrapposto ad I/O sequenziale; accesso tipo filesystem alle volte viene contrapposto al raw I/O. Quindi questa non è una vera e propria classificazione in cui ognuno è in mutua esclusione rispetto agli altri. E' anche possibile un accesso a memory mapped che è un modo per vedere i file come se fossero delle struct o come se fossero dei vettori in memoria (cioè si fa a = 7, b = uguale qualche cosa, e in realtà si va a lavorare su un file che è come se fosse associato a locazioni di memoria virtuale). Poi potete avere dei dispositivi che lavorano in DMA ma comunque stiamo parlando di dispositivi a blocchi. Se invece pensiamo a dispositivi a caratteri normalmente avete delle operazioni di get e put oppure getch e putch ma sono operazioni tipicamente sequenziali in cui si dice che il prossimo carattere da stampare è questo o il prossimo carattere letto è questo; quindi è chiaramente un'operazione che è più vicina a quello che è la tastiera oppure la visualizzazione di una riga sul video.

Network Devices

Nei sistemi operativi si ha spesso a che fare con dispositivi di rete. Normalmente il socket è quello privilegiato in Linux Unix e anche windows. L'operazione su socket è la select. Vi dico solo che a seconda di come, su quale sistema operativo e quali componenti stiamo realizzando, dietro ai socket o ai dispositivi di comunicazione di rete ci possono essere delle pipe o delle architetture a coda.

Clocks and Timers

Vediamo ancora in relazione all'I/O per completare la panoramica che non tutti gli I/O sono fatti per scambiare dati (input o output) ma ci sono degli I/O che o richiedono di essere temporizzati oppure l'informazione che in qualche modo viene gestita avrebbe un tempo; quindi clock e timer sono essenzialmente dei dispositivi che servono o a chiedere di essere svegliati dopo un certo tempo oppure periodicamente con un certo periodo oppure chiedono semplicemente delle generazioni di forme d'onda. Però normalmente abbiamo dei dispositivi programmabili che generano qualche cosa di temporizzato. La funzione `ioctl()` in Unix scopre molte di queste cose.

Nonblocking and Asynchronous I/O

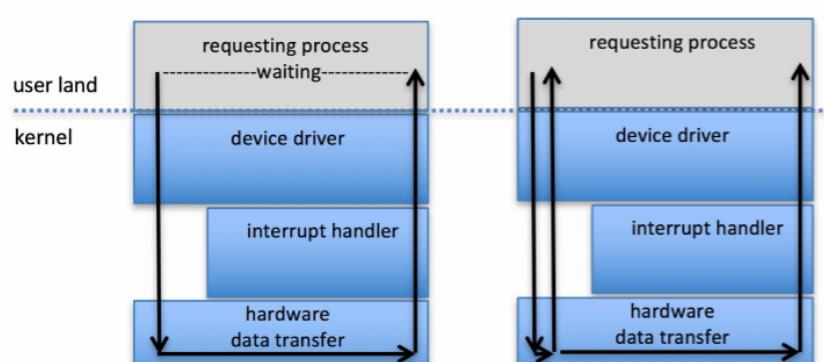
Ora vediamo ancora un aspetto di cui abbiamo parlato un attimo fa senza però approfondire, che è il cosiddetto I/O sincrono e asincrono, e vediamo le due accezioni: bloccante/sincrono e non bloccante/asincrono. Introduciamo questi termini per ricordare che bloccante significa sostanzialmente qualcosa di più generale mentre sincrono è più specifico:

- bloccante: un processo quando comanda qualcosa si blocca; un I/O è bloccante se il processo che lo esegue rimane fermo sospeso fin quando è completato
- non bloccante: un processo fa partire l'I/O e non si ferma ad aspettare; quindi la funzione di I/O viene chiamata, ritorna subito il risultato (nota: se ricordate le `read` e le `write` ritornano al programma chiamante quante cose sono riuscite a scrivere o leggere, in un I/O bloccante questo risultato sarà giusto, in uno non bloccante potrebbe non essere significativo quindi un I/O non bloccante è parziale, cioè parte, ritorna un risultato che dice quello che sono riuscito a fare ma non è conclusivo). I/O non bloccante può essere implementato in vari modi. Ad esempio usando dei buffer cioè il processo copia il dato in una parte in memoria e poi dice "adesso pensaci tu e io posso fare altro". Cioè ho parcheggiato un dato in una zona di memoria e poi parte qualche cosa. Si può lavorare in multithreading: faccio sì che un thread faccia I/O e l'altro thread faccia delle altre esecuzioni; è come dire me la gioco a livello manuale nel sincronizzare i vari threads e un thread non viene bloccato dall'esecuzione di I/O dell'altro thread. I/O non bloccante può ritornare un conteggio dei byte scritti o letti. La `Select` può essere l'operazione che può permettere poi eventualmente di sapere se il dato per cui non avevo aspettato l'esecuzione di I/O è stato effettivamente scritto letto. Ora questi sono tutte affermazioni vaghe che servono solo a dire attenzione che nel momento in cui si pensa a un I/O non bloccante sostanzialmente si dice "I/O vai avanti per conto tuo, io magari faccio altro, non aspetto. Se ho bisogno di sapere che l'I/O è finito te lo chiedo mediante una `select` o altro ma vuol dire che il processo va avanti nel momento in cui l'I/O è partito".
- Asincrono: è di fatto un I/O non bloccante "ben corredato", cioè una I/O non bloccante con un supporto in più per gestire il termine dell'I/O. Mettiamola così, chiamiamo asincrono un I/O che è non bloccante e per il quale nel meccanismo delle istruzioni è previsto implicito un modo ben strutturato per far sì che il programma se vuole aspettare o sapere che I/O è terminato può farlo. Tipicamente non è facilissimo, ci sono almeno due strategie che possono essere usate e sono mettersi in attesa su una qualche primitiva di sincronizzazione (quindi faccio partire un I/O asincrono, faccio altro e dopo un po' mi metto in attesa con una `wait` e questa primitiva di sincronizzazione sarà segnalata in modo automatico, non in modo manuale, dal completamento dell'I/O). L'altra possibilità è di specificare al sistema sostanzialmente una funzione di callback che verrà eseguita al termine dell'I/O.

In definitiva I/O asincrono è un I/O non bloccante più un meccanismo per gestire il termine, invece I/O sincrono è di fatto un sinonimo di bloccante.

Two I/O methods

Qui vediamo una rappresentazione lato user e lato kernel di I/O sincrono (bloccante) e I/O non



bloccante/asincrono. Da un canto (parte A) avete un processo che richiede l'I/O e aspetta il termine. Invece vedete sulla destra (Figura B) un I/O asincrono: il processo che richiede I/O ottiene un ritorno immediato prima che l'operazione sia completata ma esiste un meccanismo tale per cui al termine dell'I/O il processo potrà ricevere una segnalazione.

Vectored I/O

Il cosiddetto vectored I/O è semplicemente una variante di quanto abbiamo visto finora (una read particolare ad esempio), nel senso che permette di gestire un vettore di richieste di I/O invece di un singolo I/O.

Kernel I/O Subsystem

Osserviamo il sottosistema di kernel di I/O e vediamo che dal punto di vista del kernel, dopo che abbiamo visto prevalentemente la gestione per i dispositivi, esiste un problema di schedulazione: come abbiamo già visto per i dischi esiste da parte del kernel il problema che ha più richieste di operazioni su dispositivi di I/O; di per sé ci sono più richieste su dispositivi diversi ma anche più richieste sullo stesso dispositivo. Un modo per gestire queste richieste è di mettere una coda davanti ai dispositivi: significa una gestione cronologica di tipo fifo. In alcuni casi ci sono sistemi operativi che cercano di avere un criterio più di fairness: vuol dire cercare non necessariamente di rispettare un criterio cronologico ma di avere un qualche criterio di equilibrio oppure di correttezza o di suddivisione dei servizi tra i vari richiedenti. Un altro criterio può essere la qualità del servizio. In generale sapete che quando si parla di scheduling ci possono essere vari criteri.

Un altro aspetto che può essere estremamente importante dal punto di vista delle prestazioni e non solo è il cosiddetto concetto di buffering: s'intende un concetto che a prima vista può sembrare una perdita di tempo nel senso che si fa fare ai dati un passaggio in più; cioè quando si devono trasferire dei dati ad esempio in output da una locazione di memoria (pure user cioè di un processo) a un dispositivo di I/O si immagazzinano i dati in memoria prima di trasferirli al dispositivo. Questa azione può anche essere effettuata in un trasferimento eventuale tra due dispositivi però l'idea principale in questo momento è da processo a dispositivo o viceversa. Perché lo si fa? Nel caso di due dispositivi, può essere una motivazione legata alla velocità diversa dei due dispositivi ma anche quando si parla di trasferimento tra memoria e dispositivo il concetto è che sorgente e destinazione del trasferimento hanno velocità diversa; il principio si applica se trasferiscono da A a B e A è molto più veloce di B o, in pratica se l'operazione di trasferimento viene fatta coinvolgendo simultaneamente A e B uno dei due è pesantemente rallentato rispetto alle sue possibilità. Per cui siccome la memoria è tendenzialmente veloce si trasferisce da A alla memoria e poi A è libero, poi dalla memoria a B. A questo punto il dispositivo veloce trasferirà velocemente in memoria, il dispositivo lento trasferirà lentamente da memoria però ognuno va con la sua velocità. Un altro aspetto che può essere risolto mediante buffering è la differenza tra le dimensioni dei dati trasferiti tra A e B: cioè supponiamo che A voglia trasferire per blocchi di 1 kilobyte e B voglia trasferire per blocchi di 4 kilobyte, mettendo un buffer di almeno 4 kilobyte avremmo bisogno di 4 trasferimenti da A verso il buffer e poi un solo trasferimento dal buffer verso B.

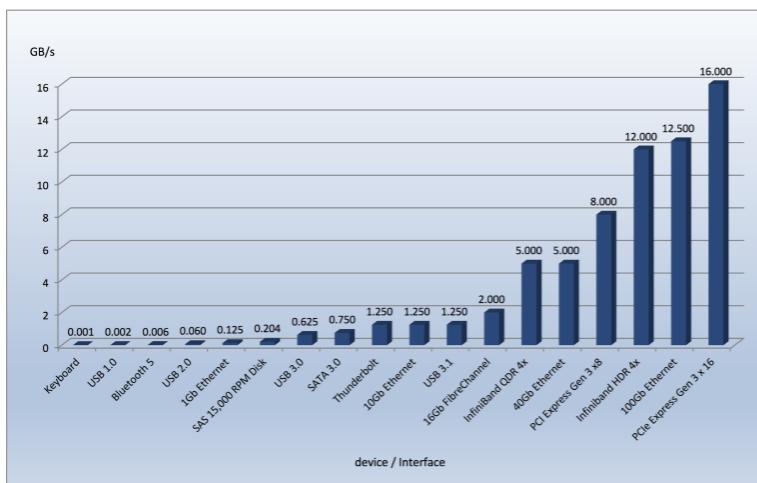
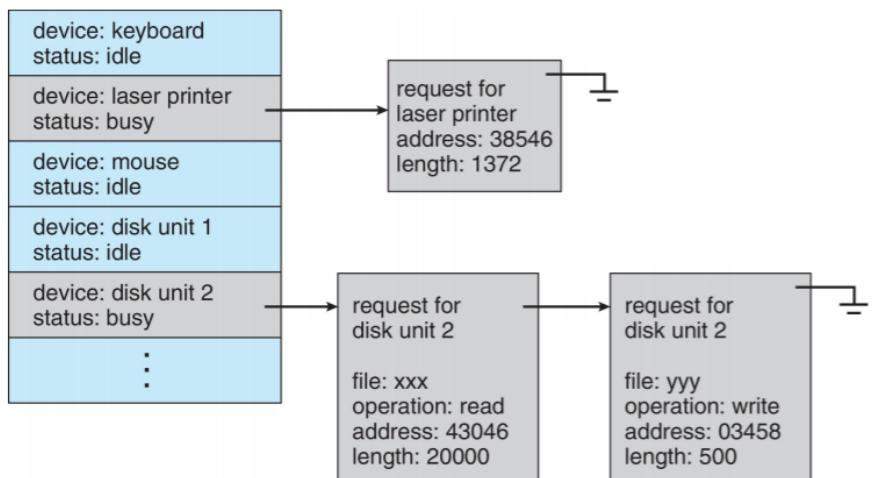
Inoltre supportare la cosiddetta copy semantics significa che è possibile che un'applicazione, un processo, richieda ad esempio di scrivere dei dati su disco e dopo aver fatto la richiesta di scrittura su disco voglia modificare i dati sostanzialmente senza aspettare la fine della scrittura su disco; è per fare in modo che indipendentemente dalla velocità di trasferimento sia possibile garantire che su disco vanno a finire i dati prima della modifica, cioè si intende che sia chiaro quali sono i dati che vanno su disco, una copia dei dati in un buffer di kernel da cui poi verranno trasferiti su disco prima di modificarli nella loro versione originale. L'idea è sostanzialmente che la bufferizzazione aiuta sicuramente a garantire che nel momento in cui c'è un duplicato è chiaro quando è stato fatto velocemente questo duplicato e di chi è la fotografia.

Detto ciò, bufferizzazione significa sostanzialmente "crea una copia da qualche parte (qui sarà in memoria kernel) del dato sia in ingresso che in uscita e sia da memoria verso il dispositivo e sia dal dispositivo verso memoria". Il problema tuttavia è questo: se da A copio verso il buffer la prossima volta che A vuole copiare nel buffer deve aspettare che si sia svuotato nel momento in cui il dato viene copiato sul dispositivo di I/O,

cioè il buffer diventa una risorsa condivisa. Il double buffering è un modo per velocizzare e ottimizzare leggermente queste operazioni, quindi il concetto di double buffering è sostanzialmente questo: supponiamo che ci siano due slot, uno etichettato kernel l'altro etichettato user (quello di kernel e user è solo il primo esempio cioè supponiamo che il kernel legga su un buffer e lo User scriva su un altro buffer oppure che kernel scriva e lo User legga dell'altro buffer, cioè mentre uno prepara un buffer l'altro usa l'altro buffer). L'altro concetto è: ci potrebbero essere diverse dimensioni di trattamento dei dati, cioè supponete che su un buffer venga preparata un'immagine un po' alla volta quindi in modo incrementale (per righe per esempio) e sull'altro c'è già un'immagine completa che viene generata. Comunque, buffer 0 viene progressivamente riempito e buffer 1 è già pieno e viene progressivamente svuotato. Con il double buffer ad un certo punto i due buffer si scambiano ruolo, cioè quello che concettualmente sarebbe "riempi buffer 0 e poi usa buffer 1, quando hai finito di usare buffer 1 (che non serve più) e quando buffer 0 è pieno (che è pronto) invece di fare shift (cioè trasferisci buffer 0 in buffer 1 e ricomincia) si scambia i ruoli dei due buffer".

Device-status Table

Il kernel può gestire sostanzialmente vari dispositivi mediante delle code. C'è un tentativo di rappresentare una tabella in cui c'è lo stato di vari dispositivi, su ognuno dei dispositivi c'è un'informazione di idle o busy; sono rappresentate delle liste che sono sostanzialmente delle code di richieste in attesa. Non mi soffermo più di tanto su questo perché è semplicemente una visualizzazione di una potenziale struttura dati.



Vedete una rappresentazione in un unico grafico delle velocità di vari dispositivi che serve appunto a dimostrare quanto ci siano grosse differenze di prestazioni. Sulla sinistra una tastiera molto lenta; poi notate le velocità di USB 1.0, 2.0, 3.0 e 3.1: quello che vedete ad esempio è che tra USB 2.0 e USB 3.0 c'è un ordine di grandezza di cambio di velocità. Avete Ethernet 10 Gigabit e via dicendo. Notate, prendendo come riferimento appunto dispositivi abbastanza noti quali Ethernet oppure USB 3.0 e 3.1, che sulla destra andiamo verso i bus usati all'interno dei pc e vedete comunque delle differenze di prestazioni notevoli. Questa figura serve più che altro a farci vedere come ci siano grosse differenze tra i vari dispositivi.

Kernel I/O Subsystem

Oltre al buffering, parliamo di altre caratteristiche che impattano sulle prestazioni. Parliamo di **caching**: in generale si intende una replica ridondante di un'informazione, che dovrebbe stare in un dispositivo o in una qualche parte di memoria, ma se ne fa una copia su un dispositivo più veloce. Quindi in quanto copia è allineata oppure non è valida quindi va allineata rispetto all'originale; non è obbligatoria ma può essere importante per le performance. Il principio del caching sarà sempre “prima guarda nel dispositivo veloce se trovi il dato prendi direttamente di lì, se no passa al dispositivo lento”. Ovviamente alle volte il caching sarà accoppiato con buffering o andrà gestito con forme di buffering in qualche modo. Lo spooling è una forma di accodamento o di parcheggio del dato però associato a un dispositivo di uscita: lo **spooling** è tipicamente usato per le stampanti. Se il dispositivo cui si fa riferimento può servire solo una richiesta alla volta (la stampante proprio questa), allora si usa il cosiddetto spooling che non è nient'altro che una gestione di parcheggio del dato e gestione di una coda di richieste. Poi la **device reservation** è invece una caratteristica importante per dispositivi che non possono essere condivisi: quindi se c'è bisogno di accesso esclusivo bisogna avere una specie di gestione in mutua esclusione del dispositivo stesso, perciò bisogna fare attenzione a potenziali forme di deadlock.

Error Handling

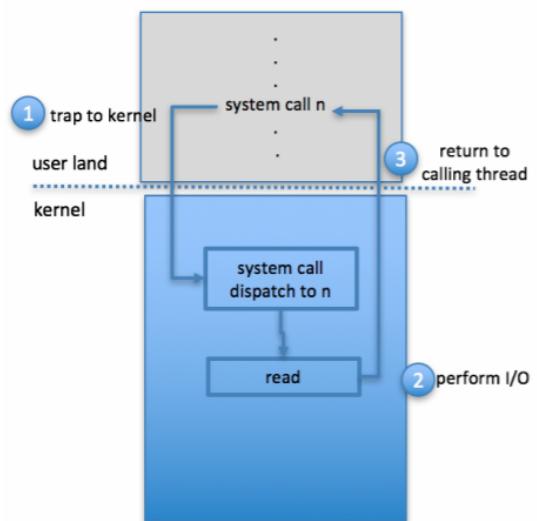
Parliamo di gestione dell'errore. Finora abbiamo parlato di prestazioni, ora vediamo cosa succede se ci sono degli errori e che errori bisogna attendersi. Il sistema operativo può essenzialmente cercare di reagire a errori di lettura su disco, dispositivo non disponibile (esempio rete che è giù), problemi su letture e su scritture o altro tramite tentativi; cioè riprova un po' di volte e vedi se era un problema transitorio (ad esempio la rete che non funziona o altre forme di eventuali errori che possono essere visti come “se aspetti un po' magari poi le cose vanno a posto”). Alcuni tentativi cercano poi anche di fare un po' di previsioni e di non andare troppo alla cieca cercando di indovinare usando le statistiche e non solo per tentativi alla cieca. Tuttavia in generale c'è la possibilità di ritornare dei codici di errori quando una richiesta di I/O fallisce e ci sono degli error log (file su cui sono registrati e sono poi leggibili visibili le tracce degli errori che sono avvenuti). E' chiaro che i dispositivi di I/O possono eventualmente dare degli errori, quindi diciamo che gli errori esistono e possono essere trattati in un certo modo.

I/O Protection

L'altro problema abbastanza connesso è quello della cosiddetta protezione: un processo user che è per definizione non privilegiato potrebbe o inavvertitamente o volontariamente in modo malevolo cercare di effettuare operazioni non corrette cioè operazioni illegali e tenendo conto che tutte le operazioni di I/O sono in qualche modo privilegiate (nel senso che per fare un'operazione di I/O bisogna avere il controllo sul dispositivo e quindi bisogna essere nel kernel), a questo punto c'è una specie di dicotomia: da un lato si vorrebbe che lo user non potesse fare cose proibite e dall'altro per usare il dispositivo di I/O ha bisogno di averne dei privilegi. Si arriva alla conclusione che l'I/O deve essere effettuato mediante system calls. Riguardo la mappatura dei dispositivi, se sono mappati in memoria o no, deve esistere una protezione per le locazioni di memoria o i registri mappati in memoria che fanno riferimento all'I/O. Comunque, in parole povere bisogna fare attenzione al fatto che bisogna fornire una protezione nei confronti delle operazioni che sono potenzialmente illegali.

Per questo si usano le System calls. Qui rivediamo uno schema di come funziona una system call che effettua un'operazione di I/O.

Il significato di questa figura è abbastanza semplice: vedete al punto uno si effettua la system call che è di fatto una trap nel kernel. Passando al kernel c'è una system call alla funzione syscall che va a determinare il numero della System call (ad esempio in questo caso la syscall read); la



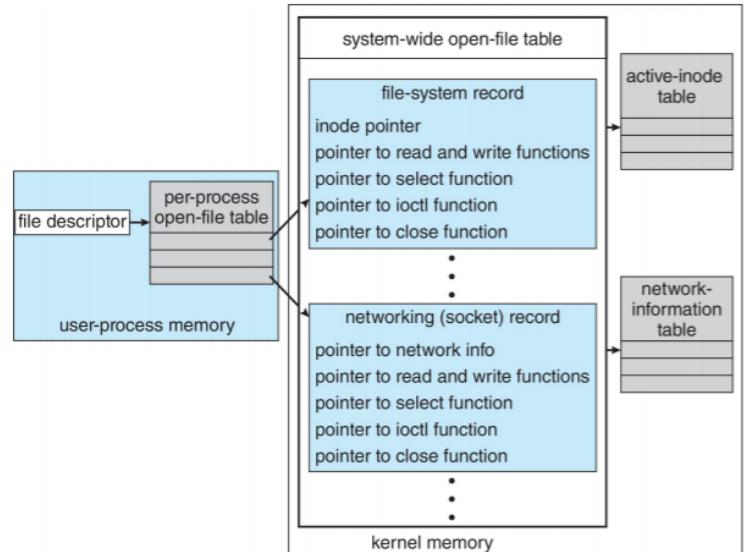
sysread effettua l'I/O e ritornerà mediante il protocollo di ritorno dalla trap al processo richiedente.

Kernel Data Structures

E qui vedete un tentativo di riassumere dal punto di vista qualitativo le strutture dati che il kernel adotta per gestire l'input/output. In pratica all'inizio vi dice che il kernel mantiene delle informazioni di stato per tutti i dispositivi di I/O che in pratica sono: open file tables, tabelle per i dispositivi di connessione di rete, stato dei dispositivi a caratteri; insomma delle tabelle che mantengono lo stato e un po' di altre informazioni sui dispositivi. Poi ci sono molte strutture dati complicate che servono più a gestire le prestazioni per i buffer, l'allocazione di memoria, dirty blocks e altro. Quindi sono strutture dati più dinamiche, più orientate a gestire le prestazioni e le azioni dei dispositivi. In alcuni sistemi operativi si usano metodi di tipo object oriented e anche message passing: s'intende sostanzialmente che, un producer o un consumer (oppure un client e un server), invece di gestire delle informazioni centralizzate in memoria si gestisce tramite messaggi o strategie di messaggistica. Ad esempio, il sistema Windows usa message passing per gestire informazioni di I/O che vanno dall'user al kernel, cioè invece di dire che queste transitano in un buffer condiviso, in qualche modo si gestirà dei messaggi; spesso gestito con delle code fifo che vengono realizzate per parcheggiare i messaggi e sono di fatto connessi a dei canali di comunicazione tra i due attori che si scambiano i messaggi. Il messaggio poi può passare tramite stadi e può subire delle modifiche successive. Un po' come quello che succede, se avete un po' di familiarità con nei protocolli di rete quando un pacchetto transita tra vari livelli dei protocolli di rete e ogni livello ci aggiunge o toglie o modifica delle informazioni. Ora tutto questo ovviamente andrebbe visto in termini di vantaggi e svantaggi nel confrontare più soluzioni ma non è questo il contesto in cui possiamo andare a fondo rispetto a queste informazioni.

UNIX I/O Kernel Structure

Diamo un'occhiata alla struttura del kernel Unix per quanto riguarda l'input/output. I dispositivi di I/O sono nascosti sotto al filesystem. C'è una memoria che fa riferimento al processo utente (per process open file table). Ora, in questa figura in realtà sembrerebbe che la per process open file table sia nella memoria user del processo; in realtà è associata al processo ma è una struttura dati del kernel, accessibile dal PCB. In ogni caso c'è una tabella dei file aperti del processo che fa riferimento alla system wide open file table; per questi file ci sono dei filesystem record che conterranno inode (vnode) a seconda che siamo su un filesystem reale o virtuale, e questi ai inode/vnode contengono i puntatori alle funzioni di lettura/scrittura, alle operazioni che si possono eseguire e dipendono dal dispositivo. Questi dispositivi in qualche modo faranno riferimento a delle operazioni; però come vedete qui, che che si tratti di un disco su cui effettivamente avremo delle letture/scritture su un file o che sia un socket su cui avremo altri tipi di operazioni, l'interfaccia di alto livello è in pratica la stessa.



Power Management

Anche la gestione del power cioè dell'alimentazione, pur non essendo trasferimento dati, (quindi non si tratta effettivamente di operazioni significative da parte della CPU) è comunque considerata una competenza del sottosistema di I/O e quindi CPU, il sistema operativo e il kernel che controlla le operazioni vengono considerati responsabili di gestire in modo efficace tutto ciò che ha a che vedere con il power (che non è soltanto on/off ma anche la misura della temperatura, il sistema di raffreddamento, l'uso della potenza in situazioni di batteria/alimentazione a rete e così via). Quindi in sostanza la complessità del sottosistema di gestione del power è importante, ed è tanto più importante quanto più i dispositivi sono alimentati a batteria e hanno bisogno di risparmio energetico. Questo anche in contesti di tipo cloud, di tipo Service e non necessariamente mobile computing.

Come caso particolare vediamo che Android implementa una gestione della potenza a livello di singoli componenti. Cioè cerca di capire le relazioni tra singoli componenti del sistema, cerca di rappresentarne la topologia (cioè di avere una visione della topologia dei dispositivi che significa a sapere dove sono e come sono connessi tra di loro). Ad esempio, vediamo che c'è il System bus, subsystem, flash, memoria USB e così via. I device driver (le componenti del kernel che gestiscono i dispositivi) mantengono traccia dello stato dei vari dispositivi, quando sono usati e in pratica sono in grado di dire al sistema "dispositivo inutilizzato"; cioè i device driver sanno quando un dispositivo non è in uso per trasferimento dati e quindi può essere spento. Questa è già ad esempio una situazione in cui un sistema intelligente di gestione del power può quanto meno usare la strategia di spegnere ciò che non è in uso. Quando tutti i dispositivi in un ramo dell'albero dei dispositivi sono inutilizzati allora si spegne tutto il sottosistema, quindi c'è quantomeno un tentativo di identificare singoli dispositivi e singole aree singole zone di dispositivi cercando di spegnere e quindi di ottimizzarne il consumo. A questo punto c'è bisogno di strutture dati di sincronizzazione, di lock, che devono gestire la possibilità di non spegnere un dispositivo quando è in possesso da parte di qualche processo/componente del kernel. Non voglio andare a fondo più di tanto ma semplicemente dirvi che esiste il problema e va in qualche modo affrontato della gestione del power. I sistemi moderni usano un'interfaccia avanzata di configurazione del power che in qualche modo collabora anche con parti hardware che servono appunto a gestire questo.

Kernel I/O Subsystem Summary

Vediamo il sommario finale. In sostanza il sottosistema di I/O coordina svariati servizi per le applicazioni che hanno bisogno di I/O:

- gestione dei nomi di file dispositivi, che tendenzialmente come avete visto, nel sistema Unix vengono fatti vedere come se fossero parte del filesystem.
- c'è il controllo degli accessi ai file e ai dispositivi
- si controllano le operazioni che possono essere fatte e che vengono fatte
- c'è l'allocazione dello spazio per il file system
- c'è l'allocazione dei dispositivi
- ci sono tecniche di buffering, caching e spooling per le stampanti
- si parla di schedulazione di I/O
- c'è un monitoraggio dello stato dei dispositivi mediante delle tabelle. C'è un problema di gestione dell'errore e un eventuale recovery dell'errore (tentativi o di tentativi intelligenti)
- ci sono problemi legati alla configurazione dei device driver: è chiaro che un device driver, essendo un software che deve controllare l'hardware, dovrà in qualche modo capire qual è l'hardware capire se è cambiato e avere una sua impostazione
- poi abbiamo parlato anche di gestione del power

I livelli più alti del sottosistema di I/O dovranno sicuramente avere un'interfaccia uniforme, i livelli più bassi saranno più vicini ai dispositivi ovviamente.

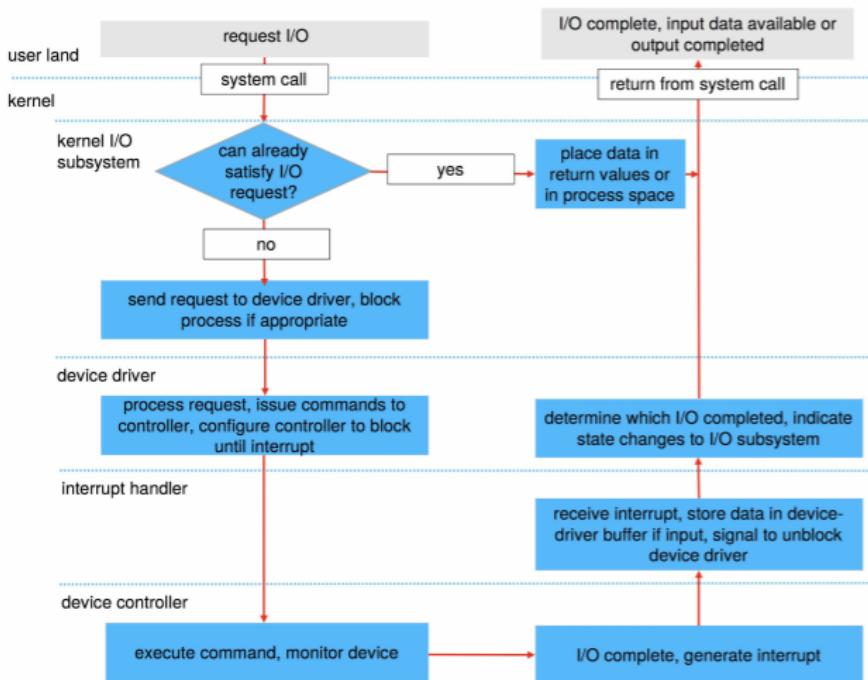
Transforming I/O Requests to Hardware Operations

Vediamo ora i passi per eseguire un'operazione dalla richiesta di I/O fino all'operazione hardware in sé stessa. Quindi supponiamo ad esempio una lettura di un file da disco per un dato processo.

- 1) determinare quale è il dispositivo che contiene il file, quindi passare dal nome del file al dispositivo o dal file descriptor al dispositivo
- 2) tradurre da nome a dispositivo
- 3) bisogna leggere fisicamente i dati dal disco in un buffer
- 4) rendere il dato disponibile al processo che ha richiesto il dato stesso
- 5) ritornare

Tutto questo detto a parole sembra sensato cioè si localizza il dispositivo, si leggono i dati dal dispositivo e li si mette in un buffer e si ritorna al processo richiedente.

Life Cycle of an I/O Request



Tutto questo lo vediamo rappresentato qui tenendo conto un pochino delle ultime novità su buffer di cui abbiamo parlato.

Ora guardiamo questa figura partendo a livelli, cioè il livello in alto è la user land, il livello in mezzo è il kernel e scendendo ci si avvicina fino all'hardware. Da un lato vediamo la richiesta di I/O, colonna di sinistra, e colonna di destra vediamo l'I/O completato. La richiesta di I/O scatenerà sicuramente una system call infatti vedete che nella colonna di sinistra in alto avete una system call sotto la richiesta di I/O. A questo punto la prima domanda che può

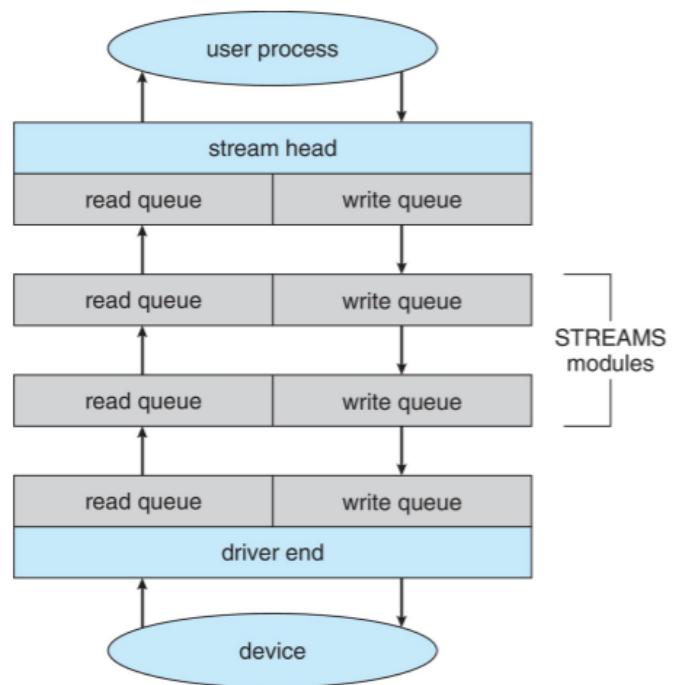
essere effettuata è: può questa system call essere già soddisfatta? Questa è la vera domanda che potrebbe trovare una risposta se ci fossero o se si utilizzassero efficacemente dei sistemi di caching oppure forme di buffer o read ahead o cose simili; il discorso è questo: se la system call dice "il dato che tu vuoi leggere dal dispositivo ad esempio c'è già in memoria, metti il dato nella destinazione (guardate a destra di questo Yes che sta a metà) e ritorna direttamente. Quindi avete sostanzialmente una specie di bypass di tutto ciò che sta sotto che consiste con la possibilità che la richiesta di I/O sia immediatamente soddisfacibile perché di fatto i dati sono già in memoria. In generale invece se la risposta a quel test fatto in alto a sinistra è no, occorre effettuare l'I/O vero e quindi si scende al vero sottosistema di I/O, cioè si manda la richiesta al device driver, si manda wait al processo se necessario (dipende dal fatto che l'I/O sia sincrono o asincrono; in questo caso sembra una rappresentazione più connessa all'I/O sincrona ma la si potrebbe anche vedere in altri termini) e supponiamo I/O sincrono. A questo punto il device driver processa la richiesta, manda i comandi al controller (passa direttamente al di là dell'interrupt handler e manda quindi il comando al device controller) che li esegue e a un certo punto l'I/O sarà completato. Viene generato un interrupt (in basso a destra); l'interrupt handler riceve l'interrupt, immagazzina il dato nel buffer del device driver (la sua destinazione se input, oppure segnala semplicemente di sbloccare il device driver se è un'operazione di output) e passa al controllo al device driver (o lo sveglia). Questo capisce che l'I/O è stato completato, indica un cambiamento di stato al sottosistema di I/O e si ritorna dalla system call. Quindi gli

attori sono il controller, interrupt handler e device driver. L'idea però è quella che ci sia il sottosistema di I/O che fa la richiesta, il processo può venire bloccato e si deve attendere che ci sia un interrupt che sveglia (quindi la parte in alto viene bloccata e sarà risvegliata alla fine di tutto quanto). Anche il device driver di per sé verrà bloccato e si metterà in attesa che arrivi l'interrupt.

STREAMS

E vediamo alcuni esempi concreti. Stream è un esempio di sistema di comunicazione tra processo user level e un dispositivo a caratteri in Unix system file. E' un meccanismo che assomiglia abbastanza come concetto ai protocolli di rete e si basa su un canale di comunicazione full duplex (half duplex e full duplex fanno riferimento al fatto che un canale di comunicazione sia mono direzionale o bidirezionale). Stream consiste di una stream head (un'interfaccia verso il processo utente) e un driver end (un'interfaccia verso il dispositivo) e livelli intermedi tra di loro; cioè invece di mettere direttamente i due lati a contatto l'uno con l'altro, ci sono dei livelli intermedi come se fossero dei layer a livello di protocolli di rete. Ogni modulo intermedio contiene una read e write queue; il principio è che le code fifo riescano a sostanzialmente ammortizzare le differenze di prestazioni e le richieste multiple che arrivano da un lato a fronte di servizi ritardati o velocizzati dall'altro lato. Anziché parcheggiare in buffer si usa una strategia message passing, quindi dal lato del processo utente è sincrono, ma internamente cerca di ottimizzare le prestazioni usando meccanismi asincroni e di adattarsi alle differenze di prestazioni dei vari dispositivi.

Questa rappresentazione di Streams si basa sul concetto che il processo utente dal lato dello stream head, vede due canali direzionali: uno dei dati che arrivano al processo utente, uno dei dati che vanno dal processo utente verso stream end. Potreste provare a connettere direttamente i due lati e tutto funzionerebbe. Stream prevede sostanzialmente delle code di lettura e delle code di scrittura che servono in pratica a disaccoppiare prestazioni dei due estremi in alto e in basso.



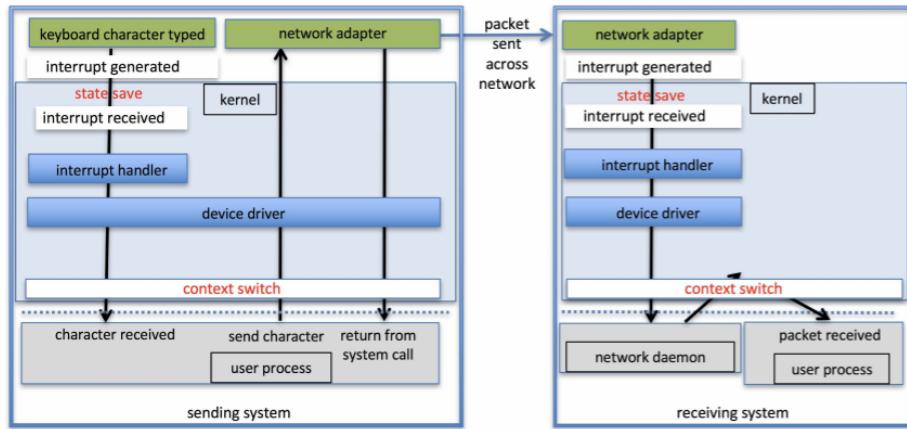
Performance

Dal punto di vista della performance possiamo dire che globalmente l'I/O impatta moltissimo sulle prestazioni. Diciamo che l'I/O chiede alla CPU di eseguire i device driver, il codice di I/O del kernel che è quello che causa context switch a causa delle interruzioni delle, spostamento di dati. Quindi impatta tantissimo sulle prestazioni di un sistema. Tenete poi conto che ad esempio ci sono sistemi che sono pesantemente orientati a effettuare quasi unicamente I/O, perciò è importante.

Intercomputer Communications

Se vogliamo parlare un attimo di rete, le comunicazioni tra computer sono rappresentate qui facendo vedere un processo utente da un lato (sending system) e un altro processo (receiving system) compresi i sottosistemi che in qualche modo sono coinvolti in questa comunicazione. L'importante è che c'è il pacchetto spedito in rete, i due Network Adapter che sono i dispositivi di rete sui due computer, la keyboard

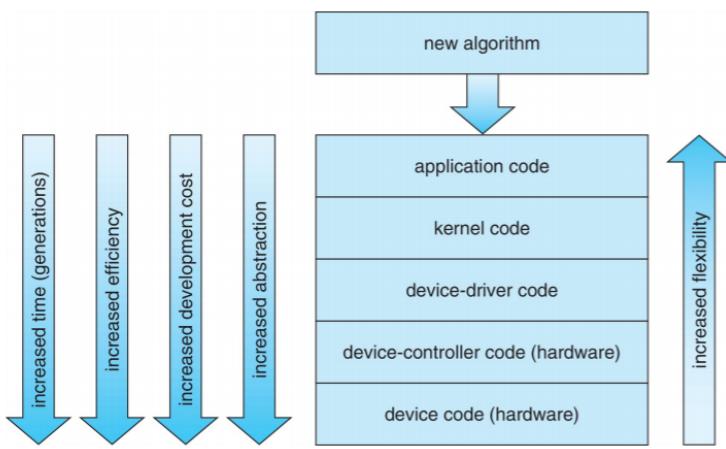
(che supponiamo che sia ad esempio un generatore di dati), c'è il device driver da entrambe le parti, ci saranno i dispositivi che generano interrupt, context switch; da un lato ci sono dei caratteri ricevuti, dall'altro dei caratteri spediti, ci sono delle system call, pacchetto ricevuto e così via. Cioè avete una complessità di operazioni tra sottosistemi e tra livelli diversi di gestione che coinvolgono kernel, i dispositivi di I/O e i processi utente.



Improving Performance

Per migliorare le prestazioni in definitiva, al di là di fare tutti i componenti fatti bene, di usare tutto ciò che abbiamo visto finora, è ovvio che sarà importante ridurre i context switch, che sarà importante ridurre le copie di dati, ridurre gli interrupt e usare trasferimenti di dati sufficientemente grandi, i controller intelligenti, ridurre il polling, usare DMA, usare dispositivi hardware fatti bene ovviamente e poi trovare un ragionevole bilanciamento tra CPU, memoria, bus e prestazioni di I/O (non è mai equilibrato un sistema in cui c'è un sottosistema decisamente più performante in senso relativo rispetto agli altri) e alle volte cercare di spostare al kernel delle azioni che sarebbero fatte dai processi utente. Insomma qui si sta dicendo che si potrebbero cercare di rendere più efficienti delle azioni e trasferirli come competenza dal lato utente al lato kernel. In definitiva minimizzare o ridurre le operazioni costose e cercare di bilanciare un sistema migrando le operazioni critiche nelle parti più efficienti cioè dal lato kernel.

Device-Functionality Progression



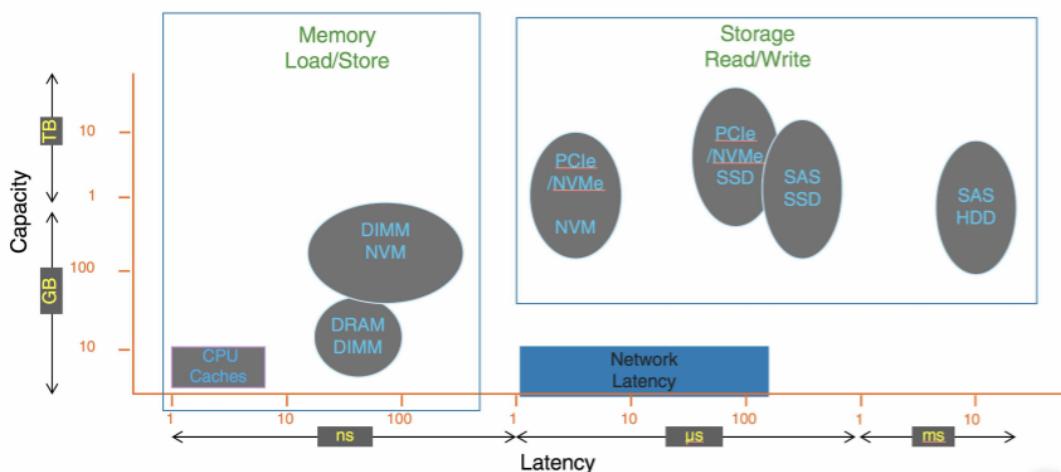
Ora se vogliamo riassumere un pochino questi concetti possiamo vedere similitudini a quello che succede quando si fa un nuovo algoritmo: ci si inventa una nuova forma di trasferimento dati e tendenzialmente la si comincia a realizzare a livello di codice applicativo; prima o poi se è veramente qualcosa di importante si andrà direttamente a implementarla in hardware. Questo è successo ad esempio a certi codici o certi protocolli di crittografia o di protezione dei dati che oggi sono direttamente gestiti dall'hardware. Cioè in sostanza all'inizio si fa software di alto livello poi se diventa critico lo si migra nel codice di kernel poi diventa magari parte del device driver poi diventa parte del device controller o addirittura del dispositivo hardware. Questo spinge una certa competenza dall'alto in basso man mano che passa tempo e ci sono generazioni successive di questo codice oppure di questo tipo di azione che si realizza. Passando dal software all'hardware, o comunque scendendo di livello, si migliora l'efficienza ma costa di più cioè lo sviluppo di

sostanza all'inizio si fa software di alto livello poi se diventa critico lo si migra nel codice di kernel poi diventa magari parte del device driver poi diventa parte del device controller o addirittura del dispositivo hardware. Questo spinge una certa competenza dall'alto in basso man mano che passa tempo e ci sono generazioni successive di questo codice oppure di questo tipo di azione che si realizza. Passando dal software all'hardware, o comunque scendendo di livello, si migliora l'efficienza ma costa di più cioè lo sviluppo di

qualcosa che passa dai livelli più alti ai livelli più bassi e più costoso. E' chiaro che più siamo verso software e più c'è flessibilità, il contrario il viceversa. Questo mi fa vedere come non necessariamente una certa azione è congelata a un certo livello di astrazione.

I/O Performance of Storage

Guardando il grafico: i dispositivi di memoria che hanno latenze che stanno nei nanosecondi, CPU e cache che sono quelli più rapidi siamo sull'ordine di pochi nanosecondi ma in termini di memoria sono scarsi (fino ai 10 gigabyte), ci sono le memorie RAM e siamo in intervalli che vanno fino alle centinaia di gigabyte, andando avanti verso destra passiamo a latenze che arrivano ai microsecondi con i dispositivi di rete, poi SSD e così via. Gli HDD gli HDD in fondo cioè con prestazioni e velocità che vanno dall'ordine dei microsecondi fino ai millisecondi. Però possiamo raggiungere delle capacità che vanno attualmente fino ai 10 terabyte anche un po' di più. E questo termina questo capitolo sull'I/O.



11. OS161 – I/O

Vedremo un tentativo di analisi della gestione dei dispositivi di I/O in os161, con due scopi: capire meglio come funzionano le operazioni di basso livello in os161 – almeno per quanto riguarda la console – e cercare di avere un'implementazione concreta delle operazioni di I/O. Iniziamo osservando che il file system in os161 consta di due livelli: un layer cosiddetto “virtual file system” che si trova in kern/vfs, in cui c’è un livello del file system indipendente dal file system realmente utilizzato, quindi un ambiente in cui si possono eventualmente aggiungere nuovi file system; il vero file system è invece in kern/fs, cioè una cartella in cui si trova l’implementazione del simple file system (sfs). In realtà quello che voi avete sempre usato è emu file system, che ha un layer di implementazione del file system assolutamente trasparente, nel senso che rimanda tutto quanto direttamente al dispositivo host.

Passiamo ora a vedere la parte di I/O di os161 e diciamo subito che, in pratica, bisognerebbe imparare il funzionamento hw del bus del sistema (basato su microprocessore MIPS) e sul cosiddetto “lamebus”, che è un’architettura di sistema molto semplice, basata su 32 slot, a ognuno dei quali è associata una area di indirizzamento di dimensione fissa, ed ogni regione, fra queste 32, è mappata nello spazio di indirizzamento fisico in un secondo schema fisso. Non c’è DMA controller, per cui tutti i trasferimenti sono fatti passando dalla CPU. [os161-io pag.3] Qui ho riportato semplicemente una parte iniziale di lamebus.h, che fa vedere la numerazione dei dispositivi per come sono definiti in .c [/]. Una delle 32 regioni ha uno spazio di indirizzamento di 64kB, quindi sono 2MB totali (64kB x 32). Lo spazio di indirizzamento fisico di questa regione parte dalla costante LAMEBASE, poi ci possono essere differenze a seconda del processore. Tra i 32 dispositivi c’è il controller del bus stesso, che è nello slot 31 (l’ultimo), in cui i 64kB dello spazio di indirizzamento sono divisi in due: la parte bassa, ovvero i primi 32kB, sono divisi in 32 regioni di

configurazione da 1kB, una per ogni slot (cioè, il controller del bus controlla tutti gli slot del bus, compreso se stesso); la parte alta è divisa in 32 regioni di controllo da 1kB, una per ogni CPU (quindi non saranno usate tutte, dipende da quante CPU abbiamo).

Di tutte queste cose non scenderemo nei dettagli. [os161-io pag.5] Vi ho riportato qui i puntatori alla documentazione online di os161 in cui potete vedere tutte queste info, ma non serve conoscere tutto [/].

[00:09:03 – 00:11:20] Visione delle pagine web di sopra [/]

Più interessante è l'analisi dei dispositivi di I/O. Abbiamo visto una prima mappatura dei dispositivi quando abbiamo parlato di lamebus (TIMER è definito uguale a 2, DISK uguale a 3, etc). Se guardate nella descrizione online dei dispositivi di I/O vedete le stesse cose. Di ognuno di questi si può reperire la descrizione: vediamo quella della Serial console (che coincide con tastiera o video in modalità testuale). Notiamo che la descrizione coincide con quella fornita sulle slide (pag.6) ed è la console che avete usato quando si è trattato di realizzare la prima versione delle system call di input-output. Questa console è una semplice interfaccia in cui, scrivendo nel primo dei registri di questo dispositivo il carattere che si vuole, esso viene stampato. Leggendo da questo registro si ottiene l'ultimo carattere che è stato stampato da tastiera, non c'è nient'altro che viene gestito qui. Inoltre, l'interfaccia viene gestita in modo sincrono (una write deve essere completata prima che se ne possa fare una nuova).

Vediamo nella slide, a sinistra, il device id e le revisioni (poco importanti), a destra invece i registri. Il primo è il registro su cui si scrive e si legge, il secondo e il terzo sono quelli di Interrupt ReQuest di scrittura/lettura, l'ultimo è in realtà inutilizzato.

Di questa console seriale voi potete dare un'occhiata al file lser.c (pag.7), che è il driver dell'interfaccia seriale, che inizia descrivendo i registri mediante delle define – da notare che sono gli stessi indirizzi descritti nella slide precedente – e in seguito definisce delle parole di configurazione – da notare che sono tutte potenze di 2, poiché si tratta di settare ad 1 UN SOLO bit tra tutti quelli della parola di controllo (0001, 0010, 0100, etc). Nel resto del file ci sono delle funzioni di gestione dell'interrupt.

In lser.h (pag.8) invece si trova una descrizione di una struct di controllo software, in cui ci sono dei puntatori a ciò che serve per far funzionare quest'interfaccia seriale (è la struct che contiene le informazioni utili al driver perché funzioni). Poi, sotto la struct, ci sono i prototipi delle funzioni che servono.

Nella pagina web di prima ci sono informazioni simili.

Vediamo come si colloca questa interfaccia seriale nell'ambito delle operazioni di I/O come abbiamo visto nel capitolo 12. Per prima cosa rivediamo il ciclo di vita di una richiesta di I/O (pag.9). Vi faccio notare che, da un canto, sulla colonna di sinistra vedete la richiesta di I/O, dall'altro, su quella di destra vedete la parte di completamento dell'I/O. Possiamo vedere lo user land in alto (con la system call), poi la parte kernel, che si divide in tre parti: il kernel I/O sub system, che risponde direttamente alla system call ed è indipendente dal dispositivo, il device driver che dipende dal dispositivo, e infine l'interrupt handler. Noi daremo un'occhiata, per quanto riguarda l'interfaccia seriale, a due operazioni (che sono quelle che voi avete utilizzato all'interno delle system call per write e read): putch() e getch(), che scrivono e leggono sull'interfaccia seriale. [os161-io pag.10] In questo contesto siamo già in basso, siamo al di sotto della system call, perché, se vi ricordate, la syswrite() e la sysread() arrivano a chiamare putch() e getch() (almeno per quanto riguarda la console, ovvero per i file descriptor 0,1 e 2). Qui vediamo una prima sezione in alto che riguarda console.c e console.h, cioè il kernel I/O subsystem, dove console.c è un file indipendente dal dispositivo, poi vediamo una seconda sezione più in basso che riguarda lser.c e lser.h, cioè i driver dipendenti dal dispositivo. Poi abbiamo, sotto, l'hardware, con a destra il vero e proprio handler dell'interrupt. Le frecce stanno a significare che una certa funzione, al suo interno, chiama quella puntata dalla freccia. In dettaglio: la putch() (chiamata all'interno del kernel), cerca la console mediante una variabile globale chiamata the_console e redirige la putch() a putch_intr(), che oltre al carattere da scrivere riceve il puntatore alla console. Questa putch_intr() aspetta, mediante un semaforo, di poter effettuare l'operazione, quindi aspetta di essere svegliata dalla catena di interrupt, non fa polling; poi chiama la funzione cs->cs_send(), che è stata mappata alla funzione lser_write(), che è la funzione driver propria del

dispositivo seriale. Se scendiamo a livello più basso, vediamo che la lser_write() riceve come parametro sia il carattere che il puntatore a dati di tipo lser_softc, struct definita in lser.h. Questo puntatore rimanda a tutte le informazioni sulla console seriale, quindi è sufficiente reperire dalla struct ciò che serve per fare una scrittura sul registro opportuno, chiamando la funzione bus_write_register() e passando opportunamente i parametri. Terminata l'esecuzione, la console seriale scatenerà un interrupt che avvisa che la console è pronta per una nuova scrittura: la funzione handler(), a questo punto, chiamerà lser_irq(), che al suo interno capirà se si tratta di un interrupt read o write, chiamerà sc->ls_start(), che viene mappata con con_start(), la quale avrà il compito di far ripartire il ciclo di write agendo sul semaforo della scrittura (attenzione: sono presenti due semafori, uno per la lettura – che non viene usato qui – ed uno per la scrittura). [/]

[00:27:07 – 00:35:31] Dimostrazione in debug su os161 [/]

[os161-io pag.11] La getch() è un'altra funzione presente all'interno di console.c, che serve a fare input di un carattere dalla console seriale. Mentre per la putch() si ha un'azione di write diretta sulla sinistra della slide 10, per l'input viene gestito tutto in interrupt, cioè: la getch() ridirige la chiamata a getch_intr(), la quale si sincronizza sul semaforo della lettura e non fa altro dopo, se non trattare il dato che è stato ricevuto (lettura dal buffer). La console dall'altra parte, quando si pingerà un tasto, farà scattare un interrupt che azionerà gli stessi meccanismi descritti per la putch(), con la differenza che lser_irq chiamerà la funzione sc->ls_input() e, in cascata, con_input(). Quest'ultima, una volta scritto il carattere sul buffer che ha in comune con getch_intr(), agisce sul semaforo della lettura facendo ripartire il ciclo. [/]

[00:38:08 – 00:45:41] Dimostrazione in debug su os161 [/]

12. OS161 – File

[os161-file pag.2] Partiamo dall'osservazione che quello che avete implementato voi (open, close, read, write) è un sottoinsieme delle operazioni che si fanno in relazione a un file system, e qui vedete un estratto dei prototipi che si trovano in unistd.h (che non contiene solo materiale sul file system ma anche altro). E' da notare che oltre alle normali funzioni che avete già visto ci sono lseek() e dup2(). Come si vede nei commenti, la open() riceve più parametri di quelli a cui siamo abituati (quattro anziché due), ma di fatto possiamo ignorare il terzo ed il quarto, se non c'è bisogno di altre parametrizzazioni riguardo a sicurezza e cose simili. [/]

[os161-file pag.3] Rivediamo quello che è l'interfaccia system call per il file system, per ricordarci che a livello user abbiamo delle applicazioni che si interfacciano al kernel mediante l'interfaccia delle system call; il file system sotto, è, per quanto riguarda Unix e os161, basato sul vfs, che è un'interfaccia che permette di accedere agli i-node e ai direttori, e anche ai file individuali. Poi sotto ci sono i dispositivi a blocchi che non abbiamo visto più di tanto in os161, a parte l'aver detto che vi si accede mediante l'operazione di vOP read e write. [/]

[os161-file pag.5] La system call open() si pone a livello user mentre la sys_open() si pone a livello kernel e si chiama con praticamente gli stessi parametri, a parte il fatto di doversi ricordare che il filename è un puntatore a memoria user dal punto di vista del kernel (o meglio, un puntatore a processo user): è infatti convenzione chiamare una funzione del tipo "sys_" + <<nome della funzione originale>> che abbia parametri corrispondenti a quelli originali della system call lato user. [/]

[os161-file pag.6] Cosa ci serve per implementare la open() e la close()? (Ci concentriamo su queste due funzioni con eventuali cenni anche su read() e write()) Per prima cosa: come si fanno a rappresentare i file aperti per ogni processo? Poi: dove teniamo le informazioni relative a questi file aperti?

Il criterio sarà: non consideriamo più di tanto la struttura dei file (per quanto riguarda come è organizzato il file all'interno toccherà all'i-node/v-node nascondere tutto ciò), concentriamoci invece sui metadati, tenendo presenti le potenziali condivisioni. Qui ho messo l'accento sul fatto che un processo child può condividere i file col processo parent. Cosa vuol dire? Possono esserci due processi che stanno lavorando

sullo stesso file: condividere il file tra due processi vuol dire che se uno legge o scrive, l'altro se ne accorge e continua a leggere e a scrivere da dove il precedente aveva terminato (condividono l'offset nel file). Questo va tenuto in conto perché non sempre è così, anzi alle volte ci sono due processi che aprono un file in parallelo e ognuno dei due vuole leggere o scrivere indipendentemente dall'altro. [/]

[os161-file pag.7] Passiamo alla prima domanda che ci siamo posti: Come si fanno a rappresentare i file aperti per ogni processo? La soluzione che vi sto presentando è un pochino più complicata di quello che vi ho proposto nel lab.5. Per prima cosa: supponiamo di voler rappresentare le informazioni che ci servono per accedere a un file aperto mediante una struttura che chiamiamo openfile (una struct). Ad ogni open() corrisponderà una struct, in linea di massima. A questo punto la fileTable (per-process), invece di essere semplicemente un vettore di puntatore a v-node, è un qualcosa in più: un vettore di puntatori a openfile. L'array userà come indice un file descriptor (FD). In prima battuta quindi, a ogni file aperto, corrisponderà una struct openfile; perciò quando facciamo read(), write(), lseek() o altro, il FD sarà l'indice alla tabella dei file aperti del processo, e la riga corrispondente conterrà un puntatore a un openfile. [/]

[os161-file pag.8] Cosa contiene la struct openfile? Il puntatore a file (che significa il puntatore a ciò che ci serve per accedere al file, in pratica il v-node/i-node, che si ottiene chiamando vfs_open()). Poi ci potranno essere delle info che servono a dire come si accede (read-only, write-only, etc. – non presenti nel mio esempio), un offset (ad ogni openfile corrisponderà un offset, cioè dove stiamo scrivendo o leggendo dentro al file), un lock e un reference count (motivato dal fatto che una struct openfile potrebbe essere condivisa). Per ora ci interessano il puntatore al v-node e l'offset. [/]

[os161-file pag.9] Questa è una rappresentazione dell'insieme delle possibili strutture dati, così che possiamo iniziare a capire meglio cosa sia la struct open file. La fileTable è un vettore in cui, ad ogni riga, corrisponde un puntatore a una openfile struct. Quest'ultima potrebbe essere allocata individualmente, ma la scelta conforme a ciò che si fa nei sistemi di riferimento Unix è quella di fare una system fileTable che sia un vettore, quindi: la fileTable contiene un puntatore alla system fileTable. Ogni entry della system fileTable (una struct openfile) contiene, tra le altre cose, anche un puntatore a un v-node. Da notare che il v-node è di fatto una copia (o comunque un riassunto) di ciò che sarebbe l'on-device FD (o File Control Block). [/]

[os161-file pag.10] Un file può essere aperto più volte? Sì: possiamo avere due openfile separate indicate da due file descriptor (anche nello stesso processo). [/]

[os161-file pag.11] Una openfile può essere condivisa fra più thread concorrenti? Sì, ma solo con dei lock. Quindi: un sistema single-threaded può accedere alla stessa openfile senza preoccupazioni, un sistema multi-threaded invece ha bisogno di meccanismi di locking. [/]

[os161-file pag.12] Questa è una soluzione che fa riferimento alla struttura più generale, ancora senza multi-threading se volete. Quello che notate è che ci possono essere due modi per condividere dei file. Voi avete un v-node o i-node che viene condiviso da due openfile struct, che vuol dire che si sono fatte due open() di quel file, e a quel file, in pratica, si pensa di accedervi con offset diversi: ad esempio un processo che legge al byte 100 e un altro processo che scrive o legge al byte 1200, che sono due azioni diverse indipendenti l'una dall'altra sul file. Qui si è indicato, ad esempio, che uno stesso processo potrebbe vedere il file numero 3 e il file numero 5 che per lui sono due file diversi ma che in realtà sono lo stesso file. Lo stesso file potrebbe anche essere visto da due processi diversi. Qui si vuole far vedere che è possibile che si vogliano avere due openfile, quindi due azioni di r/w sullo stesso file indipendenti l'una dall'altra; ma è anche possibile, ad esempio, che ci siano due processi diversi, che potrebbero essere parent e child, che in qualche modo, sono riusciti ad avere accesso alla stessa openfile perché vogliono lavorare sullo stesso file. Potrebbe ad esempio essere che il FD numero 3 del processo x sia diventato lo standard output di un altro processo (qui entra in ballo la system call dup2() di cui non vi parlerò). La conclusione è che ci sono due forme di condivisione, non una come abbiamo detto a lezione nel capitolo 14, dove sembrava che si potesse solo condividere l'i-node. In realtà il vero motivo per cui si fa questa tabella a tre livelli (fileTable, system fileTable, i-node) è tale per cui l'i-node è unico per un file, ma ci sono due livelli di condivisione: uno a livello di i-node (con openfile multiple) e un altro a livello di openfile singola, perché la openfile contiene l'offset nel file. Quindi possiamo, in questo caso, avere due processi che stanno leggendo/scrivendo il file nello stesso posto, uno dei due scrive o legge e l'altro se ne accorge (continuerà quindi a leggere o a scrivere da

dove l'altro aveva finito); oppure ci può essere un altro processo (o lo stesso processo) che in un'altra parte legge o scrive, tramite un altro file descriptor, in parti totalmente diverse del file. [/]

[os161-file pag.13] Queste tabelle, dove dovranno stare? Una cosa è dire dove definiamo le tabelle, un'altra in che file mettiamo le singole azioni. Un modo semplice per definire la fileTable è all'interno della struct proc, cioè è associata al processo una process fileTable e dentro la struct proc potreste avere un puntatore alla fileTable, che viene allocata dinamicamente, o la fileTable allocata staticamente.

[00:19:29 – 00:24:16] Visualizzazione della struct proc all'interno di proc.h per quanto riguarda le fileTable, dettagli delle system fileTable in file_syscalls.c [/]

[os161-file pag.14] Come si realizza la sys_open()? Essa dovrà essere una funzione che gestisce le tabelle dei file aperti in modo opportuno. A una chiamata della sys_open() verrà creata una struct openfile all'interno del system fileTable, otterrà un v-node chiamando vfs_open(), inizializzerà l'offset nella openfile, e gestire quindi la fileTable. [/]

[os161-file pag.15] La vfs_open() è quella che ritorna il puntatore all'i-node. [/]

[00:25:40 – 00:29:26] Visualizzazione della sys_open() [/]

[os161-file pag.16] La sys_close() è più semplice perché ha solo bisogno del FD e deve fare le operazioni inverse: dato il FD si va a localizzare il puntatore alla struct openfile, e a questo punto ci possono essere varie implementazioni (open multiple o singole). La fileTable è un vettore in questo esempio ma può essere anche una lista linkata. [/]

[00:30:36 – 00:32:15] Visualizzazione della sys_close() [/]

[os161-file pag.17] Voglio solo ricordare che per implementare una sys_read() e una sys_write() occorre giocare con lo UIO e con le vOP r/w, che si interfacciano con i driver dei dispositivi a blocchi. A questo punto, una volta che abbiamo capito che si parte dal file descriptor, che permette di accedere alla tabella dei file aperti del processo, e che questa permette, tramite la openfile struct, di raggiungere il v-node, sappiamo, guardando nella funzione loadelf e nella loadsegment, come si fa a gestire una vOP read o write, preparando in precedenza la struct UIO e l'IO-vec. L'unica cosa che non viene prevista qua, già nella tabella dei file aperti del processo, è la concorrenza (si sono volutamente ignorati i lock, però si potrebbero usare). [/]

[os161-file pag.18] A questo punto, trovato il file descriptor, occorre gestire l'offset dalla openfile. Questa è la parte su cui ho speso meno energie e non ho parlato, in precedenza, di come una read o una write possano riuscire a scrivere o leggere meno del previsto. Tuttavia, la UIO ha un campo offset e l'openfile può tenere conto dell'offset. [/]

[00:34:29 – 00:38:22] Visualizzazione della sys_read() [/]