Appunti lezioni SO

1)Un **computer** è una macchina *digitale* (informazione rappresentata in forma numerica discreta), *elettronica* (memorizzazione e manipolazione dell'informazione sono di tipo elettronico), *automatica* (è in grado di eseguire operazioni autonomamente) e *programmabile* (un programmatore può cambiare la successione di operazioni da seguire), capace di effettuare trasformazioni o elaborazioni su dati. Le operazioni da eseguire sono comandate con istruzioni in un linguaggio comprensibile all'elaboratore. Le istruzioni (piccole sequenze di bit) costituiscono il **software**.

Little endian: byte più significativi sono nei byte di indirizzo maggiore (LSB – Least Significant Byte first)

Big endian: byte più significativi sono nei byte di indirizzo minore (MSB- Most Significant Byte first)

Una struttura dati **pacchettizzata** la si dichiara tramite **__attribute__((packed)).**

Una CPU che lavora a 64 bit mi dice l'ampiezza in bit del **bus degli indirizzi** e quindi che posso indirizzare codeal massimo 2^64 byte di memoria.

Il **DMA** (Direct Memory Access) è un meccanismo che consente alle periferiche di accedere direttamente alla memoria per lo scambio di dati, senza controllo della CPU.

I **livelli di privilegio** (**rings**) della CPU sono un metodo di protezione hardware per quanto riguarda i modi di esecuzione della CPU. Il privilegio più alto (livello 0) è il **kernel del SO**, seguito dai **servizi del SO** (livello **1 e 2**) e poi il livello delle **applicazioni**, con privilegio più basso.

La **CPU** esegue istruzioni, effettua calcoli, controlla I/O; il **BUS** trasferisce i dati tra la memoria e gli altri dispositivi. La memoria **RAM** (**volatile**) mantiene i programmi quando devono essere eseguiti e i dati che i programmi usano, ma solo finchè il computer è acceso. La memoria **ROM** (**permanente**) contiene il **BIOS**, cioè le istruzioni per i servizi di base, usate dalla CPU al momento dell'accensione. I **DISCHI** mantengono enormi quantità di dati e di programmi, anche dopo lo spegnimento del computer.

Il **BOOTSTRAP**, ovvero il programma di avvio, viene eseguito dalla macchina al momento dell'accensione ed è già presente in memoria ROM, mentre il SO è nella memoria secondaria. Il bootstrap trasferisce il SO nella memoria principale e gli dà il controllo.

WORD: blocco di byte della stessa dimensione in byte del bus dei dati. Indica quanti byte possono essere trasferiti al massimo con un'unica operazione del BUS tra CPU e memoria.

Lo **stack** contiene i record di attivazione delle chiamate a funzione, la **heap** contiene le zone di memoria allocata dinamicamente, **data** contiene le variabili globali e **text** contiene il codice eseguibile.

3)Quando un **processo** esegue una chiamata ad interrupt, la CPU entra nella modalità di esecuzione kernel e utilizza uno o più **stack** di sistema, separati rispetto allo stack utilizzato per le chiamate a funzione. Ciò permette al kernel di proteggere i record di attivazione utilizzati nelle chiamate ad interrupt. Perciò, **ogni processo mantiene uno stack per ciascun livello di privilegio della CPU.**

I processi, gli interrupt e le eccezioni eseguono ognuno in un proprio contesto denominato **task.** I **task** mantengono le informazioni sui segmenti di memoria utilizzati e sul valore corrente dei registri principali. È compito dello **scheduler** decidere per quanto tempo eseguire ciascun task. Salvare e ripristinare lo stato di un task consente **interleaving,** ovvero intervallare l'esecuzione di più task che utilizzano a turno la CPU, dando l'impressione di un'esecuzione contemporanea.

Una **system call** è il meccanismo utilizzato a livello utente per richiedere un servizio a livello kernel del SO.

Il sistema operativo mette a disposizione diversi servizi:

1)Gestione della Protezione:

• protezione della CPU - sfrutta le capacità della CPU di operare in due diverse modalità:

• **modalità kernel**, che permette di utilizzare tutte le istruzioni ISA, tutti gli indirizzi di memoria, tutti i registri, tutte le periferiche.

• **modalità utente**, che limita le istruzioni eseguibili, l'accesso alla memoria, ai registri e alle periferiche.

• protezione della Memoria.

2)Gestione della CPU:

• scheduling (quale task deve usare la CPU e per quanto tempo?)

3)Gestione della Memoria:

• Indirizzamento, Segmentazione, Protezione, Memoria Virtuale, Paging.

4)Gestione dell'Input/Output:

• Periferiche I/O (tastiera, video, rete, dischi), Storage (organizzazione del filesystem).

5)Chiamate di Sistema (System Calls), Librerie di sistema, programmi di utilità:

• Servizi messi a disposizione in diverse modalità. Le system call sono disponibili mediante chiamate ad interrupt effettuabili in assembly.

Un **processo** è l'istanziazione in memoria di un programma eseguibile. Al lancio di un programma, il **loader:**

- **Crea i segmenti di memoria**
- **Carica in memoria text e data**
- **Crea lo stack utente**
- **Crea gli stack di sistema per l'esecuzione delle syscalls**
- **Copia eventuali argomenti passati a riga di comando**
- **Ordina alla CPU di eseguire la prima istruzione eseguibile**

Al momento dell'esecuzione, un programma utente può invocare una syscall. Essa è implementata nella gestione di un certo interrupt n. La syscall viene invocata con l'istruzione INT n; il programma passa i parametri all'interrupt mettendoli in registri; avviene una chiamata ad interrupt con switch di contesto, ovvero passaggio da modo utente a modo kernel; esecuzione su stack del kernel con successivo ritorno a modo utente e ripristino del contesto.

ECCEZIONI: sincrone rispetto alle istruzioni eseguite dalla CPU

- **TRAP:** l'istruzione che causa il trap viene sospesa temporaneamente per essere portata a buon fine dopo la gestione del trap. (es. **trap per debugging, divisione per 0, disconnesione dalla rete**).
- **FAULT:** l'istruzione che ha causato il trap viene interrotta per poi essere rieseguita da capo. (es. **page fault**, ossia l'impossibilità di un processo di accedere a una pagina presente nel suo spazio di indirizzo virtuale, ma che non è nella memoria fisica. Il sistema cerca di caricare tale pagina e poi fa ripartire il processo).
- **ABORT:** causata da un errore irrimediabile, l'istruzione è abortita e il processo killato. (es. **segmentation fault**, ovvero quando un processo tenta di accedere a un'area di memoria a cui non gli è permesso accedere e che quindi potrebbe danneggiare altri dati).

INTERRUPT: sincroni (interrupt **software**) se chiamati esplicitamente dalla CPU con l'istruzione INT n oppure asincroni (interrupt **hardware**) scatenati da una periferica. Possono essere:

- **MASCHERABILI:** è arrivato un carattere da tastiera, ma c'è qualcosa di più urgente da svolgere
- **NON MASCHERABILI:** avviso di errore sul BUS da gestire immediatamente

Nei moderni SO, più processi, più thread, più gestori di interrupt ed eccezioni sono presenti in memoria contemporaneamente (**MULTITASKING)** e cercano di eseguire concorrentemente (**TIME SHARING**).

I moderni SO sono detti **INTERRUPT DRIVEN,** poiché gran parte delle funzionalità del kernel sono eseguite all'interno di una routine di gestione di un interrupt, cioè come se si fosse verificato un evento che solleva interrupt o eccezioni.

2b) Le librerie sono dei file contenenti le implementazioni in codice macchina di alcune funzioni. Solitamente sono mantenute su disco in directory predefinite. L'estensione del file è diversa a seconda che la libreria sia statica (**.a**) o condivisa (**.o**). Infatti, le librerie si dividono in due categorie:

- **Librerie statiche**
- **Librerie condivise o linkate dinamicamente**

A link-time, le librerie **statiche** sono fisicamente accorpate ai moduli e inserite assieme a queste nell'eseguibile. Il **linker** deve sapere dove cercare su disco le librerie per copiarne il contenuto e inserirlo nell'eseguibile. Per cui a run-time l'eseguibile avrà subito in memoria il codice macchina delle librerie.

Le librerie **condivise** NON sono inserite a link-time nell'eseguibile generato e non sono fisicamente accorpate ai moduli. Infatti, per ogni chiamata a una funzione di libreria condivisa, il **linker** inserisce nell'eseguibile una parte di codice macchina (**stub**) e la posizione a run-time su disco della libreria. A run-time, lo stub cerca la libreria condivisa su disco e la carica in memoria, la rende disponibile per l'esecuzione e poi effettua la chiamata a funzione.

Il **linker** deve sapere 1)dove si trova a link-time la libreria condivisa, per sapere se fornisce la funzione richiesta e 2)dove si troverà la libreria a run-time su disco, poiché deve inserire la posizione nel codice assieme allo stub.

5c) Busy waiting: meccanismo di sincronizzazione nel quale un thread attende il verificarsi di una condizione e lo fa verificando continuamente se la condizione sia vera. C'è un enorme consumo di CPU

Deadlock: si verifica quando 2 o più thread aspettano indefinitamente un evento che può essere causato solo da uno dei thread bloccati.

Starvation: un thread in attesa di una risorsa, non riesce mai ad accedervi per le continue richieste da thread con privilegi maggiori. Ciò può essere causato da un errore nell'algoritmo di sincronizzazione.

Race condition: fenomeno che si presenta in sistemi concorrenti e avviene quando il risultato finale dell'esecuzione dipende dalla sequenza di esecuzione.

Liveness: un thread non deve rimanere bloccato per sempre, per cui ci sono due proprietà:

- **Progresso:** se nessun processo è all'interno della sua sezione critica e ci sono thread in attesa di entrare in tale sezione, allora la decisione non può essere rimandata.
- **Bounded waiting (attesa limitata):** se un thread richiede l'accesso alla sezione critica può essergli negata un numero finito di volte.

La variabile **errno** è un tipo di variabile globale che contiene il valore dell'ultimo errore di esecuzione ed è definita dentro al header errno.h. Si può conoscere il tipo di errore che viene generato tramite la funzione rientrante strerror_r.

Un **processo zombie** è un processo informatico che, nonostante abbia terminato la propria esecuzione, possiede ancora un **PID** e un **PCB**, necessario per permettere al processo padre di leggere il valore di uscita.

Le **API** per Pthread distinguono le funzioni in 3 gruppi:

- **Thread management:** funzioni per creare, eliminare e attendere la fine dei pthread
- **Mutexes:** funzioni per supportare la sincronizzazione semplice tramite mutua esclusione
- **Condition variable:** funzione a supporto di una sincronizzazione più complessa, dipendente dal valore delle variabili

Un'operazione, che usa dati condivisi, è **atomica**, rispetto alle operazioni che usano gli stessi dati, se viene eseguita in modo indivisibile. In caso di interruzione di un'istruzione, per renderla comunque atomica si deve attendere la fine dell'interruzione e far ripartire dall'inizio l'operazione interrotta. In caso di parità di condizioni di partenza, il risultato di un'operazione atomica sarà sempre lo stesso.

6a) IPC = comunicazioni fra processi, si distinguono in:

- **a memoria condivisa**: i processi hanno un'area di memoria condivisa per il passaggio di informazioni (es. variabili condivise fra thread). Posix mette a disposizione il segmento di memoria condivisa System V
- **a scambio di messaggi**: sfruttano i messaggi e non la memoria; tramite questi vengono fatte le sincronizzazioni. Le funzionalità principali sono *send* e *receive.* Le operazioni possono essere sincrone e asincrone. Un esempio sono i processi su computer diversi, ma anche sullo stesso computer, ad esempio quelli con più core

7b) Il SO tiene traccia sia della parte di RAM occupata sia di quella libera; deve allocare memoria e deve liberarla quando non serve più ad un processo ad esempio. Il SO scarica su disco contenuti che dovrebbero essere nella memoria fisica. **Binding address**, ovvero la rilocazione, indica l'associare indirizzi di memoria fisica ai dati e alle istruzioni di un programma e può avvenire:

- durante la compilazione
- durante il caricamento del programma in memoria
- durante l'esecuzione

Il più primitivo, cioè quello durante la **compilazione**, funziona che il compilatore determina dove caricare il programma in memoria (es. da indirizzo 384 a 768). Se quell'area di memoria è già occupata, il compilatore non se ne preoccupa minimamente e continua con ciò che deve fare. Il vantaggio è la semplicità, mentre lo svantaggio è l'impossibilità di caricare il programma in memoria, in caso di memoria occupata (si deve sapere l'occupazione della memoria a run-time -> no multiprogrammazione). Tutt'oggi è utilizzato per sistemi molto semplici e veloci.

Durante il **caricamento del programma in memoria**, il codice generato dal compilatore contiene indirizzi relativi (e non assoluti come nel primo caso); ad esempio, vai 10 byte più avanti/indietro dell'istruzione attuale (JUMP). In questo caso, il loader si occupa dove mettere in memoria tale segmento. Se decidesse di metterlo dall'indirizzo 384 a 768 e la JMP comunica di andare all'indirizzo relativo 378, allora l'indirizzo assoluto diventerà 384+378=762. I vantaggi sono la gestione della multiprogrammazione e il non utilizzo di hardware particolare, mentre lo svantaggio è la richiesta di traduzione degli indirizzi da parte del loader.

Per quanto riguarda la rilocazione durante l'**esecuzione**, si aggiunge una parte hardware che gestisce gli indirizzi, ovvero la MMU (memory management unit). Agisce sulla base di dati che acquisisce in precedenza durante la fase di caricamento in memoria dell'eseguibile. Gli indirizzi, specificati dalle istruzioni in assembly, sono indirizzi logici (CPU) che vengono passati alla MMU, che ha il compito della mappatura sugli indirizzi fisici (MEMORIA). Esempio: i.logico: 897, rilocazione: 24000 -> i.fisico: 24897.

L'**allocazione di memoria** consiste nel reperire e assegnare uno spazio di memoria fisica all'eseguibile. Con l'allocazione **contigua** tutto lo spazio assegnato a un programma deve essere formato da celle consecutive, mentre con quella **non contigua** assegno aree di memoria separate. La MMU, basata su rilocazione, gestisce solo allocazione contigua. Si può distinguere anche fra allocazione statica e dinamica. Con quella **statica** un programma mantiene sempre la propria area di memoria fino alla sua terminazione e non è possibile rilocarlo. Invece, con quella **dinamica**, durante l'esecuzione un programma può essere spostato all'interno della memoria.

L'allocazione **a parti fisse** funziona che il SO si riserva una parte di memoria e crea della partizioni di memoria variabile per il resto del sistema. È quindi un'allocazione statica, molto semplice, ma con molto spreco di memoria e

parallelismo limitato dal numero di partizioni. Lo spreco è determinato dal fatto che se un processo occupa una dimensione inferiore a quella della partizione, allora lo spazio è inutilizzato (**frammentazione interna -> dentro le partizioni**).

L'allocazione **a partizioni dinamiche** si basa sull'assegnazione a richiesta di memoria ai processi, ma si presentano comunque zone inutilizzate, sia per effetto di terminazione dei processi sia per non completo utilizzo dell'area disponibile da parte dei processi attivi. Per cui dopo allocazioni e deallocazioni ci si trova con spazio libero, ma suddiviso in piccole aree (**frammentazione esterna -> fuori dalle partizioni**). Si potrebbe ovviare al problema tramite la compattazione, ovvero trasferire le aree di memoria, in modo che si formi un blocco unico libero, ma è un'operazione molto onerosa e non può essere usata per sistemi interattivi.

Per tenere traccia delle partizioni libere e di quelle occupate si usano le strutture dati, ovvero mappe di bit e liste con puntatori. Nella **mappa** ogni **bit** indica lo stato di occupazione di un'area di memoria (1 occupato, 0 libero). Bisogna determinare una dimensione dell'unità di allocazione. Il vantaggio di questo metodo è la dimensione fissa e calcolabile a priori della struttura, ma è costoso calcolare un'area di memoria libera. Tramite la **lista con puntatori** si tiene traccia dei blocchi allocati e quelli liberi. Ogni elemento specifica la sua entità e la sua dimensione.

La selezione del blocco libero può essere:

- **First Fit**: viene preso il primo blocco dall'inizio grande abbastanza per il processo (migliore)
- **Next Fit**: parte da dove si era fermato con l'ultima allocazione e non dall'inizio
- **Best Fit**: seleziono il blocco più piccolo fra quelli grandi abbastanza
- **Worst Fit**: seleziona il più grande fra i blocchi liberi (peggiore)

La **paginazione** è un sistema di sfruttamento della memoria per processi, utilizzabile anche per sistemi non aventi memoria virtuale. È un approccio contemporaneo, che riduce frammentazione interna e minimizza la frammentazione esterna; necessita però di un hardware adeguato. Lo spazio di indirizzamento logico viene suddiviso in blocchi di dimensioni fisse, detti **pagine.** La memoria fisica è vista dal SO come un insieme di blocchi di ugual dimensione, detti **frame.** Quando un processo viene allocato in memoria, il loader trova un numero di frame liberi, in cui mettere lo stesso numero di pagine. **Non è necessario che i frame siano consecutivi**, infatti è questa la differenza principale che si ha con la paginazione. Perciò mi tocca sapere ogni pagina dov'è stata collocata, ogni frame preciso (v. slide 35-36 di 7b). La dimensione della pagina deve essere una potenza di due per la semplificazione del passaggio da i.logico a i.fisico. Con pagine piccole, la tabella cresce di dimensioni. Di solito, si utilizzano valori tipici come: **4KB**, **4MB**. La tabella delle pagine viene gestita dalla MMU. Non cambia l'offset nel passaggio da indirizzo logico a indirizzo fisico. Il problema che sorge è **dove mettere la tabella delle pagine**. Una prima soluzione è nei registri dedicati, ma è troppo costoso. Un altro metodo può essere includerla totalmente in memoria, ma si accederebbe in memoria due volte, prima per la tabella poi per il dato. La soluzione è data da **TLB**, un registro che agisce come memoria cache per le pagine. Si tiene traccia in un'altra tabella di una chiave (indice della pagina) e di un valore(frame). Si controlla se c'è già il valore corrispondente alla pagina e in caso di TLB miss si controlla nella tabella delle pagine.

In un sistema con **segmentazione,** la memoria per un programma è suddivisa in aree differenti dal punto di vista funzionale. Ad esempio, si hanno:

- aree **text**: contengono codice eseguibile, sono in sola lettura, possono essere condivise tra più processi
- aree **dati**: contengono le variabili globali, possono essere condivise oppure no
- area **stack**: read/write, non è condivisa

Con la segmentazione, lo spazio di indirizzamento logico è costituito da un insieme di segmenti. Un segmento è un'area di memoria di dimensione variabile, caratterizzata da un **nome** e da una **lunghezza**(**offset**). La suddivisione dei segmenti viene fatta dal programmatore o dal compilatore. La tabella dei segmenti viene gestita dalla MMU e se chiamo il mio segmento S, dovrò cercare nella tabella alla posizione s-esima. La tabella contiene l'indirizzo fisico e un limite per controllare che l'offset non superi la dimensione del segmento. Il problema si presenta nell'allocare segmenti di dimensione variabile e quindi si usano le stesse tecniche viste prima, ovvero:

- tecniche di allocazione dinamica(es. First Fit)
- compattazione

I sistemi moderni usano sia **segmentazione** che **paginazione,** anche se non c'è memoria virtuale. Ogni segmento è suddiviso in pagine, caricate in frame liberi della memoria(non necessariamente contigui). La MMU deve avere supporto per entrambe ovviamente e in tal modo si beneficia sia di segmentazione (condivisione e protezione) sia di paginazione(no frammentazione esterna).

Si è sovrapposta una nuova tecnica da qualche decennio, ovvero la **memoria virtuale**. Permette di eseguire concorrentemente processi che necessiterebbero di maggior memoria di quella presente. Non è necessario che tutto lo spazio di indirizzamento sia presente in memoria contemporaneamente, ma solo ciò che serve sul momento. Ogni processo ha accesso a uno **spazio di indirizzamento virtuale** che può essere più grande di quello fisico. Gli indirizzi virtuali sono mappati o su indirizzi fisici o su memoria secondaria. Se si accede a indirizzi di quest'ultima, i dati associati sono trasferiti in memoria principale. La **paginazione su richiesta** usa la tecnica di paginazione, sapendo però che alcune pagine possano essere in memoria secondaria. Nella **tabella delle pagine** si usa un bit che indica se la pagina è presente in memoria centrale o no. Se un processo tenta di accedere a una pagina non in memoria, allora il processore genera un page fault e il **pager** si occupa di caricare la pagina mancante in memoria e di aggiornare la tabella.

Con il termine **swap** si intende l'azione di copiare l'intera area di memoria usata da un processo. Se si copia dalla memoria secondaria alla principale si parla di **swap-in**; viceversa, si parla di **swap-out**. Era una tecnica utilizzata in passato quando ancora non esisteva la paginazione su richiesta. Quest'ultima può essere vista come una tecnica di swap di tipo lazy, in cui viene caricato solo ciò che serve. Ecco perché alcuni SO indicano il pager con il nome di **swapper**.

Suppongo che un'istruzione macchina del codice in pagina 0 faccia riferimento alla pagina 1, la MMU scopre, tramite la tabella delle pagine, che la pagina 1 non è in memoria principale. Perciò viene scatenato il **page fault**, che viene catturato dal SO. Quest'ultimo cerca in memoria secondaria la pagina e la carica in memoria principale. Si aggiorna la tabella delle pagine e si riavvia l'esecuzione dell'istruzione. Un problema si presenta quando in memoria non ci sono frame liberi; per cui, la pagina "vittima" sarà la meno "utile". Si utilizzano degli algoritmi di rimpiazzamento, il cui scopo è **minimizzare** i page fault.

7c) Il **copy on write** è un meccanismo di condivisione delle pagine da parte del processo padre e del processo figlio, dove solo in caso di scrittura il SO crea una copia delle pagine per entrambi e modifica quella opportuna. NO IN LETTURA. Ad esempio, se il processo figlio tenta di modificare una pagina dello stack, il SO considera la pagina da copiare nello spazio degli indirizzi del processo figlio, il quale modifica la sua copia e non quella del padre. Le pagine per il copy on write sono tenute libere dal SO.

La **sostituzione delle pagine**, in caso di nessun frame libero, deve essere registrata nella tabella delle pagine. Se si va a sostituire una pagina che è stata modificata dopo essere stata inserita in memoria, allora bisogna salvare tale pagina, privilegiando perciò le pagine che non sono state modificate. Lo scaricamento si può quindi evitare, ma occorre aggiungere alla pagina un bit di modifica(**dirty bit**), gestito via hardware:

- quando la pagina viene caricata è a 0
- quando la pagina è scritta è a 1
- quando la pagina è scaricata viene fatta la scrittura su disco solo se il bit è a 1

La **stringa dei riferimenti** è un elenco di indici di pagine, che comunica l'ordine con cui vengono accedute le pagine dal processo. Il numero di page fault diminuisce con l'aumentare del numero di frame.

Un primo approccio per la scelta della pagina da scaricare è usare un algoritmo FIFO. Si sceglie quindi la pagina più vecchia, cioè quella con istante di caricamento più basso. È facile da implementare, ma non ha gran successo, perché non ha un buon criterio per la sostituzione; potrebbe scegliere un dato importante usato spesso. Addirittura, le prestazioni peggiorano con l'aumentare del numero di frame (**Anomalia di Belady**).

L'algoritmo **ottimale** o minimo (non implementabile) scarica la pagina che non verrà utilizzata per il periodo di tempo più lungo (si dovrebbe predire il futuro, impossibile). L'algoritmo **LRU** scarica la pagina che non viene acceduta da più tempo, supponendo che non verrà usata. Rappresenta una buona politica di sostituzione ma è costoso da implementare. Poche architetture possono permettersi hardware di LRU, poiché molto costoso.

Determinare quanti frame mettere a disposizione per ogni processo è dato da m/n con m frame e n processi oppure con un'allocazione proporzionale con un numero minimo di frame per processo. Si può pensare a una **sostituzione** delle pagine **globale**, cioè liberando un frame da un processo qualunque. In caso di un numero di frame inferiore al minimo si causa **THRASHING,** con continui page fault e swap. Per prevenire, una soluzione è evitare che un primo processo in thrash mandi in thrash un altro, sospendendo preventivamente il primo processo.

7a) Nei sistemi moderni, la gestione della memoria è sostenuta dal processore e utilizzano la segmentazione paginata, ovvero segmentazione e paginazione combinate. L'indirizzo logico è suddiviso in più parti: la prima fa riferimento alla **directory delle pagine**, trovo il riferimento alla tabella delle pagine, alla quale accedo grazie al secondo componente dell'indirizzo logico e con l'ultima parte(**offset**) riesco a individuare lo spazio di indirizzamento fisico.

Gli **indirizzi** specificati dalle **istruzioni macchina** di un programma in esecuzione sono **virtuali,** gli indirizzi **spediti sul bus** degli indirizzi sono **fisici**, gli indirizzi nelle variabili **puntatori** sono **virtuali.**

Fra stack, memory mapping segment e heap vengono poste aree di memoria di dimensione variabile, in modo tale che chiunque volesse attaccare la memoria non sa bene in quale area debba farlo. Tale meccanismo si chiama di **randomizzazione** e un esempio può essere visto con il comando **sysctl -w kernel.randomize_va_space.** Tale compito viene fatto dal loader, nel momento del caricamento in memoria dell'eseguibile.

8a) Lo **scheduling** della CPU è il meccanismo sul quale sono basati i sistemi che supportano la multiprogrammazione, ovvero più processi in memoria. L'obiettivo è quello di massimizzare il tempo d'uso della CPU. In alcuni casi, si parla di **scheduler a breve termine,** al contrario di quello a lungo termine che assegnava quali processi caricare in memoria (es. sistemi batch). Lo **scheduler a medio termine** decide di sospendere temporaneamente un processo in esecuzione e lo toglie dalla memoria, per poi essere ripreso più avanti. Lo scheduler decide quali processi far diventare running fra quelli ready.

I processi sono costituiti da cicli di esecuzione (**CPU burst**) e attese per l'I/O (**I/O burst**). Il ciclo di vita di un processo:

- inizia con un CPU burst
- alterna I/O burst e CPU burst
- durante un ciclo di CPU burst chiede al SO di terminare

I/O bound: molti cicli di CPU burst ma brevi perché frequentemente interrotti da I/O

CPU bound: meno cicli di CPU burst ma più duraturi

PCB: Process Control Block, struttura dati con cui viene descritto ogni processo. Dipende da SO a SO, ma contiene tutte le informazioni che servono al sistema per gestire il processo e per eseguirlo. Le info sono:

- **stato del processo**
- **PC**(Program Counter) = (**segmento:offset**): prossima istruzione da eseguire per tal processo
- **Registri della CPU**
- **Informazioni sullo scheduling**

La **ready queue** è una struttura dati in cui lo scheduler mantiene i PCB dei processi pronti per eseguire.

Il meccanismo di **prelazione (preemptive scheduling)** è una politica adottata nel caso in cui un nuovo processo entri nella ready queue. Afferma che se il processo ha maggior priorità di quello in esecuzione in precedenza, allora lo spodesta. **Senza prelazione**, il controllo della CPU resta al processo in esecuzione in precedenza.

Il **dispatcher** è una parte del SO che passa effettivamente il controllo della CPU al processo selezionato dallo scheduler. Si occupa di passaggio alla **modalità utente**, riavvio dell'esecuzione del programma utente dal punto in cui era stata interrotta (**jmp**) e **context switch**, cioè ripristino del contesto del processo che sta per essere eseguito. Il **dispatch latency** deve essere minimizzato per assicurare efficienza al SO.

Il **throughput** (produttività) è il numero di processi completati nell'unità di tempo, mentre il **turnaround time** è il tempo tra quando il processo viene avviato e quando viene terminato. Il **tempo di attesa** è il tempo che trascorre un processo nella ready queue. Il **response time** è il tempo che intercorre tra la prima richiesta fatta a un processo e la prima risposta che questo dà.

Algoritmi di scheduling

- **FIFO:** il più semplice, si considera l'ordine di entrata nella ready queue; il primo che entra è il primo che esce. (v. esempio slide 26 di 8a)
- **SJF (Shortest Job First):** seleziona il prossimo processo da eseguire, prendendo dalla ready queue il processo che ha il CPU burst più breve. Se ci sono più processi con stesso CPU burst si usa la tecnica FIFO. È uno dei migliori per quanto riguarda il tempo di attesa medio, ma soffre di starvation. Le predizioni sulla lunghezza del prossimo CPU burst sono fatte sulla base di: è probabile che un prossimo CPU burst abbia lunghezza analoga. Si ottiene con una formula tramite una media esponenziale
- **Priority Scheduling:** ogni processo è associato a un intero che rappresenta la sua priorità. La CPU è allocata al processo con il valore intero più piccolo, ovvero quello con priorità più alta. SJF è un caso speciale di scheduling con priorità. Un processo più invecchia, più gli viene data maggior priorità
- **Round Robin:** progettato per operare efficacemente in sistemi time sharing. È una variante di FCFS (FIFO), in cui la coda è gestita in maniera circolare. Ogni volta che un processo viene messo in esecuzione, lo scheduler imposta un timer che ne limita il tempo di esecuzione. Se il tempo scade, il timer genera un interrupt e passa la CPU a un altro processo
- **Code multiple:** adatto a gestire processi interattivi (foreground) e background (batch). La ready queue viene partizionata in più code separate e ciascun processo è associato a una coda. Vengono applicati scheduler diversi su code diverse

9a) Il SO si occupa di:

- Inizializzare i dispositivi di memoria secondaria tramite formattazione fisica e logica
- Creare il blocco di boot, che si occupa di avvio della macchina e SO
- Gestire i blocchi difettosi del disco
- Gestire in modo efficiente l'area di swap (partizione separata)

Ogni richiesta di scrittura/lettura su disco avviene attraverso una syscall bloccante di I/O che specifica:

- Se l'operazione è di input o di output
- L'indirizzo del disco su cui effettuare il trasferimento
- L'indirizzo di memoria su cui effettuare il trasferimento
- Il numero di byte da trasferire

Se il disco sta già soddisfacendo altre richieste, mette le nuove in coda. La forma più semplice di **scheduling su disco** è rappresentata dall'algoritmo **FCFS** (FIFO). Poco efficiente, non viene considerata la prossimità delle richieste alla posizione corrente della testina → **SSTF** (Shortest Seek Time First). Si sceglie la richiesta che necessita meno tempo; si ha un miglioramento notevole → **SCAN:** andando da un estremo all'altro del disco, si servono man mano le richieste che si incontrano. Emula il comportamento di un ascensore → **C-SCAN:** opera come SCAN, ma non inverte mai la direzione. Per cui alla fine si riposiziona sulla traccia 0, senza servire richieste nel viaggio di ritorno → **C-LOOK:** si ferma la testina quando si incontra l'ultima richiesta e si riposiziona da quel punto. Solitamente si usa SSTF se il disco non è molto utilizzato, altrimenti C-LOOK.

9b) Il **file system** è un'astrazione del modo di allocazione dei dati e organizzazione della memoria. Un **file** è un insieme di informazioni correlate e registrate nella memoria secondaria con un nome. Può avere una sua struttura interna, in base a formato e tipo. Le sue caratteristiche sono nome, tipo, locazione e dimensione. Altri attributi sono protezione (informazioni su lettura, scrittura ed esecuzione), ora e data di ultimo utilizzo e ultima lettura.

Il file system è organizzato in una **struttura gerarchica** che mantiene i file all'interno di contenitori, detti **directory**. La **scrittura** su file avviene per blocchi attraverso una chiamata ad una system call che specifica nome del file e le informazioni che si vogliono inserire. Uguale per la **lettura**, a parte che lo scopo è specificare la locazione di memoria in cui deve essere trasferito il file. Il sistema mantiene in memoria l'elenco dei file in uso in un'apposita tabella detta **tabella dei file aperti**. A ciascun file aperto sono associati: un puntatore alla posizione corrente, un contatore delle aperture e la posizione su disco.

Il SO fornisce diversi metodi di accesso ai file:

- **Sequenziale:** le info del file vengono elaborate in ordine (es. compilatore)
- **Diretto:** il file è costituito da un insieme ordinato di blocchi a cui si accede direttamente
- **Tramite indice:** al file è associato un file indice con lo scopo di velocizzare le ricerche

La **directory** la si può considerare come una tabella in cui sono memorizzate le informazioni relative al file e necessarie a individuare la posizione del file e a gestirlo (copiarlo, cancellarlo, eseguirlo…). Deve consentire velocità nel trovare il file, una gestione dei nomi e grouping, ovvero raggruppati in base alle loro caratteristiche. Ce ne sono più tipi:

- **Single level**: struttura lineare, ma inefficace con più utenti, che sono costretti a usare nomi diversi per uno stesso file
- **Two level**: ogni utente ha una sottodirectory separata UFD (user file directory) e quindi possono usare stessi nomi, ma non struttura a piacimento
- **Ad albero**: consente agli utenti di creare proprie sottodirectory e di organizzare a piacimento i file. La radice è detta **root directory** e ogni directory (o sottodirectory) può contenere file o sottodirectory a sua volta. Si definisce anche una directory corrente rispetto alla quale effettuare la ricerca dei file che può essere cambiata dall'utente (con cd). I **path name** possono essere **assoluti** (partono dalla root) o **relativi** (partono dalla directory corrente).

9c) Strutture memorizzate su disco:

- **Boot control block:** blocco di controllo dell'avviamento, contenente informazioni necessarie all'avviamento del SO
- **Volume control block:** blocco di controllo dei volumi
- **Strutture delle directory:** utilizzate per memorizzare i file
- **File control block:** blocchi di controllo dei file che contengono le info sui file stessi

Per il SO i file sono memorizzati in un descrittore, detto File control block (**FCB**) che contiene:

- **Nome**
- **Proprietario**
- **Dimensioni**
- **Permessi di accesso**
- **Posizioni dei blocchi:** info per sapere dove si trova su disco il file

Il descrittore prende il nome di **inode** nei sistemi Unix, perché si usa sia per i file che per le directory. Le applicazioni per creare un nuovo file effettuano una chiamata al file system logico (system call) che alloca un nuovo FCB. Il sistema carica la directory in memoria centrale, la aggiorna con il nuovo FCB e la risalva su disco. Una volta creato il file per essere letto o scritto deve essere aperto, tramite una system call open.

Quando un processo vuole aprire un file, consulta la struttura delle directory per sapere dove si trova il file e viene effettuata l'operazione di apertura e ciò fa sì che venga inserita in una tabella del processo un indice (**FD**), ovvero un puntatore alla tabella di sistema dei file aperti, all'interno della quale ci sono le informazioni di accesso al file (**FCB**).

Virtual File Systems (VFS): è un livello astratto posto al di sopra del file system, il cui scopo è quello di permettere alle applicazioni l'accesso a differenti tipologie di file in differenti file system.

Un elemento che incide fortemente sulle prestazioni del file system è il metodo di **allocazione dei blocchi,** ci sono 3 metodologie di allocazione:

- **Contigua:** ogni file deve occupare un insieme di blocchi contigui nel disco; non c'è bisogno di spostare la testina, accesso semplice. Il problema sta nell'avere una porzione di disco sufficientemente grande da contenere tutto il file; le strategie sono first fit e best fit. Il sistema soffre di frammentazione esterna, risolta con la deframmentazione. Se si effettua preallocazione (spazio molto più grande della dimensione attuale del file), si genera frammentazione interna
- **Concatenata:** ogni file è costituito da una lista concatenata di blocchi del disco da distribuire in qualunque parte del disco stesso; non c'è frammentazione esterna, perché ogni blocco è allocato singolarmente e non c'è nemmeno interna, perché non si deve pre-allocare il file. Scomodo per quanto riguarda l'accesso diretto e lo spazio che occupa
- **Indicizzata:** ogni file ha il proprio blocco indice in cui l'i-esimo elemento del blocco indice punta all'i-esimo elemento del file; supporta l'accesso diretto senza frammentazione, ma comporta spreco di spazio. La dimensione e la memorizzazione del blocco indice è critica:
    - **Schema concatenato:** il blocco indice occupa esattamente un blocco. Se non è sufficiente, l'ultimo puntatore del blocco indice punta a un altro blocco indice
    - **Indice multilivello:** il blocco indice di primo livello punta ad altri blocchi indice di secondo livello e così via
    - **Schema combinato:** una parte dei puntatori del blocco indice punta direttamente ai blocchi del file (**blocchi diretti**) e una parte punta invece a indici multilivello (**blocchi indiretti**). Se il file è piccolo si usano solo blocchi diretti.