RIASSUNTO SISTEMI OPERATIVI

L01-02:

Sistema operativo -> E' un insieme di programmi che hanno vari scopi e usano l'hardware. Gli scopi sono il controllo e la gestione della macchina, nonché l'interfaccia con l'utente, dove funziona come tramite.
Il s.o. si pone come tramite anche per la comunicazione tra Utente e HW quando l'utente non ha i privilegi necessari (superuser)

Interrupt -> devono risvegliarsi componenti del s.o. per farle reagire a variabili tipo: Errori con eccezioni, richieste di servizi, i/o hw interrupt di sys.
Chiamate di sistema -> le system call permettono una serie di azioni che garantiscono all'utente accesso a funzionalità del HW regolamentate dal SO, che può operare altre scelte in base a altri user o situazioni precise.
Sistema pulito -> no accesso diretto a HW, ma tramite SO

L03:

I SO non esistevano, i programmi venivano caricati in memoria su schede -> job in memoria -> load -> exec -> fail/exit.
Programma + dati = job. Idea -> programmino SEMPRE in esecuzione, per scaricare e caricare jobs dopo la terminazione -> è un SO, usa HW e I/O.
Spooling: meccanismo per sovrapporre operazioni i/o con una coda.
S.O. **multiprogrammato** -> prevede l'esistenza di più job in memoria. Usa batch -> li esegue in fila. L'idea è allocare certe quantità di tempo a jobs diversi che vengono tutti eseguiti su una CPU che comunque esegue un calcolo alla volta = time sharing SO.

L04:

Sistemi paralleli: sono sist. multiprocessore che hanno affidabilità e condividono risorse. Sistemi di rete (networking), sistemi distribuiti (astrazione cloud) e cluster (aggregazione).
Sistemi real time -> hanno forti vincoli temporali (aerei, sicurezza, app streaming) hard e soft rispettivamente.

L05-06:

Gestione dei processi -> programma in esecuzione -> insieme di istruzioni -> differenza fra programma in lista e in esecuzione.
Program counter = descrizione di cosa fa il programma e cosa fa in esecuzione.
Ogni processo richiede risorse, le quali vengono assegnate -> programma + program counter (in exec) = risorse (cpu ram ecc).

Un processo è una unità di lavoro -> ogni processo rappresenta un insieme coerente di attività e di richieste che svolgono un certo compito.
Sys multitasking : + processi insieme.

L07: cenni storici

L08:

Struttura del s.o: sistema analitico -> formato da un insieme di procedure (syscall)
Utente fa syscall che scatenano utilità del SO.
In un unico blocco di codice hai user -> shell -> kernel -> Interface hw - controller

Sistema a livelli: Vantaggi - modularità /debug/ block box (i livelli sono separati), problemi di ordinamento ed efficienza -> è una situazione idealmente buona ma complicata. La struttura modulare è tipo Kernel = insieme di oggetti

Macchina virtuale = simulazione hw, ma non è realmente un SO. Non implementa tutte le funzionalità del SO ma simula solo l'HW.

Per avviare un sistema operativo il primo passo è il Bootstrap all'interno di una ROM, o nel hw se c'è il concetto di firmware.

L09:
Gestione dei processi (cpu, mem, i/o). Un processo è una entità a sé con un compito ben preciso. Nel multitasking, i processi lavorano in modo concorrente, non contemporaneamente.
Processo = programma. Dinamica -> program counter + registri + stack + dati.
PCB / process control block: è un record di informazioni che riassumono ciò che il SO deve sapere riguardo al processo tipo:
Stato del processo, program counter (wait, ready), registri CPU, info scheduling + io e memoria.

Stato di un processo:
NEW -> PID creato, info PCB. Ammissione nel sistema, allocazione memoria.
READY -> pronto all'esecuzione ma deve aspettare la competizione scheduler
EXEC -> processo in esecuzione. 3 possibili outcome: termina / torna ready / va in WAIT -> (aspetta i/o x es)

L10:
Context switch: salva lo stato del processo per ripristinarlo quando deve ripartire per dare intanto la CPU ad un altro programma che ne ha bisogno.
Il context switch ci mette un po' di tempo (poco) ad essere eseguito.

L11:

Scheduling:

Nei 3 stati principali (ready -> exec -> wait ->) c'è bisogno di un algoritmo che determini cosa e quando viene eseguito per ottimizzare le prestazioni e l'utilizzo delle risorse della macchina.
Nelle transazioni NEW -> READY: scheduler a lungo termine -> quale processo deve immettere? Pianificazione attivata non frequentemente
READY -> EXEC: scheduler invocato ogni volta che SO si libera. Secondo certi criteri si sceglie quale eseguire (scheduler a breve termine o cpu scheduling)

I processi possono essere CPU bound (principalmente lavoro) oppure i/o bound (richiedono principalmente operazioni i/o).

L12:

Gerarchia dei processi - PID = fork();
Ogni processo ha sempre solo un genitore. La root sarà il processo P0 che tramite fork() genera altri processi.
Altrettanto importante è la terminazione de processi. Se il padre muore, i figli muoiono in cascata oppure init adotta i processi figli del padre.

L13:

Thread: unità di base per l'utilizzo della CPU. Composto da PC, registri e stack. Più thread formano un processo. Condividono spazio di memoria e risorse.
Può fare context switch veloci. Condivide la memoria -> comunicazioni veloci.
Svantaggi: routine non devono provocare problemi di condivisione.

Se il programma supporta il multithread posso lavorare con il thread switch al posto del context switch, che è molto più leggero e veloce. Problema -> no protezione fra i thread.

Gestione thread: Su unix Pthread, libreria ad alto livello per la gestione.
Pthreads -> utente; l' SO non ne sa niente
-> kernel (aiuta a gestire i thread, SO ne tiene traccia) la più usata
OSS: non si può fare thread switch fra processi diversi, serve prima context switch.

L14:

Burst: il burst è il tempo in cui un processo usa la CPU ininterrottamente. Durata permanenza su CPU = T quindi tempo medio di permanenza E Ti / n
NB: il burst ci permette di dire quanto tempo passa in exec prima che ci sia un cambiamento di stato del processo.

2 scenari:
Molta CPU -> calcoli pesanti, rendering -> CPU bound
Molto I/O -> scrivo file, uso i/o rete -> I/O bound
La maggior parte dei servizi di sistema sono I/O bound. La maggior parte dei processi utente sono CPU bound, ma non in tutte le fasi. Le 2 categorie coesistono, in modo bilanciato.
Ci sono 2 tipi di scheduler: con e senza prelazione. Con prelazione -> possibilità di riportare un P che è in EXEC in ready (lo fa SO). Permette un controllo maggiore.
Prelazione: La preemption (o pre-rilascio o ;) è, in informatica, l'operazione in cui un processo viene temporaneamente interrotto e portato al di fuori della CPU, senza alcuna cooperazione da parte del processo stesso, al fine di permettere l'esecuzione di un altro processo.

NOTA CONTEXT SWITCH : context switch è quella parte del kernel del sistema operativo che cambia il processo correntemente in esecuzione su una delle CPU. Questo permette a più processi di condividere una stessa CPU. Utile sia nei sistemi con un solo processore, consente di eseguire più programmi contemporaneamente, sia nell'ambito del calcolo parallelo, consente un migliore bilanciamento del carico. 1. salvare lo stato della computazione del processo correntemente in esecuzione. Queste informazioni sullo stato del processo vengono generalmente salvante nel PCB del processo. 2. Lo scheduler sceglierà un processo dalla coda READY e accederà al suo PCB per ripristinare il suo stato nel processore, in maniera inversa rispetto alla fase precedente.

L15:
Criteri di scheduling:
Indici di prestazioni -> utilizzo CPU e % di utilizzo CPU su finestra analisi.
-**Frequenza** di **completamento** (throughput) = n.processi/tempo
-**Tempo** di **completamento** (turnaround) = tempo per completare il processo. Inizio - fine = durata. Non è stabile.
-**Tempo** di **attesa**: Somma di tutti i tempi passati in coda ready. Il tempo in coda ready è sprecato perché la CPU non era pronta /stava facendo altro.
In tutto questo lo scopo del SO è -> prevedere buona gestione anche a sistema mediamente carico (difficile).
Sistema interattivo: si dice tale se fra evento -> effetto passa poco tempo.
-Tempo di risposta: tempo fra evento -> effetto. Non dipende solo da OS.

L16:
Algoritmi di scheduling:
1 - FIFO - first in first out(serve). Inserimento ed estrazione sono O(1) costanti. E' molto efficiente ma non ha potere di ottimizzazione.
NB: è beneficio collettivo se i processi lunghi vanno dopo processi veloci.
Non garantisce ta mediobasso o ottimo / ha effetto convoglio / + efficienza

L17:

SJF - Shortest Job First. Quando l'obiettivo dello scheduler è avere TA medio basso non si usa il FIFO. (nb -> ta basso non vuol dire che tutti hanno ta basso) SJF prende il processo che ha il tempo di esecuzione più breve e lo ordina in modo crescente.

Problema: si presuppone che si conosca a priori la durata del processo, cosa non possibile. L'algoritmo è quindi non applicabile nella realtà, ed i jobs non si usano +.

Inserimento SJF O(log n) ed estrazione O(1)

Props: + ta medio ottimo - basato su conoscenza a priori - non usabile

Si può rendere usabile il SJF -> guardo passato per predire futuro. I processi tendono a comportarsi nello stesso modo.

Si fa una media esponenziale (2 dati 2 molt 1 somma)

Con tn burst, Tn previsione burst n-esimo e Tn+1 previsione per la volta dopo.

Tn+1 = a tn + (1-a)Tn con a random (0,1) che da peso alle considerazioni. "a" si tiene in media verso 1/2.

L18:

Scheduling a priorità: 0 Max …. K-1 bassa

Sceglie quello con la priorità minima. Se assegnamo le priorità in base al burst di cpu otteniamo SJF! L'alg di priorità generalizza SJF.

In generale le priorità possono essere:

Interne al sys -> calcolate da SO, su indici del processo (PCB)

Esterne al sys -> sys admin, utente, background

* ha prelazione ! Può anche non esserci ma aiuta l'efficienza. La prelazione può portare però anche a problemi -> SO deve avere meccanismo per evitare la starvation (processo priorità bassa non viene mai eseguito)
* Si introduce process aging -> più tempo passa in coda ready più > priorità.

L19:

Alg. Scheduling -> **Round Robin:** (molto usato, tuttora)

Condivide la disponibilità di calcolo della cpu fra i vari processi. Implementa il time sharing. Tutti i processi hanno al stessa priorità, ognuno ha una fetta di tempo uguale. **CPU -> suddivisa in parti uguali per tutti i processi** con Q = quanto di tempo. La coda è circolare.

Vantaggi: +prelazione, sistema interattivo, ottimi tempi di risposta. La scelta di Q è molto importante: troppo alto non guadagno molto, rischio come fifo.

Troppo basso: ottimizzo ma frammento troppo e ho CS pesante.

- no convoglio - no starvation

L20:
Scheduling a più livelli = estensione di un algoritmo a priorità.
Da Liv 0 a Liv k-1 -> lavoro organizzato su più code, ogni coda racchiude processi con le stesse caratteristiche. Code diverse -> priorità diverse.
Le code diverse possono anche essere gestite con diversi alg di scheduling.
ES:
L0 -> processi foreground (i/o) -> round robin
L1 -> processi background (< prio) -> FIFO con prelazione
Lo scheduling a più livelli sceglie su due fasi:
Prima identifica la coda di processi con cui lavorare -> attiva l'lalg di scheduling della coda -> concetto di retroazione (feedback) = si da la possibilità di cambiare la coda di appartenenza quando si viene reinseriti.
- Può soffrire di **starvation** (while true).

L21:
Scheduling real-time:
Hard-rt: operazione terminata < tempo
Soft-rt: processi critici (vincoli temporali meno importanti).
Ha complessità O(1) come scheduler.
I processi critici con priorità alta possono avere più importanza del SO stesso.
Anche se la prelazione sul SO è pericolosa/ S1- incrementiamo la priorità così finisce prima S2- Identifico punti sicuri nel kernel in cui posso sospendere (difficile)

L22: (da rivedere)
Memoria condivisa -> comunicazione fra processi -> definiti in sistema multitasking. (vale anche per un processore single core dove usiamo il time sharing). **SO** blocca ogni forma di interazione fra i processi (default) -> illusione per ogni processo che le risorse siano tutte sue.

Processo
-> indipendente -> non condivide dati con altri / exec non influenzata
-> cooperanti -> scambiano dati fra loro o si influenzano

Vantaggi: parallelismo e modularità.
Meccanismi:
Sincronizzazione -> sospende l'esecuzione in modo da aspettare qualcun altro
Scambio dati -> attraverso mem condivisa, scambio messaggi etc:

L23:
Mailbox -> su appunti

L24:
Sezione critica:
A basso livello, due processi diversi potrebbero entrare in una sezione di codice (che dovrebbe essere atomica) dove entrambi accedono e modificano una variabile che potrebbe influenzare il loro funzionamento. Quando entrambi i processi la

chiamano allo stesso tempo -> **RACE CONDITION**. I context switch potrebbero poi portare a risultati inaccurati ed errori di calcolo.
Creiamo la sezione critica -> Sys call esordio - SC - Sys call epilogo -> durante la sezione critica non sono consentiti CS e il codice della porzione (più piccola possibile) verrò eseguito atomicamente.

Props sezione critica:
1) Mutua esclusione: non è possibile che 2 processi eseguano SC insieme.
2) Attesa limitata: quando un processo vuole entrare in SC gli altri possono passare davanti N volte. Esiste un N max per evitare la starvation.
3) Progresso: Solo i processi interessati a entrare in SC concordano chi ci entra.

L25:
Alg sezione critica:
1 implementazione -> basta impedire a sistema di eseguire context switch durante la SC -> disabilito momentaneamente gli interrupt di sistema. Scelta pesante, causa no risposte i/o e rischia instabilità sistema.
Non garantisce l'attesa limitata.
2 implementazione -> variabile di lock var 0 sc vuota var 1 sc occupata. Non garantiscono la mutua esclusione.
3 implementazione -> alternanza stretta -> non flessibile, non garantisce il progresso.

Algoritmo di Peterson:
Garantisce tutte e 3 le proprietà ma ha busy waiting. Si usano dei flag e una variabile turno nel codice che indicano l'interesse ad entrare nella sezione critica.
If (flag == 1 && turno ==1) -> sc / busy wait.

L26:
Algoritmo di Baker: estensione Peterson
Viene chiamato il processo che può entrare nella sua sezione critica.
E' possibile la race condition. Rispetta tutte e 3 le proprietà.
Si utilizzano 1 array e 1 var=> NUM[N] che è il biglietto, con i turni per entrare nella sezione citrica e FLAG[N] che descrive l'interesse a prendere il biglietto.
La race condition può accadere quando due processi cercano di prendere il biglietto, e possono entrambi ricevere lo stesso numero.

L27:
Appunti

L28:
Tutti gli algoritmi visti finora utilizzano busy waiting -> rovina prestazione e perde tempo. [APPUNTI]

L29:
Semafori: sono oggetti utilizzati per la sincronizzazione. Sono una generalizzazione delle chiamate di sleep() e wakeup(). Contiene un contatore di sveglie.
Problema-> più processi possono fare down contemporaneamente. Il contatore è una variabile condivisa -> il semaforo richiede una sua sincronizzazione.
Possiamo dare il controllo del codice al SO -> c'è un po' di busy waiting.
Max 1 proc in SC -> elimino il busy waiting.
Tutte e 3 le proprietà della sezione critica vengono rispettate.

Semaforo produttore consumatore.

consumatore:
pieni.down()
If(pieni==0) sleep()
Else consuma()
Vuoti.up();
	Produttore:
	vuoti.down
	if(vuoti==0) sleep()
	Else produci
	pieni.up();

L30: es appunti

L31:
Algoritmo dei 5 filosofi
Concorrenza di alcune azioni con controllo ad accesso delle risorse.
Concetto: 5 filosofi su tavolo rotondo, 5 bacchette. Per mangiare un filosofo deve poter prendere le bacchette sia a dx che sx.

Impl1 : 1 semaforo per bacchetta -> problema: tutti prendono una bacchetta ma nessuno riesce a prendere l'altra. Tutti si addormentano e la sita va in DEADLOCK.
Per evitare il deadlock:
1 Sol: filosofo mancino -> c'è sempre la possibilità di continuare,
2 Sol: filosofo timido -> se trova una delle due bacchette occupate, rinuncia -> MA se trova sempre la bacchetta occupata rinuncia all'infinito -> starvation.

L32:
Lettori e scrittori: devono funzionare in mutua esclusione. Ruoli divisi ma base di dati condivisa senza mutua esclusione. Implementato con un mutex, un db e int n lettori. C'è pericolo di starvation dello scrittore dato che db.up() potrebbe non essere mai rilasciato.

lettore:
mutex.down
n.lettori++
if(n.lettori==1)db.down
Mutex.up
read();
	Scrittore:
	mutex.down
	n.lettori—
	if(n.lettori==0)db.up
	mutex.up

L33:

Algoritmo del barbiere:

L'idea è che il barbiere serve i clienti uno ad uno e dorme se non ci sono clienti.
1 barbiere (server), N sedie, clienti. No clienti -> barbiere dorme. No sedie -> cliente va via. Cliente si siede, barbiere svegliato dal cliente.
Cli(0), barb(0), mutex(1), int in_attesa= 0

Barbiere:	Cliente:

cli.down
mutex.down
in_attesa—
Barb.up
Mutex.up
taglia()
	mutex.down
	if(in_attesa<n)
	In_attesa++
	Cli.up
	mutex.up
	barb.down
	taglia()
	Else
	Mutex.up

NOTA : COSA E' UN MUTEX:

In informatica il termine mutex (contrazione dell'inglese mutual exclusion, mutua esclusione) indica un procedimento di sincronizzazione fra processi o thread concorrenti con cui si impedisce che più task paralleli accedano contemporaneamente ai dati in memoria o ad altre risorse soggette a corsa critica (race condition).

L34:

Deadlock: particolare problema di sincronizzazione -> processi concorrenti su più risorse. Esiste un insieme di processi in cui ogni processo attende un evento che solo un altro processo dell'insieme è in grado di generare. L'esecuzione dei processi concorrenti si blocca completamente. Il problema è l'acquisizione delle risorse per i processi.

Cond necessarie (non è detto ci sia):

1. Risorse usate in modo esclusivo
2. Possesso e attesa (hold n wait) quando acquisisce una risorsa -> next
3. Assenza di prelazione
4. Attesa circolare -> Pi aspetta Pi+1… Pi+2…

L35:

Grafo di Holt:

E' un modo per osservare una "foto" in un determinato istante del sistema operativo e per capire lo stato di risichiate e di allocazione delle risorse.
Servono due tipi id nodi -> O processo e ■ risorsa
Da vedere su [APPUNTI] per capire meglio

L36:
Algoritmo dei **banchieri**: usato per prevenire i deadlock nell'allocazione delle risorse
Prevenzione del deadlock da parte dell'utente -> bisogna togliere una delle 4 regole del deadlock oppure / Prevenzione da SO (non molto usata) -> se conosco le richieste future.
Rilevazione: periodicamente controlliamo se è avvenuto un deadlock e nel caso kill.

- Richiesta di **risorse** -> so -> ordine di assegnamento, garantisce senza deadlock. Senza richieste il deadlock non è sicuramente evitato.

[APPUNTI] per approfondire

L37:
Memoria e la sua gestione: guardiamo la RAM. Supponiamo di avere multitasking e programmi in coda ready.
Il programma + dati vanno in RAM + mem secondaria.
Si crea una immagine del processo in memoria. A seconda di dove viene allocato, bisogna sistemare i riferimenti a var/dati.
Nella transizione new -> ready allochiamo la memoria e risolviamo gli indirizzi.
Ci sono due tipi di indirizzi:
Simbolici: alto livello, sorgente, in un Array o stack
Relativi: basso livello, compilatore
=> vanno all'oggetto, linker e poi eseguibile. Il loader carica immagine di processo in memoria e si ha assegnamento indirizzi è visione logica.

L38:
Partizioni continue e problemi di allocazione -> [APPUNTI]

L39:
Frammentazione: che è spreco di memoria. Può essere interna o esterna.
Fram **esterna** => è esterna ai processi allocati. Lo spazio totale libero è maggiore di Pn ma diviso in parti minori di Pn. Serve la compattazione (che però è molto costosa).
Fram **interna**: sempre spreco di spazio ma interno alla partizione. P1 occupa meno spazio di quello disponibile, però lo utilizza comunque tutto.

Per risolvere la frammentazione:
Allocazione interna fissa -> SI
Allocazione esterna fissa -> SI
Allocazione interna variabile -> NO
Allocazione esterna variabile -> SI
Useremo quindi una allocazione interna non contigua per risolvere il problema della paginazione.

L40:
Paginazione: tecnica per risolvere la frammentazione. E' una allocazione non contigua. Tutti i SO usano queste tecniche.
2 tecniche principali -> **Paginazione**
-> **Segmentazione**

Supponiamo una locazione contigua = immagine logica del processo mappata nella RAM attraverso una traslazione degli indirizzi.

1) paginazione: consideriamo l'immagine del processo -> la dividiamo in dimensioni predefinite e sempre uguali (dim D scelta hw).

Indirizzo logico = pagina * D + offset -> suddivido in multipli di D
C'è relazione biunivoca tra indirizzo logico e pag + offset.
Prendo le singole pagine e le alloco in punti liveri. Posso anche pre suddividere la RAM in dim D.
Finche ci sono spazi liberi in ram posso allocare. Non c'è più frammentazione.
PERO' ci sono riferimenti alle pagine di cui bisogna tenere traccia -> uso la tabella delle pagine (mmu) dove mi salvo le referenze alla posizione delle pagine.

Indirizzo logico x-> (mmu) indirizzo fisico? T[i] -> frame in cui è la pag i.
Alla CPU non interessa dove è un indirizzo fisico -> fa MMU.

Frammentazione esterna ? NO
Frammentazione interna ? Tollerabile, è limitata.

L41:
Paginazione 2 livello /tlb
Svantaggio della paginazione: se dobbiamo consultare la tabella delle pagine, la lettura è in row e l'operazione è molto costosa.
Per risolvere il problema usiamo l'hw:
Idea 1) usare registri per copiare la tab delle pagine (si evita lettura in ram) ma non in tutti i casi i registri sono sufficienti rispetto al numero di riferimenti che ho.
Idea 2) memoria associativa (forma di caching) -> recupera velocemente quelli che già conosco (translation lookaside buffer) o TLB -> associa le pag + grame che uso più frequentemente li mette in registri veloci.
La tabella delle pagine è pesante e va in ram -> sol: ricorsivamente patiniamo la tabella della pagine -> tabella della tabella = pag a 2 livelli

L42:
Segmentazione: altra tecnica di gestione della memoria, **NON contigua**. Deriva dalla compilazione dell'immagine logica.
Vantaggio: compilazione -> unica fase che conosce il significato dello spazio di memorizzazione che andiamo a richiedere.

Da sola però non è una soluzione alla frammentazione. Nella **paginazione** c'è piccola frammentazione interna e no esterna.
Usiamo: segmentazione patinata -> NO fram esterna e poca interna.
Vero vantaggio -> genero indirizzi automaticamente dal sorgente.

L43:
Memoria virtuale: possibilità in più per gestire la memoria (framm)
+ Già presente negli OS + possibilità di eseguire un processo non tutto in ram
+ Posso eseguire un processo più grande della ram
+ Miglioro le prestazioni.
Cosa è ? Paginazione + swapping (scambio frame tra ram e disco, e aggiorno)
[APPUNTI]

L44:
Selezione ottimale della vittima + evitare l'anomalia di Belady
Page fault: Quando la pagina cercata non è presente nei registri che sto usando (ram) e devo andarla a cercare dove la ho copiata.
Anomalia di belady: Il tasso di page fault con n+1 frame è > del tasso di PF con n frame. Normalmente aumentare gli n frame dovrebbe ridurre il page faulting.
[APPUNTI]

L45:
Altro algoritmo per la **selezione della vittima** - **Orologio** - algoritmo ottimale.
La pagina usata più in futuro è la vittima -> funziona però solamente se conosciamo il **futuro** (mai). Per renderlo usabile dobbiamo approssimarlo e usiamo la **LRU**.
Lease recently used = lo rendiamo al **passato**, la pagina più vecchia che non abbiamo usato viene scelta come vittima.
[APPUNTI] per approfondire

L46:
Allocazione frame:
Su N frame di un processo, non è conveniente allocarli tutti in memoria, ma nemmeno troppo pochi. Il meglio sta vicino al minimo necessario, ma non troppo.
N processi, M frame = omogenea Pi = M/N
NB: non è vero che più memoria alloco, più me ne serve per elaborare.
La vittima può essere selezionata localmente interna al processo oppure globalmente in tutta la RAM.

L47:
Thrashing: evento che può verificarsi se ci sono troppe richieste da parte dei processi e non c'è abbastanza memoria libera -> ogni processo andrà in PF (non ha abbastanza frame) e tutti i processi finiscono in wait e pochi eseguono. Ne consegue un crollo delle prestazioni, tutto in palla. Bisogna evitare di riempire la RAM al massimo.

L48:
Gestione dell' I/O:
Le periferiche IO sono tante e diverse]-> gestite con astrazione.
HW fa il lavoro. Sulla periferica inserita nella porta dell'elaboratore c'è un controller che converte i comandi in azioni.
SW aiuta = programma che interpreta nel modo corretto i comandi, se c'è il concetto di **driver**
Anche il kernel fornisce dei servizi di:
Scheduling (ordine e ottimizza i tempi d'attesa),
Buffering, caching, spooling (memorizza coda richieste e crea ordine) ed in più gestisce gli errori dell' HW tramite codici di errore.

L49:
Gli HDD sono strutturati in Settori, blocchi e tracce. Il T di accesso = T ricerca + latenza. La banda = bytes trasferibili / unità di tempo. Il T trasferimento= bytes da trasferire / banda.
Possiamo numerare i blocchi e rimappare quelli difettosi. Vicino ai settori possiamo creare settori di scorta.
Anche la testina di lettura va ottimizzata e ci sono degli algoritmi per migliorare il più possibile.

1) **FIFO** -> ordine delle richieste così come sono
2) **SSTF** -> shortest seek time first -> va a prendere per primi quelli con seek time più brevi
3) **SCAN** -> Sposta la testina fra min e max e serve le richieste se ne trova in coda.
4) **C-SCAN** -> è come lo scan verso una parte ma torna indietro velocemente dove non serve nessuno. TA richieste più basso -> movimento a vuoto
5) **LOOK** -> Fa come scan ma si ferma all'ultima richiesta da servire e torna alla prima-> evita di arrivare ai bordi
6) **C-LOOK** -> fa come il c-scan ma arrivata all'ultima, torna direttamente al min della coda.

Nessuno di questi è ottimale ma look e c-look sono buoni e vengono usati.

L50:
File system e amministrazione del disco -> va configurato -> disco formattato:
-Ad alto livello (settori, tracce, blocchi)
-A basso livello -> organizzazione logica <-> file system
Il disco ha un blocco di avvio, usato per avviare il SO.
Il disco ha delle partizioni logiche -> permette di usare 1 disco come se fossero molteplici e indipendenti (anche dual boot è ammesso)

File system: è la visione logica della memorizzazione. Contiene metadati (nome, identificatore, dimensione ecc), directories, partizioni, insieme specifiche e operazioni per gestione e modifica.
L' SO usa una forma di caching sui file aperti per risparmiare tempo.
Le strutture delle directory sono principalmente di due tipi.
Foresta di alberi per winzoz (C:/, D:/, ecc) e albero per UNIX (/).
La struttura è DAG(grafo diretto aciclico) -> convergenze sui vari nodi/file -> foglia.
Salta la proprietà di singolo genitore e crea problemi per i puntatori -> più genitori possono avere lo stesso figlio e quindi più percorsi portano allo stesso file.
Si usano hard link e link simbolici -> [APPUNTI] per app.
Link simbolico: si sceglie a quale foglia puntare qualsiasi sia la strada scelta
Hard link: crea un riferimento al file identico all'altro. Indistinguibile. 2 riferimenti allo stesso file, da due strade diverse.

L51:
Come viene implementato il file system:
A **livelli**: Fisico -> FS base (r/w blocchi) -> Organizzazione dei file -> FS logico (gestione metadati).

Apertura di un file -> accesso a dir corrente -> identifica FCB che contiene altri file
Realizzazione directory -> spesso sono alberi -> liste nome e *ptr a FCB
Allocazione dei file […]
+ APPUNTI PER COMPLETEZZA

COSE DA RICORDARE:

I 5 filosofi:
E' considerato un classico problema di sincronizzazione, perché rappresenta una vasta classe di problemi di controllo della concorrenza.
L'idea consiste nell'avere 5 filosofi che o pensano o mangiano e 5 piatti di cibo per ognuno. Ci sono però 5 forchette e un filosofo può mangiare solo se ha entrambe le forchette in mano. Una volta presa una forchetta, la tiene. Questo spesso porta in deadlock.
Ci sono diverse soluzioni: Rappresentare ogni bacchetta con un semaforo -> che però può comunque portare a deadlock.
Oppure, solo 4 filosofi possono stare insieme a tavola invece di 5; un filosofo può prendere le sue bacchette solo se sono entrambe disponibili (filosofo timido); un filosofo dispari prende prima la bacchetta di sx poi quella a dx, mentre gli altri pari prendono prima a dx e poi a sx (filosofo mancino)

I Semafori:
Un semaforo è un tipo di dato astratto gestito da un sistema operativo multitasking per **sincronizzare l'accesso a risorse** condivide fra processi o thread. E' composto da una variabile intera, la sua interfaccia e una coda di processi. E' stato ideato da **Dijkstra**.
Molto usati sono i semafori binari con valori possibili 0 e 1. Tale semaforo viene definito **mutex**. Un semaforo può essere modificato da parte del codice utente solo con 3 chiamate si sistema:
Inizializzazione: sem inizializzato con valore intero positivo.
Down(): semaforo decrementato. Se ha valore negativo dopo il decremento, processo sospeso e accodato.
Up(): semaforo incrementato. Se ci sono processi in coda, a seconda del tipo di scheduling, uno viene tolto e messo in ready.
Un semaforo binario garantisce la **mutua esclusione** nell'accesso a una risorsa semplice. Si chiama UP prima usare la risorsa e DOWN dopo averla usata.

Baker:
E' una estensione dell'algoritmo di Peterson: viene chiamato il processo che può entrare nella sua sezione critica. Ogni processo richiede un "biglietto" per entrare nella sezione critica e quando la variabile del turno corrisponde con quella del biglietto, il processo è autorizzato a entrare in sezione critica.
Rispetta tutte e 3 le proprietà della sezione critica.
Può esserci race condition quando due processi richiedono il biglietto insieme, e gli può venire assegnato lo stesso numero.

Banchieri:
E' un algoritmo usato per prevenire il deadlock durante l'assegnazione delle risorse. Si usano diverse strutture dati che codificano lo stato di assegnazione del sistema. Il grafo di assegnazione (di Holt) non è sufficiente, perché ci sono più istanze di ciascun tipo di risorsa. Sia N numero processi ed M numero tipi di risorsa
-Disponibili: vettore lunghezza m indica numero delle istanze disponibili per ciascun tipo di risorsa.
-Massimo: matrice NxM definisce la richiesta massima di ciascun processo.

-Assegnate: Matrice NXM definisce numero delle istanze di ciascun tipo di risorsa attualmente assegnate a ogni processo.
-Necessità: matrice NxM indica la necessità residua di risorse relativa a ogni processo.

Orologio:
E' un algoritmo per la selezione ottimale della vittima: last recently used page gets thrown away. Look at the past to try and predict the future.

La Segmentazione in detttaglio:
La segmentazione, in ambito informatico, è una comune tecnica di gestione della memoria che suddivide la memoria fisica disponibile in blocchi di lunghezza fissa o variabile detti *segmenti*. Uno dei possibili fini di tale tecnica è l'implementazione di meccanismi di protezione della memoria. Un'altra tecnica comune che assicura la protezione fra processi diversi è il paging.
In un sistema operativo che usa la segmentazione, un indirizzo di memoria contiene una parte che identifica un *segmento*, e una parte che specifica l'offset entro il segmento indicato.

Oltre all'indirizzo fisico ed alla lunghezza, ad un segmento può essere associata una combinazione di permessi in base ai quali si determina a quali processi è consentito o negato un certo tipo di accesso. In questo modo ad esempio è possibile distinguere segmenti di programma, di dati e di stack. Il gestore della memoria può così fare in modo che da un segmento di programma vengano caricate solo istruzioni (e non, ad esempio, dati) o che le informazioni caricate da un segmento di dati non vengano interpretate come istruzioni per il processore. Inoltre, un segmento può possedere un flag che specifica se il segmento stesso è presente nella memoria principale o se va recuperato da un supporto di memoria secondaria prima di essere utilizzabile. Se un processo cerca di accedere ad un segmento che non è attualmente disponibile nella memoria principale, si verifica un'eccezione hardware che permette al sistema operativo di caricare il segmento d'interesse dalla memoria secondaria.

Proprietà sezione critica:
-Mutua esclusione
-Attesa limitata
-Progresso

Condizioni per deadlock:
-Risorse usate in modo esclusivo
-Hold and wait
-Assenza di prelazione
-Attesa circolare