Componenti comuni dei SO

- Process management: gestione dei processi
- Protection system
- Networking: gestione internet
- Command-interpreter system: interfaccia verso l'utente

Gestione dei processi

Un processo è un programma in esecuzione, che richiede risorse tra cui tempo CPU, memoria, file, dispositivi I/O. Si occupa della creazione/sospensione dei processi.

System Call

Interfaccia tra il SO e l'ambiente esterno, disponibile in assembly e altre lingue. Le system call sono chiamate molto generiche che richiedono molti parametri.
I system program, o "utility" sono implementate con le system call.

System Structure

I primi SO erano scritti senza preoccuparsi troppo dei problemi di ingegneria del software, erano istruzioni scritte in un solo blocco (non divise in moduli).
Al giorno d'oggi il SO UNIX è costituito da

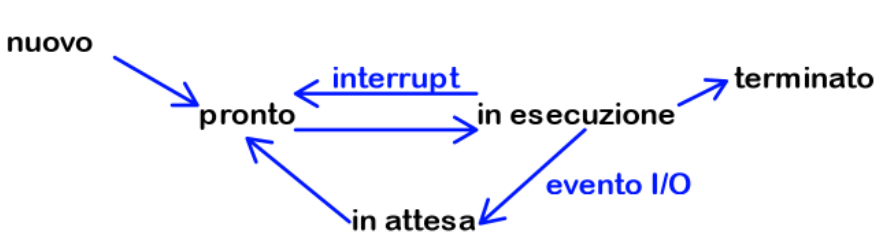
- programmi di sistema
- kernel, che fornisce file system, gestione della memoria e altre funzioni del SO
  I SO sono divisi con una struttura a livelli, questo cambiamento è dovuto alla necessita di una struttura più dinamica dovuta in parte all'avento di internet.

Microkernel

Il kernel conitne solo le funzioni più essenziali: i processi di base e la gestione della memoria.
I servizi principali del SO sono poi aggiunti in cima come moduli che interagiscono sfruttando i microkernel. Questi avvantaggiano l'estensione, portabilità e rendono più facili da modificare i SO.

Processi

Processo: un programma in esecuzione, deve avanzare in maniera sequenziale. Un processo contiene program counter, stack, sezione dei dati. Durante l'esecuzione, un processo cambia stato.



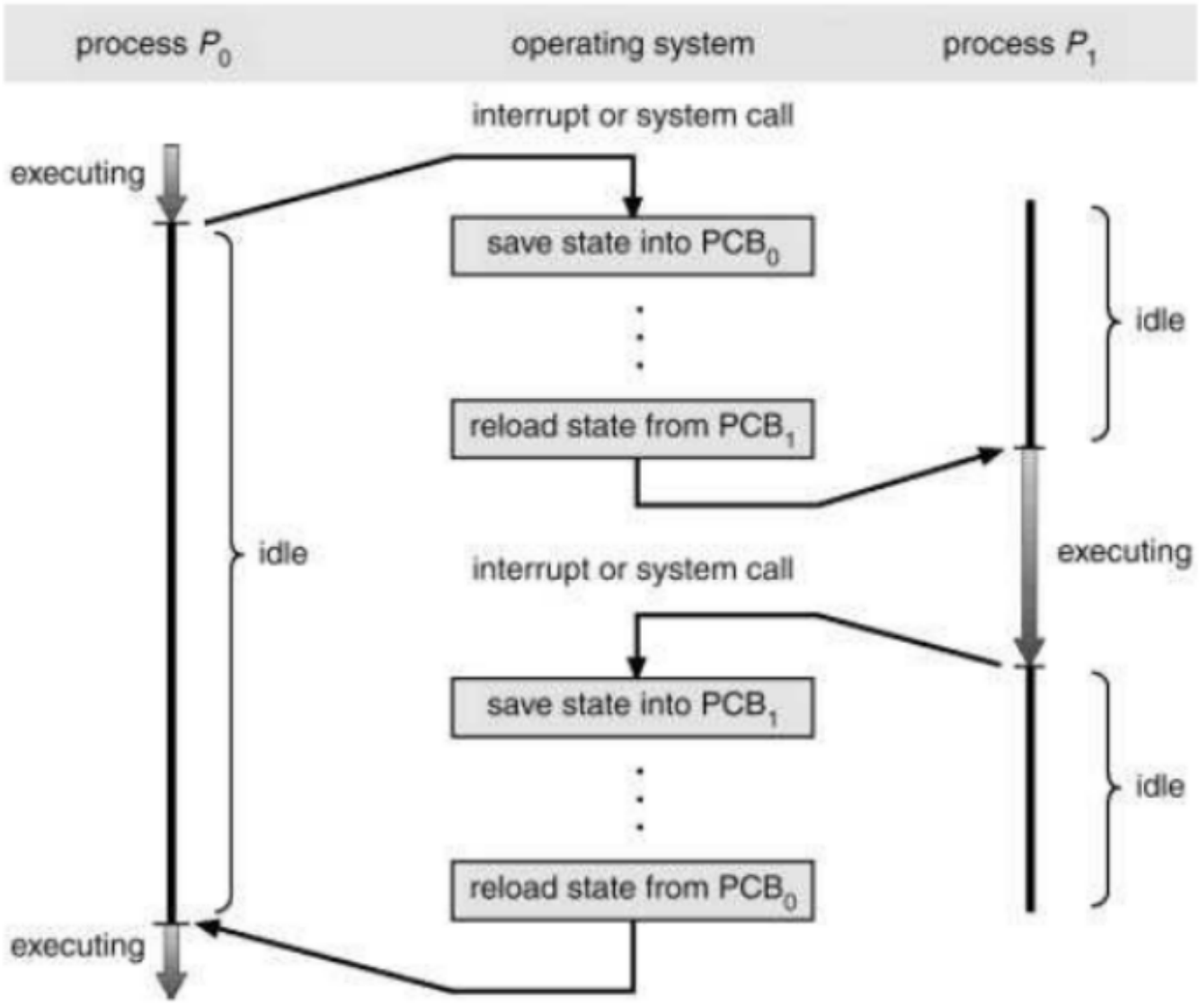
La creazione dei nuovi processi viene gestita dal SO. Quando terminano tutti i dati in memoria non servono più e vengono cancellati.
Un processo in attesa può solo diventare ready. Verrà poi deciso se potrà essere eseguito o no.

Process Control Block (PCB)

Gestisce tutte le informazioni relative ai processi, è in grado di gestire i context switch (passaggio da un processore ad un altro).
Bisogna assicurarsi che dopo che c'è stato un context switch, il processo può continuare come prima.
Bisogna tenere traccia di questo momento dell'esecuzione per ripartire da esso salvando i contenuti del processo sul PCB.

Esecuzione Context Switch

- Durante l'esecuzione di P0 deve essere effettuato un context switch
- Lo stato di P0 viene salvato nel PCB di P0
- Vengono lette dal PCB di P1 le istruzioni per far riprendere il P1
- C'è un'altra interrupt, si salva lo stato di P1 nel suo PCB e si carica lo stato di P0 dal suo PCB.



Creazione e terminazione di processi

Molti processi (di sistema) sono creati quando la macchina viene avviata oppure un utente può richiedere la creazione di un nuovo processo. Se due processi lavorano sulle stesse risorse, queste vengono duplicate solo se necessario.
La terminazione dei processi può essere di vari tipi:

- normale
- dovuta ad un errore
- dovuta a gravi errori (Fatal Error)
- Il processo viene sovrascritto da un altro processo(Kill)

Thread

Processi singoli e multithread

Un thread possiede i propri dati, quelli della procedura che sta eseguendo, ma opera anche su dati globali visibili da tutti i thread. Un thread ha un puntatore che indica dove è arrivato all'esecuzione del processo. Il diagramma degli stati dei thread è uguale a quello dei processi.
I processi singlethreaded sono tipici dei SO tradizionali (i processi sono autonomi). I processi multithread lavorano su dati condivisi, sono leggeri e senza PCB.

Esempio di utilizzo di un thread su un web server

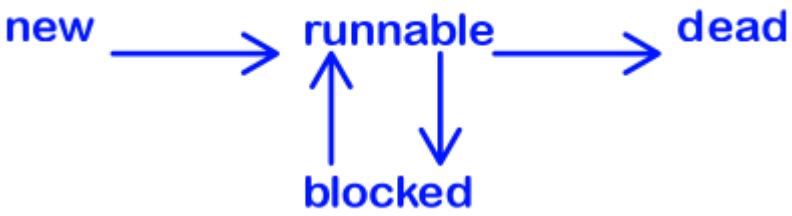
- Dispatcher thread: ascolta le richieste in arrivo
- Worker thread(s): gestiscono la propria richiesta
- Web page cache: contiene le pagine più visitate, se non c'è bisogna andare a fare la richiesta sulla memoria secondaria. Quando si attende risposta dalla memoria secondaria si può attivare un altro worker thread, che può terminare prima se accede ad un elemento della cache.

Vantaggi

- Più reattivo: dato che le richieste sono su più thread bisogna attendere la fine dell'esecuzione del processo e un altra istruzione può essere eseguita subito.
- Sfrutta il parallelismo: diversi thread girano su più processi in parallelo.
- Facilita la programmazione

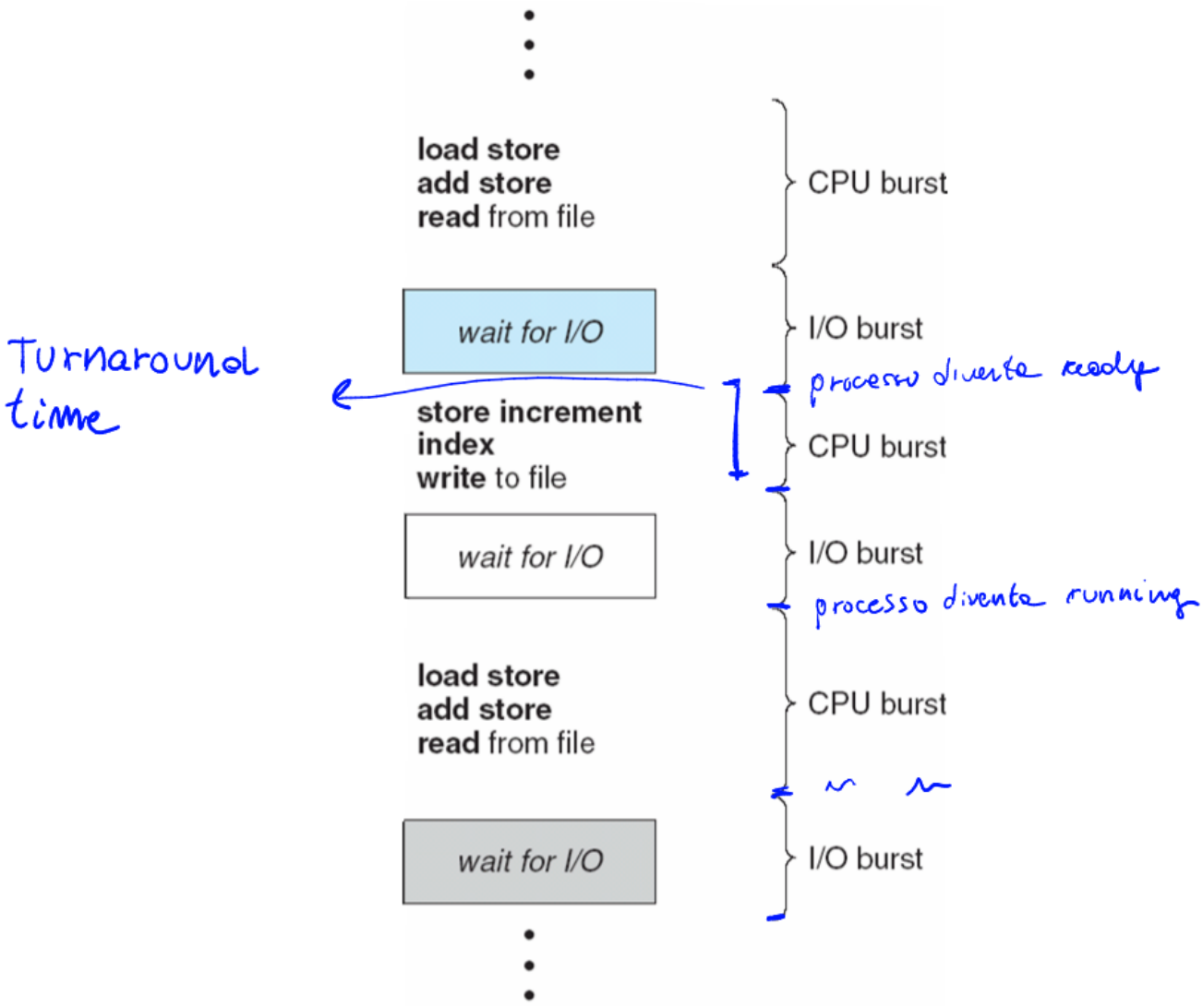
Svantaggi

- Interferenze
  Esistono SO che prevedono l'esistenza dei thread. Ci possono essere dei SO che non usano i thread ma dei processi standard in cui i thread sono aggiunti ai livelli superiori attraverso funzioni di libreria con le quali si gestisce il context switch.



CPU scheduling

Alternanza CPU e I/O nel corso dei processi



Scheduler

Sceglie tra i processi in memoria quelli che sono pronti per essere eseguiti, e alloca nella CPU uno di essi. Queste decisioni sono prese quando un processo:

- Va da running a waiting
- Va da running a ready
- Va da waiting a ready
- Termina
  Questo tipo di scheduling è non preventivo e avviene durante il context switch. Non interviene in un processo che è in esecuzione e non utilizza un quanto di tempo.
- Long time scheduling: sceglie quali processi portare sulla memoria centrale
- Medium time scheduling: avviene con frequenza intermedia, si vogliono togliere dei processi dalla memoria centrale (swap-out) e portarli in memoria secondaria.
- Short time scheduling: scheduling dei processi pronti per essere eseguiti. Si sceglie un numero di processi da far eseguire uguale a quello dei processori.

Criteri di scheduling

- Utilizzo CPU (massimizzare): la CPU deve essere nello stato busy il più possibile
- Throughput (massimizzare): numero di processi completati per unità di tempo
- Tempo di esecuzione di un processo (minimizzare)

- Tempo in attesa nella ready queue (minimizzare)
- Response time (minimizzare): tempo che un processo impiega per passare da ready a running.

Tipi di scheduling

First come fist served

Paragonabile ad una lista FIFO, è la gestione più semplice dei processi ready, ma non minimizza la waiting time.

Shortest job first (SJR)

Si associa ad ogni processo la lunghezza della sua prossima CPU burst. Si usano queste lunghezze per eseguire il processo con il tempo più breve:

- Non prevenzione: il processo in esecuzione non viene interrotto fino a quando non necessita più della CPU, ad esempio quando completa la sua attività o quando attende un'operazione di I/O.
- Prevenzione: un nuovo processo può interrompere quello in esecuzione se il suo tempo di CPU burst è minore del tempo rimanente per il processo attualmente in esecuzione. Il SO interviene per terminare il processo ancora in esecuzione. Questo schema è noto come Shortest-Remaining-Time-First (SRTF): viene dato il massimo della CPU al processo con il tempo di CPU burst rimanente più breve. Funziona se il tempo di context switch è irrilevante.
  Questo processo è ottimale per ridurre il tempo di attesa medio di una serie di processi, tuttavia le proprietà dei comportamenti futuri sono possono essere prevedute e quindi può essere applicato solo in maniera approssimata. Si possono incorre in problemi come:
- Starvation: un processo non riesce ad essere eseguito perché viene sempre messo da parte
- Deadlock: nessun processo avanza, il sistema è completamente bloccato.

> Per assumere il comportamento futuro di un processo si approfitta del principio di località: si stima la lunghezza della prossima CPU burst usando la lunghezza dei CPU burst precedenti. Il tempo recente deve essere valutato di più rispetto a quello lontano. $0 \leq \alpha \leq 1$ è la stima del peso della località, quando è a 0 il passato recente va ignorato e se $\alpha \to 1$ il passato recente diventa molto importante.

Priority scheduling

Un valore di priorità è associato ad ogni processo, alla CPU è associato il processo con la priorità maggiore, e può sempre essere non preventivo o preventivo (se c'è un processo con priorità più alta c'è il content switch). Un problema che si potrebbe incontrare è la starvation, dove i processi di priorità minore rischiano di non essere mai eseguiti. Questo si risolve con l'aging: si aumenta la priorità del processo al passare del tempo.
La priorità dei processi è in genere dinamica. I processi di background hanno priorità minore di quelli in foreground, dove l'utente si aspetta una risposta immediata.

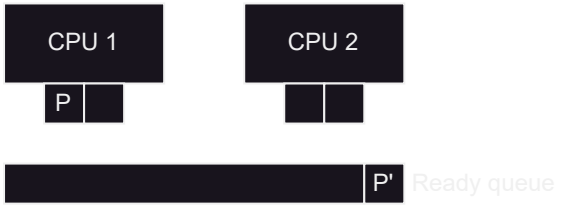
Round Robin

La politica più importante, usato nei sistemi che usano il time sharing. Ogni processo ha a disposizione un quanto di tempo di circa 10-100ms, quando questo tempo scade, il processo è messo in fine della

coda (pre-emption).
Se ci sono n processi nella ready queue e il quanto di tempo è q. ogni processo ha a disposizione $\frac{1}{n}$ del tempo della CPU in blocchi di al massimo q unità di tempo.
Se q è grande il sistema tende al FIFO, se q è piccolo deve essere comunque grande rispetto al tempo del context switch, altrimenti il guadagno di tempo è minimale (la grandezza del quanto di tempo influenza il numero di context switch).

Code Multilivello

La ready queue viene divisa in più code: foreground e background che possono usare politiche diverse. Per scegliere da quale livello prelevare quando c'è la CPU libera si sceglie prima la lista con priorità maggiore (per evitare starving si usa l'aging).



Se P è in cache della CPU1, ma si libera prima CPU2, P va al processore 2 e questo, oltre che ad essere lento e sprecare la memoria nella cache di CPU1 può creare problemi se P nella cache era stato modificato. Quindi si divide la queue unica in una per ogni processore. L'affinità del processore stabilisce se usare load balancing per bilanciare il numero di processi tra i processori.