

Sistema operativo:

L' SO è un insieme di programmi che consentono una visione dell' hardware astratta, molto più facile da usare, una gestione delle risorse, come la gestione della memoria di massa o la gestione dei processi fra le varie CPU. L' SO fornisce anche un insieme di servizi richiamabili tramite system call o tramite interfacce a linea di comando (SHELL) o grafiche. L' SO esegue in modalità kernel, quindi, ha accesso libero alle risorse.

Processo:

Un processo è un programma in esecuzione ed evolve dallo stato di ready allo stato di running. Nel caso il processo debba fare un'operazione di input/output, passerà dallo stato di running allo stato di waiting e quando finirà passerà allo stato ready; se invece viene interrotto dallo scheduler, secondo la politica di scheduling, passerà allo stato di ready in attesa della CPU. Ogni processo ha un PID process identifier, PPID parent process identifier, UID user id, GID group id. Un programma diventa processo quando il suo eseguibile viene caricato in memoria, esso si compone di più parti: text, stack, heap, data. In kernel space i processi sono raccolti in una struttura dati detta process table formata da tanti process control block quanti sono i processi. I processi possono essere processi demoni: eseguono in modalità utente, non hanno un terminale associato ma agiscono in background (demone di stampa). *Linux: lista doppamente concatenata*

System call:

Se un processo deve svolgere un'operazione a basso livello, sull' hardware, deve ricorrere all' aiuto del SO che tramite una system call esaudisce la richiesta del programma. Il programma che richiede un'operazione sull' HW, come una read o una write, chiama una funzione che a sua volta chiama una system call. I processi che girano nello user space devono ricorrere alle system call per poter interagire con l' HW, l' SO che gira in kernel mode non ne ha bisogno perché ha già accesso a tutto.

Mount:

Permette la fusione di file system in un unico file system.

Fork:

Il processo figlio esegue lo stesso codice del padre, ma in uno spazio degli indirizzi diverso. Il processo figlio eredita dal padre: i file aperti dal padre, il gestore dei segnali, ID utente e di gruppo... Il processo figlio differisce dal padre per: valore di PID, descrittori dei file diversi, segnali pendenti, tempo di CPU. Linux per implementare la fork utilizza la tecnica del copy-on-write, cioè che un file aperto in lettura dal padre non viene copiato dopo la fork ma solo nel caso uno dei due richieda l' accesso in scrittura.

Processi zombie:

Nel caso un processo figlio muoia autonomamente tramite la exit() ed il padre, ancora attivo, non ha effettuato la wait(), il processo figlio prende il nome di processo zombie, le risorse utilizzate vengono rilasciate tranne la entry nella process table così che se il padre successivamente farà la wait() avrà lo stato di terminazione del figlio. Se il processo padre morirà normalmente allora il processo zombie morirà anch' esso; se il padre dovesse morire in modo anomalo allora il processo zombie verrebbe ereditato dal processo init.

Processi orfani:

Se il processo padre dovesse morire prima del figlio, allora il processo figlio diventerà un processo orfano. Affinché il processo orfano non diventi uno zombie per sempre, perché non ci sarà nessuno interessato alla sua morte, sarà adottato da un processo dello stesso gruppo o dal processo init.

mantieni directory di lavoro

Exec: non interviene nei file utilizzati dal process chiamante, mantiene PID, PPID, GID, t. di CPU
Si tratta di una famiglia di funzioni che fa sì che un processo già in run esegua un nuovo codice. Non fa altro che eliminare tutte le risorse che aveva il determinato processo e crearne di nuovi eseguendo un nuovo file eseguibile, passato come parametro all' exec. L' exec non crea un nuovo processo ma sostituisce i dati del text e lo stack con un nuovo programma. Si differenziano fra:

Execl (pathname, list₁, ..., NULL);
Execlp (name, list₁, ..., NULL);
Execle (pathname, list₁, ..., char V[][]);

Execv (path, char V[][]);
Execvp (name, char V[][]);
Execve (path, char V[][], char e [][]);

La colonna di sinistra funziona con le liste (quindi prenderà in input oltre il file eseguibile un numero variabile di argomenti terminati da NULL), la colonna di destra funziona con i vettori (quindi prenderà in input oltre il nuovo eseguibile un array di stringhe); la prima riga vuole l' intero pathname dell' eseguibile, la seconda riga vuole solo il nome dell' eseguibile perché lo prende dalla variabile d' ambiente PATH, la terza riga oltre il nuovo eseguibile ed un array o una lista prenderà come parametro un array di stringhe che definirà il nuovo ambiente dell' eseguibile.

controlli sul 2° e 3° ambito del t

Thread: attributi: detached state, scheduling policy, scheduling parameter, inheritance ad attributi regola
Un thread è la più piccola entità di esecuzione che viene schedulata dal kernel, un processo può avere più thread al fine di realizzare un obiettivo comune. Il vantaggio dei thread sta nel fatto di poter decomporre le varie attività interne ad un processo grazie al fatto che condividono le risorse e lo spazio degli indirizzi con tutti i thread di un processo. I thread di uno stesso processo condividono: lo spazio degli indirizzi del processo, le variabili globali ed i file aperti; differiscono invece dello stack, dei registri, come il program counter che conterrà l' indirizzo di memoria dell' istruzione successiva che differirà da quella degli altri thread perché ognuno svolge attività diverse, dell' identificatore e dello stato.

Differenza thread/ processi:

I processi hanno spazi degli indirizzi separati fra loro e comunicano attraverso meccanismi di IPC; non ci sono meccanismi di protezione fra thread a differenza dei processi. A differenza dei processi non c'è competizione fra thread ma collaborazione; il context switch fra thread di uno stesso processo è più veloce rispetto al context switch di un processo. I processi sono come raccoglitori di risorse dove sono i thread a fare tutto il lavoro collaborando.

Thread users space/kernel space:

I thread in user space sono implementati a livello user quindi lo scheduler vede un processo, a livello utente sarà il sistema a run time ad occuparsi di fare lo scheduling dei thread; i vantaggi dei thread in user space sono: l' efficienza di esecuzione e la scalabilità dei thread in quanto in user space la memoria è notevolmente maggiore alla memoria del kernel. I suoi svantaggi sono: le system call bloccanti che → a: system call bloccano tutti i thread in quanto lo scheduler vede quel thread come un processo; il page fault, cioè il non bloccanti blocco del processo, quindi di tutti i thread, causato dalla mancanza della pagina in memoria; lo scheduling dei thread. I thread in kernel space vengono tutti salvati e gestiti dal kernel in una thread table, non occorre quindi un sistema a run time per la gestione e il blocco di un thread su una system call o una page fault non è un problema perché sarà il kernel ad occuparsi di fare il context switch su un altro thread

non misfisi in kernel space perché i thread
non utili quando ci sono processi I/O bound
non quindi fanno system call bloccanti per
ci meglio in kernel space

Race condition/mutua esclusione:

Situazione in cui l' esito dell' esecuzione di un insieme di processi che condividono una risorsa comune dipende dall' ordine in cui vengono eseguiti, per cui il risultato sarà impredicibile. La race condition si può evitare tramite la mutua esclusione, cioè far sì che quella risorsa o quel gruppo di risorse sia accessibile da un solo processo alla volta; per far ciò non possono esserci due processi in regione critica, cioè nella parte ~~unica~~ di codice in cui si accede alla memoria condivisa, nessun processo può attendere per sempre l' accesso in regione critica, non si devono fare assunzioni sulla velocità e nessun processo può bloccarne un altro al di fuori della regione critica.

2^o processi in reg.
non attendere
per sempre,
regione critica,
nessun processo
blocca un altro

Produttore-Consumatore/Semafori:

Nel problema produttore consumatore abbiamo un problema nella mancanza di sincronizzazione dei due processi; perché, supponendo il buffer sia vuoto, nel caso il consumatore abbia appena fatto il controllo sul buffer ed esso è vuoto subito dopo deve andare in sleep() fino al risveglio da parte del produttore, ma supponiamo che esso venga interrotto, prima della sleep e dopo il controllo del buffer, dallo scheduler e venga avviato il produttore esso creerà un item nel buffer e sveglierà il consumatore; ma siccome il consumatore non era ancora andato in sleep() il risveglio andrà a vuoto e quando lo scheduler ritornerà al consumatore esso andrà in sleep() e non verrà mai più risvegliato. Per risolvere ciò vengono usati i semafori di Dijkstra, cioè meccanismi di sincronizzazione fra processi. Nel problema produttore consumatore ne vengono utilizzati 3, 2 per la sincronizzazione e 1 per la mutua esclusione. Si parte inizializzando i due semafori di sincronizzazione, uno con il numero di elementi vuoti del buffer (empty), l' altro con il numero di elementi pieni nel buffer (full); dopodiché il produttore effettua la down() su full e l' up() su empty (down e up sono due funzioni atomiche che svolgono il controllo e l' assegnamento in un' unica istruzione di CPU; quindi, non interrompibili dallo scheduler) ed il consumatore effettua la down() su full e la up() su empty.

IPC (inter process communication):

I Canali di ipc sono oggetti creati dal kernel che: risiedono permanentemente in memoria kernel e accessibili tramite identificatore generato dal kernel stesso. Sono 3 i sistemi di comunicazione: semafori, shared memory e code di messaggi. Le system call per usarle sono rispettivamente : semget, semctl, semop; shmget, shmat, shmdt; msgget, msgsnd, msgrcv, msgctl. I semafori sono meccanismi di sincronizzazione per i processi che vogliono accedere a delle risorse condivise, in Linux sono implementati tramite array di semafori e possono assumere valori ≥ 0 . La memoria condivisa è una zona di memoria, nel kernel, condivisa alla quale possono accedere più processi. Un esempio tipico d' uso di semafori e memoria condivisa sono i processi produttore consumatore che rispettivamente accedono ad un buffer di memoria condiviso e creano od eliminano item dal buffer condiviso. Per fare ciò è necessario un sistema di sincronizzazione con 3 semafori: uno per la mutua esclusione al buffer e due che avranno rispettivamente il numero di item pieni e vuoti del buffer. Una coda di messaggi è una struttura a cui si possono inviare o ricevere messaggi tra processi, essa comprende anche la sincronizzazione fra ricezione ed invio di messaggi quindi senza dover implementare altre strutture per la sincronizzazione. Un esempio d' uso è lo scambio di comandi fra due processi, in questo caso sarebbe consigliato usare una shared memory essendo i messaggi molto piccoli e frequenti. È invece consigliato una shared memory, quindi insieme ai semafori, per condividere quantità significative di dati

semafori: nme., array, > 0 , max = 1 binari, max ≥ 1 generali

memoria: zona memoria kernel

code messaggi: struttura invio e ricezione msg, con sincronizzazione fra invio e ricezione vantaggio di priorità free memory

Semafori:

Sono una struttura di IPC di system V usata per la sincronizzazione fra le operazioni di più processi. Essi assumono solo valori interi > 0 : se il valore massimo del semaforo sarà > 1 allora si parla di semaforo generalizzato, se il valore massimo sarà $= 1$ si parla di semaforo binario. Questa variabile va incrementata o decrementata tramite le due funzioni dei semafori `up()` e `down()`. La caratteristica principale di queste due funzioni è l' atomicità cioè che la fase di test e set del semaforo sia interrompibile dallo scheduler. Le due funzioni seguono dunque la politica "all or nothing" ossia o sono svolte tutte insieme o nulla; ciò evita la race condition e fa che quando un processo ha iniziato il test sul semaforo nessuno può accedervi se non dopo che l' operazione si è conclusa.

Socket:

Quando la comunicazione avviene tra processi che non risiedono sulla stessa macchina vengono usati i socket. Essi permettono una comunicazione bidirezionale interfacciandosi con lo stack protocollare TCP/IP che permette la comunicazione attraverso Internet.

Scheduling:

Lo scheduler è un modulo dell' SO che decide quali tra i processi in ready eseguire quindi mandarli in run. Lo scheduler si basa su degli algoritmi di scheduling preemptive o non preemptive; preemptive, lo scheduler permette l' interruzione dei processi in run per dare la CPU ad un altro task; non preemptive, lo scheduler prende un processo e lo lascia in run fino al rilascio volontario della CPU o perché si blocca. Gli obiettivi di uno scheduler sono:

- Equità: equità fra i vari task
- Efficienza: utilizzo della CPU efficiente
- Tempo di risposta: minimizzare il tempo di risposta, cioè il tempo di completamento del task e il tempo di prima attivazione (utenti interattivi)
- Tempo medio di attesa dell' output: minimizzare il tempo medio di attesa dell' output
- Throughput: massimizzare il numero di task completati

Questi obiettivi sono incompatibili fra loro, cioè se si vuole migliorare un aspetto fra questi cinque ne deve andare a discapito degli altri. La decisione di scheduling viene presa quando qualcosa nel sistema cambia come per esempio la creazione o la terminazione di un processo. Lo scheduling nei sistemi interattivi avviene solitamente con un algoritmo: ROUND ROBIN scheduling, consiste nell' assegnare ad ogni processo un quanto di tempo superato il quale lo scheduler fa preemption e mette il processo che era in run alla fine della coda dei processi in ready to run ed esegue quello subito dopo il processo iniziale. Per effettuare preemption si deve svolgere il context switch cioè il salvataggio di tutti i registri della CPU cosicché quando quel processo tornerà in run non noterà alcuna differenza da quando è stato interrotto; il quanto di tempo quindi non deve essere né troppo vicino alla durata del context switch, perché non ci sarebbe nemmeno il tempo di fare il context switch che già deve cambiare processo, e nemmeno troppo lontano per evitare che la CPU rimanga troppo tempo sullo stesso processo e ci sia un alto tempo di risposta. Nei sistemi batch vengono invece spesso utilizzate algoritmi FIFO (first in first out)

Signal/Sigaction:

Un processo che riceve un segnale avrà tre opzioni:

- Consentire che avvenga l' azione di default
- Ignorare il segnale, tranne SIGKILL e SIGSTOP che non possono essere ignorati
- Gestire il segnale tramite una funzione detta handler che ogni volta che si presenta uno specifico segnale lo intrappa, tranne SIGKILL e SIGSTOP

La funzione che permette di fare ciò è la signal che prende come parametri il segnale rappresentato da un intero e la funzione handler, che può essere un puntatore ad una funzione, oppure le costanti SIG_DFL o SIG_IGN che rispettivamente fanno che venga svolta l' attività di default o venga ignorato il segnale.

Usando la funzione signal, però, ci si imbatte in problemi di inconsistenza causati dalla race condition, cioè ogni qualvolta si rimposta il gestore dei segnali all' interno della funzione gestore così che la gestione si mantenga ogni qualvolta si riceva il segnale, c' è un breve intervallo di tempo in cui se si riceve lo stesso segnale esso produrrà l' azione di default quindi creando inconsistenze. Un' interfaccia più robusta della signal è la sigaction che prende come parametri il segnale rappresentato sempre da un intero, una struct di tipo sigaction che sarà usata per settare l' azione da intraprendere alla ricezione del segnale ed un' altra struct di tipo sigaction che conterrà il vecchio gestore del segnale. A differenza della signal i segnali gestiti dalla sigaction non verranno resettati all' azione di default ma avranno sempre lo stesso gestore fino ad un' altra sigaction o alla morte del processo. La maschera dei segnali che viene settata nella struct sigaction permette di catturare i segnali e consegnarli solo quando l' esecuzione non è nel gestore dei segnali; viene così evitata la race condition.

Evitare race condition: riguarda che consente i regnoli solo quando non è nel gestore

Deadlock:

Supponiamo due processi che vogliono accedere ad una risorsa condivisa, per poter accedere alla risorsa bisogna richiederla usarla ed infine rilasciarla, se la risorsa è impegnata allora il processo che la richiede attende che si liberi o viene bloccato fino a quando la risorsa non viene rilasciata. Se si verificano le seguenti condizioni:

- Condizione di mutua esclusione: ciascuna risorsa può essere impegnata da un solo processo alla volta
- Condizione di possesso e attesa: se un processo detiene già alcune risorse può richiederne di altre e mettersi in attesa di esse
- Condizione di No preemption: risorse già impegnate in un processo non possono essere rimosse forzatamente
- Condizione di attesa circolare: la più importante, ci deve essere un' attesa circolare di almeno due o più processi in cui ciascun processo è in attesa di una risorsa detenuta dal processo successivo della catena

Per prevenire i deadlock va attaccata almeno una condizione delle 4:

- Prevenzione alla mutua esclusione: possono essere metodi di spooling, cioè un processo demone che detiene ed usa solamente lui la risorsa senza richiederne altre; con il problema però che non tutte le risorse possono utilizzare metodi di spooling
- Prevenzione alla condizione di possesso e attesa: far sì che tutte le risorse disponibili vengano date al processo che ne richiede meno in modo che termini e rilasci tutte le risorse già impegnate

The diagram illustrates four states (a, b, c, d) of three resources (A, B, C) and a free list.

	Has			Max		
	A	B	C	A	B	C
(a)	2	2	2	2	2	2
(b)	4	2	2	4	2	2
(c)	4	4	2	4	4	2
(d)	4	-	-	4	-	-

Free list: 2, 2, 2, 4, 2, 2, 4, 4, -

- Prevenzione alla condizione di no preemption: non sempre possibile
- Prevenzione alla condizione di attesa circolare: numerare le risorse e richiederle solo in ordine crescente o decrescente