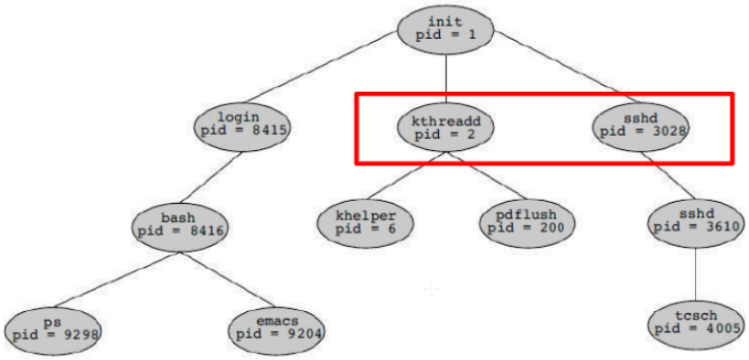
19 Maggio 2023

ARCHITETTURE DI CALCOLO LEZIONE 17

Gestione dei sistemi operativi

Tutti i processi sono generati da un processo padre, ad eccezione di “init”, il processo progenitore creato all’avvio del sistema operativo.
In un SO Unix, Init può “dare vita” a molti altri processi, tra cui:



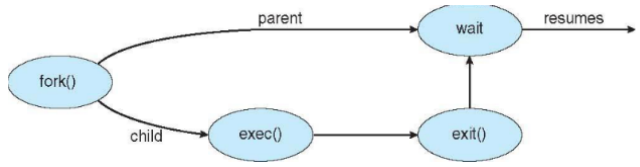
•Kthreadd (la “d” finale sta per “demon”)
•Sshd (la “d” finale sta per “demon”): è un processo server che permette ad una macchina remota di collegarsi su una macchina locale per agire localmente. Nel caso in cui un utente remoto si colleghi alla nostra macchina, allora sshd dovrà creare un processo figlio “sshd” che gestisca questa connessione.

- Login: quando login viene avviato, esso può generare il processo figlio “bash”, ovvero il processo della shell. Quest’ultimo, a sua volta, crea “ps”, che gestisce il listing dei processi attivi, ed “emacs”, processo di scrittura attivato nell’editing.

Questo è l’albero dei processi di unix, il primo processo si chiama INIT ed è il processo che si crea quando si avvia il sistema operativo; subito dopo l'avvio init da qui vengono creati altri processi, si crea per esempio il processo di login che è quello che permette agli utenti di autenticarsi e poi tutta una serie di altri processi. La d che vedete alla fine dei nomi dei processi sta per demon ovvero processi che non sono manipolati dall’utente ma background che hanno particolari funzioni molto importanti.
Qui, per esempio vengono mostrati i processi login, kthreadd, sshd, bash, khelper e cosi via.
Non mi soffermerò su cosa facciano questi processi, però faremo alcuni semplici esempi, il processo SSHD, un processo server che gira su una macchina e che permette a una macchina remota di collegarsi sulla macchina per agire in locale, quindi si collega con un client SSH ad SSHD e poi lavora; se un utente remoto usando l’SSH si collega mediante l’SSHD alla nostra macchina, questo processo dovrà creare un nuovo processo figlio che gestisce questa connessione.
Ma da come qui vedete il secondo processo SSHD può creare a sua volta un nuovo processo e così via.
Un altro esempio mostrato qui è ciò che succede con il processo INIT, che crea LOGIN il processo che interviene per gestire l’accesso dell’utente, il quale potrebbe a sua volta crea dei processi figli, come il processo Bash ovvero il processo della Shell, ciò che l'utente dopo l'avvio si vede comparire a schermo, la Shell quindi bash, a seguito dell’input immesso dall'utente potrebbe creare degli altri processi come il processo PS che effettua il listing dei processi utili o anche il processo emacs che si occupa della scrittura e dell’editing.

Nei SO Unix i processi sono creati tramite la system call “fork ()”, a seguito della quale si avrà un processo padre ed uno o più processi figli in esecuzione.

Il padre va in “wait”, in attesa della fine del figlio, mentre quest’ultimo va in “exec”, per eseguire il compito assegnatogli. Al termine di questo, il figlio esegue l’exit, il SO dealloca le risorse destinate precedentemente al processo e sia il padre che il figlio “muoiono”.



I processi sono soggetti ad un ciclo di vita, questi diagrammi di stato sono usati per descrivere tantissimi scenari tra cui, la descrizione della creazione dei processi, la creazione dei processi si fa in unix attraverso una funzione di sistema che si chiama fork (di fatto in gergo tecnico si dice ti fork a child).
Il processo padre si trova in questo punto ed esegue un’operazione di fork, si crea una biforcazione, e il processo genitore va in questo stato qua sopra, mentre il figlio in questo stato qua sotto; in altre parole in un primo momento esiste soltanto il padre, il padre invocando la fork e crea un figlio, da questo momento esistono due programmi in esecuzione, il programma figlio che fa una cosa ed il programma padre genitore che ne fa un'altra, di solito quello che succede è che il programma figlio subito dopo la creazione esegue un comando exec che permette di eseguire un programma specifico, per esempio il figlio potrebbe eseguire il listing dei file in una directory o qualsiasi altra cosa, quando l'esecuzione del figlio termina si va nello stato di exit, exit è il comando che programma usa per comunicare al sistema operativo che ha terminato.
Quando il sistema operativo riceve una exit dealloca la memoria del processo, quindi libera tutte le risorse e il processo torna ad essere un semplice file sul disco, di solito quello che succede che il processo genitore una volta che fa partire il figlio va in uno stato di attesa e rimane li senza fare nulla finché il figlio non completa l'operazione.
“Perché non lo può fare direttamente il padre dato che va in stato di attesa?”
Ci sono diversi motivi, in primis qui ne crea uno di processo ma ne potrebbe creare tanti; quindi se ne crea uno giustamente dice non serve a niente e hai ragione, ma ne dovesse creare 5 o 6 è chiaro che non potrebbe fare tutto il padre, dopodiché in questo esempio il genitore fa una wait ovvero una chiamata di attesa bloccante, tuttavia esistono casi in cui non si fa la wait ma il processo prosegue in modalità background e quindi lavorano in parallelo.

Commento del codice

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Si tratta di codice in linguaggio “C” che gestisce il ciclo di vita di un processo figlio. Innanzitutto, si importano le librerie con il comando “include”. In seguito, si definisce la variabile “pid” come pid-type (è un po’ come inizializzarla ad intero).
Si avrà, così, che il figlio avrà pid = 0 mentre il padre riceverà un pid > 0.
Per imporre ai due processi due comportamenti diversi, si userà un “if”, per cui:

- Se pid < 0, apparirà un messaggio di errore
- Se pid = 0, il figlio eseguirà la funzione “execlp” sul programma “bin/ls” (ovvero si andrà a visualizzare il contenuto di una direct)

- Se pid > 0, il padre eseguirà “wait (NULL)”, ovvero andrà in un’attesa incondizionata fintanto che il processo figlio non termina

Quando il figlio avrà completato il suo compito, sia il padre che il figlio daranno “return 0”, che significa che i processi sono terminati con successo.

Il funzionamento di questo schema è mostrato in questo pezzo di codice ed è scritto nel linguaggio C, è costituito da una parte superiore dove vedete queste #include in cui stiamo importando le librerie che vogliamo usare, quindi qui stiamo attraverso questi comandi aggiungendo a questo programma le funzioni che sono implementate da queste librerie di sistema e per fare standard input output, quindi stampare il risultato a video, il programma in questo caso è costituito solo dal main ovvero non stiamo creando funzioni, definiamo una variabile pid di tipo pid Type (una sorta di intero) cioè una variabile che useremo per identificare il processo.
La prima cosa che facciamo nel programma invocare questa funzione fork, che restituisce il valore pid (process id e non la variabile pid).
Cosa succede quando io qui invoco fork e mi restituisce PID?
Il padre invocando la fork ottiene qui un numero e contemporaneamente la creazione di un processo operativo, questo numero che io ottengo sarà l'identificatore del processo figlio qui che viene deciso dal sistema operativo.
Vi faccio vedere un esempio se io torno qui dietro (slide dei processi) se io sono in processo SSHD padre e faccio la fork di un processo figlio, io che sono 3028 e ricevo dal figlio 3610, ma chi lo decide questo 3610?
Lo decide il sistema operativo! non SSHD, assegna un id tra quelli liberi, non assegna mai due id allo stesso processo perché altrimenti non ci sarebbe modo di distinguere diversi processi.
Quindi quando il processo padre ottiene questo pid sarà in realtà il codice identificatore del figlio.
Cosa succede contemporaneamente ?
Quando si crea il figlio, esso sarà inizialmente un clone del padre, cioè il figlio compare nella memoria ed esegue esattamente lo stesso codice che esegue il processo padre, ma lui inizia a eseguire questo codice da questo punto in poi (dalla fork), quindi padre e figlio si ritrovano entrambi qui prima dell'if, ma noi vogliamo che il padre il figlio facciano cose diverse, vogliamo che il padre si metta in attesa della terminazione del figlio e che il figlio esegua un programma.
Come facciamo a imporre sullo stesso codice due comportamenti differenti ?
Banalmente lo facciamo attraverso gli if.
Nel primo if qui c'è scritto: se pid è minore di zero, vuol dire che c'è stato un errore, questo perché quando faccio una fork e creo un processo, se questo ha come valore di pid è -1 o comunque un numero negativo è un indice di errore perché indica che non è stato possibile creare il figlio, per qualche motivo e il sistema operativo manda questo numero negativo; se quindi pid è minore di zero stampo su video tramite printf ovvero una funzione di <stdio.h> (stderr —> stampa su standard erro, lo Stream di dove si mandano gli errori il messaggio) “fork fail” cioè è fallita la fork ed infine restituisco uno (risultato che di prassi indica errore) per dire completa il programma.
Di solito cosa succede o che pid è uguale a zero oppure pid è diverso da zero ma maggiore di 0.
Come si fa a capire se sono il padre o se sono il figlio?
Il padre avrà come valore della variabile pid l’indirizzo di processo del figlio mentre il figlio avrà come valore di pid zero.
Il padre capisce di essere padre perché pid ha un valore diverso da zero, in quanto rappresenta l'indice di processo del figlio (pid assegnato dall’S.O.), il figlio capisce di essere il figlio perché la sua variabile pid vale zero.
Il processo figlio eseguirà questo blocco di istruzione perché si ritroverà nel caso in cui figlio pid è uguale a zero, devo eseguire qualcosa invocando la funzione execLP definita dentro la libreria di c <sys/types> e qui dentro come parametro dobbiamo passare il programma ad eseguire.
Il programma da seguire in questo caso il /bin/ls che sarebbe il programma di sistema che fa il listing, cioè visualizza il contenuto di una directory.
Se invece non è uguale a zero e non è neanche minori di zero, significa che si tratta del processo padre, quindi il processo entra qui nell’ultimo if e deve attendere la terminazione del figlio, questo lo fa attraverso la system call wait(NULL) (null sta per tempo incondizionato altrimenti si potrebbe passare il tempo per il quale il processo deve rimanere in attesa), rimane bloccato qui fintanto che il processo figlio non termina; terminato il processo figlio, il processo padre stamperà a video (sullo schermo) tramite la funzione printf della libreria <unistd> il “Child complete”.
Sia il padre che il figlio alla fine fanno return(0) (risultato che di prassi indica successo di esecuzione) che vuol dire terminazione con successo.
“nel caso in cui ci sono dei fork multipli quindi ci sono più processi figli il pid non è crescente?”
Dovrebbe.
“quindi in questo caso con pid == zero entra solamente il primo processo figlio?”
Ogni processo figlio eseguirà una copia di questo programma, quindi ogni processo figlio vedrà il proprio pid (variabile tipo pid type) pari a zero, quando io definisco questa variabile praticamente io sto dando valore a zero, solamente il padre se questa istruzione (pid = fork();) quindi solamente il padre modifica questo valore, tutti gli altri no, questo perché gli altri processi iniziano ad eseguire da qui (primo if), quindi siccome il figlio inizia ad eseguire da qui non ha variato la variabile pid, se tu creassi 1000 figli tutti i figli avranno questa variabile pari a zero, quindi tutti i figli eseguiranno questo (istruzione secondo if).
In caso volessi eseguire istruzioni diverse in ogni processo figlio ?
Se vuoi far fare cose diverse a diversi figli devi distinguerli, diciamo quello che potresti fare è anche fare delle fork e successivamente fare una wait solo alla fine, quindi fai fork e fai partire un processo che fa una cosa, un'altra fork e fai partire un altro processo che fa un'altra cosa; che poi è quello che fa in init, ovvero tutta una serie di fork in sequenza in funzione di un file di configurazione.

É importante ricordare un paio di cose molto semplici, questo è un codice condiviso dal padre e dai figli, ma inizialmente viene seguito dal padre, quando i figli vanno ad esistere (in inglese come to exist) iniziano ad esistere da questo punto in poi (dopo la fork), quindi un figlio viene riconosciuto attraverso la variabile nel pid che nel suo caso vale 0 e quindi va ad eseguire questo (condizione pid == 0), il padre verra invece riconosciuto perché ha un valore diverso ma maggiore di 0.
Dopo l'esecuzione della fork vi sono due processi che eseguono identiche copie del programma ma mentre il valore della variabile pid è differente, attenzione non vuol dire che il pid del processo figlio è zero ma il processo figlio ha la variabile pid==0, per cui entreranno in parti differenti dello stesso codice.
Questa funzione exec (in questa versione execlp) cosa fa ?
Dice al figlio che cosa eseguire, quindi che cosa fare, noi possiamo dire al figlio di fare qualsiasi cosa purché sia una cosa di cui il figlio può occuparsi, esempio siccome il figlio eredita dei permessi del padre noi non possiamo chiedere al figlio di lavorare su risorse di cui il padre non è proprietario, quindi questo ls (listing) verrà fatto ovviamente nella home directory del padre, o volendo passare dei parametri sempre nelle directory di cui il padre è proprietario o ha visibilità.
Questo è un fondamentale tecnico di sicurezza cioè non può un padre bypassare i limiti che lui ha delegando al figlio diritti che non può avere.

Importante da ricordare

- Il codice per il processo padre e figlio è lo stesso ma si possono far eseguire loro azioni diverse utilizzando degli “if”
- Il pid dei processi figli è sempre 0 perché eseguono il codice solo a partire dal *

La terminazione dei processi può avvenire in due modi:

- Tramite exit
  - Le risorse del processo sono deallocate dal sistema operativo
  - Il processo figlio ha eseguito l’ultima istruzione
- Tramite abort
  - Il processo figlio non è più utile
  - Il processo ha usato risorse in eccesso
  - Il processo padre termina (alcuni SO non permettono l’esistenza di processi orfani, quindi, alla terminazione del processo padre, si verifica la terminazione a cascata di tutti i suoi processi figli)

Un’ulteriore tipologia di terminazione si verifica nel caso dei processi zombie. Se il processo figlio dovesse terminare prima del padre, allora quest’ultimo non avrebbe un processo su cui andare a fare la wait e non potrebbe usufruire delle risorse prodotte dal figlio; per questo i SO mantengono in vita i figli, che prendono il nome di “processi zombie”, anche quando non sono più utili.
Il caso opposto, ovvero quando il processo padre termina prima del figlio, genera processi orfani, che possono essere “killati” dal SO oppure adottati dal processo “init”.

La terminazione dei processi può avvenire in due modi: • il processo fa una exit che è quello che abbiamo visto prima, quindi il processo dichiara spontaneamente di voler terminare la propria esistenza, ciò causa deallocazione delle risorse; • il processo può essere come si dice in gergo abortito, ovvero viene eliminato dal genitore, ciò avviene quando un processo non dovesse terminare spontaneamente per esempio perché va errore ed è responsabilità del processo genitore ucciderlo, può capitare ad esempio che il processo padre crea un processo figlio che sta eccedendo nell'uso delle risorse condivise col padre allora in questo caso il processo padre potrebbe decidere di terminarlo.
Un'altro aspetto da considerare è che in teoria se un processo padre non fa la wait, il processo padre potrebbe arrivare a terminare prima che il processo figlio termini, possibile soprattutto se il processo padre non fa nulla mentre il processo figlio deve fare cose molto lunghe.
Per questo alcuni sistemi operativi impediscono ad un processo figlio di continuare a vivere se il processo padre è morto, quindi vuol dire che appena il processo padre termina il processo figlio deve essere chiuso, questo determina una chiusura a cascata dei processi, cioè in sostanza può capitare di chiudere un programma che da solo determina la chiusura di tanti altri programmi, questo capita quando gli altri programmi erano figli o discendenti e killati uccisi a cascata, quindi alcuni sistemi non permettono l'esistenza di processi orfani.
In realtà ci sono tanti casi speciali, ne discutiamo solo alcuni.

Un'altro caso che può capitare è questo supponiamo che un padre crei un figlio e che il padre si metta in wait sul figlio, tuttavia il padre non si deve mettere in wait sul figlio subito ma potrebbe capitare che il padre debba fare delle cose e dopo che ho fatto quelle cose si mette in wait sul figlio ma può succedere però che il figlio nel frattempo ha già finito.
Quindi dall'istante 0 all'istante 5 il figlio finisce il suo lavoro, il processo padre dall'istante zero all'istante 5 ha fatto altre cose e dopo debba altre cose dopodiché si mette in wait sul figlio che in realtà ha già finito.
Cosa succede?
Il sistema operativo in questi casi prende il processo figlio e anziché di deallocarlo dal sistema subito lo dealloca solo quando arriva il momento della wait del padre, quindi in questo caso si dice che il processo figlio che è terminato ma su cui il padre va in attesa è un processo zombie, cioè è già morto, cioè il processo figlio ha già finito però non viene deallocato siccome il padre deve usare le risorse che il figlio produce prima di cancellarlo, per cui il padre deve fare la wait.
Il senso qual'è ?
Se io deallocassi il figlio che ha fatto un lavoro utile al padre, tutte le cose che ha fatto il figlio spariscono, allora io lo tengo in vita al solo scopo di passare al padre i risultati.
Se un processo padre termina senza fare la wait potrebbe capitare che i figli stanno ancora girando, se i figli stanno ancora girando, quando il padre ha terminato, il sistema operativo potrebbe in teoria o "killarli" oppure assegnare questi figli ad INIT che è il processo genitore sovrastante.

Commento del codice

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

        /* alloca la memoria */
        ZeroMemory(&si, sizeof(si));
        si.cb = sizeof(si);
        ZeroMemory(&pi, sizeof(pi));

/* genera processo figlio */
if (!CreateProcess(NULL, /* usa riga di comando */
  "C:\\WINDOWS\\system32\\mspaint.exe", /* riga di comando */
  NULL, /* non eredita l'handle del processo */
  NULL, /* non eredita l'handle del thread */
  FALSE, /* disattiva l'ereditarieta' degli handle */
  0, /*nessun flag di creazione */
  NULL, /* usa il blocco ambiente del genitore */
  NULL, /* usa la directory esistente del genitore */
  &si,
  &pi))
{
   fprintf(stderr, "generazione del nuovo processo fallita");
   return -1
}
/* il genitore attende il completamento del figlio */
WaitForSingleObject(pi.hProcess, INFINITE);
printf("il processo figlio ha terminato");

/* rilascia gli handle */
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
}
```

Il codice a sinistra è la versione per sistemi operativi Windows dell'esempio precedente. Le differenze sono:
•Uso della libreria "window.h" al posto di "unistd.h"
•Chiamata di sistema "create process" al posto di "fork"
•"Create process" necessita di ricevere parametri, al contrario di "fork"
•Il metodo wait si chiama "waitForSingleObject", il quale pone il padre in uno stato d'attesa fino al termine dell'esecuzione di un solo processo figlio.

Questi casi esistono in Linux come esistono in Windows, quello che cambia è un po la sintassi.
Quello che vedete qui in realtà è esattamente lo stesso esempio che abbiamo visto prima di Linux ma scritto con la sintassi di Windows, non dovete assolutamente imparare quello che c'è qui, commetterò solamente qualche elemento, vi dovrete ricordare che il metodo fork dentro di un Windows si chiama createprocess.
Quindi in un ipotetico programma in linguaggio C che gira sotto Windows, quali sono le differenze fondamentali?
Il nome delle funzioni, le chiamate di sistema cambiano un pò, la libreria dove sono presenti tutte le System call windows che in questo caso si chiama windows.h.
Il metodo fork che è dentro Linux non riceve parametri mentre il metodo CreateProcess riceve una decina di parametri; il metodo wait si chiama WaitForSingleObject, questo perché in realtà ci sono anche delle wait per più figli, cioè in teoria posso fare quella che si chiama barriera di sincronizzazione su n figli, che è un'altra funzione in cui si effettua la wait su tanti figli cioè attendi fintanto che tutti i figli completano il rispettivo lavoro.
Vediamo un esempio ipotetico ma reale di uso di questo meccanismo della fork, supponete di avere aperto la Shell di unix, la shell permette di eseguire tanti comandi, cosa succede ogni volta che nella Shell digitate un comando?

La Shell crea una copia di se stessa, delegando alla copia ovvero al figlio l'esecuzione del comando che avete effettuato, quindi quando la shell riceve un comando utente fa la fork di se stessa, li dentro fa la exec del comando.
Immaginate di digitare nella shell "ls -l prog*>proginfo", questo comando fa la lista di tutto il contenuto della directory project che cominciano con prog e metti il risultato in un file che si chiama proginfo; quando eseguite questo comando nella shell, la shell in realtà fa un fork, crea un'altra shell e dentro questo figlio fa la exec di questo comando e la shell (padre) si mette in wait; quando il figlio completa l'esecuzione del comando la Shell ritorna in primo piano col prompt e vi ritrovate nella cartellina questo file proginfo, tuttavia si potrebbe anche decidere di fare eseguire al figlio questa attività senza bloccare il padre, cioè si può anche evitare che la Shell (padre) si blocchi fino a completamento dell'operazione figlio, è sufficiente aggiungere una "&" alla fine della linea di comando, che attiva esecuzione in background, cioè state dicendo perché al padre: "crea una nuova Shell che esegue questo comando ma lancialo in background, cioè il figlio deve fare le sue attività senza bloccare il padre che deve essere pronto a ricevere un'altro comando".

Un'applicazione pratica della creazione dei processi figli può essere osservata quando l'utente digita un comando nella shell. In questo caso, la shell genera una copia di se stessa, delegando a quest'ultima l'esecuzione del comando (exec); se al codice viene aggiunta una "&", allora si impedisce il blocco della shell fino alla terminazione del processo "shell-copia".

Quando è utile secondo voi questo ?
Quand'è che conviene mettere "&" ?
Quando bisogna fare un altro processo in contemporaneo, quando abbiamo bisogno dei risultati di più processi simultaneamente oppure quando un processo è molto lungo che altrimenti bloccherebbe l'inserimento di altri comandi finche non termina l'esecuzione dei primi, immaginate che voi fate un comando del tipo zippare una cartella di 10 giga, processo che può richiedere minuti o ore, quindi lo mando in background per lasciare il terminale pronto ad altri comandi dell'utente.
Quindi conviene mandare in background i processi quando sono di lunga durata o voglio contemporaneamente eseguirne altri, se voglio eseguire tanti processi tutti insieme, potrei fare uno script in cui lancio tanti processi uno dopo l'altro mettendo sempre la & in modo che partono tutti in parallelo.
Un altro caso, in cui potrebbe essere utile, è quando collegandoci sulla Shell di un server tramite chiave SSH lanciamo il programma e poi ci scolleghiamo dal server, se non apponiamo la "&"appena ci scolleghiamo dal server, il processo SSH che ci rappresentava termina ma siccome aveva creato anche il comando lanciato anche il comando stesso viene terminato, questo perché il figlio non può sopravvivere alla terminazione del padre, se invece apponiamo la "&" lo si lancia in parallelo, non essendoci la wait il comando continuare a girare anche disconnettendoci.
Come viene gestito un processo pesante che occupa le risorse ?
Il meccanismo delle quote è legato alla user non è legato al processo, quindi sostanzialmente esiste la possibilità in alcuni sistemi operativi di mettere dei limiti nell'uso delle risorse, per esempio possiamo mettere delle quote di spazio utilizzato, se ovviamente io ho uno spazio e lo esaurisco, qualsiasi processo viene killato, il padre riceve un sigkill per cui anche i figli per via della kill del processo padre, è il sistema operativo che "se superi le quote uccide tutti".

In sintesi, si possono verificare due comportamenti:
- Il padre si pone in attesa della terminazione del figlio (esecuzione in foreground)
- Il padre procede nell'esecuzione concorrentemente al figlio (esecuzione in background; va aggiunta la "&")

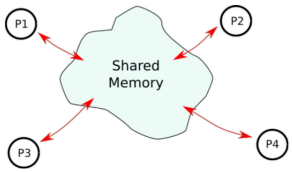
Architettura multi-processo

Google Chrome è un browser multi-processo, al contrario della maggior parte dei browser che sono della tipologia multi-thread.
Ogni qualvolta che l'utente apre una scheda in Chrome, il browser crea un nuovo renderer che permette di visualizzare graficamente il codice html della nuova pagina.
I renderer sono eseguiti indipendentemente dagli altri renderer, secondo la metodologia "sand- boxing", in modo che se uno dei processi dovesse fallire, questo non porti al fallimento anche di tutti gli altri.

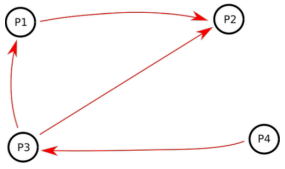
Un esempio di sistema che sfrutta questo approccio che abbiamo appena descritto in particolare “la creazione di più processi” per un'applicazione che poi usiamo tutti abbastanza spesso, Google Chrome.
Chrome è un browser che adotta un'architettura “multiprocesso”.
Quasi tutti i browser in circolazione sono “multithread” mentre Chrome fa eccezione essendo “multiprocessing”; ciò vuol dire che quando noi apriamo il browser di Chrome ti permette di aprire più schede contemporaneamente per navigare.
Ogni volta che viene aperto Chrome crea un nuovo processo, in particolare il processo che gestisce la visualizzazione di un sito web nella scheda prende il nome di Renderer cioè il processo di rendering, cioè la visualizzazione dell’HTML.
Come sapete il sito web è descritto da un file che si chiama HTML da cui mi fai l'associate-link che sono i fogli di stile, nativamente questi file sono del codice poi il browser visualizza questo codice e lo può apparire come insieme di immagini di suoni, testi ecc…, quindi il renderer è il processo che dà tutto il codice HTML visualizza graficamente il contenuto.
Il concetto qual è ?
Dentro Chrome c'è questo processo generale che il browser che per ogni pagina che apriamo crea un renderer per visualizzare la pagina
Qual è la caratteristica di questi render ?
Sono eseguiti gli uni indipendenti dagli altri, si usa un approccio che in gergo si chiama sandboxing, sandbox (consente l’accesso a disco e rete limitati per minimizzare gli effetti negativi di exploit di sicurezza) significa “eseguire un processo in una scatola separata dalle altre” in modo che se quel processo fallisce, non causa il fallimento degli altri processi, ma solamente di se stesso.
Il fallimento del browser che è il padre di tutti i processi causa il fallimento di tutti i renderer, ma il fallimento di un renderer che un figlio non causa il fallimento degli altri renderer.
Perché dovrebbero fallire dei renderer ?
Non tanto negli ultimi anni in cui la qualità del codice dei siti web è molto migliorata, ma in epoca più antica, il codice dei siti web era pieno di errori e i browser quando fallivano l'interpretazione spesso crashavano.
Oggi può succedere soprattutto in presenza dei siti che usano dei plug-in esterni già scritti con errori.
cosa hanno deciso intelligentemente gli ingegneri di Chrome ?
Usiamo un approccio in cui se mi fallisce un sito non mi crasha il browser, quindi con questo approccio multiprocesso semplicemente loro sono stati in grado di risolvere il problema.
I browser multithreaded funzionano diversamente, di fatto se fallisce unthread fallisce tutto.

Comunicazione tra processi

Quando i processi possono subire l’influenza di altri processi o possono influenzarne altri, essi sono detti processi cooperati, altrimenti si chiamano processi indipendenti (la maggior parte dei processi sono indipendenti per evitare problemi di crash collettivo).



Shared memory: i processi condividono dati in me- moria e accedono in lettura e scrittura a tali dati condivisi



Message passing: i processi si scambiano informazioni tramite messaggi (simile a quanto avviene sulla rete)

Vantaggi della cooperazione fra processi:
•Condivisione di informazioni: ambienti con accesso concorrente a risorse condivise
•Accelerazione del calcolo: possibilità di elaborazione parallela (soprattutto nelle macchine multi-core, in presenza di dati molto grandi da elaborare, si procede con l’approccio “divide et impera”, ovvero si divide il problema in sotto-problemi che saranno elaborati in parallelo, risparmiando tempo)
- Modularità: funzioni distinte che accedono a dati condivisi (es. kernel modulari)
- Convenienza: per il singolo utente, possibilità di compiere più attività in parallelo

I processi cooperanti possono interagire tramite:
- Scambio esplicito di dati

- Sincronizzazione su un particolare evento
- Condivisione dell’informazione

Qualche lezione fa, io ho detto che i processi possono comunicare fra loro e ci sono diversi modi per far si che ciò avvenga. Quando i processi comunicano, sono detti processi cooperanti, quando non comunicano sono detti processi indipendenti (se l’esecuzione dell’uno non influenza l’esecuzione dell’altro).
La maggior parte dei processi del sistema operativo sono indipendenti, però esistono tanti processi che sono cooperanti, per esempio nel caso di Google Chrome i render sono processi indipendenti tra loro, ma coperanti col browser, quindi c'è un momento in cui il browser coopera con gli oggetti che si occupano della visualizzazione.
Dunque un processo può essere indipendente rispetto ad altri ma cooperante rispetto ad altri ancora, naturalmente quando andiamo a ricostruire le relazioni tra processi io devo tenere conto del fatto che l'indipendenza la cooperazione sono concetti relativi ovviamente agli insiemi che consideriamo.
La cooperazione fra processi nei sistemi operativi moderni è uno strumento fondamentale per tutta una serie di vantaggi la prima motivazione che in particolare è quella di poter condividere informazioni tra processi, quindi processi che devono condividere informazioni tipo file, risultati intermedi e così via devono ovviamente cooperare per fare tutto ciò.
Un'altra motivazione per la cooperazione e l'accelerazione del calcolo, nelle macchine soprattutto quelle multicoore si prende il problema, si divide in sottoproblemi, e poi ogni il sotto problema lo diamo in pasto ad un processo che lavora in parallelo agli altri usando un processore diverso.
Ci sono tanti casi, soprattutto nel dominio dell'analisi dei dati in cui noi abbiamo da analizzare dati molto grandi e abbiamo due possibilità o abbiamo un unico processo che analizza tutto “in modo sequenziale”, quindi usando un unico processore oppure dividiamo questo problema, questo dato da analizzare, in sotto-problemi e assegniamo una parte dell'analisi a ad un processo diverso, questi processi girano il parallelo usando processi diversi e quindi se tutto va bene si dovrebbe risparmiare tempo; quindi ne deriva l'accelerazione del calcolo basato sul principio della suddivisione del lavoro.
Un altro motivo per cui dividiamo i programmi in più processi è la modularità: se avete un programma complesso, basta pensare a Microsoft Word il quale è un programma molto complesso, è dotato di un modulo che si occupa della visualizzazione, un'altro per il salvataggio dei file, moduli per fare la ricerca dei contenuti e tante diverse funzioni.
È ragionevole pensare che in programmi così complessi, che queste diverse funzioni siano implementate da processi diversi, che lavorano insieme, cooperano per uno scopo complessivo.
E poi c'è una motivazione di convenienza, anche in quei computer in cui c'è un solo utente che usa la macchina avere a disposizione più processi che fanno più cose insieme e che cooperano per gli scopi dell'applicazione ovviamente è sicuramente molto utile.
Far cooperare i processi è utile per tutti i motivi che abbiamo detto, ma c'è un problema di comunicazione “concorrente”, cioè l'interazione tra processi abbiamo visto si possono fare con memoria condivisi e scambi subito dei dati, ma quando noi condividiamo informazioni fra tutti i processi ci possono essere problemi di sincronizzazione.

Per poter attuare la comunicazione tra processi, si deve gestire il problema della sincronizzazione. Infatti, due processi potrebbero scrivere contemporaneamente sullo stesso file, generando un errore; per questo motivo l’operazione di scrittura è detta di mutua esclusione, cioè un solo processo può scrivere su un file in un dato momento. La lettura contemporanea di uno stesso file da parte di più processi è, invece, consentita. Per ragioni analoghe anche la funzione di lettura e scrittura non può avvenire contemporaneamente tra due processi (ciò che verrebbe letto non sarebbe più attuale al proseguire del processo di scrittura). Tutte queste operazioni di sincronizzazione sfruttano delle primitive del SO o dei linguaggi di programmazione (come Java o Python) che permettono di far comunicare opportunamente i processi.
I sistemi operativi offrono meccanismi per realizzare queste diverse forme di cooperazione, ad esempio:

- Send e receive
- Semafori
- Monitor
- Chiamata di procedure remota

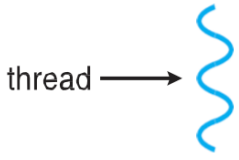
Si distinguono tutti i mezzi di cooperazione e comunicazione tra processi in due grande famiglie, che sono quelli a memoria condivisa e quelli basati su scambio di messaggi.
Tipicamente, nelle macchine multicore si predilige l'approccio shared-memory, mentre nei sistemi distribuiti (es. cluster di computer costituti da tanti nodi) si usa il meccanismo di messagge-passing.

- Lo scambio di messaggi costituisce il modello di riferimento per la comunicazione tra processi.
- La memoria condivisa è, invece, il modello di riferimento per la comunicazione tra thread; i thread, infatti, condividono tutte le risorse del processo a cui appartengono, tra cui la memoria.

Cosa succede se due processi comunicano attraverso un file e contemporaneamente entrambi i processi cercano di scrivere su quel file per condividere le informazioni?
Tutto va bene fin tanto che le letture e le scritture si alternano.
Che succede se contemporaneamente due processi provano a scrivere?
La sovra-scrizione non è possibile, poiché la scrittura di un file avviene in "mutua esclusione" cioè o scrive uno o scrive l'altro.
Naturalmente questa scrittura a mutua esclusione, serve ad evitare che due processi sovrascrivono i contenuti su un file vicendevolmente, questa è una funzionalità gestita dal sistema operativo con delle procedure: semafori,monitor…
Questi meccanismi, servono proprio ad evitare i problemi di modifica concorrente delle informazioni.
I sistemi operativi fanno un continuo uso di questi strumenti quasi tutte le comunicazioni tra processi avvengono attraverso scambi di informazioni sul file, quindi questi file devono essere in qualche modo letti e scritti in modo opportuno.
Diciamo che: se un file e due o più processi leggono contemporaneamente il file non è un problema, leggere un file non modifica quindi anche 1000 processi possono leggere; però la scrittura da parte di due processi o più processi non è mai consentita così come non è consentito scrivere e leggere nello stesso momento, questo non perché si creino problemi nel contenuto ma perché c'è il rischio che la lettura vada a leggere un contenuto che ancora non è finalizzato, se io vado a leggere un file prima che il file viene chiuso in scrittura, come dice in gergo, quello che leggo in realtà è un'immagine del passato quindi chi legge il contenuto potrebbe erroneamente pensare che il contenuto che legge è corretto mentre non è più corretto.
Tutte queste necessità di sincronizzazione le possiamo ottenere in due modi: usando le funzioni primitive del sistema operativo oppure quelle fornite da alcuni linguaggi di programmazione tra cui Java e anche Python.
In letteratura si distinguono tutti i diversi mezzi di cooperazione e comunicazione tra processi in due grandi famiglie: • quelli a memoria condivisa e • quelli basati su scambi di messaggi.
Nelle macchine multicore tipicamente si usa l'approccio shared memory, nei sistemi distribuiti quindi cluster di computer costituiti da tanti nodi si usa tipicamente il meccanismo del message passing, questo perché se io ho tante macchine diverse non ho una memoria unica quindi non posso sfruttare la memoria condivisa per far comunicare i processi, perciò l'unico modo che ho in un sistema distribuito per far comunicare i processi e mandare dei messaggi da un nodo all'altro usando una rete ad alta velocità.
In teoria il messaggio passing si può usare anche su singole macchine che hanno una memoria condivisa, ma risulterà meno efficienti.
Quindi se io ho una macchina con una memoria condivisa tra più processi è ragionevole usare il meccanismo della Shell memory, se invece ho un sistema distribuito sono costretto a usare necessariamente il messagge passing, con l'eccezione di alcuni software che permettono di visualizzare la memoria condivisa anche su macchine che non hanno una memoria condivisa, ma chiaramente è una virtualizzazione, fanno vedere una memoria condivisa che in realtà non c'è.
Anche i thread utilizzano la Shared memory?
Si esatto, i thread sono come dei processi che girano all'interno di un processo più grande e la caratteristica dei thread è proprio che condividono la memoria fra loro.

Thread

Un thread (dall'inglese filo o trama) è l'unità base d'uso della CPU e comprende un identificatore di thread (TID), un program counter, un insieme di registri ed uno stack.



I thread possono essere visti come processi più leggeri, cioè processi che non hanno in realtà una propria autonomia ma che esistono soltanto all'interno di altri processi. La caratteristica dei thread è la condivisione della memoria fra di loro e con il processo a cui appartengono. Si possono immaginare i thread come un flusso di esecuzione all'interno di un flusso di esecuzione più grande.

- La maggior parte delle applicazioni attuali sono multi-thread.
- I thread sono stati ideati per risolvere il problema di context switch connesso all'esecuzione di programmi in parallelo; tramite il multi-thread si risparmia tempo perché non bisogna ricaricare i dati dalla memoria.
- I thread possono, inoltre, comunicare tra loro tramite semplice scambio di variabili grazie alla modalità di comunicazione shared memory.
- I thread vengono eseguiti all'interno delle applicazioni.
- La gestione dei processi può diventare molto onerosa, sia dal punto di vista computazionale che per l'utilizzo di risorse. In particolare, nella:
    - Creazione —> allocazione nello spazio degli indirizzi e successiva popolazione.
    - Context-switch —> salvataggio e ripristino degli spazi di indirizzamento (codice, dati, stack) di due processi.

I thread sono come dei processi però più leggeri, cioè processi che non hanno in realtà una propria autonomia ma che esistono soltanto all'interno di altri processi.
Possiamo vedere il thread come un flusso di esecuzione all'interno di un flusso di esecuzione più grande.
Se io creo un processo di norma il processo che creo è single thread, cioè ha un singolo thread, ovvero un singolo flusso di esecuzione; tutti i programmini che avete visto prima sono single thread, cioè io eseguo solo un flusso di esecuzione.
Col multithreading è suggerito creare programmi che hanno più flussi di esecuzioni, cioè che in parallelo fanno più cose, ciascuno dei flussi di esecuzione prende il nome di thread.
Prima di vedere concretamente qualcosa in più sui trader vi dico i motivi per cui sono stati inventati trader, perché uno potrebbe dire: perché fare un programma che c'ha più flussi di esecuzione quando potrei fare direttamente più programmi che vanno in parallelo?
Abbiamo già capito che se noi facciamo girare in parallelo più programmi, c'è un problema di context switch, cioè ogni volta che un processo deve far partire un'altro processo al suo posto ci deve essere una fase in cui i dati di quel processo si salvano nel PCB, si prendono i dati del PCB del processo nuovo e si rimettono in memoria, e si fa tutta questa operazione ogni volta che ci un cambio di contesto.
Abbiamo visto che questo cambio di contesto, context switch, è un costo che noi dobbiamo considerare perché la CPU mentre fa cambio di contesto non esegue nulla dei nostri programmi.
Per evitare il contest switch legato all'alternanza dei processi di un sistema operativo si può scegliere di avere processi a loro volta internamente divisi in thread, in modo che si passa da un thread all'altro internamente senza che proprio questo determini un cambio di processo.
Il passaggio dell'esecuzione da un thred ad un'altro tread è molto più rapido di quanto non sia il passaggio da un processo a un'altro processo, questo perché i thread di uno stesso processo condividono la memoria del processo principale, questo vuol dire che quando io passo da un thred all'altro la memoria è già lì, è condivisa fra tutti thread per cui non c'è bisogno di cambiare contesto, sicuramente c'è bisogno di cambiare il Program Counter, perché ogni thread avrà il suo punto di esecuzione che è diverso da quello degli altri, ma il fatto che la memoria sia condivisa ci evita l'overhead (richiesta di risorse eccessiva) di dover caricare in memoria tutto quanto visto che già è tutto in memoria.
Un'altro elemento da considerare è che il fatto che la memoria sia condivisa è anche uno strumento per far parlare fra loro i tread di uno stesso processo in modo rapido, possono comunicare tra loro banalmente scrivendo i valori in una variabile, perché quella variabile sarà visibile da tutti thread dello stesso processo, quindi è un approccio shared memory.
Questa slide dice se io ho un'applicazione che genera molti processi, per esempio un server web ogni volta che ricevo una richiesta dovrebbe creare un processo nuovo per seguire quella richiesta, quindi il sistema operativo dovrebbe gestire singolarmente ciascuno di questi processi: collocare risorse, eccetera eccetera.
Soprattutto in questo tipo di applicazioni che generano tantissimi figli piuttosto che creare nuovi figli, creiamo dei thread che girano all'interno del processo con una molta maggiore leggerezza rispetto ai processi classici.
Ci sono ovviamente applicazioni che beneficiano di più dei threading rispetto ad altre, per esempio nell'ambito della sensoristica ci sono applicazioni che ricevono dati da tanti sensori, ad esempio i sensori ambientali, piuttosto che avere un processo che si occupa di prendere i dati da ciascuno di questi sensori possiamo avere un unico processo complessivo che fa partire tanti thread uno per quanti sono i sensori per leggere i dati; è meglio avere tanti thread che leggono dei vari sensori piuttosto che tanti processi, l'applicazione risulterà molto più leggera.

Il multithreading è così diffuso ormai che persino il kernel del sistema operativo, cioè il cuore del sistema operativo è costituito da tecnologia multithread.
Inoltre, il multithreading si adatta molto alle architetture multicore che si usano oggi.
Multicore vuol dire ?
Il processore c'ha tanti sotto-processori che sono in grado di eseguire le operazioni in parallelo gli uni con gli altri.
Se io ho una macchina con 16 core, posso eseguire in parallelo fino a 16 thread e in questo modo posso far andare tutto più veloce, quindi se io il kernel che anziché fare una sola cosa per volta ne fa 16, il sistema operativo sarà molto più veloce; in questa maniera il threading viene gestito in modo molto efficiente.

Ci sono applicazioni che beneficiano del multithreading più di altre. Ad esempio, nell'ambito della sensoristica ci sono applicazioni che ricevono dati da tanti sensori; piuttosto che avere un processo che si occupa di prendere i dati da ciascuno di questi sensori, possiamo avere un unico processo complessivo che fa partire tanti thread. Il multithreading è così diffuso che ormai anche il kernel risulta tale, connesso a sistemi multicore; se un kernel, invece di fare una sola cosa per volta, è capace di farne 16 (16 core), l'efficienza aumenta esponenzialmente.

Quale è la differenza fondamentale tra un processo e un thread? Il thread vive soltanto all'interno di un processo, con cui condivide la memoria RAM. Inoltre, il fallimento di un singolo thread killa tutto; il fallimento di un processo figlio, invece, non dà problemi agli altri figli.

Thread, in inglese significa trama o filo, infatti il disegnino che si usa per rappresentare è una specie di filo che rappresenta il flusso d'esecuzione, quindi un filo d'esecuzione.
Il thread è rappresentato nel programma come codice da eseguire, ovviamente come ogni codice che si esegue saremo in ogni momento arrivati a un certo punto dell'esecuzione.
Come facciamo in un codice a sapere a che punto siamo arrivati ?
In un processo in esecuzione quale registro ci dice qual è la prossima istruzione da eseguire ?
Il program counter.
Come esiste il program counter dell'intero processo, esistono i program counter dei thread quindi se voi avete 10 thread dentro un processo, ognuno di questi avrà il suo program counter che ci dice dove è arrivato quel thread, questo perché ovviamente devono poter essere sospesi riavviati devo sapere dove erano arrivati.
Allo stesso modo, così come i processi hanno un ID (un pid) anche i thread hanno un ID che prende il nome di "thread id".
Ci sono chiaramente delle analogie: identificatori, program account,ecc ; ma ci sono anche delle differenze, per cui se io vi dovessi chiedere:
Qual è la differenza fondamentale fra un processo e thread?
La risposta che mi aspetto è che il thread vive soltanto all'interno di un processo e condivide la memoria RAM, quindi le variabili che usa con il processo principale e anche con quelle degli altri thread dello stesso processo; questo significa che se un thread fallisce e non è opportunamente gestito, fallisce l'intero processo, mentre se fallisce il processo e questo non è il processo padre non dovrebbe influenzare gli altri.

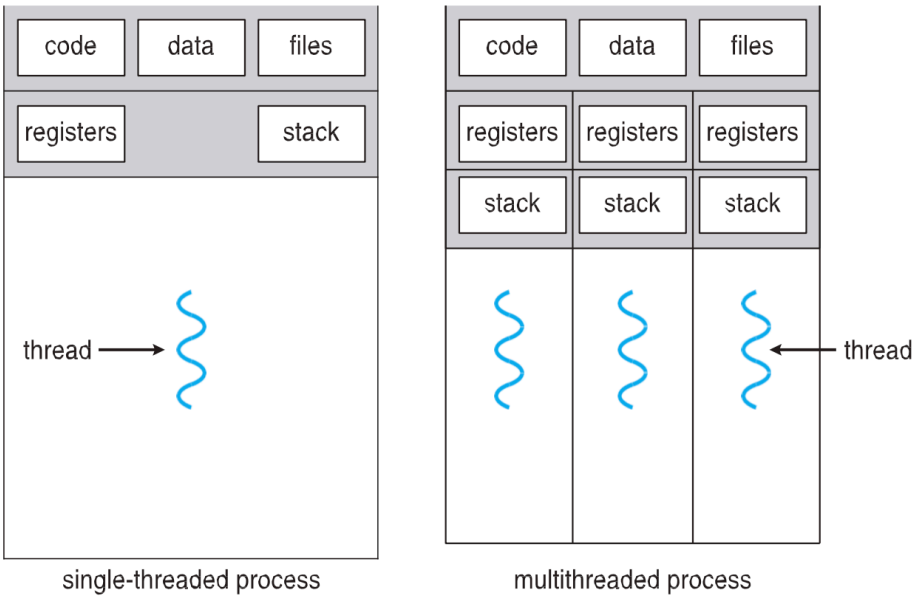
Come già detto, i thread sono detti anche processi leggeri (light-weight process), perché hanno un contesto ridotto (condividono lo spazio degli indirizzi).
Poiché i thread appartenenti ad uno stesso processo condividono codice, dati e risorse:

- Se un thread altera una variabile non locale al thread, la modifica è visibile a tutti gli altri thread.
- Se un thread apre un file, allora anche gli altri thread possono operare sul file. Un processo multithread è in grado di eseguire più compiti in modo concorrente.

Tali analogie ma anche differenze sono anche alla base di un differente nome a volte usato per il thread, che è lightweight process, cioè processo leggero.
Di fatto è un processo con un proprio program counter, dei registri associati, però è leggero poiché il cambio di contesto avviene solamente all'interno del processo stesso e la memoria è condivisa, è più leggero in quanto quando si passa da un thread all'altro dello stesso processo si devono fare meno operazioni di quanto non se ne facciano quando si passa da un processo indipendente a un'altro processo indipendente.

Questo approccio che si è diffuso negli ultimi 25 anni di realizzare programmi multithreaded è in contrapposizione con l'approccio classico, ovvero approccio heavyweight process, in cui tutto viene effettuato da un unico processo.

Heavyweight process (processo pesante): è un processo tradizionale composto da un solo thread. Un processo è pesante, in riferimento al contesto (spazio di indirizzamento, stato) che lo definisce. Questa immagine mostra la differenza tra processo classico single thread e multithread:



Quest'immagine fa vedere la differenza tra un processo classico single thread ed uno multi thread; in particolare un processo classico single thread ed è un processo che ha un unico codice da eseguire, un unico filo di istruzione ed ovviamente caratterizzato da tutto ciò che di solito ha un programma, quando andiamo in un programma multithread abbiamo, per esempio, tre codici in parallelo ciascuno di questi codici avrà un proprio stak, ricordo che lo stak è la parte di memoria in cui si mettono le variabili create. Un thread può avere le sue funzioni, quindi volta che un thread invoca una funzione alloca una area di memoria dello stack, chiaramente ha i suoi registi perché essendo il thread un codice esecuzione avrei registi in cui ci sono le variabili: c'è il program counter, le variabili usate dall' ALU ma poi tutta la parte che afferisce alla memoria condivisa cioè i file aperti, i dati su cui lavora, il codice principale, questi sono comuni fra tutti i thread; ovviamente il thread non può entrare nelle aree personali di altri thread ma solo a ciò che e condiviso tra loro.
Tutto ciò è molto utile, ma non solo per rendere più efficiente un programma ma anche per il programmatore per modularizzare il codice, voi potreste fare un progetto in cui uno si occupa di diversi thread, ad esempio uno del thread di salvataggio, uno del thread di visualizzazione, ecc, chiaramente poi si dovrà coordinare all'interno del programma principale per far comunicare questi thread.
I thread possono coesistere nello stesso processore, nello stesso core?
Si, grazie ad una tecnica denominata iperthreading: sullo stesso core puoi avere tipicamente due thread.

Esempi:

- **Browser web**
  - Un thread per la rappresentazione sullo schermo di immagini e testo.
  - Un thread per il reperimento dell'informazione in rete.

- **Text editor**
  - Un thread per ciascun documento aperto (i thread sono tutti uguali ma lavorano su dati locali diversi).

- **Relativamente ad entrambi è anche possibile avere:**
  - Un thread per la visualizzazione dei dati su schermo
  - Un thread per la lettura dei dati immessi da tastiera
  - Un thread per la correzione ortografica e grammaticale o Un thread per il salvataggio periodico su disco

Riassumendo

1. Il context switch tra thread è molto più veloce
2. Un sistema operativo composto da thread è più efficiente
3. Con le CPU multicore i programmi composti da più thread possono essere eseguiti sui diversi core in parallelo.
4. I thread di uno stesso task non sono tra loro indipendenti perché condividono codice e dati.
5. È necessario che le operazioni non generino conflitti tra i diversi thread di un task.

È importante, quando si programma un thread su un sistema operativo multithreading, non commettere errori per evitare il crash del sistema operativo e/o il danneggiamento della memoria condivisa con gli altri thread.

Riassumendo, il passaggio da un thread all'altro avviene molto velocemente, perché la memoria non va ricaricata quindi il Context-switch è molto più veloce, questo approccio anche nel sistema operativo rende più efficiente l'esecuzione, poi se sono presenti più core o più processori allora a sua volta l'esecuzione è fisicamente più veloce.
Ovviamente non ci sono solamente aspetti positivi, perché il fatto stesso che vi ho fatto vedere prima Google Chrome come browser multiprocesso, i ci segnala quali sono i limiti, il multithreading va usato con notevole attenzione alla programmazione, cioè bisogna evitare di scrivere thread che possono crashare perché causerebbe il crash di tutto il programma, un'altro aspetto è la concorrenza, visto che condividono la RAM, ciò è comodo ma è anche rischioso perché se un thread sovrascrive dei valori in RAM che servivano ad un'altro thread in modo inappropriato ovviamente i dati si perdono, servono delle tecniche per evitare di danneggiare i dati.
Uno potrebbe pensare quindi che avendo più thread la macchina va molto più veloce, quindi da una macchina con 16 thread mi posso aspettare che si scriva un programma che va 16 volte più veloce rispetto a un programma che ha un solo thread, in realtà se voi anche avete una macchina con tantissimi core o processori, l'accelerazione che riusciamo a ottenere su questa macchina non è uguale al numero di core che abbiamo, c'è un limite.

Legge di Amdahl

La legge di Amdahl permette di prevedere il miglioramento atteso massimo in una architettura di calcolatori o in un sistema informatico quando vengono migliorate solo alcune parti del sistema. Se si ha una macchina multicore, i potenziali guadagni in termini di prestazioni ottenuti dall'aggiunta di core, nel caso di applicazioni che contengano sia componenti seriali (non parallelizzabili) sia componenti paralleli, è legata all'equazione:

S: è la percentuale non parallelizzabile
N: numero di core o i processi a disposizione

C'è uno studio con un interessante risultato che ci dice che se noi abbiamo una macchina con tantissimi core, l'accelerazione che riusciamo a raggiungere non è linearmente proporzionale al numero di core che quella macchina ha, c'è appunto un limite studiato con la legge di Amdahl.
Amdahl chi era ?
Era un dipendente di una azienda intel, e lui voleva suggerire un modo per riuscire a rendere i processi sempre più veloci, questo perché ?
Perché negli 70 le aziende si affacciavano a due filoni per costruire macchine sempre più performanti: • con un unico processore veloce e performante o • più processori poco performanti.
Qual è l'approccio migliore? 1 processore che va a 100 o 2 che vanno a 50 ?
Amdahl, dimostrò che se una macchina possiede tanti processori (n), il tempo di esecuzione non diventa t/n perché dimostrò che lo speed up, cioè l'accelerazione che posso ottenere eseguendo un programma su n processori non è pari al numero dei processori ma è pari a: 1 / [S+ (1-S)/n], dove S nella parte del programma sequenziale, cioè quella parte di programma che io non posso far realizzare in parallelo.

Esempio: Se un'applicazione è al 75% parallela ed al 25% seriale, passando da 1 a 2 core, si ottiene uno speedup pari a 1.6; se i core sono 4, lo speedup è invece di 2.28. In generale, per Nè —> lo speedup tende a 1/S.

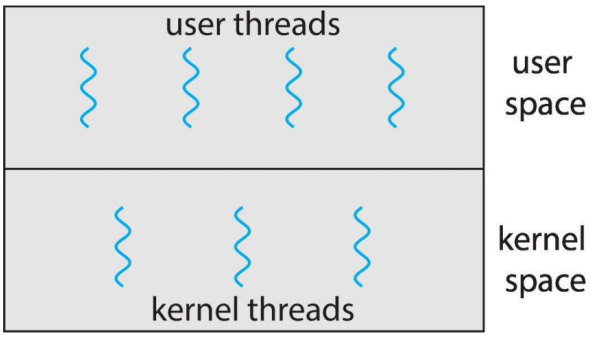
L'intuizione che lui aveva, era che il concetto del parallelismo aveva un limite in quanto ci sono delle istruzioni che devono per forza essere sequenziali quindi il ragionamento: dato un programma, prendo il data-set, lo divido in n pezzi e ogni pezzo di programma, ogni processo, e ogni thread lo realizzo in parallelo in 1/n… non è possibile!
Ci sono delle operazioni preliminari e delle operazioni conclusive in ogni programma che non possono essere realizzate in parallelo, come aprire il file, quindi se noi, dato un programma, identifichiamo qual è la percentuale non parallelizzabile, per esempio diciamo che è S=0,25 cioè se il 25% di un programma è sequenziale, solo sul 75% posso parallelizzare.
Questa ci formula dice che lo speed up, cioè l'accelerazione che ottengo, è sempre minore uguale di questa quantità, dove s è la percentuale non parallelizzabile ed è n è il numero di core o processo disponibile a disposizione.
Se noi mettiamo questi valori, per esempio qui mettiamo 25% e qui ci mettiamo qualsiasi numero e andiamo a calcolare lo speed up, avremo delle belle sorprese!
Se abbiamo che un'applicazione S pari al 25% quindi ben tre quarti sono parallelizzabili, ma il 25% no, passando da un core a due core, cioè da un singolo processore a due, lo speed up non diventa due, passando a due, ma diventa solo 1,6; se mettiamo qui n pari a quattro lo speed up diventa 2.28 e così via.

- Se il 40% di un'applicazione è seriale il massimo aumento di velocità è di 2.5 volte

La porzione seriale di un'applicazione ha un effetto dominante sulle prestazioni ottenibili con l'aggiunta di nuovi core.
I processori non possono andare più veloci per sempre perché l'aumento di velocità comporta produzione di calore e il calore distrugge la CPU.
La legge di Amdahl ci dice che l'accelerazione che possiamo ottenere sul sistema multicore non dipende solo dal numero dei core utilizzata, ma è legata alla porzione seriale del programma in esecuzione, indipendentemente dal numero dei core che utilizza.

Sostanzialmente aumentando il numero dei core, quel 25% di parte non parallela fa sì che alla fine, lo speed up tende ad essere 1/S. Possiamo mettere anche un miliardo di processori ma quello che conta in realtà è la percentuale sequenziale del programma. Quindi Amdahl diceva: piuttosto che concentrarci a fare 10.000 processori facciamone 10 ma facciamolo più veloce.
Per molti anni in effetti aveva avuto ragione, ma dopo purtroppo, si è scoperto che i processori non possono andare più veloci per sempre, poiché velocità implica produzione di calore, e il calore distrugge la CPU, infatti nei processori ci sono le ventole di raffreddamento, però c'è un limite fisico oltre il quale la ventola non può più girare per salvare la cpu.
Per questo negli anni 2000 si è diffuso molto il multi-core, in parallelo si sono sviluppate maggiori velocità e contemporaneamente maggior numero di core.

È importante ricordare come questa legge o teorema di Amdahl, seppure diciamo con le sue limitazioni, ci che l'accelerazione che possiamo ottenere sul sistema multi-core non è pari al numero dei core ma in realtà è determinata al limite dalla porzione sequenziale del programma stesso.



Ci sono due approcci per gestire i thread:
•A livello utente (user threads)
•A livello sistema o thread a livello kernel (kernel threads)

Thread a livello utente

Sono gestiti come uno strato separato al di sopra del nucleo del sistema operativo. Sono realizzati tramite librerie di funzioni per la creazione, lo scheduling e la gestione dei thread, senza alcun intervento diretto del nucleo

- POSIX threads è la libreria per la realizzazione di thread utente in sistemi UNIX-like
- Windows threads per sistemi windows
- Java threads per la JVM.

I thread a livello utente sono gestiti da uno strato sopra il sistema operativo, sono tipicamente creati usando le librerie dei linguaggi di programmazione ed esistono anche delle librerie posix (posix è uno standard che definisce in ambito unics il formato delle system call in modo che siano uniformi fra le diverse versioni), quindi quando parliamo di thread a livello utente ci riferiamo ai thread che noi possiamo creare usando sostanzialmente strumenti di alto livello, sopra il sistema operativo.

Thread a livello kernel

Sono gestiti direttamente dal SO: il nucleo si occupa di creazione, scheduling, sincronizzazione e cancellazione dei thread nel suo spazio di indirizzi. Supportati da tutti i sistemi operativi attuali: Windows, Solaris, Linux, Mac OS, iOS e Android.

Dentro il sistema operativo esistono i kernel thread, gestiti direttamente dal kernel e il sistema operativo è in grado di trattare questi kernel thread come se fossero dei processi, quindi tutte le funzioni tipiche, di creazione, schedulazione, sincronizzazione eccetera eccetera.

Relazioni tra i thread

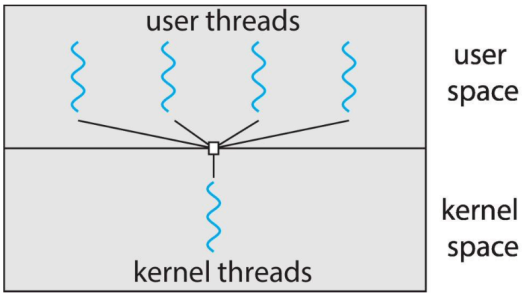
Esistono tre possibili relazioni tra i thread utenti e thread del sistema operativo:

- Modello molti a uno (M:1)
- Modello uno a uno (1:1)

- Modello molti a molti (M:M)

Qual'è la relazione fra i thread creati dall'utente e i thread gestiti dal sistema operativo ?
Esistono tre possibili relazioni: • il modello molti a uno, • il modello 1 a 1 e • il modello molti a molti.
Questi tre modelli ci dicono qual è il modo con cui i thread dell'utente sono mappati sui thread del sistema operativo.
La figura di riferimento è questa: abbiamo i thread creati dai programmi e i thread gestiti dal s.o.; allora andiamo ad analizzare i thread match.

MODELLO MOLTI A UNO



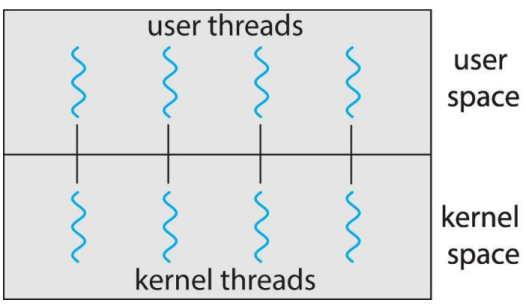
È l'approccio più semplice. Sostanzialmente noi possiamo creare tanti thread nel programma (Java, Python, ...) ma tutti questi thread sono poi eseguiti all'interno del kernel come unico thread. In altre parole: i thread sono implementati a livello dell'applicazione e il loro scheduler non fa parte del SO, che continua ad avere la visibilità del processo.

Vantaggi
- Gestione efficiente dei thread nello spazio utente (scheduling poco oneroso)
- Non richiede un kernel multithread per poter essere implementato

Svantaggi
- L'intero processo rimane bloccato se un thread invoca una chiamata di sistema di tipo bloccante
- I thread sono legati allo stesso processo a livello kernel e non possono essere eseguiti su processori fisici distinti

L'approccio più semplice è detto modello molti a uno; sostanzialmente noi possiamo creare tanti thread nel programma, ma tutti questi thread sono poi eseguiti all'interno del kernel come unico thread. Quindi vuol dire che più thraed sono mappati su unico kernel thread, quindi dal punto di vista della programmazione noi abbiamo definito più thread ma all'atto pratico non girano veramente in parallelo, perché vengono visti dal sistema operativo come un unico processo.
Si tratta chiaramente di un approccio che ha pro e contro, più contro che pro diciamo è un approccio che si usava nelle prime versioni dei thread, avevano i vantaggi di permettere all'utente di ragionare in termini più thread ma aveva lo svantaggio che di fatto si perde il parallelismo, perché se poi tutto finisce dentro un unico thread è chiaro che non c'è concorrenza.
L'approccio molti ad uno è vantaggioso per il sistema operativo perché è molto leggero, tutta via da molti anni in disuso.

MODELLO UNO A UNO



È l'approccio più intuitivo e utilizzato oggi. A ciascun thread a livello utente corrisponde un thread a livello kernel.
Il kernel "vede" una traccia di esecuzione distinta per ogni thread.
I thread vengono gestiti dallo scheduler del kernel (come se fossero processi; si dicono thread nativi).

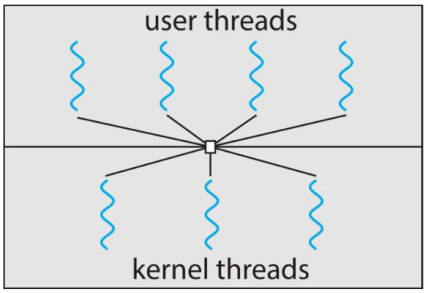
Vantaggi
- Scheduling molto efficiente
- Se un thread effettua una chiamata bloccante, gli altri thread possono proseguire nella loro esecuzione
- I thread possono essere eseguiti su processori fisici distinti

Svantaggi
- Possibile inefficienza per il carico di lavoro dovuto alla creazione di molti thread a livello kernel (limiti imposti a priori)
- Richiede un kernel multithread per poter essere implementato

Esempi
- Windows XP, Vista e versioni attuali
- Linux, Solaris (versione 9 e successive)

L'approccio più intuitivo e anche quello che si usa di più oggi è uno a uno, cioè ogni thread che io creo nel programma viene gestito con un thread indipendente dal sistema opeartivo. Il numero di thread percepiti dall'utente coincide col numero di thread percepiti e gestiti dal sistema operativo.
In questo caso si parla di thread nativi perchè sono thread che sono nativamente visti come tali dal sistema operativo quindi vengono schedulati e vanno in concorrenza gli uni con gli altri sfruttando il parallelismo.
Si tratta chiaramente di un approccio molto efficiente in termini di schedulazione, perché si sfruttano tutte le risorse disponibili, ma presenta anche alcuni svantaggi: la necessità di gestire la concorrenza, un certo sovraccarico del kernel perché se abbiamo 1000 programmi ogni programma crea 1000 thread, avremo un milione di trend da gestire da parte del kernel.
È un estremizzazione però poi lo scheduler deve gestire, non più 1000 processi ma, un milione di thread.
Tutte le versioni di Windows e Linux ormai portano questo modello.
Poi ci sarebbe in teoria un modello ibrido che cerca di contemperare i vantaggi e gli svantaggi dei 2 approcci precedenti.

MODELLO MOLTI A MOLTI



È un modello ibrido che cerca di contemplare i vantaggi e svantaggi dei due approcci precedenti. Si mettono in corrispondenza più thread a livello utente con un numero minore o uguale di thread a livello kernel. In altre parole, il sistema dispone di un pool di thread -worker - ognuno dei quali viene assegnato di volta in volta ad un thread utente.

Vantaggi
- Possibilità di creare tanti thread a livello kernel quanti sono necessari (per la particolare applicazione e sulla particolare architettura), eseguibili in parallelo su architetture multiprocessore.
- Se un thread invoca una chiamata di sistema bloccante, il kernel può fare eseguire un altro thread.

Svantaggi
- Difficoltà nel definire la dimensione del pool di worker e

- Le modalità di cooperazione tra i due scheduler

Esempi:
- Windows con il ThreadFiber package § Solaris (versioni precedenti alla 9)
- Tru64 UNIX

In questo modello: molti a molti, noi possiamo allocare un numero di thread, di livello sistema, che è diverso e tipicamente inferiore rispetto al numero di thread ci sono a livello utenti. Facendo un mapping, cioè per esempio dicendo: questo thread si prende carico di seguire questi due, facciamo una sorta di raggruppamento.
Questo raggruppamento serve a mantenere un livello di parallelismo a livello di sistema fisso, cioè non esagerare con un parallelismo, senza però inibire il parallelismo nei programmi.
Dal punto di vista pratico questo si realizza con un concetto quello di thread pool, un'insieme di thread predefinito, dieci ad esempio e poi questo thread pool che è il numero fisso e si occupa di eseguire ciascuno del codice in parallelo.
Il mapping va gestito da un'altro thread.
Quindi c'è un thread, un mapper che si occupa di fare questo poolling.
La ripartizione si fa in base al carico di norma, di solito si fa una specie di monitoraggi delle risorse richieste dei vari thread e si raggruppano facendo una distribuzione uniforme del carico, come nei modelli master Worker, in cui un worker corrisponde a un thread di livello sistema che si occupa di un determinato numero di processi.
Se uno volesse usare questi threader dovrebbe usare linguaggi moderni, che hanno delle librerie che permettono appunto di creare dentro un programma come si crea una classe di una funzione. Creare un thread specificando quali operazioni deve fare quel thread.
Il linguaggio che più di altri ha introdotto questa possibilità di usare i thread è Java.
È un linguaggio di successo grazie al multi treading, la facilità con cui posso creare thread che girano in parallelo, la cosa interessante è che nei primi anni in cui Java creava il multi-treading di fatto i processori erano tutti mono trader, il multicore non c'era. Molti impararono ad usare questo trading, senza il vantaggio di velocità ma solamente la modularità, infatti con l'avvento del multi-core poi questi programmi hanno iniziato a girare più veloci.

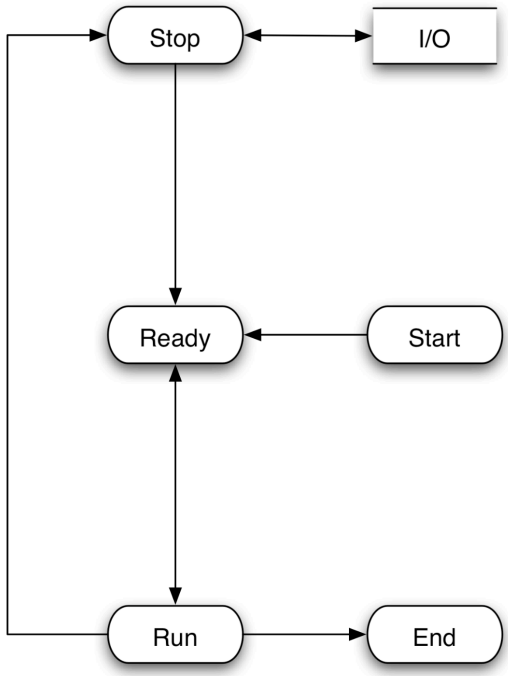
SCHEDULING DELLA CPU

Andremo a trattare i seguenti argomenti al riguardo:
- Concetti di base
- Criteri di scheduling
- Metriche di scelta degli algoritmi di scheduling, tra cui: FCFS, SJF, Round-Robin, Con priorità, A code multiple.
- Scheduling in multiprocessori
- Scheduling real-time
- Valutazione di algoritmi

Lo scheduling della CPU
In particolare vedremo: alcuni concetti di base sulle funzioni dello scheduler, quali sono i criteri di scheduling, quali sono le metriche che usiamo per decidere di usare un algoritmo piuttosto che un altro, gli algoritmi di scheduling principale che si usano nei sistemi operativi, quali sono i principali tipi di scheduling che vengono utilizzati che sono: •Scheduling Shortest-Job-First (SJF), •Scheduling First-Come, First-Served (FCFS), • Scheduling con Priorità, •Scheduling a code multiple, alcuni cenni dello scheduling sui sistemi multi-core e multi-processore e lo scheduling real-time.

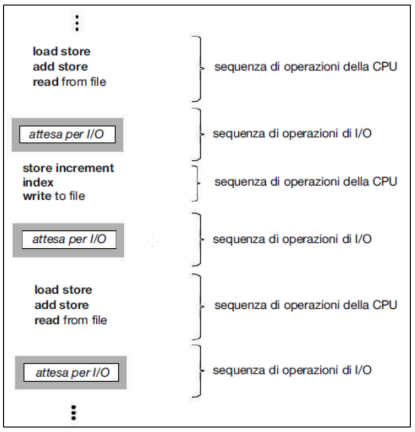
Lo scheduling della CPU è l'elemento fondamentale dei SO con multiprogrammazione; questo perché la necessità dello scheduling nasce dall'esigenza di massimizzare l'utilizzo delle risorse e del processore durante l'uso dei vari programmi.
La massimizzazione della CPU si ottiene assegnando al processore i processi che sono pronti (ready) per eseguire delle istruzioni. Il concetto di coda assume una

connotazione diversa: i processi da eseguire sono in coda secondo un criterio di esecuzione specifico e non in semplice ordine “cronologico”.



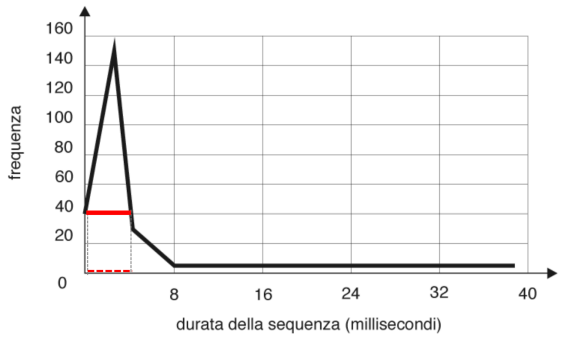
Per comprendere meglio, si pensi alla coda postale: all’arrivo, l’ultimo arrivato è l’ultimo della coda ma il primo ad essere servito non è necessariamente il primo della coda ma potrebbe essere una persona in un'altra posizione che ha, ad esempio, il numerino per la spedizione di un pacco cui reparto si è liberato, mentre la postazione del pagamento bollettino, che corrisponde al tipo di servizio richiesto dal primo in fila, risulta ancora occupato. In modo analogo avviene lo scheduling dei programmi nella CPU, in cui i programmi rapidi hanno generalmente la precedenza su quelli lenti, al fine di velocizzare il tempo di utilizzo totale della CPU.
Non si parla tanto di coda, quindi, ma di lista, in cui un elemento viene estratto senza rispettare l’ordine di sequenza.

L'attività di scheduling del processore è l'attività fondamentale dei sistemi operativi multi programmati (dico nei multi programmati poiché in un sistema mono-programmato ci giro un solo programma per volta, c'è poco da schedulare).
Lo scheduling è importante perché dobbiamo in qualche modo massimizzare l'uso della risorsa fondamentale del computer che è il processore, l’obiettivo generale dello scheduling è massimizzare l'uso della CPU; si massimizza il lavoro della CPU scegliendo fra tutti i processi che sono pronti, quindi quelli sono stato ready, quello che è più opportuno mandare in esecuzione.
Nelle scorse lezioni abbiamo parlato di coda, ma questo concetto lascia intendere che noi mandiamo in esecuzione sempre il processo che si trova in cima alla coda, questo potrebbe essere vero e sopratutto funzionale se siamo capaci di mettere sempre in cima alla coda il processo che più ci conviene eseguire; in realtà abbiamo due possibilità di vedere questa coda: • chiamandola coda, quando in realta non lo è perché noi estraiamo da qualsiasi posizione, oppure • estrarre effettivamente dalla prima posizione al patto che il processo non lo inseriamo sempre in ultima posizione, ma nella posizione opportuna quando arriva.
Immaginate le due alternative: una persona va alla posta e si mette in coda per l'ultimo, ma quando c'è da andare allo sportello non viene chiamato il primo della coda ma uno diverso, l’altra alternativa è che uno arriva e viene messo effettivamente in una coda, ma se deve fare una cosa veloce per esempio lo mettono davanti.

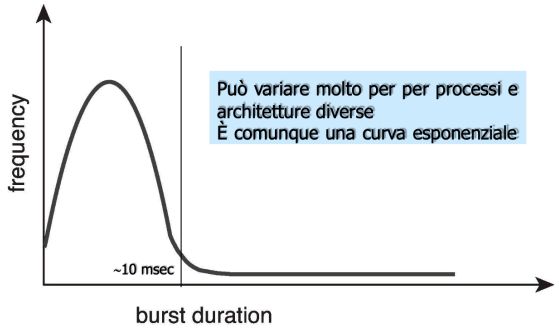


Importante è il Ciclo CPU Burst-I/O Burst: L'esecuzione di un processo comincia con una sequenza di operazioni d'elaborazione svolte dalla CPU (CPU burst), seguita da una sequenza di operazioni di I/O (I/O burst), quindi un'altra sequenza di operazioni della CPU, di nuovo una sequenza di operazioni di 1/O, e così via. Ogni sequenza di operazione di CPU prende il nome di CPU burst,

letteralmente raffica di operazioni di CPU, così come ogni sequenza di I/O è una "raffica" di operazioni di I/O. L'ultima sequenza di operazioni della CPU si conclude con una richiesta al sistema di terminare l'esecuzione del processo (exit).
Questa alternanza di CPU e I/O burst fa parte dell'incastro operato dallo scheduler nel tentativo di massimizzare l'uso della macchina.



La durata dei burst è molto variabile. La distribuzione dei CPU burst dipende dalle attività dei diversi programmi.
La frequenza dei CPU burst brevi è molto alta mentre la frequenza dei CPU burst lunghi è molto bassa. Ad esempio, nel grafico a destra, se si indica con le ascisse la durata della sequenza in millisecondi e con le ordinate la frequenza della durata, si vede che la maggior parte dei CPU burst hanno breve durata, mentre la frequenza dei burst lunghi è bassa.

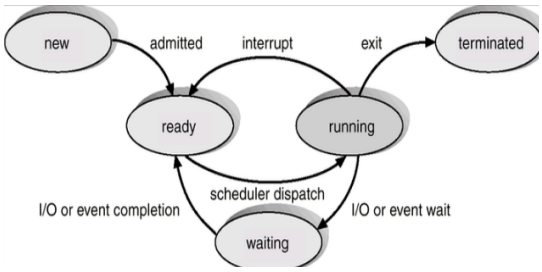


Processi CPU bound: passano più tempo ad utilizzare la CPU, programmi di calcolo
Processi I/O bound: eseguono più operazioni di lettura e scrittura, programmi di database (scrittura e memorizzazione)
Lo scheduling per le due tipologie di processi è diverso.

Nei sistemi operativi succede che il processo arriva e non viene messo in fondo al lavoro ma siccome magari un processo veloce venga messo all'inizio della coda, quindi in realtà il concetto di coda è un concetto non legato strettamente alla fifo (first in first out) ma è legato più che altro al concetto di lista, cioè una struttura dati in cui il dato processo permane finche non viene scelto.
Un concetto che è molto importante, che è il concetto di alternanza CPU-burst I/O-burst.
Ricordo che un processo nella sua vita fa sempre una sequenza di operazioni di cpu, seguite da una serie di operazioni di I/O seguite di nuovo da operazioni CPU e così via.
Ogni sequenza di operazioni di cpu prende il nome di Cpu-burst (burst significa raffica), cioè raffica di operazioni di CPU; lo stresso discorso vale per le operazioni di I/O.
L'ultima operazione che fa un programma è un'operazione di CPU, in cui il programma termina se stesso, la exit.
Se noi andiamo a guardare un programma qualsiasi troveremo delle operazioni legate alla cpu-burst, poi una sequenza di operazioni di I/O, che dal punto di vista del codice significa "essere in attesa del completamento dell'input o dell'output, chiaramente quando un processo è in attesa di I/O lascia la CPU a vantaggio di qualcun altro. Questo è il ciclo.
L'alternanza di CPU ed IO è alla base del gioco di incastri che fa lo scheduler, quindi lo scheduler cerca sempre di incastrare processi quando un processo va a fare un I/O-burst ci va a mettere un processo che deve fare un CPU-burst e viceversa.
Quanto sono lunghi questi CPU-burst e questi IO-burst ?
Dipende, ci sono diverse distribuzioni nelle durate dei CPU-burst, di solito i burst sono molto brevi, cioè il burst di cpu lunghi sono rari, se noi su un grafico, andassimo rappresentare sulle x la durata CPU-burst e sulle y con quale frequenza si presentano questi burst di diversa durata, vediamo che questo picco abbiamo 160 burst di lunghezza di 3 millisecondi; ci sono tantissimi i burst che durano poco, mentre i burst di 32 o 40 ms sono molto pochi, 5 o 10.
Il grafico ci dice che la gran parte dei burst sono molto brevi, pochissimi sono quelli lunghi.
La curva esponenziale del grafico ci dice che: i burst di lunghezza superiore a questa soglia qui (10 ms) sono veramente pochi, quasi tutti ricadono al di sotto di questa soglia.

Esiste un'altro importante concetto, ovvero la differenza tra processi CPU- Bound e ALU-bound; i primi hanno una prevalenza sull'orizzonte temporale di operazioni, mentre processi alu-bound passano più tempo a scrivere e leggere che fare operazione.
Vi vengono in mente dei processi che hanno delle prevalenze di ALU-bound o CPU-bound ?
Per esempio i programmi di database sono programmi alu-bound, mentre i programmi di calcolo sono cpu-bound.
Lo scheduling deve tenere conto di queste differenze.

Scheduler a breve termine

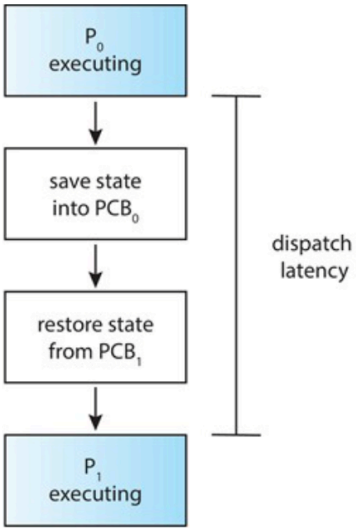


Lo scheduler a breve termine seleziona uno tra i processi in memoria pronti per essere eseguiti (ready queue) e lo assegna alla CPU.
Lo scheduler interviene quando un processo:
•Passa dallo stato running allo stato waiting.
•Passa dallo stato running allo stato ready.
•Passa dallo stato waiting allo stato ready.
•Termina.

Lo scheduler della CPU è anche detto scheduler di breve termine, lo scheduler della cpu interviene quando il processo deve passare dallo stato running allo stato waiting, quando il processo va dallo stato running allo stato ready, dallo stato waiting passa allo stato ready e quando il processo termina, lo scheduler gestisce tutte queste fasi.
Alcune di queste azioni che abbiamo citato, sono azioni nonpreemptive in cui lo scheduler prende semplicemente atto dell'azione, in altri casi invece lo scheduler impone l'azione per cui sono dei processi preemptive.

Nei casi 1 e 4 lo scheduling è **nonpreemptive** (senza prelazione), cioè il processo stesso cambia il proprio stato.
Negli altri casi è **preemptive** (con prelazione), in cui è lo scheduler ad imporre l'azione.

Quali sono i casi nopreemptive ?
Sono il caso 1 e 4, quando il processo va dallo stato running allo stato waiting, in questo caso è il processo che si autosospende, questo perché si va allo stato waiting quando si vuole fare un'operazione di IO, non è lo scheduler che ci sospende, siamo noi a spenderci, cosi come quando il processo termina, lo fa perché ha finito il suo codice non perché lo ha deciso lo scheduler.
Tutti gli altri casi sono preemptive perché lo scheduler impone dei cambi di stato.



All'interno dello scheduler troviamo il dispatcher, che ha il compito di passare il controllo ai processi selezionati dallo scheduler della CPU per la loro esecuzione. Esso svolge:
•Il context switch
•Il passaggio al modo utente
•Il salto all'istruzione da eseguire del programma corrente.
Tali operazioni sono eseguite tramite codice specifico in grado di ottimizzare le azioni. Per quantizzare l'efficienza di tali operazioni si parla di latenza di dispatch: tempo impiegato dal dispatcher per fermare un processo e far eseguire il successivo.

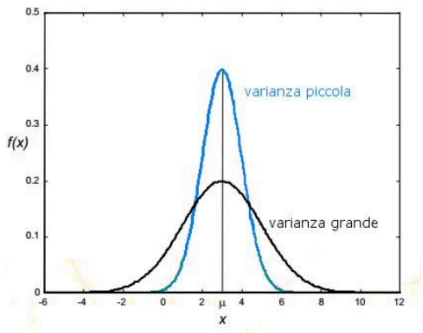
All'interno dello scheduler della cpu esiste un componente il modulo dispatcher che si occupa del cambio di contesto, quindi si occupa di passare il controllo al processo selezionato e contestualmente di salvare lo stato del processo viene deselezionato; si occupa anche di passare l'esecuzione del processo in modalità user mode a kernel mode.
È stato implementato un modulo separato perché cosi si aumenta l'efficienza rispetto al dover implementare delle funzioni che gestiscano il tutto, è stato infatti coniato il termine latenza di dispatch per indicare il tempo che questo modulo impiega per cambiare il contesto del processo in esecuzione.
Sostanzialmente quindi la potenza dei dispatcher comprende queste due azioni: salvare lo stato del precedente processo per ripristinare lo stato del pc per il nuovo processo.

Criteri di Scheduling

Nella scelta di una strategia di scheduling occorre tenere conto delle diverse caratteristiche dei programmi. I CRITERI da considerare sono:

- Utilizzo della CPU - avere la CPU il più attiva possibile
- Throughput - numero di processi completati nell'unità di tempo
- Tempo di turnaround (tempo di completamento)- tempo totale per eseguire un processo
- Tempo di waiting (tempo d'attesa) - tempo totale di attesa sulla ready queue
- Tempo di risposta - tempo che intercorre da quando viene inviata una richiesta a quando si produce una prima risposta.

Quali sono i criteri di scheduling ? Ovvero quali parametri dobbiamo in qualche modo considerare nella scelta dell'algoritmo?
I parametri sono: • l'uso della CPU, quindi cercare di rendere la cpu più attiva possibile; •Throughput o produttività, che si definisce come il numero di processi completati nell'unità di tempo; • il tempo di turnaround o anche detto tempo di puntellamento, ovvero il tempo necessario per eseguire un processo, cioè il tempo di percorrere dall'inizio alla fine un processo; • il tempo di waiting o tempo di attesa, che è il tempo totale che un processo passa in attesa sulla ready queue; • il tempo di risposta, che è il tempo che intercorre da quando viene inviata una richiesta di esecuzione fino a quando la richiesta a massimizzare l'uso della CPU.



$$\sigma^2 = \frac{\sum_{i=1}^{n} (x_i - \overline{x})^2}{n}$$

La varianza è la media aritmetica dei quadrati delle differenze tra ogni valore X_i della distribuzione e un valore medio preso come riferimento.

Gli scopi dietro tali criteri sono:
•Massimizzare l'uso della CPU e il throughput
•Minimizzare il tempo di turnaround, di waiting e di risposta
•Ottimizzare i valori medi
•Minimizzare la varianza del tempo di risposta

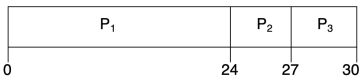
Spesso, piuttosto che prediligere valori con medie minori è preferibile minimizzare la varianza per avere un range di valori accettabili in tutti i casi anziché trovarsi con tempi esagerati. Ad esempio, si supponga di creare un dispositivo con tempo di risposta di 5 millisecondi di media, tuttavia tale media si ottiene da una curva che va da 1 a 9 millisecondi, cosa non accettabile; si preferisce in tal caso una media di 6 millisecondi con valori distribuiti da 5 a 7 e quindi con varianza minore.

Scheduling First-Come, First-Served (FCFS)

- Il primo arrivato è il primo servito (gestito con coda FIFO).

Processo	Burst Time
P_1	24
P_2	3
P_3	3

- Supponiamo che i processi arrivino tutti a T=0 nell'ordine: P_1 , P_2 , P_3
  Lo schema di Gantt è:

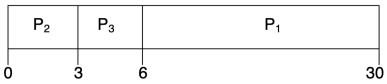


- Tempo di waiting per: $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Tempo di waiting medio: (0 + 24 + 27)/3 = 17

Supponiamo che i processi arrivino nell'ordine

P_2 , P_3 , P_1 .

- Lo schema di Gantt è:



- Tempo di waiting per $P_1 = 6; P_2 = 0; P_3 = 3$
- Tempo di waiting medio: (6 + 0 + 3)/3 = 3
- Molto meglio che nel caso precedente.

- *Effetto convoglio:* i processi "brevi" attendono i processi "lunghi".

Vediamo ora l'esempio in figura: si vuole ottimizzare il flusso d'azione di una serie di processi. Seguendo una logica FIFO (first in first out), si osserva che, se l'ordine dei processi è in successione P1, P2 e P3, il tempo di attesa medio per l'esecuzione di tutti i programmi è 17.
Se, invece, l'ordine d'arrivo dei programmi è P2, P3 e P1, si osserva una netta diminuzione del tempo medio di waiting (3) dovuto al fatto che i processi più rapidi vengono eseguiti prima di quelli più lenti.

Questo fenomeno, tipico dei sistemi FCFS, è detto effetto convoglio, secondo cui i processi "brevi" devono attendere i processi "lunghi": organizzare al meglio l'esecuzione dei processi incide positivamente sul tempo d'attesa e quindi sull'efficienza della macchina.