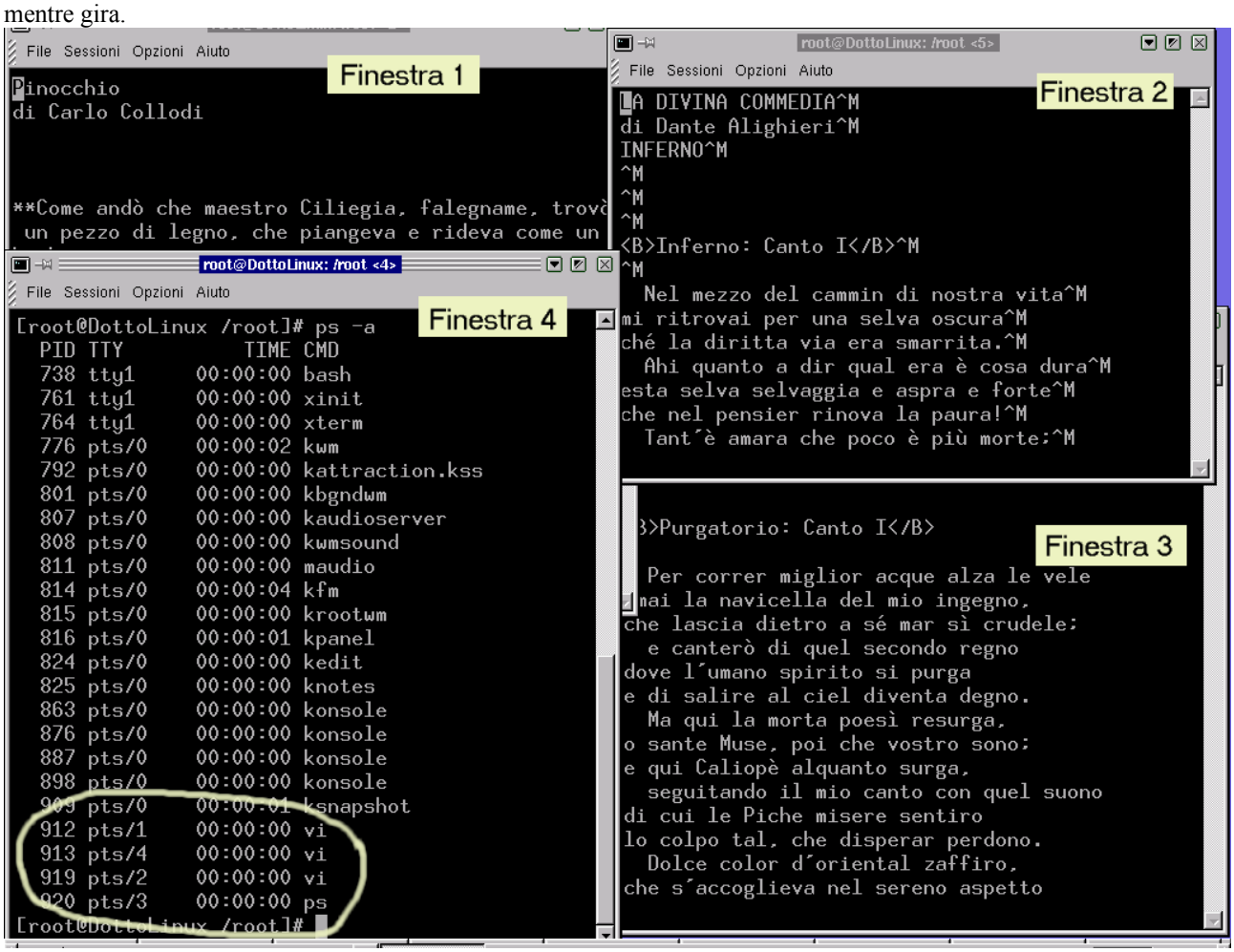
# 1 I processi e la loro gestione

Quello di processo è un concetto astratto, come evidenziato dalla bella definizione di Tanenbaum:

*Processo: l'astrazione di un programma che sta girando.*

La realizzazione pratica di questo concetto astratto richiede:
- il codice macchina del programma da eseguire
- i dati del programma
- una struttura dati utile al kernel del S.O. per la gestione del processo ("descrittore di processo")

Il S.O. dovrà tenere traccia di tutte le informazioni necessarie perché ogni processo sospeso possa riprendere l'esecuzione quando verrà il momento di farlo.
Creazione dei processi
Un processo esiste quando il suo codice ed i suoi dati sono già stati caricati in memoria ed il suo descrittore è stato creato.
Al momento della creazione di ogni processo ad esso viene associato un numero univoco, che verrà usato in seguito come "chiave" per riferirsi a quel processo.
In Unix questo numero si chiama Process Identifier, o process ID (PID). Conoscere il PID di un processo è indispensabile se gli si vogliono spedire dei messaggi.
Come accennato nel capitolo precedente, un file contente un programma può dare origine a molti processi diversi, dato che, in un sistema multiprogrammato, ogni utente può lanciare una o più copie delle stesso programma.
Ciascuna di queste copie genera un processo diverso anche dagli altri processi che eseguono lo stesso programma.
Prendendo in prestito la terminologia della programmazione ad oggetti si potrebbe dire che il programma è una "classe", mentre un processo è un'"istanza" della classe.
Dunque un processo è una delle tante possibili istanze in esecuzione di un programma, una "istanza"[1] del programma mentre gira.




1 attenzione, questo termine viene usato nella programmazione ad oggetti con un significato diverso da questo!

Figura 1: lo stesso programma eseguito da tre processi diversi

La Figura 1 mostra tre processi che eseguono lo stesso programma in Linux, nelle finestre (1) , (2) e (3).
Il programma è "vi", un semplice editor di testi. Ciascuna delle tre finestre mostra lo stato di un processo diverso, la finestra (4) mostra il risultato dell'esecuzione del comando di S.O. ps –a, che visualizza tutti i processi attivi in un determinato istante. I processi numero 912, 913 e 919 sono i tre processi nelle finestre da 1 a 3 ed eseguono lo stesso programma "vi".
Le tre diverse "istanze" di vi sono peraltro del tutto indipendenti fra di loro, infatti stanno lavorando sui tre diversi file pinocchio.txt (1), inferno.txt e purgator.txt.

Per i programmi al di fuori del kernel ed anche per la gran parte dei programmi del kernel, ogni processo vede una macchina virtuale, per cui agisce come se disponesse di una CPU e di una memoria solo per sé stesso.

Per esempio in Linux gli unici processi che non hanno memoria virtuale sono i thread interni al kernel ed i "demoni" (daemons):.

Sarà compito del gestore della memoria, che è parte del S.O., realizzare un meccanismo per proteggere la memoria e fare in modo che un processo non possa interferire con la quella assegnata ad un altro, né intenzionalmente, né per errore.
Per realizzare questa protezione il S.O. al momento della creazione di ogni processo provvederà a riservargli una porzione della memoria virtuale.
Se un qualsiasi altro processo non "di sistema" tenterà di accedere alla memoria riservata per il processo appena creato esso verrà terminato dal S.O.
In questo modo si è assicurati che, se un processo tenta di danneggiare la memoria altrui sarà solo esso a subirne le conseguenze, e non tutto il sistema.
Quando un processo ha una porzione di memoria virtuale ad esso riservata si suole dire che il processo "esegue in un suo spazio di memoria".

## *1.1 Stati di un processo*

Nei sistemi multiprogrammati a CPU singola tutti i processi, tranne uno, sono sospesi. Un unico processo è invece in esecuzione sulla CPU.
Dunque ogni processo può essere almeno in due "condizioni" diverse: una condizione (stato) in cui è in esecuzione ed un'altra nella quale è sospeso.
Questo evidenzia una differenza importante fra processi e programmi: un processo ha uno stato, un programma no.
In un sistema multiprogrammato se un processo ha necessità di una risorsa che è attualmente impegnata da un altro, esso deve essere temporaneamente sospeso, in attesa che la risorsa venga liberata.
Naturalmente il processo che viene sospeso deve poter essere "congelato" e ripreso in modo che continui senza intoppi.
Quando un processo viene sospeso è perciò indispensabile copiarne in memoria ogni informazione che permetta di riprenderne l'esecuzione come se nulla fosse successo.
La struttura dati ove viene copiato lo stato di un processo al momento di una sospensione viene detta "**descrittore di processo**".
Mentre un programma in un S.O. monoprogrammato non ha la necessità di avere un descrittore, ogni processo di un S.O. multiprogrammato deve averne uno.
Per alcuni dettagli sulla gestione dei descrittori si veda nel prosieguo del capitolo.

Dunque abbiamo individuato due "stati" per un processo: uno stato di "esecuzione" ed uno di "sospensione".

Chiamiamo "**running**" ("in esecuzione") la condizione in cui si trova un processo nel momento in cui, avendo disponibilità della CPU, sta proseguendo nella sua esecuzione.

Per lo stato di "sospensione" è necessario fare delle distinzioni; in particolare bisogna evidenziare la ragione per la quale il processo è sospeso.
Se il processo ha la disponibilità di tutte le risorse tranne la CPU ciò significa che potrebbe proseguire normalmente, se non fosse stato sospeso dal S.O. per poter lasciar eseguire un altro processo.

Chiamiamo "**ready**" ("pronto") la condizione in cui si trova un processo quando è pronto ad eseguire ma non lo può fare perché manca temporaneamente della CPU.

Un processo pronto ha tutte le risorse di cui ha bisogno tranne la CPU, può eseguire in qualsiasi momento, basta che il S.O. decida che è il suo turno.
Un altro caso in cui un processo potrebbe essere sospeso è quando ha chiesto al S.O. una risorsa indispensabile per la sua prosecuzione e non l'ha ancora ottenuta.

In questa condizione il processo non proseguirebbe nella sua esecuzione anche se avesse la CPU, perché rimarrebbe bloccato in un loop in attesa della risorsa mancante ("busy waiting" o "attesa attiva").
Per questo tutti i S.O. general purpose sospendono i processi che richiedono una risorsa e non li fanno più eseguire fino a che non gliela possono assegnare.
Questo significa che abbiamo individuato un nuovo stato del processo, nel quale esso è sospeso, ma non può eseguire fino a che non succede qualcosa..

Chiamiamo "**blocked**" ("bloccato") o "**waiting**" ("in attesa ") la condizione in cui si trova un processo quando, per qualsiasi ragione, non ha disponibilità di una risorsa diversa dalla CPU e per questa ragione non viene fatto eseguire dal S.O..

Il fatto che un processo abbia questi diversi stati individua anche un'altra funzione del S.O., cioè il tener traccia dello stato di ciascuno dei processi.
La parte di S.O. che svolge questa funzione fa senz'altro parte del kernel, per cui tutti gli eventi che possono far cambiare di stato un processo devono avvenire con la CPU che lavora in "kernel-mode".
Il processo può chiedere al S.O., con una chiamata di sistema, di essere sospeso o terminato, ma deve essere il kernel, che è il solo ad aver accesso ai descrittori di processo, ad effettuare la transizione di stato.

Disegniamo ora un grafo degli stati[2], che comprenda gli stati di un processo e le relative transizioni:

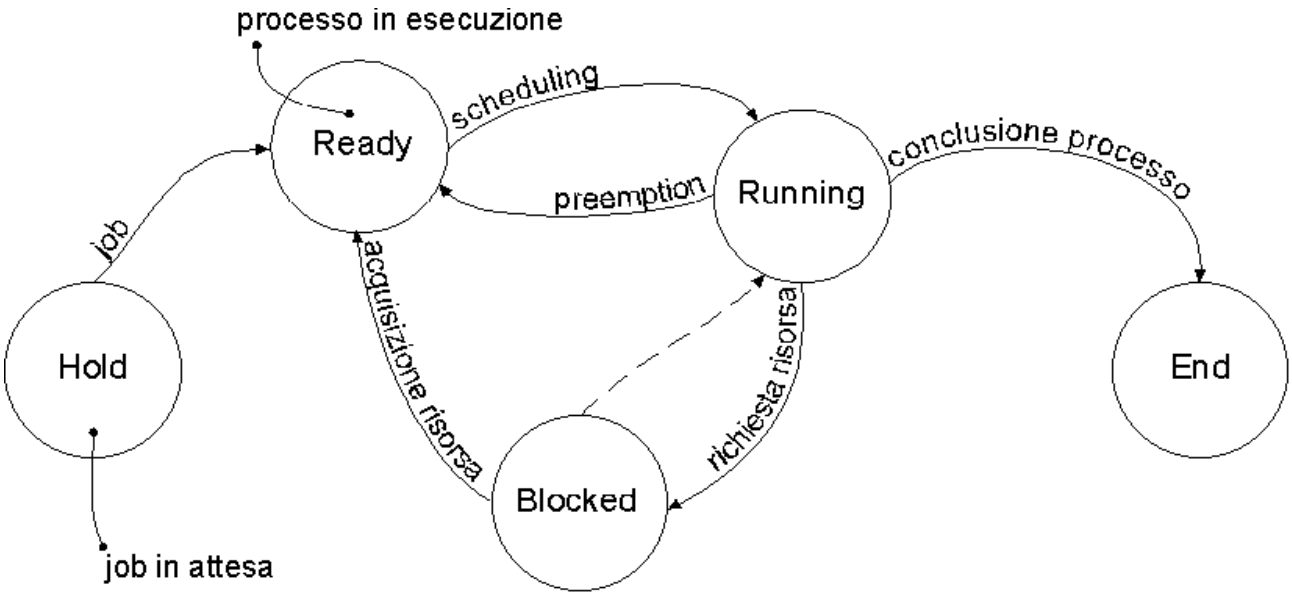



**Figura 2: grafo degli stati di un processo**

La Figura 2 comprende anche due stati "accessori" in cui il processo va a finire quando è in condizioni particolari.
Lo stato "**hold**" (attesa del job) vede il processo fuori dalla memoria in attesa di essere creato (se vogliamo essere pignoli, in questa condizione esso non è ancora un processo).
Un sistema batch potrebbe ritardare l'esecuzione dei job quando è troppo congestionato, in questi casi il job potrebbe rimanere in stato di hold. Molti S.O. non hanno uno stato di hold.
È stato anche disegnato uno stato di "end", nel quale il processo giunge solo per essere concluso.

Hold –> Ready
La fase di creazione di un processo è indicata dalla transizione fra "hold" e "ready". Viene allocata la memoria per il processo, il relativo programma viene caricato in memoria, viene creato un descrittore di processo ed il processo viene messo in stato di pronto.
I processi interattivi non possono stare in "hold", per cui vengono messi subito in stato di ready oppure, se il sistema è sovraccarico, vengono rifiutati.
La decisione di far passare un job allo stato di ready viene presa da un componente del S.O. che si chiama "**job scheduler**" o "scheduler di lungo termine".
Il job scheduler lavora su una coda di job da eseguire in batch e decide in che ordine essi debbono essere eseguiti. La decisione può tener conto di molti fattori, che contribuiscono alla massimizzazione del throughput del sistema.

2 per la definizione di grafo si veda l'indice analitico del Volume 1

Ready –> Hold
Questa transizione viene detta "roll out" e riguarda un job non interattivo che viene temporaneamente scaricato dalla memoria perché il sistema è congestionato. Ciò allo scopo di far terminare gli altri processi e di ricaricare il job escluso quando il computer è più libero.
Si fa notare che in tutti gli stati, tranne che in "hold", il processo rimane in memoria anche quando non esegue.

Ready –> Running
Questa transizione permette al processo di avanzare nella sua esecuzione. LA CPU viene accordata al processo da un sottoprogramma del S.O. che si chiama "**dispatcher"** o "**process scheduler** "[3] ("pianificatore" dei processi) ed è uno scheduler di breve termine.

Il process scheduler ha il compito di scegliere, fra tutti i processi pronti, quello da mandare in esecuzione.

Il verbo "to schedule" significa "pianificare" o "programmare" (nel senso di "mettere in programmazione" o "decidere quando si deve fare" qualcosa).

La scelta del process scheduler su quale processo far eseguire avverrà secondo precisi criteri, volti a rendere più efficiente il computer.
Il process scheduler viene chiamato ogni volta che succede qualcosa che fa cambiare lo stato di un qualsiasi processo.
Ciò può succedere per esempio in conseguenza di:
- interrupt del real time clock
- completamento di azioni di I/O
- chiamate di sistema
- eventi di sincronizzazione dei processi (p.es. wait o signal, vedi oltre)

Running –> Ready
Questa transizione avviene anche se il processo potrebbe tranquillamente proseguire.
È un'azione "arbitraria" del S.O. che decide di togliere la CPU al processo perché esso l'ha avuta per un tempo sufficiente o perché un altro processo più importante deve eseguire.
Si tratta dunque di una "preemption", come definita nel Capitolo precedente.
Nei sistemi cooperativi questa transizione avviene solo se il processo cede "volontariamente" il passo ad altri processi, per assicurare un funzionamento più uniforme del sistema (es. loop di "eventi" in Windows 3 e <).

Running –> Blocked
Quando il processo ha necessità di una risorsa la deve chiedere al S.O., con una chiamata di sistema.
Il S.O. interviene ed effettua una verifica.
Se la risorsa è accessibile ed il processo ha il diritto di averla essa gli viene allocata. Poi il processo viene messo in stato di ready, pronto a proseguire, oppure gli viene restituito subito il controllo.
Se invece la risorsa non è disponibile, allora il processo viene sospeso dal S.O. e messo in stato di blocked, ove rimarrà fino a quando la risorsa non diverrà disponibile.
Se molti processi devono avere accesso ad una stessa risorsa il S.O. crea una "**coda dei processi**" in attesa della risorsa.
Quando il processo che detiene la risorsa la libererà, il S.O. sceglierà il processo da mettere in stato di ready, fra quelli in attesa nella coda.
Spesso nel momento in cui un processo si blocca viene creato od evocato un altro processo che deve procurare la risorsa (p.es. se viene richiesta una parte di un file si crea un processo di I/O che va nell'hard disk e legge l'informazione richiesta).
La transizione potrebbe avvenire anche su richiesta del processo, che potrebbe voler "dormire" per qualche tempo, anche se non ha necessità di alcuna risorsa. In questo caso si sospenderà in una coda "degli allarmi" e verrà risvegliato quando il tempo richiesto sarà esaurito.

Blocked –> Ready
Quando la risorsa per la quale il processo era stato bloccato diviene disponibile il S.O. rimette il processo in condizione di ready, pronto per essere scelto dal process scheduler per il suo prossimo slice di esecuzione.
La transizione da Blocked a Running, indicata in Figura 2 da un arco tratteggiato, è possibile, ma la sua esistenza dipende dal Sistema Operativo.
Quando un processo passa in stato di "Ready" il kernel decide se il processo che stava eseguendo attualmente deve essere sospeso, per far eseguire al suo posto quello che è appena giunto allo stato di "pronto" ("preemption").

Running –> End
Questa transizione conclude il processo. Un processo si può concludere regolarmente, quando chiede di essere terminato dopo aver concluso la sua esecuzione, ma anche "improvvisamente", quando entra in una condizione d'errore.
Per esempio ciò può accadere quando esegue un'istruzione od un accesso alla memoria che non gli sono concesse (errore di protezione) o quando incorre in errori software che non può recuperare (stack esaurito, divisione per zero ..).

3 di solito si intende con "dispatcher" qualcosa di meno "impegnativo" di un "process scheduler"

Quando un processo viene terminato la memoria e tutte le altre risorse che riteneva vengono liberate e messe a disposizione di altri processi.
Lo stato "end" è indicato per poter dar conto della fine del programma. Dato che il processo subito dopo viene eliminato non è detto che venga effettivamente scritto nel descrittore di processo.

In Unix esiste effettivamente uno stato di "zombie", nel quale viene posto un processo quando è finito, per qualsiasi ragione, ma ha ancora un descrittore di processo, che non è stato cancellato per errore.

*Politiche di scheduling dei processi*

Lo scheduler fa tutto il possibile perché il sistema sia usato al meglio delle sue possibilità.
Deve perciò fare in modo che la CPU ed i dispositivi siano costantemente impiegati, che nessun processo rimanga senza essere eseguito per molto tempo, che gli utenti ricevano risposte in tempi accettabili, che quando il sistema è sotto carico estremo le sue prestazioni non decadano bruscamente ("graceful degradation").
Naturalmente è impossibile garantire "matematicamente" l'ottimalità di tutte le condizioni elencate, che spesso sono in contrasto fra di loro, per cui lo scheduler deve cercare buone soluzioni di compromesso, che diano prestazioni accettabili.
Si dice "politica" di scheduling (scheduling "policy" o "strategy") l'insieme delle tecniche utilizzate da un S.O. per decidere l'ordine di esecuzione dei processi.
Ben si comprende come l'algoritmo di scheduling dei processi sia decisivo per le prestazioni di un Sistema Operativo, sia perché viene eseguito molto spesso, e quindi deve essere di veloce esecuzione, sia soprattutto perché le sue decisioni possono portare ad una sottoutilizzazione del sistema.

Lo scheduling dei sistemi può essere pre-emptive, non preemptive e cooperativo.
In uno schema non preemptive i processi perdono la CPU quando terminano la loro esecuzione oppure quando si bloccano in attesa di I/O o di una risorsa.
I S.O. con scheduling preemptive possono togliere la CPU al processo anche quando essi ha eseguito per troppo tempo (es. tutti i S.O. in tempo reale).
Se lo scheduling è cooperativo il processo lascia la CPU solo quando fa una specifica chiamata al S.O. per cedere il controllo. La perdita della CPU quindi non dipende né dal tempo che il processo ha usato, né dal fatto che esso abbia richiesto un I/O.

Nel corso del tempo sono state sviluppate molte politiche di scheduling, in pratica almeno una per ogni S.O. In questo paragrafo ne discuteremo alcune delle più comuni.

First Come First Served

Il primo che arriva è il primo servito: la coda del processi è strettamente FIFO (First In First Out). Questa politica era usata nei sistemi batch, nei sistemi odierni non viene più usata per i processi interattivi, perché i processi che usano molto la CPU (CPU bound) tendono a monopolizzarla.

Shortest Remaining Time Next

Viene eseguito il processo che "finirà primo". Questa politica prevede di avere una buona stima di quanto durerà il processo e tende a fare in modo che i processi che sono vicini alla conclusione, oppure quelli brevi, siano eseguiti prima, in modo che possano liberare prima le loro risorse.
Questo algoritmo è in grado di minimizzare il tempo di attesa per l'esecuzione dei processi ed è molto valido se si può stimare bene per quanto dureranno i processi; questo è molto difficile per i processi interattivi, per cui lo scheduling a "tempo rimanente" viene usato solo per i job batch.

Round Robin

Questa è tecnica classica dei sistemi time-sharing. La CPU viene assegnata "a turno", per un numero di quanti di tempo prestabilito, a ciascun processo, uno alla volta.
Se il processo alla fine del suo tempo non è ancora terminato, o bloccato, verrà sospeso e messo nella coda dei processi pronti.
Il vantaggio principale del **round robin** è che la CPU viene suddivisa equamente fra i processi. Ciò implica buoni tempi di risposta per i terminali interattivi.
Inutile d ire che la durata del quanto di tempo è decisiva per le buone prestazioni del sistema, se è troppo breve i troppi cambi di contesto fanno diminuire l'efficienza del sistema, se è troppo lunga il ritardo di risposta dei terminali interattivi potrebbe divenire inaccettabile.

Priorità

Questo metodo, detto anche "scheduling event driven", prevede l'assegnazione di una priorità ad ogni processo. Poi, ogni volta che il dispatcher interviene, sceglie per l'esecuzione il processo pronto con la migliore priorità.
Il maggior problema di questo approccio è che i processi con cattiva priorità possono essere scavalcati continuamente da quelli più importanti, fino a correre il rischio di non eseguire mai, quando il sistema è sotto carico estremo ("**starvation**" = morte per fame!).

Per ovviare a questo problema si possono porre priorità dinamiche, variabili nel corso del tempo in modo che la possibilità di eseguire migliori con il tempo che il processo attende nella coda dei pronti.
Uno scheduling con priorità permette risposte rapide e predicibili agli eventi che riguardano i processi più importanti. È quindi adatto ai sistemi realtime.

Code multilivello

Lo scheduling a code multiple combina le tecniche precedentemente descritte per utilizzare i vantaggi di ciascuna, sia pur con u certo aggravio dovuto alla gestione più complicata.
I processi vengono classificati in base alle loro caratteristiche ed immessi in code diverse, all'interno delle quali operano scheduler con algoritmi diversi. Per i processi che hanno caratteristiche realtime, come quelli del S.O. e quelli di I/O si può applicare un algoritmo event driven, un round robin per i processi interattivi ed un SRTN per i batch.
Naturalmente per decidere quale coda usare ogni volta ci vorrà uno "scheduler degli scheduler". Per i dettagli si veda il libro di Milenkovic, al cap. 3.

## Interazioni fra i processi

Per quanto ogni processo possa eseguire in splendido isolamento rispetto a tutti gli altri, spesso si impone la necessità per i processi di comunicare fra di loro.
Questo accade sempre fra i processi che fanno parte del Sistema Operativo ed anche fra alcuni programmi utente particolarmente sofisticati, che suddividono la loro esecuzione fra diversi processi separati.
Ogni S.O. deve perciò mettere a disposizione dei meccanismi per la comunicazione fra i processi.

### *Competizione e cooperazione*

Due processi si dicono "in **competizione**" quando sono in lotta per l'accesso alla stessa risorsa.
Quando i processi sono in competizione il S.O. deve applicare tecniche sofisticate per allocare la risorsa. I problemi dovuti all'allocazione di risorse contese fra i processi sono fra i più spinosi da risolvere; qualche dettaglio verrà dati nel capitolo "Programmazione concorrente".
Due processi in competizione potrebbero non aver bisogno di comunicare fra di loro, ma solo di trovare il momento giusto per usare la risorsa; potrebbero avere solo necessità di "**sincronizzarsi**".
Due processi si dicono "in **cooperazione**" quando uno può condizionare direttamente l'avanzamento dell'altro.
Per poter coordinare fra loro le attività dei processi cooperanti è indispensabile che fra di essi si instauri una comunicazione. Dunque la cooperazione fra i processi impone al S.O. la presenza, oltre a funzioni per la sincronizzazione fra i processi, di funzioni per il reciproco scambio di messaggi.

### *Thread*

Si può dire che esistono due tipi di processi, processi "pesanti" e "leggeri" (heavyweight process o lightweight process)
I processi "pesanti" sono quelli che abbiamo trattato fino ad ora. Essi vengono chiamati solamente "processi".
I processi "leggeri" sono meglio conosciuti come "**thread**". I due tipi di "processi" si differenziano principalmente nella gestione della memoria.
Nei Sistemi Operativi che gestiscono i thread ("multithreaded") ogni processo può generare un numero arbitrario di thread, ciascuno dei quali condivide lo stesso spazio di memoria del processo che li ha generati.
Dunque un thread può interferire sulla memoria degli altri thread e dello stesso processo "padre". Il malfunzionamento di un thread può provocare il malfunzionamento anche del processo che lo ha generato e/o degli altri thread dello stesso processo.

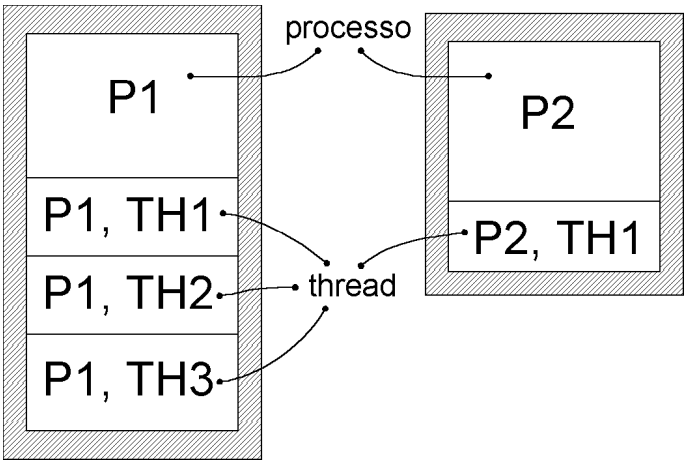



**Figura 3: spazi di memoria in processi e thread**

Ogni processo ha il suo spazio di memoria riservato per cui deve avere nel suo descrittore di processo riferimenti ad ogni parte di memoria ad esso allocata.

Se il sistema ha memoria virtuale, il descrittore deve anche contenere le informazioni necessarie al gestore della memoria per ritrovare, nello swap file, le parti della memoria del processo che sono state temporaneamente scaricate in memoria (swapped out). Con queste informazioni è in grado di ripristinare il contenuto della memoria fisica quando ne viene richiesto l'accesso.
Tutte le informazioni che permettono di gestire la memoria assegnata ad un processo vengono indicate come il "**contesto di memoria**" (**memory context**) di quel processo.

Tutti i discorsi già fatti, e quelli ancora da fare, che riguardano i processi si applicano anche ai thread, cambia solo la modalità di gestione della memoria.
Infatti, dato che i thread hanno lo stesso contesto di memoria del processo che li ha generati, il cambio fra due thread dello stesso processo ("thread switch") è un evento molto meno "costoso" del cambio fra diversi processi ("process switch").
Un "thread switch" deve cambiare solo i registri della CPU, mentre un "process switch" cambia anche i riferimenti alla memoria; deve perciò leggere diverse informazioni dal descrittore del nuovo processo.

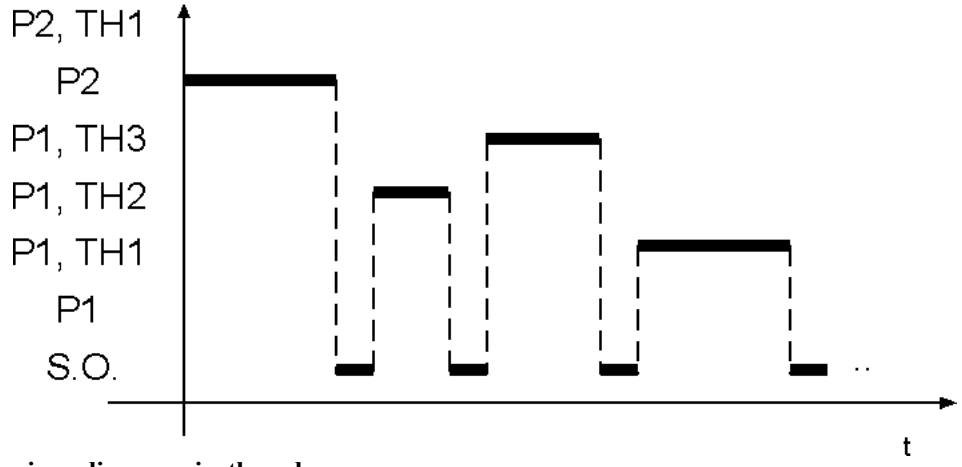
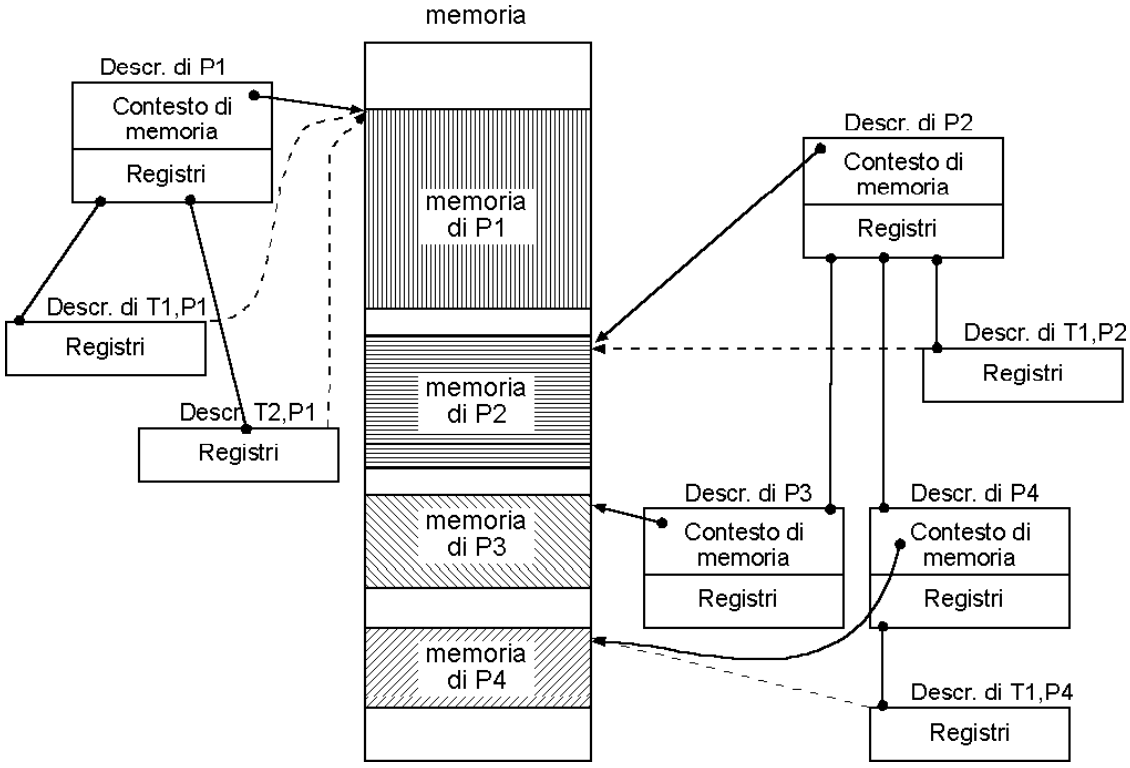



**Figura 4: esecuzione di processi e thread**

Nelle CPU moderne, che sono al loro interno altamente parallele e sono dotate di molta memoria cache, il cambio di registri non è l'unica penalizzazione che ci si può attendere da un thread switch, perché si rompe la "località"[4] del programma e perciò diviene probabile che le pipeline si svuotino, che ci siano "cache miss" e dei "page fault" della memoria virtuale.




.

4 per dettagli sul principio di località si veda l'indice analitico (località, principio di) del Volume1

**Figura 5: processi figli e thread**

Nella Figura 5 i descrittori di segmento contengono il valore dei registri ed un riferimento alla memoria, mentre i descrittori di thread non hanno riferimenti alla memoria, dato che "puntano" implicitamente alla memoria del processo "padre".

Tipi di contesto e di switch di contesto un un S.O.
!!!! da fare !!!!

Processi "figli"
Ogni processo di un S.O. multitasking può generare altri processi. I processi generati da un altro processo sono processi uguali agli altri ed hanno un loro spazio di memoria riservato. Se terminano in modo anomalo non coinvolgono altri processi.

Esempio:
Il "demone"[5] internet di Linux si chiama inetd. Quando si lancia questo programma esso "diventa" un processo, che a sua volta fa partire tutti i servizi relativi ad internet che sono stati configurati su quel computer (per esempio: telnet, ftpd ed eventualmente httpd, nfsd, smbd, ..) (per il dettaglio sui protocolli che si veda il prossimo volume). Ciascuno di quei servizi deve far eseguire almeno un processo, per cui il lancio di inetd può creare diversi altri processi.
Il comando Unix per visualizzare l'albero dei processi che sono stati generati a partire da un certo processo è pstree:

```
# pstree inetd 123 -p
inetd(123)-+-in.ftpd(234)
           `-in.telnetd(216)---login(218)---bash(219)
```

123 è un codice univoco del processo inetd, viene detto pid (**P**rocess **Id**entifier), è diverso ogni volta che il processo viene creato. L'opzione –p indica di mostrare i PID di tutti i processi visualizzati.
Il risultato mostra che nel momento in cui pstree è stato eseguito erano attivi, oltre al processo "padre" inetd due processi "figli" in.ftpd e in.telnetd, i demoni dei protocolli internet FTP e telnet. A sua volta "in.telnetd" ha un "figlio" ed un "nipote", rispettivamente "login" che gestisce la sessione dell'utente tramite telnet, e "bash", l'interprete di comandi di quell'utente.

## Gestione dei processi

### *Descrittori di processo*

Contenuto dei descrittori di processo:
PID
Diritti per la per la sicurezza
Priorità per l'esecuzione
Processo genitore
Programma eseguito
Contesto del processo (Program Counter, Stack Pointer, flags, registri (tutti))
Stato corrente
Memoria allocata
Altre risorse allocate (es. file aperti)
Informazioni per la contabilizzazione (CPU usata, risorse usate)

### *Liste dei processi*

Abbiamo visto che sono molte le occasioni nelle quali un processo necessita di cambiare di stato

Ogni coda di processi è realizzata praticamente come una lista a puntatori. I descrittori di processo sono gli elementi della lista.
In una lista a puntatori ogni elemento contiene un puntatore all'indirizzo di memoria del successivo elemento. L'ultimo elemento contiene l'indicazione che non seguono altri elementi.
La lista è la struttura dati più efficiente per quanto riguarda gli aggiornamenti, perché i suoi elementi possono essere "sparpagliati" nella memoria e mantenere comunque un'organizzazione sequenziale, anche se con la necessità di utilizzare ogni volta il puntatore.

5 Nel gergo Unix si chiama **"demone"** (**daemon**) un processo non associato ad un utente od a un terminale. Spesso un daemon è lanciato dal programma "init", al boot del sistema e funziona "in background", dietro le quinte, senza alcuna interfaccia utente.
L'effetto dell'esecuzione di un daemon è l'aggiunta di alcuni servizi a quelli disponibili sul computer (es. il daemon "inetd" permette di usare i protocolli internet (TCP/IP)).

In alcuni casi la lista dei descrittori di processo è bidirezionale, e contiene collegamenti non solo all'elemento successivo ma anche al precedente. Ciò permette algoritmi di ricerca più veloci, anche se complica la gestione.
Alle informazioni contenute nel descrittore di processo aggiungiamo dunque un puntatore al prossimo elemento ed, eventualmente, uno al precedente.

### 1.1.1 Processi in Linux

In Unix viene creato "dal nulla" un solo processo (process 0), gli altri sono tutti "discendenti" da quel processo (figli, nipoti ..). Il processo che ha PID 0 fa alcune inizializzazioni, crea il primo processo (PID 1), poi si mette a "non far niente" in un loop a vuoto (diventa il "processo idle"). Ogni volta che lo scheduler non ha nessun processo da far eseguire, dà il controllo al processo idle.
Il processo 1 fa anch'esso alcune inizializzazioni, poi lancia il programma init, che fa partire il S.O. e tutti i demoni che devono partire inizialmente e che sono scritti nel file /etc/inittab.
La generazione di un nuovo processo da parte del suo "genitore" ("parent process") viene detta "spawning".
Per generare un processo in Unix c'è la chiamata di sistema "**fork**", che genera un nuovo processo duplicando tutte le proprietà del processo genitore, inclusi il codice ed i file aperti. Il processo generato è perciò una sorta di clone del genitore.
All'uscita della fork esiste un nuovo processo, pronto ad eseguire quando lo scheduler vorrà. I due processi genitore e figlio sono "uguali" ma indipendenti, due istanze dello stesso programma, riconoscibili fra loro solo per un diverso PID.
Dunque i programmi più semplici devono contenere il codice sia del processo padre, sia quello di tutti i figli.
Il processo può decidere se eseguire il codice del padre o quello di uno dei figli guardando il valore di ritorno passato dalla fork.

Se si vuole che i due processi padre e figlio non vengano duplicati completamente, alcune risorse possono essere condivise (p.es. file e memoria virtuale).
Se invece si vuole che i due processi abbiano codice diverso si può usare la chiamata a sistema "**exec**" che permette al processo di sostituire i riferimenti al codice del padre con quello indicato come argomento alla funzione. In questo modo i processi padre e figlio divengono diversi anche nel codice. Exec cambia il codice del processo e poi lo esegue.
Per sospendersi fino a che il processo figlio non ha concluso la sua esecuzione il processo padre usa "**wait**".
Per concludere un processo si usa "**exit**".
Per scambiare messaggi fra processi e per sincronizzarli si usa la funzione "**signal**", che, come dice il nome, spedisce un segnale al processo.
Ogni processo che riceve un segnale è obbligato ad elaborarlo, pena la morte. Infatti se una signal non viene recepita da un processo esso viene terminato dal S.O. .
Per elaborare una signal ci dev'essere nel programma una parte di codice che viene fatta eseguire automaticamente al suo arrivo. Se la signal non è interessante per il processo esso la deve esplicitamente ignorare.
Alcune signal sono generate dal sistema in caso di errore:
- errore di segmentazione: accesso alla memoria al di fuori dello spazio d'indirizzi del processo
- tentativo di eseguire un'istruzione di macchina privilegiata
Ogni processo che termina deve spedisce al processo padre un segnale

Altre chiamate di sistema che fanno riferimento ai processi:
**nice**: cambia la priorità del processo, la può eseguire solo il superuser (root)
"**pause**", "**alarm**": in combinazione con signal permettono di sospendere un processo per un determinato tempo.

Context switch in Unix
Avviene quando:
- giunge una richiesta d'interruzione
- il processo che esegue fa una chiamata di sistema
Viene salvato lo stato del processo corrente, cioè il contenuto di tutti i registri, incluso il program counter, nel descrittore di processo.

L'architettura di alcune CPU prevede la presenza di diversi set di registri duplicati in modo, che per passare da un processo al S.O. basti cambiare il set utilizzato, il contesto di un processo interrotto rimane intatto in uno dei set di registri, mentre il S.O. usa gli altri. In questo modo il cambio di contesto fra processo e S.O., e viceversa, richiede solo di cambiare il set di registri che si usa, senza accedere alla memoria.

Il nome del descrittore di processo nel kernel di Linux è task_struct. Il vettore "task" contiene i puntatori a ciascuno dei descrittori di tutti i processi che esistono in un istante.
Il descrittore di processo ha molte decine di campi, molti dei quali sono puntatori (vedi in Appendice A2 la struttura task_struct).
Inoltre esiste una lista bidirezionale che collega tutti i processi esistenti.