Sistemi Operativi Modulo I

Primo canale (A-L) e Teledidattica

A.A. 2021/2022

Corso di Laurea in Informatica

I Processi

Igor Melatti

Sapienza Università di Roma
Dipartimento di Informatica

–

Requisiti di un SO

- *Compito fondamentale: Gestione dei processi*
  - ovvero: delle diverse computazioni che si vuol eseguire su un sistema computerizzato
- Il sistema operativo deve:
  - permettere l'esecuzione alternata (*interleaving*) di processi multipli
  - assegnare le risorse ai processi, e proteggere dagli altri processi le risorse assegnate ad un processo
  - permettere ai processi di scambiarsi informazioni
  - permettere la sincronizzazione tra processi
- Nel seguito, ci concentriamo su sistemi operativi moderni, in grado di gestire l'esecuzione di più processi contemporaneamente (*multiprogramming*)

Che cos'è un *processo*?

- *Un programma in esecuzione*
- Un'istanza di un programma in esecuzione su un computer
- L'entità che può essere assegnata ad un processore per l'esecuzione
- Un'unità di attività caratterizzata dall'esecuzione di una sequenza di istruzioni, da uno stato corrente, e da un insieme associato di risorse
- Un processo è composto da:
  - codice (anche condiviso): le istruzioni da eseguire
  - un insieme di dati
  - un numero di attributi che descrivono lo stato del processo

Processi, Esecuzioni e Programmi

- Per adesso, "processo in esecuzione" vuol dire "un utente ha richiesto l'esecuzione di un programma, che ancora non è terminato"
- Vedremo che questo non significa necessariamente che il processo sia in esecuzione su un processore
  - o meglio, non è detto che, fissato un istante tra la richiesta della sua esecuzione e la sua terminazione, esso sia in esecuzione su un processore
  - decidere se mandare in esecuzione un processo su un processore è uno dei compiti fondamentali di un sistema operativo
- Dietro ogni processo c'è un *programma*
  - nei sistemi operativi moderni, è solitamente memorizzato su archiviazione di massa, ad esempio un disco rigido
  - possono far eccezione i processi creati dal sistema operativo stesso
  - solo eseguendo un programma si può creare un processo
  - eseguendo un programma si crea *almeno* un processo

Processi, Esecuzioni e Programmi

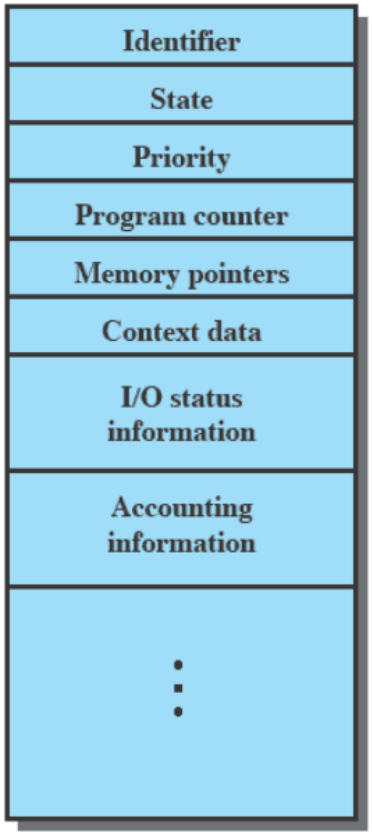
- Un processo ha 3 macrofasi: creazione, esecuzione, terminazione
  - la terminazione può essere prevista
    - es1: un programma deve leggere numeri, ordinarli e scrivere l'output riordinato; alla fine dell'ultimo passo, il processo è terminato
    - es2: word processor (basato su eventi); se l'utente clicca sulla X della finestra, il processo è terminato
    - se l'utente non lo chiude esplicitamente, potrebbe anche rimanere in esecuzione per sempre
  - oppure non prevista
    - ad esempio, il processo esegue un'operazione non consentita: viene attivato automaticamente un interrupt o un'eccezione, che può portare alla chiusura forzata del processo *da parte del sistema operativo*
    - ad esempio, il programma che ordina i numeri cerca di leggere della memoria non assegnata a lui
    - dichiara un array da 100 numeri, e cerca di leggere il 101-esimo...

Elementi di un Processo

- Finché il processo è in esecuzione, ad esso sono associati un certo insieme di informazioni, tra le quali:
  - identificatore
  - stato (*running*, ma non solo...)
  - priorità
  - *hardware context*: valore corrente dei registri della CPU
    - include il program counter
  - puntatori alla memoria (che definisce l'*immagine* del processo)
  - informazioni sullo stato dell'input/output
  - informazioni di accounting (quale utente lo esegue)

Process Control Block

- Contiene gli elementi del processo
- Creato e gestito dal sistema operativo
- Permette al SO di gestire più processi contemporaneamente
- Contiene sufficienti info per bloccare un programma in esecuzione e farlo riprendere più tardi dallo stesso punto in cui si trovava

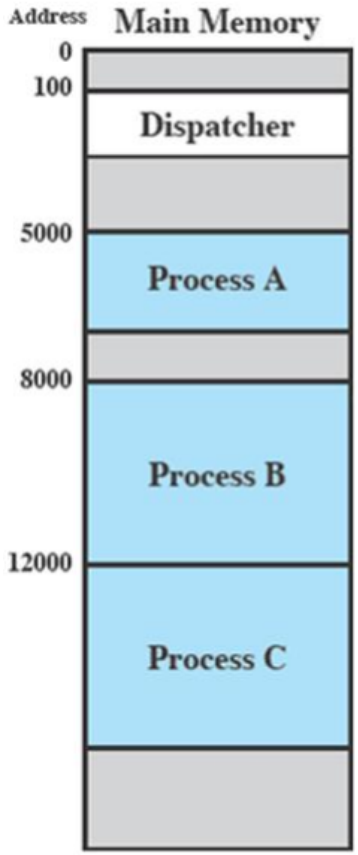


Traccia di un Processo

- Il comportamento di un particolare processo è caratterizzato dalla sequenza delle istruzioni che vengono eseguite
- Questa sequenza è detta trace (**traccia**)
- Il dispatcher è un piccolo programma che sospende un processo per farne andare in esecuzione un altro
  - ovviamente, il dispatcher fa parte del sistema operativo
  - è sempre in memoria, anche per i sistemi a microkernel

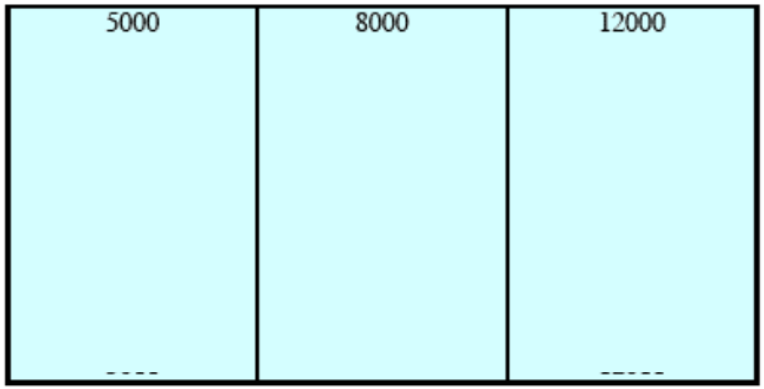
Esecuzione di un Processo

- Si considerino 3 processi in esecuzione
- Sono tutti in memoria (più il dispatcher)
- Si ignori per ora la memoria virtuale



La Traccia dal Punto di Vista del Processo

Ogni processo viene eseguito senza interruzioni fino al termine



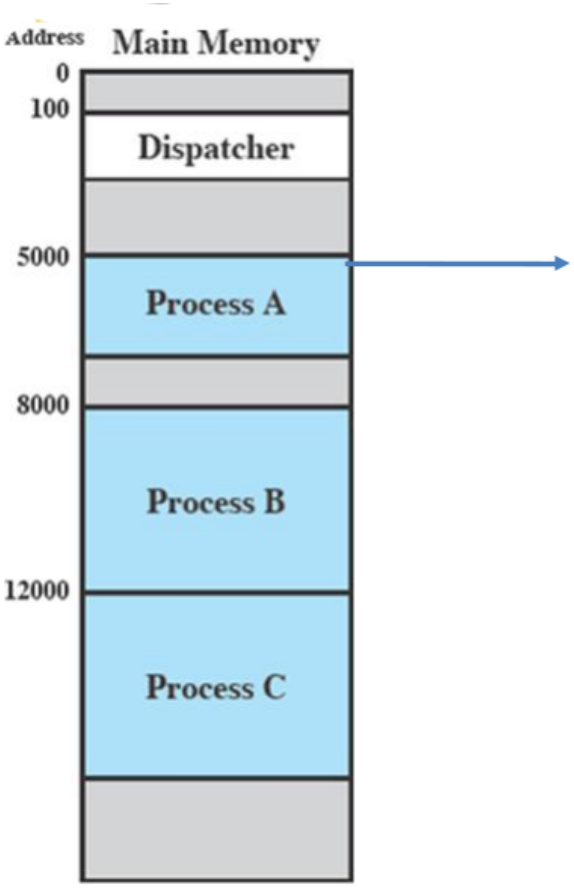
(a) Trace of Process A (b) Trace of Process B (c) Trace of Process C

La Traccia dal Punto di Vista del Processo

Ogni processo viene eseguito senza interruzioni fino al termine

(a) Trace of Process A	(b) Trace of Process B	(c) Trace of Process C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

La Traccia dal Punto di Vista del Processore



1	5000
2	5001
3	5002
4	5003
5	5004
6	5005
----------	Timeout
7	100
8	101
9	102
10	103
11	104
12	105
13	8000
14	8001
15	8002
16	8003
----------	I/O Request
17	100
18	101
19	102
20	103
21	104
22	105
23	12000
24	12001
25	12002
26	12003
27	12004
28	12005
----------	Timeout
29	100
30	101
31	102
32	103
33	104
34	105
35	5006
36	5007
37	5008
38	5009
39	5010
40	5011
----------	Timeout
41	100
42	101
43	102
44	103
45	104
46	105
47	12006
48	12007
49	12008
50	12009
51	12010
52	12011
----------	Timeout

100 = Starting address of dispatcher program

Modello dei Processi a 2 Stati

- Un processo potrebbe essere in uno di due stati
  - In esecuzione (per davvero!)
  - Non in esecuzione (ma è comunque un processo "attivo")
- Troppo semplice

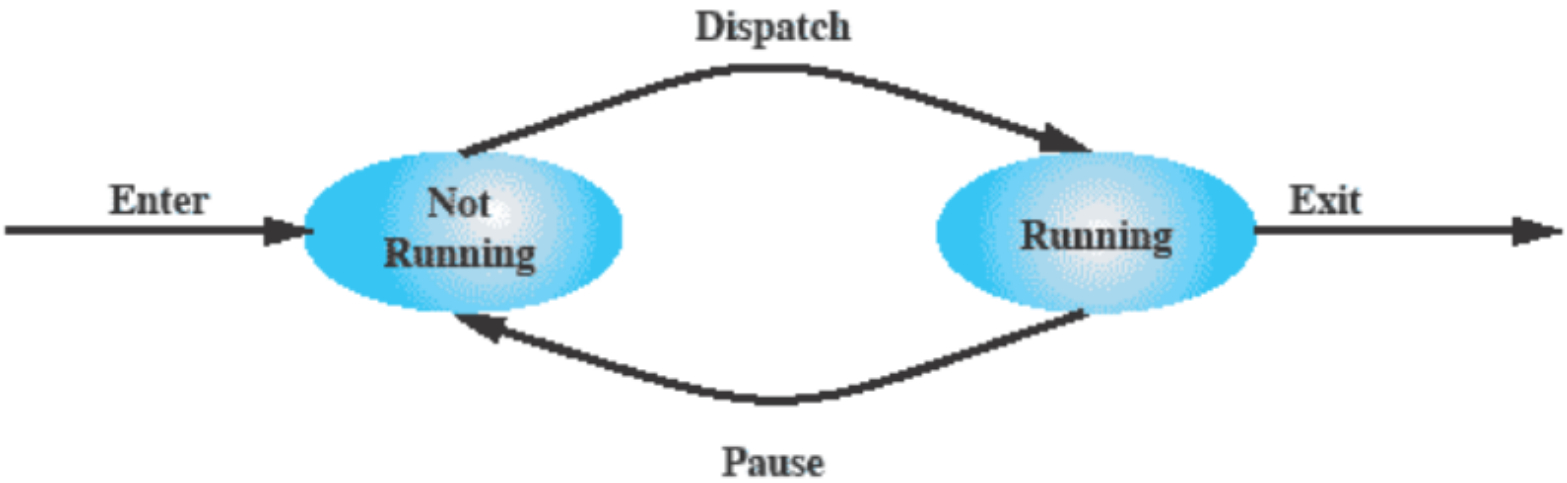


Diagramma a Coda

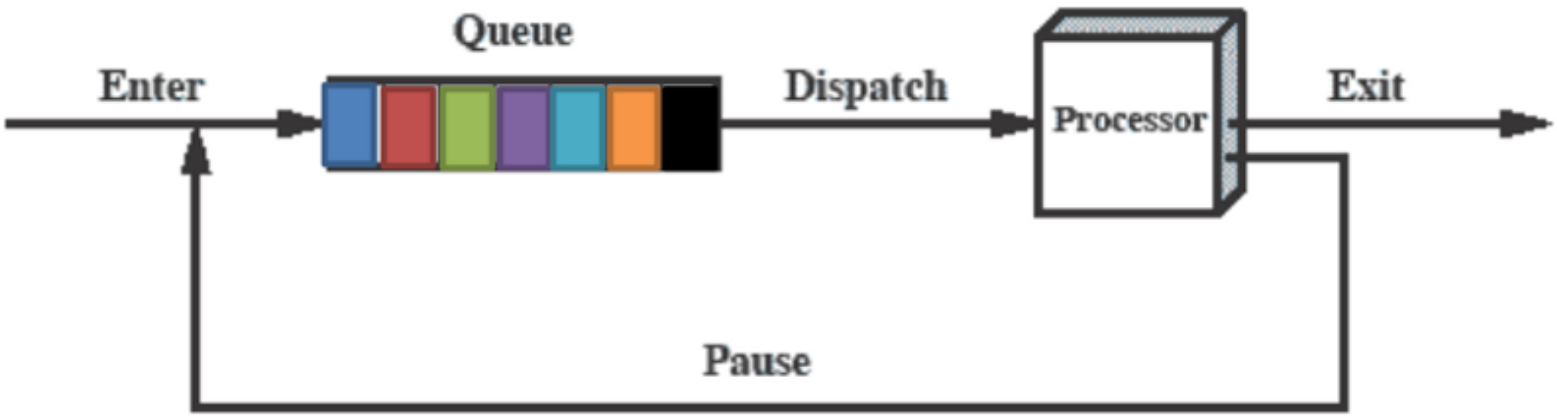
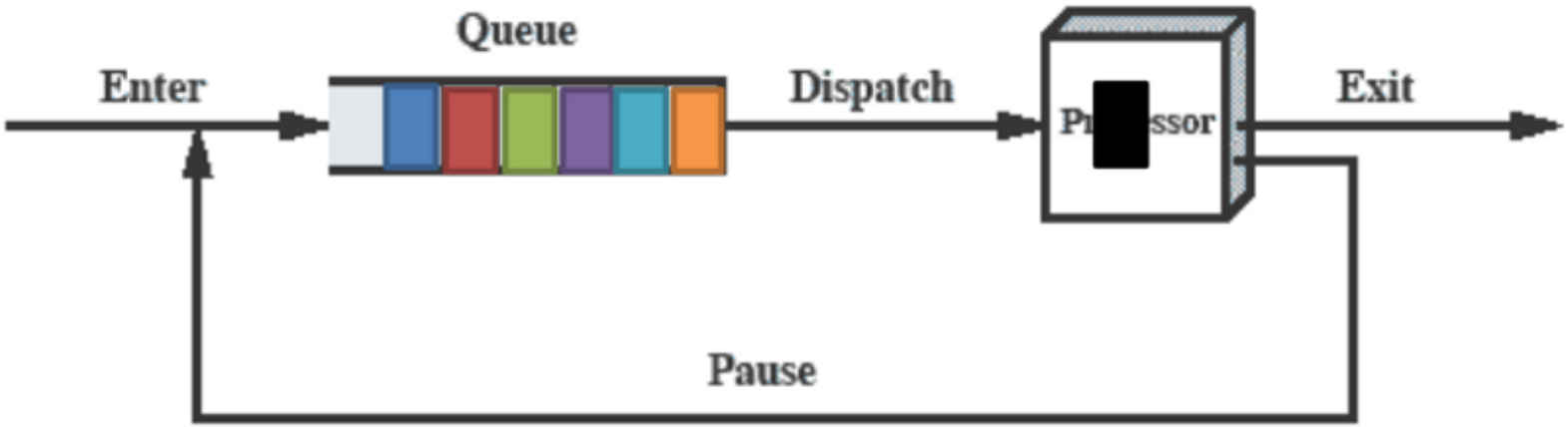


Diagramma a Coda



I processi vengono mossi dal dispatcher del SO dalla CPU alla coda e viceversa, finché un processo non viene completato

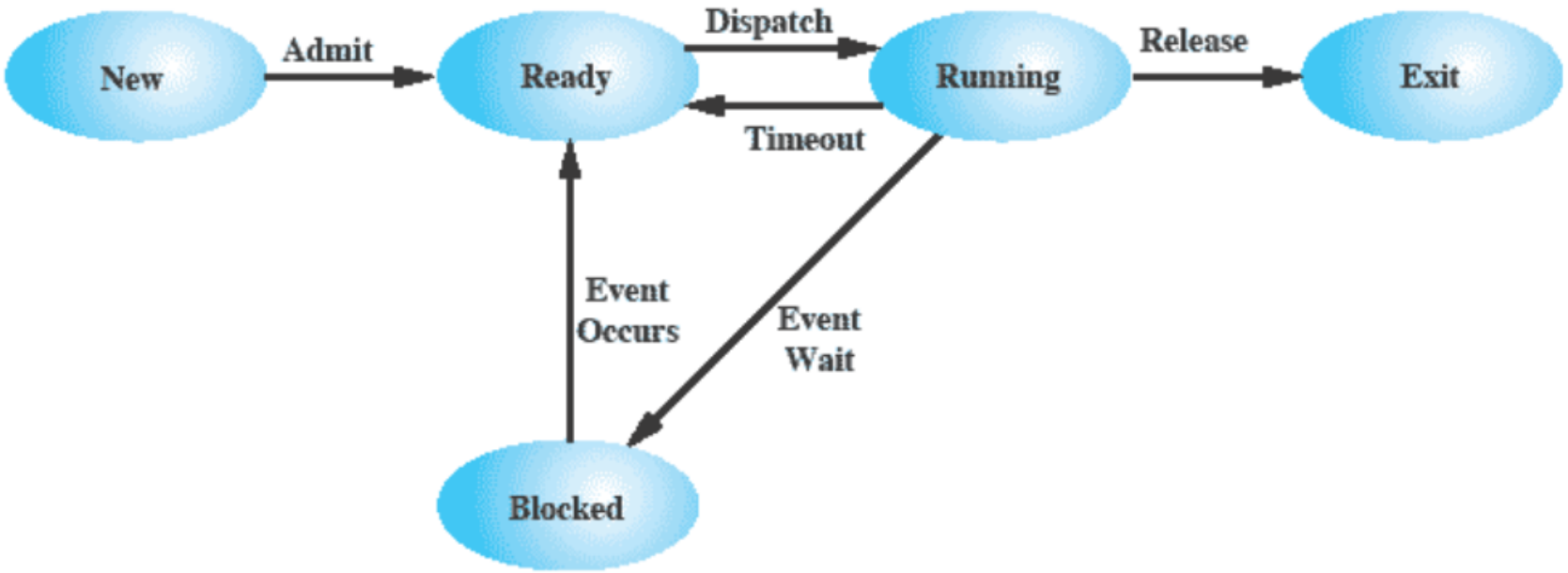
Creazione di Processi

- In un certo istante, in un sistema operativo ci sono $n \geq 1$ processi
  - come minimo c'è un'interfaccia o grafica (GUI) o testuale (CLI) in attesa di comandi dell'utente
  - ovviamente, c'è (almeno) un processo che gestisce tali interfacce
- Se l'utente dà un comando, quasi sempre si crea un nuovo processo
  - es.: click col mouse su un'icona, o scrittura di un comando
- Process spawning: un processo crea un altro processo
  - il *processo padre* è il processo originale, ovvero quello che crea
    - nell'esempio di sopra, l'interfaccia grafica
  - il *processo figlio* è il nuovo processo appena creato
  - il vecchio processo resta in esecuzione, quindi si passa da n processi ad $n + 1$ processi

Terminazione di un Processo

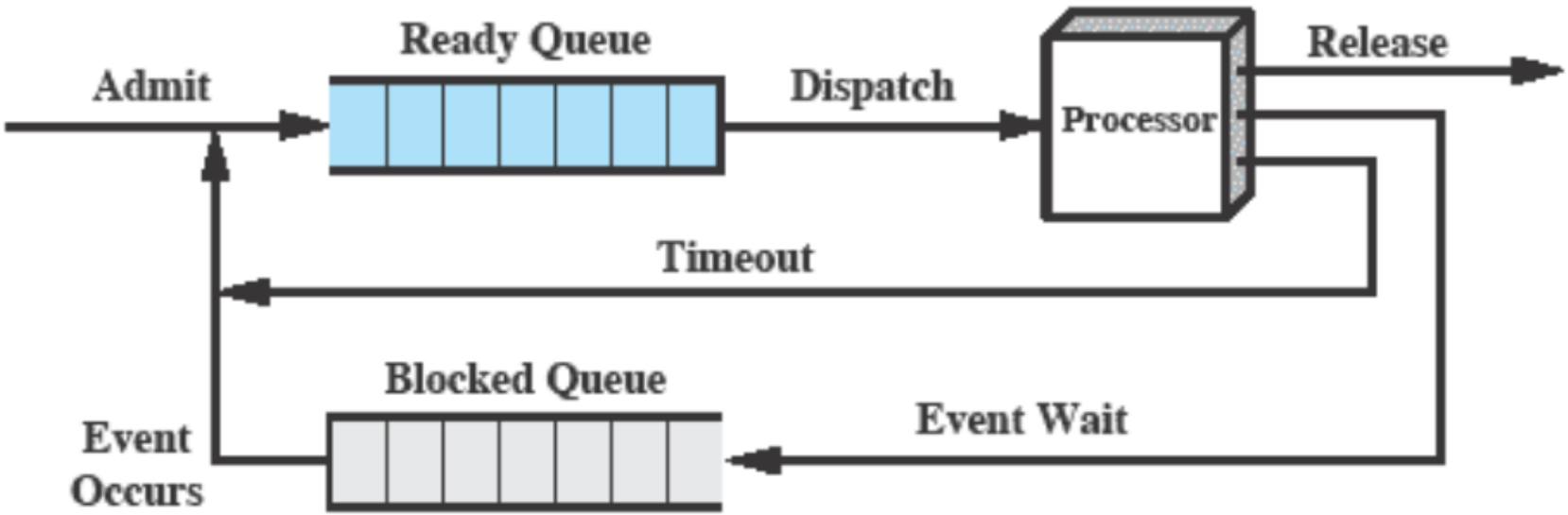
- Normale Completamento
  - istruzione macchina HALT, che generi un'interruzione per il SO
  - in linguaggi ad alto livello, l'istruzione HALT è invocata da una system call come `exit`
  - inserita dai compilatori dopo l'ultima istruzione del `main` di un programma...
- Uccisioni:
  - dal SO, per errori come:
    - memoria non disponibile
    - errore di protezione
    - errore fatale a livello di istruzione (divisione per zero...)
    - operazione di I/O fallita
  - dall'utente (es.: X sulla finestra)
  - da un altro processo (invio segnale, ad es. da terminale)
- Si passa da $n \geq 2$ processi ad $n - 1$
  - c'è sempre un processo "master" che non può essere terminato, a meno di non spegnere il computer

Modello dei Processi a 5 Stati

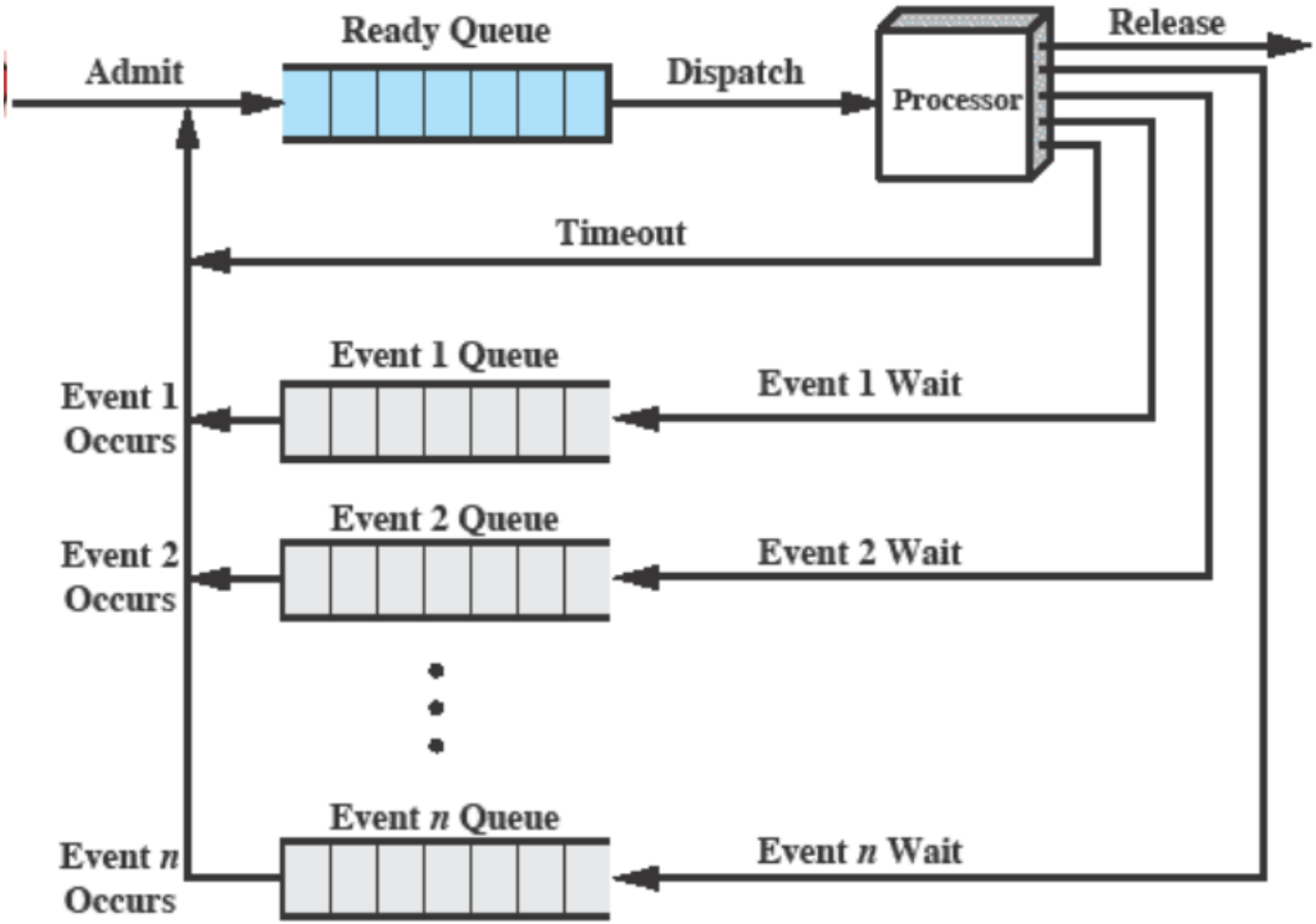


Si può andare anche da ready o blocked ad exit (un processo ne killa un altro)

Usando Due Code



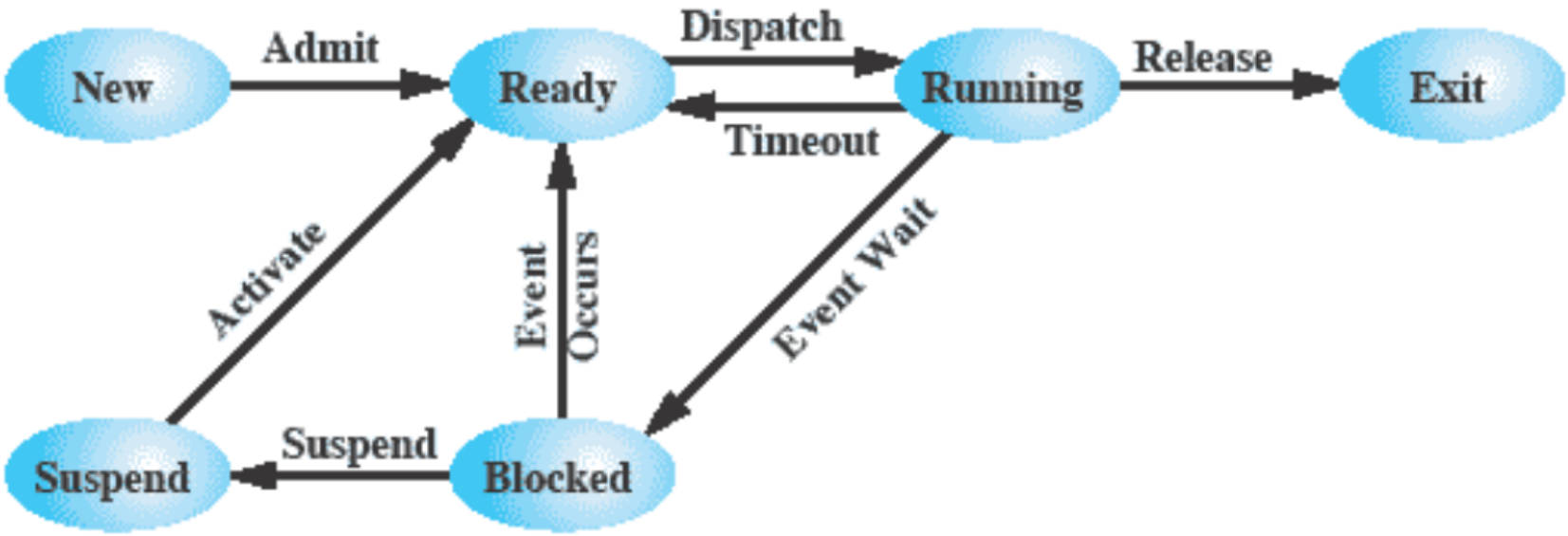
Molteplici Code Bloccanti



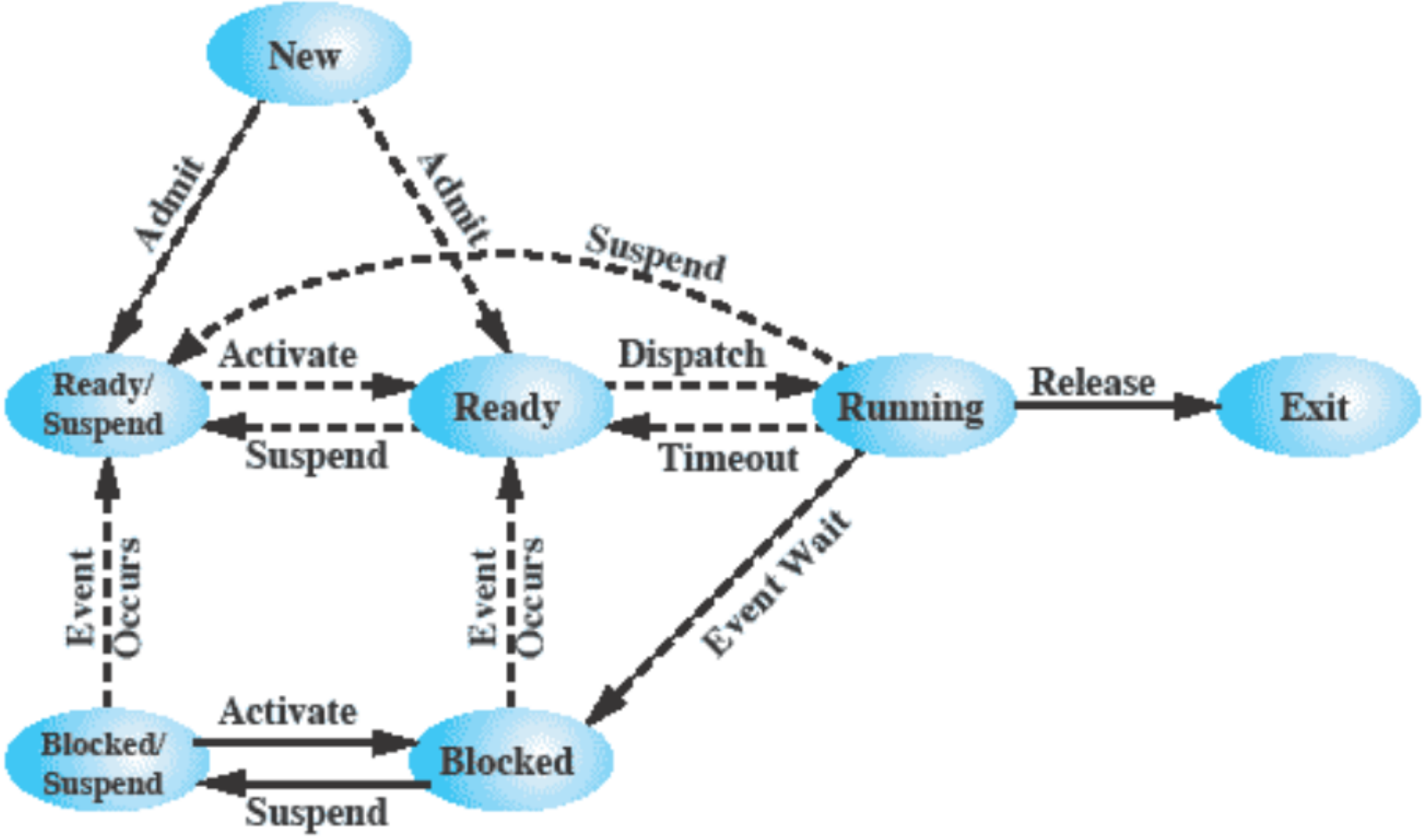
Processi Sospesi

- Il processore è più veloce dell'I/O, quindi tutti i processi attualmente in memoria potrebbero essere in attesa di I/O
  - si spostano (swappano) su disco, così da liberare memoria e da non lasciare il processore inoperoso
- Lo stato "blocked" diventa "suspended" quando il processo è swappato su disco
- Due nuovi stati
  - blocked/suspend (swappato mentre era bloccato)
  - ready/suspend (swappato mentre non era bloccato)

Uno Stato Suspended



Due Stati Suspended

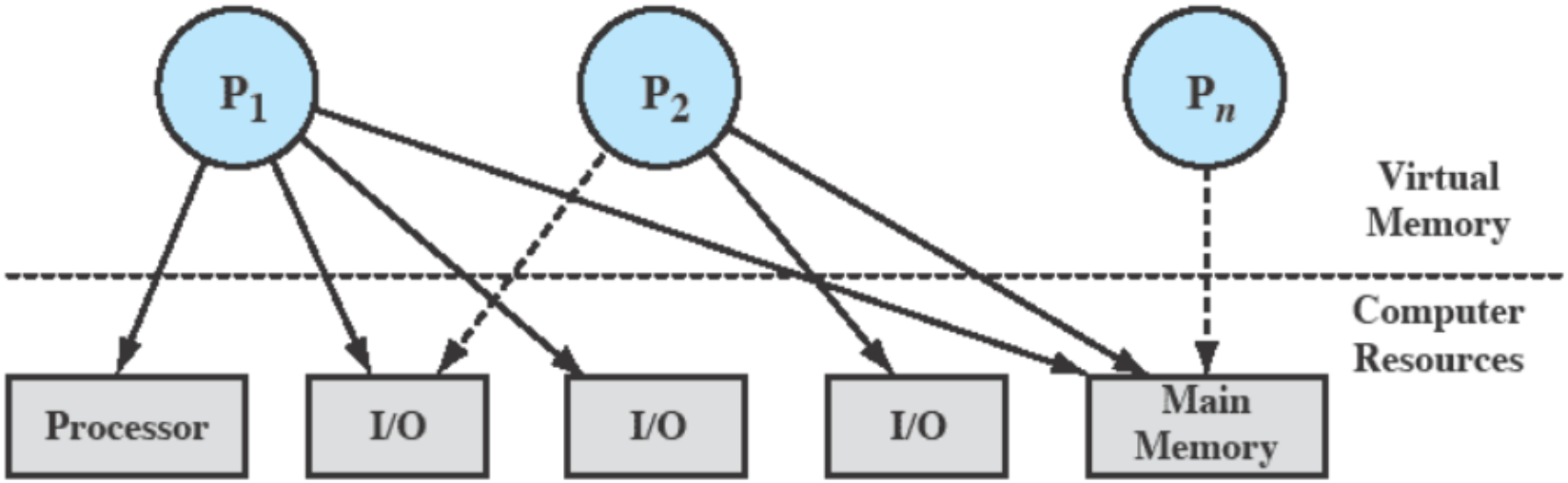


Si può andare anche direttamente ad exit da un qualsiasi stato diverso da new (un processo ne killa un altro)

Motivi per Sospendere un Processo

Motivo	**Commento**
Swapping	Il SO ha bisogno di rilasciare abbastanza memoria da portarci dentro un processo ready
Interno al SO	Il SO sospetta che il processo stia causando problemi
Richiesta utente interattiva	Ad esempio: debugging, o correlato all'uso di una risorsa
Periodicità	Il processo viene eseguito periodicamente (p.e. per monitoraggio di sistema o per accounting) e può venire sospeso in attesa della prossima esecuzione
Richiesta del padre	Il padre potrebbe voler sospendere l'esecuzione di un figlio per esaminarlo o modificarlo, o per coordinare l'attività tra più figli

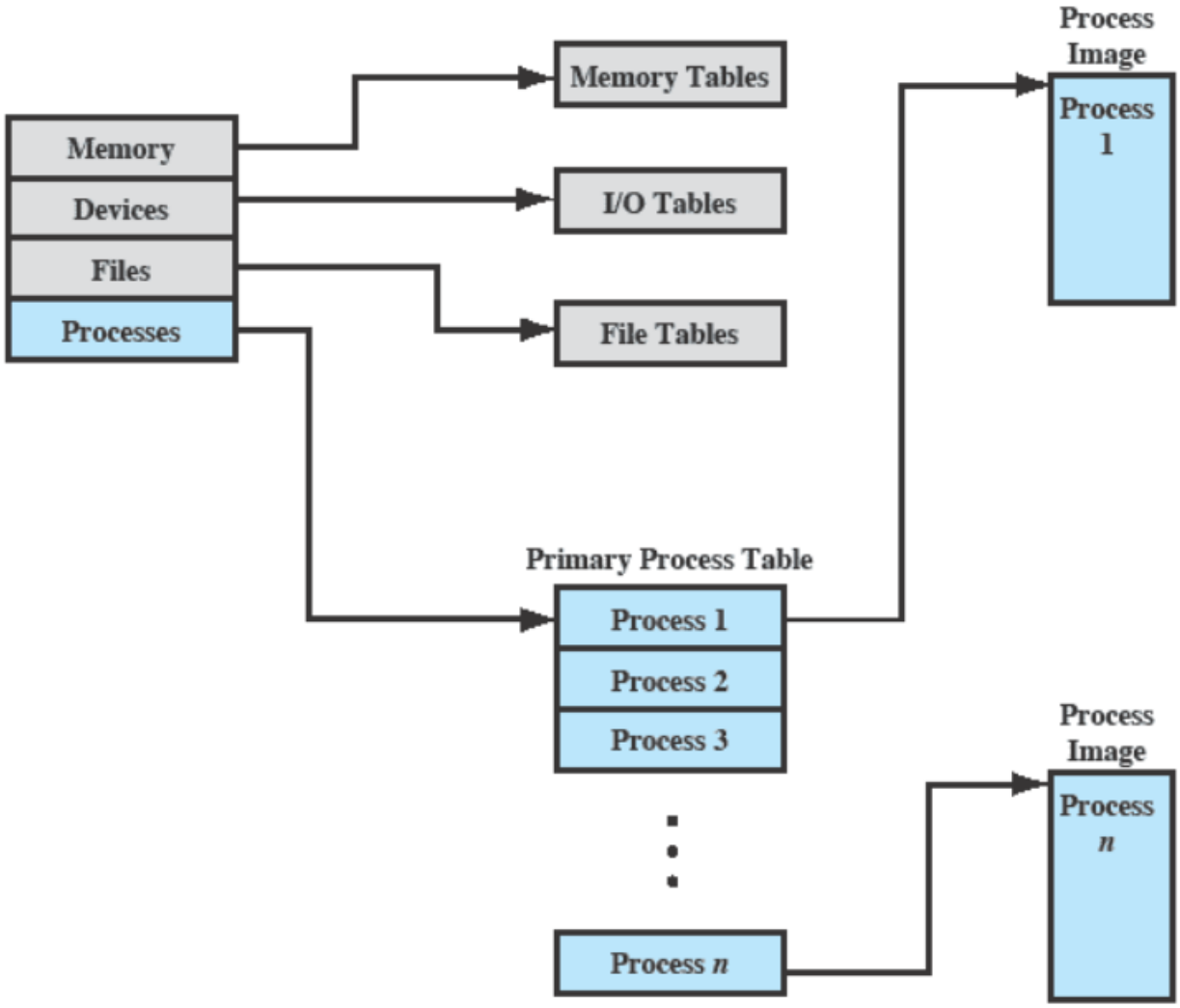
Processi e Risorse



Strutture di Controllo del SO

- In fondo, il SO è l'entità che gestisce l'uso delle risorse di sistema da parte dei processi
  - In primis, il processore
- Poiché il SO deve gestire sia i processi che le risorse, deve conoscere lo stato di ogni processo e di ogni risorsa
- A tal proposito, per ogni entità da gestire il SO costruisce e mantiene una o più tabelle

Tabelle di controllo del SO



Ovviamente, ci sono molti riferimenti incrociati

Tabelle di Memoria

- Le tabelle di memoria sono usate per gestire sia la memoria principale che quella secondaria
  - quella secondaria serve per la memoria virtuale, ci torneremo
- Devono comprendere le seguenti info:
  - allocazione di memoria principale da parte dei processi
  - allocazione di memoria secondaria da parte dei processi
  - attributi di protezione per l'accesso a zone di memoria condivisa
  - informazioni per gestire la memoria virtuale

Tabelle per l'I/O

- Usate dal SO per gestire i dispositivi e i canali di I/O
- Il SO deve sapere:
  - se il dispositivo è disponibile o già assegnato
  - lo stato dell'operazione di I/O
  - la locazione in memoria principale usata come sorgente o destinazione del trasferimento di I/O

Tabelle dei File

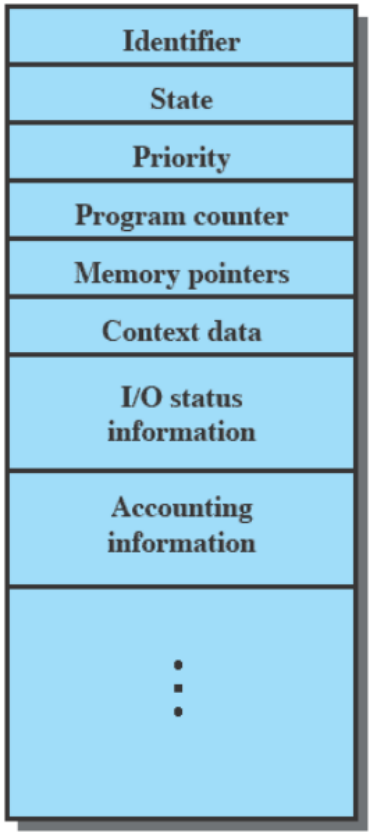
- Queste tabelle forniscono informazione su:
  - esistenza di files
  - locazioni in memoria secondaria
  - stato corrente
  - altri attributi
- Memorizzate parte su disco e parte in RAM

Tabelle dei Processi

- Per gestire i processi il SO deve conoscerne i dettagli:
  - stato corrente
  - identificatore
  - locazione in memoria
  - etc
- Blocco di controllo del processo (Process Control Block, PCB)
  - le informazioni in esso contenute sono spesso chiamate *attributi* del processo
- Si dice process image (immagine del processo) l'insieme di programma sorgente, dati, stack delle chiamate e PCB
  - eseguire un'istruzione cambia l'immagine: per esempio, modificando un registro o una cella di memoria
  - unica possibile eccezione: un'istruzione di jump all'istruzione stessa

Attributi dei Processi

- Le informazioni in ciascun blocco di controllo possono essere raggruppate in 3 categorie:
  - identificazione
  - stato
  - controllo



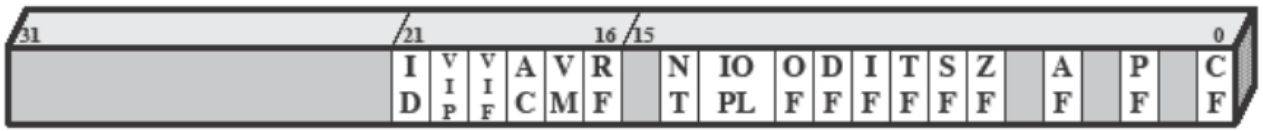
Come si Identifica un Processo

- Ad ogni processo è assegnato un numero identificativo, quindi unico: il PID (Process IDentifier)
- Molte tabelle del SO usano i PID per realizzare collegamenti tra le varie tabelle e la tabella dei processi
  - ad esempio, la tabella dei dispositivi I/O deve mantenere, per ogni dispositivo, quale processo lo sta usando
  - basta mettere il PID, e implicitamente si può accedere alle informazioni sul processo corrispondente

Lo Stato del Processore

- Da non confondere con lo stato, o meglio la modalità, del processo (ready, blocked, ...)
- È dato dai contenuti dei registri del processore stesso
  - registri visibili all'utente
  - registri di controllo e di stato
  - puntatori allo stack
- Parola dello stato di programma (Program status word, PSW)
  - contiene informazioni di stato
  - esempio: il registro EFLAGS sui processori Pentium

EFLAGS nel Pentium 2



ID = Identification flag
VIP = Virtual interrupt pending
VIF = Virtual interrupt flag
AC = Alignment check
VM = Virtual 8086 mode
RF = Resume flag
NT = Nested task flag
IOPL = I/O privilege level
OF = Overflow flag

DF = Direction flag
IF = Interrupt enable flag
TF = Trap flag
SF = Sign flag
ZF = Zero flag
AF = Auxiliary carry flag
PF = Parity flag
CF = Carry flag

Control Block del Processo

- Contiene informazioni di cui il SO ha bisogno per controllare e coordinare i vari processi attivi
- Identificatori:
  - del processo (PID)
  - del processo padre (Parent PID, o PPID)
  - dell'utente proprietario
- Informazioni sullo stato del processore:
  - registri utente (accessibili in linguaggio macchina/assembler)
  - program counter
  - stack pointer
  - registri di stato: risultati di operazioni aritmetico/logiche, modalità di esecuzione, interrupt abilitati/disabilitati

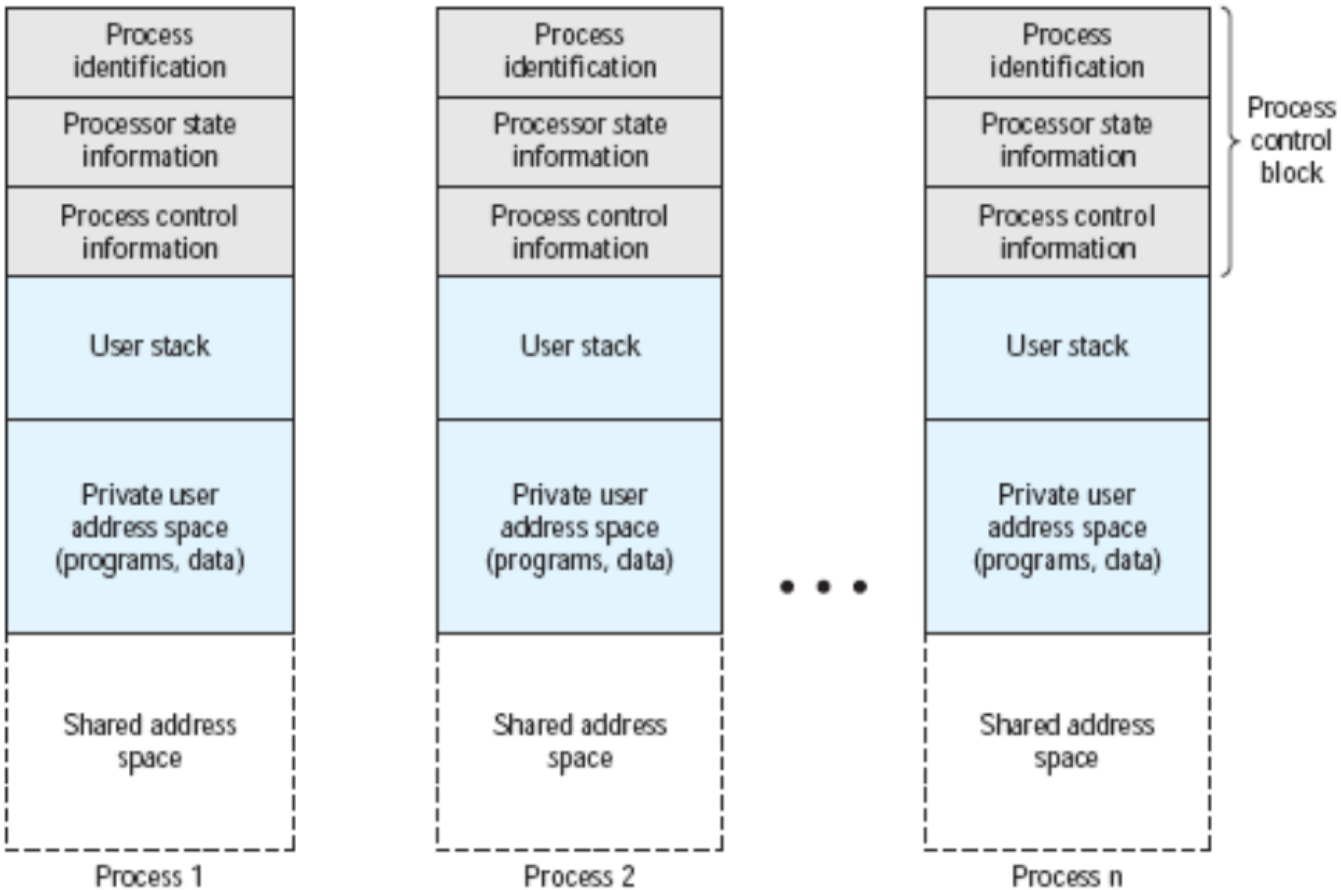
Control Block del Processo

- Informazioni per il controllo del processo:
  - stato del processo (ready, suspended, blocked, ...)
  - priorità
  - informazioni sullo scheduling (ad es.: per quanto tempo è stato in esecuzione l'ultima volta)
  - l'evento da attendere per tornare ad essere ready, se attualmente in attesa
- Supporto per strutture dati
  - puntatori ad altri processi
  - per mantenere liste concatenate di processi nei casi in cui siano necessarie (es., code di processi per qualche risorsa)
- Comunicazioni tra processi
  - flag, segnali, messaggi per supportare comunicazioni tra processi

Control Block del Processo

- Permessi speciali
  - non tutti i processi possono accedere a tutto
- Gestione della memoria
  - puntatori ad aree di memoria che gestiscono l'uso della memoria virtuale
  - es: pagine virtuali attualmente in uso
- Uso delle risorse
  - file aperti
  - uso di risorse (compreso il processore) fino ad ora

Immagini dei Processi in Memoria Virtuale



Control Block del Processo

- La struttura dati più importante di un sistema operativo
  - definisce lo stato del SO stesso
- Richiede protezioni
  - una funzione scritta male potrebbe danneggiare il blocco, rendendo il SO incapace di gestire i processi
  - ogni cambiamento nella progettazione del blocco ha effetti su molti moduli del SO

Modalità di Esecuzione

- La maggior parte dei processori supporta almeno due modalità di esecuzione
- Modo sistema
  - pieno controllo: ad es., si possono eseguire istruzioni macchina che bloccano gli interrupt
  - si può accedere a qualsiasi locazione di RAM
  - per il kernel
- Modo utente
  - molte operazioni sono vietate
  - per i programmi utente
- Pentium ne ha addirittura 4
  - ma Linux usa solo il primo (ristretto, *modalità utente*) e l'ultimo (senza limitazioni, *modalità sistema o kernel*)

Kernel Mode

- Per le operazioni effettuate dal kernel
- Gestione dei processi (tramite PCB)
  - creazione e terminazione
  - pianificazione di lungo, medio e breve termine (*scheduling* e *dispatching*)
  - avvicendamento (*process switching*)
  - sincronizzazione e comunicazione
- Gestione della memoria principale
  - allocazione di spazio per i processi
  - gestione della memoria virtuale
- Gestione dell'I/O
  - gestione dei buffer e delle cache per l'I/O
  - assegnazione risorse I/O ai processi
- Funzioni di supporto
  - Gestione interrupt/eccezioni, accounting, monitoraggio

Da User Mode a Kernel Mode e Ritorno

- Si basa su un'idea semplice: un processo utente inizia sempre in modalità utente
- Cambia e si porta a modalità sistema in seguito ad un *interrupt*
  - la prima cosa che fa l'hardware, prima di cominciare a fare le copie per invocare l'handler, è cambiare la modalità
  - da utente a sistema
- Questo permette di eseguire l'interrupt handler in modalità kernel
  - ma l'interrupt handler è dentro i(l kernel de)l sistema operativo, quindi è ok che succeda
- L'ultima istruzione dell'interrupt handler, prima di restituire il controllo al processo utente, dovrà fare nuovamente lo switch a modalità utente

Da User Mode a Kernel Mode e Ritorno

- Con questo schema, un processo utente può cambiare la modalità a sé stesso, ma *solo per eseguire software di sistema*
- Codice eseguito per conto dello stesso processo interrotto, che lo ha esplicitamente voluto
  - *system call* (vedere slide successiva)
  - in risposta ad una sua precedente richiesta di I/O, o comunque di risorse
- Codice eseguito per conto dello stesso processo interrotto, che non lo ha esplicitamente voluto
  - errore fatale (*abort*): il processo spesso viene terminato
  - errore non fatale (*fault*): trasparente rispetto al processo
- Codice eseguito per conto di un qualche altro processo
  - in risposta ad una sua precedente richiesta di I/O, o comunque di risorse

System call sui Pentium

- Pezzo di codice che:
  - 1) prepara gli argomenti della chiamata mettendoli in opportuni registri
  - 1bis) tra tali argomenti c'è un numero che identifica la system call (*system call number*)
  - 2) esegue l'istruzione `int 0x80`, che appunto solleva un interrupt (in realtà, un'eccezione)
  - 2) in alternativa, dal Pentium 2 in poi, può eseguire l'istruzione `sysenter`, che omette alcuni controlli inutili
- Da notare che anche il creare un nuovo processo è una system call: in Linux, *fork* (oppure *clone*, più generale)
  - l'handler di questa system call verrà ovviamente eseguito in modalità kernel
  - può quindi modificare la lista dei PCB e fare quanto descritto di seguito

Creazione di un Processo

Per creare un processo, il SO deve:

- Assegnargli un PID unico
- Allocargli spazio in memoria principale
- Inizializzare il process control block
- Inserire il processo nella giusta coda
  - ad es., ready oppure ready/suspended
- Creare o espandere altre strutture dati
  - ad es., quelle per l'accounting

Switching tra Processi

Pone svariati problemi

- Quali eventi determinano uno switch?
- Occorre distinguere tra switch di modalità di un processo e switching di processi
  - switch di modalità: da modalità utente a sistema e viceversa
    - con il meccanismo visto prima, basato sugli interrupt
  - switching tra processi: per qualche motivo l'attuale processo non deve più usare il processore, che va concesso invece ad un altro processo
- Cosa deve fare il SO per tenere aggiornate tutte le strutture dati in seguito ad uno switch tra processi?

Quando Effettuare uno Switch

Quando il SO si riprende il controllo, togliendolo al processo attualmente in esecuzione. Questo può succedere perché:

Meccanismo	**Causa**	**Uso**
Interruzione	Esterna all'esecuzione dell'istruzione corrente	Reazione ad un evento esterno asincrono; include i quanti di tempo per lo scheduler
Eccezione	Associata all'esecuzione dell'istruzione corrente	Gestione di un errore sincrono
Chiamata al SO	Richiesta esplicita	Chiamata a funzione di sistema (caso particolare di eccezione)

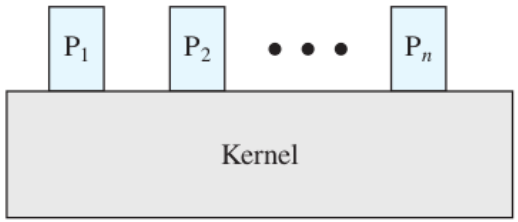
Process Switch: Passaggi

- Tutto in kernel mode
  1. Salvare il contesto del programma (registri e PC)
  2. Aggiornare il process control block, attualmente in running
  3. Spostare il process control block nella coda appropriata: ready; blocked; ready/suspend
  4. Scegliere un altro processo da eseguire
  5. Aggiornare il process control block del processo selezionato
  6. Aggiornare le strutture dati per la gestione della memoria
  7. Ripristinare il contesto del processo selezionato
- Se si passa dal processo A al processo B: i primi 3 passi sono per A, gli ultimi 3 passi sono per B, il 4 è al di fuori

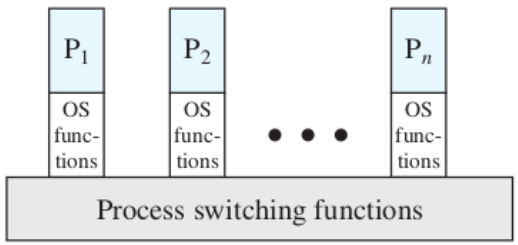
Il SO è un Processo?

- Il SO è solo un insieme di programmi, ed è eseguito dal processore come ogni altro programma
- Semplicemente, ogni tanto (in realtà, molto spesso) lascia che altri programmi vadano in esecuzione, per poi riprendere il controllo tramite interrupt
- È quindi lui stesso un processo?
- Se sì, come viene controllato?

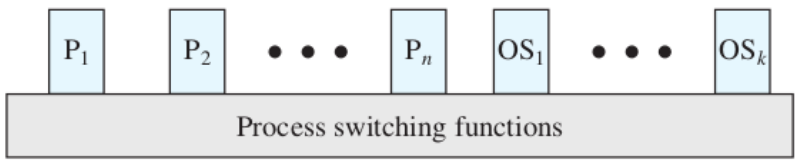
Esecuzione del SO



(a) Separate kernel



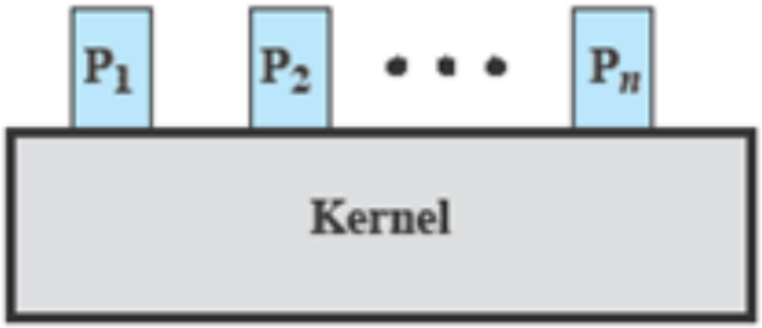
(b) OS functions execute within user processes



(c) OS functions execute as separate processes

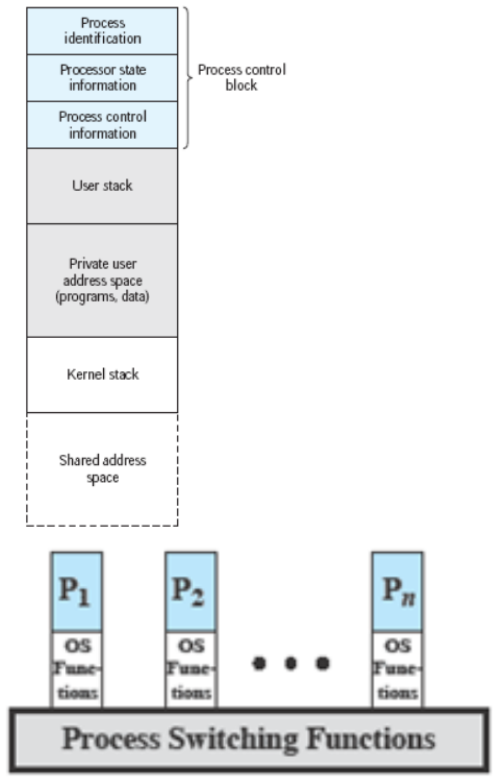
Il Kernel non è un Processo

- Kernel eseguito al di fuori dei processi
- Il concetto di processo si applica solo ai programmi utente
- Il SO è eseguito come un'entità separata, con privilegi più elevati
- Ha una sua zona di memoria dedicata sia per i dati che per il codice sorgente che per lo stack



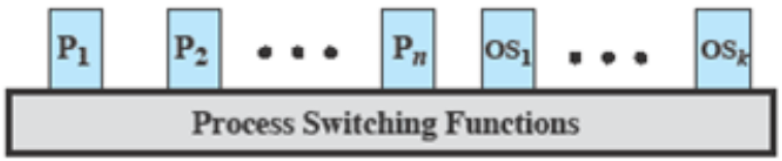
Esecuzione *all'interno* dei Processi Utente

- Il SO viene eseguito nel contesto di un processo utente (cambia solo la modalità di esecuzione)
- Non c'è bisogno di un process switch per eseguire una funzione del SO, solo del mode switch
- Comunque, stack delle chiamate separato; dati e codice macchina condiviso coi processi
- Process switch solo, eventualmente, alla fine, se lo scheduler decide che tocca ad un altro processo



SO è Basato sui Processi

- SO implementato come un insieme di processi di sistema
- Ovviamente, con privilegi più alti
- Partecipano alla competizione per il processore accanto ai processi utente
- Ad es., se un processo effettua una system call, questo crea un nuovo processo (di sistema)
- Lo switch tra processi, però, non è un processo

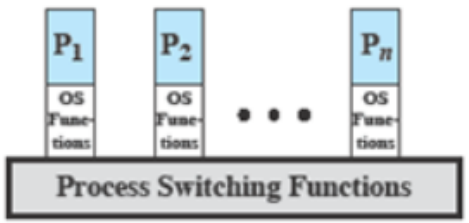


Caso Concreto: Linux

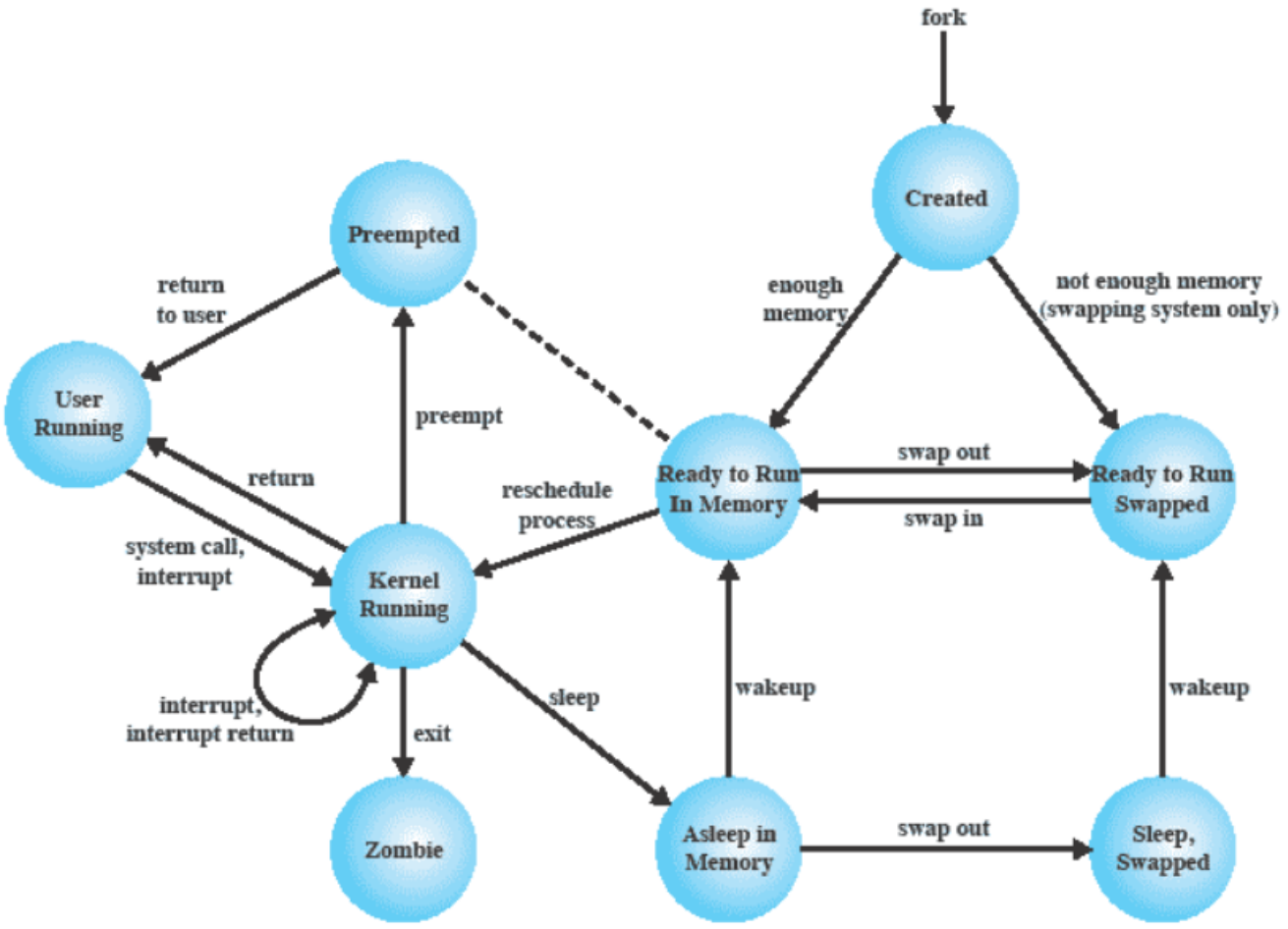
- Via di mezzo tra la seconda e la terza opzione
- Le funzioni del kernel sono per lo più eseguite tramite interrupt, "on behalf of" il processo corrente
  - può quindi succedere, per interrupt asincroni, che la gestione di un interrupt causato da un certo processo sia effettuata durante l'esecuzione di un altro processo
- Ci sono però anche dei processi di sistema (*kernel threads*) che partecipano alla normale competizione del processore, senza essere invocati esplicitamente
  - creati in fase di inizializzazione
  - operazioni tipiche: creare spazio usabile nella memoria principale liberando zone non usate
  - scrivere sui dispositivi di I/O le operazioni bufferizzate in precedenza
  - eseguire operazioni di rete

Unix SVR4 System V Release 4

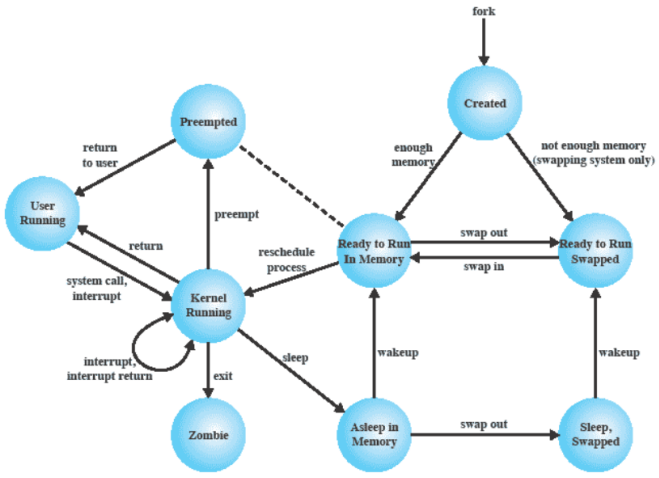
- Usa il modello qui sotto, dove la maggior parte del SO viene eseguito all'interno dei processi utente
- Processi di sistema – solo modo Kernel
- Processi utente
  - modo utente per eseguire programmi ed utilità
  - modo kernel per istruzioni del kernel



UNIX: Transizioni tra Stati dei Processi



UNIX: Transizioni tra Stati dei Processi



- E se un processo non fa mai una chiamata per l'I/O o altra chiamata di sistema, resta in esecuzione per sempre?
- Mancano exit e tutte le transizioni che ci finiscono dentro
- Non interrompibile quando in kernel mode: non va bene per processi real-time

UNIX: Stati dei Processi

User Running in esecuzione in modalità utente

Kernel Running in esecuzione in modalità kernel o sistema

Ready to Run, in Memory può andare in esecuzione non appena il kernel lo seleziona

Asleep in Memory non può essere eseguito finché un qualche evento non si manifesta; il processo è in memoria

Ready to Run, Swapped può andare in esecuzione (non è in attesa di eventi esterni), ma prima dovrà essere portato in memoria

UNIX: Stati dei Processi

Sleeping, Swapped non può essere eseguito finché un qualche evento non si manifesta; il processo non è in memoria primaria

Preempted il kernel ha appena tolto l'uso del processore a questo processo (preemption), per fare un context switch

Created appena creato, ma ancora non pronto all'esecuzione

Zombie terminato, ma resta nelle tabelle dei processi perché il processo che l'ha creato possa prendersi il suo valore di ritorno

Il Processo Unix

- Un insieme di strutture dati forniscono al SO le informazioni per gestire i processi
- Nelle slide che seguono, le informazioni sono raggruppate:
  - per contesto a livello utente
  - per contesto a livello registro
  - per contesto a livello di sistema

Il Processo Unix: Livello Utente

Process text il codice sorgente (in linguaggio macchina) del processo

Process data sezione di dati del processo

- qui vengono mappati i valori delle variabili

User stack stack delle chiamate del processo

- nel fondo, contiene anche gli argomenti con cui il processo è stato invocato

Shared Memory Memoria condivisa con altri processi, usata per le comunicazioni tra processi

Il Processo Unix: Livello Registro

Program counter Indirizzo della prossima istruzione del process text da eseguire

Processor status register registro di stato del processore, relativo a quando è stato swappato l'ultima volta

- il suo formato dipende dall'hardware su cui il SO viene eseguito

Stack pointer puntatore alla cima dello user stack

General purpose registers contenuto dei registri accessibili al programmatore, relativo a quando è stato swappato l'ultima volta

- il suo formato dipende dall'hardware su cui il SO viene eseguito

Il Processo Unix: Livello Sistema

Process table entry puntatore alla tabella di tutti i processi, dove individua quello corrente

- sempre in memoria principale
- PID, modalità del processo, utente proprietario, evento di cui si è in attesa, ...

U area informazioni per il controllo del processo

- informazioni addizionali per quando il kernel viene eseguito da questo processo

Per process region table definisce il mapping tra indirizzi virtuali ed indirizzi fisici (page table)

- indica anche se per questo processo tali indirizzi sono in lettura, scrittura o esecuzione

Kernel stack stack delle chiamate, separato da quello utente, usato per le funzioni da eseguire in modalità sistema

Process Table Entry

Process status	Current state of process.
Pointers	To U area and process memory area (text, data, stack).
Process size	Enables the operating system to know how much space to allocate the process.
User identifiers	The **real user ID** identifies the user who is responsible for the running process. The **effective user ID** may be used by a process to gain temporary privileges associated with a particular program; while that program is being executed as part of the process, the process operates with the effective user ID.
Process identifiers	ID of this process; ID of parent process. These are set up when the process enters the Created state during the fork system call.
Event descriptor	Valid when a process is in a sleeping state; when the event occurs, the process is transferred to a ready-to-run state.
Priority	Used for process scheduling.
Signal	Enumerates signals sent to a process but not yet handled.
Timers	Include process execution time, kernel resource utilization, and user-set timer used to send alarm signal to a process.
P_link	Pointer to the next link in the ready queue (valid if process is ready to execute).
Memory status	Indicates whether process image is in main memory or swapped out. If it is in memory, this field also indicates whether it may be swapped out or is temporarily locked into main memory.

U-Area

Process table pointer	Indicates entry that corresponds to the U area.
User identifiers	Real and effective user IDs. Used to determine user privileges.
Timers	Record time that the process (and its descendants) spent executing in user mode and in kernel mode.
Signal-handler array	For each type of signal defined in the system, indicates how the process will react to receipt of that signal (exit, ignore, execute specified user function).
Control terminal	Indicates login terminal for this process, if one exists.
Error field	Records errors encountered during a system call.
Return value	Contains the result of system calls.
I/O parameters	Describe the amount of data to transfer, the address of the source (or target) data array in user space, and file offsets for I/O.
File parameters	Current directory and current root describe the file system environment of the process.
User file descrip-tor table	Records the files the process has open.
Limit fields	Restrict the size of the process and the size of a file it can write.
Permission modes fields	Mask mode settings on files the process creates.

Creazione di un Processo in UNIX

- Effettuata tramite la chiamata di sistema `fork()`
- In seguito a ciò, il SO, in Kernel Mode:
  1. Alloca una entry nella tabella dei processi per il nuovo processo (figlio)
  2. Assegna un PID unico al processo figlio
  3. Copia l'immagine del padre, escludendo dalla copia la memoria condivisa (se presente)
  4. Incrementa i contatori di ogni file aperto dal padre, per tenere conto del fatto che ora sono anche del figlio
  5. Assegna al processo figlio lo stato Ready to Run
  6. Fa ritornare alla fork il PID del figlio al padre, e 0 al figlio
- Dopodiché, il Kernel può scegliere tra:
  - continuare ad eseguire il padre.
  - switchare al figlio
  - switchare ad un altro processo

I Thread (lett.: Fili)

- Finora, ciascun processo compete con tutti gli altri per l'esecuzione
  - quindi c'è un'esecuzione in parallelo
  - o "finta": i processi si alternano troppo velocemente perché l'utente se ne accorga
  - o "vera": ci sono più processori, o un processore con più core, o alcuni processi sono in realtà in esecuzione su dispositivi di I/O
- Tuttavia, per alcune applicazioni è importante essere a loro volta organizzate in modo parallelo
  - questa volta, è lo stesso programmatore ad organizzarle in tal modo nel codice sorgente
  - esempio tipico: un'applicazione grafica
    - di per sé, è un processo
    - ma conviene almeno avere, contemporaneamente, un'esecuzione che monitori i click del mouse, un'altra che ridisegni la finestra, e un'altra che faccia effettivamente i calcoli richiesti

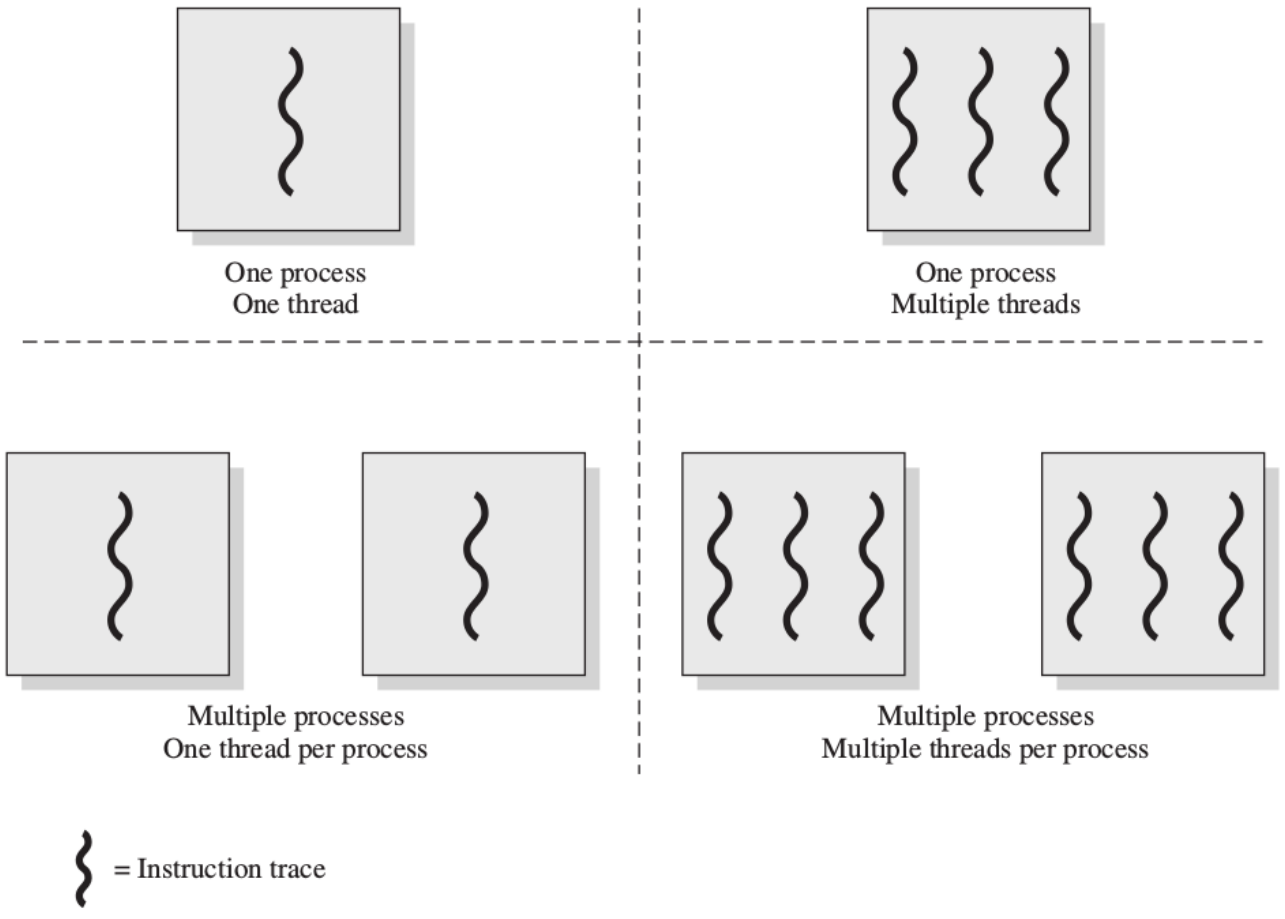
I Thread

- Queste diverse esecuzioni di uno stesso processo sono appunto chiamate *thread*
- Diversi thread di uno stesso processo condividono tutte le risorse tranne:
  - lo stack delle chiamate
    - quindi, le variabili *locali* a ciascuna funzione *non* sono condivise
  - il processore
- Condividono quindi tutte le altre risorse, ad es.: memoria (stack esclusi), files, dispositivi di I/O etc.
  - se un thread acquisisce un dispositivo di I/O (ad es., un file), è come se l'avesse acquisito l'intero processo
  - quindi, anche tutti gli altri thread di quel processo vedono quel file

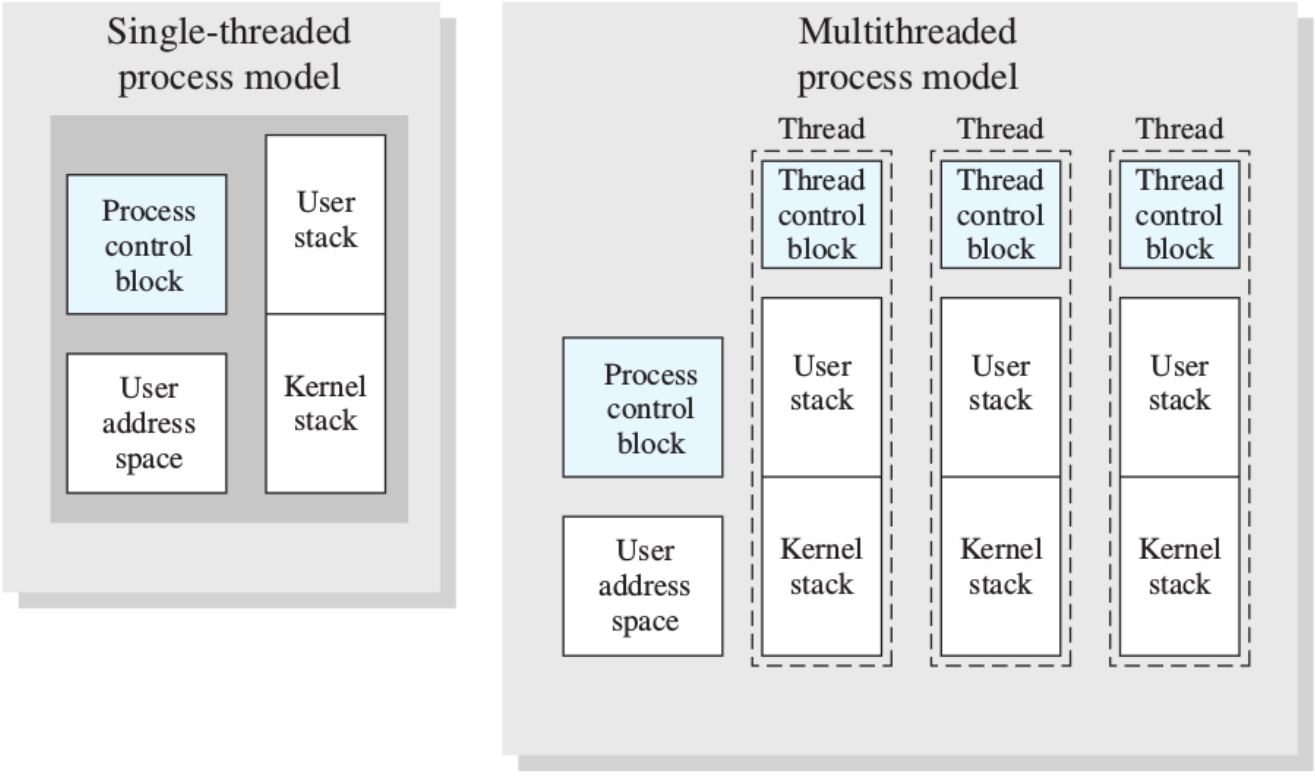
I Thread

- Teoricamente viene molto bene: si può dire che il concetto di processo incorpori le seguenti 2 caratteristiche
  - gestione delle risorse
  - scheduling/esecuzione
- Per quanto riguarda le risorse, i processi vanno presi come un blocco unico
- Per quanto riguarda lo scheduling, i processi possono contenere diverse tracce (thread)
- Nel qual caso, vanno trattati in maniera divisa

I Thread


One process
One thread
One process
Multiple threads
Multiple processes
One thread per process
Multiple processes
Multiple threads per process
= Instruction trace

I Thread

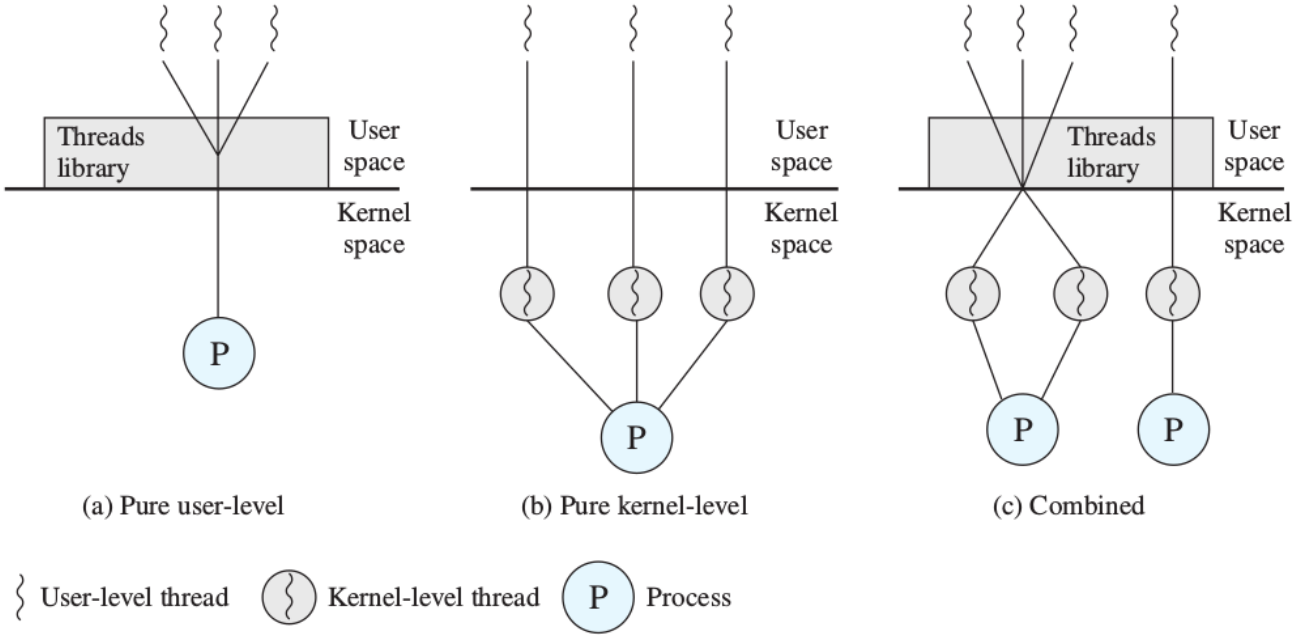

Single-threaded process model
Process control block
User stack
User address space
Kernel stack
Multithreaded process model
Thread
Thread
Thread
Thread control block
Thread control block
Thread control block
Process control block
User stack
User stack
User stack
User address space
Kernel stack
Kernel stack
Kernel stack

I Thread

- È più semplice/efficiente: crearli, terminarli, fare lo switching, farli comunicare
- Ogni processo viene creato con un thread, dopodiché è possibile fare le seguenti operazioni:
  - spawn: "germina" un nuovo thread
  - block: non per I/O, ma esplicito (ad es. perché deve aspettare un altro thread)
  - unblock: anche qui esplicito
  - finish

ULT vs. KLT

User Level Thread vs. Kernel Level Thread



ULT vs. KLT

- Meglio gli ULT perché:
  - è più facile ed efficiente lo switch (non richiede il mode switch al kernel)
  - si può avere una politica di scheduling diversa per ogni applicazione
  - permettono di usare i thread anche sui sistemi operativi che non li offrono nativamente
- Peggio gli ULT perché:
  - se un thread si blocca, si bloccano tutti i thread di quel processo
    - a meno che il blocco non sia dovuto alla primitiva **block** dei thread
    - con i KLT, si blocca *solo* il thread richiedente
  - se ci sono effettivamente più processori o più cores, tutti i thread del processo ne possono usare uno solo
  - se il sistema operativo non ha i KLT, niente thread per le routine del sistema operativo stesso

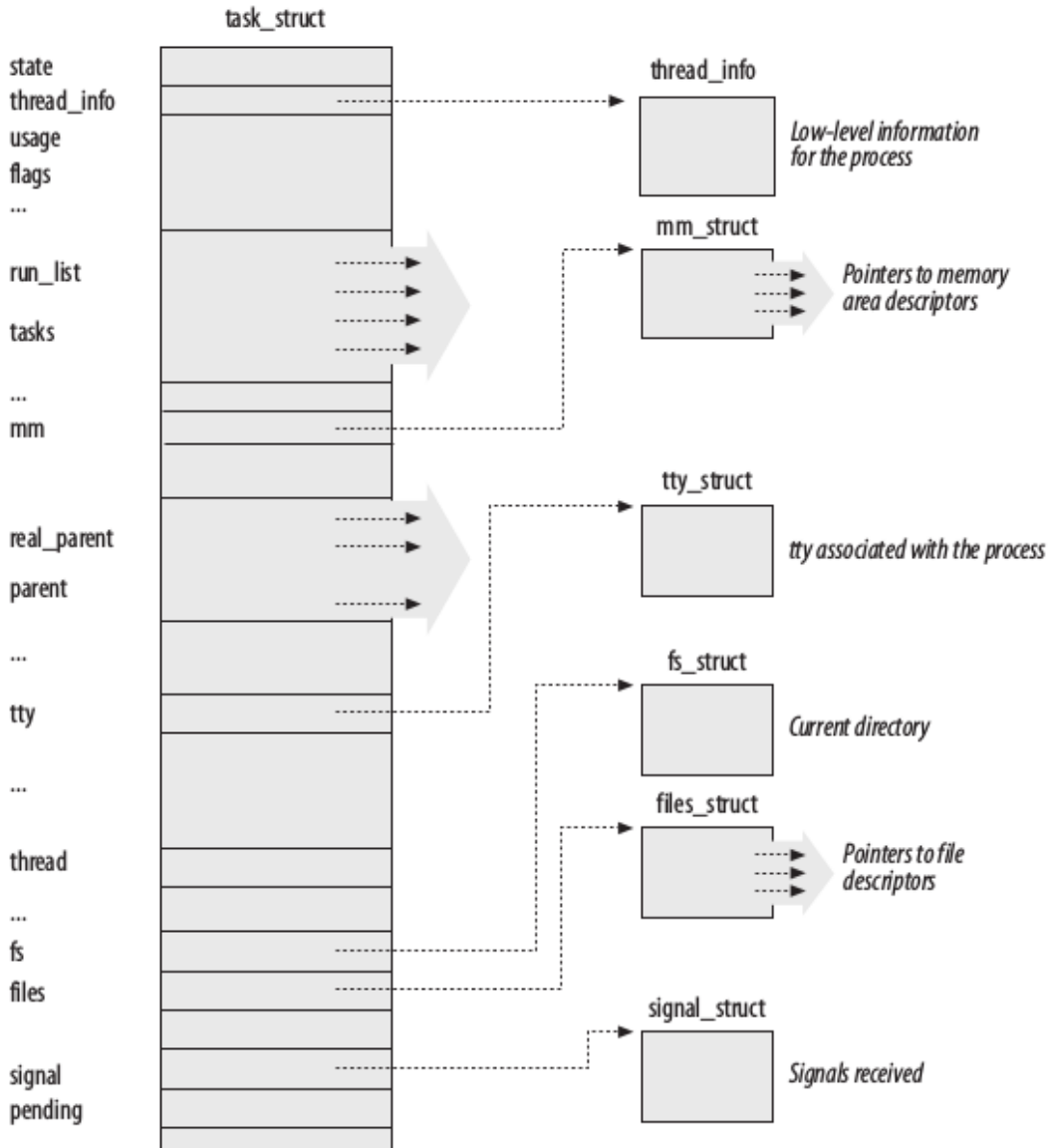
Processi e Thread in Linux

- Unità base: LWP (Lightweight process)
  - è il nome che Linux dà ai thread
- Possibili sia i KLT che gli ULT
  - KLT usati principalmente dal sistema operativo
  - qualsiasi utente può scrivere degli ULT, che vengono poi mappati in KLT (pthreads)
- Identificazione: piccolo inghippo terminologico, utente e sistema usano termini diversi
- Per l'utente che ad esempio usi il comando `ps` da terminale:
  - il PID è unico per tutti i thread di un processo
  - il `tid` (task identifier) identifica ogni singolo thread
  - da notare che il `tid` *non* è un semplice numero che va da 1 a *n*
    - con *n* numero di thread del processo
  - anzi, c'è sempre un thread per il quale il `tid` coincide con il PID
  - questo perché...

Thread in Linux: Visione di Sistema

- Il `pid` (entry del PCB) è unico *per ogni thread*
  - proprio perché l'unità base è l'LWP, che coincide col concetto di thread
- L'entry del PCB che dà il PID comune a tutti i thread di un processo è il `tgid`
  - thread group (leader) identifier: coincide con il PID del primo thread del processo (vedere slide precedente)
- Una chiamata a `getpid()` restituisce il `tgid`
- Per processi con un solo thread, ovviamente `tgid` e `pid` coincidono
- C'è un PCB *per ogni thread*
  - diversi thread di uno stesso processo duplicheranno svariate informazioni
  - in realtà si tratta di duplicazioni di *puntatori*, quindi l'overhead è basso
- Il comando `ps` confonde un po' le cose: PID è a tutti gli effetti il TGID; il PID vero è indicato come LWP o SPID o TID

Il PCB in Linux



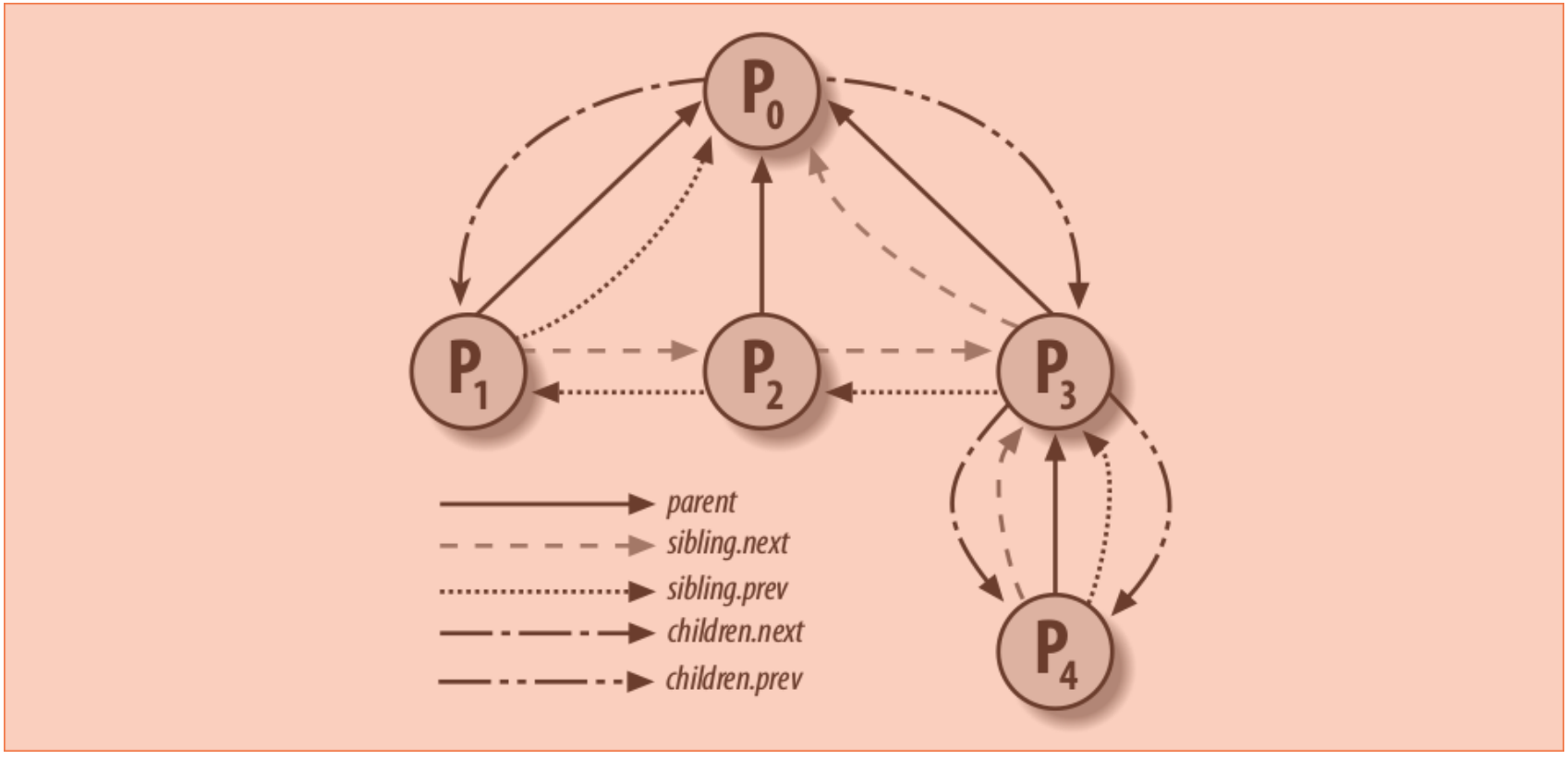
Il PCB in Linux

- La struttura `thread_info` è organizzata per contenere anche il kernel stack
  - ovvero, lo stack delle chiamate da usare quando il processo passa in modalità sistema
- `thread_group`: punta agli altri thread di questo stesso processo
- `parent` e `real_parent`: puntano al padre del processo
- ci sono anche link per i fratelli e i figli

Gli Stati dei Processi in Linux

- È sostanzialmente come quello a 5 stati
  - ovvero, non fa un'esplicita menzione a processi suspended
- Tuttavia, internamente il kernel usa questi stati "atipici"
  - `TASK_RUNNING`: in realtà, include sia Ready che Running
  - `TASK_INTERRUPTIBLE`, `TASK_UNINTERRUPTIBLE`, `TASK_STOPPED`, `TASK_TRACED`: sono tutti Blocked
    - la differenza la fa il *motivo* per cui sono blocked
  - `EXIT_ZOMBIE`, `EXIT_DEAD`: sono entrambi Exit

Processi Parenti



In aggiunta a questo, ogni P_i ha i suoi thread

Segnali ed Interrupt (in Linux)

- Non bisogna confondere *segnali* con *interrupt* (o *eccezioni*)
- I segnali possono essere inviati da un processo utente ad un altro
  - tramite system call; c'è anche un comando shell per farlo
- Quando ciò succede, il segnale appena lanciato viene aggiunto all'opportuno campo del PCB del processo ricevente
  - quando il processo viene nuovamente schedulato per l'esecuzione, il kernel controlla prima se ci sono segnali pendenti
  - se sì, esegue un'opportuna funzione chiamata *signal handler*
  - signal handlers di sistema: possono essere sovrascritti da signal handlers definiti dal programmatore
    - alcuni segnali hanno handler non sovrascrivibili
  - in ogni caso, sono eseguiti in *user mode*
  - invece, i gestori di interrupt/eccezioni sono eseguiti in *kernel mode*

Segnali ed Interrupt (in Linux)

- I segnali possono anche essere inviati da un processo in modalità sistema
- In questo caso, può accadere che questo sia dovuto ad un interrupt a monte
  - esempio tipico: eseguo un programma C scritto male, che accede ad una zona di memoria senza averla prima richiesta
  - il processore fa scattare un'eccezione
  - viene eseguito l'opportuno exception handler (in kernel mode), che essenzialmente manda il segnale `SIGSEGV` al processo colpevole
  - quando il processo colpevole verrà selezionato nuovamente per andare in esecuzione, il kernel vedrà dal PCB che c'è un segnale pendente, e farà in modo che venga eseguita l'azione corrispondente
  - l'azione di default per tale segnale è: fai terminare il processo
  - ma può essere riscritta dall'utente
  - quando verrà eseguita tale azione, sarà in user mode