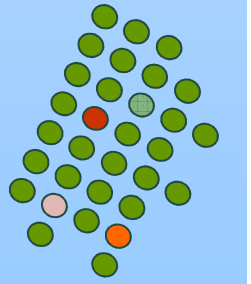
Processi

Capitolo 3 -- Silberschatz



Concetto di processo

- Un SO esegue una varietà di attività:
  - Sistemi batch – job
  - Sistemi time-sharing – programmi utenti o task
- Nel libro i termini job e processo sono usati in modo intercambiabile
- Processo – un programma in esecuzione; l' esecuzione di un processo avviene in modo sequenziale
- Un processo include:
  - program counter
  - contenuto registri CPU
  - sezione testo
  - sezione dati
  - heap
  - stack

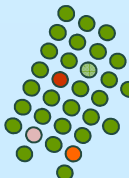
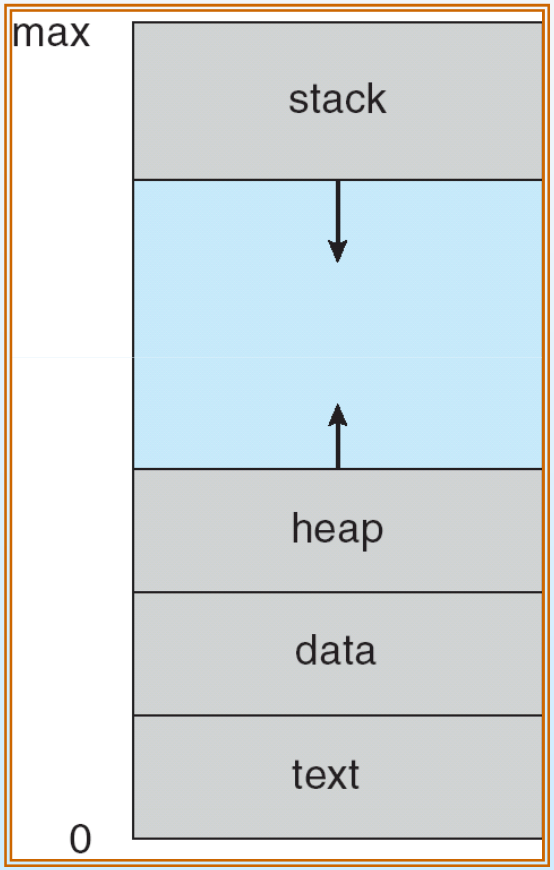


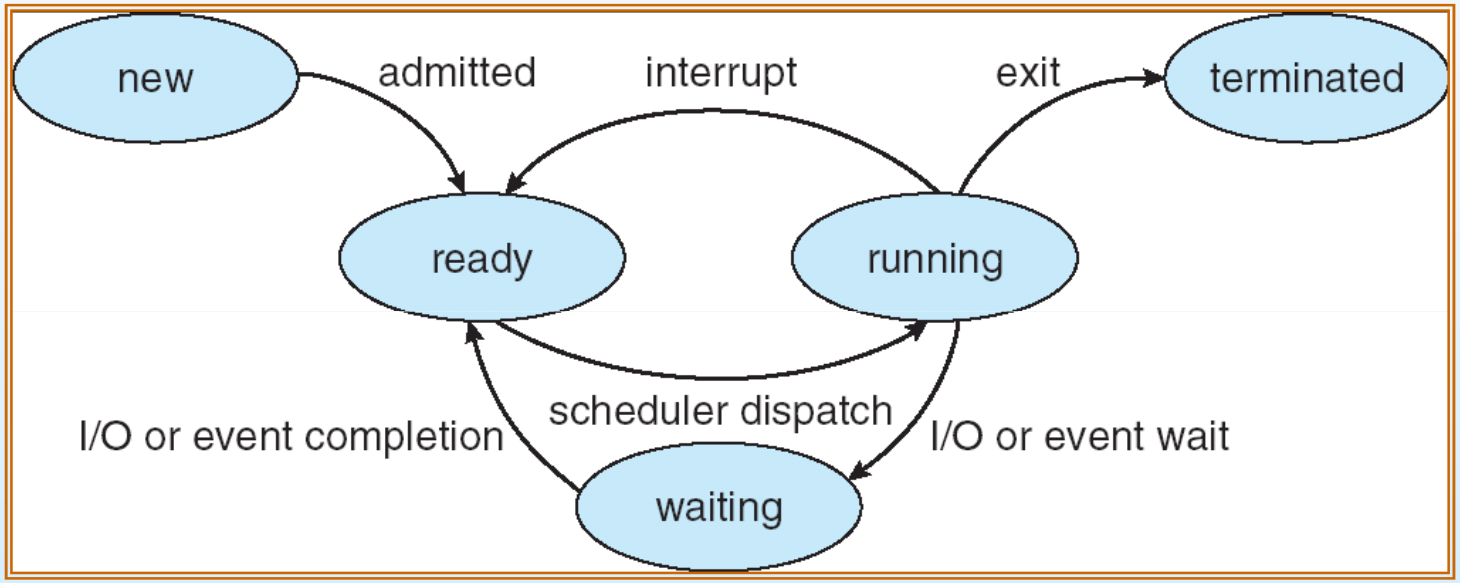
Immagine in memoria di un processo



Stati di un processo

- Durante l'esecuzione un processo cambia stato
  - **new**: Il processo è stato creato
  - **running**: Le sue istruzioni vengono eseguite
  - **waiting**: Il processo è in attesa di qualche evento
  - **ready**: Il processo è in attesa di essere assegnato ad un processore
  - **terminated**: Il processo ha terminato l'esecuzione

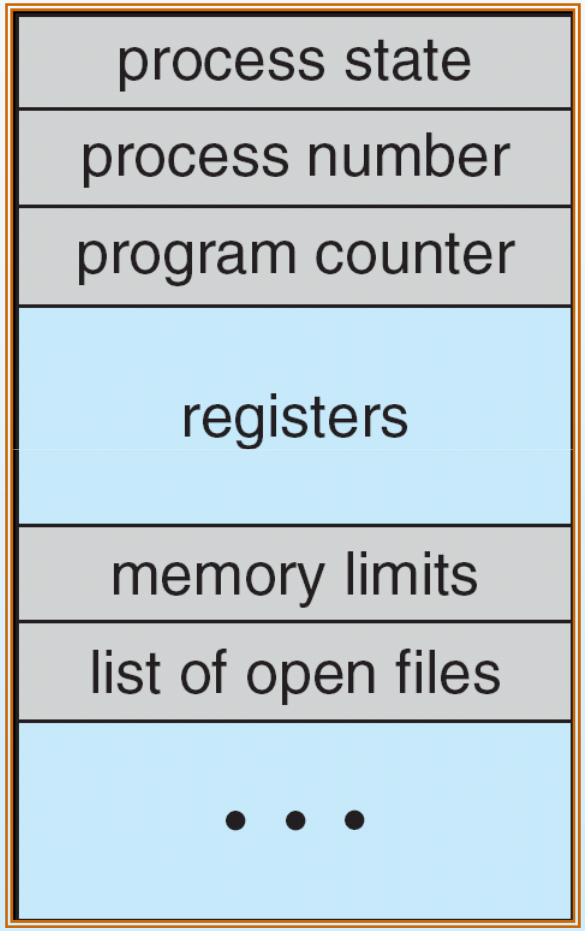
Diagrammma di Stato dei Processi



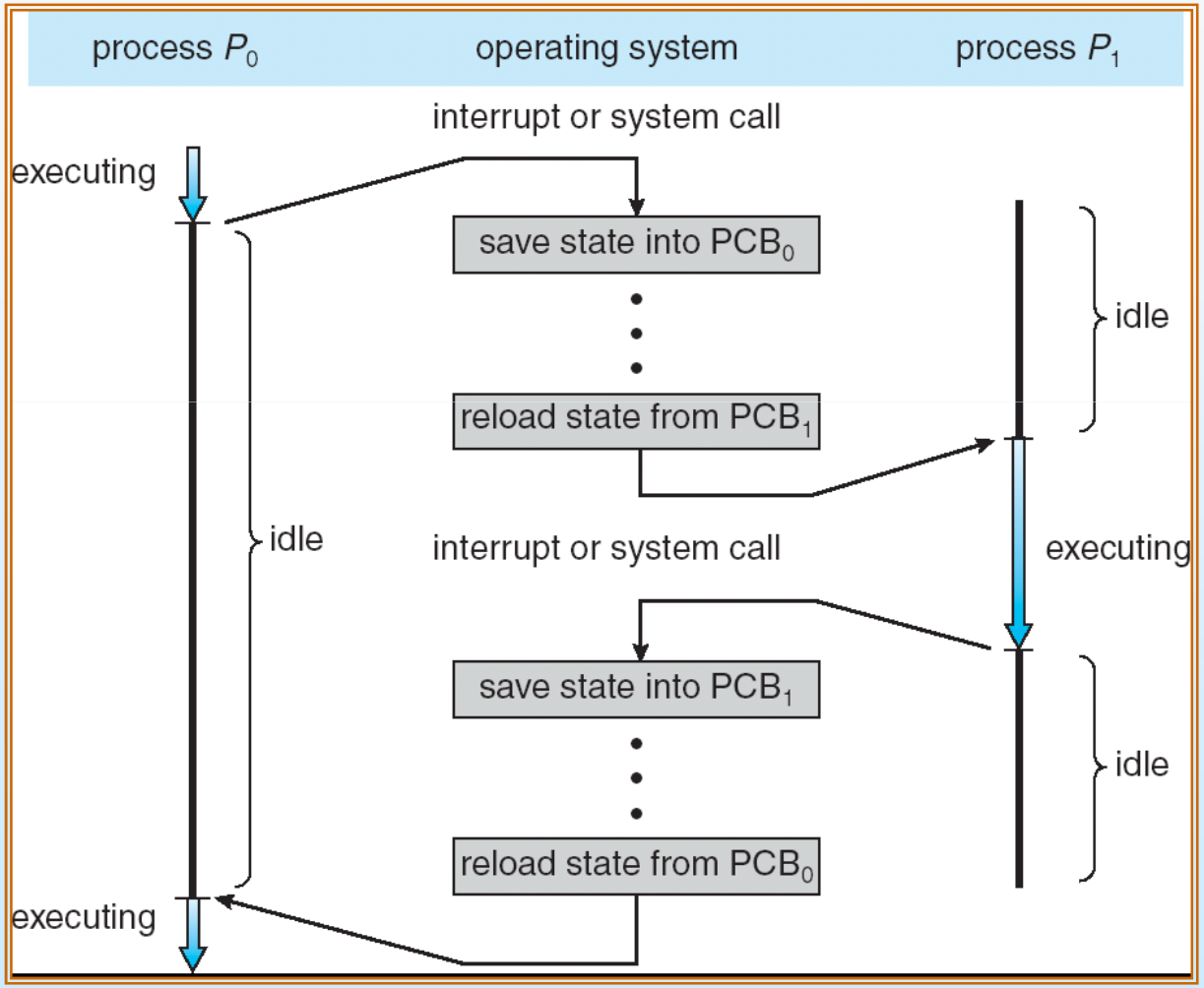
Process Control Block

Contiene informazioni per la gestione del processo

- Stato del processo
- Identificatore
- Program counter
- Registri della CPU
- Informazioni per lo scheduling CPU
- Informazioni per la gestione della memoria
- Informazioni di contabilizzazione
- Informazioni sullo stato dell' I/O



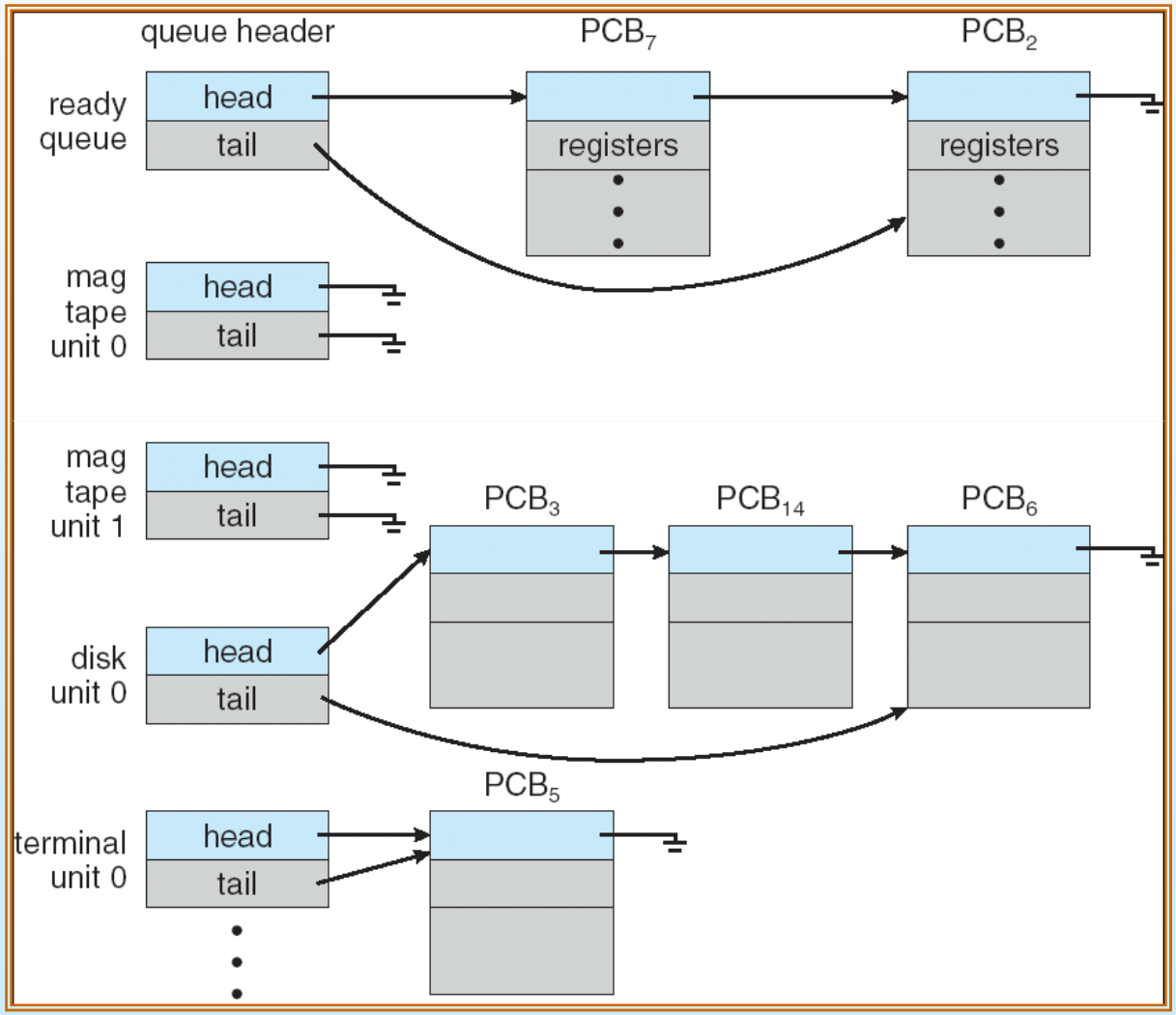
La CPU commuta da Processo a Processo



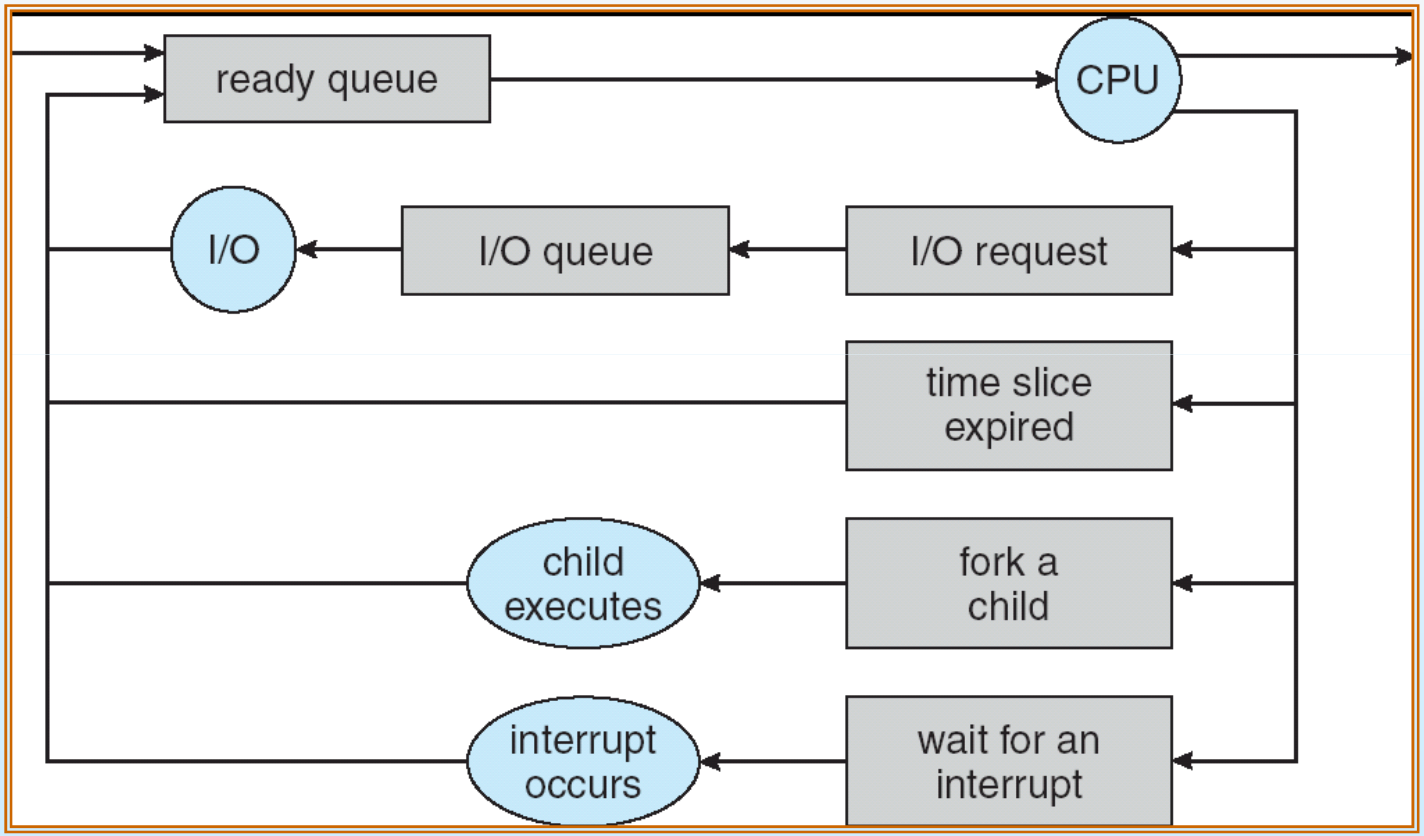
Code di scheduling per i processi

- **Job queue** – insieme di tutti i processi nel sistema
- **Ready queue** – insieme di tutti i processi in memoria centrale pronti per l'esecuzione
- **Code dei dispositivi** – insieme dei processi in attesa di qualche dispositivo di I/O
- I processi, durante la loro vita, migrano tra varie code

Ready Queue e varie code di I/O



Ciclo di vita di un processo



Schedulatori

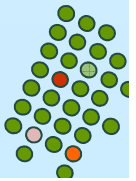
Il sistema si serve di uno schedulatore per selezionare un processo

- **Schedulatore a lungo termine** (o job scheduler) – seleziona i processi che devono essere caricati in memoria centrale (ready queue)

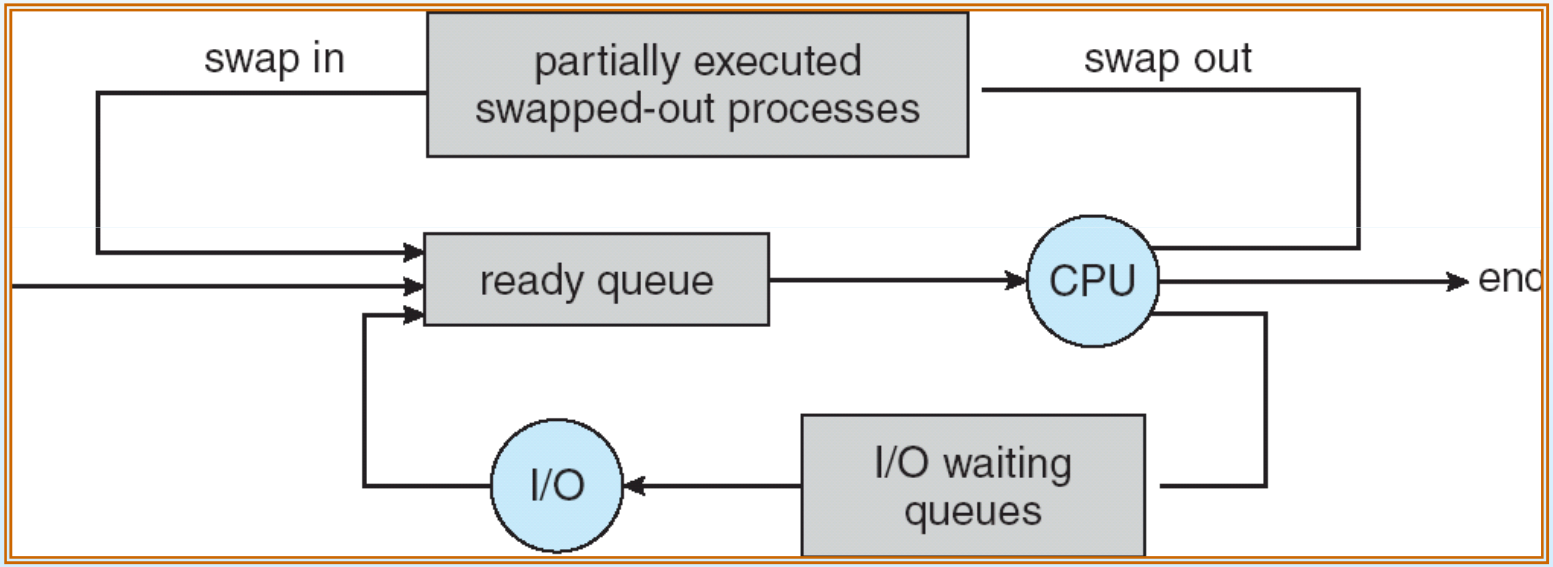
- **Schedulatore a breve termine** (o CPU scheduler) – seleziona il prossimo processo che la CPU dovrebbe eseguire

Schedulatori

- Lo schedulatore a breve termine viene invocato molto *frequentemente* (millisecondi) deve essere veloce
- Lo schedulatore a lungo termine viene invocato *raramente* (secondi, minuti) può essere più lento
- Lo schedulatore a lungo termine controlla il *grado di multiprogrammazione* ed interviene quando un processo termina
- I processi possono essere descritti come:
  - **Processi I/O-bound** – consumano più tempo facendo I/O che computazione, contengono molti e brevi CPU burst
  - **Processi CPU-bound** – consumano più tempo facendo computazione; contengono pochi e lunghi CPU burst



Schedulatore a medio termine



Cambio di contesto (Context switching)

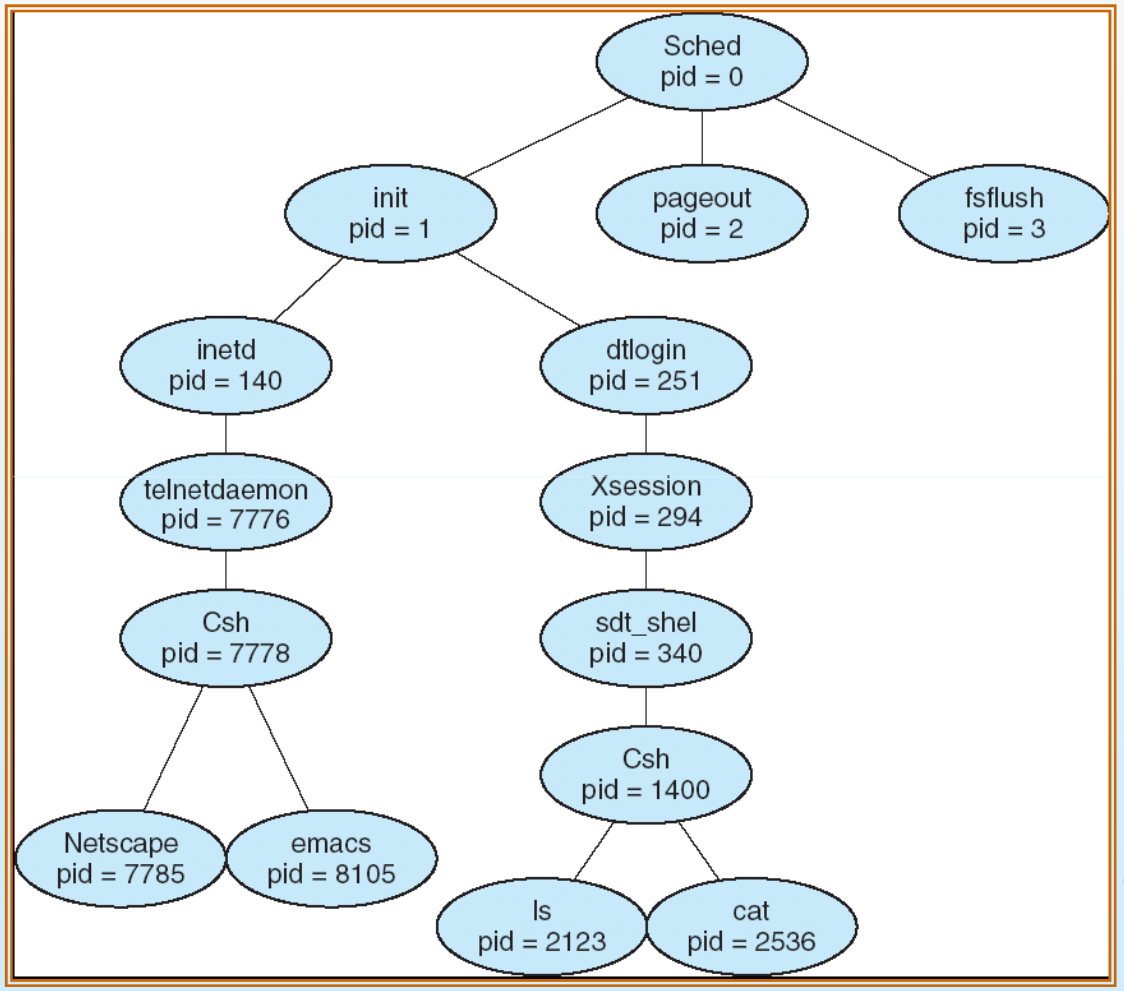
- Quando la CPU viene allocata ad un altro processo, il SO deve salvare lo stato del processo in esecuzione e caricare lo stato del nuovo processo
- Il tempo per un context switch è tempo sprecato (overhead); il sistema non fa nulla di utile mentre commuta
- Il tempo dipende dal supporto hardware
- Il context switch viene realizzato dal **dispatcher**

Creazione di processi

- Il sistema operativo fornisce meccanismi per creare e terminare processi
- Un processo padre crea processi figli che, a loro volta, creano altri processi, formando un albero di processi
- Ogni processo ha un identificatore (process identifier) PID

Un albero di processi di Solaris

Sched genera:
-**fsflush, pageout** (che si occupano della gestione della memoria e del file system) e
- **init** (che il progenitore di tutti i processi utente)



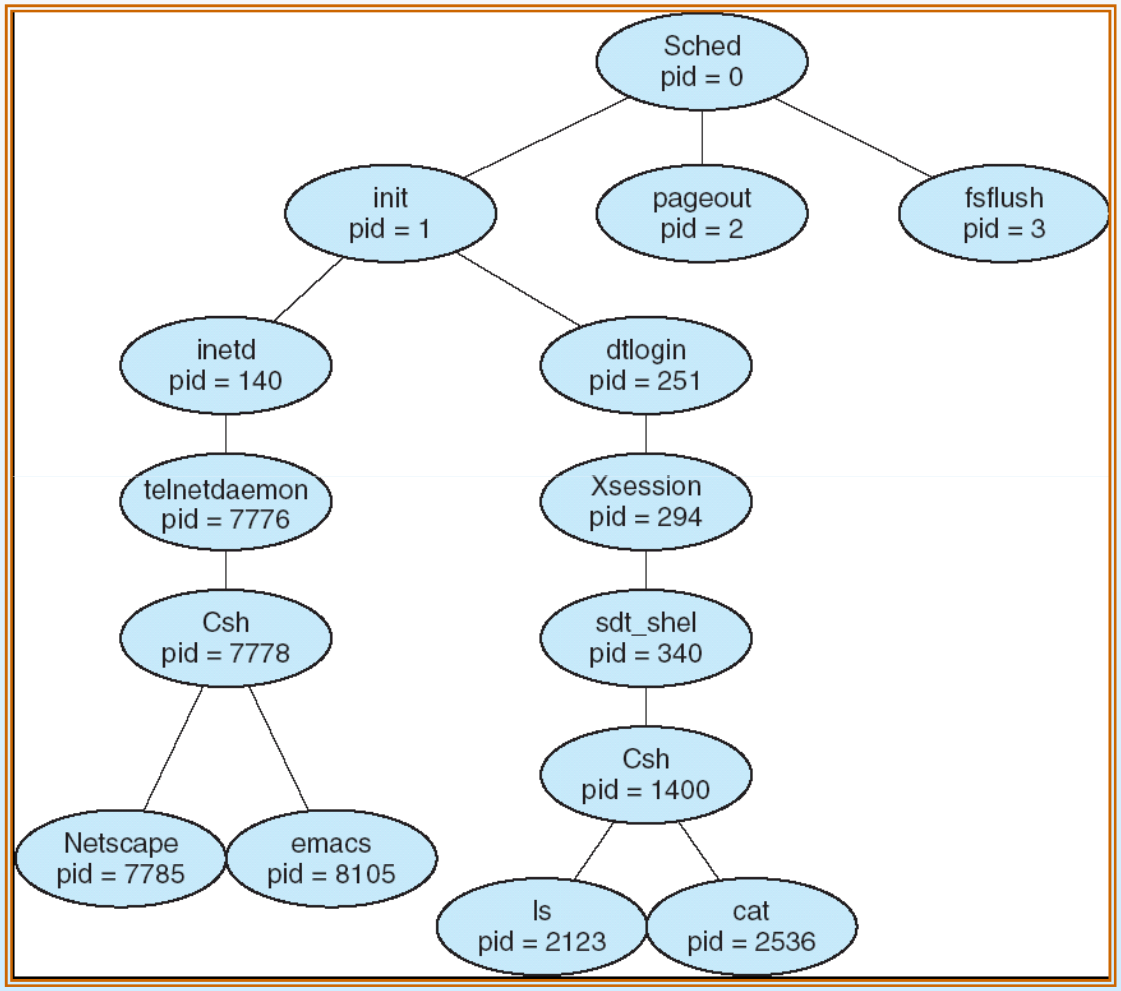
Un albero di processi di Solaris

Init genera:
-**inetd** (che gestisce i servizi di rete: telnet, ftp)

-Il processo **csh (pid=7778)** si riferisce ad un utente che ha avuto acceso al sistema mediante telnet ed ha avviato Netscape ed emacs

-**dtlogin** (che rappresenta la schermata di accesso al sistema).

-**dtlogin** crea una sessione X-windows che a sua volta genera **sdt_shel** al di sotto del quale risiede la shell dell'utente.



Creazione di processi: implementazione

- Condivisione di risorse
  - Padre e figli condividono tutte le risorse
  - figli condividono un sottoinsieme delle risorse del padre
  - Padre e figlio non condividono niente
- Esecuzione
  - Padre e figli vengono eseguiti in concorrenza
  - Padre aspetta fino alla terminazione dei figli

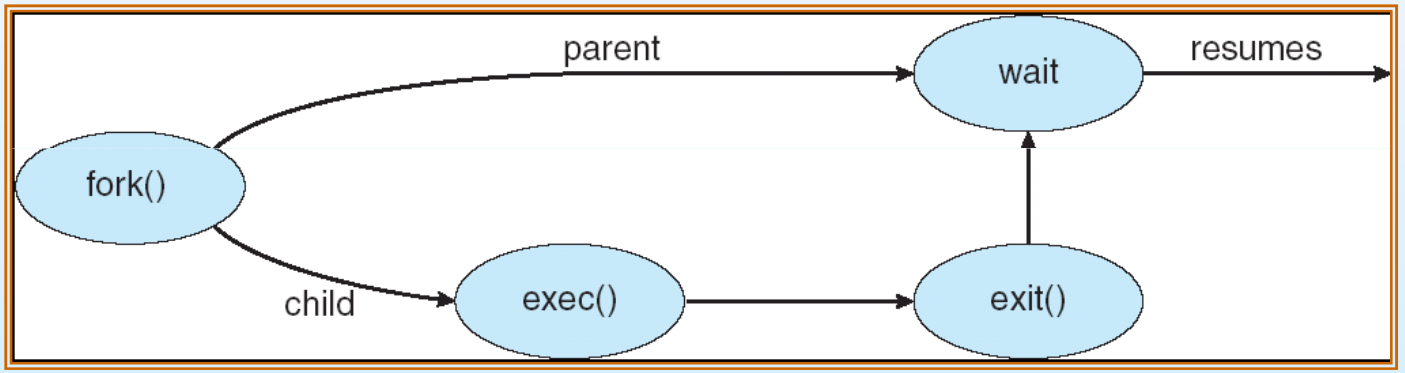
Crazione di processi: implementazione

- Spazio di indirizzamento
  - Lo spazio del figlio è un duplicato di quello del padre
  - Il figlio carica in esso un nuovo programma
- Esempi UNIX
  - La chiamata **fork** crea un nuovo processo
  - La chiamata **exec** viene usata dopo una **fork** per sostituire l'immagine in memoria del processo con un nuovo programma
  - La chiamate **wait** permette al padre di aspettare che il figlio termini

Terminazione di un processo

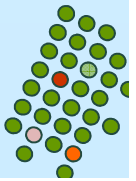
- Un processo esegue l'ultima istruzione e chiede al SO di eliminarlo (**exit**)
  - Trasmette un valore di output al padre (via **wait**)
  - Le risorse del processo sono recuperate dal SO
- Il processo padre può terminare l'esecuzione dei processi figli (e.g., **TerminateProcess()** in Win32)
  - Un figlio ha ecceduto nell'uso delle risorse
  - Il compito assegnato al figlio non è più richiesto
  - Se il padre ha terminato
    - Alcuni SO non permettono ai processi figli di continuare
    - Tutti i figli vengono terminati – *terminazione a cascata*

Creazione di un Processo con padre che aspetta il figlio



C Program Forking Separate Process

```
int main()
{
pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                exit(-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");
                exit(0);
    }
}
```

C Program Forking Separate Process

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

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

ZeroMemory( &si, sizeof(si) );
si.cb = sizeof(si);
ZeroMemory( &pi, sizeof(pi) );

// Start the child process.
if( !CreateProcess( NULL, // No module name
                          (use command line).
   "C:\\WINDOWS\\system32\\mspaint.exe",
                          // Command line.
    NULL, // Process handle not inheritable.
    NULL, // Thread handle not inheritable.
    FALSE, // Set handle inheritance to FALSE.
    0, // No creation flags.
    NULL, // Use parent's environment block.
    NULL, // Use parent's starting directory.
    &si, // Pointer to STARTUPINFO structure.
    &pi ) // Pointer to PROCESS_INFORMATION
structure.
)
{
   printf( "CreateProcess failed (%d).\n", GetLastError() );
   return -1;
}

// Wait until child process exits.
WaitForSingleObject( pi.hProcess, INFINITE );

// Close process and thread handles.
CloseHandle( pi.hProcess );
CloseHandle( pi.hThread );
}
```