Processi

Pietro Braione

Reti e Sistemi Operativi – Anno accademico 2021-2022

Obiettivi

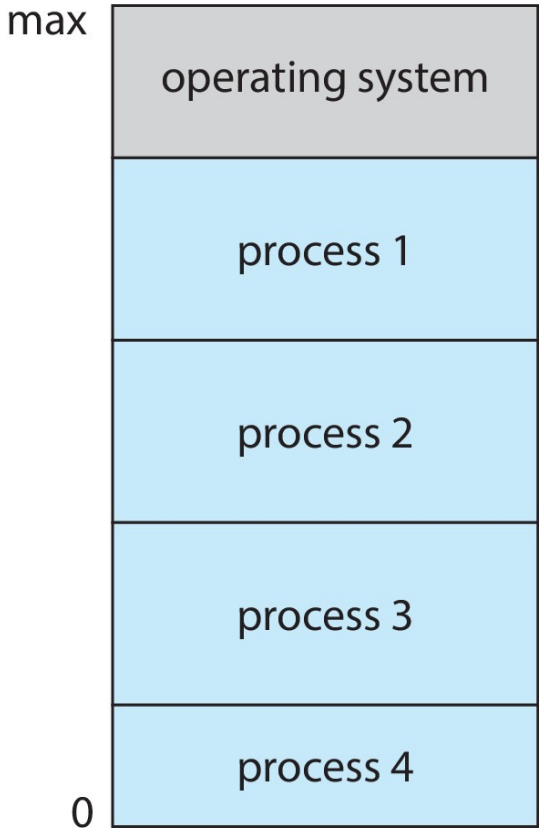
- Concetto di processo
- Scheduling dei processi
- Operazioni sui processi
- IPC (memoria condivisa, message passing)

Multiprogrammazione e multitasking

- Un sistema operativo esegue un certo numero di programmi in maniera concorrente
- Obiettivo: mantenere impegnata la (o le) CPU il maggior tempo possibile
- Le tecniche adottate sono la multiprogrammazione e il multitasking
- **Multiprogrammazione**:
  - Il sistema operativo mantiene in memoria un sottoinsieme dei programmi da eseguire, o **processi**
  - Quando una CPU non è impegnata, il sistema operativo seleziona un processo non in esecuzione attraverso una procedura detta di **scheduling** e lo manda in esecuzione
  - Quando un processo in esecuzione deve attendere (ad es. per un'operazione di I/O), l'OS passa ad eseguire un altro processo
- **Multitasking**, o **time-sharing**:
  - L'estensione logica della multiprogrammazione in cui la CPU cambia processo così frequentemente che viene data l'illusione che i processi stiano eseguendo ciascuno su una CPU dedicata
  - In tal modo l'utente può interagire con ogni processo mentre è in esecuzione (interactive computing)
  - A tale scopo il tempo di risposta deve essere < 1 sec

Configurazione di memoria con multiprogrammazione

- Se i processi non possono essere contenuti tutti in memoria: lo **swapping** li sposta dentro/fuori dalla memoria
- La **memoria virtuale** permette di eseguire un processo non completamente in memoria
- (ne parleremo più avanti)

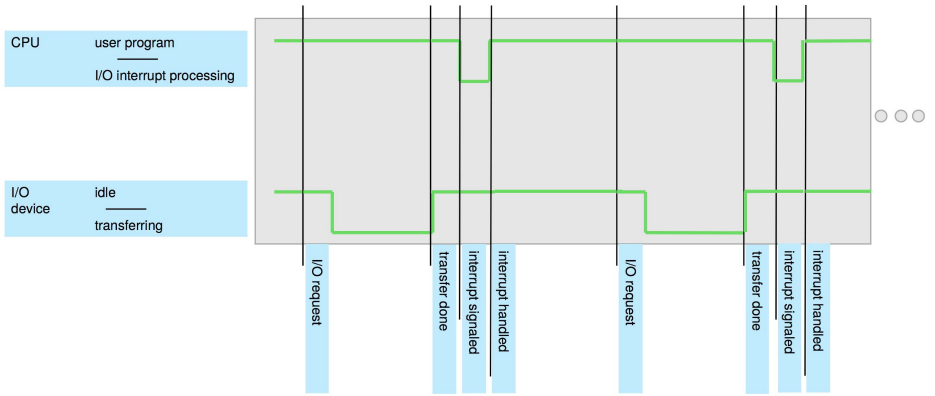


La rottura del modello di Von Neumann

- Nella macchina di Von Neumann il processore esegue un programma ininterrottamente (fetch-decode-execute) e si blocca su un'operazione di I/O finché non è terminata
- Tali tecniche invece richiedono che un programma in esecuzione venga interrotto per mandarne in esecuzione un altro
- A tale scopo i computer attuali utilizzano il meccanismo degli **interrupt**, che permettono (appunto) di interrompere una computazione effettuata da una CPU al verificarsi di un evento

Gli interrupt (1)

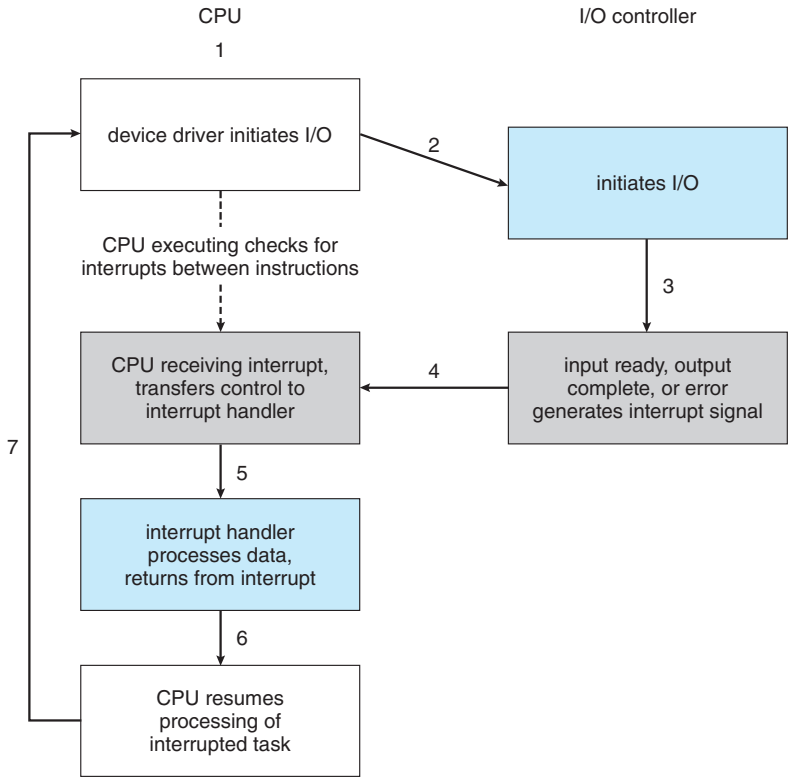
- Sono segnali che vengono inviati alla CPU e servono per effettuare operazioni asincrone rispetto al flusso di esecuzione della CPU
- Gli interrupt possono essere di due tipi:
  - Interrupt hardware: La CPU riceve un segnale di interrupt attraverso linee di solito integrate nel bus di sistema (I/O, timer)
  - Interrupt software o **eccezioni**: errori (divisione per zero, accesso illegale alla memoria) o istruzioni macchina particolari che permettono di effettuare chiamate a servizi di sistema (**system calls**)



Gli interrupt (2)

- L'effetto della ricezione di un segnale di interrupt è quello di trasferire l'esecuzione (salto) ad una opportuna locazione di memoria:
  - Il processore possiede una tabella configurabile (**vettore delle interruzioni**), nella quale, per ogni **linea di richiesta di interrupt** (linea IRQ), è contenuto l'indirizzo del salto
  - A tale locazione di memoria occorre inserire una opportuna **routine di gestione dell'interrupt** (**ISR**)
- La ISR deve salvare lo stato della computazione interrotta, «servire» l'interruzione (ad esempio, trasferire un dato da registro del controller a memoria), e una volta terminato ripristinare la computazione interrotta

Ciclo di I/O guidato dagli interrupt



- Una volta iniziata l'operazione di I/O, la CPU è libera di effettuare altre computazioni, senza bisogno di attendere il termine dell'operazione
- Quando l'operazione è terminata, la periferica genera un segnale di interrupt
- Questo causa il trasferimento del controllo all'ISR che gestisce la terminazione dell'operazione di I/O

Attività del sistema operativo

- All'accensione del computer viene caricato un programma di avvio, o bootstrap (parte del firmware, non dell'OS); questo può caricare un ulteriore bootloader (stavolta parte dell'OS), e così via in cascata
- Alla fine viene caricato e mandato in esecuzione il kernel, e i bootloader vengono scaricati dalla memoria
- Il kernel carica i **demoni, o processi di sistema** (programmi di sistema che offrono servizi e sono sempre attivi)
- Il kernel è **interrupt-driven**:
  - Il kernel contiene le ISR per tutti i possibili interrupt
  - Fino a che non avviene un evento o una system call (che genera anch'essa un interrupt) il kernel non esegue, e la CPU esegue un certo programma applicativo
  - Quando viene generato un interrupt, hardware o software, viene trasferito il controllo alla corrispondente ISR contenuta nel kernel, terminata la quale il controllo torna al programma applicativo (lo stesso o un altro)

Gli interrupt e la duplice modalità di funzionamento

- Notare che, se le ISR fanno parte del kernel interrupt-driven, occorre che vi sia una transizione da modalità utente a supervisore quando avviene il salto alla ISR
- Similmente, quando il kernel ritorna dalla ISR verso il programma applicativo, occorre una transizione inversa da modalità supervisore a modalità utente
- System call e interrupt hardware sono molto simili, ed è per questo che sono gestite con lo stesso meccanismo

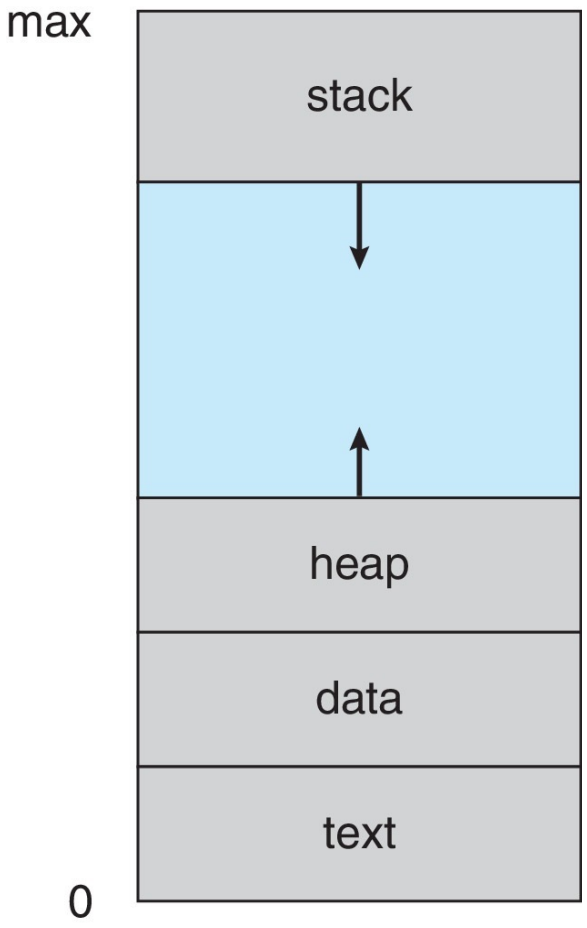
Processi

- Un **processo** è un programma in esecuzione; l'esecuzione di un processo è sequenziale (ma presto rilasseremo questa assunzione)
- Un processo è composto da parti:
  - Lo stato dei registri del processore, incluso il program counter
  - Il codice del programma (**text section**), in memoria centrale
  - Lo **stack** delle chiamate, contenente parametri, variabili locali e indirizzo di ritorno, anche questo in memoria centrale
  - La **data section**, contenente le variabile globali, anche questa in memoria centrale
  - Lo **heap**, contenente la memoria allocata dinamicamente durante l'esecuzione, anche questo in memoria centrale

Attenzione!

- Un programma è un'entità passiva (tipicamente un file eseguibile su disco), un processo è un'entità attiva (è un programma *in esecuzione*)
- Un programma «diventa» un processo quando viene caricato nella memoria centrale
- Un programma può generare diversi processi:
  - Molti utenti eseguono lo stesso programma
  - Uno stesso programma viene eseguito più volte, anche contemporaneamente, dallo stesso utente

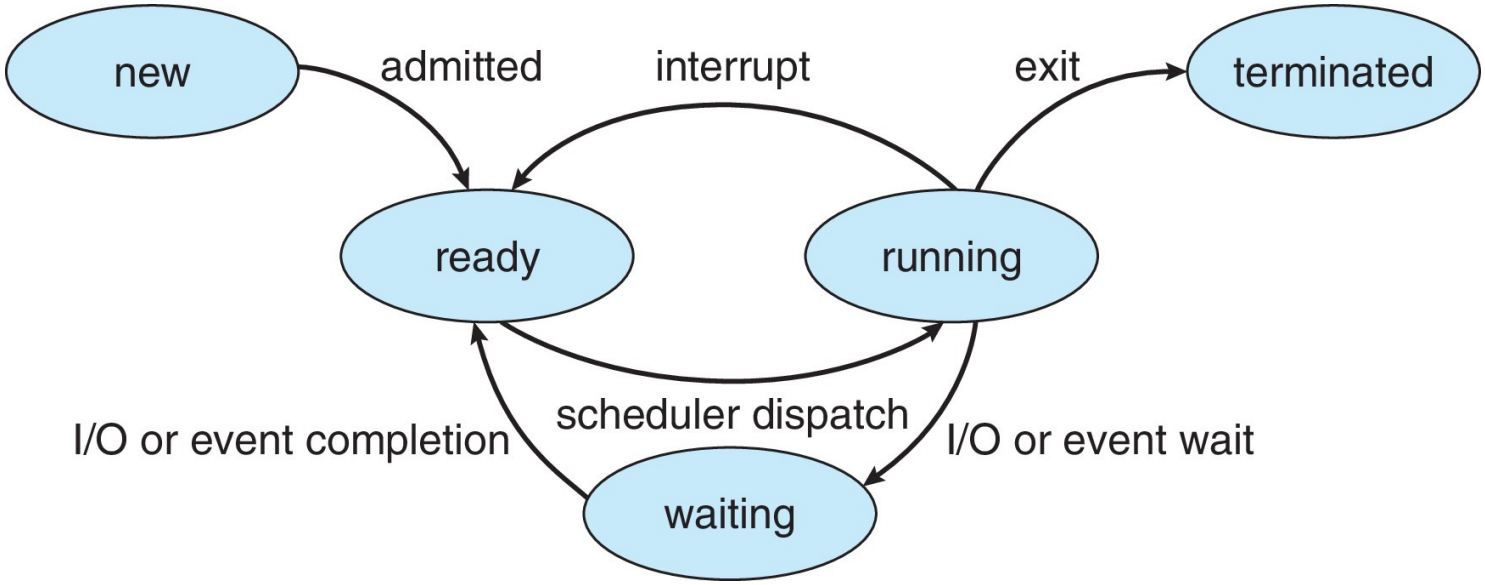
La memoria di un processo



Stato di un processo

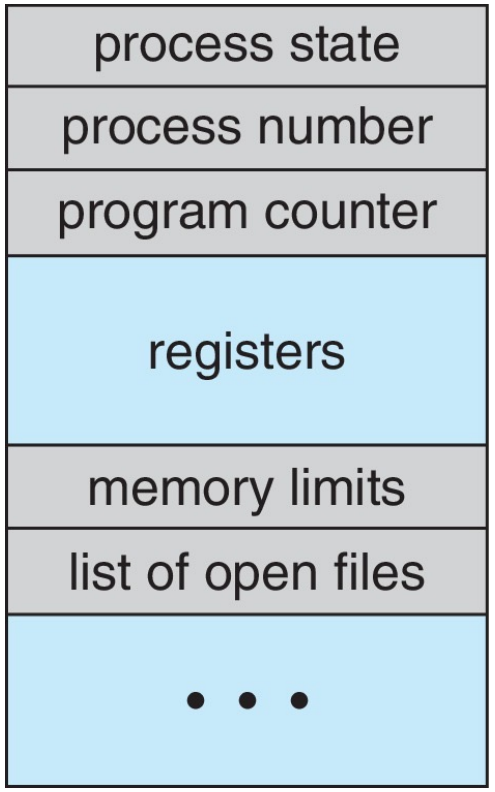
- Durante l'esecuzione, un processo cambia più volte **stato**
- Gli stati possibili di un processo sono:
  - Nuovo (**new**): il processo è creato, ma non ancora ammesso all'esecuzione
  - Pronto (**ready**): il processo può essere eseguito (in attesa dell'assegnazione di una CPU)
  - In esecuzione (**running**): le sue istruzioni vengono eseguite da qualche CPU
  - In attesa (**waiting**): il processo non può essere eseguito perché in attesa di qualche evento (ad es. il completamento di un'operazione di I/O)
  - Terminato (**terminated**): il processo ha terminato l'esecuzione

Diagramma di transizione di stato dei processi



Process Control Block (PCB)

- Detto anche task control block
- Contiene le informazioni relative ad un processo:
  - Process state: ready, running…
  - Process number (o PID): identifica il processo
  - Program counter: contenuto del registro «istruzione successiva»
  - Registers: contenuto dei registri del processore
  - Informazioni relative alla gestione della memoria: memoria allocata al processo
  - Informazioni sull'I/O: dispositivi assegnati al processo, elenco file aperti…
  - Informazioni di scheduling: priorità, puntatori a code di scheduling…
  - Informazioni di accounting: CPU utilizzata, tempo trascorso…



PCB e multithreading

- Fino ad ora abbiamo assunto che un processo abbia un singolo flusso di esecuzione sequenziale (ossia, un singolo processore)
- Se supponiamo che si possano avere *molti* processori per un singolo processo:
  - Più istruzioni possono eseguire concorrentemente, e quindi più percorsi (**thread**) di esecuzione
  - Tutti i thread di uno stesso processo condividono la stessa memoria (e gli stessi file aperti…)
- In tal caso il PCB deve contenere informazioni sufficienti per ogni thread di esecuzione
- Ne parleremo nella prossima lezione

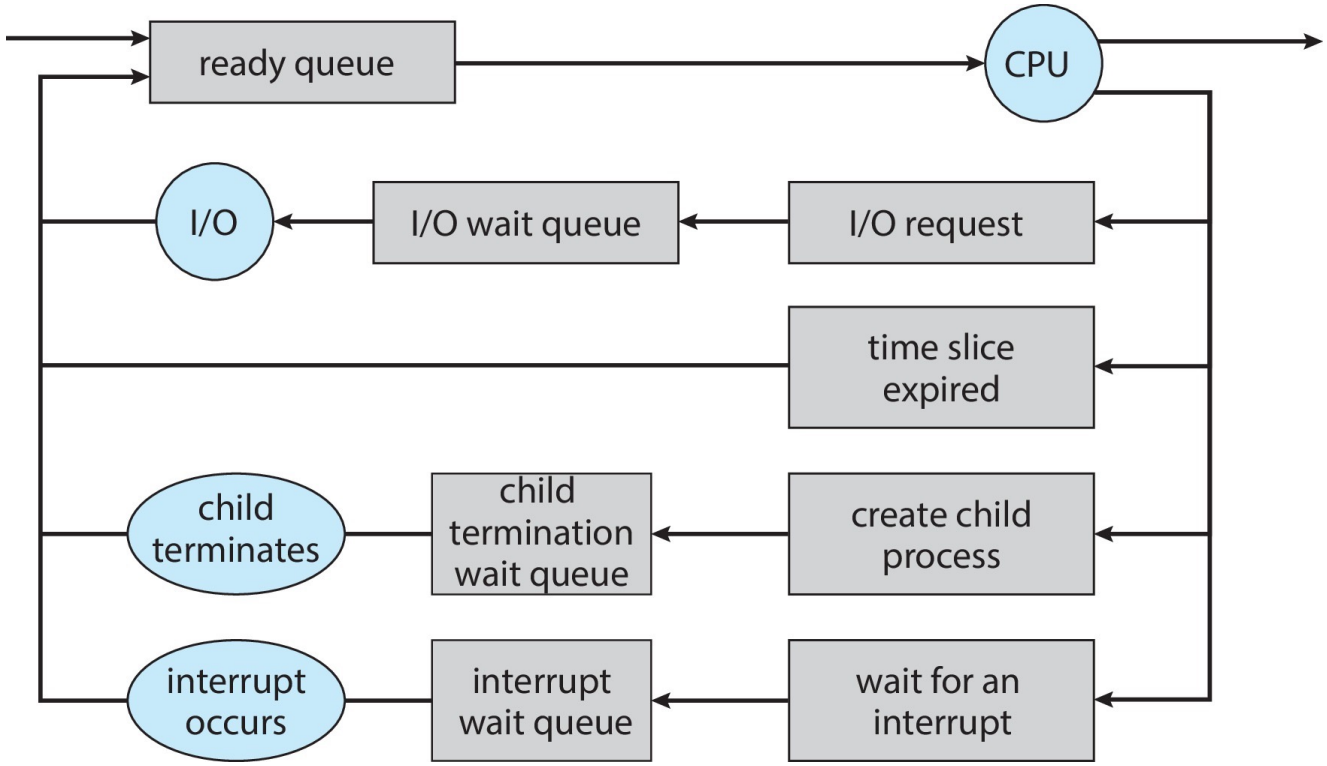
Scheduling dei processi

- Lo **scheduler dei processi** sceglie il prossimo processo da eseguire tra quelli in stato ready
- Mantiene diverse code di processi:
  - **Ready queue**: processi residenti in memoria, pronti e in attesa per l'esecuzione
  - **Wait queues**: diverse code per i processi che non sono pronti per l'esecuzione perché in attesa di un qualche evento (es. completamento operazione di I/O)
- Durante la loro vita, i processi migrano da una coda all'altra

Timer

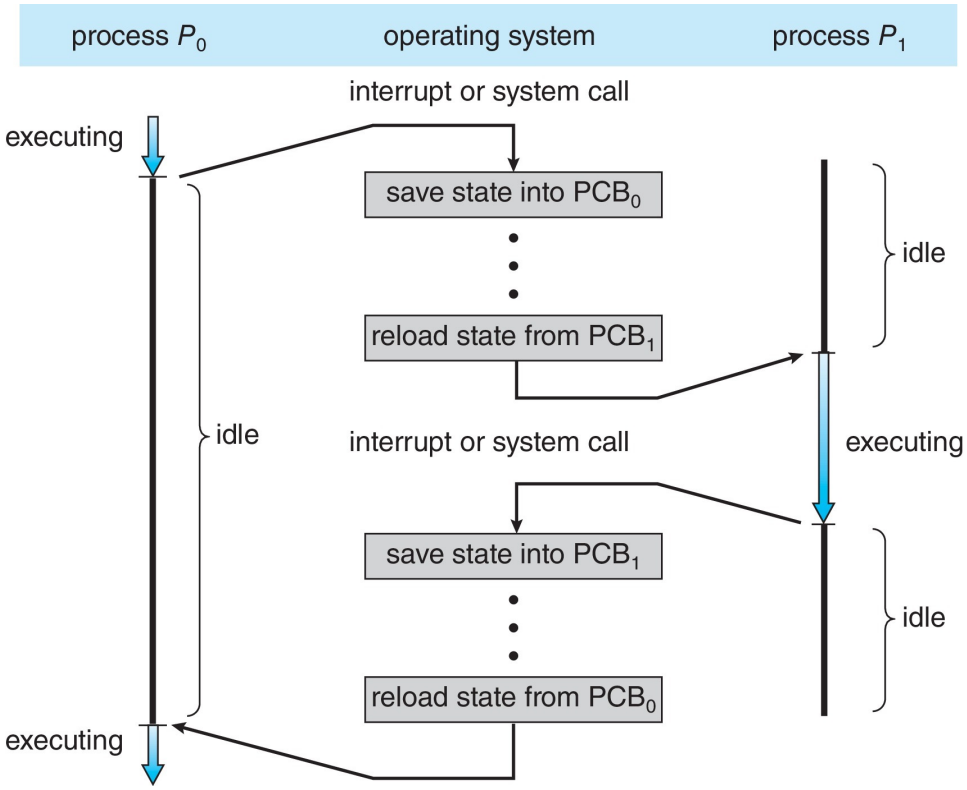
- Occorre garantire che nessun programma occupi la CPU per troppo tempo (sicuramente nel time-sharing, ma anche nella multiprogrammazione batch può essere utile, ad esempio se il programma va in loop infinito)
- A tale scopo si usano **timer** che inviano un interrupt periodico
  - Realizzati con un clock a frequenza fissa ed un contatore
  - Ad ogni colpo di clock il contatore è decrementato, quando è uguale a zero viene resettato e generato l'interrupt
  - Il valore iniziale del contatore (configurabile) definisce la frequenza degli interrupt
- All'interrupt del timer il sistema operativo riprende il controllo e può verificare se il processo corrente ha utilizzato il processore per un tempo eccessivo
- In tal caso il processo corrente è messo nella ready queue e viene attivato lo scheduler per selezionare un altro processo

Rappresentazione dello scheduling di processi



Commutazione di contesto (1)

- Quando la CPU deve passare dall'esecuzione di un processo a quella di un altro processo avviene una commutazione di contesto (**context switch**)
- Il sistema operativo deve salvare il contesto (stato, registri…) del processo precedente, e caricare quello del processo da eseguire
- Il contesto di un processo è contenuto nel PCB
- Notare che questo è un tipico *meccanismo*, mentre lo scheduling è una tipica *politica*



Commutazione di contesto (2)

- Il tempo necessario per il context switch è puro overhead: non viene eseguito alcun lavoro utile
- Più è complesso il sistema operativo, più è complesso il PCB (il contesto), più tempo occorre per un context switch
- Alcuni processori offrono supporto speciale per minimizzare il tempo di context switch (es. banchi di registri multipli)

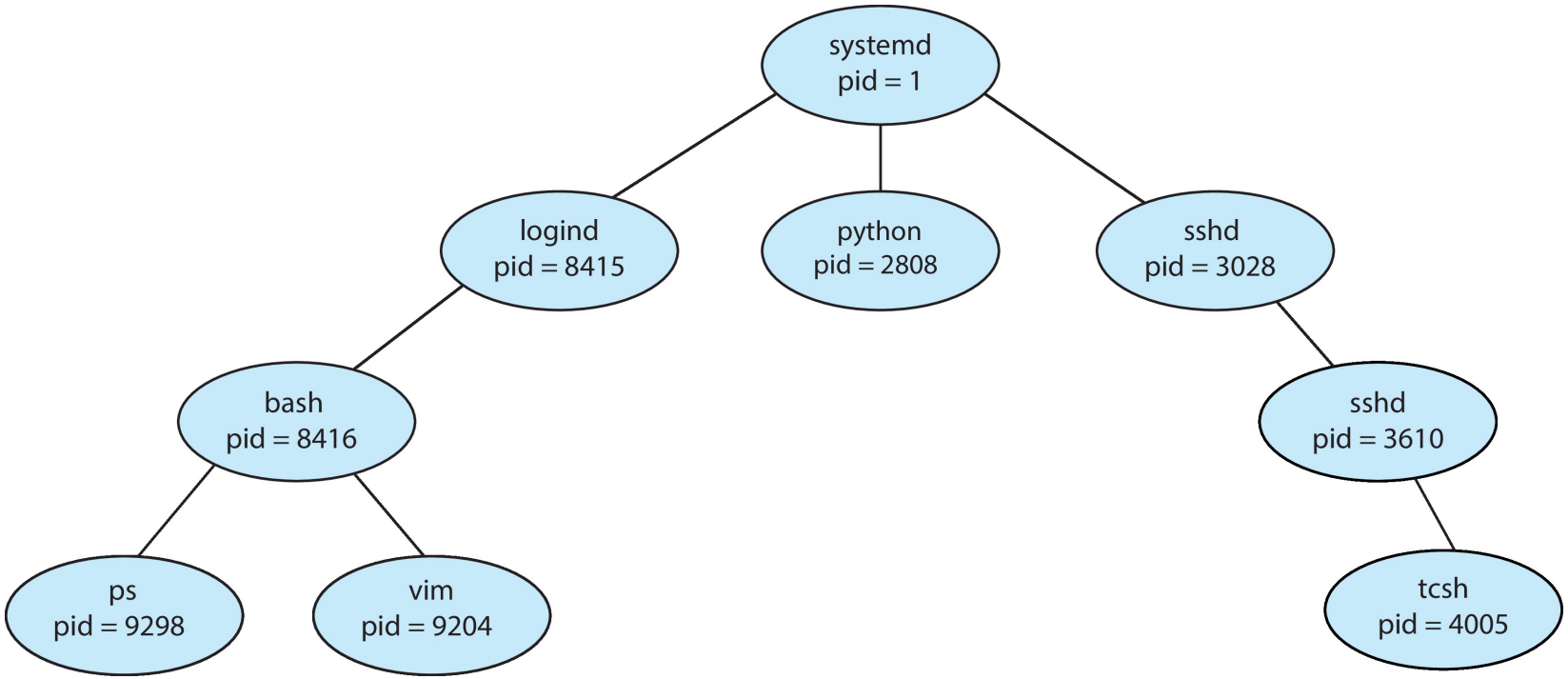
Operazioni sui processi

- Ogni sistema operativo di solito mette a disposizione ai processi delle chiamate di sistema per manipolare altri processi
- (più di regola crea qualche processo «iniziale» che avvia i processi utente e di sistema)
- Discutiamo le due operazioni fondamentali:
  - Creazione di processi
  - Terminazione di processi

Creazione di processi (1)

- Di solito nei sistemi operativi i processi sono organizzati in maniera gerarchica
- Un processo (**padre**) può creare degli altri processi (**figli**)
- Questi a loro volta possono essere padri di altri processi figli, creando un **albero di processi**
- La relazione padre/figlio è di norma memorizzata nel PCB, perché è importante rispetto alle politiche di condivisione risorse e di coordinazione (ved. dopo)

Un albero di processi in Linux

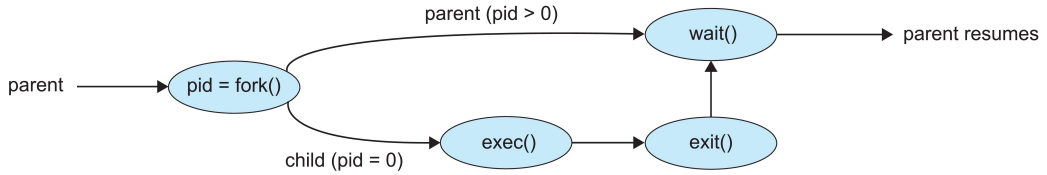


Creazione di processi (2)

- Possibili politiche di condivisione di risorse:
  - Padre e figlio condividono tutte le risorse…
  - …o un opportuno sottoinsieme…
  - …o nessuna
- Possibili politiche di creazione spazio di indirizzi:
  - Il figlio è un duplicato del padre (stessa memoria e programma)…
  - …oppure no, e bisogna specificare quale programma deve eseguire il figlio
- Possibili politiche di coordinazione padre/figli:
  - Il padre è sospeso finché i figli non terminano…
  - …oppure eseguono in maniera concorrente

Esempio: sistema UNIX

- `fork()` crea un nuovo processo figlio, che è un duplicato del padre ed esegue concorrentemente
- `exec()` viene usata dopo una `fork()` per sostituire il programma del figlio con un programma diverso
- `wait()` viene chiamata dal padre per attendere la fine dell'esecuzione del figlio



Terminazione di processi (1)

- I processi richiedono la propria terminazione al sistema operativo tramite chiamata di sistema (`exit()` nel caso di UNIX e UNIX-like)
  - Accetta come parametro un codice di ritorno (di errore)
  - Il sistema operativo elimina il processo e recupera le sue risorse
  - Quindi ritorna al processo padre (chiamata `wait()`) il codice di ritorno
  - Implicitamente chiamata se il processo esce dalla main function
- Un processo padre può forzare la terminazione di un figlio con la chiamata `abort()`. Possibili ragioni:
  - Il figlio sta usando risorse in eccesso (tempo, memoria…)
  - Le funzionalità del figlio non sono più richieste (ma è meglio terminarlo in maniera «ordinata» tramite IPC)
  - Il padre esce prima che il figlio termini (in alcuni sistemi operativi)

Terminazione di processi (2)

- Alcuni sistemi operativi non permettono ai processi figli di esistere se il loro padre ha terminato: tutti i figli di un processo padre terminano quando termina il padre
  - Terminazione in cascata: anche i nipoti, pronipoti… sono terminati
  - La terminazione viene iniziata dal sistema operativo
- Il processo padre può attendere la terminazione di un figlio usando (nei sistemi UNIX e UNIX-like) la chiamata di sistema `wait()`. La chiamata di sistema ritorna informazione di stato e il PID del processo che termina:

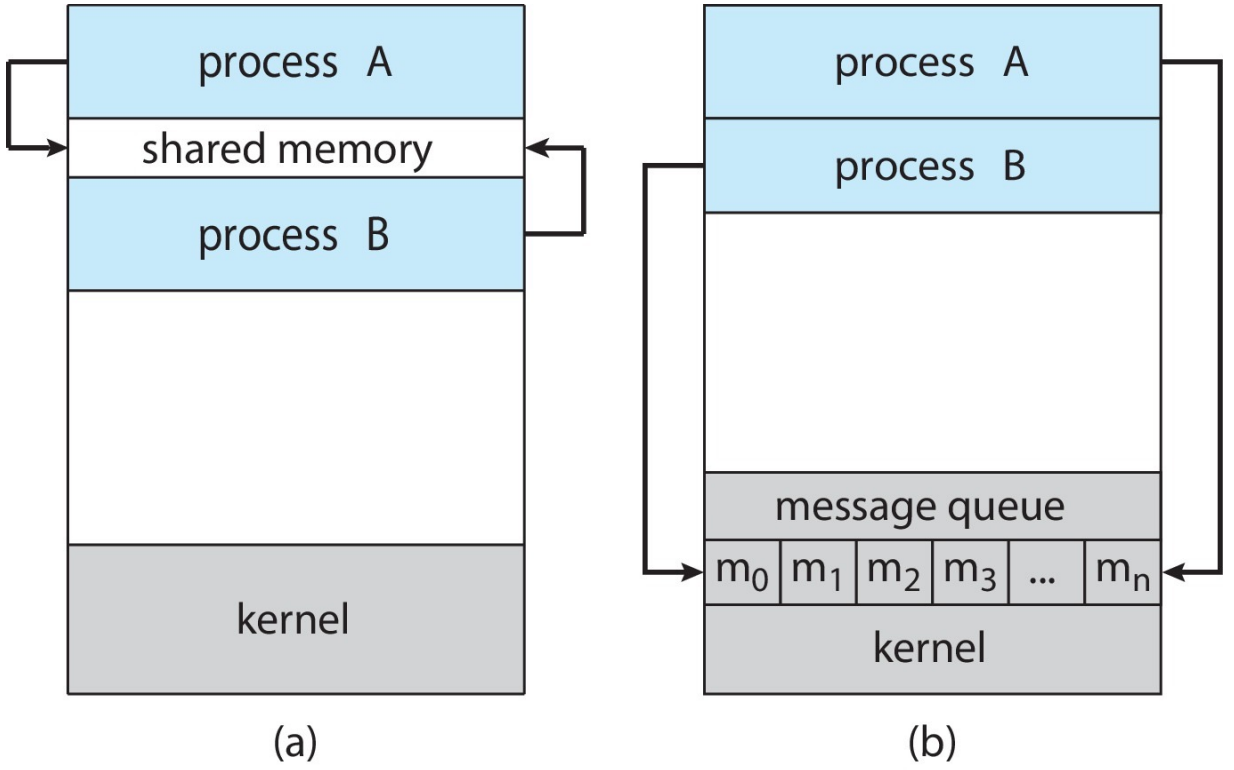
```
pid = wait(&status);
```

- Se un processo termina ma il processo padre non lo sta aspettando (non ha invocato `wait()`) il processo è detto essere **zombie**
- Se il processo padre termina prima del processo figlio, questo è detto essere un **orfano**

Comunicazione interprocesso

- I processi possono essere indipendenti o cooperare
- Un processo coopera se «influenza» o «è influenzato da» uno o più altri processi
- Possibili motivi:
  - Condivisione informazioni
  - Accelerazione computazioni
  - Modularità ed isolamento (come in Chrome)
- Per permettere ai processi di cooperare il sistema operativo deve mettere a disposizione meccanismi di **comunicazione interprocesso** (IPC)
- Due tipi di meccanismi:
  - Memoria condivisa
  - Message passing

Modelli di IPC



IPC tramite memoria condivisa

- Una zona di memoria condivisa tra i processi che intendono comunicare
- La comunicazione è controllata dai processi che comunicano, non dal sistema operativo
- Un problema importante è permettere ai processi che comunicano tramite memoria condivisa di sincronizzarsi
- Allo scopo i sistemi operativi mettono a disposizione primitive per la sincronizzazione dei processi (e dei thread)

IPC tramite message passing

- Permette ai processi *sia* di comunicare *che* di sincronizzarsi
- I processi comunicano tra di loro senza condividere memoria, attraverso la mediazione del sistema operativo
- Questo mette a disposizione:
  - Un'operazione `send(message)` con la quale un processo può inviare un messaggio ad un altro processo
  - Un'operazione `receive(message)` con la quale un processo può (mettersi in attesa fino a) ricevere un messaggio da un altro processo
- Per comunicare due processi devono:
  - Stabilire un **link di comunicazione** tra di loro
  - Scambiarsi messaggi usando `send` e `receive`

Pipe (1)

- Canali di comunicazione tra i processi
- Varianti:
  - Unidirezionale o bidirezionale
  - (se bidirezionale) Half-duplex o full-duplex
  - Relazione tra i processi comunicanti (sono padre-figlio o no)
  - Usabili o meno in rete
- Pipe convenzionali:
  - Unidirezionali
  - Non accessibili al di fuori del processo creatore…
  - …quindi di solito condivise con un processo figlio attraverso una `fork()`
  - In Windows sono chiamate «pipe anonime»

Pipe (2)

- Named pipes:
  - Bidirezionali
  - Esistono anche dopo la terminazione del processo che le ha create
  - Non richiedono una relazione padre-figlio tra i processi che le usano
- In UNIX:
  - Half-duplex
  - Solo sulla stessa macchina
  - Solo dati byte-oriented
- In Windows:
  - Full-duplex
  - Anche tra macchine diverse
  - Anche dati message-oriented