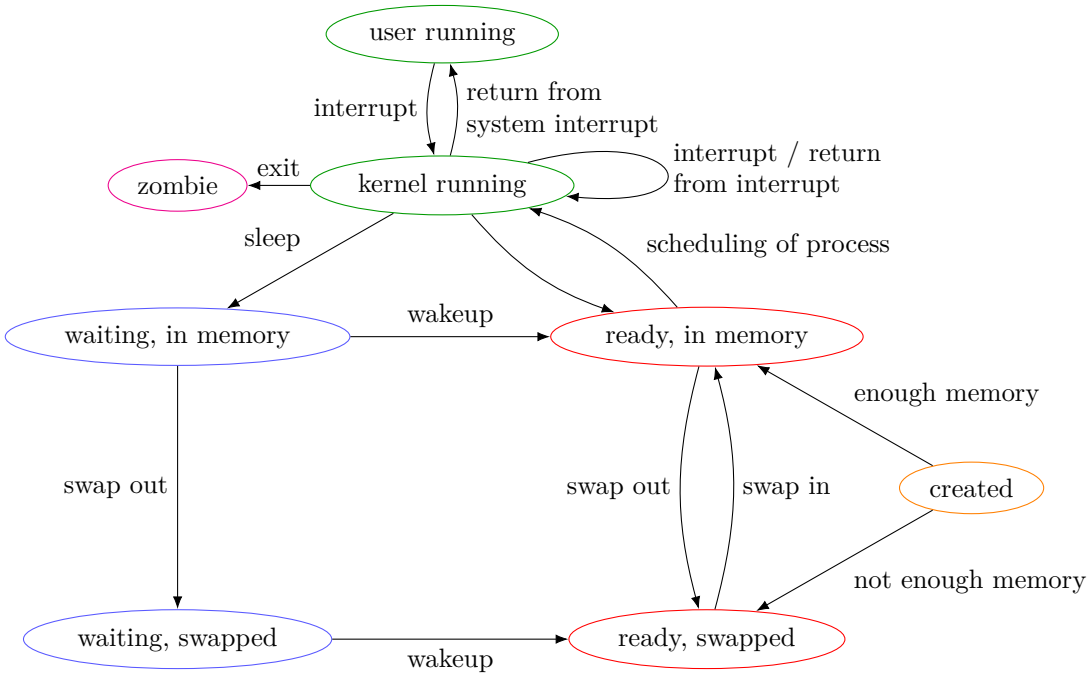
Azzolini Riccardo 2019-10-10

Processi

1 Stati dei processi in UNIX System V

UNIX System V definisce 8 stati in cui i processi si possono trovare:



1.1 Ready e waiting

Ciascuno degli stati ready e waiting è diviso in due sottostati, **in memory** e **swapped**, per la gestione della *memoria virtuale*. Il SO può eseguire più processi di quanti ce ne stiano in RAM: alcuni di essi hanno le aree di testo, dati e stack caricate *interamente in memoria* (stati *in memory*), mentre altri le hanno *interamente* su uno **swap device** (stati *swapped*),[1] un dispositivo logico che in genere corrisponde a una porzione del disco appositamente riservata.

[1] Nei SO moderni, invece, ogni singolo processo è *in parte* sulla RAM e *in parte* sul disco.

Con la memoria virtuale, i processi competono non solo per la CPU, ma anche per la memoria: per eseguire, un processo ha bisogno di entrambe, dato che la CPU opera esclusivamente su istruzioni e dati situati in RAM (e non sul disco).

- Quando serve memoria (solitamente per eseguire altri processi), il SO esegue lo **swapping out**, spostando un certo numero di processi dalla RAM allo swap device. Nella scelta di quali processi "espellere" dalla memoria, si preferiscono quelli waiting, ma, se ciò non basta a liberare spazio sufficiente, possono essere spostati anche processi ready.
- Lo **swapping in**, cioè lo spostamento dallo swap device alla RAM, viene invece effettuato dal SO a certi intervalli di tempo, solo per i processi ready e in base alla quantità di memoria libera.

Siccome è necessario accedere al disco, le operazioni di swapping richiedono molto tempo.

1.2 Running

Lo stato running è diviso in **user running** e **kernel running**. Un processo è user running quando la CPU sta eseguendo il suo codice. Se, nel frattempo, si ha un interrupt (causato da un dispositivo hardware o dal programma stesso), avviene una transizione user running $\rightarrow$ kernel running, e viene eseguito il codice dell'interrupt handler, il quale chiama poi lo scheduler. Nel caso in cui quest'ultimo seleziona nuovamente lo stesso processo, si ha una transizione kernel running $\rightarrow$ user running, altrimenti il processo diventa ready o waiting (a seconda del motivo dell'interruzione), e la CPU viene assegnata a un altro.

L'esecuzione di un interrupt handler può a sua volta essere interrotta da in interrupt di priorità superiore: per mettere in evidenza il fatto che vengono eseguiti handler diversi, si dice che si verifica una transizione kernel running $\rightarrow$ kernel running.

Un processo si trova temporaneamente in stato kernel running anche quando viene schedulato, poiché il SO deve preparare i registri, ecc. per l'esecuzione di tale processo.

1.3 Created e zombie

Quando viene chiesto di eseguire un programma, il SO deve costruire un processo e assegnarlo a tale programma. Durante tale fase di costruzione, il processo è in stato **created**, dopo di che diventa ready:

- in memory se c'è spazio sufficiente in RAM,
- swapped out altrimenti.

Un processo appena creato non può andare in stato di waiting perché è sicuramente pronto a eseguire la sua prima istruzione.

Infine, quando un processo termina, entra nello stato di **zombie**, che serve prevalentemente a raccogliere statistiche sulla sua esecuzione prima che esso venga cancellato completamente. Lo stato zombie si raggiunge sempre a partire da kernel running, perché un processo viene sempre terminato dal SO:

- quando il processo ha finito i suoi compiti, l'ultima istruzione che esegue (su UNIX) è la system call *exit* (aggiunta implicitamente dal compilatore, anche se non scritta nel codice sorgente del programma), che chiede al SO di terminare il programma;
- se il processo si arresta a causa di una situazione anomala (ad esempio, una violazione di memoria), è comunque il SO (per la precisione, un interrupt handler) a terminarlo.

2 Implementazione dei processi

Il SO deve tener traccia di tutti i processi presenti nel sistema. A tale scopo, esso mantiene una struttura dati chiamata **process table**, che contiene, per ogni processo, una voce chiamata **Process Control Block** (**PCB**) o *descrittore di processo*.

La process table è situata nella kernel area della memoria, quindi vi può accedere solo il SO, e non i processi user.

3 Contenuti del Process Control Block

I dati contenuti nel PCB variano in base al SO, ma alcuni devono per forza essere presenti:

- L'identificativo del processo (**PID**, **Process IDentifier**), che è unico per ciascun processo esistente al momento.
- Lo **stato del processo** (running, waiting, ready, ecc.).
- Lo **stato della CPU**, cioè in particolare i valori dei registri, sia generali che di controllo, che vengono salvati quando il processo perde la CPU, in modo da poter essere ripristinati se e quando il processo tornerà running.
- La **priorità**, e/o altri parametri da usare per lo scheduling.
- Dei puntatori a varie zone di memoria, le quali contengono le informazioni necessarie per accedere:
  - all'area di testo;

  - all'area dati;
  - all'area di stack;
  - ai file aperti;
  - ai dispositivi aperti.

  Queste informazioni non sono memorizzate direttamente nel PCB perché hanno dimensioni variabili, mentre la gestione della process table risulta più semplice ed efficiente se i PCB hanno una dimensione fissa.

- Uno o più **PCB pointer** (puntatori ad altri PCB), che permettono la realizzazione delle strutture dati (ad esempio liste/code) usate per gestire lo scheduling, lo sblocco di processi waiting al verificarsi degli eventi, ecc.

3.1 Scheduling con coda dei processi ready

Un esempio di uso dei PCB pointer è l'implementazione di uno scheduler basato su una coda dei processi ready. Tale scheduler assegna sempre la CPU al processo che è ready da più tempo.

Sfruttando i PCB pointer, la realizzazione della coda coda richiede solo due puntatori aggiuntivi (per le due estremità della coda).

Oltre a essere molto semplice, questa politica di scheduling è anche efficiente, perché ha un basso overhead: sia l'operazione di inserimento in coda di un nuovo processo ready che quella di selezione e rimozione dalla coda del processo da schedulare hanno complessità costante.

3.2 Memoria virtuale con paginazione

Nei sistemi moderni, i processi possono essere allocati in RAM in modo *non contiguo*. Ciò viene gestito mediante un meccanismo di **paginazione**.

La memoria fisica è **paginata**, cioè divisa in più **page frame**, tutti della stessa dimensione. Ad esempio, una memoria da 2^{32} byte (4 GB) potrebbe essere divisa in 2^{22} frame da 2^{10} byte (1 kB) ciascuno. Gli indirizzi di memoria avrebbero allora il seguente significato:

- 22 bit più significativi: *numero di pagina* (frame);
- 10 bit meno significativi: *offset* dall'inizio della pagina.

Analogamente, le aree di testo, dati e stack dei processi sono divise in **pagine** della stessa dimensione dei frame (nell'esempio, 2^{10} byte). Ciascuna pagina può essere caricata in *qualsiasi* page frame. Il SO deve quindi tenere traccia di quali sono i page frame occupati da ciascun processo.

Questo è il motivo per cui le informazioni di accesso alla memoria a cui punta il PCB hanno dimensione variabile: un processo più "grande" (che richiede più memoria) ha un elenco di frame occupati più lungo.