Sistemi Operativi

(modulo di Informatica II)

I processi

Patrizia Scandurra

Università degli Studi di Bergamo

a.a. 2008-09

Sommario

- Il concetto di processo
- Schedulazione dei processi
- Operazioni sui processi
- Processi cooperanti

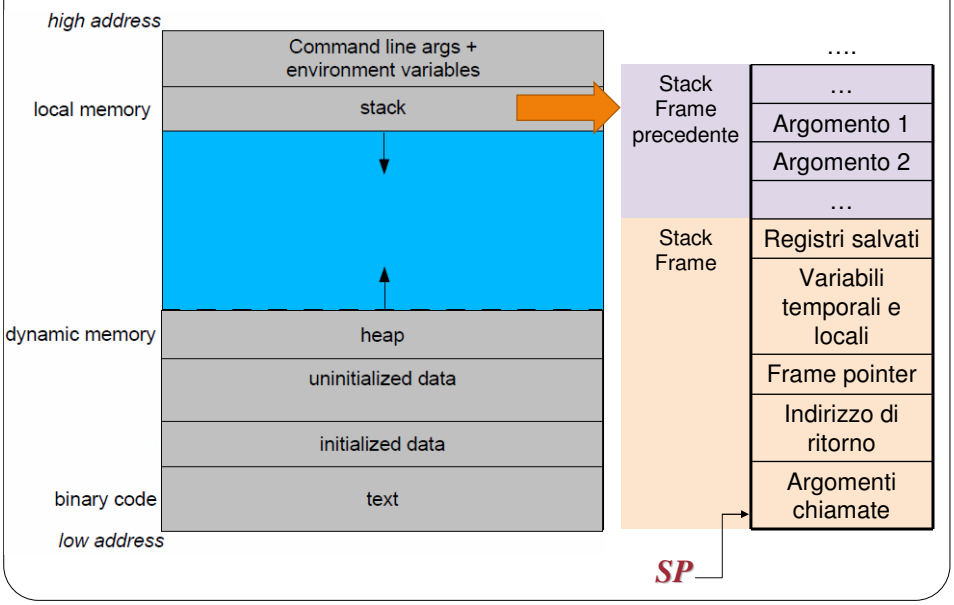
Il concetto di processo (1)

- Un SO esegue una varietà di programmi:
  - Sistema a lotti (batch system) – lavoro (jobs)
  - Sistema a condivisione di tempo (time-sharing) – programmi utente (tasks)
- I termini **job** e **processo** spesso usati come sinonimi
- ***Processo*** – "*un programma in esecuzione*"
  - l'esecuzione del processo deve progredire in modo sequenziale
- Un processo è molto di più ….

Il concetto di processo (2)

- *Componenti*:
  - **Codice** del programma (*code section*)
  - **Dati** del programma
    - Variabili globali in memoria centrale (*data section*)
    - Variabili locali e non locali (stack)
    - Variabili temporanee generate dal compilatore (registri del processore)
    - Variabili allocate dinamicamente (heap)
      - anche se l'allocazione e la deallocazione della memoria all'interno di questo segmento sono a carico del programmatore
  - **Stato di evoluzione della computazione**
    - Program counter
    - Valore delle variabili
    - Informazioni di gestione del contesto della chiamata di procedure (indirizzo di ritorno, frame pointer, stack pointer)

Memory layout di un processo



Programma ≠ Processo

- **Programma**
  - Entità passiva
  - Lista istruzioni
- **Processo**
  - Entità attiva
  - Valori delle variabili
  - Risorse in uso
- Anche se due processi possono essere associati allo stesso programma, essi sono due differenti *istanze di esecuzione* dello stesso codice!

Evoluzione della computazione di un processo

- Il processo è una funzione che trasforma informazioni eseguendo le istruzioni del programma
  - partendo dai valori iniziali
    - eventualmente acquisiti durante l'esecuzione stessa attraverso le periferiche
  - e producendo i risultati finali
    - emessi attraverso le periferiche

Uso della CPU da parte di un processo

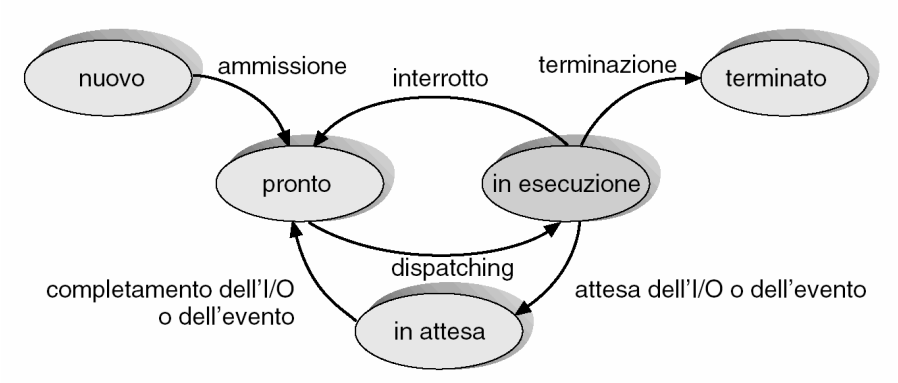
Durante la sua computazione, un processo può:

- **usare il processore**
  - la computazione evolve effettivamente
- **attendere** l'uso del processore, **pur avendo** tutte **le** altre **risorse** informative o fisiche necessarie
  - la computazione potrebbe evolvere, ma non lo fa poiché le istruzioni non possono essere eseguite
- **attendere** che **una risorsa** informativa o fisica diventi disponibile
  - la computazione non può evolvere poiché mancano alcune risorse oltre al processore

Lo stato di un processo

- E' lo **stato di uso del processore** da parte di un processo
- Possibili stati:
  - Nuovo (new): Il processo è stato creato
  - In esecuzione (running): le istruzioni vengono eseguite
  - In attesa (waiting): il processo sta aspettando il verificarsi di qualche evento
  - Pronto all'esecuzione (ready): il processo è in attesa di essere assegnato ad un processore
  - Terminato (terminated): il processo ha terminato l'esecuzione

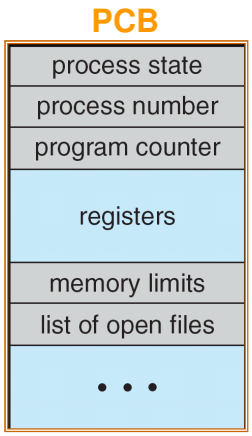
Diagramma degli stati di un processo



Supporti per la gestione dei processi (1)

Process Control Block (PCB)

- Struttura dati del kernel che mantiene le informazioni sul processo
  - Stato del processo
  - Identificatore del processo (Numero)
  - Program counter
  - Registri della CPU
  - Info per la gestione della memoria centrale (limiti di memoria)
  - Info sullo stato dell'I/O (ad es.: file aperti)
  - Info per la schedulazione della CPU
  - Info per l'accounting ecc…



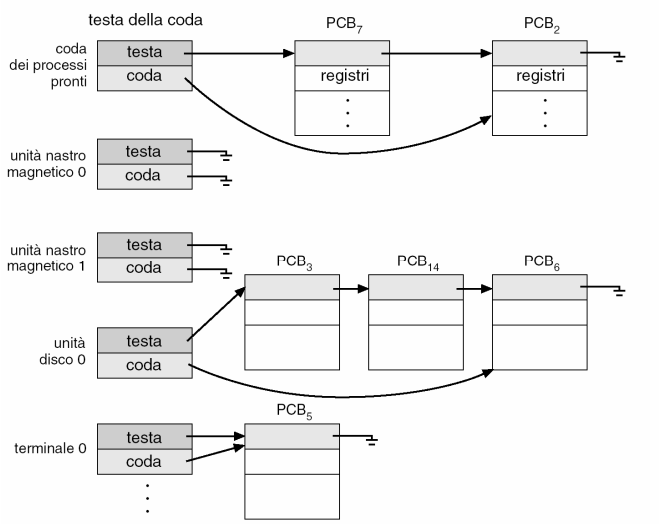
Supporti per la gestione dei processi (2)

Le code di schedulazione dei processi

- Coda di **lavori** *(job queue) – contiene tutti i processi nel sistema*
- Coda dei **processi pronti** *(ready queue) – contiene tutti i processi* che risiedono nella memoria centrale, pronti e in attesa di esecuzione
- Coda della **periferica di I/O** *(device queues) – contiene i processi* in attesa di una particolare periferica di I/O
- Il processo si muove fra le varie code

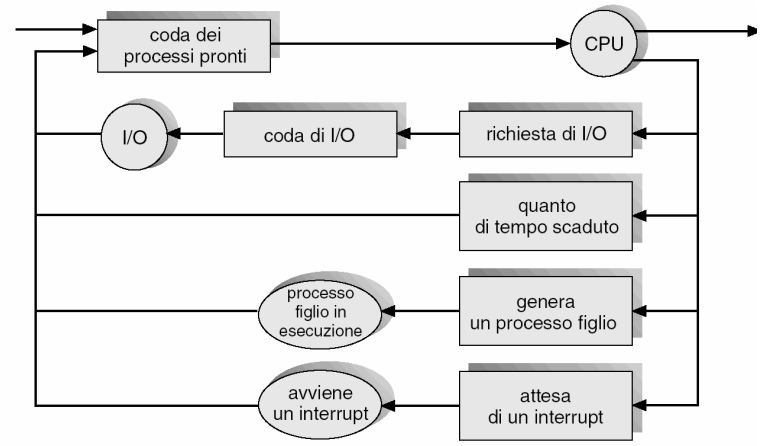
Supporti per la gestione dei processi (2)

Code dei processi nei vari stati



Supporti per la gestione dei processi (3)

Transizioni tra le code



Gli schedulatori (1)

- ***Schedulatore a lungo termine*** *(long-term scheduler)* – seleziona quale processo deve essere inserito nella coda dei processi pronti
- ***Schedulatore a breve termine*** *(Short-term scheduler)* – seleziona quale processo deve essere eseguito e alloca la CPU a uno di essi
- ***Schedulatore a medio termine*** *(Medium-term scheduler)* – *SWAPPING* rimuove un processo dalla memoria centrale e lo pone in memoria di massa
  - Supportato solo da alcuni SO come i *sistemi time-sharing*

Gli schedulatori (2)

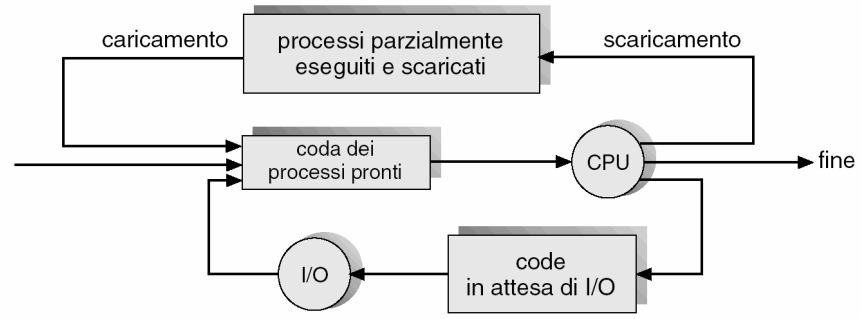


Diagramma delle code con aggiunta dello schedulatore a medio termine

→Rimuove processi dalla memoria centrale: swapping out
→Reintroduce in memoria centrale i processi: swapping in

Gli schedulatori (3)

- Lo schedulatore a breve termine è eseguito molto frequentemente (millisecondi) ⇒ deve essere veloce
- Lo schedulatore a lungo termine è eseguito molto meno frequentemente (secondi, minuti) ⇒ può essere lento
  - Lo schedulatore a lungo termine **controlla il livello di multiprogrammazione**
  - In alcuni **SO può essere assente** (ad es. in sistemi time-sharing, come Unix e MSWindows

Processi CPU-bound e I/O-bound

- I processi possono essere classificati come:
  - **processo I/O-bound** – processo che spende più tempo facendo I/O che elaborazione
    - molti e brevi utilizzi di CPU
  - **processo CPU-bound** – processo che spende più tempo facendo elaborazione che I/O
    - pochi e lunghi utilizzi di CPU
- Un sistema di buone prestazioni presenta una combinazione bilanciata di processi CPU-bound e I/O-bound
- Lo swapping può essere necessario per migliorare la distribuzione dei processi tra le due tipologie
  - oppure perché un cambiamento di richieste della memoria centrale ha superato la memoria disponibile

Il cambio di contesto (1)

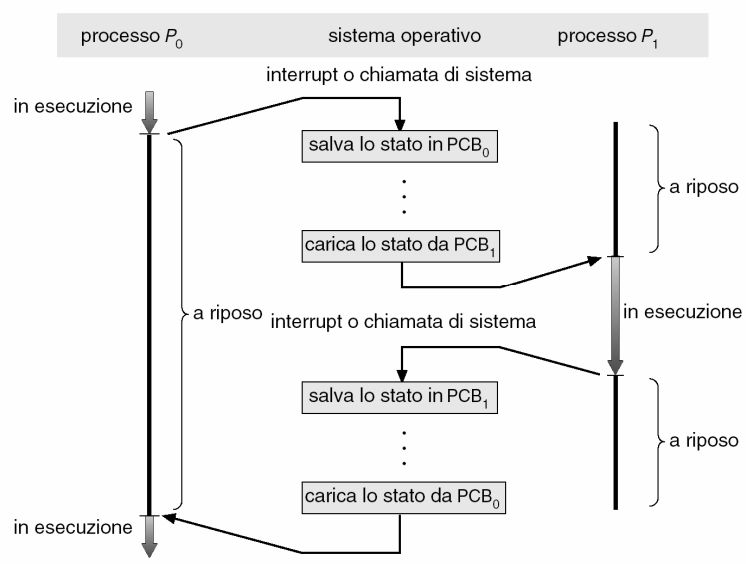
context switch

- E' il **passaggio della CPU da un processo ad un altro**

  Context Switch = ***sospensione del processo in esecuzione*** + ***caricamento del nuovo processo da mettere in esecuzione***

- Il tempo per il cambio di contesto è puro **tempo di gestione del sistema**
  - poichè durante il cambio non vengono compiute operazioni utili per la computazione dei processi
- I tempi per i cambi di contesto dipendono sensibilmente dal supporto hardware
  - tipicamente inferiore ai 10 millisecondi

Il cambio di contesto (2)



Processi come flussi di operazioni

- Processo = flusso di esecuzione di computazione
- Flussi separati = processi separati
  - Come si generano?
  - Come un programma si trasforma in un insieme di processi?
  - Come interagiscono i processi?
- Flussi sincronizzati
  → processi evolvono sincronizzandosi
- Flussi indipendenti
  → processi evolvono autonomamente

Modellazione della computazione a processi

- Modelli di computazione:
  - Processo monolitico
  - Processi cooperanti
- Modelli di realizzazione del codice eseguibile:
  - Programma monolitico
  - Programmi separati
- Realizzazione dei modelli di computazione:
  - Programma monolitico è eseguito come processo monolitico
  - Programma monolitico genera processi cooperanti
  - Programmi separati sono eseguiti come processi cooperanti (ed eventualmente generano ulteriori processi cooperanti)

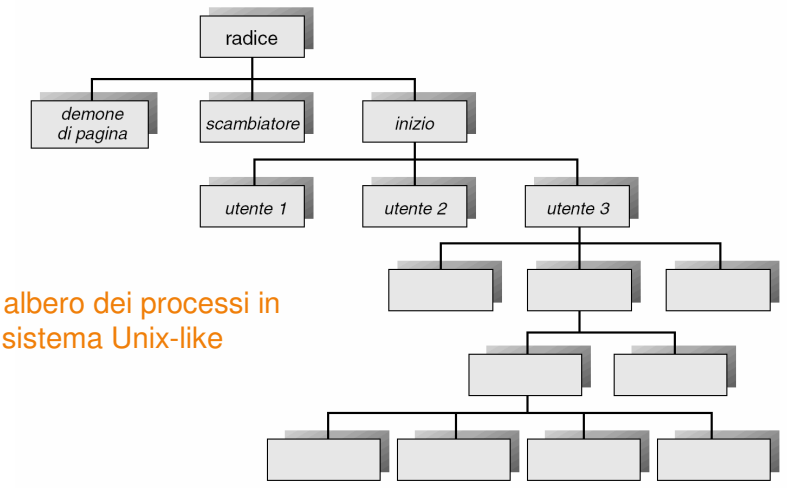
Creazione (o generazione) di un processo

- Il processo in esecuzione invoca una chiamata di sistema che crea e attiva un nuovo processo
  - Ad es. in Unix POSIX la chiamata *fork()*

  - Processo generante → processo padre
  - Processo generato → processo figlio

Albero dei processi

Il processo padre crea processi figli, i quali a loro volta creano altri processi formando un albero di processi



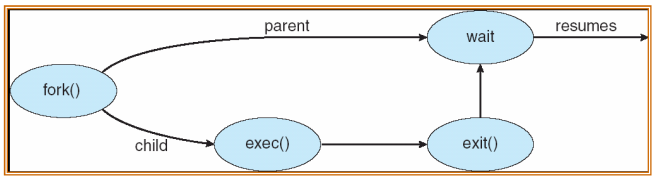
Es. albero dei processi in un sistema Unix-like

Risorse dei processi

- Condivise col padre
- Parzialmente condivise col padre
- Indipendenti dal padre (ottenute dal sistema)
- Passaggio dei dati di inizializzazione

Esecuzione dei processi

- Due scenari possibili:
  - Il padre continua l'esecuzione in modo concorrente ai figli
    - modalità asincrona
  - Il padre attende finchè tutti (o alcuni) i suoi figli sono terminati
    - modalità sincrona



POSIX

Spazio di indirizzamento (1)

- Lo spazio di indirizzamento del processo figlio è sempre **distinto** da quello del processo padre
- Due possibili scenari:
  - Il figlio è un **duplicato del padre**
    - stesso programma
    - stessi dati all'atto della creazione
  - Il figlio ha un **nuovo programma** caricato nel proprio spazio di indirizzamento

Esempio in UNIX

La chiamata di sistema *exec, usata dopo la fork, carica un nuovo* programma nello spazio di memoria del processo che la esegue

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

int main(int argc, char *argv[])
{
int pid;

   /* genera un altro processo */
   pid = fork();

   if (pid < 0) { /* si è verificato un errore */
       fprintf(stderr, "Fork fallita");
       exit(-1);
   }
   else if (pid == 0) { /*  processo figlio */
       execlp("/bin/ls","ls",NULL);
   }
```

Terminazione di un processo (1)

- Terminazione normale dopo l'ultima istruzione tramite la chiamata *exit*
  - se "figlio" può restituire un **valore di stato** (di solito un intero) al "padre"
    - Nell'es. tramite la chiamata di sistema *wait()*
  - le risorse del processo sono deallocate dal SO

```
    }
    else if (pid == 0) { /*  processo figlio */
        execlp("/bin/ls","ls",NULL);
    }
    else { /* processo padre */
        /* il processo padre attenderà il completamento del figlio */
        wait(NULL);
        printf("Figlio terminato");
        exit(0);
    }
}
```

Terminazione di un processo (2)

- Terminazione in caso di anomalia (aborto)

- Il padre può terminare l'esecuzione di uno dei suoi figli per varie ragioni:
  - Eccessivo uso di una risorsa
  - Compito non più necessario
  - Terminazione a cascata
    - se il padre sta terminando, alcuni SO non permettono ad un processo figlio di proseguire

Processi cooperanti

- Un ***processo indipendente*** *non può influenzare o essere* influenzato dagli altri processi in esecuzione
- Un ***processo cooperante*** *può influenzare o essere influenzato* da altri processi in esecuzione nel sistema

Cooperazione

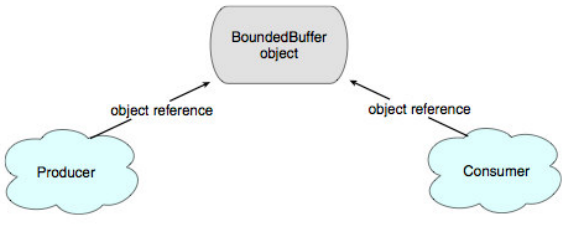
- **Cooperazione** = lavoro congiunto di processi per raggiungere scopi applicativi comuni con condivisione e scambio di informazioni
- **Vantaggi** del processo di cooperazione:
  - condivisione delle informazioni (ad es. un file o una directory condivisa)
  - velocizzazione della computazione (possibile in verità solo con più CPU o canali di I/O)
  - modularità (dividendo le funzioni del sistema in processi o thread separati)
  - convenienza (un singolo utente può lavorare su molte attività)

Il problema del produttore-consumatore (1)

- Modello molto comune nei processi cooperanti:
  - un processo ***produttore*** *genera informazioni*
  - *che sono utilizzate da un* processo ***consumatore***
- Un oggetto temporaneo in memoria (**buffer**) può essere riempito dal produttore e svuotato dal consumatore
  1. *buffer illimitato (unbounded-buffer): non c'è un limite teorico alla* dimensione del buffer
     - Il consumatore deve aspettare se il buffer è vuoto
     - Il produttore può sempre produrre
  2. *buffer limitato (bounded-buffer): la dimensione del buffer è fissata*
     - *Il consumatore deve aspettare se il buffer è vuoto*
     - *Il produttore deve aspettare se il buffet è pieno*

Il problema del produttore-consumatore (2)

- Il buffer può essere
  - fornito dal SO tramite l'uso di funzionalità di comunicazione tra processi ***InterProcess Communication (IPC)***
  - oppure essere esplicitamente scritto dal programmatore dell'applicazione con l'uso di **memoria condivisa**
- Simuliamo in Java il problema permettendo ai processi di condividere il buffer tramite il passaggio di riferimento



Buffer limitato – Interfaccia per le implementazioni del buffer in Java

```
public interface Buffer
{
   //  i produttori chiamano questo metodo
   public abstract void insert(Object item);

   // i consumatori chiamano questo metodo
   public abstract Object remove();
}
```

Bounded-buffer in Java – memoria condivisa

```
import java.util.*;

public class BoundedBuffer implements Buffer
{
   private static final int BUFFER_SIZE = 5;
   private int count; // numero di elementi nel buffer
   private int in; // punta alla successiva posizione libera
   private int out; // punta alla successiva posizione piena
   private Object[] buffer;

   public BoundedBuffer() {
      // il buffer è inizialmente vuoto
      count = 0;
      in = 0;
      out = 0;

      buffer = new Object[BUFFER_SIZE];
   }

   //i produttori chiamano questo metodo
   public void insert(Object item) {
      // Figura 4.11
   }

   // i consumatori chiamano questo metodo
   public Object remove() {
      // Figura 4.12
   }

}
```

Bounded-buffer in Java – memoria condivisa: insert()

```
public void insert(Object item) {
   while (count == BUFFER_SIZE)
      ; //  non fare nulla - non ci sono buffer liber

   // aggiungi un elemento al buffer
   ++count;
   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
}
```

Bounded-buffer in Java – memoria condivisa: remove()

```
public Object remove() {
   Object item;

   while (count == 0)
      ; // non fare nulla - nulla da consumare

   // rimuovi un elemento dal buffer
   --count;
   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;

   return item;
}
```