Concetto di processo

Il processo è un programma in esecuzione

- È l'**unità di esecuzione** all'interno del SO
- Solitamente, esecuzione **sequenziale** (istruzioni vengono eseguite in sequenza, secondo l'ordine specificato nel testo del programma)
- SO multiprogrammato consente ***l'esecuzione concorrente di più processi***

D'ora in poi faremo implicitamente riferimento sempre al caso di SO multiprogrammati

Concetto di processo

Programma = entità passiva

Processo = entità attiva

Il processo è rappresentato da:

- **codice** (*text*) del programma eseguito
- **dati:** variabili globali
- **program counter**
- alcuni **registri** di CPU
- **stack**: parametri, variabili locali a funzioni/procedure

Concetto di processo

Processo = {PC, registri, stack, text, dati}

Inoltre, a un processo possono essere associate delle ***risorse di SO***. Ad esempio:

- file aperti
- connessioni di rete
- altri dispositivi di I/O in uso
- ...

Stati di un processo

Un processo, durante la sua esistenza può trovarsi in vari **stati**:

- Init: ***stato transitorio*** durante il quale il processo viene caricato in memoria e SO inizializza i dati che lo rappresentano
- Ready: processo è ***pronto per acquisire la CPU***
- Running: processo ***sta utilizzando la CPU***
- Waiting: processo è ***sospeso in attesa di un evento***
- Terminated: ***stato transitorio*** relativo alla fase di terminazione e deallocazione del processo dalla memoria

Stati di un processo

Transizioni di stato:

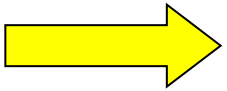
Stati di un processo

In un sistema ***monoprocessore e multiprogrammato***:

- ***un solo processo*** (al massimo) si trova nello ***stato running***
- più processi possono trovarsi negli ***stati ready e waiting***

  necessità di ***strutture dati per mantenere in memoria le informazioni su processi in attesa***

  - di acquisire la CPU (ready)
  - di eventi (waiting)



Descrittore di processo

Rappresentazione dei processi

Come vengono rappresentati i processi in SO?

- Ad ogni processo viene associata una struttura dati (descrittore): **Process Control Block** (**PCB**)
- **PCB** contiene tutte le ***informazioni relative al processo***:
  - Stato del processo; - Program counter
  - Contenuto dei registri di CPU (SP, IR, accumulatori, ...)
  - Informazioni di scheduling (priorità, puntatori alle code, ...)
  - Informazioni per gestore di memoria (registri base, limite, ...)
  - Informazioni relative all'I/O (risorse allocate, file aperti, ...)
  - Informazioni di accounting (tempo di CPU utilizzato, ...)
  - ...

Process Control Block

Il sistema operativo gestisce i PCB di tutti i processi, organizzandoli in opportune strutture dati (ad esempio *code* di processi)

Scheduling dei processi

È l'attività mediante la quale SO effettua delle scelte tra i processi, riguardo a:

- ***caricamento in memoria centrale***
- ***assegnazione della CPU***

Scheduler

È quella parte del SO che si occupa della selezione dei processi a cui assegnare la CPU

Nei sistemi time sharing, allo scadere di ogni quanto di tempo, SO:

- decide ***a quale processo*** assegnare la CPU
  (scheduling di CPU)
- effettua il ***cambio di contesto (context switch)***

Cambio di contesto

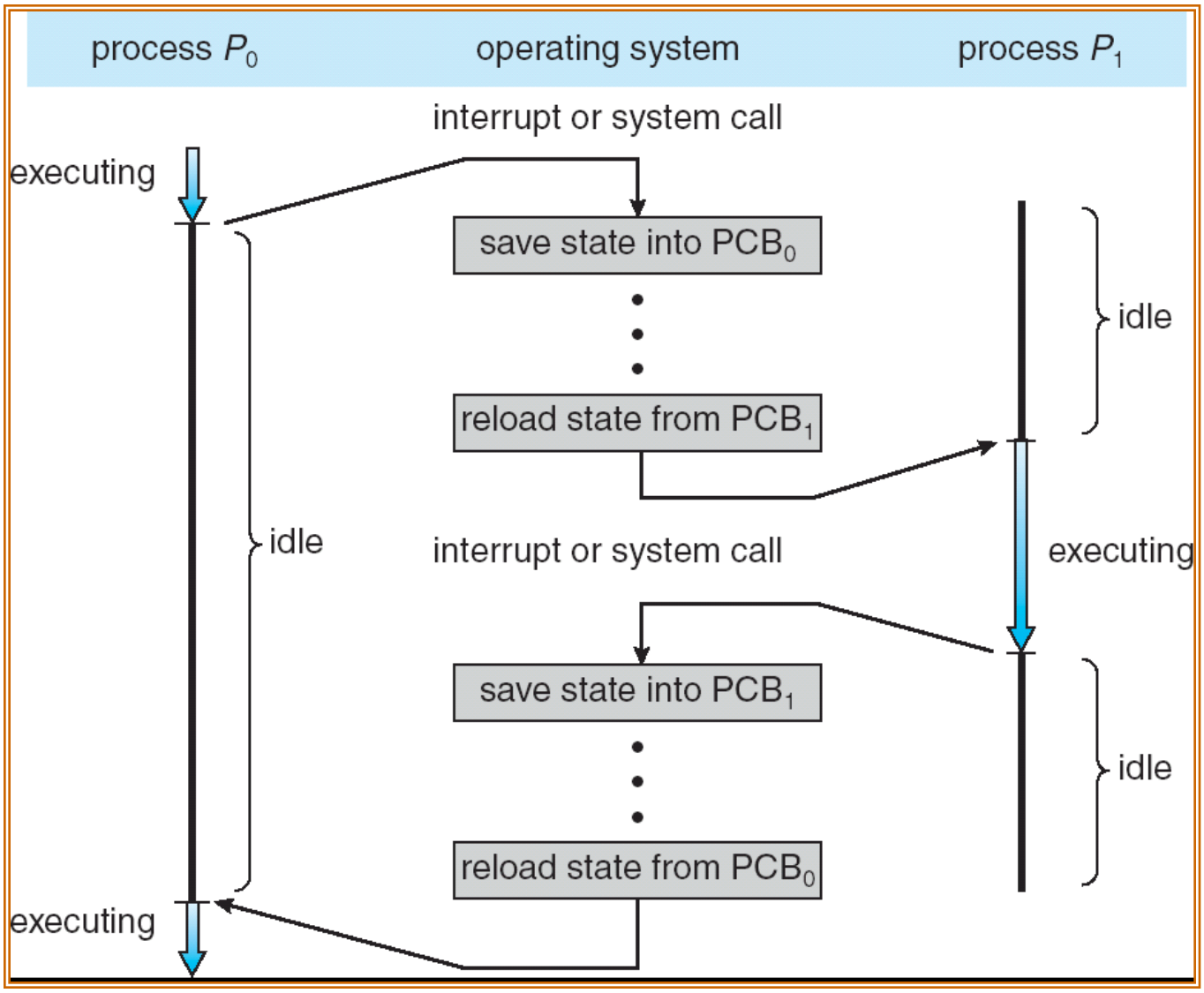
È la fase in cui ***l'uso della CPU viene commutato da un processo ad un altro***

Quando avviene un **cambio di contesto** tra un processo P_i ad un processo P_{i+1} (ovvero, P_i cede l'uso della CPU a P_{i+1}):

- **Salvataggio dello *stato* di P_i**: SO copia PC, registri, ... del processo ***deschedulato*** P_i nel suo PCB
- **Ripristino dello *stato* di P_{i+1}**: SO trasferisce i dati del processo P_{i+1} dal suo PCB nei registri di CPU, che può così riprendere l'esecuzione

➔ Il passaggio da un processo al successivo può richiedere ***onerosi trasferimenti da/verso la memoria secondaria***, per allocare/deallocare gli spazi di indirizzi dei processi (vedi gestione della memoria)

Cambio di contesto



Scheduler

Lo scheduler gestisce

- la **coda dei processi pronti:** contiene i PCB dei processi che si trovano in stato *Ready*

Altre strutture dati necessarie

- **code di waiting** (una per ogni tipo di attesa: dispositivi I/O, timer, ...): ***ognuna di esse contiene i PCB dei processi waiting*** in attesa di un evento del tipo associato alla coda

Scheduling e cambio di contesto

Operazioni di scheduling determinano un costo computazionale aggiuntivo che dipende essenzialmente da:

- **frequenza** di cambio di contesto
- dimensione **PCB**
- costo dei trasferimenti da/verso la memoria

  ➔esistono SO che prevedono **processi *leggeri*** (***thread***) che hanno la proprietà di condividere codice e dati con altri processi:

  - ☒ **dimensione PCB ridotta**
  - ☒ **riduzione overhead**

Interazione tra processi

I processi, pesanti o leggeri, *possono* interagire

Classificazione

- **processi indipendenti**: due processi P1 e P2 sono indipendenti se l'esecuzione di P1 non è influenzata da P2, e viceversa
- **processi interagenti**: P1 e P2 sono interagenti se l'esecuzione di P1 è influenzata dall'esecuzione di P2, e/o viceversa

Processi interagenti

Tipi di interazione

- **Cooperazione:** l'interazione consiste nello ***scambio di informazioni*** al fine di eseguire un'attività comune
- **Competizione:** i processi interagiscono per ***sincronizzarsi nell'accesso a risorse comuni***
- **Interferenza:** ***interazione*** non desiderata e potenzialmente ***deleteria*** tra processi

Processi interagenti

Supporto all'interazione

L'interazione può avvenire mediante

- **memoria condivisa** (modello ad *ambiente globale*): SO consente ai processi (***thread***) di condividere variabili; l'interazione avviene tramite l'accesso a ***variabili condivise***
- **scambio di messaggi** (modello ad *ambiente locale*): i processi non condividono variabili e interagiscono mediante meccanismi di trasmissione/ricezione di messaggi; SO prevede dei meccanismi a supporto dello ***scambio di messaggi***

Processi interagenti

Aspetti

- ***concorrenza → velocità***
- suddivisione dei compiti tra processi
  → ***modularità***
- ***condivisione di informazioni***
  - assenza di replicazione: ogni processo accede alle stesse istanze di dati
  - necessità di ***sincronizzare i processi*** nell'accesso a dati condivisi

Processi UNIX

UNIX è un sistema operativo *multiprogrammato a divisione di tempo*:

unità di computazione è il processo

Caratteristiche del processo UNIX:

- **processo pesante con codice *rientrante***
  - ✓ ***dati non condivisi***
  - ✓ ***codice condivisibile* con altri processi**
- funzionamento dual mode
  - ✓ processi di utente **(modo *user*)**
  - ✓ processi di sistema **(modo *kernel*)**
  - ☞ diverse potenzialità e, in particolare, diversa visibilità della memoria

Scripting: realizzazione file comandi

File comandi

Shell è un ***processore comandi*** in grado di interpretare ***file sorgenti in formato testo e contenenti comandi ➔ file comandi (script)***

Linguaggio comandi (vero e proprio linguaggio programmazione)

- Un *file comandi* può comprendere
  - ***statement per il controllo di flusso***
  - ***variabili***
  - ***passaggio dei parametri***

NB:

- ***quali statement*** sono disponibili dipende da ***quale shell*** si utilizza
- file comandi viene ***interpretato*** (non esiste una fase di compilazione)
- file ***comandi deve essere eseguibile*** (usare **`chmod`**)

Scelta della shell

La prima riga di un file comandi deve specificare ***quale shell si vuole utilizzare***: **`#! <shell voluta>`**

- Es: **`#!/bin/bash`**
- # è visto dalla shell come un commento ma…
- #! è visto da SO come identificatore di un file di script

  SO capisce così che l'interprete per questo script sarà **`/bin/bash`**

- Se questa riga è assente viene scelta la shell di preferenza dell'utente

File comandi

È possibile memorizzare ***sequenze di comandi all'interno di file eseguibili***:

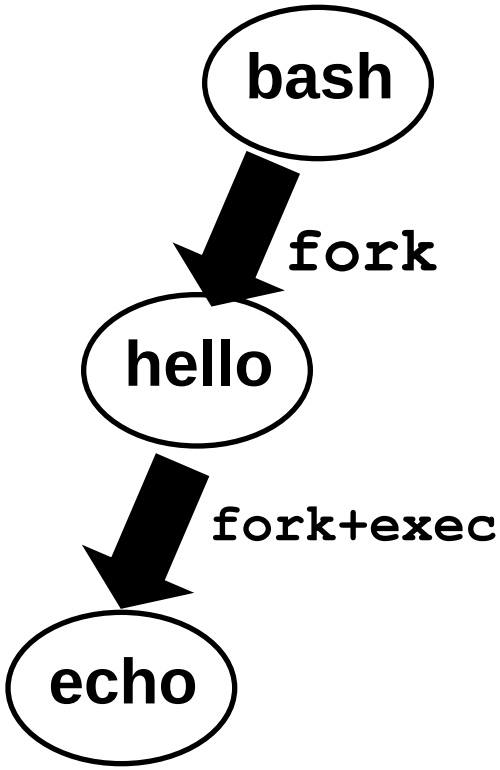
file comandi (script)

Ad esempio:

```
#!bin/bash
echo hello world!
```

file hello

```
bash-2.05$ hello
hello world!
```



Passaggio parametri

```
./nomefilecomandi  arg1  arg2  ...  argN
```

Gli argomenti sono ***variabili posizionali*** nella linea di invocazione contenute nell'ambiente della shell

- **$0** rappresenta il comando stesso
- **$1** rappresenta il primo argomento ...

- è possibile far scorrere tutti gli argomenti verso sinistra

  ➔ **`shift`**

  $0 non va perso, solo gli altri sono spostati ($1 perso)

	$0	$1	$2
prima di **shift**	DIR	-w	/usr/bin
dopo **shift**	DIR	/usr/bin	

- è possibile riassegnare gli argomenti ➔ **`set`**
  - **`set exp1 exp2 exp3 ...`**
  - gli argomenti sono assegnati secondo la posizione

Altre informazioni utili

Oltre agli argomenti di invocazione del comando

- **`$*`** insieme di ***tutte le variabili posizionali***, che corrispondono arg del comando: $1, $2, ecc.
- **`$#`** ***numero di argomenti*** passati (***$0 escluso***)
- **`$?`** valore (int) restituito dall'ultimo comando eseguito
- **`$$`** id numerico del processo in esecuzione (pid)

Semplici forme di input/output

- **`read`** **`var1 var2 var3`** **`#input`**
- **`echo`** **`var1 vale $var1 e var2 $var2`** **`#output`**
  - **read** la stringa in ingresso viene attribuita alla/e variabile/i secondo corrispondenza posizionale

Strutture di controllo

Ogni comando in uscita restituisce un ***valore di stato***, che indica il suo ***completamento o fallimento***

Tale valore di uscita è posto nella variabile **?**

- **$?** può essere riutilizzato in espressioni o per controllo di flusso successivo

Stato vale usualmente:

- zero: comando OK
- valore positivo: errore

Esempio

```
host203-31:~ paolo$ cp  a.com  b.com
cp: cannot access a.com
host203-31:~ paolo$ echo $?
2
```

test

Comando per la ***valutazione di una espressione***

- **`test -<opzioni> <nomefile>`**

Restituisce uno stato uguale o diverso da zero

- valore ***zero*** ➔ ***true***
- valore ***non-zero*** ➔ ***false***

ATTENZIONE: convenzione opposta rispetto al linguaggio C!

- Motivo: i codici di errore possono essere più di uno e avere significati diversi

Alcuni tipi di test

`test`

- **`-f <nomefile>`** esistenza di file
- **`-d <nomefile>`** esistenza di direttori
- **`-r <nomefile>`** diritto di lettura sul file (**`-w`** e **`-x`**)
- **`test <stringa1> = <stringa2>`** uguaglianza stringhe
- **`test <stringa1> != <stringa2>`** diversità stringhe

ATTENZIONE:

- gli **spazi intorno a `=`** (o a `!=`) sono ***necessari***
- stringa1 e stringa2 possono contenere metacaratteri (attenzione alle espansioni)
- **`test -z <stringa>`** vero se ***stringa nulla***
- **`test <stringa>`** vero se ***stringa non nulla***

Strutture di controllo: alternativa

```
if <lista-comandi>
  then
     <comandi>
  [elif <lista_comandi>
   then <comandi>]
  [else <comandi>]
  fi
```

ATTENZIONE:

- le parole chiave (do, then, fi, ...) devono essere o ***a capo o dopo il separatore ;***
- if controlla il valore in uscita ***dall'ultimo comando di <lista-comandi>***

Esempio

```
# fileinutile
# risponde "sì" se invocato con "sì" e un numero
  < 24
if test $1 = sì -a $2 -le 24
  then echo sì
  else echo no
fi
```

```
#test su argomenti
if test $1; then echo OK
  else echo Almeno un argomento
fi
```

Alternativa multipla

```
# alternativa multipla sul valore di var
case <var> in
   <pattern-1>)
       <comandi>;;
   ...
   <pattern-i> | <pattern-j> | <pattern-k>)
       <comandi>;;
   ...
   <pattern-n>)
       <comandi> ;;
esac
```

Importante: nell'alternativa multipla si possono usare metacaratteri per fare pattern-matching (non sono i "soliti" metacaratteri su nome di file)

Esempi

```
read  risposta
case  $risposta in
  S* | s* | Y* | y* ) <OK>;;
  * ) <problema>;;
esac
```

```
# append: invocazione  append [dadove] adove
  case $# in
      1) cat >> $1;;
      2) cat < $1 >> $2;;
      *) echo  uso: append [dadove] adove;
         exit 1;;
  esac
```

Cicli enumerativi

```
for <var> [in <list>] # list=lista di stringhe
do
   <comandi>
done
```

- scansione della lista <list> e ***ripetizione del ciclo per ogni stringa presente nella lista***
- scrivendo solo `for i` si itera con valori di **i in $***

Esempi

- **`for i in *`**
  - esegue per tutti i file nel direttorio corrente
- **`for i in \`ls s*\``**

  `do <comandi>`

  `done`
- **`for i in \`cat file1\``**

  `do <comandi per ogni parola del file `*`file1`*`>`

  `done`
- **`for i in 0 1 2 3 4 5 6`**

  `do`

  `    echo $i`

  `done`

Ripetizioni non enumerative

```
while <lista-comandi>
do
   <comandi>
done
```

Si ripete per tutto il tempo che il valore di stato dell'ultimo comando della lista è zero (successo)

```
until <lista-comandi>
do
   <comandi>
done
```

Come while, ma inverte la condizione

Uscite anomale

- vedi C: **continue**, **break** e **return**
- **exit** [**status**]: system call di UNIX, anche comando di shell

Esempi di file comandi

Esercizio da svolgere in lab (o a casa):

- scrivere un file comandi che ogni 5 secondi controlli se sono stati ***creati o eliminati file in una directory***. In caso di cambiamento, si deve visualizzare un messaggio su stdout (quanti file sono presenti nella directory)
- il file comandi deve poter essere invocato con ***uno e un solo parametro***, la directory da porre sotto osservazione (→ fare opportuno controllo dei parametri)

Suggerimento: uso di un file temporaneo, in cui tenere traccia del numero di file presenti al controllo precedente

Esempi di file comandi: soluzione

```
echo 1 > loop.$$.tmp

while :
do
  sleep 5s
  if [ `ls $1 | wc –w` -ne `cat loop.$$.tmp` ]
  then
     ls $1 | wc –w > loop.$$.tmp
     echo in $1 sono presenti `cat
          loop.$$.tmp` file
  fi
done
```