Programmazione e Architetture degli Elaboratori - Foglio 4

Luca Manzoni, Michele Rispoli, Pietro Morichetti

Esercizio 01

Si consideri un sistema di scheduling per i processi: P_1, P_2, P_3, P_4, P_5 con le seguenti caratteristiche di tempi di esecuzioni ed ordine di priorità (ordinamento crescente, quindi priorità 1 è la più alta mentre priorità 5 è la più bassa); si faccia riferimento alla tabella riportata qui di seguito.

Processo N°	**Tempo di Esecuzione**	**Priorità**	**Tempo di Arrivo (Caso A)**	**Tempo di Arrivo (Caso B)**
P1	10	4	0	0
P2	1	5	0	1
P3	2	1	0	4
P4	1	3	0	6
P5	5	2	0	7

Dove nel caso di tempo di arrivo del caso A, i processi arrivano nell'ordine: P_1, P_2, P_3, P_4, P_5; si svolgano le seguenti richieste:

1. Si illustri [Caso A], l'ordine di esecuzione in caso si segua la politica SJF (Shortest Job First); si calcoli media e deviazione standard del tempo di attesa.

   Sol:

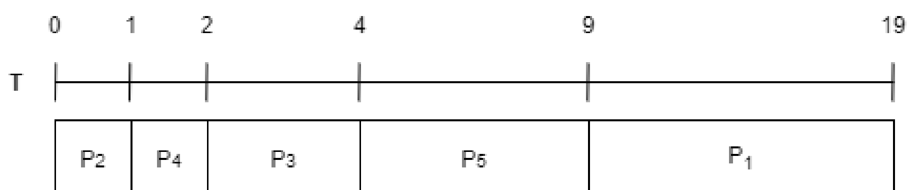


Figure 1: scheduling a base SJF

$$\mu_{exec} = \frac{10+1+2+1+5}{5} = 3.8$$
$$\mu_{wait} = \frac{9+0+2+1+4}{5} = 3.2$$
$$\sigma = 3.18$$

2. Si illustri [Caso A] l'ordine di esecuzione in caso si segua la politica di priorità non preemptiva; si calcoli media e deviazione standard del tempo di attesa.

   Sol:

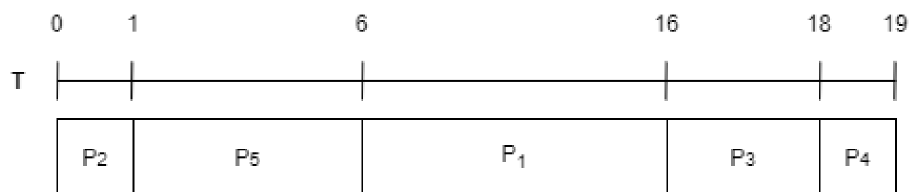


Figure 2: scheduling a base prioritaria

$$\mu_{exec} = \frac{10+1+2+1+5}{5} = 3.8$$
$$\mu_{wait} = \frac{6+0+16+18+1}{5} = 8.2$$
$$\sigma = 7.49$$

3. Si illustri [Caso B], per mezzo di un diagramma di Gannt, l'ordine di esecuzione in caso si segua la politica FCFS (Firs-Come First-Served); si calcoli media e deviazione standard del tempo di attesa.

   Sol:

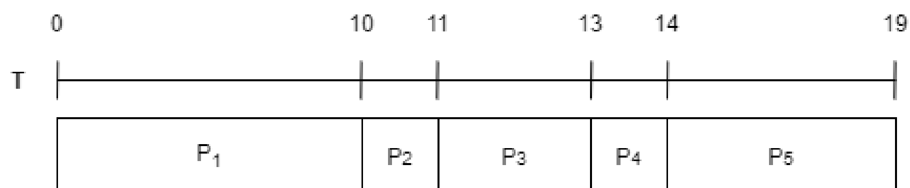


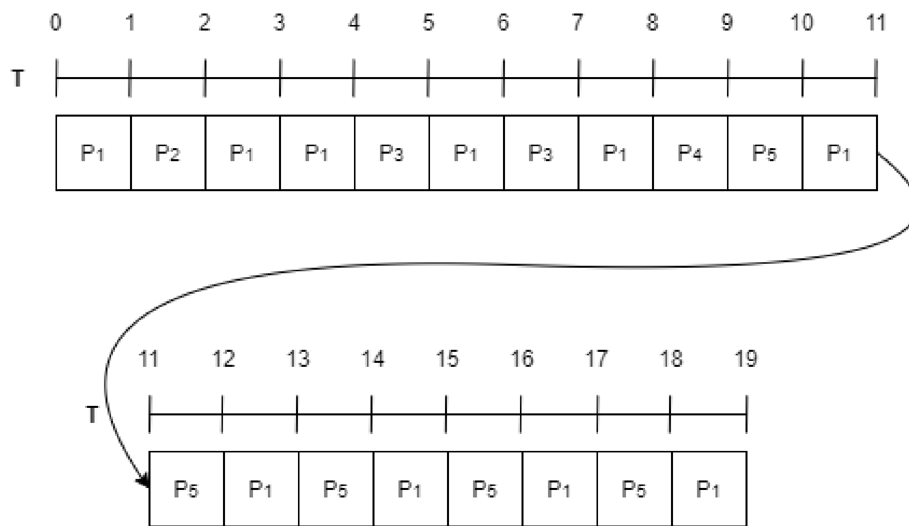
Figure 3: scheduling a base FCFS

$$\mu_{exec} = \frac{10+1+2+1+5}{5} = 3.8$$

$$\mu_{wait} = \frac{0 + 10 + 11 + 13 + 14}{5} = 9.6$$
$$\sigma = 5.00$$

4. Si illustri [Caso B], per mezzo di un diagramma di Gannt, l'ordine di esecuzione in caso si segua la politica RR (Round Robin) con quantum pari ad 1 (ossia una unità di tempo di esecuzione) e quantum pari a 2.

   Sol:



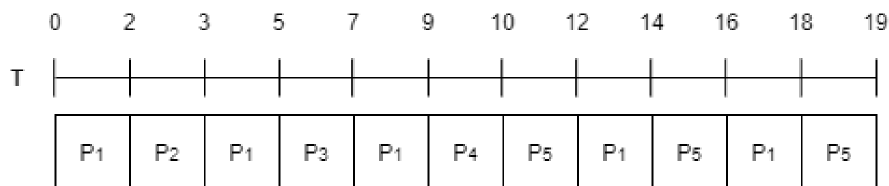
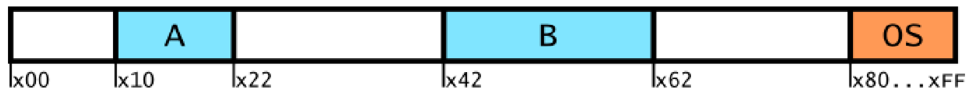


Figure 4: scheduling a base RR

Esercizio 02

Un calcolatore che dispone di una memoria con ben 256 possibili indirizzi (da 0x00 a 0xFF) si trova, ad un certo punto, con la seguente configurazione in memoria:



1. Quanti blocchi sono allocati rispettivamente per il processo A, il processo B ed il sistema operativo?

   Sol: Effettuando i calcoli in esadecimale otteniamo:

$$M(A) = \langle 22 \rangle_{16} - \langle 10 \rangle_{16} = \langle 12 \rangle_{16} = \langle 18 \rangle_{10} \text{ blocchi}$$

$$M(B) = \langle 62 \rangle_{16} - \langle 42 \rangle_{16} = \langle 20 \rangle_{16} = \langle 32 \rangle_{10} \text{ blocchi}$$

$$M(OS) = \langle FF \rangle_{16} - \langle 80 \rangle_{16} + 1 = \langle 80 \rangle_{16} = \langle 128 \rangle_{10} \text{ blocchi}$$

2. Dei nuovi processi richiedono all'OS di essere allocati in memoria (in questo ordine):

   (a) Il processo C richiede 10 blocchi
   (b) Il processo D richiede 20 blocchi
   (c) Il processo E richiede 10 blocchi

   Illustrare con un diagramma simile a quello presente nel testo dell'esercizio (quindi specificando esplicitamente gli indirizzi) la configurazione di memoria del calcolatore a seguito dell'allocazione dei processi specificati nei casi in cui la memoria venga gestita secondo i diversi paradigmi first fit, next fit, best fit e worst fit (Nota: l'ultimo processo a essere stato allocato prima di questi tre è il processo B).

   Sol:

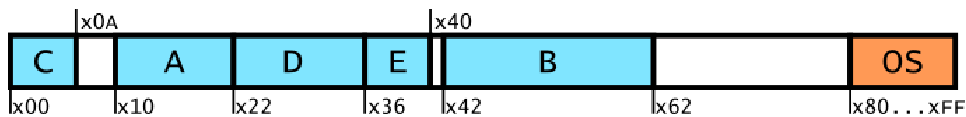


Figure 5: First fit

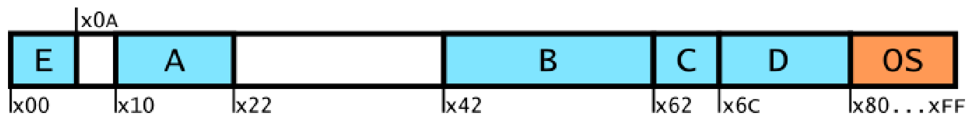


Figure 6: Next fit

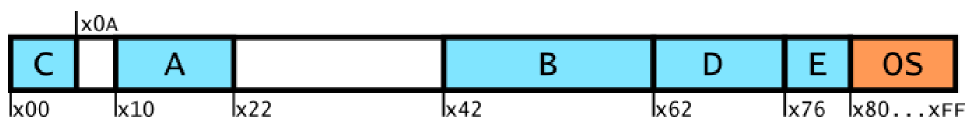


Figure 7: Best fit

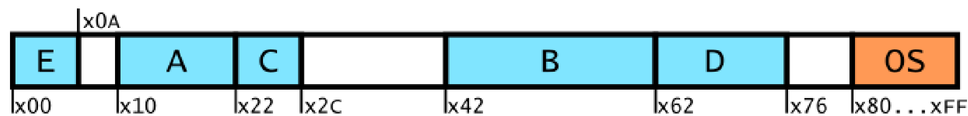


Figure 8: Worst fit

Esercizio 03

In questo esercizio, oltre ai comandi bash già visti, useremo `ps`, `top` (o `htop` equivalentemente), `pstree` e `jobs`.
Nota per chi usa Cygwin: è necessario installare i pacchetti `psmisc` e `procps-ng` per i comandi `pstree` e `top`.

Nota: variabili in `bash`
È possibile definire variabili in bash (i nomi delle variabili sono case sensitive!) e accedere ai valori in esse contenuti. Ad esempio:

```
a=5
b=stringa
c="stringa con spazi"
```

(nota che non ci sono spazi tra il simbolo `=`, nome e valore della variabile)
Per *espandere* le variabili (i.e. avere accesso al contenuto) dobbiamo precederne il nome col simbolo `$` (nota che usiamo `echo` per stampare il valore a schermo, altrimenti bash avrebbe interpretato il contenuto della variabile come un comando da eseguire)

```
echo $a
echo $b
echo $c
```

Col comando `set` possiamo visualizzare a schermo tutte le variabili che abbiamo definito, assieme a quelle che già sono definite nella sessione corrente (come potrete notare, ce n'è un bel po'). Ci sono poi alcune variabili speciali, che non figurano nell'output di `set` ma possono ugualmente essere usate. A noi interessa in particolare la variabile `$` (eh sì, lo stesso simbolo che usiamo per espandere le variabili), che contiene il PID del processo `bash` corrente.

1. Consulta la manpage del comando `pstree` e scopri a cosa servono le opzioni `-p` e `-s`

2. Apri una shell e stampa a schermo il PID del processo `bash` corrente. Verifica poi che il processo figuri nell'output dei comandi `ps`, `top` (o `htop`, se installato) e `pstree -p` e che all'interno della cartella `/proc` sia presente una sottocartella chiamata esattamente con il PID rinvenuto.

   Sol:

```
echo $$
ps
top
pstree -p
ls -l /proc | grep $$
```

3. Possiamo stampare il sotto-albero di processi radicato in un processo con PID noto semplicemente fornendo il PID in questione in input a `pstree`. Sapendo questo:

   (a) esegui il comando `pstree -ps` fornendo in input il PID della shell corrente

   (b) esegui il comando `bash` e ripeti il procedimento del punto precedente (attenzione: il PID non è lo stesso di prima!). Che differenze noti nell'output delle due chiamate a `pstree`?

   (c) Ripeti la procedura del punto precedente per un paio di volte. Cosa noti nell'output di `pstree`?

   (d) Esegui il comando `exit` finché `pstree` (chiamato come nei punti precedenti) non mostra lo stesso output della prima chiamata, fatta al primo punto di questo esercizio. Sapresti descrivere cosa stiamo facendo a livello di processi?

   (e) (NOTA: Questo punto non si può fare su Repl, serve un emulatore di terminale sulla propria macchina) Cosa succede, inoltre, se esegui il comando `exit` ancora una volta?

   Sol:

   ```
   pstree -ps $$
   bash
   pstree -ps $$
   ...
   exit
   ...
   ```

   A ogni passo creiamo un nuovo processo `bash` figlio di quello precedente, come mostrato dall'output di `pstree`, che mostra una catena di processi sempre più lunga. Nell'ultimo punto ogni chiamata a `exit` termina il processo `bash` più annidato, attuando di fatto un procedimento inverso a quello realizzato nei punti precedenti. L'ultimo `exit` fa chiudere la finestra del terminale.

4. Apri una shell ed esegui il comando

```
echo start ; sleep 3 ; echo end
```

dopodiché esegui il comando

```
echo start ; sleep 3 & echo end
```

aspetta qualche secondo e premi ⟨INVIO⟩senza nessun comando. Che differenze noti nel comportamento della shell nei due casi?

Sol: Nel primo caso i messaggi “start” e “end” vengono stampati a 3 secondi di distanza, e nel frattempo il terminale è inutilizzabile. Nel secondo caso i messaggi vengono stampati subito, intervallati però da un messaggio che indica un numero tra parentesi quadre seguito da un altro numero (i.e. `[jobid]PID`). Premendo ⟨INVIO⟩a vuoto dopo qualche secondo appare un messaggio che indica che il processo `sleep 3` è terminato.

Esercizio 04

Si risolvano i seguenti esercizi su Multi-Processing.

1. Si scriva un programma C in cui, dato un'array di dimensione N (ad esempio 10), il processo principale genera un solo processo figlio; i due processi tentano poi di alternarsi nello stampare, uno per volta, i valori nell'array nell'ordine originale (i.e. uno stampa quelli in posizioni pari, l'altro in quelle dispari). Cosa possiamo notare?

   Sol: I due processi non riescono a mantenere una sequenza ordinata di stampe (a meno di introdurre dei dispendiosissimi `sleep`, ma funziona solo perché le attese sono lunghe rispetto ai calcoli, e comunque fare questo tipo di sincronizzazione manuale è una pessima idea). Questo fatto ci porta ad introdurre un altro concetto fondamentale legato al multi-processing, ossia l'introduzione dei semafori.

   ```
   #include <sys/types.h>
   #include <unistd.h>
   #include <stdio.h>

   int main(){
       int N = 10;
       int array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

       pid_t pid = fork();
   ```

```
    if(pid == 0){
        for(int i = 0; i < N; i += 2){
            //sleep(2); <- SYNC MANUALE, DA NON FARE
            printf("Processo Figlio: %d\n", array[i]);
        }
    }else{
        //sleep(1); <- SYNC MANUALE, DA NON FARE
        for(int i = 1; i < N; i += 2){
            //sleep(2); <- SYNC MANUALE, DA NON FARE
            printf("Processo Padre: %d\n", array[i]);
        }
    }

    return 0;
}
```

2. Si scriva un programma C in cui, dato un'array di interi, il processo principale genera un processo figlio per calcolare la media tra i soli valori pari ed un secondo processo figlio per calcolare la media tra i soli valori dispari (Nota: Occhio! parliamo dei valori, non degli indici!). Il processo principale (processo padre) dovrà invece determinare quali valori sono il massimo ed il minimo.

   Sol:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(){
    int N = 10;
    int array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    pid_t pid1 = fork();

    if (pid1 > 0) {

        pid_t pid2 = fork();

        if(pid2 > 0){

            int mx = array[0];
            int mn = array[0];
```

```
            for(int i = 1; i<N; i++){
                if(array[i]>mx){mx = array[i];}
                if(array[i]<mn){mn = array[i];}
            }

            printf("Processo Padre:\n"
                   "Max : %d\n"
                   "Min : %d\n", mx, mn);
        } else {

            double somma_pari = 0;

            for (int i = 0; i < N; i++) {
                if (array[i] % 2 == 0)
                    somma_pari = somma_pari + array[i];
            }

            printf("Processo Figlio:\n"
                   "media valori pari: %2.2f\n", somma_pari/N);
        }
    } else {

        double somma_dispari = 0;

        for (int i = 0; i < N; i++) {
            if (array[i] % 2 == 0)
                somma_dispari = somma_dispari + array[i];
        }

        printf("Processo Figlio:\n"
               "media valori dispari: %2.2f\n", somma_dispari/N);
    }
    return 0;
}
```

3. Si scriva un programma C in cui, dato un numero intero N, il processo principale genera un processo un figlio, il quale a sua volta genera un altro processo figlio, e così via, fino ad ottenere N processi in tutto. In particolare, ogni processo attende la terminazione del proprio processo figlio.

   Sol:

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>


int dichiarazione_processo() {
    printf("Sono un processo figlio. "
        "Il mio ID:%d  Il mio parent ID:"
            "%d\n", getpid(), getppid());
    return 1;
}

int genera_processi(int N) {

    if(N == 0){return 0;}

    int pid = fork();

    if(pid == -1) {exit(0);}

    if(pid == 0) {
        dichiarazione_processo();
        N = N - 1;
        genera_processi(N);
        exit(0);
    }else{
       wait(NULL);
    }

    return 0;
}

int main(){
    int N = 5;
    genera_processi(N);
    return 0;
}
```