Esercizi Laboratorio Software

*(*tricky, ^solo pseudocodice)*

Esercizio #1 – *gestione dei processi, fork(), wait()*

Si supponga di avere a disposizione un file di testo `pippo.text` contenente una stringa di 60 caratteri, come segue

1111111111222222222233333333334444444444555555555566666666666

Si vuole sincronizzare un processo P1 (padre) con il corrispondente processo P2 (figlio) come segue: il padre legge 20 bytes dal file e successivamente modifica i primi 10 caratteri, il figlio legge i primi 20 caratteri dal file, il padre ora legge ancora 10 caratteri. Si utilzzi un solo descrittore di file senza dupicarlo. Utilizzare una sincronizzazione esplicita basata su `fork()` e `wait()`. Cosa se ne conclude circa l'utilizzo della sincronizzazione tramite `wait()`? La sincronizzazione tramite `wait()` è flessibile? È facilmente modificabile? È totalmente sicura? A che byte punta il descrittore del file al termine delle operazioni richieste?

Esercizio #2 – *gestione dei processi, valore di ritorno del figlio*

Si vuole sviluppare una semplice toy-application in cui l'utente specifica da terminale il valore che il figlio dvorà restituire. Il processo padre, una volta che il figlio ha terminato, deve leggere il valore di ritorno specificato e stamparlo a video. Documentarsi circa l'utilizzo del valore di ritorno nelle *man pages* della `wait()`.

Esercizio #3 – *gestione dei processi, segnali*

Proporre una soluzione alternativa all'**Esercizio#1**, in modo tale da utilizzare i segnali per la comunicazione dell'avvenuta terminazione – da parte del padre – della scrittura su file. Si utilizzi a tale scopo il segnale `SIGUSR1`. Cosa cambia dal punto di vista della comunicazione? Riportare il diagramma temporale dell'evoluzione dei due processi. Cosa succede se il processo figlio termina (inaspettatamente) prima di inviare il segnale `SIGUSR1` al padre? Cosa succede al processo figlio se il padre termina prima del dovuto?

Esercizio #4 – *gestione dei processi, segnali*

Un processo P1 crea un processo P2; il processo P2 rimane in attesa sui segnali `SIGUSR1` e `SIGUSR2` per tutta la durata della sua esecuzione, mentre da terminale l'utente invia una serie di segnali `SIGUSR1` e un unico `SIGUSR2` al processo P2. Al termine della sua esecuzione (definito dall'invio del segnale `SIGUSR2`) il processo P2 stampa a video il numero di interrupt software ricevuti tramite `SIGUSR1`, i.e. quante volte il segnale è stato effettivamente inviato a P2 il segnale `SIGUSR1`. Si supponga di avere la seguente evoluzione temporale, in cui P1 crea P2 al tempo `t1`; l'utente invia `SIGUSR1` a P2 al tempo `t2` e un segnale di `SIGSTOP` a P2 al tempo `t3` (non importa il valore assoluto dei tempi, ma il loro ordinamento all'interno dell'intervallo considerato); due segnali`SIGUSR1` sono inviati a P2 tra `t4` e `t5`, mentre `SIGCONT` è inviato al tempo `t6`. P1 rimane in attesa della terminazione di P2 durante tutta l'esecuzione. Cosa stampa a video P2 a `t=t7` se l'utente invia `SIGUSR2`? Si rifletta sul perchè di questo risultato. Proporre un'implementazione dello scenario dato.

t	1	2	3	4	5	6	7	…
P1	Fork()	Wait()						
Utente		SIGUSR1 a P2	SIGSTOP a P2	SIGUSR1 a P2	SIGUSR1 a P2	SIGCONT a P2	SIGUSR2 a P2	
P2							Print()	

Esercizio #5 – *gestione dei processi, condivisione dei dati*

Cosa stampa il seguente programma? Perchè? Rispondere senza compilare il codice, ma riflettendo sull'effetto delle chiamata `fork()`.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <sys/wait.h>
 4 #include <sys/types.h>
 5
 6 int j;
 7
 8 int main()
 9 {
10     pid_t pid;
11     int *k;
12
13     /* Initialization */
14     k = (int *)malloc(sizeof(int));
15     j = 1;
16     *k = 2;
17
18     /* Create child process */
19     pid = fork();
20     if(pid == 0)
21     {
22          j = 45;
23          (*k)++;
24          printf("CHILD : \t"
25              "j = %d\t"
26              "k = %d\n", j, *k);
27
28          exit(0);
29     }
30     else
31     {
```

```
32          wait(NULL);
33          printf("PARENT :\t"
34              "j = %d\t"
35              "k = %d\n", j, *k);
36
37          free(k);
38          return 0;
39      }
40  }
41
```

Esercizio #6– *thread, processi, comunicazione, parallelismo*

Proporre una soluzione al problema della moltiplicazione tra matrici intere. Per semplicità si considerino due matrici `A` e `B` quadrate di dimensione `NxN`. La soluzione deve trarre vantaggio dall'implicito parallelismo del problema; utilizzare a tale scopo un numero adeguato di thread e/o processi (in combinazione opportuna, se necessario). Per semplicità si consideri inoltre `N = 5`. Per la sincronizzazione proporre un meccanismo adeguato. Riflettere sui problemi di sincronizzazione? Quali sono? Come vengono risolti?

* Esercizio #7 *thread, mapped memory, sincronizzazione*

Si risolva ancora l'esercizio precedente. Questa volta, però, i due operandi della moltiplicazione vengono letti da un file in ingresso con la seguente struttura:

`A1,A2,…,AM:B1,B2,…,BM`

M è la dimensione delle matrici (ancora una volta quadrate), `Aj` rappresenta il valore dell'elemento `j`-esimo del primo operando (matrice di sinistra), mentre `Bj` il valore dell'elemento `j`-esimo del secondo operando (matrice di destra); `M` vale `N*N`, dove N è specificato (per semplicità) all'interno del codice. La matrice è "linearizzata" per righe. Ad esempio, la seguente matrice

$$A = \begin{matrix} 1 & 4 & 5 \\ 5 & 6 & 2 \\ 3 & 4 & 6 \end{matrix}$$

viene rappresentata nel file come segue: 1,4,5,5,6,2,3,4,6.
Le strutture dati dei due operandi devono essere condivise in memoria, e non devono essere variabili globali. Il risultato dell'operazione deve essere salvato in un opportuno segmento di memoria condiviso, e tale dato verrà a sua volta salvato su un file alla fine dell'intera computazione. Si utilizzi a tale scopo della memoria mappata, e non accesso diretto ai file, sia in lettura che in scrittura.

Esercizio #8 – *processi, condivisione di dati, memoria condivisa*

Proporre una soluzione all'**Esercizio#5** utilizzando un segmento di memoria condivisa. In particolare occorre far vedere come uno dei due processi (e.g. il figlio) scriva un dato in memoria, e l'altro (il padre) legga tale dato.

Esercizio #9 – *comunicazione tra processi, memoria condivisa*

Riproporre la soluzione all'**Esercizio#3** in cui il file di testo `pippo.text` è ora sosituito con un opportuno buffer in memoria condivisa. Che problemi si possono riscontrare utilizzando un segmento di memoria condivisa? Come avviene la sincronizzazione tra i due processi? Che differenze si trovano rispetto alla soluzione originale?

Esercizio #10 – *comunicazione tra processi, mapped memory*

Riprendendo l'**Esercizio#9**, proporre un'implementazione basata su mapped memory.

Esercizio #11 – *comunicazione tra processi, pipe*

Dato un processo P1 (il padre) e un processo P2 (il figlio), si vuole fare in modo che il padre scriva del testo (e.g. "Hello World!") sul rispettivo end-point del canale di comumicazione; il figlio legge un byte alla volta dal rispettivo punto di accesso al canale, e stampa conseguentemente a video il testo ricevuto. Si utilizzi una pipe per la comunicazione tra padre e figlio.

Esercizio #12 – *processi, argomenti da linea di comando*

Scrivere un programma C per calcolare il valore dell'`N`-esimo elemento nella successione di Fibonacci, dove `N` è specificato da linea di comando. L'output del programma è eseguito dal figlio.

Esercizio #13 – *processi, memoria condivisa*

Riprendere l'esercizio precedente, apportando le seguenti modifiche: il figlio genera l'intera successione fino all'elemento `N`, e comunica al padre la successione tramite un segmento di memoria condivisa. Sarà il padre ora ad eseguire l'output. Inoltre, il valore `N` sarà comunicato al figlio tramite memoria condivisa, e non accedendo ad una potenziale variabile ereditata dal padre.

* Esercizio #14 – *processi, pipe, modello pipeline, sincronizzazione*

Sempre considerando la successione di Fibonacci, ora l'implementazione richiede `N` processi.
Ogni processo `j` comunica la somma parziale della successione (fino all'elemento `j`-esimo) al processo `j+1`; la relazione tra i due procesi è padre-figlio. La comunicazione avviene tramite pipe (implementazione tramite processi del *modello pipeline*). Ogni processo `0<=j<N` deve stampare il valore della somma parziale (o totale, nel caso del processo N-esimo).
Per semplicità si può considerare come dato (ad esempio una routine invocata una sola volta all'inizio) l'array di interi contenente l'intera successione; ogni processo andrà poi a prelevare il rispettivo elemento in posizione `j` dall'array. Riflettere sul problema della sincronizzazione e si trovi una soluzione elegante al problema, che sia parametrica rispetto ad `N` (`N>=3`).

Esercizio #15 – *processi*

Riprendendo il programma dell'**Esercizio#5**, a cui sono state apportate le modifiche riportate qui sotto (in rosso le modifiche al codice), riflettere sul perchè dell'output dato considerando il meccanismo *Copy-on-Write* spiegato a lezione. Cosa vi sareste aspettati dall'ouput?

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/wait.h>
```

```c
#include <sys/types.h>

int j;

int main()
{
    pid_t pid;
    int *k;

    /* Initialization */
    k = (int *)malloc(sizeof(int));
    j = 1;
    *k = 2;

    /* Create child process */
    pid = fork();
    if(pid == 0)
    {
         j = 45;
         printf("CHILD : \t"
             "address(k) = %p\n", k);
         (*k)++;
         printf("CHILD : \t"
             "address(k) = %p\n", k);
         printf("CHILD : \t"
             "j = %d\t"
             "k = %d\n", j, *k);

         exit(0);
     }
    else
    {
         wait(NULL);
         printf("PARENT :\t"
             "j = %d\t"
             "k = %d\n", j, *k);

         free(k);
         return 0;
    }
}
```

L'output sulla mia macchina è il seguente (non interessano i valori assoluti degli indirizzi, ma la loro relazione):

```
CHILD :          address(k) = 0x804a008
CHILD :          address(k) = 0x804a008
```

Esercizio #16 (Fossati, Zaccaria) – *sincronizzazione, mutex*

Si risolva il problema della sincronizzazione tra un produttore e un consumatore utilizzando i mutex: un thread scrive una stringa e altri thread la leggono. Non è possibile avere contemporaneamente scritture e letture.

Esercizio #17 – *sincronizzazione, semafori e thread*

Proporre una soluzione al problema del produttore/consumatore basata su semafori (per i thread).

Esercizio #18 – *processi*

Cosa stampa il seguente programma dato un valore di `num_of_processes`?

```
 1   #include <stdio.h>
 2   #include <unistd.h>
 3   #include <stdlib.h>
 4
 5   #define DEFAULT     3
 6
 7   int main(int argc, char *argv[])
 8   {
 9        int num_of_processes, j;
10
11        if(argc < 2)
12        {
13             num_of_processes = DEFAULT;
14        }
15        else
16        {
17             num_of_processes = atoi(argv[1]);
18        }
19
20        for(j = 0; j < num_of_processes; j++)
21        {
22             fork();
23        }
24        fprintf(stdout, "X\n");
25
26     return 0;
27 }
```

^ Esercizio #19 – *semafori*

Sono dati tre processi `P1`, `P2` e `P3` che condividono una variabile `x`. Lo pseudo-codice per ciascuno dei processi è riportato di seguito:

P1
```
wait(S)
x = x-1
post(T)
```

P2
```
wait(R)
x = x+2
post(T)
wait(R)
x = x+1
post(T)
```

P 3
```
wait(T)
if (x>0) then
  post(R)
else
  post(S)
endif
wait(T)
print x
```

Assumendo che all'inizio dell'esecuzione `x` vale `−1` e i semafori valgono `S=0`, `R=1` e `T=0`, cosa stampa `P3`?

Esercizio #20 – *processi*

Quanti processi genera il seguente codice? Cosa stampa a video?

```
 1   #include <unistd.h>
 2   #include <sys/types.h>
 3   #include <stdlib.h>
 4
 5   int main ( )
 6   {
 7        pid_t pid;
 8        int i, count;
 9        count = 1;
10        for (i = 0; i <= 3; i++)
11        {
12             pid = fork();
13             if(pid == 0 && i / 2 * 2 == i)
14             {
15                  count++;
16                  exit(0);
17             }
18        }
19        printf("Process %d finished\n", getpid());
20        printf("count is %d\n", count);
21        return 0;
22   }
```

Esercizio #21 (Fossati, Zaccaria) – *semafori*

In un *negozio del barbiere* il barbiere dorme fino a che non ci sono clienti. All'arrivo un cliente può:

1. svegliare il barbiere (nel caso stesse dormendo);
2. sedersi e aspettare che il barbiere abbia finito con il cliente attuale;
3. se tutte le sedie della sala d'attesa sono occupate il cliente se ne va.

Simulare il comportamento appena descritto tramite semafori.

^ Esercizio #22 – *semafori*

Si considerino un processo **P**roduttore e un processo **C**onsumatore, e un buffer con capacità 1. Date le seguenti porzioni di codice, e supponendo che le istruzioni p1 e p2 vengano eseguite per prime, stabilire se il sistema si comporta effettivamente come produttore/consumatore.

PRODUTTORE

```
semaphore empty = 1
while(1)
{
   p1: down(&empty);
   p2: enter_item(item);
}
```

CONSUMATORE

```
semaphore empty = 1
while(1)
{
  c1: remove_item(item);
  c2: up(&empty);
}
```

Esercizio #23 – *semafori, produttore/consumatore*

Proporre una soluzione al problema del produttore/consumatore con un buffer a capacità `1`. Si faccia anche riferimento, se necessario, a quanto visto per l'esercizio precedente.