Caselle riservate

Ex. 1	
Ex. 2	
Ex. 3	
Ex. 4	
Ex. 5	
Ex. 6	
Tot.	

Sistemi Operativi

Compito d'esame

27 Gennaio 2020

Matricola _____ Cognome _____ Nome _____

Docente: ◯ Quer ◯ Sterpone

Non si possono consultare testi, appunti o calcolatrici a parte i formulari distribuiti dal docente. Risolvere gli esercizi negli spazi riservati. Fogli aggiuntivi sono permessi solo quando strettamente necessari. Riportare i passaggi principali.
Durata della prova: 100 **minuti.**

1. Si supponga di eseguire il programma seguente

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
void *t1 (void *p){
  int *n = (int *) p;
  printf ("--- thread: id=%d\n", *n);
  pthread_exit (NULL);
}
int main () {
  pthread_t thread;
  int i, n, v[2];
  char str[100];
  setbuf (stdout, 0);
  for (i=0; i<2; i++)
    if (fork()>0) {
      v[i] = 0;
    } else {
      v[i] = 1;
    }
  n = v[0] + v[1]*2;
  pthread_create (&thread, NULL, t1, &n);
  pthread_join (thread, NULL);
  sprintf (str, "echo '- echo: n=%d'", n);
  system (str);
  sprintf (str, "-- exec: n=%d", n);
  execlp ("echo", "bash", str, NULL);
  return 1;
}
```

Si riporti il grafo di controllo del flusso e l'albero di generazione dei processi a seguito della sua esecuzione. Si indichi inoltre che cosa esso produce su video e per quale motivo.

2. Descrivere la sintassi delle system call `wait` e `exit` e fornire un esempio di come sia possibile utilizzarle per fare in modo che un processo figlio, alla terminazione, passi un valore intero al processo padre. Cosa succede se il padre non chiama `wait` e il figlio termina? Che meccanismo viene utilizzato dal kernel per identificare la terminazione di un processo?

   Due processi figlio, P_1 (di PID 123) e P_2 (di PID 456), unici figli del processo genitore, terminano. Indicare, per i due frammenti di codice seguenti, l'output generato dalla funzione `printf` nel caso in cui P_1 termini prima e nel caso in cui termini dopo P_2 (4 casi in totale).

Code 1

```
while (wait()!=123);
pid = wait();
printf ("%d\n", pid);
```

Code 2

```
waitpid (123, (int *) 0, 0);
pid = wait ();
printf ("%d\n", pid);
```

3. Si illustri il problema del *Produttore e Consumatore*, nel caso di P produttori e C consumatori, e si riporti in pseudo-codice un possibile schema di implementazione. Si indichi la funzione dei vari semafori motivandone l'utilizzo.

   Si adatti quindi la soluzione precedente al caso in cui siano presenti esattamente 3 produttori e 1 unico consumatore. Ciascun produttore generi elementi in una coda a lui dedicata. Il consumatore consumi elementi dando maggiore priorità alla coda con un numero più elevato di elementi memorizzati.

   Suggerimento: utilizzare dei contatori per tenere traccia del numero di elementi presenti in ciascuna coda oppure ricorrere a una funzione (tipo `sem_getvalue`) in grado di ritornare il valore di un semaforo.

4. Si implementi uno script BASH che riceva il path di una directory su linea di comando. Lo script, dopo aver verificato il passaggio del numero corretto di parametri, deve selezionare, nel sotto-albero di direttori con radice la cartella specificata, tutti i file regolari con dimensione minore di 10MB il cui nome inizi con la stringa "spesa" seguita da un numero qualunque e dall'estensione `.xyz` (ad esempio, `spesa1.xyz`, `spesa200.xyz`). Si assuma che ciascuno di questi file contenga un testo con un formato simile ai seguenti:

spesa1.xyz
```
Product Quantity Unit_price
pasta 2 5
pizza 1 8
pasta 1 6
```

spesa200.xyz
```
Product Quantity Unit_price
pizza 2 2
frutta 3 5
```

dove la prima riga è di intestazione e le righe successive contengono un nome di prodotto, una quantità e un prezzo unitario (separati da singoli spazi).

Per ogni file selezionato, lo script deve generare un file con lo stesso nome ma con estensione `.dat`, privo di intestazione, che contenga per ogni prodotto il totale, ottenuto sommando le quantità moltiplicate per i prezzi unitari di tutte le righe in cui quel prodotto compare.

Per i file di esempio precedentemente riportati, i file generati devono essere i seguenti:

spesa1.dat
```
pasta 16
pizza 8
```

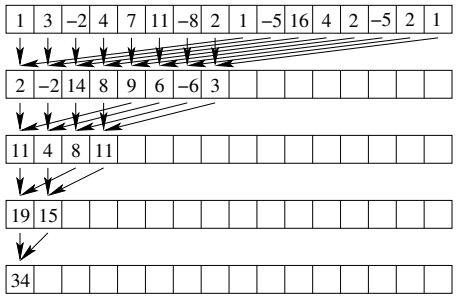
spesa200.dat
```
pizza 4
frutta 15
```

5. Una funzione riceve come parametri un vettore di interi (`vet`) e la sua dimensione (`n`), che si suppone essere pari ad una potenza di 2:

   ```
   int array_sum (int *vet, int n);
   ```

   La funzione deve ritornare la somma degli elementi del vettore, calcolandola utilizzando una versione concorrente dell'algoritmo riportato in seguito e illustrato dalla figura su un vettore di dimensione `n = 16`:

   ```
   int i, k;
   k = n/2;
   while (k != 0) {
     for (i=0; i<k; i++) {
       vet[i] += vet[i+k];
     }
     k=k/2;
   }
   ```

   In particolare, la funzione deve applicare i passi del precedente algoritmo facendo in modo che tutte le operazioni di somma siano eseguite (in parallelo) da $n/2$ thread separati. Ogni thread risulti associato a una delle prime $n/2$ celle del vettore. Ogni thread si occupi di eseguire tutte le somme il cui risultato debba essere memorizzato nella cella del vettore a esso associata. Si noti che il numero di somme che ciascun thread dovrà eseguire dipende dalla posizione della cella del vettore a esso associata. Gestire la sincronizzazione tra i thread mediante semafori, in modo che tutte le somme vengano effettuate rispettando le precedenze.

6. Chiarire le principali differenze tra un file di tipo ASCII (o testo) e uno di tipo binario. Quali vantaggi e svantaggi offrono questi ultimi?

   Si illustrino inoltre le principali differenze tra le funzioni `fopen` e `open`, tra `fprintf` e `write`, e tra `fscanf` e `read`.

   Nella memorizzazione di un file si chiariscano quindi le differenze tra l'allocazione concatenata e quella indicizzata, illustrandone vantaggi e svantaggi. Nell'allocazione indicizzata in ambiente UNIX/LINUX si indichi inoltre che cosa si intende con i termini "directory block", "directory entry", "data block" e "i-node".