Caselle riservate

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

Sistemi Operativi

Compito d'esame

27 Gennaio 2020

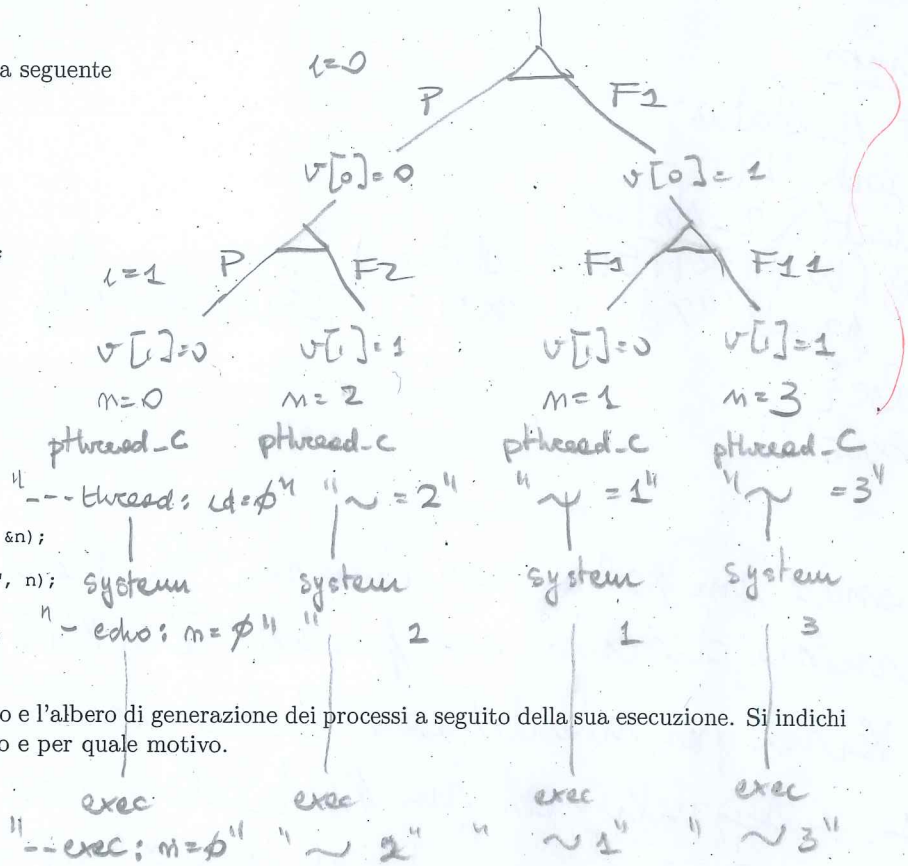
Matricola _____ Cognome _____ Nome _____

Docente: ◯ Quer ◯ Sterpone

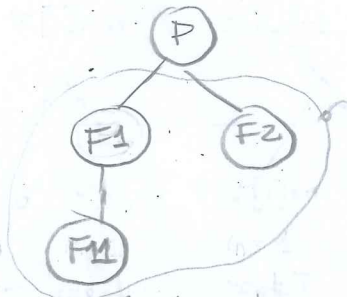
Non si possono consultare testi, appunti o calcolatrici a parte i formulari distribuiti dal docente. Risolvere gli esercizi negli spazi riservati. Fogli aggiuntivi sono permessi solo quando strettamente necessari. Riportare i passaggi principali.
Durata della prova: 100 minuti.

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.



ognuno fa 1 system
poi
ognuno fa 1 exec

ordine

ogni gruppo in ordine qualunque

```
--- thread: id=0
--- thread: id=2
--- thread: id=1
--- thread: id=3
- echo: n=0
- echo: n=1
- echo: n=2
- echo: n=3
-- exec: n=0
-- exec: n=1
-- exec: n=2
-- exec: n=3
```

1
2
3

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);
```

Sintassi: pid_t wait(int *wstatus); → Attende un processo figlio
int exit(int status); → causa la normale terminazione di un processo.

Esempio:
```
int r_status;
if(fork()){
  wait(&r_status);
  if(WIFEXITED(r_status))
    printf("%d", WEXITSTATUS(r_status));
}else{
  exit(1);
}
```

Quando un padre non chiama la wait e il figlio termina diventa un processo ZOMBIE.
Il kernel per identificare la terminazione di un processo usa i segnali, col in particolare un segnale di tipo SIGCHLD indirizzato al padre del processo.

	code 1	code 2
P_1 poi P_2	456	456
P_2 poi P_1	-1	456

→ wait restituisce -1 nel caso il processo non abbia figli

→ Il processo P_2 diventa ZOMBIE e poi è catturato dalla wait

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.

1P 1C

```
P
while (1) {
    produce (&val);
    wait (empty);
    enqueue (val);
    signal (full);
}
```

```
C
while (1) {
    wait (full);
    dequeue (&val);
    signal (empty);
    consume (val);
}
```

init (full, 0)
init (empty, SIZE)
init (MEp, 1)
init (MEc, 1)

PP CC

```
while (1) {
    produce (&val);
    wait (empty);
    wait (MEp);
    enqueue (val);
    signal (MEp);
    signal (full);
}
```

```
while (1) {
    wait (full);
    wait (MEc);
    dequeue (&val);
    signal (MEc);
    signal (empty);
    consume (val);
}
```

P_1, P_2, P_3 — idem con $I = 1, 2, 3$

```
while (1) {
    produce (&val);
    wait (empty[i]);
    enqueue (queue[i], val);
    signal (full[i]);
}
```

init (full[i], 0)
init (empty[i], SIZE)

C

```
while (1) {
    n1 = sem_getvalue (full1)
    n2 = sem_getvalue (full2)
    n3 = sem_getvalue (full3)
    n = max (n1, n2, n3)
    wait (full[n]);
    dequeue (&val);
    signal (empty[n]);
    consume (val);
}
```

queue[n]

WITH COUNTERS

```
init(full, 0); init(empty[i], SIZE); init(m[i], 1); i=0,1,2
int counter[i]=0;
```

P_0, P_1, P_2 (Prod) (i = process idx)

```
while(1){
    produce(&val);
    wait(empty[i]);
    wait(m[i]);
    enqueue(queue[i], val);
    counter[i]++;
    signal(m[i]);
    signal(full);
}
```

Cons.

```
while(1){
    wait(full);
    wait(m[0]);
    wait(m[1]);
    wait(m[2]);
    n = max_idx(counter[0],
        counter[1], counter[2]);
    for(i=0; i<3; i++)
        if(i != n) signal(m[i]);
    dequeue(queue[n], &val);
    counter[n]--;
    signal(m[n]);
}
```

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
```

```bash
#!/bin/bash

################################################################################################
# Esercizio 4 - Esame
27/01/2020                                                                    #
#   Lanciare con: ./es4.sh
<folder>                                                                   #
################################################################################################

# Controllo argomenti
if [ $# -ne 1 ]; then
    echo "Utilizzo: es4.sh <folder>"
    exit 1
fi

# Selezione dei file e salvataggio della lista dei loro percorsi in un file
temporaneo
find $1 -type f -size -10M -regex '.*\/spesa[0-9]+\.xyz$' > 'list.txt'

# Scansione dei percorsi dei file selezionati
while read filename; do

    # Dichiarazione di un nuovo vettore in cui sommare le spese
    declare -A sums

    # Scansione del contenuto del file saltando la prima riga e somma delle spese
per prodotto
    isfirst=0
    while read product quantity price; do
        if [ $isfirst -ne 0 ]; then
            let sums[$product]+=quantity*price
        fi
        isfirst=1
    done < $filename

    # Composizione del nome del file di output
    name=$(basename $filename ".xyz")
    outfile="$name.dat"

    # Stampa delle spese per prodotto su file
    for product in "${!sums[@]}"; do
        echo $product ${sums[$product]}
    done > $outfile

    # Cancellazione vettore delle somme
    unset sums

done < 'list.txt'

# Rimozione del file temporaneo
rm -f 'list.txt'

exit 0
```

```bash
#!/bin/bash

################################################################################################
# Esercizio 4 - Esame
27/01/2020                                                                              #
#   Lanciare con: ./es4b.sh
<folder>                                                                              #
################################################################################################

# Controllo argomenti
if [ $# -ne 1 ]; then
    echo "Utilizzo: es4b.sh <folder>"
    exit 1
fi

# Selezione dei file e salvataggio della lista dei loro percorsi in un file
temporaneo
find $1 -type f -size -10M -regex '.*\/spesa[0-9]+\.xyz$' > 'list.txt'

# Scansione dei percorsi dei file selezionati
while read filename; do

    # Rimozione intestazione e ordinamento delle righe per prodotto in un nuovo
file temporaneo
    cat $filename | tail -n +2 | tr -s " " | sort -t " " -k 1 > "tmp.txt"

    # Composizione del nome del file di output
    name=$(basename $filename ".xyz")
    outfile="$name.dat"

    # Scansione del file del contenuto del file sommando e stampando le spese per
prodotto
    current=""
    tot=0
    while read product quantity price; do
        if [ "$current" == "" ]; then
            current=$product
            tot=0
        elif [ "$product" != "$current" ]; then
            echo $current $tot >> $outfile
            current=$product
            tot=0
        fi
        let tot+=quantity*price
    done < "tmp.txt"
    echo $current $tot >> $outfile

    # Rimozione del file temporaneo
    rm -f "tmp.txt"

done < 'list.txt'

# Rimozione del file temporaneo
rm -f 'list.txt'

exit 0
```

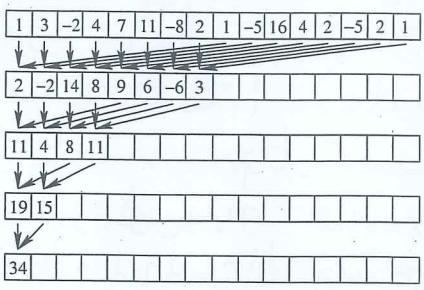
```bash
#!/bin/bash

################################################################################################
# Esercizio 4 - Esame
27/01/2020                                                                              #
#   Lanciare con: ./es4c.sh
<folder>                                                                              #
################################################################################################

# Controllo argomenti
if [ $# -ne 1 ]; then
    echo "Utilizzo: es4c.sh <folder>"
    exit 1
fi

# Selezione dei file e salvataggio della lista dei loro percorsi in un file
temporaneo
find $1 -type f -size -10M -regex '.*\/spesa[0-9]+\.xyz$' > 'list.txt'

# Scansione dei percorsi dei file selezionati
while read filename; do

    # Scansione prodotti nel file
    for product in $(cat $filename | tail -n +2 | cut -d " " -f 1 | sort | uniq);
do

        # Estrazione entry nel file per il prodotto corrente in un file temporaneo
        cat $filename | tail -n +2 | grep $product | cut -d " " -f 2,3 > "tmp.txt"
        expense=0

        # Somma delle spese per il prodotto corrente
        while read quantity price; do
            let expense+=quantity*price
        done < "tmp.txt"

        # Composizione del nome del file di output
        name=$(basename $filename ".xyz")
        outfile="$name.dat"

        # Stampa sul file di output
        echo "$product $expense" >> $outfile

        # Rimozione del file temporaneo
        rm "tmp.txt"
    done

done < 'list.txt'

# Rimozione del file temporaneo
rm -f 'list.txt'

exit 0
```

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.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

typedef struct {
    int *vet;
    sem_t *sem;
    int n;
    int id;
} args_t;

void * adder(void * arg) {

    // Get argurments
    sem_t *sem = ((args_t *) arg)->sem;
    int *vet = ((args_t *) arg)->vet;
    int id = ((args_t *) arg)->id;
    int n = ((args_t *) arg)->n;

    // Perform addition synchronizing with the other threads
    int k=n/2;
    while(k != 0) {
        if(k < n/2)
            sem_wait(&sem[id + k]);
        vet[id] += vet[id + k];
        k=k/2;
        if(id >= k) {
            sem_post(&sem[id]);
            break;
        }
    }

    // Terminate thread
    pthread_exit(0);
}

int array_sum(int *vet, int n) {
    int k=n/2;
    pthread_t *tids;
    args_t *args;
    sem_t *sem;

    // Allocate thread id array
    tids = (pthread_t *) malloc(k*sizeof(pthread_t));

    // Initialize semaphores
    sem = (sem_t *) malloc(k*sizeof(sem_t));
    for(int i=0; i<k; ++i) {
        sem_init(&sem[i], 0, 0);
    }

    // Allocate array of args
    args = (args_t *) malloc(k*sizeof(args_t));
    for(int i=0; i<k; ++i) {
        args[i].id = i;
        args[i].vet = vet;
        args[i].n = n;
        args[i].sem = sem;
    }

    // Start threads
    for(int i=0; i<k; ++i) {
        pthread_create(&tids[i], NULL, adder, &args[i]);
    }

    // Wait for sum to be complete
    pthread_join(tids[0], NULL);
```

```
    // Destroy semaphores
    for(int i=0; i<k; ++i) {
        sem_destroy(&sem[i]);
    }

    // Free memory
    free(tids);
    free(sem);
    free(args);

    // Return sum
    return vet[0];
}

int main(int argc, char **argv) {
    int res = 0;
    for(int i=0; i<10000; i++) {
        if(i%1000==0) printf("%d\n", i);
        int vet[16] = {1, 3, -2, 4, 7, 11, -8, 2, 1, -5, 16, 4, 2, -5, 2, 1};
        int newres = array_sum(vet, 16);
        if(i == 0) res = newres;
        else if(res != newres) printf("Discrepancy %d %d\n", res, newres);
    }
    printf("Result: %d\n", res);
}
```

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 `printf` e `write`, e tra `scanf` 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".

File di testo è "line-oriented" e "byte-oriented", codificato in ASCII (e altre codifiche). File binario è "bit-oriented".
Vantaggi binario: tipicamente più compatto, permette la serializzazione
Svantaggi binario: non leggibile con editor general-purpose, incompatibilità tra architetture differenti, occorre conoscere la codifica.

Differenze fopen, open, ...
Le prime (fopen, printf e fscanf) sono funzioni di libreria POSIX e ANSI C, le seconde (open, write e read) sono system calls. L'implementazione delle prime è effettuata tramite le seconde. Le prime forniscono input/output formattato e bufferizzato, le seconde invece sono byte oriented, ... riportare eventualmente il prototipo (se conosciuto).

Allocazione concatenata: Ogni blocco contiene puntatore a successivo.
Vantaggi: Permette allocazione dinamica file, elimina frammentazione esterna.
Svantaggi: Efficiente solo per accesso sequenziale, spazio sprecato per salvare i puntatori, la perdita di un blocco impedisce l'accesso ai blocchi successivi.

Allocazione indicizzata: Un blocco (i-node) contiene i puntatori a tutti i blocchi di dati.
Vantaggi: Accesso diretto a qualsiasi blocco del file
Svantaggi: Overhead dovuto all'accesso dell'i-node, peso i-node, peso l'intero file.

Directory block: lista di directory entries.
Directory entry: coppia filename + i-node number o dirname + i-node number
Data block: blocco in cui sono memorizzati i dati effettivi del file.
i-node: porzione di memoria non volatile in cui vengono memorizzate informazioni sul file (permessi, timestamps, ...) e i puntatori ai data block.