Caselle riservate

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

Sistemi Operativi

Compito d'esame

21 Giugno 2019

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 indichi quali sono le principali differenze tra una codifica ASCII e una UNICODE, quali le differenze tra un file di testo e uno binario e che cosa si intende per serializzazione.

   Utilizzando le funzioni di input/output diretto POSIX

```
int open (const char *path, int flags, mode_t mode);
int read (int fd, void *buf, size_t nbytes);
int write (int fd, void *buf, size_t nbytes);
off_t lseek (int fd, off_t offset, int whence);
int close (int fd);
```

   si scriva un programma che sia in grado di creare una copia di un file binario invertendo l'ordine dei byte in esso contenuti (il primo byte diventa l'ultimo e viceversa). Il programma riceve su linea di comando il nome del file di input (che si suppone esistere) e il nome del file di output. Si osservi che non è consentito memorizzare l'intero file in memoria centrale.

(1) ASCII (Extended ASCII) : 8 bit, 255 caratteri
Diverse versioni (Latin, Cyrillic, etc.)

UNICODE : standard industriale con diverse codifiche
UTF-8 (include ASCII), UTF-16, UTF-32
la versione 6.3 rappresenta più di 110000 simboli

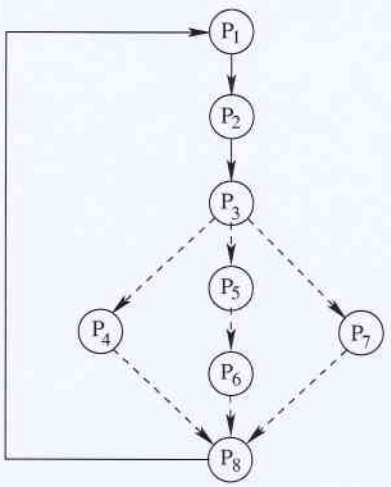
(2) File = serie di bit
Testo : i bit sono raggruppati e interpretati secondo la codifica utilizzata (ASCII, UNICODE); line-oriented.

Binari : sequenze di bit

(3) Serializzazione = processo di trasformazione di una struttura dati (oggetto) in una sequenza di byte direttamente memorizzabile

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

#define N 12

int main (int argc, char **argv) {
  char c;
  int fdR, fdW, pos;

  if (argc != 3) {
    fprintf (stderr, "usage: %s inputFileName outputFileName\n", argv[0]);
    exit (-1);
  }

  fdR = open (argv[1], O_RDONLY);
  if (fdR==-1){
    fprintf(stderr, "Cannot open input file %s\n", argv[1]);
    exit(-1);
  }
  fdW = open (argv[2], O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
  if (fdW==-1){
    fprintf(stderr, "Cannot open output file %s\n", argv[2]);
    exit(-1);
  }

  pos = 0;
  while (read (fdR, &c, sizeof (char)) > 0)
    pos++;

  //printf ("Input FIle Size = %d bytes\n", pos);

  while (pos > 0) {
    pos--;
    lseek (fdR, pos*sizeof (char), SEEK_SET);
    read (fdR, &c, sizeof (char));
    write (fdW, &c, sizeof (char));
  };

  close (fdR);
  close (fdW);

 return (0);
}
```

2. Dato il seguente grafo di precedenza, realizzarlo utilizzando il **minimo** numero possibile di semafori. I processi rappresentati devono essere processi ciclici (con corpo del tipo `while(1)`). Gli archi tratteggiati indicano che solo due rami (tra i rami P_4, P_5/P_6, e P_7) devono essere eseguiti (arbitrariamente) per ogni iterazione (ovvero devono essere eseguiti P_4 e P_5/P_6, oppure P_7 e P_5/P_6 oppure ancora P_4 e P_7). Utilizzare le primitive `init`, `signal`, `wait` e `destroy`. Riportare il corpo dei processi ($P_1, \ldots, P_8$) e l'inizializzazione dei semafori.




```
init(S1, 1);
init(S2, 0);
init(S3, 0);
init(S457, 0);
init(S^457, 0);
init(S6, 0);
init(S8, 0);
```

```
destroy(S1);
destroy(S2);
destroy(S3);
destroy(S457);
destroy(S^457);
destroy(S6);
destroy(S8);
```

P1
```
while(1) {
    wait(S1);
    PRINT("P1");
    signal(S2);
}
```

P2
```
while(1) {
    wait(S2);
    PRINT("P2");
    signal(S3);
}
```

P3
```
while(1) {
    wait(S3);
    PRINT("P3");
    signal(S457);
    signal(S457);
}
```

P4
```
while(1) {
    wait(S457);
    PRINT("P4");
    signal(S8);
    wait(S^457);
}
```

P5
```
while(1) {
    wait(S457);
    PRINT("P5");
    signal(S6);
    wait(S^457);
}
```

P7
```
while(1) {
    wait(S457);
    PRINT("P7");
    signal(S8);
    wait(S^457);
}
```

P6
```
while(1) {
    wait(S6);
    PRINT("P6");
    signal(S8);
}
```

P8
```
while(1) {
    wait(S8);
    wait(S8);
    PRINT("P8");
    signal(S^457);
    signal(S^457);
    signal(S1);
}
```

3. Si descriva l'utilizzo dei segnali nel sistema operativo Unix/Linux con relativi vantaggi e svantaggi. Si descrivano in particolare le system call `signal`, `kill`, `pause` e `alarm`.

   Un programmatore desidera richiamare la funzione `npCompleteFunction` dall'interno dal programma principale. Tale funzione, pur essendo senza parametri, può richiedere tempi di esecuzione molto elevati. Volendo evitare di rimanere troppo a lungo in attesa della sua terminazione, il programmatore desidera fare in modo che:

   - Prima di chiamare la funzione stessa, il programma si predisponga ad auto-inviarsi e a gestire il segnale `SIG_CHLD` dopo 1000, 5000 e 10000 secondi.
   - Una volta richiamata la funzione, il programma chieda all'utente se si desidera continuare l'esecuzione della funzione oppure se si preferisce terminarla ad ogni ricezione del segnale `SIG_CHLD`.
   - Il programma esegua la scelta dell'utente terminando oppure continuando l'esecuzione della funzione `npCompleteFunction`.

   Si osservi che **non** è possibile utilizzare la system call `alarm` per realizzare il comportamento desiderato.

Signal
Kill
Pause
Alarm

Vedere lucidi o soluzioni esami precedenti.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <time.h> /* For function time() */

int random_range(int min, int max){
  return min + rand() / (RAND_MAX / (max - min + 1) + 1);
}

static void sig_handler(int signum) {
  if ( signum==SIGCHLD ) {
    int answer;
#if 0
    printf ("SIGCHLD: Do you want to stop the function (0/1): ");
    scanf ("%d", &answer);
#else
    answer = random_range(0,1);
    printf ("SIGCHLD: Do you want to stop the function (0/1): %d\n",
      answer);
#endif
    if ( answer==1 )
      exit(-1);
    else
      return;
  }
}

/* Mock function */
void npCompleteFunction(void) {
  int t, i;

  srand(time(NULL));
  t = random_range(1, 30);
  printf("RAND: %d\n", t);
  for (i=1; i<t; i++)
    sleep(1);

  printf("npCompleteFunction(): done\n");
}

int main() {
  int pid;

  signal(SIGCHLD, sig_handler);
  if ((pid = fork()) < 0) {
    perror("fork");
    exit(1);
  }
  if (pid == 0) {
    /* Child: Send 3 signals */
    sleep(5); /* sleep(1000) */
    kill(getppid(), SIGCHLD);
    sleep(10); /* sleep(5000) */
    kill(getppid(), SIGCHLD);
    sleep(10); /* sleep(10000) */
    kill(getppid(), SIGCHLD);
    exit(0);
  } else {
    /* Father */
    npCompleteFunction();
  }

  return 1;
}
```

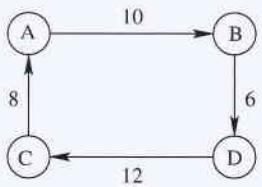
SOLUZIONE 2:

generare 3 figli (con 3 fork)

ognuno dei quali fa una sleep, una kill e la exit.

4. Un grafo diretto pesato è memorizzato in un file di testo con il seguente formato. Ogni riga del file specifica un arco del grafo. Per ogni arco sono indicati l'identificatore del vertice di partenza, quello del vertice di arrivo, e il peso dell'arco (valore intero). Il successivo è un esempio corretto di file e del grafo corrispondente.

```
A B 10
C A  8
B D  6
D C 12
```



Si implementi uno script BASH in grado di:

- Ricevere sulla riga di comando il nome di un file con la struttura precedentemente definita.
- Leggere da tastiera delle sequenze di identificatori di vertici separati da uno spazio.
- Verificare i vertici specificati formino un percorso sul grafo. In caso affermativo visualizzare il peso totale di tale percorso. In caso negativo terminare lo script.

Per il grafo riportato precedentemente, il successivo costituisce un esempio di esecuzione corretto:

```
script.bash nome_file.txt
> A B D C
path corretto, peso complessivo 28.
> C A B
path corretto, peso complessivo 18.
> A B C D
path non valido ... fine script.
```

```
#!/bin/bash

# Check arguments
if [ $# -ne 1 ]; then
    echo "Usage: es4.sh <input_file>"
    exit 1
fi

# Read sequences of vertices from the user
while read sequence
do

    # Initialize variables
    src=""
    tot=0

    # Parse sequence
    for dst in $sequence; do

        # Handle first vertex
        if [ "$src" == "" ]; then
            src=$dst
            continue
        fi

        # Check if there is an edge between current source and destination
        found=0
        while read from to weight; do
            if [ "$from" == "$src" ] && [ "$to" == "$dst" ]; then
                let "tot+=weight"
                found=1
                break;
            fi
        done < $1

        # Manage termination
        if [ $found -eq 0 ]; then
            echo "path non valido ... fine script."
            exit 0
        fi

        # Update source vertex
        src=$dst
    done

    # Print sequence total weight
    echo "path corretto, peso complessivo $tot"
done
```

```
#!/bin/bash

# Check arguments
if [ $# -ne 1 ]; then
    echo "Usage: es4b.sh <input_file>"
    exit 1
fi

# Read adjacency matrix in associative array
declare -A adj
while read from to weight; do
    adj[$from,$to]=$weight
done < $1

# Read sequences of vertices from the user
while read sequence
do

    # Initialize variables
    src=""
    tot=0

    # Parse sequence
    for dst in $sequence; do

        # Handle first vertex
        if [ "$src" == "" ]; then
            src=$dst
            continue
        fi

        # Get weight of the edge between the current source and destination

        weight=${adj[$src,$dst]}

        # Check if edge exists
        if [ "$weight" == "" ]; then
            echo "path non valido ... fine script."
            exit 0
        fi

        # Update total path weight
        let "tot+=weight"

        # Update source vertex
        src=$dst
    done

    # Print sequence total weight
    echo "path corretto, peso complessivo $tot"
done
```

5. **Per i candidati iscritti al corso nell'anno accademico 2018–2019.**
Si scriva un programma concorrente multi-thread che riceve sulla riga di comando due valori interi `n` e `p`.

Il thread principale crea `n` buffer globali (condivisi) di interi e dimensione costante arbitraria, esegue `p` thread produttori e un unico thread consumatore, quindi termina.

Ogni thread produttore itera all'infinito effettuando le seguenti operazioni: dorme un numero casuale di secondi variabile tra 0 e 3, seleziona in maniera casuale uno degli `n` buffer, memorizza in tale buffer il proprio identificatore (ovvero un numero intero incluso tra 0 e `p-1`) non appena gli è possibile.

Il consumatore itera all'infinito cercando di leggere valori dai buffer il più rapidamente possibile, ovvero senza rimanere bloccato in attesa indefinita su nessun buffer. I valori letti vengano visualizzati a video con il numero del buffer da cui sono stati letti.

Suggerimento: Il consumatore può utilizzare la system call `sem_trywait` per evitare di attendere indefinitamente la scrittura di un dato su un buffer vuoto.

Per i candidati iscritti al corso prima dell'anno accademico 2018–2019.
Scrivere uno script AWK in grado di risolvere il problema dell'esercizio numero 4.

```
BEGIN {
  while(getline < ARGV[1]) {
    adj[$1,$2]=$3
  }
  while(getline < "-") {
    tot=0
    for(i=1; i<NF; i++) {
      if(adj[$i,$(i+1)] != "") {
        tot += adj[$i,$(i+1)]
      } else {
        printf("path non valido ... fine script\n")
        exit 0
      }
    }
    printf("path corretto, peso complessivo %d\n", tot)
  }
}
```

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

#define BUFFER_SIZE 10

typedef struct prod_s {
  pthread_t tid;
  int id;
  sem_t *semaphores;
  int **buffers;
  int *cursors;
  int n;
} prod_t;

typedef struct cons_s {
  pthread_t tid;
  sem_t *semaphores;
  int **buffers;
  int *cursors;
  int n;
} cons_t;


int randInRange(int min, int max) {
  return (rand() % (max + 1 - min)) + min;
}

void* producer(void *arg) {
  prod_t *args = (prod_t *) arg;

  // Thread loop
  while(1) {

    // Sleep a random number of seconds
    int r = randInRange(0, 3);
    printf("Thread [%d] is sleeping for %d seconds\n", args->id, r);
    sleep(r);

    // Select buffer
    int i = randInRange(0, args->n-1);

    // Acquire buffer lock
    sem_wait(&(args->semaphores[i]));

    // Check if there is still space in the selected buffer
    if(args->cursors[i] < BUFFER_SIZE) {
      args->buffers[i][args->cursors[i]++] = args->id;
      printf("Thread [%d] has written its id into buffer %d\n", args->id, i);
    }

    // Release buffer lock
    sem_post(&(args->semaphores[i]));
  }
}

void* consumer(void *arg) {
  cons_t *args = (cons_t *) arg;

  // Thread loop
  int i = 0;
  while(1) {
```

```
    // Try to acquire buffer lock (non-blocking)
    //printf("Trying to acquire lock on buffer %d\n", i);
    if(sem_trywait(&(args->semaphores[i])) == 0) {

      // Skip buffer if empty
      if(args->cursors[i] == 0) {
        sem_post(&(args->semaphores[i]));
        continue;
      }

      // Extract and print last buffer element
      int id = args->buffers[i][--(args->cursors[i])];
      printf("Id %d fetched from buffer %d\n", id, i);

      // Release buffer lock
      sem_post(&(args->semaphores[i]));
    }

    // Update buffer index
    i = (i+1) % args->n;
  }
}

int main (int argc, char **argv) {
  pthread_t tid;

  // Read parameters
  int n = atoi(argv[1]);
  int p = atoi(argv[2]);

  // Allocate n buffers and their semaphores
  int **buffers = (int **) malloc(sizeof(int*)*n);
  int *cursors = (int *) malloc(sizeof(int)*n);
  sem_t *semaphores = (sem_t*) malloc(sizeof(sem_t)*n);
  for(int i=0;i<n;i++) {
    buffers[i] = (int*) malloc(sizeof(int)*BUFFER_SIZE);
    cursors[i] = 0;
    sem_init(&(semaphores[i]), 0, 1);
  }

  // Create p producer threads
  for(int j=0; j<p; j++) {
    prod_t *args = (prod_t *) malloc(sizeof(prod_t));
    args->semaphores = semaphores;
    args->buffers = buffers;
    args->cursors = cursors;
    args->n = n;
    args->id = j;
    pthread_create(&(args->tid), NULL, producer, (void *) args);
  }

  // Create consumer thread
  cons_t *args = (cons_t *) malloc(sizeof(cons_t));
  args->semaphores = semaphores;
  args->buffers = buffers;
  args->cursors = cursors;
  args->n = n;
  pthread_create(&(args->tid), NULL, consumer, (void *) args);

  // Wait consumer
  int *res;
  pthread_join(args->tid, (void**) &res);
}
```

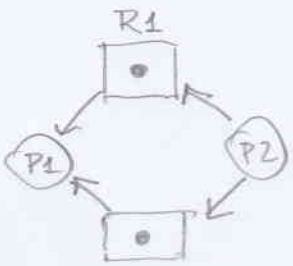
6. Si descriva il problema dello stallo modellandolo mediante grafo di allocazione delle risorse. Si indichino le tecniche per l'identificazione di uno stallo, mostrando un esempio in cui lo stallo sussiste e uno in cui non sussiste. Si indichi che cosa si intende per grafo di attesa e per arco di reclamo. Si indichi che cosa si intende per grafo di rivendicazione e per arco di richiesta.

Infine si illustri il meccanismo di prevenzione del deadlock basato sull'utilizzo gerarchico delle risorse. Si dimostri che tale tecnica previene il verificarsi di deadlock.

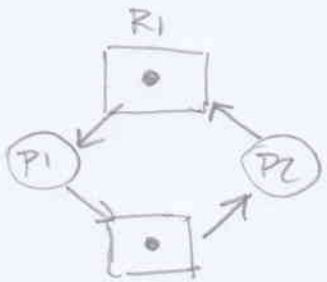
Stallo ≜ un P/T richiede una risorsa non disponibile, entra in uno **stato di attesa** che non termina più

Implica starvation

Graf allocazione ≜ V = processi ○ oppure risorse □
G(V,E) E = di richiesta (P → R) oppure di assegnazione (R → P)



No Stallo



Stallo

Graf di attesa ≜ dal precedente eliminando i vertici risorse

Graf di rivendicazione ≜ dal graf di allocazione aggiungendo archi di reclamo o intenzione di richiesta (- - - →)

NO NO NO

Rilevazione : ricerca cicli sul graf (condizione NON sufficiente per risorse multiple ⇒ Banchiere)

NO NO NO

Utilizzo gerarchico delle risorse :
la risorsa viene concessa se $F(R_{new}) > F(R_{old})$

Tale condizione previene l'attesa circolare :

$F(R_0) < \dots < F(R_n) < F(R_0)$
$F(R_0) < F(R_0)$
ASSURDO