Caselle riservate

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

Sistemi Operativi

Compito d'esame

17 Febbraio 2016

Matricola _____ Cognome _____ Nome _____

Docente: ◯ Quer ◯ Sterpone

L'unico materiale consultabile durante la prova scritta consiste nei tre formulari predisposti dal docente. Riportare i passaggi principali. L'ordine sarà oggetto di valutazione.
Durata della prova: 100 **minuti.**

1. Si supponga che un disco rigido sia costituito da 20 blocchi, che i blocchi liberi siano indicati con 0 e quelli occupati con 1, e che la situazione iniziale del disco sia la seguente:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	0	0	1	1	1	0	0	1	1	1	1	0	0	0	1	1	1	1	0

Indicare le principali caratteristiche delle metodologie di allocazione di file contigua, concatenata, indicizzata e FAT. Utilizzando tali tipi di allocazione descrivere come possono essere allocati nel disco precedentemente descritto i seguenti file: `File1` di dimensione uguale a 5 blocchi, `File2` di dimensione uguale a 3 blocchi.

Contigua Ogni file occupa un insieme contiguo di blocchi Per ciascun file il direttorio specifica lindirizzo del primo blocco b e la lunghezza del file n, cosicchè il file occupa i blocchi $(b, b+1, b+2, \ldots, b+n-1)$.

Vantaggi: strategia molto semplice, per ogni file si memorizzano poche informazioni, permette accessi sequenziali immediati, permette accessi diretti semplici.

Svantaggi: occorre decidere la politica di allocazione la tecnica spreca spazio (frammentazione esterna), crea problemi sull'allocazione dinamica dei file.

Nell'esempio: File 1 non può essere allocato; File 2 può essere allocato in 0, 1 e 2 (b=0, n=3).

Concatenata Ogni file può essere allocato gestendo una lista concatenata di blocchi. Il direttorio contiene un puntatore al primo e uno all'ultimo blocco del file. Ogni blocco contiene un puntatore al blocco successivo. I blocchi di ciascun file sono sparsi per l'intero disco.

Vantaggi: risolvere i problemi dell'allocazione contigua.

Svantaggi: risulta efficiente solo per accessi sequenziali, la memorizzazione dei puntatori richiede spazio ed è critica dal punto di vista dell'affidabilità.

Nell'esempio: File 1 può essere allocato come $0 \to 1 \to 2 \to 6 \to 7$; File 2 può essere allocato come $12 \to 13 \to 14$.

FAT Metodo utilizzatod da MS-DOS. Deriva dalla rappresentazione concatenata. Tabella con un elemento per ogni blocco del disco. La sequenza dei blocchi appartenenti a un file è individuata nel direttorio mediante la sequenza di puntatori presenti (direttamente) nella FAT (invece che allinterno dei blocchi stessi come nella rappresentazione concatenata).

Nell'esempoio si avrebbe una tabella di 20 posizioni contenenti i puntatori indicati nell'allocazione concatenta. File 1: $0 \to 1 \to 2 \to 6 \to 7$; File 2: $12 \to 13 \to 14$. Il direttorio conterrebbe l'indice del blocco di partenza dei due file (0 e 12).

Indicizzata Per permettere un accesso diretto (efficiente) ingloba tutti i puntatori in una tabella di puntatori detta blocco indice. Ogni file ha la sua tabella.

Nell'esempio: al File 1 sarebbe associata una tabella contenente: $0, 1, 2, 6, 7, -1, \ldots$. al File 2 una tabella del tipo $12, 13, 14, -1, \ldots$. Tali tabelle dovrebbero essere memorizzate in blocchi dati, però rimane solo il blocco 19 libero e quindi una delle due tabelle (e il relativo file) non sarebbe memorizzabile.

2. Scrivere un programma che riceva una valore intero n sulla riga di comando e crei n processi figlio. Tutti i figli di posizione pari (figli 0, 2, 4, etc.) devono stampare il proprio identificativo di processo e entrare in uno stato di pausa. Tutti i figli di posizione dispari (figlio 1, 3, 5, etc.) devono stampare in proprio identificativo di processo e terminare. Il padre, una volta creati tutti i figli, attende n intervalli di un secondo. Al termine di ciascun intervallo di posizione pari termina il figlio pari (in posizione corrispondente) utilizzando un segnale opportuno. Al termine di ciascun intervallo di posizione dispari aspetta la terminazione del figlio dispari (in posizione corrispondente).

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
static void signalHandler ();
static void child (int);
int
main (
  int argc,
  char *argv[]
  )
{
  int i, n;
  pid_t *pid;
  n = atoi (argv[1]);
  pid = (pid_t *) malloc (n * sizeof (pid_t));
  if (pid==NULL) {
    fprintf (stderr, "Allocation Error.\n");
    return (1);
  }
  if (signal(SIGUSR1, signalHandler) == SIG_ERR) {
    fprintf (stderr, "Signal Handler Error.\n");
    return (1);
  }
  for (i=0; i<n; i++) {
    pid[i] = fork ();
    if (pid[i]==0) {
      // Child
      child (i);
    }
  }
  // Father
  for (i=0; i<n; i++) {
    sleep (1);
    if ((i%2)==0) {
      fprintf (stdout, "Father Killing an Even Child (id=%d,pid=%d)\n", i, pid[i]);
      kill (pid[i], SIGUSR1);
    } else {
      fprintf (stdout, "Father Waiting an Odd Child (id=%d,pid=%d)\n", i, pid[i]);
      waitpid (pid[i], NULL, 0);
    }
  }
  return (0);
}
static void
signalHandler (
  void
) {
  return;
}
static void
child (
  int i
) {
  if ((i%2)==0) {
    fprintf (stdout, "  -- Even Child Pause (id=%d, pid=%d)\n", i, getpid());
    pause ();
  } else {
    fprintf (stdout, "  -- Odd Child Exit (id=%d, pid=%d)\n", i, getpid());
  }
  exit (1);
}
```

3. Si illustri il problema dei *Readers e Writers*. Se ne riporti la soluzione mediante primitive semaforiche nel caso di precedenza ai Readers illustrando il significato di ciascun semaforo. Che cosa si intende per "precedenza ai Readers"?

   Partendo dal problema precedente, si realizzi uno schema di sincronizzazione per una situazione in cui sono presenti due insiemi di Readers, denominati R_1 e R_2, e un insieme di Writers, denominati W. Ogni membro di R_1 (R_2) può accedere alla sezione critica insieme a altri membri di R_1 (R_2), però membri di R_1 e R_2, oppure R_1 e W, oppure R_2 e W devono accedere alla sezione critica in mutua esclusione.

   Dare precedenza ai reader significa privilegiare l'accesso dei reader rispetto a quello dei writer ovvero i reader non devono attendere a meno che un writer sia nella SC. Il semaforo `w` serve per realizzare la mutua esclusione tra Readers e Writes o tra diversi writers. Il semaforo `meR` serve per realizzare la mutua esclusione tra Readers nella fase di modifica di `nr`. `nr` conteggia il numero di Readers nella sezione critica.

```
Semafori e variabili:
nr = 0;
init (w, 1);
init (meR, 1);
```

```
Reader:
wait (meR);
  nr++;
  if (nr==1)
    wait (w);
signal (meR);
...
lettura
...
wait (meR);
  nr--;
  if (nr==0)
    signal (w);
signal (meR);
```

```
Writer:
wait (w);
...
scrittura
...
signal (w);
```

   Due insiemi di reader e un writer:

```
Semafori e variabili:
n1 = n2 = 0;
init (s1, 1);
init (s2, 1);
init (busy, 1);

Writer:
wait (w);
...
scrittura
...
signal (w);
```

```
Reader 1:
wait (s1);
  n1++;
  if (n1==1)
    wait (busy);
signal (s1);
...
lettura
...
wait (s1);
  nr--;
  if (nr==0)
    signal (busy);
signal (s1);
```

```
Reader 2:
wait (s2);
  n2++;
  if (n2==1)
    wait (busy);
signal (s2);
...
lettura
...
wait (s2);
  n2--;
  if (n2==0)
    signal (busy);
signal (s2);
```

4. Scrivere un script BASH in grado di ricevere sulla riga di comando due stringhe. La prima stringa identifica il nome di un file di ingresso, la seconda quello di uscita dello script.

   Il file di ingresso include il calendario di un mese con il formato rappresentato nella metà di sinistra della figura successiva (si noti che il formato è simile, ma semplificato, a quello ottenuto con il comando di shell `cal`). Tutti i campi sono costituiti da due caratteri e sono separati da un singolo spazio.

   Il file di uscita deve contenere lo stesso calendario ma con formato rappresentato nella metà di destra della figura successiva (si noti che il formato è simile, ma semplificato, a quello ottenuto con il comando di shell `ncal`). Lunghezza dei campi e spaziatura sono identici a quelli del file di ingresso.

```
Febbraio 2016
Lu Ma Me Gi Ve Sa Do
01 02 03 04 05 06 07
08 09 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 XX XX XX XX XX XX
```

```
Febbraio 2016
Lu 01 08 15 22 29
Ma 02 09 16 23 XX
Me 03 10 17 24 XX
Gi 04 11 18 25 XX
Ve 05 12 19 26 XX
Sa 06 13 20 27 XX
Do 07 14 21 28 XX
```

```
#!/bin/bash
head -1 $1 > $2
cat $1|tail -7 $1 |cut -f1 -d " " | tr '\n' " " >> $2
echo " " >> $2
cat $1|tail -7 $1 | cut -f2 -d " " | tr '\n' " " >> $2
echo " " >> $2
cat $1|tail -7 $1 | cut -f3 -d " " | tr '\n' " " >> $2
echo " " >> $2
cat $1|tail -7 $1 | cut -f4 -d " " | tr '\n' " " >> $2
echo " " >> $2
cat $1|tail -7 $1 | cut -f5 -d " " | tr '\n' " " >> $2
echo " " >> $2
cat $1|tail -7 $1 | cut -f6 -d " " | tr '\n' " " >> $2
echo " " >> $2
cat $1|tail -7 $1 | cut -f7 -d " " | tr '\n' " " >> $2
echo " " >> $2
```

5. Si scriva uno script AWK in grado di gestire gli acquisti di un piccolo magazzino, secondo le specifiche illustrate dall'esempio successivo.

prodotti.txt		fornitore1		fornitore2		fornitore3		uscita.txt			
coca-cola	pr01	pr02	1.25	pr02	0.95	pr06	1.10	coca-cola	pr01	fornitore3	1.00
fanta	pr02	pr03	1.15	pr01	1.05	pr03	0.99	fanta	pr02	fornitore2	0.95
sprite	pr03	pr06	0.90			pr01	1.00	sprite	pr03	fornitore3	0.99
breezer	pr04	pr05	1.45			pr02	1.10	breezer	pr04	–	–
gatorade	pr05							gatorade	pr05	fornitore1	1.45
orangina	pr06							orangina	pr06	fornitore1	0.90

Un primo file, di tipo "prodotti", indica i prodotti di cui il magazzino ha bisogno, indicando per ogni prodotto il nome e l'identificatore del prodotto.

Un insieme di file, di tipo "fornitori", specifica il costo dei vari prodotti presso diversi possibili fornitori, indicando per i prodotti forniti l'identificatore e il relativo prezzo.

Scrivere uno script AWK in grado di memorizzare su un file di output l'elenco dei prodotti (nome e identificatore), e per ciascuno di essi il fornitore più economico e il prezzo del prodotto. Nel caso un prodotto non sia fornito da nessun fornitore il nome del fornitore e il prezzo devono essere sostituiti dal carattere "-". Nel caso di prezzi identici la scelta del fornitore sia arbitraria.

Il nome di tutti i file gestiti dall'applicazione sono passati sulla riga di comando allo script stesso: il primo parametro identifica il file '"prodotto", l'ultimo il file di uscita, tutti i parametri intermedi (di numero ignoto) identificano i file "fornitori".

```
BEGIN {
 out = ARGV[ARGC-1]
 ARGC = ARGC-1
 while (getline < ARGV[1]) {
   pname[$2]=$1
 }
}
END {
 for (i=2; i<ARGC; i++) {
   print ARGV[i]
   while(getline < ARGV[i]) {
     if (!($1 in bestprice)) {
       bestprice[$1]=$2
       bestsuppl[$1]=ARGV[i]
     } else {
       if ($2 < bestprice[$1]) {
         bestsuppl[$1]=ARGV[i]
         bestprice[$1]=$2}
       }
     }
   }
   for (p in pname) {
      if (p in bestprice) {
       print pname[p] " " p " " bestsuppl[p] " " bestprice[p] >> out
     } else {
       print pname[p] " " p " - - " >> out
     }
  }
}
```

6. Si illustri l'*algoritmo del banchiere* riportandone la descrizione e lo pseudo-codice ottimizzato al caso illustrato di seguito (5 processi e 5 risorse).

   Analizzando l'esempio successivo, con processi $(P_1, \ldots, P_5)$ e risorse $(R_1, \ldots, R_5)$, si indichi se una richiesta di P_1 per $(0, 1, 1, 2, 0)$ potrebbe essere soddisfatta. Indicare inoltre se lo stato indicato è sicuro oppure non sicuro. Nel primo caso, si illustri l'ordine in cui i processi possono essere completati.

P.	Fine	Assegnate					Massimo					Necessità					Disponibilità				
		R_1	R_2	R_3	R_4	R_5	R_1	R_2	R_3	R_4	R_5	R_1	R_2	R_3	R_4	R_5	R_1	R_2	R_3	R_4	R_5
P_1	F	2	2	0	0	1	4	5	2	1	2						0	1	2	4	1
P_2	F	2	2	1	2	1	5	3	2	5	2										
P_3	F	2	1	3	0	1	2	2	3	4	1										
P_4	F	1	0	0	1	1	1	2	1	2	4										
P_5	F	0	0	1	3	2	1	1	6	3	2										

L'algoritmo del banchiere serve per evitare il deaclock nel caso di risorse con instanze multiple (altrimenti è sufficiente un algoritmo di determinazioe dei cicli sul grafo di assegnazione).

La richiesta di P_1 non può essere accettata in quanto per R_4 si chiedono 2 risorse mentre la necessità è 1.

Lo stato dell'esempio è sicuro. Sequenza sicura:
(disponibili = 0, 1, 2, 4, 1)
P_3 (disponibili = 2, 2, 5, 4, 2)
P_5 (disponibili = 2, 2, 6, 7, 4)
P_4 (disponibili = 3, 2, 6, 8, 5)
P_2 (disponibili = 5, 4, 7, 10, 6)
P_1 (disponibili = 7, 6, 7, 10, 7).

Verifica di una richiesta (con i e j che variano da 1 a 5):

```
se
  per ogni j Richieste[i][j]<=Necessita'[i][j]
  AND
  per ogni j Richieste[i][j]<=Disponibili[j]
  ALLORA
    per ogni j Disponibili[j]=Disponibili[j]-Richieste[i][j]
    per ogni j Assegnate[i][j]=Assegnate[i][j]+Richieste[i][j]
    per ogni j Necessita[i][j]=Necessita[i][j]-Richieste[i][j]
```

Verifica di uno stato (con i e j che variano da 1 a 5):

```
1.
pe rogni e i per ogni j Necessita[i][j]=Massimo[i][j]-Assegnate[i][j]
Fine [i] = falso per tutti i Pi
2.
Trova un Pi per cui
Fine[i]=falso && per ogni j Necessita[i][j]<=Disponibili[j]
Se tale i non esiste vai al passo 4
3.
per ogni j Disponibili[j]=Disponibili[j]+Assegnate[i][j]
Fine[i] = vero
Vai al passo 2
4.
Se Fine[i]=vero per tutti i Pi
Allora il sistema  in uno stato sicuro
```