SISTEMI OPERATIVI – 31 gennaio 2013
corso A nuovo ordinamento
e parte di teoria del vecchio ordinamento indirizzo SR

Cognome:_____ **Nome:**_____
Matricola:_____

1. Ricordate che non potete usare calcolatrici o materiale didattico, e che potete consegnare al massimo tre prove scritte per anno accademico.
2. Gli studenti a cui sono stati riconosciuti i 3 cfu di "linguaggio C" devono rispondere solo alle domande delle parti di teoria e di laboratorio Unix, e consegnare entro 1 ora e trenta minuti.
3. Gli studenti del vecchio ordinamento, indirizzo SR, devono rispondere solo alle domande della parte di teoria, e devono consegnare entro 1 ora.

ESERCIZI RELATIVI ALLA PARTE DI TEORIA DEL CORSO

ESERCIZIO 1 (5 punti)

a)
Riportate lo pseudocodice che descrive la corretta implementazione delle operazioni di WAIT e di SIGNAL, e dite che funzione hanno le system call usate nel codice.

Si vedano i lucidi della sezione 6.5.2.

b)
A cosa servono la WAIT e la SIGNAL, implementate come indicato nella domanda a)?

A risolvere i problemi di sincronizzazione fra processi senza far ricorso in modo pesante al busy-waiting.

c)
Riportate il diagramma di stato che descrive come, durante la vita di un processo, questo possa muoversi tra diversi stati.

Si vedano i lucidi della sezione 3.1.2

d)
Nel caso di un S.O. reale, che cosa "si sposta" tra uno stato e un'altro, e chi effettua lo "spostamento"?

Il process control block del processo. Il sistema operativo, che in ogni momento registra in quale stato si trova ciascun processo (new, ready, running, etc.)

ESERCIZIO 2 (5 punti)

Si consideri un sistema in cui in una tabella delle pagine di un processo l'indice più grande usabile nella tabella delle pagine di quel processo può essere 7FFF. Un indirizzo fisico del sistema è scritto su 25 bit, e la RAM è suddivisa in 4000 (esadecimale) frame.

a)

Quanto è grande, in megabyte, lo spazio di indirizzamento logico del sistema (esplicitate i calcoli che fate)?

4000(esadecimale) = 2^{14}, per cui un numero di frame è scritto su 14 bit, e la dimensione di un frame, e quindi di una pagina, è di $2^{\wedge 11}$ byte (25 - 14 = 11). Poiché il numero più grande di una pagina è 7FFF, ci possono essere al massimo $2^{\wedge 15}$ pagine, e lo spazio di indirizzamento logico è di $2^{\wedge 15}$ x $2^{\wedge 11}$ byte (pari a circa 64 megabyte).

b)
Elencate **tutte** le informazioni che deve contenere ciascuna entry di una tabella delle pagine di questo sistema, se il sistema usa l'algoritmo di rimpiazzamento della seconda chance migliorata

Il numero del frame che contiene la pagina corrispondente, il bit di validità della pagina, il reference bit, e il dirty bit.

c)
Nel caso non si verifichino mai page fault, qual è, in nanosecondi, il tempo medio di accesso in RAM del sistema se viene usato un TLB con un tempo di accesso di 5 nanosecondi, un hit-ratio del 95% e un tempo di accesso in RAM di 0,08 microsecondi? (è sufficiente riportare l'espressione aritmentica che fornisce il risultato finale)

Tmedio = 0,95 * (80+5) + 0,05 * (2*80 + 5) nanosecondi

d)
Elencate tre svantaggi dell'uso delle librerie statiche

Non possono essere condivise tra più processi, per cui occupano più spazio in RAM di quelle dinamiche.
Vengono caricate in RAM anche se non sono chiamate, per cui i processi partono più lentamente.
Se vengono aggiornate occorre ricompilare i programmi che le usano

e)
Commentate la seguente affermazione: *in un qualsiasi sistema operativo che implementi la memoria virtuale, l'effettivo tempo di esecuzione di un programma dipende principalmente dall'efficienza con la quale viene gestito il page fault.*

L'affermazione è falsa, perché dipende anche dal modo con cui il programma accede ai propri dati. Ad esempio, l'accesso per colonne ad array memorizzati per riga può produrre un elevatissimo numero di page fault, e quindi un forte rallentamento nell'esecuzione del programma. Ovviamente, anche la presenza di altri processi nel sistema può influenzare il tempo di esecuzione.

ESERCIZIO 3 (4 punti)

a) Dove sono memorizzati, da un punto di vista logico (ovviamente, fisicamente gli attributi sono in memorizzati in modo permanente in qualche blocco del disco), gli attributi di un file?

Nella directory che "contiene" il file, oppure in una struttura interna del sistema puntata da un puntatore contenuto della directory che "contiene" il file.

b) Considerate la seguente sequenza di comandi Unix (assumete che tutti i comandi lanciati possano essere correttamente eseguiti):

1: cd /tmp
2: mkdir mynewdir
3: cd mynewdir
4: echo “ciao” > pippo // crea un nuovo file di nome *pippo* contenente la stringa *ciao*
5: ln pippo pluto
6: ln –s pippo paperino
7: ln pluto topolino
8: rm pippo
9: cat topolino // *cat* stampa il contenuto del file passato come argomento
10: cat paperino

Dopo l’esecuzione di tutti i comandi:
qual è il valore del link counter nell’index-node associato al link fisico *topolino*? 2
qual è il valore del link counter nell’index-node associato al link fisico *mynewdir*? 2
cosa possiamo dire del link counter dell’index-node associato al link fisico *tmp*? Che è aumentato di 1, a causa dell’entry “..” inserita dentro la nuova sottocartella mynewdir.
Qual è l’output del comando numero 10? “no such file or directory”

c) Descrivete brevemente l’allocazione indicizzata dello spazio su disco, e la variante degli index-node (se preferite, usate degli opportuni disegni)

Vedere i lucidi della sezione 11.4.3

d) In quale caso la lettura dei dati di un file è più veloce se il file è memorizzato su un sistema raid anziche su un normale hard disk?

Quando i blocchi del file appartengono a strip memorizzati su dischi diversi del sistema raid, e possono quindi essere letti in parallelo.

ESERCIZI RELATIVI ALLA PARTE DI UNIX (6 punti)

ESERCIZIO 1 (3 punti)

(1.1) Illustrare due fra i seguenti comandi:

awk '/Mario/' anagrafica.txt *ls | sort -r* *ls -R* *ls -t .*

(1 punti)

Soluzione [slides 01_introduzione_UNIX.pdf, p. 16]

awk '/Mario/' anagrafica.txt . Awk cerca fra le linee del file *anagrafica.txt* quelle che contengono la stringa "Mario" e le stampa.

ls | sort -r lista il contenuto della directory corrente e ne stampa il contenuto invertendo l'ordinamento.

ls -R lista ricorsivamente il contenuto delle sottodirectory

ls -t lista il contenuto della directory corrente ordinando il contenuto sulla base dell'ultima modifica (i file modificati più di recente sono visualizzati prima), e poi lessicograficamente sulla base del nome.

(1.2) Cos'è un processo *zombie*, da cosa è causato e che tipo di problemi pone?

(1 punti)

Soluzione [slides 03_creazione_terminazione_processi.pdf, pp. 14-15]

Uno *zombie* è un processo che termina prima che il padre abbia effettuato una *wait*. Il kernel lo trasforma in *zombie* rilasciando gran parte delle risorse gestite dal figlio; tuttavia resta un'entry nella tabella dei processi, che ne registra il PID, lo stato di terminazione e le statistiche sull'utilizzo delle risorse. Gli *zombie* non possono essere uccisi con un segnale, neppure con SIGKILL, per assicurare al loro genitore la possibilità di eseguire una *wait()*, per esempio per conoscerne lo stato di terminazione.

Se il genitore termina senza fare la *wait()*, il processo *init* adotta il figlio ed esegue automaticamente una *wait()*, rimuovendo il processo zombie dal sistema.

Se il numero degli zombie cresce eccessivamente, si rischia di riempire la tabella dei processi, e questo può impedire la creazione di altri processi (paralizzando il funzionamento di un sistema). Poiché gli zombie non possono essere uccisi da un segnale, l'unico modo per rimuoverli dal sistema è uccidere il loro padre (o attendere la sua terminazione).

(1.3) Illustrare succintamente a cosa serve una IPC key, e quali metodi esistono per generare una key.

(1 punti)

Soluzione [slides 04_introduzione_IPC.pdf, pp. 26 e seguenti]

Le IPC key sono degli identificatori di risorse IPC. Il problema pertanto è non ottenere accidentamente l'identificatore di un altro oggetto IPC esistente, utilizzato da qualche altra applicazione. I tre metodi illustrati a lezione sono i seguenti:

- Scelta di una valore intero, che è tipicamente memorizzato in un header file incluso da tutti i programmi che usano l'oggetto IPC.
- Utilizzo della costante *IPC_PRIVATE* come valore della key nella *get* al momento della creazione dell'oggetto, che produce sempre un oggetto con una chiave unica. Esempio di invocazione:
  id = msgget(IPC_PRIVATE, S_IRUSR | S_IWUSR);
- Utilizzo della funzione *ftok()* per generare una key molto probabilmente unica. La *ftok()* ha prototipo

*key_t ftok(char *pathname, int proj);*

In particolare la funzione *ftok()* genera il valore key a partire da un *pathname* e dal valore *proj*, che permette di generare diverse key a partire dallo stesso pathname. Esempio di invocazione:

key = ftok("/mydir/myfile", 'x');

ESERCIZIO 2 (1 punti)

(2.1) Illustrare il significato della chiamata

```
shmget(IPC_PRIVATE, sizeof(registro), 0660);
```

Soluzione [slides 11_memoria_condivisa.pdf, p. 4]

shmget alloca un'area di memoria condivisa di dimensioni sufficienti a contenere una variabile di tipo "*registro*". Non viene fornita la chiave, per cui verosimilmente tale area sarà usata dal processo esecutore e dai suoi discendenti. Solo processi appartenenti all'utente o a membri del gruppo a cui appartiene il processo allocatore potranno accedere all'area.

ESERCIZIO 3 (2 punti)

Posto che la struttura sembuf è definita come segue

```
struct sembuf {
      unsigned short sem_num; //
      short sem_op;   //
      short sem_flg; //
};
```

Fornire un'implementazione *minimale* ma completa per una delle due funzioni *reserveSem* o *releaseSem* su un semaforo identificato dalla coppia *semId* e *semNum*. Commentare ogni istruzione di codice.

```
int reserveSem(int semId, int semNum) {
   struct sembuf sops;
   ...
}

int releaseSem (int semId, int semNum) {
   struct sembuf sops;
   ...
}
```

Soluzione [slides 10_semafori.pdf, p. 20]

```
int reserveSem(int semId, int semNum) {
      struct sembuf sops;

      sops.sem_num = semNum;  // preciso su quale semaforo si interviene
      sops.sem_op  = -1;     // operazione wait
      sops.sem_flg = 0;      // nessun flag particolare per l'operazione:
                             // potrebbe essere SEM_UNDO o IPC_NOWAIT
      semop(semId, &sops, 1); // esecuzione atomica delle operazioni indicate
                             // dall'array di operazioni sops sul set di semafori
                             // indicato da semId
}
```

```
int releaseSem (int semId, int semNum) {
        struct sembuf sops;

        sops.sem_num = semNum;
        sops.sem_op  = 1;
        sops.sem_flg = 0;
        semop(semId, &sops, 1);
}
```

ESERCIZI RELATIVI ALLA PARTE DI C (7 punti)

ESERCIZIO 1 (2 punti)

Si implementi la funzione con prototipo

```
int find_first(char * str, char c);
```

`find_first` è una funzione che cerca nella stringa `str` la prima occorrenza del carattere `c` e ne restituisce la posizione. Restituisce –1 se non lo trova.

```
int find_first(char * str, char c) {

  int len = strlen(str);
  for (int i=0; i<len; i++)
     if (str[i] == c) return i;
  return -1;

}
```

ESERCIZIO 2 (2 punti)

Si implementi la funzione con prototipo

```
node * creaLista(int N, float value[])
```

che crea una lista di N elementi di tipo `node`, e li inizializza ai corrispondenti valori nell'array `value`. Restituisce il puntatore alla testa della lista. NB: il primo elemento della lista creata sara` inizializzato con il primo elemento dell'array `value` (`value[0]`), il secondo con il secondo elemento dell'array `value` (`value[1]`), e cosi`via.

La struttura `node` è definita come segue:

```
typedef node* link;
typedef struct node {
     float value;
     link next;
} node;


node * creaLista(int N, float value[]) {

     node * head = NULL;
     for (int i=0; i<N; i++) {
          node * newNode = (node *) malloc(sizeof(node));
          newNode->value = value[N-i-1];
          newNode->next = head;
          head = newNode;
     }
     return head;

}
```

ESERCIZIO 3 (3 punti)

Data la struttura `node`, implementare la funzione con prototipo

```
float calcAverage(link head);
```

`calcAverage` è una funzione che calcola e restituisce la media dei valori del campo `value` degli elementi della lista. Se la lista e` vuota, la funzione restituisce 0.

Funzione da implementare

```
float calcAverage(link head) {

  float sum = 0.0; int n = 0;
  while (head != NULL) {
     sum += head->value; n++;
     head = head->next;
  }
  return (n == 0 ? 0 : sum/n);

}
```