SISTEMI OPERATIVI – 9 luglio 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.
4.

ESERCIZI RELATIVI ALLA PARTE DI TEORIA DEL CORSO

ESERCIZIO 1 (5 punti)

In un sistema operativo che adotta uno scheduling con diritto di prelazione, quattro processi arrivano al tempo indicato e consumano la quantità di CPU indicata nella tabella sottostante)

Processo	T. di arrivo	Burst
P1	0	14
P2	2	7
P3	5	3
P4	9	2

a)
Qual è il waiting time medio migliore (ossia ottimale) che potrebbe essere ottenuto per lo scheduling dei quattro processi della tabella? RIPORTATE IL DIAGRAMMA DI GANTT USATO PER IL CALCOLO. (lasciate pure i risultati numerici sotto forma di frazione, e indicate quali assunzioni fate)

Diagramma di GANT, assumendo come algoritmo di scheduling SJF preemptive:

(0)….P1 …(2) …. P2….(5)….P3….(8)….P2….(9)….P4…(11)….P2…(14)…P1….(26)

Waiting time medio:
P1 = (26 – 0) – 14 = 12;
P2 = (14 – 2) – 7 = 5;
P3 = (8 – 5) – 3 = 0;
P4 = (11 – 9) – 2 = 0;
waiting time medio = 17/4

b)
Riportate lo pseudocodice che descrive la corretta implementazione dell'operazione di SIGNAL, e dite che funzione ha la system call usata nel codice.

Si vedano i lucidi della sezione 6.5.2.

c)
All'interno di un sistema operativo, un certo processo P è correntemente in stato di "Running", e si sa che, avendo acquisito la CPU, non dovrà più rilasciarla volontariamente prima di aver terminato la propria esecuzione (in altre parole, non dovrà più eseguire operazioni di I/O, di sincronizzazione o di comunicazione con altri processi).

Quale/quali, tra gli algoritmi di scheduling **FCFS**, **SJF preemptive, SJF non-preemptive**, **round robin** garantisce/garantiscono che il processo P riuscirà a portare a termine la propria computazione? (motivate la vostra risposta, assumendo che SJF possa effettivamente essere implementato)

FCFS, e **round robin e SJF non-preemptive**. Infatti, nel caso di SJF preemptive potrebbe sempre arrivare in coda di ready un processo che deve usare la CPU per un tempo minore di quanto rimane da eseguire a P.

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 3FFF. Un indirizzo fisico del sistema è scritto su 25 bit, e la RAM è suddivisa in 2000 (esadecimale) frame.

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

2000(esadecimale) = 2^{13}, per cui un numero di frame è scritto su 13 bit, e la dimensione di un frame, e quindi di una pagina, è di $2^{\wedge 12}$ byte (25 - 13 = 12). Poiché il numero più grande di una pagina è 3FFF, ci possono essere al massimo $2^{\wedge 14}$ pagine, e lo spazio di indirizzamento logico è di $2^{\wedge 14}$ x $2^{\wedge 12}$ byte (pari a circa 64 megabyte).

b)
Il sistema deve usare una paginazione a più livelli? (motivate numericamente la vostra risposta).

Si. La tabella delle pagine più grande del sistema ha $2^{\wedge 14}$ pagine, e ogni entry contiene il numero di un frame, per cui sono necessari almeno due byte per ogni entry. La dimensione di quella tabella sarà quindi di $2^{\wedge 14}$ x 2 = $2^{\wedge 15}$ byte ossia maggiore della dimensione di un frame.

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 90% e un tempo di accesso in RAM di 0,09 microsecondi? (è sufficiente riportare l'espressione aritmentica che fornisce il risultato finale)

*Tmedio = 0,90 * (90+5) + 0,1 * (2*90 + 5) nanosecondi*

d)
Elencate tre vantaggi dell'uso delle librerie dinamiche

Possono essere condivise tra più processi, per cui occupano meno spazio in RAM.
Vengono caricate in RAM solo alla chiamata di una funzione, per cui i processi partono più velocemente.
Possono essere aggiornate senza dover 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 può dipendere pesantemente dal modo con cui il programma accede ai propri dati.

L'affermazione è vera. 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.

ESERCIZIO 3 (4 punti)

a) Descrivete (se preferite usando un disegno) il metodo di allocazione dello spazio su disco adottato dal sistema operativo Unix.

Si vedano i lucidi della sezione 11.4.3.

b) Scegliete una dimensione per i blocchi dell'hard disk e il numero di bit usati per scrivere il numero di un blocco, e calcolate di conseguenza la dimensione massima che può avere un file di quel sistema (è sufficiente riportare l'espressione che permette di calcolare la dimensione del file).

Blocco: 1024 byte, puntatore a blocco: 4 byte.

Dimesione massima: $10 \times 1k + 256 \times 1k + 256^2 \times 1k + 256^3 \times 1k$

c) come è fatta la struttura interna di una qualsiasi cartella unix, e quali entry sono sempre presenti in qualsiasi cartella unix? (se preferite usate un disegno)

È un array in cui ciascuna entry è formata dal nome di un file e dal numero di un index-node che contiene tutte le informazioni del file stesso. Qualsiasi cartella unix contiene sempre almeno le due entry "." E ".."

d) considerate i seguenti due comandi Unix:

1) ln A B
2) ln –s A B

dove A è un file esistente e B è un nuovo file. Si assuma che entrambi i comandi vengano eseguiti correttamente senza ritornare alcun messaggio di errore. Quale dei due comandi produce e un maggior consumo di spazio sull'hard disk, e perché?

Il comando 2. Infatti, un link simbolico produce sempre l'allocazione di un nuovo i-node, mentre un link fisico produce solo l'incremento del link-counter del file esistente. Entrambi generano una nuova entry nella cartella in cui viene eseguito il comando.

ESERCIZI RELATIVI ALLA PARTE DI UNIX (6 punti)

ESERCIZIO 1 (2 punti)

(1.1) Illustrare il significato e il risultato dell'esecuzione delle istruzioni presenti alle linee 2 e 3:

(1 punti)

```
1  int i=5;
2  printf( (i>>1) ? "%2d - giallo\n" : "%2d - rosso\n" , (i>>1));
3  printf( ((i>>2)&0) ? "%2d - verde\n" : "%2d - blu\n" , ((i>>2)&0));
```

Soluzione [slides 02_integrazione_linguaggio.pdf, p. 3]

Si tratta della stampa (`printf`) del risultato di un test, contenuto nelle espressioni `i>>1` e `((i>>2)&0)`.
La prima espressione testa se lo shift destro di *i* è diverso da 0: poiché il valore di tale espressione è 2, la condizione testata risulta vera ed è stampata la prima delle due stringhe: il risultato dell'esecuzione di linea 2 è quindi: *2 - giallo*. L'esecuzione dell'istruzione a linea 3 funziona in maniera analoga alla precedente: in questo caso però `((i>>2)&0)` vale 0, risultato dell'operazione *and* bitwise fra i numeri 2 e 0, quindi il risultato dell'esecuzione è: *0 - blu*.

(1.2) Illustrare le caratteristiche della classe di memorizzazione *register*.

(1 punti)

Soluzione [slides 02_integrazione_linguaggio.pdf, p. 9]

La classe di memorizzazione *register* ha come obiettivo l'aumento della velocità di esecuzione: segnala al compilatore che la variabile corrispondente dovrebbe essere memorizzata in registri di memoria ad alta velocità. Al fine di velocizzare l'esecuzione il programmatore può scegliere poche variabili alle quali viene fatto più frequentemente accesso (per esempio, variabili di ciclo e parametric delle funzioni).

Nei casi in cui il compilatore non può allocare un registro fisico, viene utilizzata la classe *automatica* (il compilatore dispone solo di una parte dei registri, che possono invece essere utilizzati dal sistema).

La variabile *register* è di norma dichiarata nel punto più vicino possibile a dove viene utilizzata, per consentire la massima disponibilità di registri fisici, utilizzati solo quando necessario. Il registro viene liberato all'uscita dal blocco.

ESERCIZIO 2 (2 punti)

(2.1) Illustrare sinteticamente il funzionamento della system call semget() a partire dal prototipo

*int **semget**(key_t key, int nsems, int semflg);*

e in particolare illustrare il funzionamento del secondo argomento, *nsems*.

(1 punto)

Soluzione [slides 07_semafori.pdf, slides 4 e seguenti]

La syscall in questione è utilizzata per creare o ottenere (nel caso esista già) l'identificatore di un set di semafori: restituisce l'id del set di semafori quando va a buon fine, e -1 in caso di errore.
Nel caso si stia utilizzando la *semget()* per creare un nuovo set di semafori, *nsems* specifica il numero di semafori in quel set, e deve essere maggiore di 0; se invece stiamo utilizzando la *semget()* per ottenere l'identificatore di un set di semafori, *nsems* deve essere minore o uguale alla dimensione di

quel set.

(2.2) Illustrare sinteticamente il funzionamento della system call *pipe* a partire dal prototipo sottostante, e spiegare come il pipe è utilizzato da ciascun processo

*int **pipe**(int fd[2]);*

(1 punti)

Soluzione [05_pipes_FIFOs.pdf, pagine 9 e sgg.]

La system call crea un pipe che permette la comunicazione fra due processi (uno antenato dell'altro), che eseguono due diversi comandi. Un pipe è uno strumento di comunicazione fra due processi: uno scrive uno stream di byte su un descrittore di file, e l'altro legge lo stream di byte da un descrittore di file. I dati passano in maniera unidirezionale attraverso il pipe in sequenza: i byte sono letti nello stesso ordine in cui sono stati scritti.

In un caso di utilizzo tipico, durante una *fork()* il processo figlio eredita copia dei descrittori di file del genitore. Subito dopo la *fork()* uno dei due processi chiude il proprio lato del pipe in lettura e l'altro chiude il proprio in scrittura, realizzando così l'unidirezionalità della comunicazione. Come con gli altri descrittori di file, è possibile utilizzare le system call *read()* e *write()* per effettuare operazioni di I/O sul pipe.

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 07_semafori.pdf, p. 18 e seguenti]

```
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

ESERCIZIO 1 (2 punti)

Si implementi la funzione con prototipo

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

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

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

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

}
```

ESERCIZIO 2 (2 punti)

Si implementi la funzione con prototipo

```
node ** creaVect(int N)
```

che alloca un vettore di N puntatori a elementi di tipo `node*`, li inizializza al valore NULL e restituisce il vettore

```
node ** creaVect(int N) {

     node ** vect = (node **) malloc(N * sizeof(node *));
     for (int i=0; i<N; i++) vect[i] = NULL;
     return vect;

}
```

ESERCIZIO 3 (3 punti)

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

```
link find_Max_inList(link head);
```

`find_Max_inList` è una funzione che trova l'elemento con valore piu` grande nel campo value e restituisce l'elemento della lista corrispondente. Se la lista e` vuota, la funzione restituisce NULL.

La struttura `node` è definita come segue:

```
typedef node* link;
```

```
typedef struct node {
     int value;
     link next;
} node;
```

Funzione da implementare

```
link find_max_inList(link head) {

  link max = head;
  while (head != NULL) {
     if (max->value < head->value) max = head;
     head = head->next;
  }
  return max;

}
```