SISTEMI OPERATIVI – 13 giugno 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) Si consideri il problema dei produttori e consumatori, con buffer limitato ad *m* elementi, dove i codici del generico produttore e del generico consumatore sono i seguenti:

semafori necessari con relativo valore di inizializzazione:
semaphore mutex = 1;
semaphore full = 0;
semaphore empty = m;

"consumatore"	*"produttore"*
repeat	**repeat**
	<produci dato>
wait(full)	*wait(empty)*
wait(mutex)	*wait(mutex)*
<preleva dato dal buffer>	
	<inserisci dato nel buffer>
signal(mutex)	*signal(mutex)*
signal(empty)	*signal(full)*
<consuma dato>	
forever	**forever**

Inserite le opportune operazioni di wait e signal necessarie per il corretto funzionamento del sistema, indicando anche i semafori necessari ed il loro valore di inizializzazione.

b) Riportate lo pseudocodice che descrive l'implementazione dell'operazione di Wait. In che modo l'uso combinato di Wait e Signal evita lo spreco di tempo di CPU che è invece tipico dei meccanismi di sincronizzazione basati su busy waiting?

Si vedano i lucidi della sezione 6.5.2

c) per quale ragione è possibile implementare le sezioni critiche contenute nel codice della Wait usando la disabilitazione degli interrupt, ma questa soluzione non è adottabile per le sezioni critiche dei processi degli utenti?

Perché i processi utenti non garantiscono che la disabilitazione degli interrupt avvenga per un tempo limitato, e neppure che gli interrupt vengano riabilitati all'uscita di una sezione critica.

d) Spiegate perché gli algoritmi di schedulino FCFS e SJF pre-emptive (supponendo sia implementabile) non sono adatti per implementare un sistema time saring

Perché non garantiscono che un processo possa entrare in esecuzione in una quantità di tempo limitata

ESERCIZIO 2 (5 punti)

Un sistema con memoria paginata usa un TLB con un hit-ratio del 90%, e un tempo di accesso di 10 nanosecondi. Un accesso in RAM richiede invece 0,09 microsecondi.

a) Qual è, in nanosecondi, il tempo medio di accesso (Medium Access Time) in RAM (esplicitate i calcoli che fate)?

ma = 0,90 * (90+10) + 0,10 * (2*90 + 10) = 90 + 19 = 109 nanosecondi

b) Il sistema viene ora dotato di memoria virtuale, usando come algoritmo di rimpiazzamento quello della *seconda chance*. Si supponga pari a 5 millisecondi il tempo necessario a gestire un page fault. Se vogliamo una degradazione massima dell'Effective Access Time del 10% rispetto alla versione del sistema senza memoria virtuale, quale deve essere il valore massimo di "p", ossia la probabilità che si verifichi un page fault? (esplicitate il ragionamento e le formule che usate, ed è sufficiente esprimere "p" mediante la formula che permette di calcolarlo)

A partire dalla formula:

eat = (1-p)·ma + p·"tempo di gestione del page fault"

usando i dati del problema, sapendo che vogliamo eat massimo pari a ma + 10% ma, abbiamo:

max eat = 109 + 0,1·109 = 119,9 ≅ 120 ≥ (1-p) ·109 + p · 5.000.000

e risolvendo rispetto a p abbiamo:

120 ≥ 109 + p · (5.000.000 -109);
p < (120 -109) / (5.000.000 -109) (ossia all'incirca meno di un page fault ogni 500.000 riferimenti in memoria)

E' possibile che un processo di questo sistema sistema si veda aumentare il numero di frame che usa e contemporaneamente aumenta anche il numero di page fault generati dal processo?

Si, perché nel caso peggiore l'algoritmo si comporta come FIFO, che soffre dell'anomalia di Belady.

c) Quali informazioni conterrà ciascuna entry di una page table di un sistema che usa l'algoritmo della seconda chance?

Numero di un frame, bit di validità, bit di riferimento.

ESERCIZIO 3 (4 punti)

Un hard disk ha la capienza di 2^{38} byte, ed è formattato in blocchi da 2048 byte.

a) Quanti accessi al disco sono necessari per leggere l'ultimo blocco di un file A della dimensione di 8192 byte, assumendo che sia già in RAM il numero del primo blocco del file stesso e che venga adottata una allocazione concatenata dello spazio su disco? (motivate la vostra risposta)

5. Ogni blocco infatti memorizza 2044 byte di dati più 4 byte di puntatore al blocco successivo (infatti, $2^{38}/2^{11} = 2^{27}$), per cui sono necessari 5 blocchi per memorizzare l'intero file.

b) Qual è lo spreco di memoria dovuto alla frammentazione interna nella memorizzazione di A (motivate la risposta)?

L'hard disk è suddiviso in $2^{38}/2^{11} = 2^{27}$ blocchi, sono necessari 4 byte per memorizzare un puntatore al blocco successivo, e ogni blocco contiene 2044 byte di dati. Il quinto blocco memorizzerà quindi 16 byte del file, e la frammentazione interna corrisponde a 2048 – 16 = 2032 byte (2028 se si considerano non sprecati i 4 byte del quinto blocco che contengono il puntatore, non utilizzato, al blocco successivo)

c) Quanto sarebbe grande, in megabyte, la FAT di questo sistema? (motivate numericamente la vostra risposta)

La FAT è un array con una entry per ciascun blocco dell'hard disk e che contiene il numero di un blocco, per cui: 2^{27} x 2^2 byte = 512 megabyte

Disegnate la FAT di un hard disk formato da 2^4 blocchi e contenente un unico file memorizzato, nell'ordine, nei blocchi 9, 5, 11, 3. Indicate anche dove è memorizzato il numero del primo blocco del file.

Si veda la figura 11.7 della sezione 11.4.2 dei lucidi.

ESERCIZI RELATIVI ALLA PARTE DI UNIX (6 punti)

ESERCIZIO 1 (2 punti)

(1.1) Illustrare il significato delle istruzioni alle line da 4 a 6:

(1 punti)

```
1: #!/bin/bash
2: VAR=PROVA
3:
4: echo "il valore è $VAR"
5: echo `ls -l $VAR`
6: echo '$VAR'
```

Soluzione [slides 14_intro_bash.pdf, pp. 14 e sgg.]

L'istruzione a linea 4 stampa la stringa contenuta fra apici doppi sostituendo a $VAR il suo valore; l'istruzione a linea 5 effettua la sostituzione, interpreta come comando e restituisce il risultato dell'esecuzione del comando (che dipende dalla presenza o meno del file in questione); l'istruzione a linea 6 non effettua alcuna sostituzione, e viene pertanto stampata letteralmente.

(1.2) Analizzare la seguente definizione di macro; illustrare cosa accade durante l'invocazione sottostante, e quale risultato viene restituito.

(1 punti)

```
#define MAX(x, y) x > y ? x : y

...

int a=1, b=2;
int result = 3 + MAX(a,b);
```

Soluzione [slides 02_integrazione_linguaggio_e_ripasso.pdf, pp. 41 e sgg.]

Il preprocessore sostituisce le direttive al preprocessore (quindi anche le macro) prima della compilazione, quindi il codice che arriva al compilatore è

```
int result = 3 + a > b ? a : b ;
```

La somma *3+a* è eseguita prima del test; poiché *3+a* è maggiore di *2*, la variabile *result* viene assegnata con il valore di *a*, cioè *1*.

ESERCIZIO 2 (2 punti)

(2.1) A cosa serve la system call *waitpid()*? Nel rispondere considerare eventuali argomenti e/o valori restituiti.

(1 punti)

Soluzione [slides 03_creazione_terminazione_processi.pdf, pp. 34 e sgg.]

La *wait* sospende l'esecuzione del processo chiamante finché uno dei figli non termina la propria esecuzione e restituisce il process ID del figlio che ha terminato la propria esecuzione. Permette semplicemente di attendere che uno dei figli del chiamante termini. La *waitpid* consente di specificare il *pid* del figlio di cui attendere la terminazione. Il prototipo è

*pid_t waitpid(pid_t pid, int *status, int options);*

In particolare,
- se pid > 0, attendi per il figlio con quel pid.
- se pid == 0, attendi per qualsiasi figlio nello stesso gruppo di processi del chiamante (padre).
- se pid < -1, attendi per qualsiasi figlio il cui process group è uguale al valore assoluto di pid.
- se pid == -1, attendi per un figlio qualsiasi. La chiamata *wait(&status)* equivale a *waitpid(–1, &status, 0)*.

(2.2) Illustrare la system call *signal()* a partire dal prototipo sottostante, e spiegare in breve cos'è un handler di segnali.

```
void (*signal(int sig, void (*handler)(int))) (int);
```

(1 punti)

Soluzione [slides 09_syscall_kill_signal.pdf, pp. 26 e sgg.]

I sistemi UNIX forniscono due modi per cambiare la disposizione di un segnale: *signal()* e *sigaction()*. La system call *signal()* è l'API originale per assegnare la disposizione di un segnale.

Il primo argomento della *signal()*, *sig*, identifica il segnale di cui vogliamo modificare la disposizione. Il secondo argomento, *handler*, è l'indirizzo della funzione da invocare quando questo segnale è inviato.

Questa funzione non restituisce (è *void*) e prende un intero. Quindi un handler di segnali ha la seguente forma generale:

```
void handler(int sig) {
  /* codice dell'handler */
}
```

Il valore di ritorno di *signal()* è la precedente disposizione del segnale. Come l'argomento dell'handler, questo è un puntatore a funzione che non restituisce nulla e che prende un intero come argomento.

ESERCIZIO 3 (2 punti)

Illustrare il funzionamento della system call *msgsnd()* e fornire un esempio di utilizzo.

Soluzione [slides 06_code_messaggi.pdf, pp. 34 e seguenti]

La syscall *msgsnd()* invia un messaggio a una coda di messaggi. Il suo prototipo è

int **msgsnd**(int *msqid*, const void **msgp*, size_t *msgsz*, int *msgflg*);

restituisce 0 in caso di successo, -1 in caso di errore. Il primo argomento è un intero che funge da l'identificatore della coda; il secondo è un puntatore a una struttura definita dal programmatore utilizzata per contenere il messaggio inviato. L'argomento *msgsz* indica la dimensione del messaggio (espresso in byte), mentre l'ultimo argomento è una bit mask di flag che controllano l'operazione di invio. Per esempio utilizzando IPC_NOWAIT, nel caso la coda di messaggi sia piena, invece di bloccarsi in attesa che si renda disponibile dello spazio, la *msgsnd()* restituisce immediatamente (con errore *EAGAIN*).

Un'invocazione tipica della syscall è quindi

msgsnd(m_id, &q, sizeof(q), IPC_NOWAIT)

ESERCIZI RELATIVI ALLA PARTE DI C

ESERCIZIO 1 (2 punti)

Si implementi la funzione con prototipo

```
int binary_search(int array[], int n, int what);
```

`binary_search` è una funzione che implementa la ricerca binaria del valore `what` nell'array `array` che si suppone ordinato per valori DECRESCENTI e restituisce la posizione. Restituisce –1 se non lo trova. `n` e` la dimensione dell'array `array`.

```
int binary_search (int array[], int n, int what) {

  int i = 0, j = n, k;
  while (i<j) {
     k = (i+j)/2;
     if (array[k] > what) i=k+1;
     else if (array[k] < what) j=k-1;
     else return k;
  }
  return -1;

}
```

ESERCIZIO 2 (3 punti)

Si implementi la funzione con prototipo

```
node * creaLista(int N, int value)
```

che crea una lista di N elementi di tipo `node`, e li inizializza al valore `value` e restituisce il puntatore alla testa della lista

La struttura `node` è definita come segue:

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


node * creaLista(int N, int value) {

     node * head = NULL;
     if (N > 0) {
          head = (node *) malloc(sizeof(node));
          head->value = value;
          head->next = creaLista(N-1, value);
     }
     return head;
```

}

ESERCIZIO 3 (2 punti)

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

```
float average(link head);
```

`average` è una funzione che calcola la media dei valori nel campo `value` della lista. Se la lista e` vuota, la funzione restituisce 0.

Funzione da implementare

```
float average(link head) {

  float sum = 0.0, n = 0;
  while (head != NULL) {
     sum += head->value;
     n++;
     head = head->next;
  }
  if (n>0) return sum/n;
  else return 0.0;

}
```