SISTEMI OPERATIVI
24 settembre 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 lettori e scrittori visto a lezione, dove i codici del generico scrittore e del generico lettore sono riportati qui di seguito.

Inserite le istruzioni mancanti necessarie per il funzionamento del sistema secondo la soluzione vista a lezione, indicando anche il semaforo mancante ed il suo valore di inizializzazione.

semafori e variabili condivise necessarie con relativo valore di inizializzazione:
semaphore mutex = 1;
semaphore scrivi = 1;
int numlettori = 0;

"scrittore"
{
wait(scrivi);
Esegui la scrittura del file
signal(scrivi)
}

"lettore"
{
wait(mutex);

numlettori++;

if numlettori == 1 **wait(scrivi);**

signal(mutex);

... leggi il file ...

wait(mutex);

numlettori--;

if numlettori == 0 **signal(scrivi);**

signal(mutex);

b) La soluzione del problema dei lettori e scrittori vista a lezione garantisce l'assenza di starvation? (motivate la vostra risposta)

No, infatti un qualsiasi processo scrittore potrebbe dover attendere all'infinito senza riuscire a ad entrare in sezione critica. Al contrario i processi lettori sono liberi da starvation (in altre parole, non è garantita l'attesa limitata)

c) Elencate quali tecniche usa un sistema operativo, in collaborazione con l'hardware del processore, per proteggersi da funzionamenti impropri (accidentali o voluti) dei programmi utente.

1) Uso di una doppia modalità di esecuzione delle istruzioni, per cui le istruzioni "delicate" possono essere eseguite solo per conto del sistema operativo. 2) uso di un timer hardware che può essere inizializzato solo dal SO e allo scadere del quale il controllo della macchina torna al SO 3) uso di tecniche per limitare le aree della RAM a cui possono accedere i programmi utente: coppia di registri base-limit, o base-offset, o paginazione della memoria.

d) Riportate lo pseudocodice che descrive l'implementazione dell'operazione di Wait, e dite che informazione fornisce il valore corrente del semaforo

Per lo pseudocodice si vedano i lucidi della sezione 6.5.2

Se il S->valore < di zero il suo valore assoluto ci dice quanti processi sono sospesi su quel semaforo. In caso contrario il valore ci dice quanti processi possono eseguire la wait su quel semaforo senza venire sospesi.

ESERCIZIO 2 (5 punti)

Un sistema con memoria paginata usa un TLB con un hit-ratio del 95%, e un tempo di accesso di 20 nanosecondi. Un accesso in RAM richiede invece 0,08 microsecondi. Quando la pagina indirizzata non si trova nel TLB, si verifica un un page fault nel 20% dei casi, e il tempo di gestione del page fault è di 1 millisecondo (indipendentemente dal fatto che la pagina vittima abbia il dirty bit a 0 o a 1).

a) Qual è, in nanosecondi, il tempo medio di accesso in RAM (esplicitate i calcoli che fate. E' sufficiente riportare l'espressione aritmetica che permette di calcolare il risultato finale)?

Tmedio = 0,95 * (80+20) + 0,04 * (2*80 + 20) 0,01 * 1.000.000 =
= 95 + 7,2 + 10000 nanosecondi

b) Il sistema usa come algoritmo di rimpiazzamento quello della *seconda chance migliorato*. Spiegate cos'è l'anomalia di Belady e dite se questo algoritmo ne soffre.

Un algoritmo di rimpiazzamento soffre dell'anomalia di Belady se, aumentando il numero di frame assegnati ad un processo, aumenta il numero di page fault per una certa stringa di riferimenti.

L'algoritmo della seconda chance migliorato ne soffre, perché nel caso peggiore si riduce ad un algoritmo FIFO, che è soggetto all'anomalia di Belady.

c) Gli algoritmi della seconda chance, e della seconda chance migliorato, sono approssimazioni di quale altro algoritmo di rimpiazzamento? Perché, di solito, non si implementa direttamente questo algoritmo?

Sono approssimazioni di LRU. LRU non viene normalmente implementato perché richiede dell'hardware troppo sofisticato e costoso.

d) Perché si implementa la memoria virtuale?

Per poter eseguire programmi che hanno uno spazio di indirizzamento logico maggiore di quello fisico, e per poter avere in esecuzione contemporaneamente programmi che, insieme, occupano uno spazio maggiore dello spazio di indirizzamento fisico.

e) Descrivete brevemente il più grave problema che si può presentare in un sistema che implementa la memoria virutale

Thrashing, in cui i processi passano la maggior parte del tempo a generare page fault e a rubarsi pagine in RAM senza portare avanti la loro computazione.

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) Se si adottasse una allocazione indicizzata dello spazio su disco, quanti accessi al disco sarebbero necessari per leggere l'ultimo byte di un file B grande 1200K byte (specificate quali assunzioni fate nel rispondere a questa domanda e motivate la vostra risposta)?

Poiché sono necessari 4 byte per scrivere il numero di un blocco, in un blocco indice possono essere memorizzati 512 puntatori a blocco, e con un blocco indice possiamo indirizzare in tutto (circa) $2^9 * 2^{11} = 2^{20}$ = 1 Megabyte. Un solo blocco indice non è quindi sufficiente a memorizzare B. Assumendo una allocazione indicizzata concatenata (ma si ottengono gli stessi risultati con una allocazione indicizzata gerarchica) usando un secondo blocco indice è possibile memorizzare l'intero file. Se il numero del primo blocco indice è già in RAM, per leggere l'ultimo carattere del file sono necessari 3 accessi al disco: lettura del primo blocco indice, lettura del secondo blocco indice, lettura dell'ultimo blocco del file.

d) Se il file B della domanda precedente fosse memorizzato sul file system di un sistema operativo Unix, quanti accessi alla memoria secondaria sarebbero necessari per leggere l'ultimo byte del file? (specificate il ragionamento e le assunzioni che fate, assumendo sempre che l'hard disk sia suddiviso in blocchi da 2048 byte)

4. Infatti, i primi dieci puntatori a blocco dell' i-node di B indirizzano in tutto 20K byte di dati. Il puntatore *single indirect* indirizza altri 1024K byte, e il puntatore *double indirect* permette di indirizzare i restanti byte del file. Assumendo che sia già in ram in numero dell'i-node di B abbiamo: un accesso al disco per leggere l'i-node di B più altri 3 accessi per seguire i due livelli di doppia indirezione e infine leggere il blocco che contiene l'ultimo byte di B (si veda la figura di un index-node della sezione 11.4.3 dei lucidi)

ESERCIZI RELATIVI ALLA PARTE DI UNIX (6 punti)

ESERCIZIO 1 (2 punti)

(1.2) Data l'istruzione int a = 8, b = 15; indicare il risultato delle istruzioni seguenti:

(1 punti)

```
printf("%d\n", ((a >> 2) | 8) >> 2);   ____________
printf("%d\n", ((a | b) ^ b ));        ____________
```

Soluzione [02_integrazione_linguaggio.pdf, slide 2 e sgg.]

```
printf("%d\n", ((a >> 2) | 8) >> 2 );  _____2_____
printf("%d\n", ((a | b) ^ b ) );       _____0_____
```

(1.3) Illustrare il significato della riga sottostante, ipotizzando che `a` e `c` siano nomi di file e `b` sia un programma

(1 punti)

```
a < b > c
```

Soluzione [slides 01_introduzione_UNIX.pdf, slide 18]

Si tratta di un esempio di combinazione di redirezione dell'input e dell'output: in questo caso il comando `b` prende in input il file `a` e dopo averlo elaborato invia il proprio output in un nuovo file chiamato `c`.

ESERCIZIO 2 (3 punti)

(2.1) Illustrare il funzionamento della system call
int **execve**(const char **pathname*, char *const *argv[]*, char *const *envp[]*);

(2 punti)

Soluzione [03_controllo_processi.pdf, slide 2 e sgg.]

La system call *execve()* carica un nuovo programma nella memoria di un processo. Con questa operazione, il vecchio programma è abbandonato, e lo stack, i dati, e lo heap del processo sono sostituiti da quelli del nuovo programma. Dopo avere eseguito l'inizializzazione del codice, il nuovo programma

inizia l'esecuzione dalla propria funzione *main()*. Non restituisce in caso di successo; restituisce -1 in caso di fallimento.

Esistono varie funzioni di libreria, tutte con nomi che iniziano con *exec-*, basate sulla system call *execve()*: ciascuna di queste funzioni fornisce una diversa interfaccia alla stessa funzionalità.

L'argomento *pathname* contiene il pathname del programma che sarà caricato nella memoria del processo. L'argomento *argv* specifica gli argomenti della linea di comando da passare al nuovo programma. Si tratta di una lista di puntatori a stringa, terminati da puntatore a *NULL*. Il valore fornito per *argv[0]* corrisponde al nome del comando. Tipicamente, questo valore è lo stesso del basename (i.e., l'ultimo elemento) del pathname. L'ultimo argomento, *envp*, specifica la environment list per il nuovo programma. L'argomento *envp* corrisponde all'array environment; è una lista di puntatori a stringhe (terminata da ptr a NULL) nella forma *name=value*.

(2.1) Spiegare il significato della chiamata

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

(1 punti)

Soluzione [08_memoria_condivisa.pdf, slide 4 e sgg.]

---

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 (1 punti)

(1 punto)

Illustrare l'istruzione seguente, spiegando ruolo e tipo dei parametri attuali della chiamata:

```
if( ( msgrcv(m_id, &q, sizeof(q), getpid(), 0 ) ) == -1)
     do_something ... ;
```

Soluzione [07_code_messaggi.pdf, slide 13 e sgg.]

La system call *msgrcv* è utilizzata per prelevare un messaggio dalla coda identificata dall'intero *m_id*; il messaggio è copiato nel buffer riferito da *q*, la cui dimensione massima è costituita da *sizeof(q)* byte. Il penultimo parametro, che costituisce il *msgtype* permette di stabilire un criterio per prelevare i messaggi: se il *msgtype* è impostato a 0, è prelevato il primo messaggio; se il *msgtype* è maggiore di 0 è prelevato il primo messaggio il cui *mtype* è eguale al *msgtype* (nell'esempio attuale, il processo chiamante seleziona un messaggio con *msgtype* pari al *pid* del processo stesso); infine, se il *msgtype* è minore di 0, la coda funziona come una coda con priorità: in questo caso è prelevato il primo messaggio con il più basso *mtype* minore o uguale al valore assoluto di *msgtype*. L'ultimo argomento è una maschera di bit che può assumere alcuni valori predefiniti, come per esempio *IPC_NOWAIT* o *MSG_NOERROR*.

ESERCIZI RELATIVI ALLA PARTE DI C

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

}
```

ESERCIZIO 3 (3 punti)

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

```
link find_Min_inList(link head);
```

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

Funzione da implementare

```
link find_Min_inList(link head) {

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

}
```