SISTEMI OPERATIVI – 9 giugno 2016
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 riportati qui di seguito. 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 (in grassetto sono indicate le parti mancanti da inserire).

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

b) si supponga la presenza di m+1 produttori che cercano di inserire "contemporaneamente" nel buffer un elemento, mentre nessun consumatore ha ancora iniziato le sue operazioni. Cosa succede a ciascuno degli m+1 produttori?

L'*m*+1-esimo produttore si addormenta sul semaforo *empty*, mentre i precedenti *m* produttori riescono a superarlo. Di questi, *m*−1 si addormentano su *mutex*, mentre un produttore lo supera e inserisce un elemento nel buffer.

c) Riportate il diagramma di stato della vita di un processo in un sistema timesharing, etichettando correttamente ogni stato e ogni arco del diagramma.

Si veda il lucido 3.1.2.

d) Perché il context switch tra peer thread è più veloce del context switch tra processi?

I peer thread condividono lo spazio di indirizzamento, che quindi non deve essere cambiato (ad esempio si continua ad usare la stessa page table) al context switch.

ESERCIZIO 2 (5 punti)

In un sistema operativo un indirizzo fisico è scritto su 27 bit, e un frame e grande 2048 byte. E' noto che se la tabella delle pagine più grande del sistema fosse anche solo di un byte più grossa, si potrebbe essere costretti ad adottare una paginazione a due livelli.

a) Qual è lo spazio di indirizzamento logico del sistema (esplicitate i calcoli che fate)?

Il numero di un frame del sistema è scritto su 27-11 =16 bit, ossia 2 byte, dunque in un singolo frame possiamo scrivere al massimo 1024 entry da due byte, e lo spazio di indirizzamento logico è pari a $2^{10} * 2^{11} = 2^{21}$ byte.

b) *Se il sistema qui sopra adottasse una Inverted Page Table, quanto sarebbe grande questa tabella? (nel rispondere a questa domanda esplicitate eventuali ipotesi che fate, e supponete che nel sistema possano essere presenti contemporaneamente al massimo 1024 processi)*

Per rispondere alla domanda è necessario ipotizzare il numero di bit usati per scrivere il PID di un processo. La IPT ha un numero di entry pari al numero di frame del sistema, e ogni enty contiene il numero di una pagina (10 bit) e il numero di un PID. Per poter numerare un massimo di 1024 processi abbiamo bisogno di almeno 10 bit. Se assumiamo di usarne 14, ogni entry della IPT occuperà 24 bit, ossia 3 byte. Ne segue che la ITP sarebbe grande $2^{16} * 3$ byte = 192 kilobyte.

c) Se l'hardware su cui gira il sistema non mette a disposizione né il bit di riferimento ne il dirty bit, in caso di necessità quale algoritmo di rimpiazzamento delle pagine dovrebbe essere implementato, e perché?

Tra le opzioni viste a lezione, l'unica ragionevole è FIFO, poiché l'implementazione di LRU è ritenuta computazionalmente troppo costosa.

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) Avete a disposizione un solo intervento (hardware o software) per diminuire la probabilità di thrashing in un sistema. Quale intervento applichereste?

Aumentare la quantità di memoria RAM del sistema.

ESERCIZIO 3 (4 punti)

a) Che cosa è nel sistema Unix un link fisico?

All'interno di una entry di una directory, la stringa di caratteri scritta a fianco del numero di un i-node.

b) Ad un i-node possono essere associati più hard link. È possibile che un hard link possa essere associato a i-node diversi (motivate la vostra risposta)?

Si, poiché si possono chiamare file diversi con lo stesso nome, purché si trovino in cartelle diverse.

c) Sia A un link fisico al file X, e sia B un link simbolico al file X. È più veloce l'accesso ai dati di X usando A o usando B? Perché? Quale dei due link produce maggior occupazione di spazio sull'hard disk, e perché?

L'accesso è più veloce tramite A, perché è associato direttamente all'index node del file X. Invece passando attraverso B occorre prima aprire l'index node associato a B, che contiene il pathname ad uno degli hard link di X.

Occupa più spazio B, perché è associato ad un index node che contiene il pathname al file X. A invece è semplicemente una entry in una cartella associata direttamente all'index node del file X.

d) Scegliete una dimensione per i blocchi di un hard disk e per il numero di bit usati per scrivere in numero di un blocco, e riportate la formula aritmetica che indica la dimensione massima di un file in un file system Unix con le caratteristiche che avete scelto.

Blocchi da 1024 byte, 32 bit per scrivere il numero di un blocco:

$10*1024 + 256 * 1024 + 256^2 * 1024 + 256^3 * 1024$ byte

ESERCIZI RELATIVI ALLA PARTE DI UNIX

ESERCIZIO 1 (2 punti)

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

Soluzione

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)

ESERCIZIO 2 (2 punti)

Descrivere l'esecuzione della *semop*, con riferimento ai possibili esiti della chiamata. In particolare illustrare come varia il comportamento della system call nell'eseguire chiamate bloccanti e non bloccanti. Considerare tutti gli eventi che possono determinare la fine del blocco.

Soluzione

La *semop* può andare a buon fine o meno a seconda del valore del semaforo e del tipo di operazione (incremento/decremento) identificata dal membro *sem_op* della struttura di tipo *sembuf*. Nel caso la *semop* esegua una chiamata bloccante, il processo chiamante resterà bloccato fino al verificarsi di uno dei seguenti eventi:

- un altro processo incrementa il valore del semaforo abbastanza da rendere possibile l'esecuzione della *semop*;
- la *semop* è interrotta da un segnale: la chiamata fallisce, e *errno* viene impostato a *EINTR*;
- un altro processo cancella il semaforo; in in questo caso la chiamata fallisce con errore *EIDRM*.

È possibile effettuare chiamate non bloccanti specificando il flag *IPC_NOWAIT* nel membro *sem_flg* della struttura di tipo *sembuf*. In caso *IPC_NOWAIT* sia specificato e non sia possibile eseguire il decremento del semaforo, *semop* fallisce con l'errore *EAGAIN*.

ESERCIZIO 3 (2 punti)

Si scriva un frammento di codice finalizzato ad evitare che un processo P possa creare figli che si trasformano in zombie. Suggerimento: si alteri la disposizione di SIGCHLD.

Soluzione

```
...

void handle_sigchld(int sig) {
 pid_t ret;
```

```
 int status;
 while ((ret=waitpid((pid_t)(-1), &status, WNOHANG)) > 0) {}
}

...

int main() {
 pid_t childPid;
 struct sigaction sa;
 sa.sa_handler = &handle_sigchld;
 sigemptyset(&sa.sa_mask);
 sa.sa_flags = SA_RESTART | SA_NOCLDSTOP;
 if (sigaction(SIGCHLD, &sa, 0) == -1) {
   perror(0);
   exit(1);
 }
 childPid = fork();

 ...
}
```

ESERCIZI RELATIVI ALLA PARTE DI C

ESERCIZIO 1 (3 punti)

Si implementi la funzione con prototipo

```
int find_and_replace(char * str, int pos, char find_char, char replace_char);
```

`find_and_replace()` è una funzione che sostituisce nella stringa di caratteri `str` ogni occorrenza del carattere `find_char` con il carattere `replace_char` ma solo nelle posizioni multiple intere di `pos`. La funzione restituisce il numero di sostituzioni effettuate.

Esempio;

- `str = dddghrddq`
- `pos = 2`
- `find_char = d`
- `find_char = z`

allora la funzione restituirebbe 3 e la stringa sarebbe `zdzghrzdq`

```
#include <stdio.h>
#include <string.h>
int find_and_replace(char * str, int pos, char find_char, char replace_char){
  int n_replaced = 0;
  if(str != NULL && pos > 0) {
    int i;
    int length = strlen(str);
    for( i = 0; i < length; i=i+pos)
    if(*(str+i)==find_char){
      *(str+i)=replace_char;
      n_replaced++;
    }
  }
  return n_replaced;
}

int main() {
  char str[20]="dddghrddq";
  int pos=2;
  char find_char = 'd';
  char replace_char = 'z';
  int n_replaces;

  printf("Old string %s pos %d find_char %c replace_char
%c\n",str,pos,find_char,replace_char);
  n_replaces = find_and_replace(str,pos,find_char,replace_char);
  printf("New string %s n_replaces %d\n",str,n_replaces);

  pos=3;
  find_char = 'g';
```

```
  replace_char = 'w';

  printf("Old string %s pos %d find_char %c replace_char
%c\n",str,pos,find_char,replace_char);
  n_replaces = find_and_replace(str,pos,find_char,replace_char);
  printf("New string %s n_replaces %d\n",str,n_replaces);

  pos=1;
  find_char = 'z';
  replace_char = 'y';

  printf("Old string %s pos %d find_char %c replace_char
%c\n",str,pos,find_char,replace_char);
  n_replaces = find_and_replace(str,pos,find_char,replace_char);
  printf("New string %s n_replaces %d\n",str,n_replaces);

  return 0;
}
```

ESERCIZIO 2 (1 punti)

Qual è l'output del seguente programma C?

Quale sarebbe l'output se nel programma l'istruzione `int x = 10;` nel main fosse commentata?

```
#include <stdio.h>
int x=90;
int g(int x){
  printf("%d\n",-x);
  if(x > 20) {
    int x = -90;
    printf("%d\n",-x);
  }
  else {
    int x = -60;
    printf("%d\n",-x);
  }
  return x;
}
void f(int x){
  printf("%d\n",-x);
  if(x > 20) {
    int x = -50;
    printf("%d\n",g(x));
  }
  else {
    int x = -70;
    printf("%d\n",g(-x));
  }
}
int main() {
  int x = 10; // Da commentare
  printf("%d\n",-x);
  f(-x);
  return 0;
}
```

Risposta primo caso: -10 10 -70 90 70

Risposta secondo caso: -90 90 -70 90 70

ESERCIZIO 3 (3 punti)

Data la struttura `node` definita come segue:

```
typedef struct node {

     int value;

     struct node * next;

} nodo;

typedef nodo* link;
```

implementare la funzione con prototipo

```
int check_first_and_last(link head, int *first_is_zero, int *last_is_zero);
```

`check_first_and_last()` è una funzione che restituisce il numero di elementi della lista `head` che sono diversi da `0`. Inoltre, nelle variabili passate per riferimento (`first_is_zero` e `last_is_zero`) restituisce TRUE se il primo (l'ultimo) elemento della lista è uguale a 0 e FALSE altrimenti. Definite opportunamente TRUE e FALSE e gestite il caso di lista vuota.

Ad esempio, data la lista `head`: 1 → 3 → 0 → `NULL` la funzione `check_first_and_last` ritorna 2. L'argomenti passato per riferimento `first_is_zero` è uguale a FALSE mentre `last_is_zero` è uguale a TRUE.

```
#include <stdio.h>
#include <stdlib.h>

#define TRUE 1
#define FALSE 0

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

int check_first_and_last(link head, int *first_is_zero, int *last_is_zero){
  int noteq = 0;
  *first_is_zero = FALSE;
  *last_is_zero = FALSE;

  if(head != NULL) {
    if(head->value == 0)
      *first_is_zero = TRUE;
    while (head != NULL) {
      if(head->value!=0)
        noteq++;
      if(head->next == NULL && head->value == 0)
        *last_is_zero = TRUE;
      head = head->next;
    }
```

```
  }
  return noteq;
}

int main() {

  link prova = (link)malloc(sizeof(nodo));
  prova->value=0; prova->next=NULL;
  link prova2 = (link)malloc(sizeof(nodo));
  prova2->value=3; prova2->next=prova;
  link prova3 = (link)malloc(sizeof(nodo));
  prova3->value=1; prova3->next=prova2;

  int first_is_zero,last_is_zero,noteq;

  noteq= check_first_and_last(prova3,&first_is_zero,&last_is_zero);
  printf("diversi     da     0     %d     first_is_zero     %d     last_is_zero
%d\n",noteq,first_is_zero,last_is_zero);

  return 0;
}
```