SISTEMI OPERATIVI – 14 settembre 2015
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) date una breve descrizione dei requisiti che deve soddisfare una qualsiasi soluzione corretta al problema della sezione critica?

Mutua esclusione: solo un processo alla volta può entrare nella sezione critica
Progresso: esiste un processo che riesce ad entrare in sezione critica in un tempo finito
Attesa limitata: qualsiasi processo che ne faccia richiesta entra in sezione critica in un tempo finito.

d) Nel codice di WAIT e SIGNAL un solo processo per volta può aggiornare la variabile semaforica e la coda di wait del semaforo. Come può essere ottenuta questa condizione?

WAIT e SIGNAL sono a loro volta sezioni critiche, che si possono implementare con meccanismi di busy waiting o con la disabilitazione degli interrupt.

ESERCIZIO 2 (5 punti)

In un sistema operativo un indirizzo fisico è scritto su 26 bit, e un frame e grande 1024 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 16 bit, ossia 2 byte, dunque in un singolo frame possiamo scrivere al massimo 512 entry da due byte, e lo spazio di indirizzamento logico è pari a $2^9 * 2^{10} = 2^{19}$ byte.

b) Il sistema deve implementare un meccanismo di rimpiazzamento delle pagine? (motivate la risposta)

Poiché lo spazio logico ha dimensione inferiore a quello fisico, la memoria virtuale è necessaria solo se si vogliono far girare contemporaneamente un insieme di processi che, insieme, occupano uno spazio superiore allo spazio fisico disponibile.

c) *Se il sistema adottasse invece una Inverted Page Table, quanto sarebbe grande questa tabella? (nel rispondere a questa domanda esplicitate eventuali ipotesi che fate)*

> Per rispondere alla domanda è necessario ipotizzare il numero di bit usati per scrivere il PID di un processo. Assumiamo ad esempio di usare 7 bit. La IPT ha un numero di entry pari al numero di frame del sistema, e ogni enty contiene il numero di una pagina (9 bit) e il numero di un PID (7 bit). Ne segue che la ITP è grande 2^{16} * 2 byte = 128 kilobyte.

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) E' vera la seguente affermazione (motivate la vostra risposta)? *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) Il 99,9% dei file del file system di un sistema operativo occupano ciascuno meno di un blocco. Quale/quali tra i metodi di allocazione dello spazio su disco visti a lezione converrà adottare, e perché?

Le allocazioni concatenata, contigua, ed NTFS sono tutte adeguate, in quanto assicurano un basso spreco di spazio e una elevata velocità di accesso ai dati dei file.

b) Quali svantaggi comporta l'implementazione di un file system con la tecnica della FAT?

Per garantire sufficiente efficienza, la FAT deve essere sempre tenuta in RAM, sottraendo spazio ai processi (deve anche essere periodicamente salvata su disco, occupando quindi risorse del SO).

c) dove è memorizzato il pathname di un file?

Da nessuna parte.

d) E' vera la seguente affermazione? (motivate la vostra risposta): "*l'accesso ai dati di un file memorizzato su un sistema raid è sempre più veloce che se il file è memorizzato su un normale hard disk.*"

L'affermazione è falsa, in quanto tutti i blocchi del file potrebbero comunque trovarsi memorizzati su uno solo degli hard disk del RAID.

ESERCIZI RELATIVI ALLA PARTE DI UNIX (6 punti)

ESERCIZIO 1

(2 punti)

Considerata la system call

int kill(pid_t pid, int sig);

i) illustrare qual è lo scopo della system call, *ii)* cosa restituisce, e *iii)* spiegare quali valori può assumere il primo argomento, *pid*.

Soluzione

---

i) La system call ha lo scopo di inviare segnali ad un processo o ad un gruppo di processi. In particolare, il segnale inviato è quello indicato dal secondo argomento sig.
ii) Restituisce 0 in caso di successo, -1 altrimenti.
iii) Il comportamento della system call è condizionato dal valore del primo argomento, che può essere:
- pid>0: il segnale sig viene mandato al processo con quel pid
- pid =0: il segnale sig viene mandato a tutti i processi dello stesso gruppo del processo che esegue la kill
- pid =-1: il segnale viene mandato a tutti i processi in esecuzione per i quali il processo chiamante ha il permesso di inviare segnali
(-pid < -1: il segnale è inviato a tutti i processi del gruppo del chiamante il cui ID è -pid)

ESERCIZIO 2

(2 punti)

Commentare la system call shmget: spiegare argomenti e valore di ritorno.

Soluzione [slides 09_semafori.pdf]

Prototipo della system call:

```
int shmget(key_t key, size_t size, int shmflg);
```

La system call shmget è utilizzata per allocare un nuovo segmento di memoria condivisa o per determinare l'ID di un segmento di memoria condivisa già esistente.
Valore di ritorno: l'identificatore di un segmento di memoria condivisa
Argomenti:
- key: chiave del segmento di memoria condivisa: se non esiste una memoria condivisa con chiave key, la shmget allocherà un nuovo segmento e restituirà un nuovo identificatore univoco per il segmento; altrimenti, esiste già un segmento con chiave key e la shmget non alloca altra memoria ma restituisce l'id del segmento già esistente
La chiave può assumere la costante IPC_PRIVATE per forzare la generazione di una nuova chiave univoca, e di conseguenza la creazione di un nuovo segmento di memoria condivisa.
-size: dimensione in byte del segmento da allocare. (Il kernel arrotonda la dimensione indicata alla dimensione al multiplo più vicino di PGE_SIZE per semplicità di gestione.)
-shmflg: flag che regolano il comportamento della system call, possono essere:
- *IPC_CREAT|IPC_EXCL per la creazione in esclusiva del segmento*
- *Diritti di accesso al segmento, abbreviati da una tripletta di valori in ottale*

ESERCIZIO 3

(2 punti)

Dire se il seguente stralcio di codice genera esattamente tre processi figli, in caso negativo come si può correggere? Il codice genera processi zombie, in caso affermativo, spiegare perché e come si può correggere?

```
main{
  int i;
 for(i=0; i<3; i++){
            int n = fork();
            if(n == 0){
                        printf("sono un figlio");
            }
            else{
                         printf("sono un padre");
            }
 }//for
}//main
```

Soluzione

i) il codice nella forma orginale genera più tre di figli poiché ogni figlio generato dal processo padre diventa padre a sua volta. Infatti, I processi figli condividono con il processo padre, e a meno di espliciti vincoli, il processo figlio eseguirà anche il codice del padre, e quindi nel caso dell'esercizio, eseguirà anche il ciclo for, generando a sua volta altri processi figli. Per correggere quest'errore sarebbe sufficiente far terminare il processo figlio subito dopo il suo segmento di codice.
ii) il codice ha la possibilità di generare zombie perché è possibile che i processi figli terminino prima del processo padre, il quale però non esegue mai una wait, di conseguenza le entry dei processi figli all'interno della tabella dei processi non vengono mai rimosse.
Per risolvere quest'errore è sufficiente utilizzare la system call wait.
Codice corretto:

```
main{
  int i;
 for(i=0; i<3; i++){
            int n = fork();
            if(n == 0){
                        printf("sono un figlio");
                        exit(0); //correzione per il punto i
            }
            else{
                         printf("sono un padre");
            }
 }//for
  while (wait(NULL)!=-1); //correzione per il punto ii
}//main
```

ESERCIZI RELATIVI ALLA PARTE DI C

ESERCIZIO 1 (3 punti)

Si implementi la funzione con prototipo

```
int check_in_even_positions(char * str, char check_char);
```

`check_in_even_positions()` è una funzione che restituisce TRUE se gli elementi di `str` in

posizioni pari sono uguali a `check_char` e FALSE altrimenti. Definite opportunamente TRUE e FALSE e gestite il caso in cui `str` sia NULL.

Esempi:

se la stringa fosse: dfdgdrdgd e `check_char` g allora la funzione restituirebbe FALSE
se la stringa fosse: dfdgdrdgd e `check_char` d allora la funzione restituirebbe TRUE
se la stringa fosse: dfsgdrdgd e `check_char` d allora la funzione restituirebbe FALSE

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

#define TRUE 1
#define FALSE 0

int check_in_even_positions(char * str, char check_char){
        int ret_value = FALSE;

        if(str != NULL) {
                int pos;
                int length = strlen(str);
                ret_value = TRUE;
                for( pos = 0; pos < length && ret_value==TRUE; pos=pos+2)
                        if(*(str+pos)!=check_char)
                                ret_value = FALSE;
        }
        return ret_value;
}

main() {
char str[20]="dfdgdrdgd";

char check_char = 'g';
printf("String %s check_char %c results %d\n",str,check_char,check_in_even_positions(
str,check_char));

check_char = 'd';
printf("String %s check_char %c results %d\n",str,check_char,check_in_even_positions(
str,check_char));

check_char = 'd';
strcpy(str,"dfsgdrdgd");
printf("String %s check_char %c results %d\n",str,check_char,check_in_even_positions(
str,check_char));
}
```

ESERCIZIO 2 (1 punti)

Qual è l'output del seguente programma C?
Quale sarebbe l'output se nel programma l'istruzione `int x = 60;` fosse commentata?

```
#include <stdio.h>
int x=20;

int g(int x){
        printf("%d\n",x);
        if(x > 20) {
                int x = 20;
                printf("%d\n",x);
                return x;
        }
        else {
```

```
                int x = -20;
                printf("%d\n",-x);
                return x;
        }
}

int f(int x){
        printf("%d\n",x);
        if(x > 20) {
                int x = 20;
                printf("%d\n",g(x));
        }
        else {
                int x = -20;
                printf("%d\n",g(-x));
        }
}
main() {
        int x = 60; // Da commentare

        printf("%d\n",x);
        f(x);
}
```

Risposta primo caso: 60 60 20 20 -20
Risposta secondo caso: 20 20 20 20 -20

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 count(link head, int value, int *gt, int *lt);
```

`count()` è una funzione che restituisce il numero di elementi della lista `head` che sono uguali a `value`. Inoltre, nelle variabili passate per riferimento (`gt` ed `lt`) restituisce il numero di elementi maggiori e minori di `value`, rispettivamente.

```
#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 count(link head, int value, int *gt, int *lt){
        int neq = 0;
        *lt = 0;
        *gt = 0;

        while (head != NULL) {
                if(head->value==value)
                        neq++;
                else if(head->value < value)
                        (*lt)++;
                else
                        (*gt)++;
                head = head->next;
        }
        return neq;
}

main() {

link prova = (link)malloc(sizeof(nodo));
prova->value=1; prova->next=NULL;

link prova2 = (link)malloc(sizeof(nodo));
prova2->value=3; prova2->next=prova;

link prova3 = (link)malloc(sizeof(nodo));
prova3->value=5; prova3->next=prova2;

int value = 10;
int lt,gt,eq;

eq = count(prova3,value,&gt,&lt);
printf("value %d : eq %d lt %d gt %d\n",value,eq,lt,gt);
}
```