SISTEMI OPERATIVI
15 febbraio 2016
corso A e B 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) (2 p.) 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 e i valori di inizializzazione mancanti.

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) (1 p.) Si consideri l'algoritmo del punto precedente, e si assuma che tutti i lettori e tutti gli scrittori accedano al file condiviso per una quantità di tempo finita. Spiegate se è possibile, e se si per quale ragione, che un lettore vada in starvation. Spiegate se è possibile, e se si per quale ragione, che uno scrittore vada in starvation.

Un processo lettore non può mai andare in starvation. Un processo scrittore può andare in starvation se si è addormentato sul semaforo scrivi a *causa della presenza di uno o più lettori in fase di lettura* e continuano ad arrivare nuovi lettori, di modo che la variabile numlettori non scende mai a zero.

c) (1 p.) riportate lo pseudocodice relativo alla corretta implementazione dell'operazione di *wait*.

Si vedano i lucidi della sezione 6.5.2.

d) (1 p.) Che significato pratico ha sull'uso del semaforo, il valore corrente della sua variabile semaforica S?

Se S > 0, indica quanti processi possono superare il semaforo senza addormentarsi nella sua coda di wait.
Se S < 0, |S| indica quanti processi sono addormentati nella coda di wait del semaforo.

ESERCIZIO 2 (5 punti)

In un sistema che adotta la memoria paginata le pagine hanno una dimensione tale da produrre una frammentazione interna media per processo di 2 Kbyte. Il sistema adotta una paginazione a due livelli, e la tabella delle pagine esterna più grande del sistema occupa esattamente un frame. Ogni entry di una tabella delle pagine (interna o esterna) occupa 4 byte, e tutti i bit vengono usati per scrivere il numero di un frame.

a) (2 p.) Quali sono le dimensioni degli spazi di indirizzamento logico e fisico del sistema?

Il sistema adotta pagine da 4 Kbyte, ossia 2^{12} byte. La tabella delle pagine esterna più grossa del sistema contiene $2^{12}/2^2$ entry, e quindi la tabella delle pagine interna più grande del sistema ha in tutto $2^{10}*2^{10} = 2^{20}$ entry. Lo spazio logico è quindi grande $2^{20} * 2^{12} = 2^{32}$ byte e quello fisico è grande $2^{32} * 2^{12} = 2^{44}$ byte.

b) (1 p.) Perché un sistema con le caratteristiche del punto a) potrebbe voler implementare la memoria virtuale?

Per poter far girare contemporaneamente un insieme di processi che occupano complessivamente uno spazio logico maggiore di quello fisico.

c) (1 p.) È possibile che un processo P che gira su un sistema A dotato di memoria virtuale venga eseguito più velocemente che su un sistema B del tutto identico ad A ma non dotato di memoria virtuale? (motivate la vostra risposta)

Si, ad esempio se in una particolare esecuzione di P su sul sistema A, P non genera mai page fault e non usa una porzione di codice che quindi non deve essere caricata in memoria primaria, risparmiando così sul tempo di caricamento rispetto al sistema B (che invece caricherà sempre in RAM l'intera immagine di P)

d) (1 p.) Si considerino i seguenti interventi su un sistema dotato di memoria virtuale:

1. salvare periodicamente sull'hard disk le pagine col dirty bit a 1.
2. usare un dispositivo di memoria secondaria più veloce.
3. aumentare la quantità di memoria primaria.
4. passare dall'algoritmo di rimpiazzamento FIFO a quello della seconda chance migliorata.
5. usare una CPU più veloce.

Quale, tra questi interventi, ha l'effetto migliore sul fenomeno del thrashing del sistema, e perché?

L'intervento 3, poiché è permette di aumentare il numero medio di pagine che un processo riesce a tenere in RAM, diminuendo così la probabilità di page fault.

Quale, tra questi interventi, non ha alcun effetto sul fenomeno del thrashing del sistema, e perché?

L'intervento 5, poiché non ha alcun effetto sulla frequenza con cui si verifica il page fault, e ha un effetto quasi nullo sul tempo di gestione del page fault.

NOTA (che non era necessario includere nella risposta): l'intervento 1 aumenta solo la probabilità che, in caso di page fault, la pagina vittima non debba essere salvata prima di essere sovrascritta; l'intervento 2 diminuisce solo il tempo di gestione del page fault, l'intervento 4 può, in alcuni casi, ridurre leggermente la frequenza del page fault. In generale l'intervento più significativo per il controllo e la prevenzione del thrashing è sempre quello di limitare opportunamente il grado di multiprogrammazione del sistema (si veda ad esempio la sezione 9.6.3)

ESERCIZIO 3 (4 punti)

Un hard disk ha la capienza di 2^{34} byte, ed è formattato in blocchi da 2048 byte.

a) (1 p.) Quanti accessi al disco sono necessari per leggere l'ultimo blocco di un file A della dimensione di 2^{13} 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 2045 byte di dati più 3 byte di puntatore al blocco successivo (infatti, $2^{34}/2^{11} = 2^{23}$), per cui sono necessari 5 blocchi per memorizzare l'intero file.

b) (1 p.) 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^{23} x 3 byte = 24 megabyte

c) (1 p.) Quali sono gli svantaggi nell'uso della FAT?

Per garantire un accesso efficiente ai file deve essere sempre tenuta in RAM occupando spazio, se viene persa si perdono tutte le informazioni sul file system, e deve quindi essere periodicamente salvata su disco.

d) (1 p.) Disegnate la FAT di un hard disk formato da 2^4 blocchi e contenente un unico file memorizzato, nell'ordine, nei blocchi 8, 4, 12, 2. 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)

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

Soluzione

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*.

ESERCIZIO 2 (2 punti)

Illustrare le operazioni IPC_RMID, IPC_STAT, IPC_SET

Soluzione

IPC_RMID: rimuove immediatamente il set di semafori e l'associata struttura semid_ds. Qualsiasi processo bloccato in chiamate semop() in attesa su semafori è immediatamente svegliato, e semop() riporta l'errore EIDRM;

- IPC_STAT: copia la struttura semid_ds associata con il set di semafori nel buffer puntato da arg.buf;

- IPC_SET: Aggiorna i membri della struttura semid_ds associata al set di semafori utilizzando i valori nel buffer puntato da arg.buf.

ESERCIZIO 3 (2 punti)

Supponendo che il file prova contenga i seguenti nomi

```
Mario
 Paolo
 Ada
 Mario
 Franca
 Aldo
```

illustrare il significato e il risultato del seguente comando

```
sort < prova > file2
```

Soluzione

Il comando combina la redirezione dell'input e dell'output: in primo luogo il contenuto del file *prova* è dato in input a *sort*, che opera l'ordinamento; successivamente il contenuto del file (ordinato al passo precedente) è scritto nel nuovo file *file2*. Se *file2* esisteva viene sovrascritto; diversamente viene creato. Il contenuto di *file2* sarà:

```
 Ada
 Aldo
 Franca
 Mario
 Mario
 Paolo
```

ESERCIZI RELATIVI ALLA PARTE DI C

ESERCIZIO 1 (3 punti)

Si implementi la funzione con prototipo

```
int contains(char * str1, char * str2);
```

`contains()` restituisce TRUE se tutti i caratteri di `str2`, indipendentemente dal loro ordine, sono anche caratteri di `str1`; FALSE altrimenti. Definite opportunamente TRUE e FALSE, gestite il caso in cui le due stringhe siano NULL (per semplicità, se almeno una delle due stringhe è NULL allora la funzione restituisce FALSE).

Esempi:

- `str1: dfdgdrdgd  str2: dfs` output: FALSE
- `str1: dfdgdrdgd  str2: dfg` output: TRUE
- `str1: dfdgdrdgd  str2: dfgd` output: TRUE

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

#define TRUE 1
#define FALSE 0

int contains(char * str1, char * str2) {
    int is_contained = TRUE;

    int j=0;
    if (str2==NULL) return TRUE;
    if (str1==NULL) return FALSE;

    while (str2[j] != '\0' && is_contained == TRUE){
        int found = FALSE;
        int i = 0;
        while (str1[i] != '\0' && found == FALSE){
            if (str1[i] == str2[j])
                found = TRUE;
            i++;
        }//while

        if (found == TRUE)
            j++;
        else
            is_contained = FALSE;
    }//while

    return is_contained;
}



int main() {
    char str1[20]="dfdgdrdgd";
    char str2[20]="dfs";
```

```
    printf("String container %s, string contained %s, results %d\n",str1,
str2, contains(str1, str2));

    strcpy(str2, "dfg");

    printf("String container %s, string contained %s, results %d\n",str1,
str2, contains(str1, str2));

    strcpy(str2, "dfgd");

    printf("String container %s, string contained %s, results %d\n",str1,
str2, contains(str1, str2));

    return 0;
}
```

ESERCIZIO 2 (1 punto)

Il seguente programma deve calcolare la divisione tra due numeri reali solo se il denominatore è diverso da zero. Correggere tutti gli eventuali errori presenti.

```
#include <stdio.h>

#define ERROR -1
#define OK 1

int divisione_sicura(double numeratore, double denominatore,
             double * risultato){

    if (denominatore =[=] 0) return ERROR;
    [*]risultato = numeratore / denominatore;
    return [OK]{risultato};
}

int main(){
    printf("inserisci numeratore: ");
    double num;
    scanf("%lf", &num);

    printf("inserisci denominatore: ");
    double den;
    scanf("%lf", &den);
    double ris;
    if ( divisione_sicura(num, den, [&]ris) == OK)
        printf("il risultato e': %lf\n", ris);
    else printf("divisione non eseguibile\n");
    return 0;
}
```

correzioni: tra [] le porzioni da aggiungere, tra {} le porzioni da rimuovere.

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 change(link head, int old, int new);
```

La funzione sostituisce tutte le occorrenze del valore `old` nella lista `head` con il nuovo valore `new`; inoltre restituisce il numero di cambiamenti effettuati.
La funzione deve gestire il caso in cui la lista passata sia vuota.
Esempio: data la lista `head`: 5→ 3 → 4 → 5 → NULL l'invocazione della funzione `change(head, 5, 3);` restituirà il valore 2 e modificherà la lista come `head`: 3→ 3 → 4 → 3 → NULL.

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

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


int change(link head, int old, int new){
    int count = 0;
    if (head == NULL) return 0;
    while (head){
        if (head->value == old) {
            head->value = new;
            count++;
        }
        head =  head->next;
    }
    return count;
}

int main() {
    link prova = (link)malloc(sizeof(nodo));
    prova->value=5; prova->next=NULL;

    link prova2 = (link)malloc(sizeof(nodo));
    prova2->value=4; prova2->next=prova;

    link prova3 = (link)malloc(sizeof(nodo));
    prova3->value=3; prova3->next=prova2;

    link prova4 = (link)malloc(sizeof(nodo));
    prova4->value=5; prova4->next=prova3;
```

```
    printf("change 5 into 3; num changes %d \n", change(prova4, 5, 3));

    link iteratore = prova4;
    while (iteratore != NULL) {
        printf("%d -> ", iteratore->value);
        iteratore = iteratore->next;
    }
    printf("NULL\n");

    return 0;
}
```