SISTEMI OPERATIVI
3 febbraio 2014
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) Esiste un caso in cui uno o più processi lettore possano andare in starvation?

Si, se il processo che sta scrivendo non termina mai

c) Sia dato un sistema operativo time sharing che implementa la memoria virtuale. Riportate il diagramma di accodamento che mostra come i processi si muovono fra le varie code di scheduling di questo sistema.

Si vedano la figura del lucido 18 del capitolo 3 sulla gestione dei processi e il lucido 9 del capitolo 9 sulla memoria virtuale.

d) Riportate lo pseudocodice che descrive l'implementazione dell'operazione di Signal, spiegate perché i semafori sono preferibili a meccanismi di sincronizzazione basati sul busy waiting (ad esempio per risolvere il problema della sezione critica)

Per lo pseudocodice si vedano i lucidi della sezione 6.5.2

Perché il pcb del processo che si addormenta sul semaforo in attesa di entrare in sezione critica viene messo nella coda di waiting del semaforo e non può più essere selezionato dallo scheduler fino a che la sezione critica non si libera.

ESERCIZIO 2 (5 punti)

Si consideri un sistema in cui in una tabella delle pagine di un processo l'indice più grande usabile nella tabella delle pagine di quel processo può essere 1FFF. Un indirizzo fisico del sistema è scritto su 24 bit, e la RAM è suddivisa in 4000 (esadecimale) frame.

a)
Quanto è grande, in megabyte, lo spazio di indirizzamento logico del sistema (esplicitate i calcoli che fate)?

4000(esadecimale) = 2^{14}, per cui un numero di frame è scritto su 14 bit, e la dimensione di un frame, e quindi di una pagina, è di $2^{\wedge 10}$ byte (24 - 14 = 10). Poiché il numero più grande di una pagina è 1FFF, ci possono essere al massimo $2^{\wedge 13}$ pagine, e lo spazio di indirizzamento logico è di $2^{\wedge 13}$ x $2^{\wedge 10}$ byte (pari a circa 8 megabyte).

b)
Se il sistema adotta una paginazione a due livelli, quanto spazio occupa, in byte, la tabella delle pagine *esterna* più grande di questo sistema? (motivate numericamente la vostra risposta).

La tabella delle pagine interna più grande del sistema ha $2^{\wedge 13}$ pagine, e ogni entry contiene il numero di un frame, per cui sono necessari almeno due byte per ogni entry. La dimensione di quella tabella sarà quindi di $2^{\wedge 13}$ x 2 = $2^{\wedge 14}$ byte (ossia maggiore della dimensione di un frame, il che implica appunto la nececessità

di paginare la tabella delle pagine interna). $2^{\wedge 14}$ byte = 16 Kbyte, e quindi questa tabella delle pagine interna occupa 16 frame. Di conseguenza la tabella esterna sarà composta da 16 entry di due byte ciascuno, ossia 32 byte.

c)
In quale caso il sistema descritto qui sopra deve implementare la memoria virtuale?

> Lo spazio di indirizzamento logico è minore di quello fisico, per cui l'unico caso è quello in cui si vogliano avere in esecuzione contemporaneamente processi che, insieme, occupano uno spazio maggiore dello spazio di indirizzamento fisico.

d)
Un sistema (hardware + sistema operativo) soffre spesso del problema del thrashing. Indicate due modifiche al sistema, una hardware e una software, che potrebbero migliorare la situazione. Motivate le indicazioni che date.

Modifica Hardware: aggiungere più ram, cosicché ogni processo del sistema avrà mediamente più frame a disposizione e meno probabilmente generà un page fault.
Modifica software: diminuire il grado di multiprogrammazione, in modo che meno processi hanno a disposizione ciascuno più frame.

ESERCIZIO 3 (4 punti)

a) nel contesto del sistema operativo Unix, che cosa è un link counter, e che cosa ci dice il suo valore?

È uno dei campi di un index-node e conta il numero di link fisici associati al file corrispondente a quell'index-node.

b) Indicate almeno un comando Unix e una system call unix che possono modificare il valore di un link counter. Cosa succede quando un link counter arriva al valore 0 (zero)?

comandi: ln, rm. System call: link, un link.
L'index node a cui quel link counter appartiene viene rimosso, e vengono marcati come liberi tutti i blocchi del disco puntati da quell'index node.

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.

ESERCIZI RELATIVI ALLA PARTE DI UNIX (6 punti)

ESERCIZIO 1 (2 punti)

(1.1) Data l'istruzione int i = 7, j = 15; indicare il risultato delle istruzioni seguenti:

(1 punti)

```
printf("%d\n", ((i & j || 1) >> 1) ^ 7 );   _____________
printf("%d\n", ((i || j | 7) >> 1) | 1 );   _____________
```

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

```
printf("%d\n", ((i & j || 1) >> 1) ^ 7 );   7
printf("%d\n", ((i || j | 7) >> 1) | 1 );   1
```

(1.2) Illustrare le caratteristiche principali delle classi di visibilità *extern* e *register*.

(1 punti)

Soluzione [slides 02_integrazione_linguaggio.pdf, slide 19 e sgg.]

Extern. Si tratta di variabili esterne a tutte le funzioni, ovvero accessibili da parte di qualsiasi funzione. Le variabili esterne possono sostituire le liste di argomenti per lo scambio di dati fra le funzioni; mantengono i loro valori anche dopo che le funzioni che le hanno modificate hanno smesso di operare. Si definiscono una sola volta al di fuori di qualunque funzione; la definizione porta il sistema a riservare una quantità appropriata di memoria per i contenuti della variabile; devono anche essere dichiarate in ogni funzione che intenda accedervi.

La classe di memorizzazione ***register*** che ha come obiettivo l'aumento della velocità di esecuzione: segnala al compilatore che la variabile corrispondente dovrebbe essere memorizzata in registri di memoria ad alta velocità. Nel caso il compilatore non possa allocare un registro fisico, viene utilizzata la classe auto (il compilatore dispone solo di una parte dei registri, che possono invece essere utilizzati dal sistema). Quando la velocità è importante, il programmatore può scegliere poche variabili alle quali viene fatto più frequentemente accesso (per esempio, variabili di ciclo e parametri delle funzioni). All'uscita dal blocco, il registro viene liberato; le variabili register sono di norma dichiarata nel punto più vicino possibile al punto in cui saranno utilizzate, per consentire la massima disponibilità di registri fisici, utilizzati solo quando necessario.

ESERCIZIO 2 (3 punti)

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 54 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*.

ESERCIZIO 3

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);
```

(2 punti)

Soluzione [slides 09_syscall_kill_signal.pdf, slide 27 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 e che prende un intero come argomento.

ESERCIZI RELATIVI ALLA PARTE DI C

ESERCIZIO 1 (2 punti)

Si implementi la funzione con prototipo

```
int strlen(char * str);
```

`strlen` è una funzione che calcola il numero di caratteri di cui è composta la stringa `str` se diversa da NULL e restituisce -1 altrimenti.

```
int strlen(char * str) {
        int length=-1;
        if(str != NULL) {
                length=0;
                while (*str != '\0') {
                        length++;
                        str++;
                }
        }
        return length;
}
```

ESERCIZIO 2 (3 punti)

Si implementi la funzione con prototipo

```
int odd_more_than_even(int * numbers, int length);
```

`odd_more_than_even` è una funzione che restituisce 1 se l'array di numeri interi `numbers` (la cui lunghezza è data dal parametro `length`) contiene più numeri dispari che pari, 0 se numeri pari e numeri dispari compaiono in quantità uguali e -1 altrimenti.

```
int odd_more_than_even(int * numbers, int length){


        int index,nodd=0,neven=0;
        for(index=0; index < length; index++){
                if(numbers[index]%2==1)
```

```
                        nodd++;
                else
                        neven++;
        }
        if(nodd > neven)
                return 1;
        else if(nodd==neven)
                return 0;
        else
                return -1;
}
```

ESERCIZIO 3 (2 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_element(link head, int value);
```

`count_element` è una funzione che restituisce il numero di occorrenze dell'elemento `value` nella lista `head`. Se la lista è vuota, la funzione restituisce, invece, -1.

Funzione da implementare

```
int count_element(link head, int value) {
     int retvalue = -1;
     if(head != NULL) {
```

```
        retvalue = 0;
        while (head != NULL) {
            if (head->value == value)
                retvalue++;
            head = head->next;
        }
    }
    return retvalue;
}
```