# SISTEMI OPERATIVI – 7 luglio 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*", e di "*Istituzioni di Sistemi Operativi*" 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)

In un sistema operativo che adotta uno scheduling con diritto di prelazione, quattro processi arrivano al tempo indicato e consumano la quantità di CPU indicata nella tabella sottostante)

| Processo | T. di arrivo | Burst |
|---|---|---|
| P1 | 0 | 14 |
| P2 | 2 | 13 |
| P3 | 5 | 3 |
| P4 | 9 | 2 |

a)
Qual è il waiting time medio migliore (ossia ottimale) che potrebbe essere ottenuto per lo scheduling dei quattro processi della tabella? RIPORTATE IL DIAGRAMMA DI GANTT USATO PER IL CALCOLO. (lasciate pure i risultati numerici sotto forma di frazione, e indicate quali assunzioni fate)

**Diagramma di GANT, assumendo come algoritmo di scheduling SJF preemptive:**

(0)….P1….(5)….P3….(8)….P1….(9)….P4…(11)….P1…(19)…P2….(32)

**Waiting time medio:**
P1 = (19 – 0) – 14 = 5;
P2 = (32 – 2) – 13 = 17;
P3 = (8 – 5) – 3 = 0;
P4 = (11 – 9) – 2 = 0;
waiting time medio = 22/4

b)
Riportate lo pseudocodice che descrive la corretta implementazione dell'operazione di WAIT, e spiegate perché il codice della WAIT deve essere implementato come una sezione critica.

*Si vedano i lucidi della sezione 6.5.2.*

c) Siano dati tre programmi concorrenti A, B e C. All'interno di A viene eseguita la procedura Pa, all'interno di B viene eseguita la procedura Pb e all'interno di C viene eseguita la procedura Pc. Si vuole essere certi che Pa, Pb e Pc vengano eseguite in quest'ordine, indipendentemente da come A, B e C vengono schedulati all'interno del sistema operativo. Fornite una possible soluzione al problema.

Semaforo AB = 0; semaforo BC = 0

| **A:** | **B:** | **C:** |
|---|---|---|
| Pa | wait (AB) | wait(BC) |
| signal (AB) | Pb | Pc |
| | Signal (BC) | |

**ESERCIZIO 2 (5 punti)**

Un sistema con memoria paginata usa un TLB con un hit-ratio del 80%, e un tempo di accesso di 20 nanosecondi. Un accesso in RAM richiede invece 0,08 microsecondi. Quando si verifica un miss, nel 10% dei casi la pagina indirizzata non è in memoria primaria, e quando la pagina indirizzata non è in memoria primaria, nel 50% dei casi ha il dirty bit a 1. Il costo di trasferimento di una pagina verso o dalla memoria di Swap è di 1 millisecondo.

a) Qual è, in nanosecondi, l'Effective Acces Time del sistema? (esplicitate i calcoli che fate, e limitatevi a scrivere l'espressione che permette di calcolare EAT)

ma = 0,80 * (80+20) + 0,18 * (2*80 + 20) + 0,01 * (1000.000) + 0,01 * (2*1000.000)

b) Se il sistema l'algoritmo di rimpiazzamento della seconda chance migliorato, quali informazioni dovrà contenere ogni entri della tabella delle pagine di un processo del sistema?

Numero del frame, bit di validità, dirty bit, bit di riferimento.

c) In che senso l'algoritmo della seconda chance è una approssimazione di LRU?

Il bit di riferimento permette di partizionare grossolanamente le pagine in "pagine usate di recente" e "pagine non usate di recente" approssimando appunto l'ordine con cui le singole pagine sono state usate.

d) E' possibile che un processore abbia una percentuale di utilizzo molto bassa, ma se lanciamo nuovi processi questi vengono eseguiti molto lentamente (motivate la vostra risposta)?

Si, se il sistema è in thrashing

e) in un nuovo sistema operativo si è visto sperimentalmente che il numero medio di page fault per unità di tempo PF è legato alla dimensione x delle pagine (espressa in byte) dalla formula:
$PF(x) = 2x^2 – 4096x + 3.000.000$. Quale sarebbe la dimensione ideale delle pagine del sistema per minimizzare il numero medio di page fault? (esplicitate i calcoli che fate)

$PF'(x) = 4x – 4096$, $x = 4096/4 = 1024$ byte.

**ESERCIZIO 3 (4 punti)**

Un hard disk ha la capacità di 64 gigabyte, è formattato in blocchi da 400 (esadecimale) byte, e usa una qualche forma di allocazione indicizzata per memorizzare i file su disco. Sull'hard disk è memorizzato un file A grande 400 Kbyte. Nel rispondere alle domande sottostanti, specificate sempre le assunzioni che fate.

a) Quante operazioni di I/O su disco sono necessarie per portare in RAM l'ultimo blocco del file A, assumendo che inizialmente sia presente in RAM solo la copia del file directory che "contiene" il file?

   L'hard disk contiene $2^{36}/2^{10} = 2^{26}$ blocchi, e sono quindi necessari 4 byte per scrivere in numero di un blocco. Un blocco indice può quindi contenere al massimo 1024/4 = 256 numeri di blocco. La risposta dipende poi dal tipo di allocazione indicizzata assunta:

   1) Allocazione indicizzata a schema concatenato: sono necessari 2 blocchi indice per tenere traccia di tutti i blocchi del file. Se assumiamo che sia già in RAM il numero del primo blocco indice, sono necessarie 3 operazioni di I/O: lettura dei due blocchi indice più lettura del blocco del file.
   2) Allocazione indicizzata a più livelli. È sufficiente usare uno schema a due livelli. Se assumiamo che sia già in RAM il numero del blocco indice esterno, sono necessarie 3 operazioni di I/O: lettura del blocco indice esterno, lettura di un blocco indice interno, lettura del blocco del file.
   3) Allocazione indicizzata Unix: è necessario usare il puntatore a di singola in direzione, quindi, assumendo già in RAM il numero dell'index-node, sono necessarie 4 operazioni di I/O: lettura dell'index-node, lettura del blocco indice puntato dal puntatore di singola in direzione, lettura del blocco indice interno, lettura del blocco del file.

b) Nel caso di accesso ai dati di file molto piccoli, è più efficiente l'implementazione scelta in Windows con NTFS o quella scelta da Unix con gli index-node? (motivate la vostra risposta)

   NTFS. Infatti per file molto piccoli, l'elemento che contiene gli attributi del file può contenere anche i dati del file stesso.

c) Quale/quali dei seguenti comandi modifica il valore del *link counter* dell'index-node associato al file di testo A? (si assuma che tutti i comandi vengono eseguiti correttamente)

   1) ls A B;  2) ln –s A B;  3) rm A B;  4) ln A B

   i comandi 3) e 4)

d) Da che cosa dipende il valore del link counter di una cartella Unix?

   Dal numero di sotto-directory che contiene.

SOLUZIONI 7/7/2015

# ESERCIZI RELATIVI ALLA PARTE DI UNIX (6 punti)

## ESERCIZIO 1 (2 punti)

Illustrare, con un esempio di invocazione, i comandi necessari a risolvere i seguenti problemi:
(1.1) Modificare i permessi di accesso ad un file in modo che l'utente abbia il controllo completo, i membri del suo gruppo possano consultare e modificare il file, e tutti gli altri utenti possano solo leggere il file.

Illustrare, quindi, come Unix organizza i permessi ai file.

**(1 punto)**

```
$ chmod 764 <nome_file>
```

Il comando Unix per la modifica dei permessi è *chmod*, per capire l'argomento 764 occorre ricordare che in Unix, l'accesso ai file è organizzato per mezzo tre categorie di utenti:
- user: è il possessore del file
- group: è il gruppo cui appartiene lo user
- other: sono tutti gli altri utenti del sistema che però non sono nello stesso gruppo dello user.

Per ciascuna di queste categorie è necessario stabilire i permessi che possono essere:
- read (r): possibilità di aprire in lettura il file
- write (w): possibilità di aprire in scrittura il file
- execute (x): possibilità di mandare in esecuzione un file eseguibile, o di attraversare una directory, nel caso in cui il file in questione fosse appunto una directory.

I permessi associati ai file sono quindi codificati per mezzo di una stringa di tre triplette di bit in cui ogni tripletta rappresenta, nell'ordine, i permessi rwx per user, group e other.
Nel caso specifico richiesto i permessi per il file sono codificati come segue:
- user: controllo completo (rwx) in bit 111 → 7
- group: lettura e scrittura (rw-) in bit 110 → 6
- other: solo lettura (r--) in bit 100 → 4

(1.2) Scrivere in modo *ordinato* all'interno di un file **`elenco.txt`**, tutti i file della directory corrente che abbiano estensione **`.c`** o **`.h`**

```
$ ls *.[ch] | sort > elenco.txt
```

Il comando list (**`ls`**) elenca tutti i file presenti nella directory passata come argomento, o in quella corrente se nessuna directory è specificata. Per selezionare i soli file con estensione .c o .h si usa la combinazione di wildchars **`*.[ch]`** dove: '*' rappresenta qualsiasi sequenza di caratteri; '.' indica che la sequenza di caratteri deve terminare con il punto; e '[ch]' indica che dopo il punto deve essere uno dei due caratteri indicati tra quadre.

## ESERCIZIO 2 (2 punti)

Spiegare il funzionamento della system call *fork()*: qual è il suo effetto in memoria centrale? Come si distingue il processo padre dal figlio? Mostrare un semplice esempio di utilizzo.

La system call *fork()* ha lo scopo di creare un nuovo processo. Il processo che invoca la fork prende il nome di processo padre, mentre il processo creato prende il nome di processo figlio. Appena dopo la *fork()* il processo figlio è caricato in memoria centrale in un proprio spazio di indirizzi diverso da quello del padre, ma il suo segmento codice, dati, stack e heap sono una copia di quello del padre.
L'esecuzione dei due processi avviene pero' in parallelo e quindi competono per l'accesso alla CPU.
In particolare, sia processo padre che processo figlio riprendono l'esecuzione a partire dalla prima istruzione successiva alla *fork()*.
Per distinguire il codice che del padre da quello del figlio è sufficiente ricorrere al valore di ritorno della *fork()*. Nel processo padre (il chiamante), la *fork()* restituisce il pid del processo appena creato. Nel processo figlio, invece, restituisce 0. La *fork()* restituisce -1 nell'eventualità che il processo figlio non sia stato creato, ad esempio per mancanza di risorse di sistema.
A livello di codice allora è sufficiente scrivere:

```
pid_t chld = fork();
if (chld == -1){
     <gestione dell'errore>
}
if (chld == 0){
     <codice del processo figlio>
}
else {
     <codice del processo padre>
}
```

**ESERCIZIO 3 (2 punti)**

Illustrare la gestione dei segnali in Unix. Indicare in particolare:
- con quali system call è possibile inviare e ricevere un segnale
- cosa accade quando un processo riceve un segnale

Per gestire i segnali si possono usare due system call principali.

```
int kill(pid_t pid, int sig);
```

ha lo scopo principale di mandare segnali; in particolare,
- *sig* è un numero intero da 0 a 32 (non in tutte le architetture), ciascuno dei quali corrisponde ad un particolare segnale.
- se *pid* > *0*, il segnale *sig* è mandato al processo *pid*
- se *pid* == *0*, il segnale *sig* è mandato a tutti i processi nello stesso gruppo del processo chiamante
- se *pid* == *-1*, il segnale *sig* è mandato a tutti i processi per i quali il processo chiamante ha il permesso di mandare loro segnali (*init* è escluso)

```
sighandler_t signal(int signum, sighandler_t handler);
```

dove:

```
typedef void (*sighandler_t)(int);
```

La system call **signal** consente di intercettare un segnale e di associare alla ricezione del segnale indicato da **signum** l'esecuzione di una funzione indicata dal puntatore a funzioni **handler.**

Alla ricezione di un segnale, il processo si comporta in uno dei seguenti modi. Se il segnale non è stato intercettato, allora viene eseguita un'operazione di default che puo' essere:
- non fare nulla,
- termina
- sospendi/riprendi l'esecuzione

Se invece il segnale è stato intercettato, allora la ricezione fa sì che l'esecuzione del flusso principale del processo si sospenda, venga eseguita la funzione associata al segnale, e poi si riprenda con l'esecuzione del flusso principale là dove era stato interrotto.

## ESERCIZI RELATIVI ALLA PARTE DI C

### ESERCIZIO 1 (1 punto)

a) Qual è l'output di questo programma C?
b) Quale sarebbe l'output se l'istruzione del main() con etichetta `decl:` fosse commentata?

```
#include <stdio.h>
int x=5;

void f(int y){
        printf("%d\n",y);
        if(y > 20) {
                int y = 20;
                printf("%d\n",y);
        }
        else {
                int y = -20;
                printf("%d\n",y);
        }
}

int main(void) {
        printf("%d\n",x);
decl:   int x = 60;
        f(x);
}
```

a): 5 60 20

b): 5 5 -20

### ESERCIZIO 2 (3 punti)

Si implementi una funzione `int min_oltre_soglia(int ai[], int dim, int soglia, int* pmin);` che, dati un array `ai` di `dim` interi (anche negativi) e un intero `soglia`, ricavi il valore minimo nell'array più grande di `soglia` e memorizzi tale valore nella variabile passata per riferimento. La funzione deve restituire TRUE se ha trovato tale valore e FALSE altrimenti. Definire opportunamente

TRUE e FALSE.
*Esempi*. Dato `ai={10,20,6,-10,50,-1,80}`
`min_oltre_soglia` richiamata su `ai` con `soglia=20` deve restituire TRUE e trovare come valore minimo 50
`min_oltre_soglia` richiamata su `ai` con `soglia=80` deve restituire FALSE
`min_oltre_soglia` richiamata su `ai` con `soglia=-10` deve restituire TRUE e trovare come valore minimo -1

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0

int min_oltre_soglia(int ai[], int dim, int soglia, int* pmin) {      int i;
      int found = FALSE;
      for (i=0; i<dim; i++) {
            if (ai[i] > soglia) {
                  if (found == FALSE) {
                        *pmin = ai[i];
                        found = TRUE;
                  }
                  else if (ai[i] < *pmin)
                        *pmin = ai[i];
            }
      }
            return found;
}
```

**ESERCIZIO 3 (3 punti)**

Data la struttura `node` definita come segue:

```
typedef struct node {
        int value;
        struct node *next;
} nodo;
typedef nodo* list;
```

implementare la funzione con prototipo `list ordered_insert(list head, list node);` che inserisce un nodo in una lista ordinata in modo crescente. La funzione restituisce la testa della lista.

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

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

```
typedef nodo* list;


list ordered_insert(list head, list node) {
      list temp = head, prev = NULL;

      while (temp != NULL && temp->value < node->value) {
            prev = temp;
            temp = temp->next;
      }

      if (prev == NULL) {
            node->next = head;
            head = node;
      }
      else {
            prev->next = node;
            node->next = temp;
      }

      return head;
}

int main(void) {
      list lista = NULL;

      list prova1 = (list) malloc(sizeof(nodo));
      prova1->value=3;
      prova1->next=NULL;
      lista = ordered_insert(lista,prova1);

      list prova2 = (list) malloc(sizeof(nodo));
      prova2->value=5;
      prova2->next=NULL;
      lista = ordered_insert(lista,prova2);

      list prova3 = (list) malloc(sizeof(nodo));
      prova3->value=4;
      prova3->next=NULL;
      lista = ordered_insert(lista,prova3);

      list temp = lista;
      while (temp != NULL) {
            printf("%d\n", temp->value);
            temp = temp->next;
      }
}
```