SISTEMI OPERATIVI – 12 Luglio 2016
corso A nuovo ordinamento
e parte di teoria del vecchio ordinamento indirizzo SR

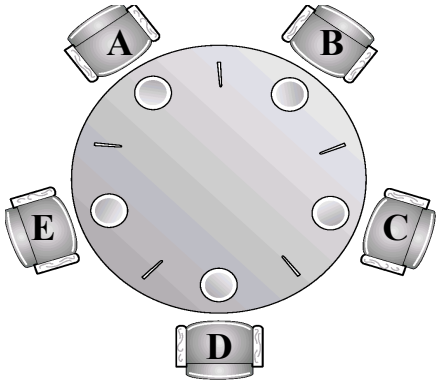
Cognome:_____ **Nome:**_____
Matricola:_____

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 (punti)

a) Si consideri il problema dei "cinque filosofi". I codici del generico filosofo e di alcune sue funzioni per una soluzione (parziale) del problema sono riportati qui di seguito. Inserite le opportune operazioni di wait e signal e di chiamata delle operazioni di gestione delle forchette necessarie per il corretto funzionamento del sistema, indicando anche i semafori necessari ed il loro valore di inizializzazione, nonché le variabili condivise



Variabili condivise:
int **state[5];**

semafori necessari con relativo valore di inizializzazione:
semaphore **mutex = 1;**
semaphore array **sem[5] = 0;**

```
filosofo (i):
while (TRUE)
{
        pensa();

        take_forks(i);

        prendi_le_due_forchette_e_mangia();

        put_forks(i);
}
```

```
test(i)
{
        if  (state[i]==AFFAMATO &&
            state[left(i)] != MANGIA &&
            state[right(i)] != MANGIA)
                { state[i]=MANGIA;
                  signal (sem[i])}
}

take_forks(i)
{
        wait (mutex);
        state[i]= AFFAMATO; // i segnala di essere affamato
        test(i);
        signal (mutex);
        wait (sem[i]); // si blocca se le forchette non sono libere
}

put_forks(i)
{
        wait (mutex);
        state[i]= PENSA;            // non è affamato e non sta mangiando
        test(left(i));              // via libera ai vicini, se possibile
        test(right(i));
        signal(mutex);
}
```

a) Discutete la soluzione prospettata con riferimento alla presenza o meno di "Deadlock" (stallo) (Sì/No), fornendo le ragioni della vostra risposta

La soluzione proposta **non** soffre del problema di "deadlock" perche' una delle 4 condizioni necessarie perché un insieme di processi entri in "deadlock", e cioè quella **dell'allocazione parziale delle risorse,** non si verifica in questa soluzione dove le due forchette necessarie per mangiare vengono prese **simultaneamente solo quando il filosofo è sicuro di poter mangiare.**

b) Discutete la soluzione prospettata con riferimento alla presenza o meno di "Starvation" (attesa illimitata) (Sì/No), fornendo le ragioni della vostra risposta

I filosofi che si coordinano usando il protocollo delineato dalla soluzione proposta **possono** andare in "starvation" se, per esempio, i due vicini di un filosofo A (quello di destra e quello di sinistra) si mettono d'accordo si alternano a mangiare avendo sempre cura di terminare di mangiare mentre l'altro sta ancora mangiando.
Infatti detti A, B e C i filosofi in questione, assumiamo che A e C stiano già mangiando quando B manifesta la sua intenzione di mangiare, ma si blocca (***sul suo semaforo che ha valore zero***) perché le forchette sono utilizzate. Se A termina di mangiare mentre C sta ancora mangiando e tenta di svegliare B (esegue l'operazione test), questa operazione non può essere completata perché la forchetta in comune tra B e C e' ancora occupata (***quindi il semaforo di A non viene posto ad uno***). Se prima che C termini, A torna a mangiare la situazione si ribalta quando C termina e tenta di cedere il passo a B che anche questa volta non può procedere. Se nuovamente C torna a mangiare prima che A termini ci ritroviamo nella situazione iniziale ed il ciclo può ripetersi all'infinito.

d) 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	7
P3	5	3
P4	9	2

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)
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 …(2) …. P2….(5)….P3….(8)….P2….(9)….P4…(11)….P2…(14)…P1….(26)

Waiting time medio:
P1 = (26 – 0) – 14 = 12;
P2 = (14 – 2) – 7 = 5;
P3 = (8 – 5) – 3 = 0;
P4 = (11 – 9) – 2 = 0;
waiting time medio = 17/4

ESERCIZIO 2 (punti)
Consideriamo un sistema operativo con memoria virtuale dove i processi che sono posti in esecuzione con degli spazi fisici allocati di dimensioni fisse e limitate danno luogo a dei page fault gestiti con opportuni algoritmi di rimpiazzamento delle pagine.
.
Supponendo di avere un processo con la seguente sequenza di riferimenti alle sue pagine (logiche):
1 3 2 4 1 4 3 5 1 3 2 4 5 6 1 3 4 6 4 1 2 6 1 3 4 2

2.a) Si illustri il funzionamento dell'algoritmo di rimpiazzamento FIFO e si descriva la cosiddetta "anomalia di Belady" assumendo di avere a disposizione (prima) una memoria fisica consistente in 3 "frame" e poi una formata da 4 "frame" (suggerimento: utilizzate gli schemi riportati qui di seguito e contate i page fault nei due casi)

S = 1 3 2 4 1 4 3 5 1 3 2 4 5 6 1 3 4 6 4 1 2 6 1 3 4 2

1	3	2	4	1	1	3	5	5	5	2	4	4	6	1	3	4	6	6	1	2	2	2	3	4	4
	1	3	2	4	4	1	3	3	3	5	2	2	4	6	1	3	4	4	6	1	1	1	2	3	3
		1	3	2	2	4	1	1	1	3	5	5	2	4	6	1	3	3	4	6	6	6	1	2	2
^	^	^	^	^		^	^			^	^		^	^	^	^	^		^	^			^	^	

3 frame: 18 page fault

S = 1 3 2 4 1 4 3 5 1 3 2 4 5 6 1 3 4 6 4 1 2 6 1 3 4 2

1	3	2	4	4	4	4	5	1	3	2	4	5	6	1	3	4	4	4	4	2	6	1	3	4	2

	1	3	2	2	2	2	4	5	1	3	2	4	5	6	1	3	3	3	3	4	2	6	1	3	4
		1	3	3	3	3	2	4	5	1	3	2	4	5	6	1	1	1	1	3	4	2	6	1	3
			1	1	1	1	3	2	4	5	1	3	2	4	5	6	6	6	6	1	3	4	2	6	1
	^	^	^	^				^	^	^	^	^	^	^	^	^				^	^	^	^	^	^

4 frame: 20 page fault

Considerando l'n-esimo riferimento contenuto nella lista dei riferimenti, gli schemi precedenti indicano lo stato della memoria allocata al processo immediatamente prima di questo riferimento (colonna n-1) e quello successivo al riferimento stesso che sarà diverso dal precedente nel caso di un page fault (colonna n). Per semplicità di calcolo (ma senza che ciò corrisponda ad una gestione ordinata delle pagine la cui vecchiaia è solamente deducibile da un "time-stamp" che denota il tempo del suo ultimo caricamento in memoria), ogni colonna è ordinata in termini di vecchiaia essendo la pagina riportata piu' in basso quella che da più tempo è in memoria e quella più in alto quella che è entrata in memoria per ultima.
Indicando con N(S,F) il numero di page fault riscontrati con una sequenza di riferimenti S avendo a disposizione F frame, l'anomalia di Belady consiste nel fatto che esistono delle combinazioni di S e F tali per cui $N(S,F) < N(S,F+1)$ contrariamente alla percezione comune che aumentando la dimensione della memoria a disposizione il numero di page fault dovrebbe diminuire (o quanto meno non dovrebbe aumentare).
Come evidenziato dall'esempio precedente l'algoritmo di rimpiazzamento FIFO porta al seguente risultato
$18 = N(S,3) < N(S,4) = 20$

2.b) Utilizzando sempre la sequenza di riferimenti precedente, quanti page fault si hanno se utilizziamo l'algoritmo di rimpiazzamento LRU avendo a disposizione una memoria formata da 4 "frame"?

S = 1 3 2 4 1 4 3 5 1 3 2 4 5 6 1 3 4 6 4 1 2 6 1 3 4 2

1	3	2	4	1	4	3	5	1	3	2	4	5	6	1	3	4	6	4	1	2	6	1	3	4	2
	1	3	2	4	1	4	3	5	1	3	2	4	5	6	1	3	4	6	4	1	2	6	1	3	4
		1	3	2	2	1	4	3	5	1	3	2	4	5	6	1	3	3	6	4	1	2	6	1	3
			1	3	3	2	1	4	4	5	1	3	2	4	5	6	1	1	3	6	4	4	2	6	1
^	^	^	^				^			^	^	^	^	^	^	^				^			^	^	^

4 frame: 16 page fault
I frame occupati dalle pagine del processo sono organizzati in una stack alla cui sommità si trova la pagina appena referenziata ed alla base quella che da più tempo non è stata utilizzata. In presenza di un riferimento ad una pagina, se questa non è presente in stack, viene liberato il frame al fondo della stack (cioè quello associato alla pagina che da più tempo non è utilizzata) ed utilizzato per memorizzare la nuova pagina, avendo cura di riorganizzare la stack in modo che questo frame che prima era ultimo diventi adesso il primo. Se la pagina è invece già presente in stack, la stessa viene riorganizzata portando in testa il frame contente la pagina in questione e riorganizzando i puntatori del resto della stack in modo che l'ordine relativo tra gli altri frame non venga alterato.

2.c) Cosa si può dire dell'anomalia di Belady nel caso di LRU?
LRU non soffre dell'anomalia di Belady.

Facendo riferimento all'esempio analizzato con questo esercizio, consideriamo la situazione della stack prima del decimo riferimento della sequenza (riferimento alla pagina 3) [1,5,3,4]. Il riferimento in questione non provoca un page fault perché la pagina 3 è già in stack e comporta solamente una riorganizzazione della stessa. Diversamente, se il decimo riferimento, anziché coinvolgere la pagina 3 fosse alla pagina 2 si avrebbe un page fault (e poi la conseguente riorganizzazione della stack).
Se noi ripetessimo l'esercizio avendo a disposizione, prima 3 frame e poi 5 frame, vedremmo che le stack corrispondenti (sempre appena precedenti al decimo riferimento) sarebbero rispettivamente [1,5,3] e [1,5,3,4,2]. Se le mettiamo in ordine [1,5,3] [1,5,3,4] [1,5,3,4,2] possiamo notare che ognuna è contenuta

in quella successiva e questo non è un caso, ma è una proprietà di questo metodo di rimpiazzamento proprio perché basato su una stack.
Possiamo allora osservare che un riferimento che non provoca un page fault con 3 frame a disposizione, non può sicuramente causare un page fault se abbiamo 4 o 5 frame (è il caso del riferimento alla pagina 3 che non darebbe page fault in nessun caso), mentre un riferimento che provoca un page fault con n frame a disposizione può non provocare un page fault avendo a disposizione più frame come è il caso nel nostro esempio se facessimo riferimento alla pagina 2 che darebbe page fault con 3 o 4 frame, ma non se avessimo 5 frame a disposizione.
Da questo ragionamento si deduce facilmente che N(S,F) >=N(S,F+1) per qualunque S e F.

2.d) Cosa si può dire dell'implementazione di LRU rispetto alla sua efficienza?

LRU è un buon algoritmo di rimpiazzamento, perché si avvicina più a OPT (algoritmo ottimale, non implementabile perché assume di conoscere il futuro) che a FIFO, **ma è difficile da implementare in modo efficiente. La gestione dello stack è costosissima se fatta via SW (lo stack va aggiornato ad ogni riferimento alla Memoria Principale)**

ESERCIZIO 3 (4 punti)

Si descrivano, se lo si desidera usando semplicemente opportuni disegni, le seguenti 4 forme di allocazione dello spazio in memoria secondaria:

1. allocazione indicizzata, schema concatenato
2. allocazione indicizzata, schema a più livelli
3. allocazione adottata nel sistema Unix mediante i-node
4. File Allocation Table. In quest'ultimo caso, riportate la FAT di un hard disk formato da 16 blocchi e che contiene un solo file memorizzato nei blocchi 9, 4, 14, 6 (esattamente in questo ordine).

Si vedano le sezioni 11.4.2 e 11.4.3 dei lucidi usati a lezione.

SOLUZIONE ESERCIZI RELATIVI ALLA PARTE DI UNIX

ESERCIZIO 1 (2 punti)

Si spieghi il funzionamento della system call msgrcv(int msqid, void *msgp, size_t maxmsgsz, long msgtyp, int msgflg) sulle code di messaggi insieme ad una descrizione degli argomenti (con particolare attenzione a msgtyp):

Soluzione

La syscall msgrcv() serve per ricevere un messaggio da una coda di messaggi.
Il primo argomento è un identificatore della coda di messaggi. Il secondo è un puntatore a una struttura 'programmer-defined' utilizzata per contenere il messaggio da ricevere, con questa forma:

```
struct mymsg {
  long mtype;   /* Message type */
  char mtext[]; /* Message body */
}
```

Il terzo argomento indica la capienza massima del membro mtext per poter effettivamente scaricare il messaggio dalla coda (la msgrcv fallisce con errore E2BIG in caso contrario). L'argomento msgtyp indica la modalità di lettura: fifo (0), accesso diretto (>0), o per priorità (<0). L'ultimo argomento msgflg è una maschera di bit formata mettendo in OR zero o più flag (ad es. IPC_NOWAIT per ricezione 'nonblocking', etc.).

ESERCIZIO 2 (1 punto)

Illustrare il funzionamento di 3 argomenti del comando *ls*.

-l: lista in formato *long*, che mostra la dimensione dei file, i permessi, etc.;
-t: lista ordinando per timestamp (data dell'ultima modifica);
-a: lista tutto, inclusi file nascosti, che iniziano con il carattere '.';
-i: lista mostrando l'inode;
-R: lista ricorsivamente le sottodirectory di quella corrente e il loro contenuto.

ESERCIZIO 3 (3 punti)

Si illustri lo pseudo-codice minimo necessario per realizzare la comunicazione unidirezionale via pipe fra due processi fratelli della stringa "Hello, world!\n".

Soluzione

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

int main(void) {
   int    fd[2], nbytes;pid_t   childpid;
   char   string[] = "Hello, world!\n";
   char   readbuffer[80];

   if (pipe(fd) == -1) {  /* handle error */ exit(1); }

   switch (fork()) {
```

```
    case -1: /* handle error */
      break;

    case 0: /* child 1 - writing end */
      close(fd[0]); // close read end
      // your logic for child 1 here, using fildes[1] to write
      write(fd[1], string, (strlen(string)+1)); // +1 is termination \0
      return (0);

    default: /* not child 1 */
      close(fd[1]); // nobody else can write
      break;
  }

  switch (fork()) {
    case -1: /* handle error */
      break;

    case 0: /* child 2 - reading end */
      // your logic for child 2 here, using fildes[0] to read
      nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
      printf("Received string [%d bytes]:%s", nbytes, readbuffer);
      return (0);
  }

  // still in the parent, logic for the common parent here,
  // after forking both children, e.g. termination control
  // waitpid, wait, etc
  return (0);
}
```

SOLUZIONE ESERCIZI RELATIVI ALLA PARTE DI C

ESERCIZIO 1 (3 punti)

Implementate la funzione con prototipo

```
int leftpad(char* str, char ch, int len, int maxstrlen);
```

La funzione modifica la stringa *str* aggiungendo caratteri **a sinistra** della stringa. *ch* è il carattere da aggiungere, *len* è il numero di volte con cui ripetere il carattere *ch*, *maxstrlen* è la lunghezza massima possibile della stringa. La funzione restituisce il numero di caratteri aggiunti. Ricordatevi di gestire i casi i cui i caratteri non devono o non possono essere aggiunti e in cui *str* è nulla.

Esempi:

- str: **pippo** char: / len: **0** maxstrlen: **10** valore restituito: **0** str dopo la chiamata: **pippo**
- str: **pippo** char: / len: **5** maxstrlen: **10** valore restituito: **0** str dopo la chiamata: **pippo**
- str: **pippo** char: / len: **3** maxstrlen: **10** valore restituito: **3** str dopo la chiamata: **///pippo**

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

#define MAXSTRLEN 10

int leftpad(char* str, char ch, int len, int maxstrlen) {
  if (str == NULL || len <= 0 || strlen(str)+len+1>maxstrlen)
    return 0;

  int i;
  for (i = strlen(str); i >=0; i--) {
    str[i+len] = str[i];
  }
  for (i = 0; i < len; i++) {
    str[i] = ch;
  }

  return len;
}

int main(void) {
  char str[MAXSTRLEN]="pippo";
  int pad, ret;

  pad = 0;
  ret = leftpad(str, '/', pad, MAXSTRLEN);
  printf("%s %d\n", str, ret);

  pad = 5;
  ret = leftpad(str, '/', pad, MAXSTRLEN);
  printf("%s %d\n", str, ret);

  pad = 3;
  ret = leftpad(str, '/', pad, MAXSTRLEN);
  printf("%s %d\n", str, ret);

  return 0;
}
```

ESERCIZIO 2 (1 punto)

a) Qual è l'output del seguente programma C?
b) E quale sarebbe l'output se nel programma l'istruzione

```
int x = 60;
```

fosse commentata?

```
#include <stdio.h>

int x=20;

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

int main(void) {
   int x = 60; // Da commentare

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

a) 60 60 20
b) 20 20 -20

ESERCIZIO 3 (3 punti)

Data la struttura `node` definita come segue:

```
#define MAXSTR 255

typedef struct node {
     char str[MAXSTR];
     struct node * next;
} nodo;
typedef nodo* link;
```

implementare la funzione con prototipo

```
int isSorted(link head);
```

isSorted è una funzione che restituisce TRUE se gli elementi della lista `head` sono ordinati e FALSE altrimenti. Definite opportunamente TRUE e FALSE e gestite il caso di lista vuota. Potete usare la funzione *int strcmp(char* s1, char* s2)*, che restituisce 0 se *s1* e *s2* sono uguali, un intero maggiore di 0 se *s1* è più grande di *s2* e un intero minore di 0 se *s1* è più piccolo di *s2*.

Ad esempio, data la lista `head`: “abate” → “abbastanza” → “abbastanza” → “zuzzurellone” → `NULL`, la funzione `isSorted` restituisce TRUE.

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

#define TRUE 1
#define FALSE 0
#define MAXSTR 255

typedef struct node {
      char str[MAXSTR];
      struct node* next;
} nodo;
typedef nodo* link;

int isSorted(link head) {
      int sorted = TRUE;

      if (head != NULL) {
            while (head->next != NULL && strcmp(head->str, head->next->str) <=
0) {
                  head = head->next;
            }

            if (head->next != NULL)
                  sorted = FALSE;
      }

      return sorted;
}

int main(void) {

      link prova = (link) malloc(sizeof(nodo));
      strcpy(prova->str, "zuzzurellone");
      prova->next = NULL;

      link prova2 = (link) malloc(sizeof(nodo));
      strcpy(prova2->str, "abbastanza");
      prova2->next = prova;

      link prova3 = (link) malloc(sizeof(nodo));
      strcpy(prova3->str, "abbastanza");
      prova3->next = prova2;

      link prova4 = (link) malloc(sizeof(nodo));
      strcpy(prova4->str, "abate");
      prova4->next = prova3;

      int sorted;

      sorted = isSorted(prova4);
      printf("iSorted = %d\n", sorted);

      return 0;
}
```