SISTEMI OPERATIVI – 17 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*", 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	7
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 …(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

b)
Riportate lo pseudocodice che descrive la corretta implementazione dell'operazione di WAIT, e dite che informazione ci fornisce il valore corrente della variabile semaforica di un semaforo

Si vedano i lucidi della sezione 6.5.2.

c) Siano dati due programmi concorrenti A e B. All'interno di A viene eseguita la procedura Pa e all'interno di B viene eseguita la procedura Pb. Si vuole essere certi che Pa venga eseguita prima di Pb, indipendentemente da come A e B vengono schedulati all'interno del sistema operativo. Fornite una possible soluzione al problema.

Si vedano i lucidi della sezione 6.5.1.

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.

a) Qual è, in nanosecondi, il tempo medio di accesso (Medium Access Time) in RAM (esplicitate i calcoli che fate)?

   ma = 0,80 * (80+20) + 0,20 * (2*80 + 20) = 80 + 36 = 116 nanosecondi

b) Il sistema viene ora dotato di memoria virtuale, usando come algoritmo di rimpiazzamento quello della *seconda chance migliorato*. Si supponga pari a 10 millisecondi il tempo necessario a gestire un page fault. Se vogliamo una degradazione massima dell'Effective Access Time del 20% rispetto alla versione del sistema senza memoria virtuale, quale deve essere il valore massimo di "p", ossia la probabilità che si verifichi un page fault? (esplicitate il ragionamento e le formule che usate, ed è sufficiente esprimere "p" mediante la formula che permette di calcolarlo)

   A partire dalla formula:

   eat = (1-p)·ma + p·"tempo di gestione del page fault"

   usando i dati del problema, sapendo che vogliamo eat massimo pari a ma + 20% ma, abbiamo:

   max eat = 116 + 0,2·116 = 139,2 ≥ (1-p) ·116 + p · 10.000.000

   e risolvendo rispetto a p abbiamo:

$139{,}2 \geq 116 + p \cdot (10.000.000 - 116)$;
$p < (139{,}2 - 116) / (10.000.000 - 116)$ (ossia all'incirca meno di un page fault ogni 2.300.000 riferimenti in memoria)

c) E' possibile che un processo di questo sistema sistema si veda aumentare il numero di frame che usa e contemporaneamente aumenta anche il numero di page fault generati dal processo?

Si, perché nel caso peggiore l'algoritmo si comporta come FIFO, che soffre dell'anomalia di Belady.

d) Quali informazioni conterrà ciascuna entry di una page table di un sistema che usa l'algoritmo della seconda chance migliorato, e quale vantaggio fornisce questo algoritmo rispetto a quello della seconda chance?

Numero di un frame, bit di validità, bit di riferimento, dirty bit.
Il vantaggio è che, usando anche il dirty bit, si può evitare di salvare la pagina vittima, se questa non è stata modificata.

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 –l A B; 3) rm A B; 4) cp A B

i comandi 2) e 3)

d) Che differenza c'è tra un sistema RAID di livello 4 ed uno di livello 5?

Nel livello 5 gli strip di parità sono distribuiti omogeneamente tra tutti i dischi, che quindi vengono sollecitati in modo uniforme. Nel livello 4 gli strip di parità risiedono su un unico disco, che viene quindi sollecitato mediamente più di tutti gli altri, dovendo essere aggiornato ad ogni modifica, aggiunta o cancellazione di un qualsiasi file

ESERCIZI RELATIVI ALLA PARTE DI UNIX (6 punti)

ESERCIZIO 1 (2 punti)

(1.1) illustrare almeno 3 opzioni del comando ls.

(1 punti)

Soluzione [01_introduzione_UNIX.pdf]

-l	lista i contenuti di una directory in formato lungo, elencando la dimensione di ogni file, permessi, owner, data di modifica;
-t	lista i contenuti di una directory ordinati per timestamp (tempo dell'ultima modifica).
-a	lista i contenuti di una directory, inclusi i file nascosti i cui nomi iniziano con il carattere '.' .
-i	lista i contenuti di una directory incluso l'inode.
-R	lista i contenuti di una directory ricorsivamente, elencando cioè i contenuti di tutte le sottodirectory.

(1.2) A cosa serve la system call *wait*? Nel rispondere considerare eventuali argomenti e/o valori restituiti.

(1 punto)

Soluzione [slides 03_creazione_terminazione_processi.pdf]

La system call *wait(&status)* ha due fini. Sospende l'esecuzione del processo chiamante finché uno dei figli non termina la propria esecuzione. Secondariamente, lo stato di terminazione del figlio è restituito nell'argomento *status*. *wait()* restituisce il process ID del figlio che ha terminato la propria esecuzione: il suo prototipo è infatti *pid_t wait(int *status);* .

ESERCIZIO 2 (2 punti)

Descrivere dettagliatamente la system call *pid_t fork(void);*

Soluzione [slides 03_controllo_processi.pdf]

La system call *fork()* crea un nuovo processo, il figlio, che è una copia quasi esatta del processo chiamante, il padre. Dopo l'esecuzione della *fork()*, esistono 2 processi e in ciascuno l'esecuzione riprende dal punto in cui la *fork()* restituisce.

Nello spazio di indirizzamento del processo figlio viene creata una copia delle variabili del padre, col valore loro assegnato al momento della fork. Il nuovo processo incomincia l'esecuzione a partire dalla prima istruzione successiva alla fork che lo ha creato.
I due processi eseguono lo stesso testo, ma mantenendo copie distinte di stack, data, e heap. Stack, dati, e heap del figlio sono inizialmente esatti duplicati delle corrispondenti parti della memoria del padre. Dopo la *fork()* ogni processo può modificare le variabili in tali segmenti senza influenzare l'altro.

All'interno del codice di un programma possiamo distinguere i due processi per mezzo del valore restituito dalla *fork()*. Nell'ambiente del padre, *fork()* restituisce il process ID del figlio appena creato. È

utile perché il padre può creare -e tenere traccia di- vari figli. Per attenderne la terminazione può usare la *wait()* o altra syscall della stessa famiglia. Nell'ambiente del figlio la *fork()* restituisce 0. Se necessario il figlio può ottenere il proprio process ID con la *getpid()*, e il process ID del padre con la *getppid()*.

ESERCIZIO 3 (2 punti)

(3.1) Illustrare a cosa serve l'utility *make*, come si usa un Makefile, quali sono gli elementi principali del Makefile, e come implementare il comando *make clean*.

(1 punti)

Soluzione [slides 02_integrazione_linguaggio]

L'utility make è uno strumento che può essere utilizzato per automatizzare il processo di compilazione. Si basa sull'utilizzo di un file (il *makefile*, appunto) che descrive le dipendenze presenti nel progetto e si avvale delle informazioni relative alla marcatura temporale dei file. In tal modo è possibile ricompilare solo i target file più vecchi dei sorgenti.
Un Makefile contiene un insieme di *target*, le regole per la compilazione e l'istruzione da eseguire per compilare a partire dai sorgenti. In particolare le *righe di dipendenza* illustrano da quali file oggetto dipende il file target; tale riga inizia dalla colonna 1, per esempio

nome_target: file_1.o file_2.o file_n.o

Le *righe d'azione* o *di comando* indicano come il programma deve essere compilato nel caso sia stato modificato almeno uno dei file *.o*; deve iniziare con una *tabulazione*. Per esempio,

```
gcc -o nome_target file_1.o file_2.o file_n.o
```

Per implementare il comando *make clean:*

```
clean:
    rm -f *.o
```

(3.2) Illustrare il funzionamento della system call *msgsnd()* e fornire un esempio di utilizzo.

(1 punti)

Soluzione [slides 07_code_messaggi]

La syscall *msgsnd()* invia un messaggio a una coda di messaggi. Il suo prototipo è

int **msgsnd**(int *msqid*, const void **msgp*, size_t *msgsz*, int *msgflg*);

restituisce 0 in caso di successo, -1 in caso di errore. Il primo argomento è un intero che funge da l'identificatore della coda; il secondo è un puntatore a una struttura definita dal programmatore utilizzata per contenere il messaggio inviato. L'argomento *msgsz* indica la dimensione del messaggio (espresso in byte), mentre l'ultimo argomento è una bit mask di flag che controllano l'operazione di invio. Per esempio utilizzando IPC_NOWAIT, nel caso la coda di messaggi sia piena, invece di bloccarsi in attesa che si renda disponibile dello spazio, la *msgsnd()* restituisce immediatamente (con errore *EAGAIN*).
Un'invocazione tipica della syscall è quindi

msgsnd(m_id, &q, sizeof(q), IPC_NOWAIT)

ESERCIZIO 1 (2 punti)

Si implementi la funzione con prototipo

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

`strsame` è una funzione restituisce 1 se la stringa `str` è composta da caratteri tutti uguali tra loro e 0 altrimenti. Ipotizzate che la funzione sia sempre richiamata con parametro `str` diverso da NULL e mai con la stringa vuota.

```
int strsame(char * str) {
     int ret_value = 1;

     while (*(str+1) != '\0' && ret_value==1) {
          if((*str) != *(str+1))
          ret_value = 0;
          str++;
     }
     return ret_value;
}
```

ESERCIZIO 2 (3 punti)

Si implementi la funzione con prototipo

```
int count_even(int * numbers, int length, int limit);
```

`count_even` è una funzione che restituisce 1 se l'array di numeri interi `numbers` (la cui lunghezza è data dal parametro `length`) contiene un numero di elementi pari superiore o uguale al parametro `limit` e 0 altrimenti. Ipotizzate che la funzione sia sempre richiamata con parametro `numbers` diverso da NULL.

```
int count_even(int * numbers, int length, int limit){
     int index,neven=0;

     for(index=0; index < length; index++){
          if(numbers[index]%2==0)
               neven++;
          }
     if(neven >= limit)
          return 1;
```

```
     else
          return 0;
}
```

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 verify_last_element(link head, int value);
```

`verify_last_element` è una funzione che restituisce 1 se l'ultimo elemento della lista è uguale al parametro `value` e 0 altrimenti. Se la lista è vuota la funzione restituisce, invece, -1.

```
int verify_last_element(link head, int value){
     int ret_value = -1;

     if(head != NULL) {
          while (head->next != NULL) {
               head = head->next;
          }
          if (head->value == value)
               ret_value = 1;
          else
               ret_value = 0;
     }
     return ret_value;
}
```