Fundamentos de Sistemas de Operação MIEI 2013/2014

1º Teste, 19 Outubro, 2013, 2 horas – versão A

Avisos: *Sem consulta; a interpretação do enunciado é da responsabilidade do aluno; se necessário indique a sua interpretação. No fim deste enunciado encontra os protótipos de funções que lhe podem ser úteis.*

Parte I – 10 perguntas com respostas "até 3 linhas", cada 1 valor

Questão 1

Indique, justificando, qual dos seguintes componentes é responsável pela inicialização do *Program Counter* do CPU para iniciar a execução de um programa: compilador, ligador (linker), núcleo do sistema (kernel), boot ROM.

Questão 2

O bit M do registo *status word* do processador (PSW) indica se o CPU pode executar instruções privilegiadas (se o bit M é 1) ou se não pode (se o bit M é 0). Indique o valor desse bit M em cada uma das seguintes situações:

a) o processo executa o código do programa do utilizador: M = _____

b) o processo efetuou uma chamada ao sistema e executa código do kernel. M = _____

c) o código que atende a interrupção do relógio (*timer*) é executado. M = _____

Questão 3

Descreva o conteúdo do descritor de um processo (*process descriptor* ou *process control block*).

Questão 4

Um escalonador de CPU usa uma única fila de processos prontos (READY queue) e um *time slice T*. Suponha que um processo estava no estado RUNNING e o seu *time slice* terminou. Explique porque é que o sistema de operação tem de guardar os registos do CPU no respectivo *process descriptor*.

Questão 5

Considere um sistema onde o escalonador de CPU usa uma única fila de processos prontos (READY queue) e um *time slice T*. Existem apenas cinco processos que são puramente "CPU-bound". Explique porque é que nesta situação cada um dos processos recebe idêntico tempo de CPU.

Questão 6

Considere o seguinte fragmento de código:

```
p = fork();
if( p == 0){
       args[0]="./xpto"; args[1]= NULL;
       if (execvp(args[0], args) < 0) fork();
} else  fork();
```

Quantos processos são criados se o ficheiro executável `xpto` existir na diretoria corrente? E quantos são criados se não existir? Justifique.

Questão 7

Considere o seguinte fragmento de código para implementar um Shell UNIX muito simples. Preencha os espaços em branco no código.

```
int main () {
    char * prog = NULL ; char ** args = NULL ;
    // read the next line from the input and parse it into the program name and its arguments
    // return false if we've reached the end of the input
    while ( readAndParseCmdLine (& prog , & args )) {
        int child_pid = fork (); // create a child process to run the command
        if ( __________ ) {
            __________ (prog , args ); // run the program
        } else { // I'm the parent waiting for the child process
            _________________________;
            return 0;
        }
    }
}
```

Questão 8

Considere uma variável y inicializada a 12 num programa que executa o código seguinte em dois *threads*:

Thread 0		Thread 1
`x = y +1;`		`y = y * 2;`

Indique os possíveis valores finais da variável x ? Justifique.

Questão 9

Complete o código seguinte para garantir que é sempre impresso o valor correto (3000000).

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
int cont = 0;



void *worker(void *arg) {
     int n, c;
     n = (int)arg; c = 0;


     for( i= 0; i < n; i++ )


          c ++;


     cont = cont + c;


}


int main(int argc, char *argv[]) {
    pthread_t p1, p2, p3;



    pthread_create(&p1, NULL, worker, (void*)1000000);
    pthread_create(&p2, NULL, worker, (void*)1000000);
    pthread_create(&p3, NULL, worker, (void*)1000000);
    pthread_join(p1, NULL); pthread_join(p2, NULL); pthread_join(p3, NULL);

    printf("%d\n", cont);
    return 0;
}
```

Questão 10

O protótipo da função *pthread_cond_wait,* que faz parte da biblioteca de Pthreads é o seguinte:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Explique porque é que é necessário o segundo argumento (mutex).

Parte II – 5 perguntas com respostas com "mais de 3 linhas", cada 2 valores

Questão 11

Explique a diferença (se existir) entre os tempos gastos na chamada de uma função e numa chamada ao sistema (como *getpid()* ou *getuid()*), discutindo o que acontece em cada caso.

Questão 12

Para suportar o que no livro OSTEP é chamado de *limited direct execution*, são necessários os três seguintes mecanismos: instruções privilegiadas, proteção de memória, e interrupção de relógio (*timer*). Explique o que pode correr mal se não existir cada um destes mecanismos, assumindo em cada caso que os outros dois existem.

Questão 13

Suponha que um determinado sistema de operação tem um escalonador com duas filas de processos prontos, QA e QB, sendo que QA tem maior prioridade do que QB. Neste caso, os processos em QB só executam quando QA está vazia. Quando um processo é criado é colocado na fila QA. Para cada fila usa-se um algoritmo Round Robin e um *time slice* T.

a) Considere um processo no estado RUNNING. Em que situações pode este processo deixar de usar o CPU?
b) Considere que se um processo da fila QA usar todo o seu time slice T, é recolocado na fila QB. Explique porque esta abordagem favorece os processos I/O bound.
c) A abordagem descrita em b) tem uma grande desvantagem. Indique essa desvantagem e como é que o algoritmo pode ser alterado para a evitar.

Questão 14

Suponha que um CPU tem uma instrução máquina FUTEX com dois operandos, **ad** e **v**. O hardware garante que a instrução é indivisível mesmo que existam múltiplos processadores. Esta instrução tem o comportamento descrito no código C seguinte:

```
int  futex(  void * ad,  int v ){  // indivisible execution
       int t = *ad; *ad = v;
       return t;
}
```

Usando esta instrução complete o código seguinte onde *lock( )* e *unlock( )* se comportam como esperado. Indique em *init( )* a inicialização do valor da variável que mantém o estado do lock (open/closed).

```
void lock ( int *v ){




}

void unlock ( int *v ){




}
```

```
// sets initial value:
void init( int *v ) {

    *v = ________________;

}
```

Questão 15

Complete o programa seguinte por forma a garantir que é sempre impresso o valor **24**.

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



int y = 5;
int x;

void *worker1(void *arg) {



     x = y - 1;


}
void *worker2(void *arg) {



     y = y * y;


}

int main(int argc, char *argv[])
{
    pthread_t p1, p2;



    pthread_create(&p1, NULL, worker1, NULL);
    pthread_create(&p2, NULL, worker2, NULL);
    pthread_join(p1, NULL); pthread_join(p2, NULL);


    printf("%d\n", x);
    return 0;
}
```

Algumas chamadas ao sistema UNIX/Linux _____

int fork( )

int execvp(char *executable_file, char * args[ ])

int wait(int *status)

Algumas funções da biblioteca de Pthreads

int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg)

int pthread_join (pthread_t thread, void **retval)

int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)

int pthread_mutex_lock (pthread_mutex_t *mutex)

int pthread_mutex_unlock (pthread_mutex_t *mutex)

int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)

int pthread_cond_signal(pthread_cond_t *cond)