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.

O kernel, porque é este que carrega o programa em memória; no cabeçalho do ficheiro executável está o endereço onde se inicia a execução; o kernel coloca este valor no *Program Counter*

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 = __0___

b) o processo efetuou uma chamada ao sistema e executa código do kernel. M = __1___

c) o código que atende a interrupção do relógio (*timer*) é executado. M = __1___

Questão 3

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

- Espaço para guardar o estado da máquina virtual do processo: conteúdo dos registos do CPU, informação sobre as zonas da RAM que lhe estão atribuídas e tabela de canais abertos

- Informação de gestão que permite ao algoritmo de escalonamento tomar decisões: pid, estado, apontadores que permitem inserção em filas, indicadores sobre a ocupação de recursos (tempo de CPU gasto recentemente, nº de operações de I/O realizadas, por exemplo)

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*.

O SO cria uma máquina virtual que simula a cada processo a existência de um processador virtual igual ao CPU real que lhe é dedicado. Quando o processo está a correr executa código que muda o conteúdo dos registos; quando perde o CPU os registos são guardados para garantir que, quando o CPU real lhe é de novo atribuído a computação continua no ponto em que se encontrava. Para garantir isso o SO carrega nos registos do CPU os conteúdos que foram salvaguardados da última vez que o CPU foi retirado ao processo.

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.

Os 5 processos são CPU-bound portanto só largam o CPU quando a fatia de tempo expira. Quando um processo perde o CPU vai para o fim da fila e só corre depois dos outros 4 também usarem toda a sua fatia de tempo. Assim cada processo está T segundos Running e depois está 4 T segundos Ready. Todos os processos usam 1/(1+4) do tempo ou seja cada um dos 5 processos recebe 1/5 do tempo.

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.

Se *xtpo* existir são criados 2 processos; se *xpto* não existir serão 3.

Na linha 1, o processo *PPai* cria um processo *PFilho*; na linha 5 é criado também um, esta linha é apenas executada pelo processo *PPai*. O processo *PFilho* executa as linhas 3 e 4 que são a chamada de *execvp*; se tudo correr bem o processo *PFilho* executa o programa guardado em *./xpto*; se *./xpto* não existir *execvp* retorna um valor negativo, a condição do *if* é verdadeira e é criado um novo processo

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 ( _child_pid == 0_ ) {
            _execvp_ (prog , args ); // run the program
        } else { // I'm the parent waiting for the child process
            __wait (NULL) _____;
            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.

Supondo que o Thread 0 executa primeiro e completamente a sua instrução e posteriormente o thread 1 executa a sua, x terá o valor final de 13.

Supondo que o Thread 1 executa primeiro e completamente a sua instrução e posteriormente o thread 0 executa a sua, x terá o valor 25.

De notar ainda que y é uma variável partilhada entre os dois threads, sendo por exemplo possível que o thread 0 leia o valor de y no meio de uma actualização que está a ser feita pelo thread 1.

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 ++;

     pthread_mutex_lock( &ex );
     cont = cont + c;
     pthread_mutex_unlock( &ex );

}


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

    pthread_mutex_t ex;
    pthread_mutex_init( &ex, NULL);

    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).

Quando se invoca *pthread_cond_wait* fez-se previamente a avaliação de uma condição que por ser verdade vai provocar o bloqueio do thread invocador. A avaliação da condição envolve uma ou mais variáveis partilhadas entre os threads e portanto tem de haver um *mutex* que assegure que essa avaliação é feita em exclusão mútua. A implementação do *pthread_cond_wait* tem de garantir que (1) antes do processo se bloquear este mutex é libertado senão mais ninguém poderá mexer nas variáveis partilhadas e (2) que o thread que é acordado pelo signal readquire o mutex antes de terminar o *pthread_cond_wait*.

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.

Uma chamada de uma função envolve colocar argumentos no stack e fazer a instrução máquina CALL; o CPU mantém-se em modo utilizador; segue-se a execução da função terminada pela instrução máquina RETURN

Uma chamada ao sistema inclui a preparação dos parâmetros da chamada, a execução de uma instrução máquina especial (interrupção por software), mudança de modo do CPU, verificação dos parâmetros e mudança de pilha; segue-se a execução do código do sistema que implementa o serviço pedido que é concluído pela mudança de pilha e pela instrução máquina INTERRUPT RETURN que muda o CPU para modo utilizador.

Pela descrição feita, é fácil concluir que, mesmo executando código de igual duração, uma chamada ao sistema demorará mais tempo do que a chamada de uma função inteiramente em código do utilizador.

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.

Se existissem instruções privilegiadas e protecção de memória mas não existisse timer, um processo poderia entrar em loop e monopolizar o CPU; assim o SO não poderia dividir o CPU com justiça pelos vários processos.

Se existissem instruções privilegiadas e timer mas não existisse protecção de memória, um processo poderia escrever em zonas de memória que não lhe pertencem e sobre o próprio código do SO. Neste caso, o SO não poderia assegurar que os resultados produzidos por um processo fossem os mesmos, quer corra isoladamente quer ao mesmo tempo que outros processos.

Se existissem protecção de memória e timer mas não existissem instruções privilegiadas, um processo podia reprogramar o hardware, mudando a duração da fatia de tempo, danificando os periféricos, mudando o conteúdo da parte dos meta-dados do sistema de ficheiros, etc.

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.

a) Quando termina, esgota a sua fatia de tempo, ou faz uma chamada ao sistema (por exemplo IO) que implique bloqueio.
b) Os processos I/O bound bloqueiam-se com muita frequência pelo que nunca usarão toda a sua fatia de tempo. Assim sendo mantêm-se na fila QA e portanto são escolhidos para correr primeiro do que os que estão na fila QB; um processo da fila QB só será escolhido para correr se não houver nenhum processo na fila QA.
c) Os processos na fila QB podem passar muito tempo sem correr (starvation). Outro defeito é que um processo que desce para a fila QB não tem forma de voltar a QA. Isto é um processo que comece por ser CPU bound mas depois seja IO bound fica sempre na fila QB. Ambos os problemas apontados podem ser resolvidos se periodicamente for reavaliada a prioridade. Por exemplo, periodicamente todos os processos da fila QB passam para QA; ou se um processo na fila QB não ocupa toda a fatia de tempo passa para a fila QA.

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 ){

    while (futex( v, 1) == 1) ;

}

void unlock ( int *v ){

    *v = __0___;
}
```

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

        *v = __0___;

}
```

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  worker2Done;
int y = 5;
int x;

void *worker1(void *arg) {
   pthread_mutex_lock( &ex );
    while (!worker2Done) pthread_cond_wait( &order, & ex ); // também podia ser if
     x = y - 1;
    pthread_mutex_unlock( &ex );
}
void *worker2(void *arg) {
     pthread_mutex_lock( &ex );     // não obrigatório neste caso
     y = y * y;
     pthread_mutex_unlock( &ex );
     worker2Done = 1 ;
     pthread_cond_signal( &order ) ;
}

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

     pthread_mutex_t ex;
     pthread_cond_t order;
     pthread_mutex_init( &ex, NULL);
     pthread_cond_init( &order );
     worker2Done = 0;
    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;
}
```