

Fundamentos de Sistemas de Operação MIEI 2015/2016

1º Teste, 20 Outubro de 2015, 2h – versão B

Nome:

Nº

Avisos: Sem consulta; a interpretação do enunciado é da responsabilidade do estudante; se necessário explicita na resposta a sua interpretação. Na última página do enunciado encontra os protótipos de chamadas ao sistema e de funções da biblioteca Pthreads que lhe podem ser úteis.

Questão 1 – 2,0 valores

Considere um sistema operativo como o Linux. Considere o seguinte código *assembly*

- (1) `movl $20, %eax` ; mover para `eax` o valor 20 que indica que se pretende fazer um `getpid()`
- (2) `int 0x80` ; invocação dos serviços do sistema operativo
- (3) `movl %eax, ...` ; valor retornado pela chamada ao sistema é guardado

Explique em detalhe o que se passa entre a execução da instrução na linha (2) e a execução da instrução na linha (3).

A instrução de linha (2) gera uma interrupção por software

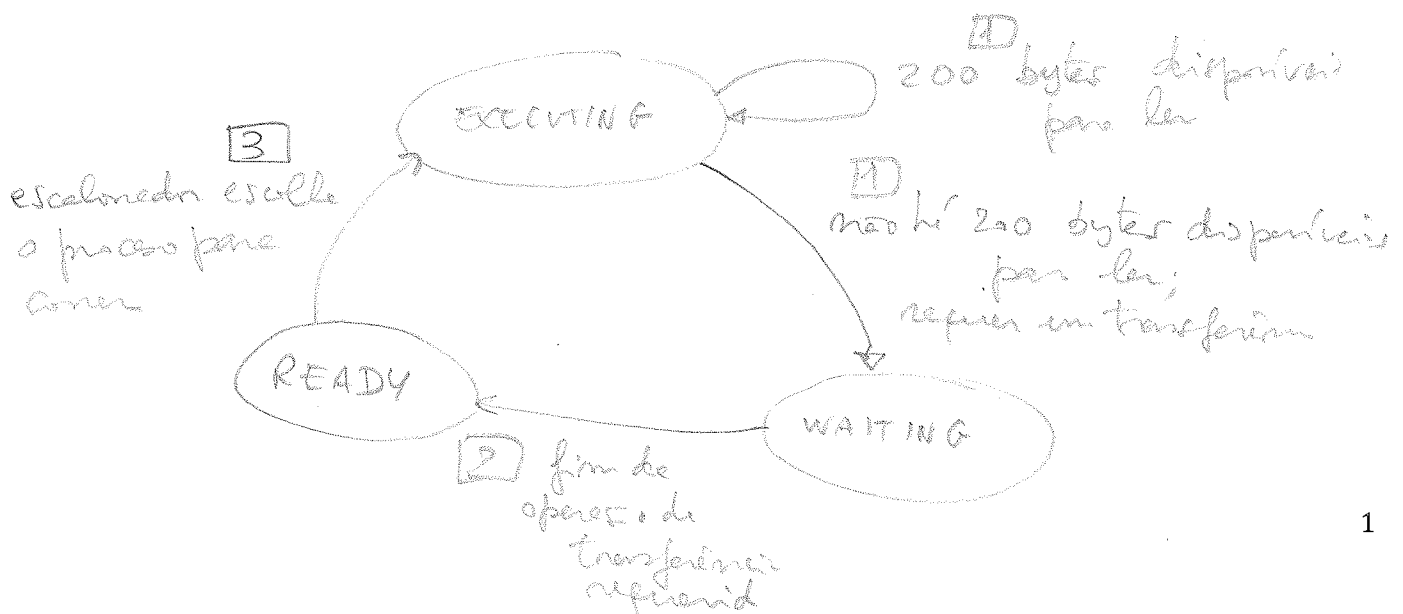
- o CPU muda para modo supervisor
- é salvaguardado o `ip` (a partir da linha (3)) e as flags
- o controlador passa para o endereço que está no endereço `0x80` do vector de interrupções
- após a execução do chamado ao sistema, o resultado volta para `eax`
- a última instrução do código do SO executa o `iret`, que repõe o CPU em modo usuário

Questão 2 – 2,0 valores

Considere um sistema operativo que suporta múltiplos processos carregados em memória em que em cada processo existe apenas um fluxo de execução (um único thread). Faça um diagrama de estados em que mostre os estados porque um processo passa quando faz uma chamada ao sistema operativo para ler 200 bytes de ficheiro:

```
char buf[200];  
...  
int f = open("xpto.c", ...); // suponha que é retornado um valor maior do que 0  
...  
int nc = read( f, buf, 200 );
```

Nos arcos que ligam os estados, indique o que é que provoca a mudança de estado em causa.



Questão 3 – 2,0 valores

Quando, num sistema UNIX, um processo executa a função `pthread_create(...)` o sistema operativo cria um novo processo e associa-lhe uma máquina virtual nova. Diga qual é o estado inicial da máquina virtual criada.

A máquina virtual criada partilha parte dos recursos com o processo criador

- entradas/saídas: mem. tabela de canais abertos do pai e pai
- memória: partilha de memória; código e dados partilhados; stack privada
- CPU: PC/IP ← endereço do sistema pai e arg. do `pthread_create`

Questão 4 – 2.0 valores

Considere o seguinte fragmento de programa.

```
#include <stdlib.h>

void servidor( ) {
    printf("Servidor\n"); return;
}

void cliente( ){
    printf("Cliente\n"); return;
}

int main( ){
    int p1, p2, w1, w2;
    p1 = fork( );
    if ( p1 == 0 ) servidor( );
    else {
        p2 = fork ( );
        if (p2 == 0) cliente( );
    }
    wait( &w1); wait( &w2);
    return 0;
}
```

- a) Suponha que se executa o programa acima a partir de um terminal. Indique, justificando, o que é aparece no terminal.

Servidor ou cliente
cliente ou Servidor
dependendo da ordem de escalonamento dos processos criados

- b) Os processos filhos executam as chamadas ao sistema `wait(...)` que estão imediatamente antes do `return 0`? Em caso afirmativo, como poderia evitar que isso acontecesse?

Os processos filhos executam o `wait()` o fim de erro.

Há várias soluções. A mais simples é acrescentar um `exit` no final das funções `servidor()` e `cliente()`.

Questão 5 – 2,0 valores

Considere um sistema operativo que suporta multi-programação e que usa uma única fila READY com uma estratégia *Round-Robin*. O SO mantém para cada processo um *descriptor*. Como é sabido, uma parte da informação guardada no descriptor é o '*Estado da Computação*'.

a) Qual o conteúdo desta parte do descriptor?

Os 3 componentes de informação vinculados ao processo
CPU - registos da CPU
RAM - programas de MMU
entrada/saída - tabel de canais abertos

b) Em que ocasiões é que esta parte do descriptor é alterada? E quando é que é usada?

O estado de computação é selo guardado no descriptor quando
o processo perde o CPU (EXECUTING → READY; EXECUTING → WAITING)
Quando o processo volta a ganhar o CPU (READY → EXECUTING)
o conteúdo do descriptor do processo é copiado para o CPU
e MMU e a tabel de canais abertos para o estar active.

Questão 6 – 2,0 valores

Considere o algoritmo de escalonamento *MLFQ* (Multi-Level Feedback Queue). Explique como é que este algoritmo favorece os processos que têm apetência por operações de entrada / saída (*I/O bound*), ao mesmo tempo que garante que os processos com apetência pelo uso do CPU (*CPU bound*) têm a garantia de que não ficam muito tempo sem usar o CPU.

A prioridade de todos os processos é recalculada periodicamente e as subidas e descidas são decididas em base no tempo de CPU atribuída recentemente. Assim:

- os processos *I/O Bound* mantêm-se na prioridade alta, porque pouco consumo de CPU, estado futuro sempre no estado WAITING
- os processos *CPU Bound* vão subindo e sua prioridade é medida que o tempo vai passando sem eles serem atribuídos CPU.

Questão 7 – 2,0 valores

Explique porque é que para implementar corretamente a função `pthread_mutex_lock(...)` é preciso que o hardware suporte uma instrução do tipo *Test_And_Set*. Recorde que a execução desta instrução é equivalente à seguinte função C, executada de forma indivisível pelo hardware

```
int Test_And_Set( int *val){  
    int temp = *val; *val = 1; return temp;  
}
```

Se não houvesse *Test_And_Set* a entrada numa região crítica seria feita da seguinte forma

- ① while (*val == 1);
- ② *val = 1;
- ④

Se o processo que está a tentar entrar na região crítica perdesse o CPU em ④, ou se houve múltiplas CPUs poderia acontecer que mais do que 1 processo entrasse *val a 1 e desta forma, mais do que 1 processo entraria na região crítica

Questão 8 – 2,0 valores

Considere o código seguinte

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define N 1000000
int array[N];

void *worker(void *arg) {
    int i; int cont = 0;
    int n = (int)arg;
    for( i= n*N/2; i < (n+1)*(N/2) ; i++ )
        if(array[i] == 3) cont++;
    return (void *) cont;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2; int i, res1, res2;
    for(i = 0; i < N; i++) array[i] = random() % 4;
    pthread_create(&p1, NULL, worker, (void*)0); pthread_create(&p2, NULL, worker, (void*)1);
    pthread_join(p1, (void **) &res1); pthread_join(p2, (void **) &res2);
    printf("%d\n", res1+res2);
    return 0;
}
```

Explique o que faz o código acima. O resultado é sempre correto? Justifique a resposta.

O código acima usa dois threads para contar o nº de vezes com o valor 3 no vector 'array'.

O resultado é sempre correcto porque:

- o thread 0 teste os valores de array do índice 0 a 499.999 e
- o thread 1 teste os valores dos índices 500000 a 999999
- os contadores das variáveis locais dos threads e portanto não há acesso concorrente

Questão 9 – 2,5 valores

Considere o seguinte programa incompleto que usa a API dos Pthreads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
(0)
void *proc1(void *arg) {
    (1)
    // corpo da função proc1
    (2)
}
void *proc2(void *arg) {
    (3)
    // corpo da função proc2
    (4)
}
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    (5)
    pthread_create(&p1, NULL, worker, NULL); pthread_create(&p2, NULL, worker, NULL);
    pthread_join(p1, NULL); pthread_join(p2, NULL);
    return 0;
}
```

Pretende-se que o corpo da função `proc2()` seja executado apenas depois de todo o corpo de `proc1()` tenha sido executado.

a) Suponha que se pretende cumprir a especificação anterior usando variáveis condição. Diga que código colocaria nas posições (0) a (5)

```
(0) int done; pthread_cond_t c; pthread_mutex_t m;
(1) pthread_mutex_lock(&m);
(2) done = 1; pthread_cond_signal(&c); pthread_mutex_unlock(&m);
(3) pthread_mutex_lock(&m); if(!done) pthread_cond_wait(&c, &m);
(4) pthread_mutex_unlock(&m);
(5) done = 0; pthread_cond_init(&c, NULL); pthread_mutex_init(&m, NULL);
```

b) Repita a alínea anterior usando semáforos.

```
(0) sem_t s;
(1) —
(2) sem_post(&s);
(3) sem_wait(&s);
(4) —
(5) sem_init(&s, 0, 0);
```

Questão 10 – 1,5 valores

Considere um sistema operativo em que dois processos podem comunicar entre si usando as duas seguintes chamadas ao sistema:

```
int send ( int pid, char *msg, int nBytes);
```

```
int recv ( char *msg, int maxBytes);
```

a) Explique porque é que é necessária a intervenção do sistema operativo para que os dois processos possam comunicar entre si.

A operação `send()` implica escrever no espaço de endereçamento do processo receptor. É sabido que um processo não tem acesso ao espaço de endereçamento de outro, pelo que

b) Supondo que tem de pedir ao SO que o faça.

- a operação `send()` é assíncrona (isto é o processo emissor não se bloqueia quando envia a mensagem)
- e que a operação `receive()` é bloqueante (isto é, o processo que invoca a operação se bloqueia se não houver nenhuma mensagem disponível)

descreva a estrutura de dados que o sistema operativo tem de criar no descritor do processos para suportar estas chamadas ao sistema, e diga quais as ações efetuadas pelo sistema operativo quando um processo invoca a operação `send()` e a operação `receive()`.

Seria preciso criar 1 file de mensagens associada ao descritor de cada processo. Assim, as operações poderiam ser descritas de seguinte forma:

```
send ( P, msg ) {
```

- inserir msg à cauda de file de mensagens de P
- se P está bloqueado à espera de uma mensagem, desbloqueá-lo

```
}
```

```
msg receive () {
```

- Se não há nenhuma msg pendente bloquear o processo
- retornar a mensagem à cabeça de file

```
}
```

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)
```

Mutexes

Inicialização

```
int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *attr) ou  
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  
int pthread_mutex_lock (pthread_mutex_t *mutex)  
int pthread_mutex_unlock (pthread_mutex_t *mutex)
```

Condition Variables

Inicialização

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr) ou  
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;  
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)  
int pthread_cond_signal(pthread_cond_t *cond)
```

Semaphores

Inicialização

```
int sem_init( sem_t *sem, int type, int initial_value) // type is always 0 when using Pthreads  
int sem_wait( sem_t *sem )  
int sem_post( sem_t * sem )
```