GABARITO DA 2ª PROVA DE SISTEMAS OPERACIONAIS – 2010/1

Prof. José Gonçalves - Data: 10/06/2010

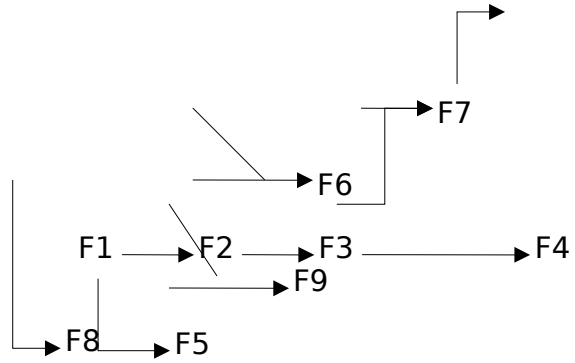
1. **(2,0)** Com relação aos sistemas *multithreading*, responda (justifique todas as suas respostas):

(a) Como as *user-level threads* e as *kernel-level threads* elas se relacionam com o algoritmo de escalonamento do sistema operacional?

User-level threads existem inteiramente dentro do espaço de usuário do processo e são escalonadas dentro da fatia de tempo deste, de acordo com o algoritmo implementado (oferecido) pela biblioteca de *threads*. O escalonador do S.O. simplesmente desconhece a existência delas. *Kernel-level threads*, ao contrário, são entidades do *kernel* sendo, portanto, conhecidas do sistema operacional. Elas constituem as unidades de escalonamento do algoritmo de escalonamento do S.O.

(b) Tarefas *CPU bound* e *I/O bound* são melhor processadas por *user-level threads* ou *kernel-level threads*?

Tarefas *CPU-bound* com computações interdependentes ou que trocam de *threads* com freqüência (ou seja, que demandam processamento *multithreading*), são melhor processadas por *user-level threads*. Como as *threads* de uma tarefa *CPU-bound* executam sem bloquear o processo, é vantajoso que a troca entre essas *threads* se dê no nível de usuário pois neste nível a troca de contexto é muito mais leve e rápida do que se envolvesse o *kernel*. Por sua vez, tarefas que têm múltiplas *threads* I/O *bound* são melhor processadas por *kernel-level threads*. Isso acontece porque elas se beneficiam dos *time slices* (quanta) adicionais que as *kernel-level threads* recebem quando a *thread* é bloqueada. Neste caso, o processo pode continuar o seu processamento com uma outra *kernel-level thread* mesmo que a primeira tenha sido bloqueada por ter feito I/O.



2. **(1,0)** Considere o seguinte grafo de precedência que será executado por três processos. Adicione semáforos a este programa (no máximo 6 semáforos), e as respectivas chamadas às suas operações, de modo que a precedência definida abaixo seja alcançada.

```
PROCESS A : begin F1 ; F2 ; F9 ; F4 ; end

PROCESS B : begin F3 ; F7 ; end

PROCESS C : begin F8 ; F5 ; F6 ; end
```

Solução não otimizada:

```
PROCESS A : begin F1 ; V(s1); F2 ; V(s2); V(s3); P(s4); F9 ; V(s3); P(s5); F4 ; end
```

```
PROCESS B : begin P(s2); F3 ; P(s3); P(s3); F7 ; V(s5); end

PROCESS C : begin P(s1); F8 ; F5 ; P(s3); V(s4); F6 ; V(s3); end
```

3. **(2,0)** Dado o problema a seguir, sincronize as ações dos processos Passageiro e Carrinho usando semáforos: "*Existem n passageiros, que repetidamente aguardam para entrar em um carrinho da montanha russa, fazem o passeio, e voltam a aguardar. Vários passageiros podem entrar no carrinho ao mesmo tempo, pois este tem várias portas. A montanha russa tem somente um carrinho, onde cabem C passageiros (C < n). O carrinho só começa seu percurso se estiver lotado*."

Uma possível solução é mostrada abaixo:

```
semaphore       passageiro = C
semaphore       carrinho = 0
semaphore andando = 0
semaphore mutex = 1
int       Npass = 0

Passageiro() {
  while (true) {
    DOWN(passageiro)
    entra_no_carrinho()      /* vários passageiros podem entrar "ao mesmo tempo" */
    DOWN(mutex)
    Npass++
    if (Npass == C) {        /* carrinho lotou */
         UP(carrinho)        /* autoriza carrinho a andar */
         DOWN(andando)       /* espera carrinho parar */
         UP(mutex)
    }
    else {
         UP(mutex)
         DOWN(andando)       /* espera carrinho lotar, passear e voltar */
    }
  }
}

Carrinho() {
  while (true){
    DOWN(carrinho)           /* espera autorização para andar */
    passeia()                /* faz o passeio e volta */
    Npass := 0               /* esvazia carrinho */
    for (int i=0; i<C; i++){
         UP(andando);      /* libera passageiro que andou de volta à fila */
         UP(passageiro);   /* libera entrada no carrinho */
    }
  }
}
```

4. **(1,5)** Escreva um monitor formado por dois procedimentos *request* e *release*, usados para gerenciar o uso de três unidades de um recurso, sendo que pode ser alocada somente uma unidade do recurso a cada *request*.

OBS: solução de referência, sem muitos controles. O objetivo é ver se o aluno sabe como estruturar um monitor (declarações de variáveis e inicializações) e como usar as operações sobre variáveis de condição.

```
monitor M() {
  int NunidadesDisponiveis;
  condition X;

  request() {
       if (NunidadesDisponiveis == 0) wait(X);
```

```
            NunidadesDisponiveis:= NunidadesDisponiveis – 1;
        }

        release() {
            NunidadesDisponiveis:= NunidadesDisponiveis + 1;
            if (NunidadesDisponiveis == 1 ) signal(X);
        }

        begin
             NunidadesDisponiveis:= 3;
        end
    }
```

5. **(1,5)** Usando-se a técnica de IPC *pipes*, pode-se implementar o comando "`ls | wc`". Resumidamente: *(i)* cria-se um *pipe*; *(ii)* executa-se um *fork*; *(iii)* o processo pai chama *exec* para executar "`ls`"; *(iv)* o processo filho chama *exec* para executar "`wc`". O problema é que normalmente o comando "`ls`" escreve na saída padrão 1 e "`wc`" lê da entrada padrão 0. Como então associar a saída padrão com a saída de um *pipe* e a entrada padrão com a entrada de um *pipe*? Isso pode ser conseguido através da chamada de sistema `int dup2(int oldfd, int newfd)`. Essa chamada cria uma cópia de um descritor de arquivo existente (`oldfd`) e fornece um novo descritor (`newfd`) tendo exatamente as mesmas características que aquele passado como argumento na chamada. A chamada `dup2` fecha antes `newfd` se ele já estiver aberto.

O programa em C a seguir implementa o comando "ls | wc", fazendo uso das seguintes chamadas de sistema: `pipe()`, `fork()`, `execlp()`, `dup2()` e `close()`. Complete as suas lacunas com os comandos apropriados.

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

int main(void)
{
   int pfd[2];                         /* declara o pipe */            0,2
   pipe(pfd);                          /* cria o pipe */
   if (fork() == 0) {                  /* Processo filho *)           0,1
     close(pfd[1]);                    /* fecha descriptor */          0,1
     dup2(pfd[0], 0);                  /* duplica descriptor */        0,4
     close(pfd[0]);                    /* fecha descriptor */          0,1
     execlp("wc", "wc", (char *) 0);   /* executa wc */
   } else {                            /* processo pai */


     close(pfd[0]);                    /* fecha descriptor */          0,1
     dup2(pfd[1], 1);                  /* duplica descriptor */        0,4
     close(pfd[1]);                    /* fecha descriptor */          0,1
     execlp("ls", "ls", (char *) 0);   /* executa ls */
   }
   exit(0);
}
```

6. **(1,0)** Considere as seguintes afirmativas em relação à técnica de IPC FIFO:

I. FIFO permite criar um canal de comunicação entre processos que não possuem ancestral comum.
II. FIFO cria um arquivo especial no sistema de arquivos, do tipo pipe, que é removido quando o processo que criou o FIFO é finalizado.
III. Diferentemente dos *pipes*, FIFOS podem servir a aplicações cliente-servidor numa mesma máquina.
IV. FIFOs também podem ser criados via *shell* com o comando $ mkfifo meuFIFO

( ) Todas estão corretas
(X) Existe apenas uma errada
( ) Todas estão erradas
( ) Existem apenas duas erradas

7. **(1,0)** Explique por que SVC's são necessárias para estabelecer uma memória compartilhada entre dois processos. Isso também é necessário entre múltiplas threads de um mesmo processo? Por que?

Processos distintos possuem espaços de endereçamento distintos, que não são acessíveis uns aos outros diretamente. Para que algum segmento de memória seja visível a dois processos, o *kernel* do sistema operacional deve ser acionado já que é ele o responsável por prover o serviço de compartilhamento de memória entre processos. Como a SVC constitui a interface para a solicitação de serviços ao S.O., a invocação para a criação de um segmento de memória compartilhada entre processos tem que ser feita via SVC.

No caso de múltiplas *threads* em um processo isso não é necessário, já que *threads* de um mesmo processo, por definição, compartilham o mesmo espaço de endereçamento, não havendo a necessidade de solicitar este serviço ao sistema.