SISTEMAS OPERACIONAIS

Profª. Roberta Lima Gomes (soufes@gmail.com)

2ª. Lista de Exercícios

Data de Entrega: não há*. O objetivo da lista é ajudar no estudo individual dos alunos.*

SINAIS

1. Qual a diferença entre bloquear um sinal e ignorar um sinal?

2. Explique a razão pela qual um tratador de sinais pode encontrar estruturas de dados inconsistentes. De que forma o programador pode evitar essas situação?

3. Explique o que faz o código a seguir:

```
/* signal_hand */
void signal_hand( int arg ) {
  int ch;
  printf("Tem certeza que quer Finalizar [s/n] ? ");
  ch = getchar();
  if( ch == 's' || ch == 'S' )  exit( 0 );
}

/*Main Program */
int main( int argc, char* argv[] )
{
  int i;
  printf("Processo: ... processando ...\n");
  signal( SIGINT, signal_hand );
  for( ; ; ) ;
  printf("Processo: ... processando ...\n");
  exit(0);
}
```

4. Considere o seguinte programa:

```
...

void main(){
  pid_t pid;
  int i;
  pid = fork();
  if (pid == 0)
    for(i=0; i<5; i++){
      kill(getppid(), SIGUSR1);
      sleep(2);
    }
  elseif (pid >0){
    signal(SIGUSR1,funcao);
    for(;;)
      pause();
  }
}

void funcao(int s){
      execlp("prog", "prog", 0);
}
```

Quantas vezes é executado o programa **prog**? Justifique.

5. Implemente um programa (código ou pseudo-código) que cria um novo processo e sincroniza o acesso ao arquivo "dados.txt" por meio do uso de sinais, da seguinte forma:

   O processo pai fica à espera (não espera ocupada!) até que chegue um sinal SIGUSR1; depois de receber o sinal, deve ler do arquivo "dados.txt" um número; depois de lido esse número, deve remover o conteúdo desse arquivo e apresentar esse conteúdo no monitor; por fim envia ao filho um sinal SIGUSR1. O processo filho é responsável por escrever um novo número no arquivo.

EXCLUSÃO MÚTUA / SEMÁFOROS / MONITORES

6. Qual o problema com a solução que desabilita as interrupções para implementar a exclusão mútua?

7. O que é espera ocupada (*busy wait*) e qual o seu problema?

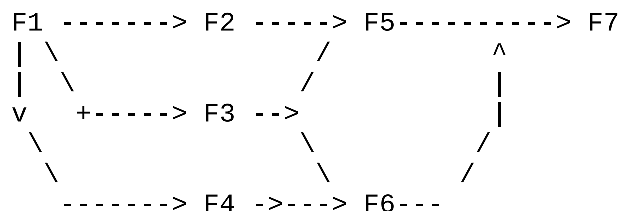
8. Em muitos sistemas operacionais existe uma chamada de sistema "yield()", cujo efeito é passar o processo (ou thread) chamador para o final da fila de aptos (prontos). Suponha que um dado kernel escalone o processador utilizando fatias de tempo e suporte a chamada "yield()". Suponha ainda que, em todas as aplicações, o tempo necessário para executar uma seção (região) crítica seja menor que a duração de uma fatia de tempo. Para este caso em particular, buscando resolver o problema da seção crítica, considere o seguinte mecanismo de sincronização:

   Imediatamente antes de entrar na seção crítica, a thread chama "yield()";
   Ao sair da seção crítica, a thread não faz nada.

   Responda as questões abaixo, sempre justificando sua resposta.

   (a) Esta solução oferece exclusão mútua?
   (b) Esta solução apresenta a possibilidade de postergação indefinida?
   (c) Esta solução funciona para mais de duas threads?
   (d) Esta solução apresenta *busy waiting*?

9. Considere o seguinte grafo de precedência:



```
F1 -------> F2 -----> F5----------> F7
| \                  /           ^
|  \               /             |
v   +-----> F3 -->               |
 \                \             /
  \                \           /
   -------> F4 ->---> F6---
```

que será executado por três processos, conforme código a seguir:

```
cobegin
  P1: begin F1; F3; end;
```

```
      P2: begin F2; F5; F7; end;
      P3: begin F4; F6; end;
    coend
```

Adicione semáforos a este programa, e as respectivas chamadas às suas operações, de modo que a precedência definida acima seja alcançada.

10. Considere o problema dos leitores e escritores, onde existem diversos processos que eventualmente fazem acessos de leitura a uma base de dados e diversos processos que eventualmente fazem acessos de escrita à mesma base. Vários acessos de leitura podem ocorrer simultaneamente, mas um acesso de escrita não pode ocorrer simultaneamente com acessos de nenhum tipo. Considere o código a seguir para os processos de leitura e de escrita. Suponha que todos os semáforos são iniciados com valor 1:

```
  leitor:
   ...
1 while(1) {
2   P(R);
3   P(M);
4   rc++;
5   If (rc==1) P(W);
6   V(M);
7   V(R);
8   LÊ;
9   P(M);
10  rc--;
11  if (rc==0) V(W);
12  V(M);
13  ... consome
```

```
   escritor:
   ...
1 while(1) {
2  ...produz
3   P(R);
4   P(W);
5   ESCREVE;
6   V(W);
7   V(R);
8 }
```

(a) Essa solução pode levar a *starvation* de escritores? (pode acontecer de um escritor nunca conseguir o acesso à base devido à seguida chegada de novos leitores?) Explique sua resposta, usando os números das linhas de código para se referir aos passos do programa.
(b) Explique o papel do semáforo M. Dê um exemplo de problema que poderia ocorrer caso as operações sobre ele fossem retiradas.

11. Um *semáforo múltiplo* estende semáforos de maneira a permitir que as primitivas *up* e *down* operem em vários semáforos simultaneamente. Isto é útil para requisitar e liberar vários recursos por meio de uma operação atômica. Como você implementaria tais primitivas usando os semáforos estudados no curso? (Observe que você, como implementador da primitiva, pode acessar o valor do contador associado ao semáforo diretamente. Ou seja, você tem disponível, na hora de implementar Down (R1, R2, ..., Rn) ou Up (R1, R2, ..., Rn), uma rotina “valor_do_contador (R)”, que devolve o valor do contador associado a um semáforo).

12. Resolva usando semáforos: “Três fumantes se encontram em uma sala com um vendedor de suprimentos para fumantes. Para preparar e usar um cigarro, cada fumante precisa de três ingredientes: tabaco, papel e fósforo, coisas que o vendedor tem à vontade no estoque. Um fumante tem o seu próprio tabaco, o segundo tem seu próprio papel, e o outro tem seu próprio fósforo. A ação se inicia quando o vendedor coloca à venda dois ingredientes na mesa, de forma a permitir que um dos fumantes execute esta prática dita como não muito saudável. Quando o tal fumante termina, ele acorda o vendedor, que escolhe então outros dois ingredientes (aleatoriamente) e coloca a venda, portanto desbloqueando outro fumante.” *O Ministério da Saúde adverte: Fumar faz mal à Saúde!*

13. Considere a seguinte modelagem, por monitor, para o problema do produtor-consumidor com *n* produtores e *m* consumidores.

```
monitor buffer {
  int itens=0; cond temItens, temEspacos;
  ...
  int pega() {
    while (1) {
        if (!itens) wait(temItens);
        pega item no buffer
        itens--;
        signal(temEspacos);
        return (item);
    }
  }

void coloca() {
    while (1) {
        if (itens==MAX) wait(temEspacos);
        coloca item no buffer
        itens++;
        signal(temItens);
    }
  }
}
```

Suponha que esse monitor funcione com a disciplina “sinaliza e continua", dada em sala (mas observe que ele admite filas separadas por variáveis de condição). Explique por que essa solução não funciona adequadamente.

14. Um porto marítimo de uma cidade pode receber no máximo N navios que entram através de um canal. Pelo canal pode transitar apenas um navio de cada vez. Os navios que saem do porto têm prioridade na travessia do canal. Cada navio é simulado por uma tarefa cujo pseudo-código é o seguinte:

```
thr_navio() {
    < Chegada à entrada do canal >
    pedido_entrada();
    < Atravessa o canal >
     entrada_OK();
    < Atraca no porto >
     pedido_saida();
    < Atravessa o canal >
     saida_OK();
     < Parte em viagem para outro porto >
}
```

Utilizando monitores, implemente em pseudo-código as funções pedido_entrada(), entrada_OK(), pedido_saida() e saida_OK().

15. Utilizando as primitivas de semáforos, explique como implementar um monitor.

~~16. Em um sistema que suporta programação concorrente apenas por meio da troca de mensagens, será criado um Servidor para controlar o uso das portas seriais. Quando um processo Cliente deseja usar uma porta serial, ele envia uma mensagem “Aloca” para o Servidor. Existem N portas seriais, todas equivalentes, mas cada uma pode ser usada somente~~

~~por um Cliente de cada vez. O Servidor informa ao Cliente a porta que ele vai usar por meio da mensagem “Porta p”. Ao concluir o uso, o Cliente envia para o Servidor a mensagem “Libera p”. Suponha que existam mais do que N processos Clientes. Mostre o algoritmo do Servidor, em português estruturado. Supor “receive” bloqueante.~~

17. ~~Utilizando troca de mensagens, explique como implementar um semáforo em um sistema distribuído.~~

IPC

18. Qual a diferença entre pipes e named pipes (fifos)?

19. Sobre o código a seguir, responda:

```
#define SIZE 6
#define READ 0
#define WRITE 1
main()
{
    pid_t pid1, pid2, pid3, pid4; int status;
    int fd[2]; char buffer[SIZE+1];
    struct rusage usage;
    pipe(fd);
    if ((pid1=fork())==0) {              // child 1
        while(1) {
         read(fd[READ], buffer, SIZE);
         buffer[SIZE]='\0';
         write(fd[WRITE], "tomato", SIZE);
      }
    } else if ((pid2=fork())==0) {     // child 2
        while(1) {
         read(fd[READ], buffer, 6);
         buffer[6]='\0';
         write(fd[WRITE], "turnip", 6);
      }
    } else {  // parent
      write(fd[WRITE],"potato",6);
      fprintf(stderr,"Parent: I wrote a potato!\n", buffer);
      sleep(3);
      read(fd[READ], buffer, 6);  buffer[6]='\0';
      fprintf(stderr,"Parent: I got back a %s!\n", buffer);
      kill(pid1, SIGINT); pid3 = wait(&status, 0, &usage);
      kill(pid2, SIGINT); pid4 = wait(&status, 0, &usage);
    }
}
```

a) O que este código faz?
b) É possível prever qual será a saída do programa?
c) O que acontece se omitirmos o comando write do pai?
d) O que acontece se omitirmos o comando write do um dos filhos?

20. ~~Como são definidas e usadas áreas de memórias compartilhadas entre processos no UNIX.~~

21. Faça uma comparação entre os diversos mecanismos disponíveis para comunicação e sincronização entre processos executados num mesmo computador.

22. Joãozinho deseja aprender como utilizar pipes no UNIX, então ele escreve um pequeno programa teste:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
void systry(int v, char* msg){
   if (-1!=v) { fprintf(stderr, msg); return; }
   perror(msg);
   exit(1);
}
int main(int argc, char* argv[]) {
   int p[2];
   systry(pipe(p), "One...");
   systry(pipe(p), "Two...");
   systry(pipe(p), "Three!\n");
}
```

Para seu espanto, uma execução desse programa produziu a seguinte saída:

```
One...Two...: Too many open files
```

Após algumas investigações, ele encontra as seguintes mensagens de erro no *manual page* do pipe(2):

```
ERRORS
    EMFILE  Too  many  file descriptors are in use by the process.
```

No entanto, ele tem certeza que na versão de unix que ele está rodando, cada processo pode abrir até 1024 arquivos. Como esse erro poderia ter ocorrido?

THREADS

23. Explique a diferença entre unidade de alocação de recursos e unidade de escalonamento?

24. Quais as vantagens e desvantagens do compartilhamento do espaço de endereçamento entre threads de um mesmo processo?

25. Quais as diferenças entre threads em nível de usuário e threads em nível de kernel? Em quais circunstâncias um tipo é melhor do que o outro?

26. Quais estruturas são utilizadas para a criação de uma thread de kernel? Como elas se diferenciam daquelas utilizadas para a criação de um processo?

27. Como o uso de threads pode melhorar o desempenho de aplicações paralelas em ambientes com múltiplos processadores?

28. Usando os mecanismos "mutex" e "variáveis condição" do Posix, implemente as operações P e V de um semáforo. Cada uma dessas operações deverá ser substituída por um código C com semântica similar. Lembre-se que estas operações devem ser atômicas. O valor inicial do semáforo é 1.

    Dicas sobre o Posix:

    pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&m);
pthread_mutex_unlock(&m);
pthread_cond_t vc = PTHREAD_COND_INITIALIZER;
pthread_cond_wait(&vc , &m);
pthread_cond_signal(&vc);

29. Em Java, tipicamente quando uma thread realiza um “wait()”, isto é feito dentro de um loop “while()”. Explique por quê.

30. Considere a seguinte classe em Java que implementa a estrutura de dados pilha com as operações básicas **push**, **pop**.

```
01. public class SimpleStack<E> {
02.
03.   private Object stack[];
04.   private int top;
05.   private int size;
06.
07.   public SimpleStack() {
08.     this(100);
09.   }
10.
11.   public SimpleStack(int size) {
12.     this.stack = new Object [size];
13.     this.top = 0;
14.     this.size = size;
15.   }
16.
17.   public boolean empty() {
18.     return (top == 0);
19.   }
20.
21. public boolean full() {
22.     return (top == size);
23.   }
24.
25.  public boolean push(E element) {
26.     if (top == size)
27.       return false;
28.
29.     stack[top++] = element;
30.     return true;
31.   }
32.
33.   @SuppressWarnings("unchecked")
34.   public E pop() {
35.     if (top == 0)
36.       return null;
37.
38.     return (E) stack[--top];
39.   }
40. }
```

Uma aplicação possui múltiplas threads que podem acessar essa estrutura de dados simultaneamente. Considerando isso, reimplemente a classe usando a sincronização via monitores.