Lista Exercícios 1
MC504A - Sistemas Operacionais
2s 2023
Prof. Carlos A. Astudillo

A continuação apresenta um conjunto de problemas que lhe serão de ajuda para exercitar-se e estudar a matéria. Estes problemas **não têm nota** associada. Porém, se recomenda fortemente resolvê-los.

Introdução a SO

1. **O que é um sistema operacional (SO)?**

   O sistema operacional é a primeira camada de software acima do hardware de uma máquina, ele possui a função de gerenciar recursos e permitir uma abstração das partes do sistema para fornecer uma prática mais simples de utilização para o usuário. O sistema operacional atua como “árbitro”, administrando a divisão de recursos como memória e processamento, entre as partes que integram o sistema. Atua também como uma espécie de “ilusionista”, pois cria a ilusão de acesso ilimitado de recursos, como na geração de memória virtual por exemplo. O S.O também fornece uma API de serviços comuns para que haja simplificação de design e abstração dos diferentes tipos específicos de serviço utilizados, por cima de uma interface genérica.

2. **Qual a diferença entre monoprogramação e multiprogramação. Exemplifique.**

   Monoprogramação se refere a prática em que apenas um processo é executado por vez no sistema, já a multiprogramação introduz o paralelismo, no qual há a divisão da memória principal entre diferentes processos e eles podem ser executados simultaneamente. Nos dias atuais a multiprogramação se tornou essencial e é predominante na maioria dos sistemas computacionais. Um exemplo de monoprogramação foi o sistema de processamento em lotes, utilizado por máquinas antigas para processar filas de jobs. Já um exemplo de sistemas em que é empregado a multiprogramação são os atuais sistemas computacionais, nos quais as máquinas possuem vários núcleos de processamento e existem diversas técnicas que empregam o paralelismo de execução.

3. **O que é um processo? Dê exemplos.**

   Um processo pode ser definido como uma tarefa executada pelo sistema, cada processo é uma instância de um programa, cada um com seu contexto (registradores,

variáveis, etc) e com sua CPU virtual e endereçamento de memória. A alternância e gerenciamento de processos são atividades realizadas pelo sistema operacional. Um exemplo de processo pode ser dado por um web browser em execução na máquina.

4. **Defina as propriedades essenciais dos seguintes tipos de SO: Batch, Multiprogramados, Tempo Compartilhado, Tempo Real e Multiprocessados.**

Os sistemas operacionais em lote realizavam o processamento de jobs sequencialmente, lendo os jobs que eram programados em Assembly ou Fortran no papel e passados para cartões perfurados, que por sua vez eram processados em fitas magnéticas, nesse tipo de sistema apenas um job poderia ser processador por vez, o que tornava a CPU ociosa caso fosse necessário uma parada para realizar operações como de entrada e saída por exemplo. Já nos sistemas multiprogramados, houve a inserção da multiprogramação, na qual a memória era dividida em vários jobs e a CPU não precisava ficar ociosa no momento que um processo fosse interrompido. Os sistemas de tempo compartilhado trouxeram a inovação da divisão de utilização da CPU em turnos para os diferentes usuários no sistema, mantendo a eficiência da multiprogramação. Sistemas operacionais de tempo real são implementados com a finalidade de reagir o mais rapidamente possível a situações específicas, já que, são sistemas projetados para atuar em sistemas críticos, como em equipamentos hospitalares e aeronaves, esses sistemas possuem alto paralelismo e alta priorização de certas atividades. Os sistemas operacionais multiprocessados são implementados para lidar com máquinas que possuem múltiplos processadores, aplicando as funções básicas de um S.O em máquinas com mais de um processador físico.

5. **O que são chamadas de sistema?**

Chamadas de sistema são instruções que permitem a comunicação entre os programas e o sistema operacional. Ao utilizar uma chamada de sistema, a execução de um programa na CPU é pausada e o controle é transferido para o sistema operacional, que irá lidar com a chamada. Chamadas de sistema podem realizar acesso ao kernel e a outros componentes privilegiados do S.O. Como as chamadas de sistema originais são escritas em assembly e variam conforme hardware específico, os sistemas operacionais possuem bibliotecas de funções que permitem realizar chamadas de sistema em linguagem de programação de alto nível, um exemplo são as funções definidas do padrão POSIX (Portable Operating System Interface), utilizadas em sistemas baseados em Unix.

6. **Explique as principais funções de um S.O. De um exemplo de um S.O como Árbitro, Ilusionista e Cola.**

O sistema operacional pode atuar como “árbitro”, mediando a divisão de recursos como tempo de CPU e memória entre as partes que integram o sistema(alocação de recursos, isolamento de falhas e comunicação inter-processos). O S.O como “ilusionista” tem a função de criar uma ilusão de acesso total ao hardware pelos processos, utilizando

estratégias de software como a geração de uma memória e CPU virtual(virtualização). O sistema operacional como “cola” se refere a função do S.O de gerar uma API através de uma interface genérica, para abstrair os detalhes dos diferentes tipos de serviço que implementam o sistema. Outras dois pontos importantes que devem compor um S.O são, confiabilidade e disponibilidade: se refere a capacidade do sistema de ser confiável, o quanto ele é imune a falhas e a sua utilização pode ser realizada sem temer problemas, já a disponibilidade se refere a qual a qualidade de operação do sistema, ele está disponível na maior parte do tempo ou possui tantas falhas que impedem a utilização? Um sistema deve ser confiável e disponível. Por outro lado pode possuir alta disponibilidade (é acessível na maior parte do tempo), mas não ser confiável, nenhum dos dois ou vice-versa.

Disponibilidade: MTTF(mean time to failure) - frequência com que o sistema apresenta falhas

MTTR(mean time to repair) - tempo médio necessário para reparação de falhas

outros aspectos importantes de sistemas operacionais: segurança e privacidade, portabilidade, desempenho e adoção.

7. **Se o computador possuir apenas um processador (principal), é possível um processamento paralelo ocorrer? Justifique a resposta.**

Não, não é possível realizar processamento paralelo real com apenas um processador que não possua múltiplos núcleos de processamento, para isso é necessário que seja empregado hardware específico que implemente o paralelismo. O sistema operacional pode empregar estratégias para criar a ilusão de paralelismo, como o uso de multithreads sobre processos e técnicas de escalonamento.

Stack

8. **Dado o seguinte código:**

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    float temperature, pressure;
6    cout << gay-lussac(temperature, pressure);
7  }
8
9  float gay-lussac(float t, float p){
10   float k = p/t;
11   return k;
12 }
```

a. Desenhe o stack antes da execução da linha 6.

Antes da linha 6, a pilha possui salvo apenas o endereço de retorno da função main(), o registrador base pointer(%ebp) que salva o endereço base da função que está sendo executada e as variáveis locais temperature e pressure.

ret. main
%ebp (base pointer)
pressure
temperature

main()

b. Desenhe o stack quando encontra-se executando a função gay-lussac (linhas 9 a 12).

Ao acessar a função gay-lussac(), o ret main, o registrador %ebp e as variáveis locais de main() permanecem na pilha. São incluídos os argumentos da função gay-lussac(), o endereço de retorno, o novo %ebp atualizado e a variável local da função.

ret. main
%ebp (base pointer)
pressure
temperature
t
p
ret. gay-lussac
%ebp
k

main()	gay-lussac()

c. Como é o stack da linha 5 a respeito do stack da execução da linha 6?

Neste ponto, como ainda não houve acesso a função gay-lussac(), o stack na linha 5 é idêntico ao stack na linha 6.

9. Dado o seguinte código:

```
1  #include <stdio.h>
2
3  int main(char C, int O){
4    int H;
5    scanf("\%d", &H);
6    if(H >= 2)
7      Wa(H,O);
8    else
9      Ca();
10 }
11
12 char Wa(int H, int O){
13   int Water=H+2+O;
14   return Ca();
15 }
16
17 char Ca(){
18   bool V=TRUE;
19   return 'd';
20 }
```

a. Desenhe o stack quando encontra-se executando a chamada da linha 7.

O
C
ret. main
%ebp (base pointer)
H
O
H
ret. Wa
%ebp(base pointer)
Water
ret. Ca

%ebp(base pointer)
V

main()	Wa()	Ca()

b. Desenhe o stack quando encontra-se executando a chamada da linha 9.

O
C
ret. main
%ebp (base pointer)
H
ret. Ca
%ebp (base pointer)
V

main()	Ca()

c. O que acontece no stack quando se executa o condicional (linha 6 e linha 8)?

O
C
ret. main
%ebp (base pointer)
H

main()

10. Um processo P1 se executa até que se desencadeia uma interrupção do temporizador. Então, o kernel toma o controle do processador e decide executar o processo P2.

a. Explique quais passos são feitos para realizar a transferência, causada pela interrupção, de um processo geral desde modo usuário ao modo kernel.

Existem três razões para o kernel assumir o controle de um processo que se executa em modo de usuário, interrupções, exceções do processador e chamadas de sistema. Uma interrupção é um sinal assíncrono que solicita uma parada do processador para verificação de algum evento externo, interrompendo a busca, decodificação e execução de instruções. O processador salva o estado de execução atual e passa a executar um manipulador de interrupção que é especificado para cada tipo diferente de interrupção. As interrupções são amplamente utilizadas para informar o kernel do término de solicitações de I/O. Ao fim da manipulação de interrupção, o processador deve retomar o estado salvo exatamente do mesmo ponto em que estava quando a interrupção foi solicitada, restaurando o contador de programa, os registradores e devolvendo o modo para o usuário.

b. Desenhe o diagrama de estados, em geral, de um processo.

Executando: o processo está fazendo uso da CPU em execução.

Pronto: o processo está aguardando na pilha de processos prontos para serem executados, até ser escalonado.

Bloqueado: o processo está com execução bloqueada por algum motivo como: aguardando I/O, dormindo, etc. Ao ser desbloqueado, o processo pula ao estado Pronto.

c. Explique o que acontece na mudança de contexto entre o processo P1 e P2.

Existe uma sequência de passos comum à mudança de contexto. Quando a parada ocorre por interrupção, exceção ou chamada de sistema, o contexto do processo P1 é salvo em uma estrutura localizada em parte da memória do kernel, chamada de pilha de interrupção, o registrador altera o ponteiro da pilha do sistema para o endereço base dessa região, e então, os registros do processo P1 são salvos

nessa pilha. Após isso o controle é passado para o kernel, e em caso de interrupção ou exceção, aciona a tabela de vetores de interrupções do sistema operacional e o manipulador de interrupção responsável para tratar a interrupção executa o procedimento necessário. Cada processo executando em modo de usuário possui sua própria pilha de interrupção em modo de kernel, um ponteiro para essa pilha é armazenado no PCB do processo, isso possibilita também a troca entre processos por interrupção do timer, repetindo os passos descritos para salvamento na pilha do kernel, mas sem acessar a tabela de vetores de interrupção. Após isso o controle da CPU é passado para um processo P2 e seu contexto salvo em sua pilha do kernel é restaurado.

d. Indique o que acontece com os *PCBs* de cada processo durante a mudança de contexto.

Para o processo que está finalizando, ocorre a atualização das informações do processo que estão presentes no PCB, como mudança do estado de executando para pronto ou bloqueado e atualização do ponteiro que aponta para a stack do kernel do processo. Para o processo que irá iniciar execução, ocorre mudança do estado de pronto para executando e o ponteiro para o stack do kernel é utilizado para restaurar o contexto salvo do processo.

e. O que contém o stack do kernel de cada processo ao finalizar a mudança de contexto?

O stack do kernel armazena o contexto atual do processo, salvando todos os elementos importantes do contexto como registradores de CPU, contador de programa, stack pointer, a pilha de modo de usuário e etc.

11. Considere o seguinte código. Suponha que as funções foo e bar são definidas pelo usuário, e que a função SysCall está definida dentro do kernel (com prototipagem dentro de system.h mas definida aqui, convenientemente).

a. A partir do código, desenhe o stack do processo e o stack do kernel quando se executa a chamada na linha 19.

Stack do processo em ambiente do usuário:

Q
ret. main
%ebp(base pointer)
Arr[0]
Arr[1]
Arr[2]
pop
Zom
Que
ret. foo
%ebp (base pointer)
holl
Que
Hip
ret. bar
%ebp (base pointer)
Az
Jack

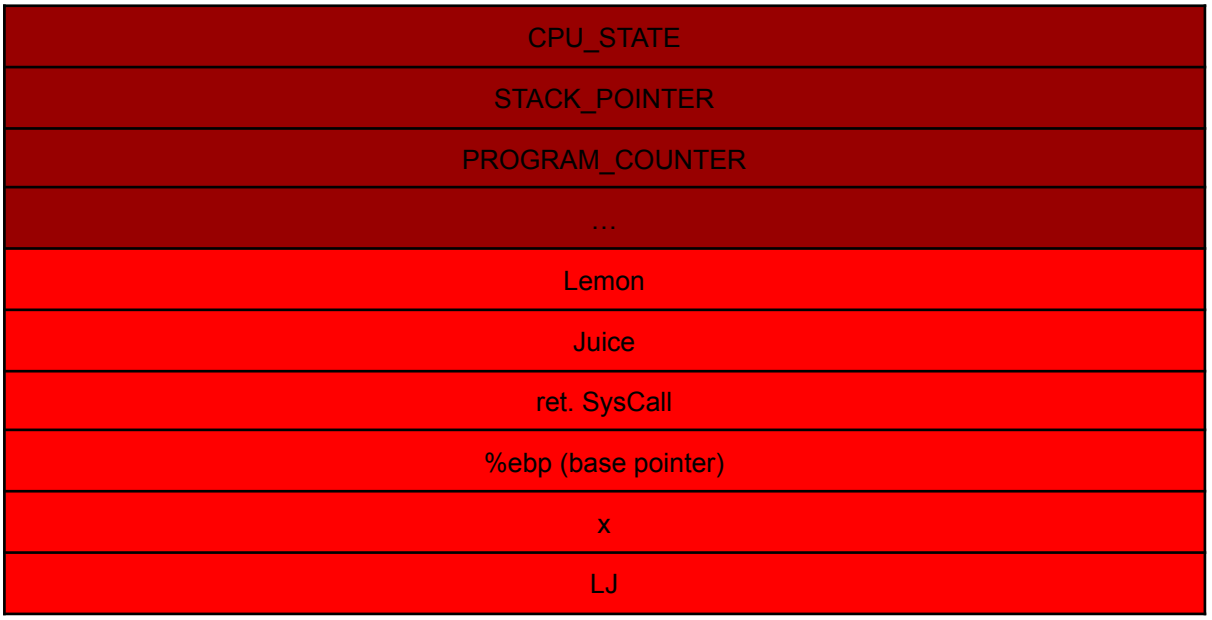
Stack do processo em ambiente de kernel:

-> vazio

b. A partir do código, desenhe o stack do processo e o stack do kernel depois da chamada na linha 20.

Stack do processo em ambiente de usuário:

Stack do processo em ambiente de kernel:



c. Explique a mudança de modo usuário a modo kernel e vice-versa provocado pela chamada ao sistema SysCall.

10 c)

```c
#include <system.h>
#include <stdio.h>

char foo(float Que, char Zom){
  char holl='a';
  bar(Que+4,holl);
  return Zom;
}

int bar(float Hip, char Oue){
  float Az=50.5 ;
  bool Jack=False ;
  return Az+Hip;
}

int main(float Q){
  float Arr[3] = { 5, 15, 3};
  char pop='p';
  foo(Arr[1], pop);
  SysCall(Arr[2], Arr[0]);
}

bool SysCall(float Lemon, float Juice){
  int x;
  float LJ=0.1;
  for (x=0; x<3; <++)
      LJ=Lemon+Juice;
  bar(LJ, 'D');
  return true;
}
```

Processos

12. **Defina três tipos de execução ou ambientes num computador que gerem uma mudança de modo usuário a modo kernel.**

    Mudanças de modo de usuário a modo kernel ocorrem quando ocorre uma interrupção de sistema. Uma interrupção ocorre quando existe um sinal de que é necessário a parada da execução para que algo seja tratado. Exceções do processador: a verificação de exceções de processador causam uma interrupção, elas ocorrem quando o processador detecta algum erro no sistema, como por exemplo, tentativa de acesso a região não autorizada de memória, divisão por zero, tentativa de gravação em modo read only, etc. Chamadas de sistema: chamadas de sistema são utilizadas para realizar comunicação entre o usuário e kernel, quando o usuário necessita de realizar uma tarefa de kernel, uma chamada de sistema é acionada e o contexto é passado para que o kernel manipule a chamada.

13. **Considere o seguinte código que está definido em ded.c.**

```c
// Definição de ded.c

#include <fildes.h>
#include <stdio.h>

int main(int argc, char** argv){

  // Se abre um arquivo para ler
  int fd = open ("sample.dat", O_RDONLY);
  char buff[20];
  ssize_t bytes_read = read(fd, buff, sizeof(buf));

  // Se gera um char fora do espaço de memoria legal.
  char* erro = (char*) 0x1000;
  *erro = 13;

  int pv = close(fd);
}
```

a. Explique em detalhe os passos que realiza o sistema operacional para ler os dados do arquivo fd e armazená-los em buff (ver linha 11). (Deve ao menos descrever as mudanças de modo, os buffer que se utilizam para transferir a informação, e possíveis chamadas ao sistema que se desencadeiam.)

Na linha 9 ocorre a chamada de sistema open(), ela recebe uma string como caminho de um arquivo a ser aberto, e, uma flag de inteiro que marca em qual modo será aberto, no caso em modo de apenas leitura. Após abrir o arquivo, é atribuído a ele pelo sistema operacional, um descritor de arquivo, que é um número inteiro que representará exclusivamente o arquivo aberto. Na linha 11 ocorre uma nova chamada de sistema, pela função read(), ela recebe o descritor de arquivo a ser aberto, um buffer de memória que armazenará os dados lidos e o tamanho em bytes que se deseja ler do arquivo. O sistema operacional acessa o arquivo e começa a ler o arquivo de um ponteiro de leitura, esses dados lidos são armazenados na posição de memória apontada pelo buffer. Após a leitura, o ponteiro de leitura do arquivo é atualizado para a região seguinte a última lida, permitindo continuar desse ponto em leituras seguintes. A função read retorna o número de bytes lidos em caso de sucesso, ou, -1 em caso de erro, pode se verificar se a leitura foi bem sucedida utilizando esses parâmetros. Na linha 11 é executada outra chamada pela função close(), ela libera a região de memória utilizada pelo arquivo, tornando inacessível para realização de leitura e escrita.

b. A linha 14 resultará numa violação de segmento de memória, explique o que acontece no sistema operacional em este momento.

Ao tentar acessar uma área de memória não possível, ocorre uma falha de segmentação. O sistema operacional gera um sinal de interrupção por exceção do processador, em sistemas baseados em Unix, esse sinal é chamado de SIGSEGV (Signal Segmentation Violation). Após detectar a falha, o sistema trata a exceção, podendo inclusive encerrar o processo se necessário.

14. **Dado o seguinte código**

```
1  void main(int argc, char** argv){
2    int child == fork ();
3    int x = 5;
4    if (child == 0) {
5      x += 5;
6    } else {
7      child = fork();
8      x += 10;
9      if (child) {
10       x +=5;
11     }
12   }
13 }
```

a. Indique quantos processos se criaram ao momento de terminar de executar o if (linha 12).

Ao passar pelo fork() na linha 2, é criado um novo processo que copia todos os dados do processo pai atual, porém obtém um PID diferente. A variável child do processo pai recebe o PID do processo filho, enquanto a mesma variável no processo filho recebe o valor 0. No processo filho o código se executa até o if da linha 4, terminando com 2 processos no total. Já no processo pai, é criado um novo processo filho na linha 7, o processo pai recebe o PID desse processo e o filho recebe 0, ao fim da execução 3 processos foram criados no total.

b. Gere um rastreamento dos processos anteriores e os valores que adquire a variável x ao longo da execução do código.

No primeiro processo filho que se executa até o if da linha 4, x vale 5 e é incrementado em 5, terminando com o valor 10. O segundo processo filho começa sua execução no fork() da linha 7, x vale 5 e é incrementado em 10, como o valor de child pro filho é 0, a execução é encerrada e x vale 15. Já o processo pai original, realiza a incrementação de 10 em x e por fim entra no if da linha 9, já que child vale o PID do último processo filho criado, portanto x termina valendo 20.

15. **Dado o seguinte código, suponha que a função getnewprocess(char**) popula o vetor que recebe por parâmetro com um novo processo e parâmetros necessários (o primeiro elemento é o nome do programa e os subsequentes são os parâmetros deste).**

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

int cont=0;
void forkthem(int, char**);

int main (int argc, char** argv){
  forkthem (4, argv);
}

void forkthem(int n, char** argv){
  if (n > 0){
    int child = fork();
    n = n-1;
    if(n%2 == 0 && child == 0){
      // getnewproces enche argv com um novo processo e argumentos
      getnewproces(argv);
      if (execvp(*argv, argv) < 0) {
        printf("*** ERROR: exec failed\n");
        exit(1);
      }
    }
    forkthem (n);
  }
}
```

a. Explique quais são as funções de fork() e execve(), e explique o que fazem passo a passo para criar um novo processo.

A função execve implementa a chamada de sistema execve, essa chamada gera o encerramento do processo atual e a substituição deste por um novo processo totalmente diferente, os parâmetros desse novo processo são passados por argumento na função execve() e sua execução ocorre a partir do início do programa. Já a função fork() implementa a chamada de sistema fork(), que cria uma cópia exatamente igual do processo atual, exceto o PID, copiando dados da memória, registradores e etc. O processo pai que chama fork() recebe como retorno da função o valor do PID do processo filho, enquanto o filho recebe o valor 0, caso seja retornado o valor -1, houve falha na criação do processo. No caso do fork(), a execução em ambos processos, pai e filho, continua a partir do ponto em que foi realizada a chamada de sistema.

b. Indique quantos processos são gerados ao finalizar a linha 8, e explique por que chegou a esta conclusão.

Ao passar pela linha 8 é executada a função forkthem() com os argumentos argv** e n = 4, na linha 13 um processo filho é criado através de um fork(), ocorre a divisão de execução entre processo pai e processo filho:

processo filho 1: ao se verificar a condicional da linha 18, PID = 0, n = 3, a condicional não é satisfeita, ocorre a chamada recursiva. Outro processo filho é criado:

processo filho 1.1: PID = 0, n = 2, acessa a condicional. getnewprocess() é executado, e se execvp() for bem sucedido, o processo filho 1.1 é substituido. Ocorre a chamada recursiva. Um novo processo é criado.

processo filho 1.1.1: PID = 0, n = 1, não acessa a condicional, feita a chamada recursiva, novo processo criado.

processo filho 1.1.1.1 PID = 0, n = 0, acessa a condicional e repete os passos de processo filho 1.1. Fim da primeira parte da recursão,

n é decrementado, o if da linha 15 não é satisfeito pois n = 3 é ímpar, é feita a chamada recursiva. Novamente um novo processo é criado, n é decrementado, dessa vez n = 2 é par e satisfaz a primeira parte da condicional da linha 15, então ocorre a divisão de execução entre processo pai e processo filho. O processo filho possui PID = 0, portanto a condicional é satisfeita,

c. Indique quantos novos programas diferentes são criados por execvp a partir de este código. Por que?

16. Um pipe chamado tofu (que recebe leituras e escritas) é utilizado pelo programa P1 e P2. De tal maneira, P1 recebe como entrada a saída de tofu, e tofu recebe como entrada a saída de P2.

a. Indique o tipo de relação entre os programas P1 e P2.

Pipe é um método de IPC (Inter Process Comunication) mais utilizado nos sistemas operacionais, existem 2 tipos de pipes. Pipes anônimos ou não nomeados: usados para comunicação unidirecional, a comunicação ocorre somente em um sentido, do processo que escreve para o processo que lê, são chamadas de anônimos pois não possuem nome ou arquivo associado, ao fim da comunicação, esse tipo de pipe são deletados. Pipes nomeados (FIFOs): usados para comunicação bidirecional, os

processos podem ler e escrever entre eles, são acessíveis por meio de um nome de arquivo no sistema de arquivos.

b. Escreva um comando de terminal que representa a situação antes descrita.

para representar um pipe bidirecional por terminal linux podemos utilizar:

mkfifo tofu (cria pipe FIFO nomeado como tofu no diretório atual)

echo “exemplo” > tofu (pode ser usado para escrever no pipe)

cat < tofu (pode ser usado para ler pipe)

P1 > tofu > P2 (comunicação entre P1 e P2 utilizando o pipe bidirecional)

17. Um programa P1 envia sua saída como entrada a P2. O programa P2 faz o mesmo com P3.

a. Indique o tipo de relação entre os programas P1, P2 e P3.

como dito anteriormente, esse tipo de comunicação representa um pipe unidirecional, que realiza comunicação em um unico sentido e é deletado ao final.

b. Escreva um comando de terminal que representa a situação.

podemos escrever um pipe unidirecional com o seguinte comando de terminal linux:

P1 | P2 | P3

18. Considere um kernel monoprocessador onde os programas de usuário podem chamar traps utilizando chamadas ao sistema. O kernel recebe e administra interrupções desde os dispositivos de entrada e saída (I/O). Existe a necessidade de administrar secções críticas dentro do kernel?

Sim, mesmo em uma máquina com monoprocessador, é necessário que seja realizada a administração de seções críticas. Existem regiões do kernel que podem conter partes que são compartilhadas entre processos no sistema, essas regiões são chamadas de seção crítica. Caso não haja administração das seções críticas para garantir exclusão mútua, ou seja, garantir que os processos não interfiram entre si no uso dessas regiões compartilhadas, podem ocorrer erros na execução dos processos.

19. Chamadas de sistema vs Chamadas de procedimentos

a. Que tão cara é uma chamada a sistema em comparação a uma chamada a um procedimento?

Uma chamada de sistema gera uma interrupção de execução, é necessário a realização de mudança de contexto de modo usuário para modo kernel e que o estado salve o contexto do processo atual, elas são mais complexas que as chamadas de procedimento, que estão vinculadas somente ao processo que as invoca. Porém, na prática, a diferença de desempenho entre chamadas de sistema e chamadas de procedimento é insignificante e não causam impacto direto na maioria dos casos.

b. Escreva um programa de prova simples para comparar o custo de uma chamada a sistema (getpid é um bom candidato em UNIX; revise o man page) em comparação com um procedimento. (Nota: cuidado com que o compilador optimize o código das chamadas a procedimentos. Não compile com o optimizador ligado.)

c. Execute seu experimento em duas arquiteturas distintas se possível (hardware distinto) e reporte os resultados.

Abaixo, escrevemos um código *shell* que não requer compilação. Um código C teria de ser compilado sem a flag -O, que ativar o otimizador.

```
$ time getpid
$ time echo "$((1 + 2))"
```

Para obter uma medida mais confiável, seria possível executar um loop:

```
$ time for ((i = 0; i < 1000; i++)); do getpid; done
$ time for ((i = 0; i < 1000; i++)); do echo "$((1 + 2))"; done
```

Executamos o código imediatamente acima no interpretador de BASH online dado neste link: (https://www.onlinegdb.com/online_bash_shell)

Como resultado, observamos que a segunda chamada (para procedimento) é muito mais rápida.

```
real    0m0.910s
user    0m0.567s
sys     0m0.154s

real    0m0.087s
user    0m0.083s
sys     0m0.004s
```

d. Explique as diferenças (se as há) entre o tempo requerido por sua chamada a procedimento e a chamada ao sistema através de uma discussão sobre que faz cada chamada (seja específico).

   Ver item A.

20. Quando um sistema operacional recebe uma chamada de sistema desde um programa, uma mudança acontece em direção ao sistema operacional com a ajuda do hardware. Nessa mudança, o hardware estabelece o modo de operação a modo supervisor, chama ao administrador de traps do sistema operacional num endereço especificado pelo sistema operacional, e permite ao sistema operacional poder retornar ao modo de usuário depois de terminar a administração da trap.

Agora, considere o stack onde o sistema operacional está executando-se quando recebe a chamada ao sistema. Deveria ser este stack diferente ao das aplicações de usuário? ou pode o sistema operacional utilizar o mesmo stack que o programa de usuário? Suponha que o programa está bloqueado enquanto a chamada de sistema retorna.

Ao executar uma chamada de sistema e realizar acesso ao kernel, os processos precisam passar a operar com uma stack específica do kernel, diferente da stack de aplicações em nivel de usuário, cada processo possui uma stack de kernel. Isso deve ocorrer por questões de segurança, para evitar que algum código malicioso proveniente do ambiente de usuário, prejudique as funções críticas do kernel, separando as permissões privilegiadas apenas para a stack do kernel, evitando que sejam causadas instabilidades no sistema como um todo.

21. Considere o seguinte código, onde a função signal estabelece uma função a ser executada com o sinal que se envia por parâmetro.

```c
void foo(int sig) {
  fprintf(stderr, "Tentando\n");
}

void bar(int sig) {
  fprintf(stderr, "Ciao\n");
  exit(0);
}

void zoo(int sig) {
  fprintf(stderr, "Tentei\n");
  signal(SIGINT, bar);
}

void main() {
  /* SIGINT é quando se pressiona CTRL-C */
  signal(SIGINT, foo);
  signal(SIGALRM, zoo);
  // Envia SIGALRM em 3 segundos depois de ser chamada
  alarm(3);
  while (1) {
    printf("zzz...\n");
    sleep(1);
  }
}
```

a. Explique o que faz a função *alarm* quando o tempo estabelecido nela termina. Detalhe o que acontece no computador, as mudanças no espaço de usuário e do kernel, e as partes envolvidas.

   A função alarm e signal são utilizadas para definir como tratar sinais assíncronos recebidos pelo sistema operacional. A função alarm define um temporizador em segundos, que ao encerrar irá gerar um sinal SIGALRM que será tratado pela função signal. Como a função alarm executa uma chamada de sistema, as mudanças que ocorrem no sistema são as mesmas que ocorrem durante uma chamada padrão de syscall (ver tópicos anteriores).

b. Como implementaria a função *alarm*?

   A implementação de alarm pode variar dependendo do sistema operacional, segundo o chatGPT uma implementação padrão para sistema baseados em Unix pode seguir os seguintes passos:

   - É realizada a chamada de sistema, com o número do sinal a ser gerado e o tempo em segundos do temporizador
   - O kernel configura um temporizador interno, utilizando contadores de relógio do sistema ou timers de hardware
   - O temporizador decresce e é monitorado pelo kernel

- Quando se encerra a contagem, é gerada uma interrupção e o kernel envia o sinal SIGALRM
- A execução do programa é interrompida e o tratador de sinal específico é chamado, isso ocorre no espaço de usuário

c. Depois de 3 segundos de iniciado o programa anterior, o que imprimirá se CTRL-C é pressionado?

Após 3 segundos de execução, o sinal SIGALRM será recebido e a função zoo será executada. Ao receber o sinal SIGINT gerado por CTRL-C, será executada a função bar.

d. Durante os primeiros 3 segundos de iniciado o programa anterior, o que imprimirá se CTRL-C é pressionado?

Antes de completar 3 segundos de execução, ao receber o sinal SIGINT por CTRL-C, será executada a função foo.

Threads

22. Threads

a. Nomeie e defina os estados de um thread e dê exemplos de situações onde um thread mude de um estado a outro.

Assim como os processos, as threads também possuem os estados pronto, executando e bloqueado. Um exemplo de mudança de estado pode ocorrer quando uma thread finaliza sua execução por limite de tempo atingido, sendo trocada do estado executando para pronto pelo escalonador do sistema.

b. Explique o que é o TCB e seu conteúdo.

TCB se refere ao Thread Control Block, uma estrutura de dados do sistema operacional, responsável por armazenar o contexto das threads do sistema, como estado, registradores, pilha de execução, para manutenção do ciclo de vida da thread (criação, escalonamento, execução, suspensão, retomada e término). O TCB se assemelha ao PCB(Process Control Block), utilizado para gerenciar processos.

c. Explique passo a passo a mudança de contexto de um thread, ademais especifique que acontece com o TCB do thread.

A mudança de contexto de uma thread é similar a mudança de contexto de um processo(ver tópicos anteriores).

23. Dado o seguinte código:

```
1  static void go(int n);
2
3  #define N 3
4  static thread_t threads[N];
5
6  int main(){
7    for(int i=0; i<N; i++)
8      thread_create(&threads[i], go, i);
9
10   for(int i=0; i<N; i++){
11     int exitValue = thread_join(threads[i]);
12     printf("Thread %d returned %d \n", i, exitValue);
13   }
14 }
15
16 void go(int n){
17     printf("Hello from thread %d\n",n);
18     thread_exit(100+n);
19 }
```

a. Explique qual é a função de thread_create()

A função thread_create (pthread_create) cria uma nova thread, recebe por parâmetro 4 argumentos. Um ponteiro para a variável onde o identificador da nova thread será armazenado, um ponteiro para um objeto de atributos que configura os atributos da thread, um ponteiro para a função que o thread irá executar, um ponteiro para os argumentos a serem passados para a função.

b. Explique qual é a função de thread_join()

A função thread_join(pthread_join) espera o término da execução de uma thread, antes de terminar a execução da thread que a chamou. Recebe dois parâmetros como argumento, o identificador da thread a ser executada, um ponteiro para onde o valor de retorno da thread será armazenado, a thread que chama a função apenas pode retomar sua execução após o término da execução da thread passada em thread_join.

c. Durante a execução do código anterior, qual é a quantidade máxima de threads que se estão executando antes de obter a mensagem Thread 1 returned?

Duas threads estão executando antes da obtenção da mensagem thread 1 returned, já que enquanto a “thread 0” está em execução por thread_join, o programa principal espera, executando “thread 1” apenas ao término da execução de “thread 0 e assim sucessivamente com “thread 2”, em qualquer ponto da execução, apenas 2 threads poderão estar executando simultaneamente, a thread em thread_join e a thread do programa principal.

d. Durante a execução do código anterior, qual é a quantidade mínima de threads que se estão executando antes de obter a mensagem Thread 2 returned?
e. Qual é a saída esperada da execução das linhas 8 e 11?

A linha 8 possui a chamada da função thread_create, que retorna um inteiro (0 em caso de criação com sucesso da thread ou um código de erro caso contrário). A linha 11 possui a chamada da função thread_join, que retorna um inteiro na mesma situação de thread_create, nesse caso a thread especificada por parâmetro na função é executada e a função go é chamada, imprimindo o número da thread em execução e finalizando sua execução retornando um inteiro por thread_exit.

24. Dado o seguinte código:

```
1  #define N 3
2  static thread_t threads[N];
3  float global_length=4;
4  float global_width=5;
5
6  float area(float w){
7    float r=global_length*w;
8    global_length--;
9    return r;
10 }
11
12 float foo(float l){
13   return l+l+l+l;
14 }
15
16 float vol(float h){
17   float v=global_length*global_width*h;
18   global_length--;
19   global_width--;
20   return v;
21 }
22
23 int main(float high){
24   float length=3, width=2;
25   thread_create(&threads[0], area, width);
26   thread_create(&threads[1], foo, length);
27   thread_create(&threads[2], vol, high);
28 }
```

a. Desenhe um esquema de como se vê a memória do processo com os threads.

Memória do processo:

-> stack do processo | stack da thread 0 | stack da thread 1 | stack da thread 2 | HEAP: | DATA: N | threads[0] | threads[1] | threads[2 | global_lenght | global_width | CÓDIGO |]

b. Desenhe o stack do processo e dos threads.

stack do processo:

-> high | ret. main | %ebp(base pointer) | width | length | w | ret. area | %ebp(base pointer) | r | l | ret. foo | %ebp(base pointer) | h | ret. vol | %ebp(base pointer) | v |

stack da thread 0:

-> | w | ret. area | %ebp(base pointer) | r |

stack da thread 1:

| l | ret. foo | %ebp(base pointer) |

stack da thread 2:

| h | ret. vol | %ebp(base pointer) | v |

c. Defina o que é uma condição de corrida, e determine se no código anterior existe uma.

Uma condição de corrida ocorre quando processos que desejam acessar dados compartilhados entre si, realizam esse acesso de forma simultânea, de forma que ao manipular esses dados, um dos processos acaba prejudicando a execução do outro por ter alterado e quebrado o funcionamento correto do dado compartilhado. Existe uma condição de corrida no código, pois ambas as threads que executam as funções area e vol, fazem acesso a variável global compartilhada global_length e alteram o seu valor.

d. Se existe uma condição de corrida no código anterior, é gerado um erro quando se executa o programa?

Não, quando ocorre a condição de corrida, não é gerado um erro que interrompa a execução do programa, porém como uma variável compartilhada está sendo alterada por duas threads, sem que haja controle de sincronização, existe alta probabilidade de que o programa apresente um erro em relação ao resultado de funcionamento esperado.

25. Considere o seguinte código:

```c
#include <system.h>
#include <stdio.h>

#define N 2
static thread_t threads[N];

char foo(float Que){
  char holl='a';
  bar(Que+4);
  sleep(15);
  return Zom;
}

int bar(float Hip){
  char Oue='a';
  float Az=50.5 ;
  bool Jack=False ;
  SysCall(Az);
  thread_yield();
  return Az+Hip;
}

int main(float Q){
  float Arr[3] = { 5, 15};
  char pop='p';
  thread_create(&threads[0], foo,Arr[0]);
  thread_create(&threads[1], bar, Arr[1]);
}

// prototipada em system.h mas definida acá
bool sysCall(float Lemon){
  int x;
  float Juice=4;
  float LJ=0.1;
  for (x=0; x<3; <++)
      LJ=Lemon+Juice;
  return true;
}
```

Suponha que as funções foo e bar são definidas pelo usuário, e que a função SysCall está definida dentro do kernel (com um protótipo dentro de system.h mas definida aqui, convenientemente).

a. Explique qual é a funcionalidade a diferença de chamar a thread_yield() e sleep() num thread.

   A função thread_yield gera um sinal para o sistema operacional, informando que a thread em execução está se voluntariado para encerrar execução e oferecer a CPU

para outra thread em estado de pronto e aguardando execução, não é garantido que a thread irá encerrar sua execução imediatamente após a chamada, dependendo do escalonamento do S.O para definir isso. Já a função sleep, bloqueia imediatamente a thread que está executando por um período de tempo especificado por parâmetro, impedindo que essa thread use a CPU durante esse período de bloqueio.

b. Especifique passo a passo a mudança a modo supervisor (ou modo kernel) gerado quando um thread chama a sysCall.

   Mesmo conceito de mudança de contexto para processos (ver tópicos anteriores)

c. Desenhe o stack do processo, kernel e threads.
d. Comente se existe condição de corrida. Se não, explique por que não existe.

26. Dado o código:

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <pthread.h>

#define N 2
static thread_t threads[N];

void escrever(){
  // Se abre um arquivo para escrever
  int fd = open ("sample.dat", O_WRONLY|O_CREAT|O_TRUNC, 0700);
  write(fd, "Escrever aqui algo interessante\n", 36);
  close(fd);
}

void ler(){
  // Se abre um arquivo para ler
  int fd = open ("sample.dat", O_RDONLY);
  char buff[20];
  ssize_t bytes_read = read(fd, buff, sizeof(buf));
  close(fd);
}

int main (int argc, char** argv){
  thread_create(&threads[0], escrever, NULL);
  thread_create(&threads[1], ler, NULL);
}
```

a. Quando um thread executa *read()* se desencadeia uma interrupção e se deve executar código privilegiado. Neste caso, só o thread que executa a chamada se detém enquanto o resto dos threads seguem executando-se com normalidade? Justifique sua resposta.

Existem 3 formas de se gerenciar threads em um sistema operacional. Threads em nivel de usuário: nesse caso as threads são invisíveis para o núcleo, que mantém apenas uma tabela de processos, o núcleo seleciona um processo pelo escalonador e apenas as threads desse processo podem ser executadas, chamando umas as outras para ciclo de execução, esse modelo é útil pois é rápido, já que não há a necessidade de realizar uma chamada de sistema para executar uma thread, porém tem uma desvantagem, já que, como o kernel não sabe da existência das threads, caso uma thread seja bloqueada, o processo todo é bloqueado, já que o S.O enxerga apenas o processo. Threads de núcleo: nesse caso o kernel mantém uma tabela de threads e pode escalonar threads de vários processos de forma independente e aleatória. Threads de múltiplos usuários e núcleo: nesse modelo existem threads de usuário e de kernel juntas, m threads de usuário compartilham n threads de kernel, diz se que a thread de usuário x é dona da thread de núcleo y. No caso do código anterior, estamos utilizando a biblioteca padrão do POSIX pthreads, que é feita baseada no modelo de threads de usuário, portanto, ao realizar uma chamada de sistema e bloquear qualquer uma das threads, todas as threads e o processo são bloqueados.

b. Explique em detalhe os passos que o sistema operacional realiza quando o thread tenta ler os dados do arquivo *fd* e armazená-los em *buff* (ver linha 20). (Deve ao menos descrever as mudanças de modo, os buffer que se utilizam para transferir a informação, e possíveis chamadas ao sistema que se desencadeiam).

   mesmo conceito dos processos (ver tópicos anteriores)

c. Comente se existe uma condição de corrida, em caso de não ter, explique.

27. Dado o seguinte código:

```c
#include <stdio.h>
#include <unistd.h>
#include <system.h>
#include <pthread.h>

#define N 2
static thread_t threads[N];

void dynamite(char** argv){
  int child = fork();
  if(child == 0)
      execvp(*argv, argv)
}

void fire(char** argv){
  char *str;
  str = (char *) malloc(15);

  excevp(*argv, argv);
}

int main (int argc, char** argv){
  int fd = open ("sample.dat", O_RDONLY);
  thread_create(&threads[0], fire, argv);
  thread_create(&threads[1], dynamite, argv);
}
```

a. Explique que acontece passo a passo quando os threads chamam às funções *fork()* e *execv()*.

b. Se um thread chama a *fork()*, duplica o filho todos os threads do pai? Justifique sua resposta.

c. Se um thread chama a *execv()*, substituirá o programa especificado ao processo completo? Justifique sua resposta.

d. O código anterior é um coquetel do caos, enquanto a programar de maneira segura com threads se refere. Comente todas as condições de corrida existentes e os possíveis problemas gerados.

e. Explique o que acontece com o arquivo aberto fd no processo filho. Que operações o processo filho pode realizar sobre o arquivo? Que informação é compartilhada ou não entre pai e filho a respeito do arquivo?

Sincronização

28. A classe Impressora imprime documentos utilizando sua função imprimir(Buffer b) que recebe como parâmetro um Buffer para obter os documentos. Os documentos são colocados num Buffer através de uma classe Biblioteca com um método armazenar(Documento d, Buffer b) que recebe como parâmetro um Documento que deve ser armazenado no Buffer.

    Uma implementação preliminar é a seguinte:

```
1  Impresora::imprimir(Buffer b) {
2    Documento d= b.obterDocumento();
3    procesar(d); // aqui procesamos o documento na impresora
4  }
5
6  Biblioteca::armazenar(Documento d, Buffer b) {
7    b.insertarDocumento(d); // insertamos
8  }
```

Porém, ao executá-la com um Buffer compartilhado temos problemas com ele.

a. Por que não funciona o código?

Esse exemplo ilustra o problema de condição de corrida que ocorre no modelo produtor(biblioteca)-consumidor(impressora). Quando o buffer de armazenamento é compartilhado simultaneamente entre as partes, é possível que o consumidor tente acessar o buffer quando não há nenhuma produção, ou que o produtor tente produzir quando o buffer está cheio e não é possível inserir mais nenhum elemento.

b. Como podemos resolvê-lo?

Uma possível solução é utilizar o modelo sleep - wakeup, quando o produtor vai inserir, ele verifica se o buffer está vazio, se estiver, ele chama wakeup para o consumidor e insere. Da mesma forma, quando o consumidor vai consumir, ele verifica se o buffer está cheio, se sim, ele chama wakeup para o produtor. Quando o produtor tenta produzir e o buffer está cheio, ele executa sleep e é bloqueado, da mesma forma, quando o consumidor tenta consumir e o buffer está vazio, ele executa sleep e é bloqueado. Dessa forma, ambos irão se acordar quando o buffer estiver adequado para utilização.

29. Suponha que no problema anterior a classe Impressora pode administrar até 5 impressoras simultaneamente. Esta nova versão da classe Impressora encapsula a nova funcionalidade através da função *processar*. Porém, a solução anterior não permite utilizar todas as impressoras simultaneamente.
    a. Por que?

    o modelo sleep-wakeup pode ocasionar condição de corrida e não funciona quando utilizamos uma versão com muitos produtores ou muitos consumidores, devido ao compartilhamento global da variável que armazena o número de elementos no buffer.

b. Como podemos resolver o problema para permitir utilizar várias impressoras simultaneamente?

   Para resolver esse problema, podemos utilizar o modelo introduzido por Dijkstra, denominado como modelo de semáforos. Esse conceito propõe a utilização de uma variável chamada de semáforo que conta o número de pedidos por desbloqueio(wakeups) e armazena esse valor. São definidos 2 generalizações de sleep e wakeup, chamadas de up e down, quando down é executado, ocorre verificação do valor do semáforo, se for maior que 0, ele é decrementado e o fluxo continua normalmente, senão, o processo que o executou é bloqueado. A oepração up incrementa o semáforo, permitindo que um processo bloqueado, escolhido arbitrariamente, seja desbloqueado e volte a execução. Para evitar que o semáforo gere condição de corrida como no modelo sleep e wakeup, as operações up e down são implementadas como chamadas de sistema, portanto ao serem executadas, as interrupções são desativadas e nenhum outro processo pode interromper o processo atual para acessar variáveis compartilhadas.

30. Numa barbearia há um barbeiro que pode trabalhar quando há clientes na barbearia. Porém, quando não há clientes este descansa.

    A barbearia está projetada com um conjunto limitado de cadeiras para que os clientes possam esperar quando o barbeiro está ocupado com um cliente.

    Quando o barbeiro termina de cortar o cabelo a um cliente, este o despacha e vai à sala de espera a chamar ao seguinte cliente. Em caso de que não haja ninguém esperando o barbeiro regressa a descansar.

    Cada cliente quando chega à barbearia observa o que acontece na barbearia. Se o barbeiro está descansando, o cliente o acorda e o barbeiro se põe a trabalhar. Do contrário, se senta numa cadeira, se há espaço, senão sai da barbearia sem seu corte de cabelo.

    Implemente duas funções, barbeiro e cliente, que simulem o comportamento descrito. (Pode utilizar um comentário para descrever onde faz trabalho o barbeiro e onde espera o cliente por seu corte de cabelo.)

31. Para a seguinte implementação de uma transferência atômica, explique se funciona, não funciona, ou se é perigosa (quer dizer, às vezes funciona e às vezes não). Explique por que e como resolveria o problema em caso de existir.

```
1  void atomicTransfer (queue *queue1, queue *queue2) {
2    item thing; /* o que transferimos */
3    queue1->lock.acquire();
4    thing = queue1->dequeue();
5    if (thing != NULL) {
6      queue2->lock.acquire();
7      queue2->enqueue(thing);
8      queue2->lock.release();
9    }
10   queue1->lock.release();
11 }
```

32. Dada uma primitiva de sincronização por hardware chamada int32 xchg(int32* lock, int32 val) que recebe um ponteiro à variável a trocar e um valor, e que retorna o valor anterior da variável lock. Se a função xchg não consegue acessar à variável lock de forma atômica, ela retornará o valor anterior nela.
    a. Implemente um mutex usando xchg, *i.e.*, implemente as funções *acquire* e *release* do mutex. Explique suas decisões de projeto.
    b. É possível usar xchg para criar um semáforo? Caso possa, escreva uma implementação de um semáforo usando xchg. Explique suas decisões de projeto. Pelo contrário, explique por que não é possível.
33. Você foi contatado pela mãe natureza para ajudá-la com a reação química que gera água, a que parece não funcionar por problemas de sincronização na *matrix*. O truque é obter dois átomos H e um átomo O todos ao mesmo tempo. Os átomos na *matrix* são threads. Cada átomo H invoca um procedimento hReady quando está pronto para reaccionar, e cada átomo O invoca um procedimento *oReady* quando está pronto. Para este problema, a mãe natureza tem o seguinte código que tenta resolver o problema. Supostamente, os procedimentos hReady e oReady esperam que existam dois átomos H e um O, posteriormente um desses procedimentos chama à função makeWater para criar a água. Depois da chamada a makeWater duas instâncias de hReady e uma instância de oReady deveriam retornar. A solução deve evitar esperar por sempre (starvation) e a espera ocupada (busy wait).

    Os semaforos da matrix implementam uma política FIFO dentro deles.

    a. Explique se o seguinte código funciona, não funciona ou é perigoso (funciona algumas vezes e outras não). Em caso de que não funcione entregue uma possível solução.

```
1  int numHydrogen = 0;
2  semaphore pairOfHydrogen(0); // init 0
3  semaphore oxygen(0);         // init 0
4
5  void hReady() {
6    numHydrogen++;
7    if ((numHydrogen % 2) == 0) {
8      pairOfHydrogen->V();
9    }
10   oxygen->P();
11 }
12
13 void oReady() {
14   pairOfHydrogen->P();
15   makeWater();
16   oxygen->V();
17   oxygen->V();
18 }
```

b. A mãe natureza entrega outra possível solução para você. Explique se o seguinte código funciona, não funciona ou é perigoso (funciona algumas vezes e outras não). Em caso de que não funcione entregue uma possível solução.

```
1  semaphore hPresent(0);     // init 0
2  semaphore waitForWater(0); // init 0
3
4  void hReady() {
5    hPresent->V();
6    waitForWater->P();
7  }
8
9  void oReady() {
10   hPresent->P();
11   hPresent->P();
12   makeWater();
13   waitForWater->V();
14   waitForWater->V();
15 }
```

34. Suponha que tem dois funções que representam um produtor e um consumidor:

```
1  void producer {
2    while(1) {
3      item = produce();
4      queue.enqueue(item)
5    }
6  }
7
8  void consumer {
9    while(1) {
10     item = queue.dequeue(item);
11     consume(item);
12   }
13 }
```

O código supõe que existe uma filha compartilhada *queue* e que as funções têm acesso a ela.

a. O código tem secções críticas? Identifique-as.

b. Usando as variáveis de exclusão mútua, proteja as regiões críticas. O que acontece com o código anterior? Ele funciona como é esperado? Explique sua resposta.
c. Se você pode usar variáveis de sincronização gerais, proteja as regiões críticas. O que acontece com o código anterior? Ele funciona como é esperado? Explique sua resposta.