Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Aula Prática 03 - Espera Ocupada v2.0

Identificação da Disciplina

Código da Disciplina	Nome da Disciplina	Turma	Professor	Período
117935	Programação Concorrente	A	Jeremias Moreira Gomes	2019/2

Objetivo da Aula Prática

O objetivo desta aula prática é apresentar técnicas básicas de comunicação entre processos/*threads* com sincronização por meio de espera ocupada.

Detalhes Acerca das Aulas Práticas

0.1. Restrições deste Trabalho Prático

Durante a escrita de quaisquer códigos, tentar não utilizar bibliotecas exóticas que não sejam as disponíveis no UNIX como biblioteca padrão. Se o fizer (pois ajuda na realização de testes, por exemplo), lembrar que no momento da correção o sistema padrão a ser utilizado é similar a uma instalação padrão das máquinas do LINF, e nada além do código será instalado apenas para uma correção individual. Então o uso de uma `<gtest.h>`, por exemplo, pode acabar fazendo com que o aluno perca pontos, porque seu código não pôde ser testado, em virtude da falta desta no computador de testes.

O atraso da entrega da atividade, incorre no decondo de 0.5 pontos a cada 60 minutos de atraso, em relação ao prazo de entrega da atividade e da nota final do aluno nesta atividade.

Esta atividade pode ser feita em dupla.

0.2. Por onde entregar os exercícios das aulas práticas?

As atividades envolvendo as aulas práticas deverão ser entregues via Aprender (https://aprender.ead.unb.br), na disciplina de Programação Concorrente. As informações para ingresso são as seguintes:

- URL: https://aprender.ead.unb.br/course/view.php?id=6775
- Chave de Acesso: `s3nhaD3ss3semestre201902`

Após o ingresso na disciplina, haverá uma atividade chamdada "Aula Prática 02 - *Threads*", para submissão dos exercícios.

0.3. O que deverá ser entregue, referente as aulas práticas?

Deverão ser entregues respostas referentes a todos os tópicos das **Seções de Atividades** ao longo deste documento. Essas atividades (dessa aula prática) estão divididas em duas categorias:

- Questionários (pergunta e resposta).
- Elaboração de Códigos.

Todos os dois tipos de atividades deverão ser entregues em um documento único contendo todas as respostas. Esse documento deverá ter identificação do aluno (nome e matrícula), identificação da disciplina, identificação da aula prática e as respostas identificadas de maneira igual as numerações em que aparecem neste documento. Para auxiliar na elaboração desse documento, pode-se utilizar esse documento (clique aqui) de referência.

Além disso, as questões de elaboração de código deverão ser entregues em arquivos (.c) separados, sendo um arquivo para cada código elaborado. O início desse código deverá vir com comentários fazendo referência ao autor do código, nome do arquivo e a identificação da atividade, da seguinte forma:

```
// autor: Jeremias Moreira Gomes
// arquivo: exemplo-arquivo.c
// atividade: 0.0.0

#include <stdio.h>

int main()
{
    printf("\n");
    return 0;
}
```

Assim, os códigos elaborados irão estar em arquivos separados e no relatório.

0.4. Como entregar as atividades das aulas práticas?

A submissão das atividades deverá ser feita em um arquivo único comprimido no tipo zip (`Zip archive data, at least v1.0 to extract`) contendo um diretório com o relatório e os códigos elaborados durante a atividade. Além disso, para garantir a integridade do conteúdo entregue, o nome do arquivo comprimido deverá possuir duas informações (além da extensão `.zip`):

- A matrícula do aluno (ou dos alunos separadas por hífen, em caso de dupla).
- O *hash* md5 do arquivo `.zip`.

Para gerar o md5 do arquivo comprimido, utilize o comando `md5sum` do Linux e em seguida faça o renomeamento utilizando o *hash* coletado. Exemplo:

```
[6189] j3r3mias@tardis:aula-01-processos|master > zip -r aaa.zip atividade-01/
  adding: atividade-01/ (stored 0%)
  adding: atividade-01/relatorio.docx (stored 0%)
  adding: atividade-01/01-hello-3-fork.c (deflated 34%)
  adding: atividade-01/03-exemplos.c (deflated 47%)
  adding: atividade-01/02-arvore.c (deflated 35%)
  adding: atividade-01/02-pid_t.c (deflated 39%)
  adding: atividade-01/01-hello-fork.c (deflated 21%)
  adding: atividade-01/03-processos-e-ordens.c (deflated 55%)
[6190] j3r3mias@tardis:aula-01-processos|master > ls -lha aaa.zip
-rw-rw-r-- 1 j3r3mias j3r3mias 2,5K set  4 23:26 aaa.zip
[6191] j3r3mias@tardis:aula-01-processos|master > md5sum aaa.zip
fe36b6799eae103a464cbc4857fce404  aaa.zip
[6192] j3r3mias@tardis:aula-01-processos|master > mv aaa.zip 160068444-fe36b6799eae103a464cbc4857fce404.zip
[6193] j3r3mias@tardis:aula-01-processos|master > ls -lha 160068444-fe36b6799eae103a464cbc4857fce404.zip
-rw-rw-r-- 1 j3r3mias j3r3mias 2,5K set  4 23:26 160068444-fe36b6799eae103a464cbc4857fce404.zip
[6194] j3r3mias@tardis:aula-01-processos|master > md5sum 160068444-fe36b6799eae103a464cbc4857fce404.zip
fe36b6799eae103a464cbc4857fce404  160068444-fe36b6799eae103a464cbc4857fce404.zip
[6195] j3r3mias@tardis:aula-01-processos|master >
```

1. Variáveis Compartilhadas

Na aula anterior, foi pedido para que o aluno implementasse uma solução que se aproveitava da possibilidade de variáveis compartilhadas para funcionar um esquema de largada em uma corrida. Uma das maneiras de se implementar tal código poderia ter sido dessa forma:

```
// Bibliotecas

int largada = 3;

void *carro(void *cid)
{
    long int id = (long int )cid;
    while (1) {
        if (largada == 0) {
            printf("Sou o carro %ld e terminei a corrida.\n", id);
            pthread_exit(NULL);
        }
    }
}

void *juizdelargada(void *empty)
{
    int sono;
    while (largada > 0) {
        printf("Esperando %d segundos no sinal %d.\n", sono, largada);
        sleep(1);
        largada--;
    }
}

int main()
{
    pthread_t t[10], l;

    for (long int i = 0; i < 10; i++) {
        pthread_create(&t[i], NULL, carro, (void *)i);
    }
    pthread_create(&l, NULL, juizdelargada, NULL);
    pthread_join(l, NULL);
    for (long int i = 0; i < 10; i++) {
        pthread_join(t[i], NULL);
    }
    return 0;
}
```

Nesse código, a variável `largada` é uma variável compartilhada que pode ser acessada por todas as *threads*. Para este caso, uma única *thread* é responsável por fazer a modificação do estado de `largada` enquanto todas as outras (exceto a *thread main*) tem por objetivo ficar monitorando o estado dela até que seu valor seja zero, indicando o início da corrida. Situações a qual duas ou mais *threads* podem acessar os mesmos dados compartilhados são conhecidas como **Condições de Corrida**. Se nenhuma ou apenas uma *thread*, como era o caso do exercício, modifica o valor da variável compartilhada, isso pode não ser um problema para a a execução de um algoritmo. Porém, se a alteração resulta na modificação por *threads* diferentes e valores diferentes, o resultado final da execução pode acabar não sendo o esperado e a condição de corrida termina por interferir na corretude do algoritmo.

1.1. Atividades

1.1.1. Incremento Compartilhado com Problemas

Para visualizar os problemas decorrentes do compartilhamento de variáveis, implemente um algoritmo que possua uma variável compartilhada contador, crie 10 *threads* e cada uma dessas deverá executar a função `incrementos` que irá incrementar esse contador em uma unidade 100 vezes (utilizando uma estrutura de repetição). Após todas as *threads* realizarem os seus devidos incrementos, a *thread main* deverá imprimir o valor final do contador.

1.1.2. Qual o problema do código anterior? Se há algum problema, ele acontece sempre? Por quê?

1.1.3. De que forma seria possível resolver o problema do código, utilizando os conhecimentos já apresentados na disciplina?

1.2. Quais são as quatro condições para se evitar condições de corrida?

2. Exclusão Mútua e Regiões Críticas

Já foram mostradas no curso diversas tentativas (a maioria frustradas) para a implementação de *locks* ou travas, por meio de algoritmos. Dentre as apresentadas, uma delas utilizava a estratégia chamada de Estrita Alternância, em que apenas uma *thread*, por vez, tem acesso a região crítica, e ambas ficam alternando a vez do acesso à região.

2.1. Atividades

2.1.1. Banco de Um Cliente

Implemente um algoritmo que crie duas *threads*. Uma irá executar a função `deposito` e a outra irá executar a função `retirada`. Ambas as funções deverão utilizar Estrita Alternância, para variarem entre depósitos ou retiradas entre si, alterando uma variável global chamada `contabancaria`. Os depósitos deverão ser de 20 unidades, enquanto as retiradas deverão ser de apenas 10. Se o valor da retirada for maior do que o conteúdo presente na conta bancária, a retirada não deve acontecer.

2.1.2. O que acontece, com o funcionamento do algoritmo, como um todo, se a *thread* responsável pelo depósito terminar sua execução?

2.1.3. Altere o algoritmo anterior para, após 10000 iterações, a *thread* que executa a função *deposito* terminar a sua execução.

2.1.4. Implemente um exemplo de estrita alternância, utilizando três *threads*.

3. Implementação de Espera Ocupada por Hardware

Foi visto em aula que, além das implementações em softwares, soluções comerciais possuem instruções de hardware que auxiliam na implementação de exclusão mútua. Em sua maioria, essas soluções se apresentam por meio de instruções que realizam um conjunto reduzidode instruções de maneira atômica, permitindo assim uma conferência e manipulação de um endereço de dados, por exemplo.

De forma análoga a descrita, a arquitetura Intel possui em sua especificação um conjunto de construções primárias, apresentadas em *Intel Itanium Processor-specific Application Binary Interface* que foram traduzidas pelo GCC no conjunto de instruções chamadas de *Atomic Built-ins* (link). Esse conjunto de instruções permite realizar acesso mais alguma outra operação de maneira atômica à um determinado endereço de memória.

Uma outra preocupação ao escrever códigos concorrentes e paralelos, é que alguns trechos de código podem ser vistos como inúteis para o compilador, cujo as otimizações podem acabar por remover ou até modificar a ordem das instruções de acordo com o que foram escritas. Assim, para fazer uso de instruções atômicas como as *Atomic Built-ins*, pode ser interessante ou necessário utilizar o modificador *volatile* na criação da variável, que irá informar ao compilador que essa variável pode ser modificada sem o conhecimento do programa principal. Como consequência desse modificador, otimizações envolvendo essa variável serão evitados pelo compilador.

3.1. Atividades

3.1.1. Incremento Atômico

Implemente o algoritmo pedido na Seção 1.1.1, utilizando, agora, a instrução atômica

`__sync_fetch_and_add`.

3.1.2. Instrução ou Conjunto de Instruções

Lendo o conjunto de instruções do binário gerado, em qual instrução função `__sync_fetch_and_add` foi convertida?

4. TOCTTOU

Para entender a criticidade e os problemas que condições de corridas podem ocasionar, esta atividade apresenta um tipo de falha de segurança chamada de *Time to Check to Time to Use* (TOCTTOU). Nesse tipo específico de falha, há um problema de condição de corrida quando há um lapso de tempo entre uma verificação de credenciais de segurança e o tempo para a execução de uma rotina propriamente dita. Essa falha acontece regularmente em sistemas de arquivos, onde após a checagem de uma credencial de segurança, em seguida um atacante mal-intencionado troca o arquivo original (em que foi checado o acesso), por um ao qual ele não o deveria ter. Esse trecho pode ser exemplificado pelo seguinte trecho:

```
if (access("file", W_OK) != 0) {
   exit(1);
}

fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

Aqui, no condicional, é verificado se o usuário tem acesso ao arquivo `file` e logo em seguida o arquivo é lido. Imagine que logo após o programa realizar a verificação, e antes da abertura do arquivo em si, um outro processo execute o seguinte:

```
symlink("/etc/passwd", "file");
```

Com isso, se o binário tem permissões para abrir o arquivo `\etc\passwd`, mas o usuário que executou o programa não tem tais credenciais, a verificação irá falhar pela modificação de estado do link simbólico entre a verificação e a abertura do arquivo, permitindo um usuário com credenciais menores acessar um arquivo protegido. Nesse caso, um binário que tem permissões acima das do usuário que as executa é apresentado como permissões especiais do tipo ***setuid***, onde ele executa com permissões de administrador, mesmo sendo um usuário comum o executando.

4.1. Atividades

Sua tarefa nessa atividade remete a escrever um *script* para burlar a segurança do arquivo `\etc\passwd`, aproveitando-se de condições de corrida para inserir um novo usuário com credenciais de administrador no sistema.

Dessa forma, serão disponibilizados o código do leitor de arquivos, as configurações do ambiente (por um arquivo Dockerfile), os comandos para montar esse ambiente e a forma de gerar as especificações de um usuário no sistema Unix.

Segue o código do arquivo `codigo.c`:

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

int main()
{
    char *fn = "/tmp/output";
    char conteudo[201];
    FILE *fp;

    scanf("%200s", conteudo);

    if (!access(fn, W_OK)) {
        fp = fopen(fn, "a+");
        fwrite(conteudo, sizeof(char), strlen(conteudo), fp);
        fclose(fp);
    } else {
        printf("Sem permissão de escrita no arquivo.\n");
    }

    return 0;
}
```

Segue o conteúdo do arquivo `Dockerfile`, com a preparação do ambiente:

```
FROM gcc:4.9

RUN apt-get update
RUN apt-get install -y vim nano htop

COPY codigo.c /tmp
WORKDIR /tmp
RUN gcc -o prog codigo.c
RUN useradd aluno

RUN chown root prog
RUN chmod 4755 /tmp/prog
RUN rm codigo.c

USER aluno
ENTRYPOINT ["/bin/bash"]
```

Caso opte por não utilizar Docker, pode-se acompanhar o próprio arquivo de configurações dele para preparar um ambiente similar ao necessário para realizar o laboratório.

Comandos para executar na construção e montagem do ambiente:

- Nomeie o arquivo de configurações do Docker como `Dockerfile`.
- Nomeie o arquivo com o código como `codigo.c`
- Coloque ambos os arquivos no mesmo diretório e entre nele.
- Para construir a imagem, digite: `docker build -t condicaodecorrida .`
- Para executar o ambiente, digite: `docker run --privileged -it condicaodecorrida`
- De fora do ambiente, em outro terminal, verifique o nome da máquina criada executando o comando `docker ps`
- Após verificar o nome, digite o seguinte comando, substituindo `<nome>` pelo respectivo nome: `docker exec -itu root <nome> sysctl -w fs.protected_symlinks=0`

Feito todos esses passos, o ambiente estará pronto para ser utilizado. Se quiser acessar mais de um terminal ao mesmo tempo dentro dessa máquina, você pode utilizar:

```
docker exec -it <nome> /bin/bash
```

4.1.1. Monte um ataque, que possa ser executado por meio um ou mais *scripts* em *bash*, para conseguir inserir um novo usuário com permissões administrativas dentro do *container* Docker que está executando.

Para a execução desta tarefa, o primeiro passo é ter um arquivo que contenha os dados de um usuário com permissões administrativas para inserir. Esses dados constituem, essencialmente do nome do usuário, o *password* e a permissão desse usuário. Para gerar o *password* no padrão utilizado pelo arquivo `/etc/passwd`, pode-se utilizar o `openssl` da seguinte forma:

```
openssl passwd -6 -salt <salt> <senha>
```

Exemplo:

```
j3r3mias@tardis:2019-2|master > openssl passwd -6 -salt jer 123
$6$jer$Og8bGWzZKkxlfgfFbRhTYKx3axvbBLtibf9DVr7r6zKV3AL55dAf8x.qEHrqI1TzsmVYurnO3xZjayIxGWjfy1
```

Com a senha, a linha a ser inserida no arquivo `/etc/passwd` deverá estar da seguinte forma:

```
j3r3mias:$6$jer$Og8bGWzZKkxlfgfFbRhTYKx3axvbBLtibf9DVr7r6zKV3AL55dAf8x.qEHrqI1TzsmVYurnO3xZjayIxGWjfy1:0:0::/home:/bin/sh+
```

A partir desse ponto falta, então, entender o ponto do código (arquivo `codigo.c`) onde ocorre o TOCTTOU, para tentar redirecionar a escrita do nome do novo usuário para o arquivo `/etc/passwd`.

O ataque deverá (da maneira como foi concebido) proceder da seguinte forma:

1. Você cria o arquivo que o programa irá escrever o conteúdo.
2. Você executa o programa e passa para ele o conteúdo que ele deverá escrever no arquivo.
3. Enquanto o programa checa as permissões do arquivo que você criou no passo 1, você transforma o arquivo criado em um link simbólico para o arquivo `/etc/passwd`.
4. Assim, o programa irá escrever o conteúdo que você passou em `/etc/passswd`, incluindo um novo usuário ao final do arquivo.
5. Por fim, o comando `su <usuario>` permitirá escalar privilégios agora de administrador no *container*.

Por último, lembre-se que você depende da Lei de Murphy para que a ordem das instruções na região crítica ocorram da forma esperada, então automatizem a repetição dessas execuções.