MINISTÉRIO DA EDUCAÇÃO
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE MINAS GERAIS
IFMG CAMPUS FORMIGA

CURSO: BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO
DISCIPLINA: "**SISTEMAS OPERACIONAIS**"
PROF.º EVERTHON VALADÃO everthon.valadao@ifmg.edu.br

TRABALHO EM GRUPO

VALOR: 30 PONTOS

DATA MÁXIMA DE ENTREGA: 09/12/2016 via portal *meuIFMG* (https://meu.ifmg.edu.br)

IMPORTANTE, leia atentamente as instruções a seguir:

- Trabalho prático em grupo, a ser realizado em trio (três alunos). Exceções devem ser previamente comunicadas e aprovadas pelo professor;

- Para a implementação do trabalho, **exige-se** a utilização:
  - da linguagem de programação C (ou C++) e da biblioteca pthreads (POSIX Threads)
  - o programa deve ser compilável no sistema operacional Linux (Ubuntu 16.04), com o compilador GCC e a biblioteca libpthread*-dev

- Deverá ser submetido no portal *meuIFMG* (https://meu.ifmg.edu.br) um arquivo compactado (.zip ou .tgz) contendo:
  - **código-fonte** do programa (.c e .h) e um **Makefile**;
  - um **relatório** sobre o trabalho, descrevendo a estrutura geral do código, decisões importantes (ex.: como lidou com ambiguidades na especificação), bugs conhecidos ou problemas (lista dos recursos que não implementou ou que sabe que não estão funcionando).

ATENÇÃO: não será tolerado plágio. A critério, poderá ser realiza reunião com os membros do grupo.

“Problema de sincronização usando threads”

GRUPO

Trabalho prático em grupo, a ser realizado em trio (três alunos). Exceções devem ser previamente comunicadas e aprovadas pelo professor.

ENUNCIADO

Este trabalho tem por objetivo fazer com que os alunos experimentem na prática a programação de problemas de sincronização entre processos e os efeitos da programação concorrente. Isso deve ser feito utilizando-se os recursos de threads POSIX (pthreads) — em particular, mutexes (pthread_mutex_t) e variáveis de condição (pthread_cond_t).

O PROBLEMA

O estacionamento do *campus* do IFMG possui vagas limitadas, de maneira que atualmente concorrem vários funcionários para cada vaga de veículo. Naturalmente, uma vaga não pode ser utilizada simultaneamente por dois carros e para resolver conflitos de uso há determinadas prioridades de uso entre funcionários. Por questões de simplicidade, considere abaixo que existe apenas uma vaga sendo concorrida por seis pessoas.

SINCRONIZAÇÃO: Uma determinada **vaga do estacionamento** é compartilhada pelo prof.º **Girafales**, prof.º **Xavier** e prof.º **Walter**. Para decidir quem podia usar a vaga, definiram o seguinte conjunto de regras:

- se a vaga estiver liberada, quem chegar primeiro para trabalhar pode estacionar seu carro;
- caso contrário, quem chegar depois tem que esperar a vaga ser desocupada.
- se mais de uma pessoa estiver esperando para estacionar, valem as precedências:
  – Girafales pode usar antes de Xavier;
  – Xavier pode usar antes de Walter;
  – Walter pode usar antes de Girafales.
- quando alguém termina de trabalhar, deve liberar a vaga para a próxima pessoa de maior prioridade, exceto em dois casos:
  – para evitar inanição (discutido a seguir);
  – quando houver *deadlock* (= um ciclo de prioridades).

Cada um destes três professores está associado a um técnico administrativo, formando uma equipe: prof.º Girafales é apoiado pela técnica **Florinda**, prof.º Xavier recebe auxílio da técnica **Jean** e prof.º Walter tem ajuda do técnico **Pinkman**. Os dois membros de uma mesma equipe podem trabalhar separadamente. Os técnicos entram no esquema da vaga da seguinte forma: cada técnico tem a mesma prioridade do professor a quem está associado. Só que se os dois membros de uma mesma equipe quiserem usar a vaga, um tem que esperar depois do outro (por ordem de chegada entre eles).

OBS.: *quer dizer, se Pinkman, Walter e Xavier estiverem esperando para estacionar e ninguém mais chegar, Xavier usa primeiro pois tem prioridade sobre Walter/Pinkman, depois usa a vaga Pinkman ou Walter, dependendo de quem chegou primeiro entre eles (pois têm a mesma prioridade); mas se Jean chegasse antes do Xavier acabar, ela teria preferência de usar em seguida a Xavier (antes de Walter/Pinkman).*

INANIÇÃO: Atente para o fato de que isso pode levar à **inanição** em casos de uso intenso da vaga, daí é preciso criar uma regra para resolver o problema: se a mesma equipe usar a vaga em seguida, o segundo membro desta equipe é obrigado a ceder a vez para um membro da outra equipe que normalmente teria que esperar por eles. Obviamente, o segundo membro da equipe só pode usar a vaga se nesse meio tempo não chegar alguém com maior prioridade.

DEADLOCK: Cada funcionário, como os filósofos daquele famoso problema, dividem seu tempo entre períodos em que fazem outras coisas (preparam aula, corrigem prova) e períodos em que resolvem estacionar o carro para lecionar no *campus*. O tempo que cada um gasta com outras coisas varia entre 3 e 5 segundos e usar a vaga gasta 1 segundo.

OBS.: *os tempos das outras coisas são apenas uma referência, você pode experimentar tempos um pouco diferentes, se for mais adequado. Certifique-se de que em alguns casos esses tempos sejam suficientes para gerar alguns deadlocks de vez em quando, bem como situações que exijam o mecanismo de prevenção de inanição.*

Se você reparar as precedências definidas, vai notar que é possível ocorrer um ***deadlock*** (Girafales → Xavier → Walter → Girafales) — e eles sabem, mas são muito orgulhosos para mudar. Para evitar isso, o **diretor** do *campus* periodicamente (a cada 5 segundos) confere a situação e, se encontrar o pessoal travado (vaga vazia e alguém de cada equipe esperando), pode escolher um deles aleatoriamente e liberá-lo para usar a vaga.

IMPLEMENTAÇÃO

Sua tarefa neste trabalho é implementar os funcionários do *campus* como **threads** e implementar a vaga como um **monitor** usando **pthreads**, **mutex** e variáveis de **condição**.

Parte da solução requer o uso de uma mutex para a vaga, que servirá como a trava do monitor para as operações que os funcionários podem fazer sobre ele. Além da mutex, você precisará de um conjunto de variáveis de condição para controlar a sincronização do acesso das equipes à vaga: uma variável para enfileirar o segundo membro de uma equipe se o outro já estiver esperando, e outra variável de condição para controlar as regras de precedência no acesso direto à vaga.

O programa deve criar os sete funcionários (threads) e receber como parâmetro o número de vezes que eles vão tentar usar a vaga — afinal, não dá para ficar assistindo eles fazerem isso para sempre. Ao longo da execução o programa deve então mostrar mensagens sobre a evolução do processo. Por exemplo:

```
$ ./estacionamento 2
Girafales quer usar a vaga
Girafales estaciona para trabalhar
Walter quer usar a vaga
Xavier quer usar a vaga
Florinda quer usar a vaga
Girafales vai pra casa estudar
Pinkman quer usar a vaga
Diretor detectou um deadlock, liberando Xavier
Xavier estaciona para trabalhar
Girafales quer usar a vaga
Xavier vai pra casa estudar
Walter estaciona para trabalhar
Walter vai pra casa estudar
Pinkman quer usar a vaga
Walter quer usar a vaga
Florinda começa a usar a vaga
...
```

Nota: a ordem dos eventos acima é apenas um exemplo, devendo variar entre execuções (devido ao escalonamento das threads) e implementações. Você não deve esperar executar o programa e obter o mesmo exemplo acima, mas deve obter o mesmo resultado, ou seja, respeitar as prioridades de uso da vaga (sincronização), evitar inanição e resolver deadlocks.

SUMARIZANDO:

- membros do mesma equipe esperam um atrás do outro para usar a vaga;
- as regras de preferência definidas acima valem a não ser que uma equipe tenha que ceder a vez;
- deadlock deve ser resolvido pela atuação do Diretor;
- inanição deve ser evitada pela regra da equipe ceder a vez.

Para implementar a solução, analise as ações de cada funcionário em diferentes circunstâncias, como o outro membro da equipe já está esperando, há alguém com maior prioridade esperando, há alguém com menor prioridade esperando. Determine o que fazer quando alguém decide que quer usar a vaga e tem alguém de maior prioridade, o que fazer quando terminam de usar a vaga, etc.

Consulte as páginas de manual no Linux para entender as funções da biblioteca para **mutex** e variáveis de condição (**cond**):

- `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);`
- `int pthread_mutex_lock(pthread_mutex_t *mutex);`
- `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
- `int pthread_mutex_destroy(pthread_mutex_t *mutex);`

- `int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)`
- `int pthread_cond_signal(pthread_cond_t *cond);`
- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`
- `int pthread_cond_destroy(pthread_cond_t *cond);`

DICAS DE IMPLEMENTAÇÃO

Como mencionado anteriormente, você deve usar uma mutex para implementar a funcionalidade equivalente a um monitor, isto é, as operações sobre a vaga serão parte de um monitor. Na prática, isso será implementado em C, mas deverá ter a funcionalidade equivalente a:

```
//PSEUDOCÓDIGO
monitor vaga
{
	... // variáveis compartilhadas, variáveis de condição

	void esperar(int funcionário) {
		printf("%s quer usar a vaga\n", nome(funcionário));
		... // verifica quem mais quer usar, contadores, variáveis de cond., etc.
	}

	void liberar(int funcionário) {
		printf("%s vai pra casa estudar\n", nome(funcionário));
		... // verifica se tem que liberar alguém, atualiza contadores, etc.
	}

	void verificar() {
		... // diretor verifica se há deadlock e corrige-o
	}
};
```

Os funcionários **professor** ou **técnico** executam as seguintes operações um certo número de vezes (definido pelo parâmetro de entrada na execução do programa):

```
vaga.esperar(p);          // exige mutex
estaciona_na_vaga(p);     // não exige exclusão mútua
vaga.liberar(p);          // exige mutex
vai_embora_estudar(p);    // espera um certo tempo aleatório
```

Use **srand**() — confira a página do manual — para gerar números aleatórios entre 0 e 1, use uma multiplicação para gerar inteiros aleatórios.

Já o funcionário **diretor** executa as seguintes operações:

```
enquanto funcionários estão ativos faça
	sleep(5);
	vaga.verificar();
```

INFORMAÇÕES ÚTEIS

Forma de operação

O seu programa deve basicamente criar uma thread para cada funcionário e esperar que elas terminem. Cada funcionário executa um loop um certo número de vezes (parâmetro de entrada na linha de comando), exceto o diretor, que deve executar seu loop até que todos os outros funcionários tenham acabado.

Codificação dos funcionários

Você deve buscar produzir um código elegante e claro. Em particular, note que o comportamento das equipes é basicamente o mesmo, você não precisa replicar o código para diferenciá-los. Além disso, o comportamento de todos os funcionários (exceto o diretor) é tão similar que você deve ser capaz de usar apenas uma função para todos eles, parametrizada por um número, que identifique cada funcionário. As prioridades podem ser descritas como uma lista circular.

Acesso às páginas de manual

Para encontrar informações sobre as rotinas da biblioteca padrão e as chamadas do sistema operacional, consulte as páginas de manual online do sistema (usando o comando Unix man). Você também vai verificar que as páginas do manual são úteis para ver que arquivos de cabeçalho que você deve incluir em seu programa. Em alguns casos, pode haver um comando com o mesmo nome de uma chamada de sistema; quando isso acontece, você deve usar o número da seção do manual que você deseja: por exemplo, `man read` mostra a página de um comando da shell do Linux, enquanto `man 2 read` mostra a página da chamada do sistema.

Manipulação de argumentos de linha de comando

Os argumentos que são passados para um processo na linha de comando são visíveis para o processo através dos parâmetros da função main():

```
int main (int argc, char * argv []);
```

o parâmetro argc contém um a mais que o número de argumentos passados (no caso deste enunciado o `argc` deverá ter valor igual a 2) e `argv` é um vetor de strings, ou de apontadores para caracteres (no caso deste enunciado o número de iterações deverá ser obtido em `argv[1]`).

Processo de desenvolvimento

Lembre-se de conseguir fazer funcionar a funcionalidade básica antes de se preocupar com todas as condições de erro e os casos extremos. Por exemplo, primeiro foque no comportamento de um funcionário e certifique-se de que ele funciona. Depois dispare dois funcionários apenas, para evitar que deadlocks aconteçam. Verifique se as equipes funcionam, inclua o diretor e verifique se deadlocks são detectados (use um pouco de criatividade no controle dos tempos das outras atividades para forçar um deadlock, para facilitar a depuração). Finalmente, certifique-se que o mecanismo de prevenção da inanição funciona (p.ex., use apenas duas equipes e altere os tempos das outras atividades para fazer com que um deles (o de maior prioridade) esteja sempre querendo usar a vaga. Exercite bem o seu próprio código! Você é o melhor testador dele (* mas não se esqueça de pedir para outras pessoas testarem para tentar identificar problemas que você tenha deixado passar desapercebidos).

Mantenha versões do seu código. Ao menos, quando você conseguir fazer funcionar uma parte da funcionalidade do trabalho, faça uma cópia de seu arquivo C ou mantenha diretórios com números de versão. Ao manter versões mais antigas, que você sabe que funcionam até um certo ponto, você pode trabalhar confortavelmente na adição de novas funcionalidades, seguro no conhecimento de que você sempre pode voltar para uma versão mais antiga que funcionava, se necessário.

O que deve ser entregue

Você deve entregar através do portal meuIFMG um arquivo .zip ou .tgz com o(s) arquivo(s) contendo o código fonte do programa (.c e .h), um Makefile e um relatório sobre o seu trabalho, que deve conter:

- Um resumo do projeto: alguns parágrafos que descrevam a estrutura geral do seu código e todas as estruturas importantes.
- Decisões de projeto: descreva como você lidou com quaisquer ambiguidades na especificação.
- Bugs conhecidos ou problemas: uma lista de todos os recursos que você não implementou ou que você sabe que não estão funcionando corretamente

Não inclua a listagem do seu código no relatório; afinal, você já vai entregar o código fonte!

Finalmente, embora você possa desenvolver o seu código em qualquer sistema que quiser (POSIX), certifique-se que ele execute corretamente no Linux Ubuntu dos laboratórios de informática ou numa máquina virtual com o sistema operacional Linux Ubuntu versão 16.04. A avaliação do funcionamento do seu código (compilação e execução) será feita neste ambiente.

Considerações finais

Este trabalho não é tão complexo quanto pode parecer à primeira vista. Talvez o código que você escreva seja mais curto que este enunciado. Escrever o seu monitor será uma questão de entender o funcionamento das funções de pthreads envolvidas e utilizá-las da forma correta. **O programa final deve ter apenas poucas centenas de linhas de código**. Se você observar que esteja escrevendo código mais longo que isso, provavelmente é uma boa hora para parar um pouco e pensar mais sobre o que você está fazendo. Entretanto, dominar os princípios de funcionamento e utilização das chamadas para manipulação de variáveis de condição e mutexes e conseguir a sincronização exata desejada pode exigir algum tempo e esforço.

1. Dúvidas: envie e-mail para everthon.valadao@ifmg.edu.br ou procure o professor fora do horário de aula (agenda).
2. Comece a fazer o trabalho logo, pois apesar do programa final ser relativamente pequeno, o tempo não é muito e o prazo de entrega não vai ficar maior do que ele é hoje (independente de que dia é hoje).
3. Vão valer pontos clareza, qualidade do código e da documentação e, obviamente, a execução correta com programas de teste.

Créditos: o enunciado deste trabalho foi baseado no material do Prof.º Dorgival Olavo Guedes Neto (UFMG).