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:

20 PONTOS (*threads* com sincronização e tratamento de *deadlocks*)

DATA MÁXIMA DE ENTREGA:

09/12/2022 via https://classroom.google.com

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 (sugestão: Ubuntu 22.04 LTS), com o compilador GCC e a biblioteca libpthread*-dev

- Deverá ser entregue via Google Classroom um arquivo compactado (.zip ou .tar.gz) contendo:
  - **código-fonte** do programa (.c e .h) e um **Makefile**;
  - um **relatório** sobre o trabalho descrevendo, de maneira sucinta e objetiva, a estrutura geral do código, decisões importantes (ex.: como lidaram 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” (versão 1.0)

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

Saulo, Vitor e Kelvin moram em uma república e compraram um microondas juntos. Para decidir como usar o forno sem brigas, definiram um conjunto de regras, apresentado adiante. Por questões de simplicidade (e por ser uma república pequena), considere abaixo que existem apenas seis pessoas específicas que irão utilizar o único forno microondas da república, conforme explicado abaixo.

1ª ETAPA) SINCRONIZAÇÃO: O **único forno microondas** da república deve ser compartilhado pelo sênior **Saulo** (quase se formando), o veterano **Vitor** (que está no meio do curso) e calouro **Kelvin** (matriculado no 1º período). Para decidir quem podia usar o forno, definiram o seguinte conjunto de regras:

- se o forno estiver liberado, quem chegar primeiro na cozinha já **pode usar** o forno;
- caso contrário, se o forno já estiver sendo utilizado, quem chegar depois **tem que esperar** o forno ser desocupado.
- se mais de uma pessoa estiver esperando na cozinha, valem as seguintes precedências:
  - **Sênior** pode usar antes de **Veterano**;
  - **Veterano** pode usar antes do **Calouro**;
  - **Calouro** só poderá usar o forno após as pessoas com maior prioridade terem-no utilizado;

  OBS.: pessoas de mesma prioridade são atendidas na ordem de sua chegada (ex.: dois veteranos ou dois sêniores).
- quando alguém termina de ser atendido, deve liberar o forno para a próxima pessoa de maior prioridade, exceto em dois casos:
  - para evitar inanição (discutido a seguir);
  - quando houver *deadlock* (i.e.: tiver sido formado um ciclo de prioridades).

Cada uma destas quatro pessoas eventualmente recebe visita de outro/a estudante de igual prioridade: **sênior Saulo** é parceiro do **sênior Samir**, **veterano Vitor** estuda com **veterana Vani**, **calouro Kelvin** desabafa com **caloura Kamila**.

Amigos de mesma prioridade podem frequentar a república separadamente e usar o forno independente um do outro. Os amigos entram no esquema de prioridades da seguinte forma: cada estudante tem a mesma prioridade da pessoa a quem está associado. Só que, se ambos amigos (de mesma prioridade) quiserem utilizar o forno microondas, um tem que esperar depois do outro (por ordem de chegada entre eles).

> OBS.: *quer dizer, se Vitor, Kelvin e Kamila estiverem esperando para usar o forno e ninguém mais chegar, Vitor usa primeiro pois tem prioridade sobre Kelvin/Kamila, depois utilizará o forno Kelvin ou Kamila, dependendo de quem chegou primeiro entre eles (pois têm a mesma prioridade). Por outro lado, se Samir chegasse antes de Vitor acabar de usar o forno, ele teria preferência de atendimento logo em seguida a Vitor (antes de Kelvin/Kamila), “furando fila”.*

INANIÇÃO: Atente para o fato de que o tratamento de fila com prioridades pode levar à **inanição** em casos de uso intenso do forno, daí é preciso criar uma regra para resolver o problema. Se uma pessoa de menor prioridade não conseguir usar o forno **por duas vezes em seguida** (porque outras pessoas com maior prioridade chegaram depois dela mas foram atendidas primeiro), então esta pessoa sofrendo inanição deverá se beneficiar de um "envelhecimento". Gradualmente incremente a prioridade da pessoa em inanição, incrementando sua categoria de prioridade (uma vez a cada dupla frustração de "furarem fila na sua frente") até que eventualmente ela seja atendida e, logo após, volte ao seu patamar original de prioridade de atendimento. Qualquer pessoa com prioridade menor que outras pode sofrer inanição!

[ATENÇÃO] Se você reparar as precedências definidas, vai notar que nesta primeira etapa do trabalho ainda não ocorrerá *deadlocks*. É isso mesmo: nesta primeira etapa do trabalho ainda não ocorrerão deadlocks, apenas eventual inanição.

DEADLOCK: Cada pessoa, como os filósofos daquele famoso problema, dividem seu tempo entre períodos em que fazem outras coisas (estudam, dormem, jogam, etc) e períodos em que resolvem ir à cozinha para esquentar algo para comer. Nesta simulação de República, o tempo que cada um gasta com outras coisas varia entre 3 e 5 segundos e utilizar o forno da cozinha leva 1 segundo.

OBS.: *os tempos das outras coisas são apenas uma referência, você pode experimentar valores de tempo um pouco diferentes, se for mais adequado aos seus testes. 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.*

Para a entrega da segunda etapa do trabalho deverá ser implementada uma pequena modificação no mecanismo de prioridades. O sênior Saulo está enamorado pela caloura Kamila, de maneira que implementou uma nova política de prioridades no uso do forno, tentando agradar seu flerte: daqui em diante, se mais de uma pessoa estiver esperando na fila para usar o forno, valem as seguintes precedências:

– **Sênior** pode usar antes de **Veterano**;
– **Veterano** pode usar antes do **Calouro**;
– **Calouro** pode usar antes de **Sênior**.

OBS.: pessoas de mesma prioridade são atendidas na ordem de sua chegada.

Agora, é possível ocorrer um ***deadlock*** (ex.: Samir → Vani → Kelvin). Mesmo percebendo seu vacilo, o sênior Saulo não quer voltar atrás na modificação de prioridades mesmo sabendo que este problema (*deadlock*) pode ocorrer. Para evitar uma trava geral na cozinha, **Flávio** (que faz faxina na república) periodicamente confere a situação da fila do forno (a cada 5 segundos) e, se encontrar o pessoal da república "travado" (forno ocioso e alguém de cada categoria de prioridade esperando uns pelos outros), escolhe uma pessoa da fila aleatoriamente e libera-a para usar o forno, destravando.

[ATENÇÃO] O tratamento de *deadlocks* deve ser implementado após você ter conseguido implementar com sucesso a primeira etapa do trabalho (faça um *backup*), de acordo com a modificação descrita acima.

IMPLEMENTAÇÃO

Sua tarefa neste trabalho é implementar as pessoas da cidade como **threads** e implementar o forno como um **monitor** usando **pthreads**, **mutex** e variáveis de **condição**.

Parte da solução requer o uso de uma mutex para o forno, que servirá como a trava do monitor para as operações que as pessoas podem fazer sobre ele. Além da mutex, você precisará de um conjunto de variáveis de condição para controlar a sincronização do acesso prioritário ao forno, uma para cada pessoa/categoria.

O programa deve criar as sete pessoas (threads) e receber como parâmetro o número de vezes que eles vão tentar usar o forno — afinal, não dá para ficarmos assistindo as impressões eternamente. Ao longo da execução o programa deve então mostrar mensagens sobre a evolução do processo. Por exemplo:

```
$ ./forno 2
Vani   entra na fila          {fila:b}
Kelvin entra na fila          {fila:bC}
Vani   está usando o forno    {fila:C}
Samir  entra na fila          {fila:CA}
Vitor  entra na fila          {fila:CAB}
Vani   libera o forno         {fila:CAB}
Faxineiro detectou DEADLOCK e sorteou Samir para usar
Samir  está usando o forno    {fila:CB}
Kamila entra na fila          {fila:CBc}
Samir  libera o forno         {fila:CBc}
Vani entra na fila            {fila:CBcb}
Vitor  está usando o forno    {fila:Ccb}
Faxineiro detectou INANIÇÃO, aumentando prioridade de Kelvin
Vitor  libera o forno         {fila:C^cb}
Kelvin está usando o forno    {fila:cb}
Kelvin libera o forno         {fila:cb}
Vani está usando o forno      {fila:c}
Vani libera o forno           {fila:c}
...
```

IMPORTANTE: para viabilizar a verificação automatizada da saída do seu programa pelo professor, MANTENHA o formato das Strings conforme exemplos de saída acima ("`%s entra na fila`", "`%s libera o forno`", "`%s está usando o forno`", "`Faxineiro detectou INANIÇÃO…`", "`Faxineiro detectou DEADLOCK…`" etc).

Nota: a ordem dos eventos acima é apenas um exemplo, devendo variar entre execuções (devido ao escalonamento das threads pelo S.O.) e detalhes de implementação. Você não deve esperar executar o programa e coincidentemente obter sempre o mesmo exemplo acima, mas deve obter o mesmo resultado, ou seja, respeitar as prioridades de uso do forno (sincronização) conforme a ordem de enfileiramento, evitar inanição e resolver *deadlock*.

SUMARIZANDO:

- as regras de preferência definidas acima devem ser respeitadas;
- pessoas com uma mesma prioridade esperam entre si por ordem de chegada;
- deadlock deve ser resolvido pela atuação do Faxineiro;
- inanição deve ser evitada com o envelhecimento gradual (promoção de prioridade);

Para implementar a solução, analise as ações de cada pessoa em diferentes circunstâncias, como se outra pessoa de mesma prioridade já está esperando, se houver alguém com maior prioridade esperando, se há alguém com menor prioridade esperando mas que tenha sido promovido a nova prioridade. Determine o que fazer quando alguém já está na fila da cozinha mas surge na fila um novo alguém de maior prioridade, o que fazer quando terminam de usar o forno, 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 o forno serão parte de um monitor. Na prática, isso será implementado em C, mas deverá ter a funcionalidade equivalente a:

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

        esperar(int pessoa) {
           printf("%s está na fila da cozinha\n", nome(pessoa));
        ... // verifica quem mais quer usar, contadores, variáveis de cond., etc.
  }

        liberar(int pessoa) {
           printf("%s  libera o forno e vai estudar\n", nome(pessoa));
        ... // verifica se tem que liberar alguém, atualiza contadores, etc.
  }

        verificar() {
        ... // faxineiro verifica se há deadlock e corrige-o
  }

};
```

Cada pessoa (exceto o faxineiro) executam as seguintes operações um certo número de vezes (definido pelo parâmetro de entrada na execução do programa):

```
esperar(p);                 // exige mutex
  atendido_pelo_forno(p);   // não exige exclusão mútua
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, e faça uma multiplicação para gerar inteiros aleatórios.

Já o **faxineiro** executa as seguintes operações:

```
ENQUANTO pessoas estão ativas FAÇA:
       sleep(INTERVALO_VERIFICACAO);
       verificar();
```

INFORMAÇÕES ÚTEIS

Forma de operação

O seu programa deve basicamente criar uma thread para cada pessoa e esperar que elas terminem. Cada pessoa executa um loop um certo número de vezes (parâmetro de entrada na linha de comando), exceto o faxineiro, que deve executar seu loop até que todos as outras pessoas tenham acabado.

Codificação das pessoas

Você deve buscar produzir um código elegante e claro. Em particular, note que o comportamento de pessoas de mesma prioridade é basicamente o mesmo, você não precisa replicar o código para diferenciá-los. Além disso, o comportamento de todas as pessoas (exceto o faxineiro) é tão similar que você deve ser capaz de usar apenas uma função para todos eles, parametrizada por um número, que identifique cada pessoa. 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 uma pessoa e certifique-se de que ela funciona. Depois dispare duas pessoas apenas, para evitar que deadlocks aconteçam. Verifique se pessoas de uma mesma prioridade funcionam, inclua o faxineiro 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. DICA: economize/comente prints e sleeps). Finalmente, certifique-se que o mecanismo de prevenção da inanição funciona (p.ex., teste com apenas duas pessoas e altere os tempos das outras atividades para fazer com que um deles (o de maior prioridade) esteja sempre querendo usar o forno.

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 um arquivo .zip ou .tar.gz 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 (S.O. padrão nos laboratórios de informática) ou numa máquina virtual com o sistema operacional Linux Ubuntu versão 22.04 LTS. A avaliação do funcionamento do seu código (compilação e execução) será feita em um ambiente POSIX.

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 (Ex.: será que realmente precisa copiar e colar tantos IF/ELSE ou consegue usar arranjos e laços de repetição para verificar um controle de fluxo mais genérico?). Entretanto, dominar os princípios de funcionamento e utilização das chamadas para manipulação de variáveis de condição / mutex e conseguir a sincronização correta 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 (agende um Meet).
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.