Projeto Final de Programação Concorrente - Problema do RU

Caetano K. Fleury de Amorim

[1] *Universidade de Brasília, Departamento de Ciência da Computação, Brasília, Brasil*
[†] caetano.korilo@gmail.com

1 Introdução

Este projeto tem como objetivo a implementação de uma simulação detalhada e realista do funcionamento do Restaurante Universitário (RU) da Universidade de Brasília, empregando técnicas modernas de programação concorrente com POSIX Threads em linguagem C. O RU é um ambiente dinâmico, onde diversos tipos de usuários disputam recursos limitados, no caso, as bandejas necessárias para se servir. A proposta central é modelar, de forma fiel, a interação entre agentes com diferentes níveis de prioridade — bolsistas, estudantes regulares, funcionários e cozinheiros — e garantir que as regras de acesso e prioridade sejam rigorosamente respeitadas. O projeto busca não apenas demonstrar o funcionamento correto da sincronização entre threads, mas também proporcionar uma base sólida para o entendimento de conceitos fundamentais de concorrência, como exclusão mútua, espera condicional, prevenção de deadlocks e justiça no acesso aos recursos. A escolha pela linguagem C e pela biblioteca POSIX Threads se deve à sua ampla utilização em sistemas reais e à necessidade de controle explícito sobre os mecanismos de sincronização, tornando o estudo ainda mais próximo de aplicações práticas em sistemas operacionais e ambientes de produção.

2 Formalização do problema proposto

O problema proposto consiste em coordenar o acesso de múltiplos agentes a um recurso compartilhado — as bandejas — em um ambiente altamente concorrente, onde a ordem de atendimento e a disponibilidade dos recursos são fatores críticos para o funcionamento eficiente do sistema. No contexto do RU, a limitação do número de bandejas representa um gargalo natural, exigindo que os agentes aguardem sua vez de acordo com regras bem definidas de prioridade. Os agentes participantes são classificados em quatro categorias principais:

- **Bolsistas**: representam o grupo com prioridade máxima, refletindo políticas institucionais de atendimento preferencial. Sua presença na fila impede que outros grupos se sirvam, garantindo que sempre sejam atendidos primeiro.
- **Estudantes**: compõem o maior grupo de usuários e só podem se servir quando não há bolsistas aguardando. Isso simula o fluxo típico de um RU, onde a demanda é alta e a prioridade precisa ser respeitada para evitar injustiças.
- **Funcionários**: possuem prioridade inferior, podendo se servir apenas na ausência de estudantes e bolsistas na fila de espera, o que reforça a hierarquia de acesso.

- **Cozinheiros**: são responsáveis por repor as bandejas periodicamente, garantindo que o sistema não entre em colapso por falta de recursos. Sua atuação é fundamental para manter o ciclo de atendimento contínuo.

Além da definição clara dos grupos, algumas restrições fundamentais norteiam o funcionamento do sistema:

- Não pode haver mais agentes servindo do que bandejas disponíveis, o que exige controle rigoroso do acesso ao recurso compartilhado para evitar inconsistências e condições de corrida.
- A prioridade entre agentes deve ser estritamente respeitada, impedindo que grupos de menor prioridade furem a fila ou causem starvation em grupos preferenciais.
- O sistema deve ser robusto o suficiente para evitar deadlocks e garantir progresso contínuo, mesmo em situações de alta concorrência ou chegada simultânea de múltiplos agentes.

Essas restrições e regras refletem não apenas a lógica de funcionamento do RU, mas também desafios clássicos da programação concorrente, tornando o problema uma excelente oportunidade para aplicação e aprofundamento dos conceitos estudados em sala de aula.

3 Descrição do Algoritmo

O algoritmo utiliza mutexes e variáveis de condição para coordenar o acesso às bandejas e garantir as prioridades. Cada agente é representado por uma thread.

3.1 Variáveis Globais e Inicialização

```
#define N_ESTUDANTES 10
#define N_BOLSISTAS 3
#define N_FUNCIONARIOS 2
#define N_COZINHEIROS 2
#define N_BANDEJAS 4

int bandejas_disponiveis = N_BANDEJAS;
int bolsistas_esperando = 0;
int estudantes_esperando = 0;
int funcionarios_esperando = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_bolsista = PTHREAD_COND_INITIALIZER;
pthread_cond_t cond_estudante = PTHREAD_COND_INITIALIZER;
pthread_cond_t cond_funcionario = PTHREAD_COND_INITIALIZER;
pthread_cond_t cond_bandeja = PTHREAD_COND_INITIALIZER;
```

Essas variáveis controlam o estado do sistema e a sincronização entre threads.

Regiões críticas e prevenção de deadlocks: O acesso às variáveis compartilhadas é protegido por um mutex, garantindo exclusão mútua nas regiões críticas. As variáveis de condição permitem que as threads esperem de forma segura, liberando o mutex enquanto aguardam. Dessa forma, evitamos condições de corrida e deadlocks, pois o mutex é sempre liberado antes de operações demoradas e não há espera circular.

3.2 Thread dos Bolsistas

Os bolsistas têm prioridade máxima. Eles incrementam um contador ao entrar na fila, esperam uma bandeja livre, e só então se servem. Após comer, devolvem a bandeja e sinalizam para outros agentes, respeitando a ordem de prioridade.

```
// entra na região crítica
pthread_mutex_lock(&mutex);
bolsistas_esperando++; // sinaliza que está esperando
// espera se não há bandeja disponível
while (bandejas_disponiveis == 0) {
    pthread_cond_wait(&cond_bandeja, &mutex);
}
bandejas_disponiveis--; // pega uma bandeja
bolsistas_esperando--; // saiu da fila de espera
pthread_mutex_unlock(&mutex); // libera o mutex
servir("Bolsista", id); // se serve
\end{verbatim>

Após comer:
\begin{verbatim}
// volta pra devolver bandeja
pthread_mutex_lock(&mutex);
bandejas_disponiveis++; // devolve bandeja
pthread_cond_broadcast(&cond_bandeja); // avisa todos
if (bolsistas_esperando > 0)
    pthread_cond_signal(&cond_bolsista);
else if (estudantes_esperando > 0)
    pthread_cond_signal(&cond_estudante);
else
    pthread_cond_signal(&cond_funcionario);
pthread_mutex_unlock(&mutex);
```

3.3 Thread dos Estudantes

Estudantes só podem se servir se não houver bolsistas esperando. O padrão é semelhante ao dos bolsistas, mas com uma condição extra:

```
// entra na região crítica
pthread_mutex_lock(&mutex);
estudantes_esperando++; // avisa que está esperando
// espera se não há bandeja ou se há bolsista esperando (prioridade)
while (bandejas_disponiveis == 0 || bolsistas_esperando > 0) {
    pthread_cond_wait(&cond_estudante, &mutex); // fica bloqueado até ser sinalizado
}
bandejas_disponiveis--; // pega uma bandeja
estudantes_esperando--; // saiu da fila de espera
pthread_mutex_unlock(&mutex); // libera o mutex
servir("Estudante", id); // se serve
```

3.4 Thread dos Funcionários

Funcionários só podem se servir se não houver estudantes ou bolsistas esperando:

```
// entra na região crítica
pthread_mutex_lock(&mutex);
funcionarios_esperando++; // sinaliza que está esperando
// espera se não há bandeja ou se há alguém de prioridade maior esperando
while (bandejas_disponiveis == 0 || bolsistas_esperando > 0 || estudantes_esperando > 0) {
    pthread_cond_wait(&cond_funcionario, &mutex); // só acorda quando não há ninguém na frente
}
bandejas_disponiveis--; // pega bandeja
funcionarios_esperando--; // saiu da fila de espera
pthread_mutex_unlock(&mutex); // libera mutex
servir("Funcionario", id); // se serve
\end{verbatim>

\subsection{Thread dos Cozinheiros}

Cozinheiros repõem todas as bandejas periodicamente:

\begin{verbatim}
while (1) {
    sleep(TEMPO_REPOSICAO); // espera o tempo de reposição (simula trabalho)
    pthread_mutex_lock(&mutex); // entra na região crítica para repor
    int falta = N_BANDEJAS - bandejas_disponiveis; // calcula quantas faltam
    // Se há bandejas para repor, repõe todas de uma vez
    if (falta > 0) {
        bandejas_disponiveis = N_BANDEJAS; // repõe tudo
        printf("Cozinheiro %d repôs as bandejas\n", id); // log didático
        // Notifica todos os tipos de agentes que podem estar esperando
        pthread_cond_broadcast(&cond_bandeja);
        pthread_cond_broadcast(&cond_bolsista);
        pthread_cond_broadcast(&cond_estudante);
        pthread_cond_broadcast(&cond_funcionario);
    }
    pthread_mutex_unlock(&mutex); // libera mutex
}
\end{verbatim>

\subsection{Criação das Threads}

No \texttt{main()}, as threads são criadas para cada agente: para cada leitor e cada escritor defini

\begin{verbatim}
// estudantes
int i = 0;
while (i < N_ESTUDANTES) {
```

```
    int* id = malloc(sizeof(int)); *id = i+1;
    pthread_create(&estudantes[i], NULL, estudante_thread, id);
    i++;
}
// funcionários, bolsistas e cozinheiros seguem lógica similar
```

4 Conclusão

O projeto demonstra como técnicas de sincronização (mutexes, variáveis de condição) podem ser aplicadas para garantir justiça, segurança e eficiência em um ambiente concorrente com prioridades. O uso de contadores auxiliares e sinais condicionais permite controlar o acesso aos recursos compartilhados sem starvation ou deadlock, mesmo com múltiplos agentes e prioridades distintas.

5 Referencias

- Documentação oficial do POSIX Threads: https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html
- Material de aula da disciplina de Programação Concorrente (UnB)