UNIVERSIDADE FEDERAL DE PELOTAS
Centro de Desenvolvimento Tecnológico
Curso de Bacharelado em Ciência da Computação

Gerenciamento de Recursos em um Internet Café Futurista

Yuri de Melo Zorzoli Nunes

Pelotas, 2025

Yuri de Melo Zorzoli Nunes

Gerenciamento de Recursos em um Internet Café Futurista

Trabalho Individual de Sistemas Operacionais apresentado como requisito parcial para a conclusão da disciplina.

Orientador: Prof. Rafael Burlamaqui Amaral

Pelotas, 2025

SUMÁRIO

1 INTRODUÇÃO

1.1 Proposta inicial

O cyber café “CyberFlux”, ambiente central deste trabalho, exemplifica uma situação comum onde a alocação de recursos entre clientes de diferentes prioridades precisa ser feita de forma equilibrada. O problema a ser resolvido é garantir que todos os clientes tenham acesso aos recursos necessários para suas atividades, sem que haja a monopolização de recursos por um único tipo de cliente ou a exclusão de outros, que não conseguem utilizar os serviços devido à falta de disponibilidade.

Este trabalho tem como objetivo o desenvolvimento de um simulador que gerencie de maneira eficiente a alocação de recursos dentro do ambiente do CyberFlux, utilizando técnicas de sincronização de threads e semáforos para controlar o acesso aos PCs, headsets VR e cadeiras. Além disso, será o simulador será implementado de forma que visa evitar o deadlock. Para testar a solução implementada, sera criado de forma forçada uma situação deadlock, de modo a validar a eficácia da abordagem adotada.

Ao final, o simulador gerará um relatório detalhado com as seguintes métricas: a quantidade total de clientes atendidos, o tempo médio de espera por recurso, o número de clientes que não conseguiram utilizar os recursos e a taxa de utilização de cada recurso. Esses dados serão fundamentais para avaliar a eficiência do sistema e fornecer ideias sobre como melhorar a gestão de recursos em ambientes compartilhados.

2 DESENVOLVIMENTO

2.1 Visão geral

O sistema simula o funcionamento de um cyber café, onde diversos clientes competem por uma quantidade limitada de computadores, headsets VR e cadeiras. Os clientes se dividem em três tipos e cada possui uma prioridade de recursos que são as seguintes: gamer, com prioridade para PC e headset VR, e caso disponível, cadeira; freelancer, com prioridade para PC, cadeira, e caso disponível, headset VR; e estudante, com prioridade apenas para PC. O sistema de priorização define que o gamer só adquira a cadeira se ela estiver disponível sem tempo de espera. E o mesmo ocorre com o freelancer e o headset VR.

O algoritmo implementa uma gestão de threads utilizando semáforos para controlar o acesso aos recursos. Cada cliente é representado como uma thread. O código é projetado para rodar em um tempo fixo, simulando as operações do cyber café, e gerar ao fim do dia de trabalho do cyber café um relatório com estatísticas de uso dos recursos, tempo médio de espera de cada cliente e o numero de clientes atendidos e não atendidos.

2.2 Configurações de tempo

No código, o tempo de espera, de uso dos recursos e da chegada de um novo cliente são definidos de forma aleatória para simular a variabilidade do comportamento dos clientes. Esses tempos foram calibrados para garantir uma simulação equilibrada e realista dentro do tempo de execução total do sistema, que foi fixado em 240 segundos.

O tempo de espera simula um sistema de paciência para aguardar um recurso, esse valor é gerado aleatoriamente dentro de um intervalo de 10 segundos. Esse sistema tem como objetivo justamente entregar uma situação mais realista onde diferentes clientes terão diferentes tolerâncias em relação ao tempo de espera, gerando consequências quando um recurso não estiver disponível por muito tempo.

O tempo de uso dos recursos, que define quanto tempo um cliente terá posse

dos recursos antes de libera-los, varia aleatoriamente entre 5 e 20 segundos. Esse intervalo garante que alguns clientes use e libere recursos de forma rápida, enquanto outros podem demorar para fazer a liberação.

Além disso, o tempo de chegada entre os clientes também é aleatório, variando de 0 a 5 segundos, simulando um fluxo aleatório de clientes tornando menos previsível a execução do sistema.

Todos os tempos foram escolhidos com o objetivo de gerar uma simulação equilibrada demonstrando um bom funcionamento da gestão do cyber café. Porem demonstrando também algumas desistências ao enfrentar a falta de recursos.

2.3 Desafios

2.3.1 Monopólio e bloqueio de recursos

Na primeira versão do sistema, era utilizado `sem_wait()`, que faz com que a thread aguardando até que o recurso solicitado seja liberado. Porem, quando o cliente era gamer ou freelancer, ele solicitava um PC, e após conseguir, ele solicitava um VR, ou Cadeira no caso do freelancer. No entanto, essa abordagem apresentava um problema: um cliente poderia adquirir um PC, bloqueando-o, e não conseguindo o outro recursos necessário para continuar, impedindo que outros clientes tivessem acesso ao PC. Essa situação tinha como consequência o monopólio de clientes do tipo gamer e freelancer, que bloqueavam o PC, que é o único recurso que um estudante precisava, deixando-o sempre esperando indefinidamente por ele.

Para evitar esse problema, foi feita a troca do `sem_wait()` pela função `sem_trywait()` dentro de um loop `while` que rodará por um tempo definido aleatoriamente. O `sem_trywait()` tem como característica, não aguardar pelo recurso caso ele não esteja disponível, mas caso ele consiga, a função retorna 0. Como essa função não espera pelo recurso, foi necessário utiliza-la dentro de um loop para que o cliente tentasse novamente solicitar o recurso. Caso o cliente não consiga algum dos recursos solicitados, ele libera o recurso que conseguiu e tenta novamente.

Abaixo temos um exemplo da tentativa de um cliente do tipo gamer obter os recursos. O `while` se manterá em looping em até 10 segundos, simulando a paciência do cliente. No looping, o cliente solicita um PC, se não estiver disponível, ele apenas tenta novamente, mas caso consiga, ele bloqueia e tenta solicitar o próximo recurso, o headset VR. Se conseguir, indicamos que ele conseguiu os recursos com a flag `got_resources` e saímos do loop com o `break`. Agora, caso ele não consiga o segundo recurso, liberamos o PC usando `sem_post()` e tentamos novamente.

É usado um `usleep(200000)` para termos um intervalo de 0,2 segundos entre cada iteração da tentativa, pois sem ele, a repetição do `while` aconteceria de forma contínua e ininterrupta, consumindo muito processamento quando diversas threads estiverem

aguardando um recurso. Além disso, esse pequeno intervalo da pequena oportunidades para outras threads acessarem recursos, reduzindo a chance de starvation.

```
int got_resources = 0;
while (difftime(time(NULL), begin) < rand() % 10) {
    if (sem_trywait(&pc_sem) == 0) {
        if (sem_trywait(&vr_sem) == 0) {
            got_resources = 1;
            break;
        }
    } else {
        sem_post(&pc_sem);
    }
    usleep(200000);
}
```

2.4 Deadlock

Como citado anteriormente, na primeira versão do sistema era utilizado a função `sem_wait()` para solicitar um recurso e aguardar caso não estivesse disponível e não havia tempo limite de espera. Com isso, mesmo que baixa, existia a chance de ocorrência de deadlock, uma vez que quando duas threads fossem executadas ao mesmo tempo, em uma situação com apenas um recurso de cada disponível, uma das threads poderia bloquear algum recurso enquanto a outra thread bloquearia os outros, bloqueando-as permanentemente.

Ao decorrer do desenvolvimento do projeto, não foram detectados deadlocks por conta da baixa probabilidade de ocorrência deles pela forma que o sistema foi construído. Porem, para exemplificar, sera forçada a situação de ocorrência de deadlock a fim de demonstrar que a solução final do sistema resolve o problema com deadlock.

2.4.1 Forçando um deadlock

2.4.1.1 Função main

Primeiramente, para simular um deadlock, é necessário reduzir a disponibilidade de recursos, de modo que quando um cliente adquire um determinado recurso, outra thread adquira um recurso diferente, levando ambas a um estado de espera. Isso ocorre pois cada thread precisa dos recursos que a outra está em posse. Para criar essa situação, os semáforos do sistema foram definidos para terem apenas um recurso disponivel de cada tipo.

Feito isso, precisamos garantir que duas threads sejam executadas ao mesmo tempo e que uma thread seja do tipo gamer e a outra do tipo freelancer. Faremos

isso passando como parâmetro para a função que a thread executara o id do enum criado com os tipos, sendo 0 para gamer e 1 para freelancer. Segue abaixo a função `main` da simulação de deadlock.

```
int main() {
    sem_init(&pc_sem, 0, 1);
    sem_init(&vr_sem, 0, 1);
    sem_init(&chair_sem, 0, 1);

    pthread_create(&threads[0], NULL, useCyberCafe, (void *)0);
    pthread_create(&threads[1], NULL, useCyberCafe, (void *)1);
}
```

2.4.1.2 Função da thread

Para a função thread o mecanismo é simples, apenas precisamos garantir que uma thread bloqueie recursos necessários a outra, e vice-versa. Por isso a necessidade de termos um cliente gamer e um freelancer, pois assim definimos que o gamer adquira o PC primeiro, depois o headset VR e depois a cadeira (que nesse exemplo não tem priorização de recursos), e que o freelancer adquira primeiro a cadeira, depois o headset VR e por ultimo o PC. Foi definido também um tempo de 2 segundos entre a solicitação de um recurso e outro, para forçar ainda mais que ocorra o deadlock, pois dessa forma, o gamer sempre vai bloquear o PC primeiro enquanto o freelancer bloqueia a cadeira. Com isso, algum dos dois conseguirá o headset VR, deixando o outro cliente na espera. Porem, caso o gamer consiga o headset VR, ele vai aguardar a cadeira que está com o freelancer, porem ele está aguardando o headset VR que está com o gamer. O mesmo ocorre caso o freelancer adquira o headset VR, onde ele irá aguardar o PC enquanto o gamer está aguardando ao headset VR.

```
void* useCyberCafe(void* arg) {
    Type_Customer customer = (int)arg;

    if (customer == GAMER) {
        sem_wait(&pc_sem);
        sleep(2);
        sem_wait(&vr_sem);
        sleep(2);
        sem_wait(&chair_sem);

        usleep(2000000);

        sem_post(&pc_sem);
```

```
14          sem_post(&vr_sem);
15          sem_post(&chair_sem);
16      } else if (customer == FREELANCE) {
17          sem_wait(&chair_sem);
18          sleep(2);
19          sem_wait(&vr_sem);
20          sleep(2);
21          sem_wait(&pc_sem);
22
23          usleep(2000000);
24
25          sem_post(&chair_sem);
26          sem_post(&vr_sem);
27          sem_post(&pc_sem);
28      }
29
30      pthread_exit(NULL);
31  }
```

2.4.2 Solução

Ao resolvermos os bloqueios desnecessários de recursos com o objetivo de evitar que clientes do tipo gamer e freelancer segurassem recursos enquanto aguarda por outros, e de criarmos um sistema de paciência onde um cliente desiste de esperar, corrigimos também o problema com deadlock, pois em nenhum momento teremos recursos bloqueados enquanto uma espera ocorre, e também nunca teremos um tempo de espera indefinido, já que em até 10 segundos, o cliente que não conseguir os recursos irá desistir.

3 COMPILACAO E EXECUCAO

Para compilar e executar o simulador, siga os passos abaixo:

3.0.1 Compilação

Utilize o compilador `gcc` com a flag `-pthread` para garantir o suporte a threads:

```
gcc -o cyber_cafe cyber_cafe.c -pthread
```

3.0.2 Execução

Após a compilação bem-sucedida, execute o programa com:

```
./cyber_cafe
```

4 CONCLUSÃO

O sistema demonstrou ser uma solução eficiente para a gestão de recursos, garantindo a distribuição equilibrada entre clientes de diferentes tipos e prioridades, prevenindo monopólios e deadlock. A utilização da função `sem_trywait()` dentro de um loop foi crucial para o funcionamento do sistema para evitar todos os problemas possíveis de uma forma simples. Além disso, a calibragem dos tempos de espera, tempo de uso e chegada de clientes permitiu uma simulação realista, representando diferentes possibilidades de comportamento dos clientes.

A simulação forçada de deadlock também foi útil para validar a robustez do sistema, demonstrando a importância do gerenciamento correto da sincronização em ambientes de concorrência.

Atualmente o sistema funciona com prioridades de recursos fixas para cada cliente. Um possível aprimoramento seria a aleatoriedade de prioridades ou até um sistema de prioridade adaptável de acordo com o tempo de espera.

Uma interface gráfica também é uma melhoria possível, pois permitiria uma melhor visualização da alocação de recursos e do comportamento dos clientes.

Com essas melhorias, o sistema pode se tornar uma ferramenta mais completa no estudo de alocação e gerenciamento de concorrências.