Sistema Simulador de Controle de Acesso ao LINF

Yan Victor dos Santos[1]

Departamento de Ciência da Computação, Universidade de Brasília

***Abstract*— A pesquisa na universidade, em geral, requer uso de equipamentos técnicos para dar suporte ao aprendizado dos alunos. Em alguns casos, a quantidade de recursos disponíveis são insuficientes para atender a demanda de todos os alunos de sua graduação. Portanto, o principal objetivo deste trabalho é apresentar um sistema de simulação de acesso ao laboratório de informática (LINF) do departamento de Ciência da Computação (CiC), UnB. O sistema foi desenvolvido como sendo o produto de pesquisa e aprendizado sobre programação concorrente.**

I. Introdução

Com o avanço da tecnologia, surge a necessidade de pesquisar a respeito de áreas que dependem do uso de computadores. Para isso, grandes universidades oferecem aos seus alunos e pesquisadores laboratórios de informática para dar suporte na aprendizagem e desenvolvimentno das pesquisas. Desta maneira, o cenário ideal seria ter recursos suficientes para atender a demanda de todos os alunos ao mesmo tempo. Embora o laboratório tenha um número considerável de máquinas disponíveis, ainda assim está longe de satisfazer a necessidade em larga escala de alunos que desejam utilizá-lo.

Este trabalho tem como objetivo simular o acesso ao laboratório de informática, considerando limitações que podem ser definidas pelo próprio pesquisador antes de cada simulação. Para isto, fizemos o uso de conceitos de programação concorrente, utilizando *threads* para representar o comportamento de um grande número de alunos. A pesquisa sobre concorrência fez uso da biblioteca <*pthread*> [3], e diversos recursos por ela oferecidos.

O resto do artigo está organizado da seguinte forma. A Seção 2 fornece os conceitos e informações necessárias para a compreensão do trabalho. A seção 3 apresenta, de forma geral, o fluxo de acesso ao linf pelos alunos, ou o problema, e como a programação concorrente pode simular esse fluxo. A seção 4 detalha a simulação e a solução do algoritmo do simulador de controle de acesso. A seção 5 apresenta o sistema de simulação e sua interface. Por fim, a seção 6 mostra as conclusões sobre o trabalho.

II. Fundamentação Teórica

A. Programação Concorrente

Como afirma-se nos *slides* em [2], o avanço tecnológico e atualidade estão cada vez mais utilizando, ou requisitando, o uso do paralelismo para melhorar o desempenho. Para suprir a necessidade do paralelismo, porém com limitações de *hardware*, a programação concorrente utiliza o paralelismo lógico, entrelaçando as ações para gerar o processo simultâneo lógico. Portanto, estes conceitos são necessários para compreender o Programa Concorrente, cujos componentes podem ser abstrações úteis de objetos reais. Essa abstração fornece suporte para o sistema de simulação.

III. Formalização do Problema

A. O problema na vida real

A proposta do trabalho apresenta o seguinte problema: Imagine um laboratório de informática de uma universidade, tal como a da Universidade de Brasília, dos cursos de CiC, Engenharia e demais. O fluxo de alunos é extremamente grande se comparado a quantidade de computadores disponíveis para acesso. Para isto, como proposta, foi desenvolvido um sistema que simula a entrada e saída de alunos no laboratório, e o uso de computadores e cadeiras como sendo os recursos requisitados por eles.

Normalmente, o LINF pode presenciar diferentes tipos de comportamentos dos agentes envolvidos. Por ser aluno da UnB, além de um frequentador do LINF, os seguintes comportamentos foram os mais observados:

- Entrada e saída constante de alunos;
- Alunos sem fazer uso de computadores, sentados em cadeiras esperando por outros alunos;
- Alunos em pé costumam ficar pouco tempo no laboratório;
- Quantidade de alunos usando o mesmo computador variam de 1 à 5 pessoas, podendo ser considerados de um mesmo grupo;
- Quantidade de computadores diferente da quantidade de cadeiras;
- Quando o LINF está cheio, os alunos costumam ir embora, seja para almoçar, ir para o centro acadêmico, etc. Esperam até que o lab. esvazie. Assim, grupos podem não obter sucesso caso o grupo inteiro não consiga um lugar.

Os comportamentos citados refletem e dividem opiniões entre os alunos, que hora sentem-se satisfeitos com a qualidade de seus equipamentos, quantidade de recursos disponíveis, e hora sentem-se insatisfeitos com a grande quantidade de alunos não atendidos por falta de espaço. Por este motivo, o foco deste trabalho é dar suporte, por meio da simulação, a estudos de caso sobre acesso a LINF's.

[1]Graduando em Ciência da Computação, Universidade de Brasília. email: yanvictor_ds@hotmail.com, matrícula: 14/0033599

B. O Problema e a Programação Concorrente

Ao visualizarmos a área de estudo e atuação da programação concorrente, é possível compreender a importância dos conceitos de barreiras, semáforos e variáveis de condição [4]. Estes conceitos tornam mais realístico o processo computacional de verificar as permissões de cada *thread*, com relação a problemas da vida real. Tais permissões dão a elas características de agentes ativos, por exemplo, os alunos que só agem se uma determinada condição do meio estiver sendo satisfeita.

Um exemplo para justificar comportamento das *threads*, é a disponibilidade de recursos da região crítica [1], que gera uma condição de corrida devido a necessidade de alunos de um mesmo grupo tentarem reservar espaço para executar uma determinada tarefa, quantidade *n* de cadeiras, entre outros. Para evitar possíveis brigas ou incoerências, necessita-se de regras de convívio, tal como garantir a exclusão mútua, onde somente um grupo poderá usar o computador, permitindo que os usuários consigam utilizar o LINF como de fato deve ser utilizado.

Por este motivo, o sistema de simulação utilizou conceitos de programação concorrente (*threads*, barreiras, semáforos, etc.) para tentar tornar realística a representação do laboratório de informática da UnB.

Para explicar o funcionamento do sistema, vamos considerar alunos como sendo *threads*, computadores e cadeiras como sendo recursos da região crítica, o comportamento destes alunos tendem a gerar uma condição de corrida, fazendo assim com que seja necessário utilizar ferramentas da *pthread.h* [3] para controlar o acesso e garantir a exclusão mútua. Desta maneira todos os grupos tendem a utilizar o computador sem grandes problemas durante a simulação.

IV. Algoritmo do Simulador

A. Explicando o problema na simulação

Uma vez que o problema real foi explicado na seção anterior, é necessário limitarmos o cenário para que seja possível computar o comportamento das *threads*. O motivo da limitação deve-se ao fato de que existem infinitos comportamentos distintos que alunos podem ter ao tentar acesso ao laboratório. Por isso, baseando-se nas experiências genéricas dos próprios alunos da universidades, vamos considerar as seguintes regras de comportamento, ou regras de convívio, para a simulação:

1) Um computador não pode ser usado por mais de um grupo ao mesmo tempo;
2) Uma cadeira não pode ser usada por mais de um aluno ao mesmo tempo;
3) Um grupo pode e deve usar um único computador;
4) Um grupo só pode utilizar um computador se todos os alunos conseguirem cadeiras para sentar;
5) Caso o grupo inteiro não consiga cadeiras suficientes para sentar, então o grupo deve dar um tempo e voltar mais tarde para verificar se o LINF está mais vazio;
6) Grupos incompletos podem ocupar cadeiras enquanto esperam por parceiros do grupo, embora não possam utilizar o computador para adiantar o objetivo. Isso quer dizer que o computador deve ficar disponível para outros alunos;
7) Grupos de alunos contém tamanho varíavel, deste 1 membro (individual) até um número *n* grande;
8) Por fim, ao terminar de usar o computador, o grupo deve avisar aos outros alunos que suas tarefas foram concluídas, liberando assim para que outros alunos possam utilizar os novos recursos disponíveis.

Repare que a ordem dos grupos pode ser aleatória, simulando a falta de padrão de acesso dos alunos, que podem usar o laboratório independente da ordem e quantas vezes quiserem, bastando ter os equipamentos disponíveis para o uso.

As regras acima tentam se aproximar do realismo (mesmo que simples) de comportamento dos agentes, como descrito na sessão anterior, baseando-se em comportamentos observados por alunos da graduação de CiC, UnB. O sistema deve ser apenas um suporte de análise para diversos fins. O que permite seu potencial, é o fato das seguintes configurações de simulação serem variáveis:

- Quantidade de alunos: QTD_ALUNOS;
- Quantidade de Cadeiras: QTD_CADEIRAS;
- Quantidade de computadores: QTD_COMPUTADORES;
- Máximo de alunos por grupo: MAX_POR_GRUPO;
- Tempo de uso das máquinas por grupos: TEMPO_USO_PC.

As variáveis em maiúsculo citadas acima podem ser alteradas diretamente no código, por meio de *define's*.

Uma vez que as regras de convívio foram definidas, vamos explicar o funcionamento do sistema, começando sobre como os grupos estão sendo formados.

B. A formação de grupos

Para cada *thread* de simulação criada, representando cada aluno, é gerada aleatoriamente a decisão de colocar ou não a *thread* atual dentro do grupo de *threads*, também atual. Quando a aleatoriedade indicar o "não", então o aluno deve ser inserido em um novo grupo. Caso indique o "Sim", então ele deve ser inserido dentro do grupo corrente no momento. O resultado final é criação randômica de diversos grupos de alunos.

A quantidade máxima de alunos em cada grupo varia de acordo com as configurações definidas. Reforçando que a criação depende diretamente das configurações definidas, mas sempre é aleatória para cada execução do programa, com o objetivo de respeitar a complexidade e imprevisões da vida real.

Uma vez que as regras de convívio do LINF para a simulação estão claras, as configurações foram definidas e os grupos estão formados, é hora de apresentar como o sistema lida com a condição de corrida.

C. A Ideia do Algoritmo

Baseando-se nas regras, vamos apresentar um fluxo de ações de alunos tentando acessar computadores do LINF, desde a chegada nele até o uso propriamente dito. Baseando-se na vida real, os passos foram definidos da seguinte maneira:

1) O aluno chega no linf e tenta pegar uma cadeira (*sem_trywait*). Caso não consiga, vai dar um passeio (*variáveis de condição*). O aluno avisa para o grupo, forçando os outros a darem um passeio junto com ele, liberando as cadeiras (*sem_post*). Entretanto, caso o aluno consiga pegar a cadeira, então deve esperar o resto do grupo chegar (*barreira*), para ver se todos conseguem cadeiras ou não;
2) Caso o grupo inteiro tenha conseguido pegar cadeiras, então tentarão pegar um único computador (variável comum). Caso o grupo não consiga, vão dar um passeio (*variáveis de condição*) e liberam as cadeiras ocupadas. Caso consigam, então pegam um computador e começam a utilizá-lo;
3) No término do uso, avisam aos demais alunos (cond_broadcast) que estão liberando cadeiras (sem_post), e o computador (tem_computador++).
4) Ao receberem a mensagem, alunos que foram dar um passeio vão imediatamente ao LINF tentar pegar cadeiras novamente. Por fim, retorna ao passo 1.

A solução é simples, e descreve bem o fluxo genérico de tentativa de acesso dos alunos. Porém, para problemas gerais da programação concorrente, foi necessário isolar algumas variáveis para impedir possíveis inconsistências.

D. Solução detalhada sobre casos específicos

Para evitar inconsistências durante a simulação, como citado anteriormente, foi usado o *lock* tradicional para permitir atribuições e acessos a dados por *threads*. Isto evita desatualização de valores. Alguns problemas específicos tiveram de ser solucionados adicionando *locks* locais. Por exemplo:

Problema 1: assemelha-se ao problema do Jantar dos Filósofos, onde nenhum grupo inteiro consegue sentar nas cadeiras, tornando todas elas indisponíveis, porém com nenhum acesso ao computador. Desta maneira, segundo o algoritmo, então todos os grupos deveriam dar o passeio, não havendo ninguém para enviar o sinal de *broadcast* para avisar que eles podem voltar do passeio. Para corrigir este problema, foi decido que o grupo só deve dar um passeio se, e somente se, existir pelo menos um grupo utilizando um computador. (*lock usado: lock_espera_tempo*)

Problema 2: grupo verifica que não há mais cadeiras disponíveis, conclui que tem pelo menos um grupo usando o computador (*Problema 1*) e então vai passear. Antes de ir realmente passear, então acontece o caso do último grupo liberar o computador e avisar que está saindo do LINF. Porém, como o grupo ainda não foi passear, mesmo concluindo que pode ir, o sinal foi recebido antes, não tendo efeito algum. Este grupo pode nunca mais voltar ao LINF. A resolução foi feita em conjunto à solução do problema anterior, definindo que as *threads* só devem passear ou mandar o sinal de maneira exclusiva, nunca os dois ao mesmo tempo. Portanto, as verificações condicionais necessárias sempre serão feitas no tempo certo. (*lock usado: lock_espera_tempo*)

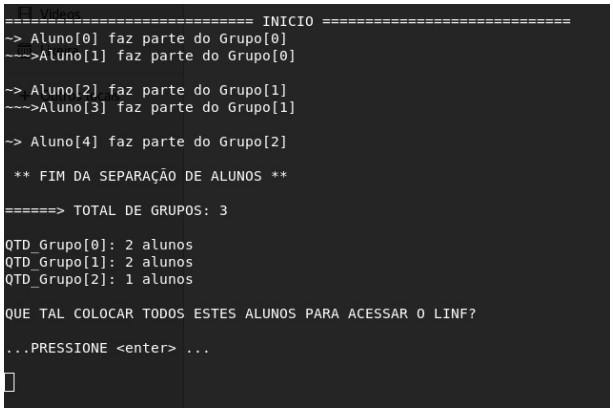

Fig. 1. Processo de Criação Randômica de Grupos

Problema 3: A atualização dos dados na saída dentro da interface do usuário pode ficar completamente inconsistente, caso a busca da informação demore mais que a escrita simultânea de alguma *thread*. Isto implicará em impressão de valor desatualizado. Para isso, foi necessário *"lockar"* todas as variáveis sensíveis. (*lock usado: lock_print*)

Problema 4: Como um grupo inteiro de diversos alunos deverá usar apenas um computador, não faz parte da solução deixar todas as threads pegarem um computador para si (tem_computador–). Para isto, elege-se um representante sempre que o grupo completo consegue sentar, e ainda há pelo menos um computador disponível. O representante é responsável por garantir o uso dos recursos, e permitir com que o grupo todo possa usufruir do recurso também.

(*lock usado: lock_pc*)

Todas as variáveis, que tem relação direta com grupos, está sendo tratada como ponteiro para alocar memória dinâmicamente, adicionando variabilidade de configurações e quantidades de recursos necessários para cada simulação. Algumas variáveis auxiliares foram criadas para melhorar a visualização da saída, como sendo infomrações auxiliares (Ex.: tamanho do grupo, cadeiras por grupo, etc.).

V. O SISTEMA

A. Interface com Exemplo

A Figura 1 apresenta o início do sistema, indicando o resultado final da criação dos grupos de acordo com as configurações definidas dentro do código. Podemos observar como o algoritmo escolheu a criação dos grupos. Nesta simulação, foram criados 3 grupos para 5 alunos, dois contendo 2 alunos, e um contendo apenas 1 aluno. Outras possibilidades poderiam ser a criação de 5 grupos de 1 aluno, 4 grupos com um grupo contendo 2 alunos, e assim por diante. Ou seja, todas as combinações possíveis.

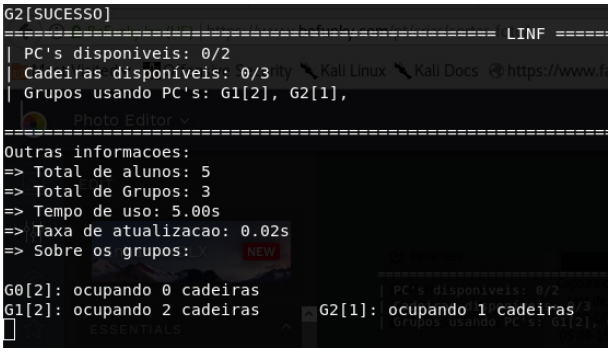

Fig. 2. Simulação em Tempo Real

A simulação contém informações básicas de como cada grupo está se comportando em tempo real. Por exemplo:

- Total de alunos: definido nas configurações;
- Total de Grupos: definido aleatoriamente;
- Tempo de uso: definido nas configurações, indica quanto tempo (ms) cada grupo usará o computador;
- Taxa de atualização: Quanto tempo (ms) leva para atualizar o *print*;
- Sobre os Grupos: Informações gerais sobre cada grupo.
- Pc's disponíveis: definido nas configurações. Temos a/b, onde 'a' é a quantidade disponível e 'b' a quantidade total;
- Cadeiras disponíveis: definido nas configurações. Temos a/b, onde 'a' é a quantidade disponível e 'b' a quantidade total;
- Grupos usando PC's: Grupos que conseguiram cadeiras e computador;
- Informações superiores: Informações individuais sobre cada grupo ou alunos específicos.

Para uma melhor experiência, busque aumentar a taxa de atualização para casos grandes, e reduzir para casos pequenos. Caso queira ver diversos grupos usando computadores ao mesmo tempo, aumente a quantidade de uso de cada grupo por computador.

Notação: Ga[b], indica que o Grupo de número 'a' contém 'b' alunos.

B. Como executar

Podemos compilar e executar o sistema utilizando o programa principal com os seguintes comandos:

- gcc nomeMain.c -o main -pthread
- ./main

Estes comandos rodam em ambiente linux. Para Windows, basta compilar utilizando o formato padrão da IDE desejada, para linguagem C. Porém, a entrada da função System deve ser "cls|clear", para suportar o ambiente Windows também.

VI. Conclusão

O desenvolvimento deste projeto permitiu melhorar a compreensão sobre programação concorrente, seus conceitos e sua utilidade para resolver problemas do cotidiano. Por mais que o sistema desenvolvido não seja necessariamente uma solução, pôde-se criar noção do quanto são poderosas as ferramentas deste paradigma. Uma vez que é complicado executar processos de forma paralela por limitações de *hardware*, a programação concorrente entra como solução para diversas áreas, dentre elas, a simulação de casos. A capacidade de alternância entre processos gera efeito semelhante à programação paralela, em conceitos lógicos.

Com este trabalho, tive a intenção de utilizar o máximo de conceitos vistos na disciplina com o objetivo de analisar melhor a funcionalidade de cada um, e como eles interagem quando são inseridos em conjunto em um mesmo sistema. Pode-se concluir que cada ferramenta possui grande utilidade para satisfazer ideias simples de algoritmos. O uso de semáforos, variáveis de condição, barreiras e *locks* contribuíram juntos para o desenvolvimento coerente do algoritmo proposto, embora não fosse necessário usar todos eles. A escolha foi apenas para fins didáticos.

A saída em terminal do sistema demonstra o quanto a solução simula, dependendo das configurações, de forma "realista"a tentativa de acesso dos alunos aos recursos do LINF. A saída e a sua constante atualiazação é coerente, principalmente se for observada para casos simples. Portanto, o simulador acaba sendo um grande auxílio para quem deseja testar casos específicos de controle de acesso, não só a laboratórios, mas a diversas outras situações semelhantes. Isso demonstra o quanto programação concorrente pode auxiliar na satisfação dessas necessidades.

References

[1] Prof. Eduardo Alchieri, Slide: "Condições de Corrida, Regiões Críticas e Exclusão Mútua". Programação Concorrente.
[2] Prof. Eduardo Alchieri, Slide: "Introdução". Programação Concorrente.
[3] Blaise Barney, Lawrence Livermore National Laboratory, "POSIX Threads Programming". Fonte: "https://computing.llnl.gov/tutorials/pthreads/"
[4] Prof. Eduardo Alchieri, Slides: "Locks", "Variáveis Condição", "Semáforos", "Barreiras". Programação Concorrente.