Universidade de Brasília - Campus Gama
Engenharia de Software
Disciplina: Estrutura de Dados e Algorítmos
Professor: Fernando William Cruz
Alunos: Luan Guimarães Lacerda (12/0125773) e Vinícius Franco (09/0135032)

Relatório
1º Projeto de Pesquisa

Controle de Memória com Listas

Terça-feira, 05 de Maio de 2015
Gama-DF

1. Descrição do Problema

Para fixar os conhecimentos relacionados às estruturas de dados, mais precisamente, resolução de problemas utilizando listas encadeadas, foi requisitado este trabalho, que consiste em utilizar os conceitos adquiridos na primeira parte da disciplina Estrutura de Dados e Algoritmos para desenvolver uma simulação do gerenciamento de memória de um computador em linguagem C.

O problema pode ser melhor definido a partir do momento em que se compreende as rotinas básicas de inserçao e remoção de nós em uma lista encadeada. Levando em consideração que cada nó da lista representa um espaço bem definido dentro da memória, a alocação e remoção desses nós devem seguir critérios baseados na aritmética dessas posições, ou seja, determinado nó ocupará apenas um e, somente um, intervalo aritmético dentro do espaço total da memória.

O gerenciamento dos nós deve ser mantido por um cabeçalho, estrutura que manterá informações importantes sobre o conjunto dos elementos. Esse cabeçalho conterá apontadores para o início da lista, para o primeiro buraco disponível e para o primeiro processo em execução (essa abordagem sera descrita adiante), além de espaço disponível, espaço em uso, quantidade de processos em execução, etc.

Como foi explicitado no enunciado do trabalho, a inserção de um processo na memória deve ser feito sempre no primeiro buraco disponível, entretanto, devido a fins simplesmente didáticos e, com a autorização prévia do professor, os alunos optaram pela ampliação do escopo de projeto, implementando outros tipos de inserção que vão além da abordagem solicitada.

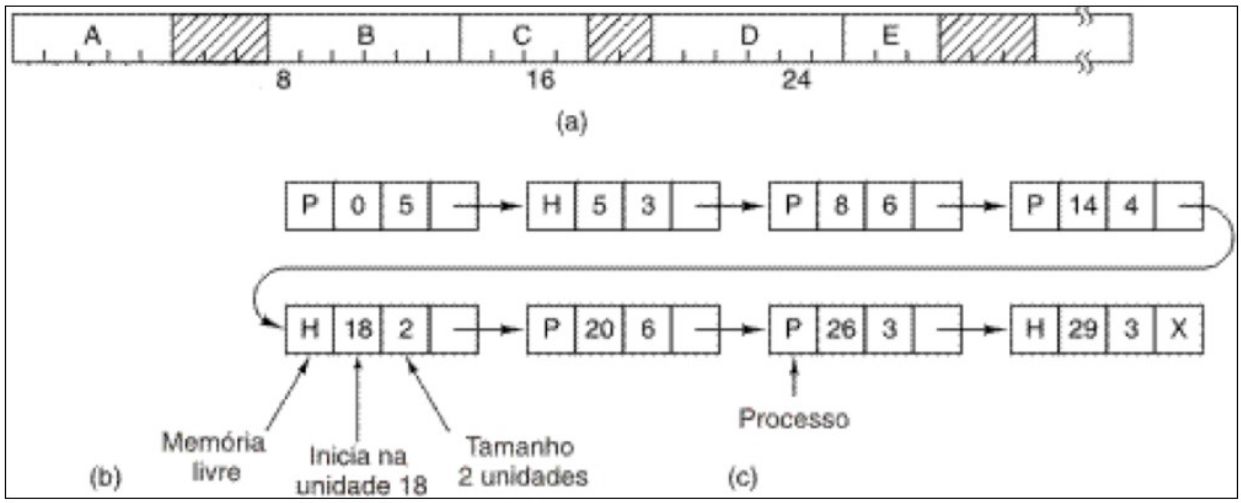
Outra premissa do problema é que dois buracos não podem coexistir lado-a-lado, isso significa que caso haja uma situação na qual dois ou mais buracos se encontrem imediatamente um após o outro, todos devem se unir em um único nó cujo tamanho e determinado pela soma dos tamanhos de cada nó unido. Essa situação pode ser observada quando, entre dois buracos, um processo é finalizado. Isso significa que o resultado será um único buraco e não três. Essa metodologia de união evita, em boa parte, a existência de múltiplos buracos excessivamente fragmentados dentro da memória.

Um ponto que deve ser resaltado é que, se houver espaço disponível para a alocação de um processo, mesmo que esse espaço se encontre fragmentado entre os buracos espalhados na memória, a reorganização dos processos dentro da mesma deve ser realizada, liberando um espaço com o tamanho necessário para alocar o novo processo.

Além disso, fornecer uma experiência visual para a simulação em softwares é um fator determinante para a compreensão do que está acontecendo dentro do programa. Portanto, uma das partes fundamentais do problema gira em torno de, com o auxílio de uma bibliotéca gráfica da linguagem C, implementar uma interface de interação com o usuário que permita o acompanhamento em tempo real da dinâmica de inclusão e exclusão de processos.

Adicionalmente, o grupo optou por uma simulação mais fiel ao que realmente ocorre em um computador, isto é, utilizou técnicas de programação concorrente como threads, semáforos e fechaduras, realizando testes entre os diferentes tipos de inserção e realizando um estudo introdutório sobre processamento paralelo.

A imagem abaixo ilustra como deve se comportar a estrutura da lista, sendo a área rachurada os espaços vazíos.



2. Metodologia

Para a realização deste trabalho primeiramente foi feito um estudo de como a memória se comporta, para depois ser possível elaborar uma abordagem para resolver o problema.

Após serem estabelecidos os pilares do projeto -- tipos de inserção, exclusão, busca -- o grupo dividiu as tarefas em duas frentes:

1. Estrutural: responsável pela elaboração da parte conceitual do projeto, ou seja, algoritmos de busca, inserção, remoção e criação de processos, manutensão da memória, além da implementação e controle dos processos concorrentes no sistema (Luan Guimarães).
2. Interface Gráfica: responsável por estudar bibliotecas gráficas, selecionando a que correspondesse mais adequadamente os requisitos do projeto. Nessa frente, foram elaborados os algoritmos de conversão dos dados obtidos da parte estrutural para que fosse possível fornecer uma experiência visual para os usuários (Vinícius Franco).

Para garantir o pleno desenvolvimento do projeto, de modo que os dois membros da equipe acompanhassem todas as etapas de construção do software e, além disso, diposessem de uma ferramenta poderosa de controle de versão, utilizou-se o Git e o repositório remoto Github.

O link para o repositório está disponível desde o dia 12 de Abril de 2015: https://github.com/luanguimaraesla/MemoryProcessSimulator.

3. Solução

3.1 O primeiro projeto

A primeira abordagem realizada pelo grupo consistia na implementação de um simples sistema no qual o usuário pudesse definir o tamanho da memória e, com o auxílio de um menu, conseguisse criar e excluir processos dentro de uma lista duplamente encadeada.

Essa abordagem consistia na implementação de uma lista encadeada heterogênea, na qual, um nó pudesse agregar uma estrutura do tipo processo ou buraco. Cada nó, além de conter esse o ponteiro (void *), armazenaria o início e o espaço ocupado dentro da memória e apontadores para os nós anterior e posterior. No programa, esse nó é chamado de MemoryCase.

Cada estrutura processo conteria um identificador para diferenciar um processo de outro, e dois ponteiros, para o próximo e anterior MemoryCases que contessem um processo. Essa dinâmica facilitaria quando houvesse a necessidade de se buscar um processo ativo na memória, já que os nós do tipo processo formariam uma lista encadeada secundária.

Os MemoryCases do tipo buraco seguiam a mesma lógica, cada estrutura buraco conteria apontadores para os buracos imediatamente vizinhos, isso permitiria uma busca muito mais veloz, já que percorrendo essa lista secundária, não haveria a necessidade de se verificar processos.

A adição de um processo se dava manualmente pelo usuário, o qual poderia digitar a quantidade de espaços que o mesmo ocuparia na memória. Então, uma subrotina era invocada, buscando o primeiro buraco que tivesse o tamanho necessário para conter esse processo. Caso os buracos, individualmente, não fossem suficientemente grandes para receber essa alocação, o programa escolhia o maior espaço disponível e deslocaria para frente os processos seguintes até que o espaço criado fosse bastante para conter o novo processo.

A remoção de um processo, assim como a inclusão, era feita manualmente. O usuário digitava o identificador do processo e, uma rápida busca na lista secundária de

processos devolveria o endereço de memória do MemoryCase a ser excluído. Então, seguindo uma rotina comum de remoção em listas encadeadas, esse nó era removido da lista de processos e adicionado a lista de buracos. O ponteiro (void *) que apontava para uma estrutura processo, logicamente, passaria a apontar uma estrutura do tipo buraco.

Como dito na introdução deste relatório, dois ou mais buracos não podem coexistir lado-a-lado, então, sempre que o próximo elemento na lista encadeada da memória fosse o mesmo elemento da lista encadeada secundária de buracos, esses dois nós deveriam ser unidos em um só nó que tivesse tamanho igual a soma dos entes unidos.

Se houvesse um estouro de memória, isto é, quando o usuário solicitasse que um novo processo fosse inserido e não tivesse espaço disponível, a inclusão desse nó seria descartada, sem mais prejuízos para a experiência do usuário.

3.2 O segundo projeto - Paginação de processos

Notou-se, durante a implementação e nas fases finais de testes estruturais que, quando a memória se encontrava em um estado muito segmentado depois da inclusão e exclusão de múltiplos nós, fazendo com que existissem vários buracos espalhados entre processos, a busca por um espaço livre era extremamente custosa e, poderia ser agravada caso não fosse encontrado um buraco de tamanho necessário para a inclusão de um processo, já que alterações em um conjunto, muitas vezes grande, de nós, deveriam ser feitas.

Uma nova abordagem experimental foi facilitada pelo modo como o primeiro protótipo havia sido desenvolvido -- estruturas processo formavam uma lista encadeada secundária.

A idéia foi baseada no insight de que um processo poderia ser segmentado em várias partes, alocando cada parte em buracos disponíveis ao longo da memória. Isso se tornou possível graças ao uso de um novo paradigma -- processos são cabeçalhos.

Um processo não mais seria um simples nó e sim, um cabeçalho cujos elementos fossem os próprios nós dentro da memória. Ou seja, seria um cabeçalho que agregasse diversos nós do tipo processo, os quais conteriam apontadores para os outros nós formadores do mesmo processo.

Percebeu-se a necessidade desses cabeçalhos possuírem apontadores para outros cabeçalhos do tipo processo, de forma que fosse possível percorrer processos com velocidade.

O gerenciamento dessa lista encadeada de cabeçalhos seria feito por um cabeçalho pai, o qual manteria informações sobre a quantidade de processos, espaço em uso e apontadores para o início e fim dessa lista.

O tempo de busca de um espaço livre diminuiu consideravelmente. O processo poderia ser segmentado em várias partes, quando o primeiro espaço era encontrado, ele era preenchido com parte do processo e retirado da lista de buracos, posteriormente, o restante era alocado no próximo buraco e assim por diante. Os elementos formadores de um processo eram unido por apontadores. Então, um cabeçalho era criado com informações e apontadores para essa sub-lista e adicionado a lista de cabeçalhos processo que, por último, era gerenciada por uma estrutura pai.

A remoção de um processo se manteve semelhante à do programa anterior. Adicionalmente, o cabeçalho processo deveria ser excluído da lista de cabeçalhos e então, a função de remoção, que antes era aplicada a somente um nó do tipo processo, passou a ser aplicada sequencialmente a todos os nós contidos no cabeçalho removido.

Testes realizados utilizando a biblioteca time.h calcularam o tempo da inserção e remoção de um grande conjunto de processos. O tempo gasto com essa nova abordagem teve um desempenho de 3% a 8% melhor, dependendo do nível de segmentação dos espaços livres na memória.

Entretando, por não ter sido projetado do zero, possíveis otimizações para esse tipo de paradigma que poderiam ser feitas, podem, se implementadas, aumentar ainda mais a diferença de desempenho.

3.3 O terceiro projeto - Threads de processos

Antes da entrega do projeto acima, o grupo foi orientado não utilizar o paradigma da paginação de processos por não representar uma solução que atendesse as solicitações do trabalho. A utilização da ferramenta Git, possibilitou a retomada do projeto em um ponto inicial que já continha funções e estruturas básicas.

Nessa nova abordagem, decidiu-se retirar o controle da memória por parte do usuário. Este definiria apenas alguns parâmetros de inicialização e então, com o auxílio de threads, todo o processo de inclusão e remoção de processos seria automatizado.

Optou-se, para fins didáticos, por construir um programa que permitisse a utilização de diferentes tipos de inserção (best-fit, worst-fit e first-fit). O tipo first-fit já estava praticamente pronto e a implementação consistiu em modificar a função de inclusão existente para incluir o processo sempre na primeira posição disponível, deslocando para frente a posição dos nós seguintes quando não houvesse espaço suficiente.

É importante que fique claro, para a compreensão dos algorítmos de inserção, remoção e busca que a lista encadeada deixou de ser duplamente encadeada e passou a ser duplamente circularmente encadeada, aderindo ao projeto uma nova gama de possibilidades, como a união de buracos do início e final da memória, assim

como a existência de um processo em um intervalo que comece no final da memória e termine no início.

Os algoritmos das inserções worst-fit e best-fit estão minuciosamente detalhados nos comentários do código e não serão explicados detalhadamente aqui. Para uma compreensão superficial, esses algorítmos decidem em que posição encaixarão novos processos a partir da verificação do módulo da diferença entre o tamanho do novo processo e o tamanho de cada buraco disponível na memória. Isso significa que uma inserção best-fit procura um buraco cujo tamanho seja o mais próximo possível do espaço necessário para inserir o novo processo, ou, em termos matemáticos, que o módulo da diferença desses tamanhos seja o menor possível. Uma inserção worst-fit é exatamente o contrário.

A forma como processos eram removidos permaneceu a mesma dos programas anteriores, a mescla de buracos vizinhos também. Como a nova abordagem conta com uma lista encadeada circular, algumas modificações de verificação de limites foram necessárias.

A essência dessa nova aplicação se dá pela autonomia do simulador em relação a interações desnecessárias com o usuário. Em um sistema computacional não temos o controle dos processos que são criados e de seu tempos de execução e, por esse motivo, o grupo decidiu automatizar esse processo tomando apenas alguns parâmetros de entrada como controladores de fluxo de criação e tempo de execução.

Cada processo passou a conter uma thread a qual era inicializada durante a criação do mesmo. Essa thread, para simular o tempo de execução de um processo real, toma um valor randômico de tempo e espera até que esse valor seja zerado -- queima de ciclos de relógio -- para só então encerrar o processo chamando a própria função de remoção.

Logicamente, múltiplos processos sendo criados e removidos são um problema em termos de acessibilidade e coerência dos dados, especialmente trabalhando com uma taxa veloz de alteração de ponteiros em uma lista. O fato é que a lista não pode ser acessível a todos as threads ao mesmo tempo, pois uma modificação inesperada poderia resultar em uma falha de segmentação, por exemplo.

Técnicas de programação concorrente foram utilizadas. Semáforos e fechaduras restringiram o acesso a lista de memória para apenas uma thread por vez. Um fato negativo em relação a essa abordagem de controle de threads é que o tempo de execução do programa, obviamente, aumentou consideravelmente.

A interface gráfica foi implementada utilizando a biblioteca xlib.h. Sempre que o programa é inicializado cria-se uma janela e, de acordo com o tamanho da memória fornecido pelo usuário, os quadrantes da exibição são repartidos por unidade de tamanho.

Dados como número de processos sendo executados, o total de processos registrados, assim como, espaço disponível e espaço em uso são informações úteis exibidas pela janela de visualização.

Para compilar e executar o programa, leia o arquivo README.MD.

4. Análise do Projeto

A equipe se sentiu motivada a realizar um trabalho que fosse consistente, para tanto houveram consultas ao professor, como também pesquisas individuais com relação às necessidades do projeto. Uma das dificuldades a principio foi entender como deveria funcionar o projeto em si, e então planejar as características que o mesmo deveria portar.

Um problema encontrado também foi a questão da biblioteca gráfica, uma vez que para uma mostragem representativa da lista não estava sendo possível encontrar bibliotecas que fossem simples e diretas. Foram testadas as bibliotecas SDL, OpenGL (glut), LibGraph, e algumas outras, porém somente com a biblioteca Xlib, nativa do Ubuntu, foi possível solucionar o problema. As bibliotecas gráficas testadas possuiam, algumas delas, a necessidade de rodarem na Thread principal, e muitas vezes necessitarem de um loop interno para realizar sua atualização de interface (OpenGL) ou por incompatibilidade com o Ubuntu (caso da LibGraph), mas a Xlib foi capaz de solucionar o problema com simplicidade e robustez, não sendo necessário muitas linhas de código, e podendo manter uma aparência agradável da representação da lista.

A manipulação e o aprendizado sobre threads foi de grande valia para os membros do grupo, introduzindo uma noção básica do funcionamento assíncrono de processos dentro de uma memória.

Adicionalmente, os testes de velocidade para os diferentes tipos de inserção proporcionaram uma importante experiência didática, evidenciando a complexidade de partes essenciais do sistema operacional e seus algoritmos.

Alguns gráficos foram preparados para demonstrarem a diferença entre os tipos de inserção, porém, minutos antes da entrega, nos últimos testes, um erro de estouro do tamanho foi encontrado, permitindo que vários processos fossem adicionados além dos limites da memória. O erro foi rapidamente corrigido, porém, esse fato comprometeu a veracidade dos dados recolhidos e, os valores encontrados e já mensionados podem estar gravemente adulterados.

Entretanto, por ser apenas um adicional de projeto, decidimos realizar novos testes e apresentar as soluções em um momento oportuno para discussão, principalmente sobre as técnicas de paginação de processos, que, de longe, foi o elemento que mais despertou curiosidade no decorrer do projeto.

5. Estrutura de Dados e Algorítmos

Devido a quantidade de funções e estruturas, e ao fato, do projeto ser livre a programadores no Github, optou-se por escrever essa parte do relatório como comentários diretamente no código. Duas versões estão diponíveis, a oficial está totalmente modularizada e, apesar de facilitar a compreensão do código em si, dificulta a leitura de todos os comentários escritos. Então, criou-se uma versão do programa com todas as funções e estruturas implementadas em apenas um arquivo, este está disponível na pasta Adicionais, juntamente com o programa de paginação de processos o qual, apesar de funcionando, ainda necessita de algumas correções e testes.