Trabalho Prático 5: Jantar dos Filósofos

Alunos em Graduação da Universidade Federal de Ouro Preto do curso Ciência da Computação:

Halliday Gauss Costa dos Santos.

Guilherme Augusto de Deus Marciel.

Matrículas: 18.1.4093.

18.1.4172

Área: Sistemas Operacionais.

Introdução:

O sistema operacional é responsável pela interface do hardware facilitando na manipulação do mesmo através dos programas do usuário. Portanto, o funcionamento do processador, que é uma peça chave para o funcionamento de uma máquina, é de responsabilidade do sistema operacional.

Uma máquina multinível possui diversos processadores e um sistema operacional que é responsável por alocar tarefas para cada processador. Essas tarefas são chamadas de processos, ou instâncias de programas. Como o número de processadores são limitados, há a necessidade de escalonar os processos, ou seja, indicar qual é o próximo processo a ser executado. O sistema operacional pode utilizar diversos algoritmos para realizar o escalonamento.

No entanto, esse não é o único problema em manipular processos, quando dois ou mais processos tentam acessar uma mesma região de memória (Região Crítica), pode haver um problema de race condition (Condição de Corrida), e o resultado dos processos envolvidos podem ser diferentes do esperado. Para resolver esse problema é necessário utilizar a exclusão mútua, que nada mais é que boquear um processo temporariamente de utilizar um determinado recurso compartilhado. O jantar dos filósofos é um problema que envolve race condition e outro problemas que serão citados a seguir.

Desenvolvimento:

A exclusão mútua resolve o problema de race condition, para isso um algoritmo deve impedir que dois ou mais processos acessem uma região crítica ao mesmo tempo. Quando um processo está utilizando uma região crítica, outro processo que quer tentar acessar a mesma região é colocado para dormir no fim de uma fila de processos que tem o mesmo objetivo. Assim que o processo sair da região crítica, o primeiro da fila entra no seu lugar e começa a acessar a mesma região.

Contudo, mesmo aplicando a exclusão mútua problemas ainda podem ocorrer. Quando um processo fica muito tempo ocioso, ou seja, está a muito tempo na fila de espera para entrar na região crítica, visto que o processo que executa a região especial está demorando muito, ocorre o problema chamado inanição (Starvation) neste caso é dito que o processo “morre de fome”. Outro problema acontece quando um ou vários processos precisam que outro(s) processo(s) termine(m) sua execução para começar a executar, porém o(s) outro(s) processo(s) também precisa de um recurso daquele processo que está esperando, assim nenhum dos processos envolvidos neste problema será executado pois ficarão esperando um recurso que nunca chegará. Esse problema é chamado de DeadLock.

Cabe ressaltar que os problemas supracitados podem envolver processos pesados(fork), ou os processos dentro de um processo (threads). Como as threads ficam dentro de uma mesma área de memória de um processo pesado, então o acesso a uma região crítica, ou a comunicação entre threads pode ser feita diretamente, pois ao criar uma thread não é gerado uma cópia das variáveis. O contrário acontece com o Fork, pois na operação de criação de um processo pesado é criado um clone do processo que chamou a operação Fork, juntamente com a cópia das variáveis, e cada nova variável copiada estará em uma posição de memória diferente do processo “pai”. É possível realizar a comunicação entre processos pesados através do PIPE, e essa comunicação é chamada de IPC (Inter-Process Communication). Como pode haver uma comunicação entre processos pesados, então os problemas supracitados também podem ocorrer. Por exemplo, dois ou mais processos pesados escrevendo e lendo em um mesmo arquivo ao mesmo tempo poderá resultar em uma leitura ou escrita incorreta.

O uso de mutex e de semáforos implementam soluções de exclusão mútua resolvendo o problema de race condition, inanição e DeadLocks, no entanto a implementação deve ser cuidadosa para que principalmente os dois últimos problemas não ocorram.

O semáforo binário resolve os problemas supracitados com o uso de uma variável atômica que pode assumir o valor 0 ou 1. Cada recurso que irá ser compartilhado entre processos deve ter uma variável semafórica que inicialmente recebe 1, indicando que o recurso está disponível para uso, assim que o recurso for utilizado o valor da variável 1 é alterada para 0 e o os processos que tentam acessar

esse recurso são colocados para dormir numa fila. Após a utilização desse recurso a variável semafórica tem seu valor alterado para 1 e o primeiro processo da fila é acordado para utilizar o recurso. Com o uso de semáforos também é possível limitar a quantidade de threads que acessam um recurso.

O uso do Mutex é bem parecido com o uso de semáforos, porém o Mutex utiliza uma variável atômica Lock que pode valer 0 ou 1 para cada recurso que será compartilhado. No Mutex também, os processos podem ou não ser colocados para dormir, portanto podem ficam a todo momento verificando se o recurso está disponível, gerando um overhead mas é mais fácil de implementar.

O problema do Jantar dos filósofos consiste em ‘n’ filósofos, tal que n > 1, sentados numa mesa redonda, onde existe um prato de macarrão para cada filósofo e um hashi na esquerda e direita de cada um. Para comer o macarrão um filósofo precisa parar de pensar e pegar os dois hashis, ou seja, o filósofo vai compartilhar seus hashis com os filósofos do lado direito e do lado esquerdo. Após se alimentar ele volta a pensar por algum tempo e depois volta a comer. Ele só pode se alimentar se conseguir pegar dois hashis.

Esse é um problema de exclusão mútua e na implementação foi considerado que cada hashi é uma região crítica, ou seja, um recurso compartilhado que deve ser acessado por somente um processo por vez, e que os filósofos são os processos. Para resolver o problema foram criados 3 algoritmos em python: um envolvendo o uso de threads e mutex, outro envolvendo threads e semáforos e por fim, o último algoritmo envolvendo fork (processos pesados) e variáveis lock.

No último algoritmo foi utilizado as classes Array e Value do pacote multiprocessing. Essas classes possibilitam que a porção de memória de suas instâncias sejam compartilhadas entre os processos pesados de maneira atômica. Portanto, essas classes realizam de maneira mais prática um IPC entre os processos. Nesse algoritmo um filósofo sempre tenta pegar primeiro o hashi da esquerda primeiramente, nos demais tem uma alternância na ao tentar pegar os hashis.

Conclusão:

A realização desse trabalho foi de suma importância para o melhor entendimento do funcionamento dos processos pesados (Fork) e threads, para a compreensão dos conceitos de IPC, PIPE, para entender os problemas, RaceCondition, DeadLock e Starvation, e como resolver esses problemas através das soluções, mutex, semáforos e variáveis lock. Além disso, o aprendizado foi considerável visto que foi feita a implementação de cada solução utilizando uma linguagem de programação.