Implementação de um Pseudo-SO

Ana Carolina Lopes, Jorge Luiz Andrade

1 Introdução

2 Ferramentas e Linguagem utilizada

Para o desenvolvimento do trabalho escolheu-se a linguagem **C++** com uso de biblioteca padrões.

3 Solução desenvolvida

3.1 Pseudo SO

A execução do programa começa com a leitura do arquivo de entrada. O arquivo é lido linha a linha e cada linha é analisada para extrair os dados relacionados a cada processo. Após a leitura de um processo, caso ele tenha entradas corretas – isto é, as requisições de dispositivos e memória estejam dentro dos limites definidos – ele é armazenado no vetor de lista de processos cujo nome é **cliente** na posição do seu tempo de inicialização.

O vetor **cliente** atua como o cliente que solicita a criação de processos. Cada entrada dele tem uma lista de processos que serão requisitados no tempo do índice daquela entrada. Por exemplo, no índice zero estão todos os processos cujo tempo de inicialização são zero. No índice 1 estão os processos de tempo de inicialização 1 e assim sucessivamente.

Após a finalização da leitura do arquivo, a **main** chama o método **pseudoSO**. O sistema operacional é simulado percorrendo-se o vetor **cliente** e executando as suas requisições. A cada entrada do vetor, os processos daquele índice são adicionados às filas correspondentes às suas prioridades. Em seguida, verifica-se se o **temporizador**, que é uma variável que atua como um clock, é maior que o último índice lido do vetor **cliente**. Caso não seja, o **escalonador** é chamado. Caso seja, o vetor **cliente** deve ser lido até que esses valores se igualem, para que todos os processos até aquele tempo entrem nas filas de prioridade. Quando a leitura de **cliente** termina, o **escalonador** é chamado até que todas as filas de prioridade se esvaziem.

O **escalonador** primeiramente verifica se a fila de processos de tempo real está vazia. Caso esteja, ele chama o **escalonadorUsuario**. Caso contrário, a memória para os processos reais é alocada e eles são executados e depois desalocados até que a fila se esvazie.

O método **escalonadorUsuario** inicialmente verifica qual a fila de maior prioridade não vazia. Caso todas estejam vazias, ele aumenta o temporizador e chama o método **age**. Caso uma das filas esteja não vazia, ele pega o primeiro processo dessa fila e realiza os seguintes passos: Verifica se memória para o processo foi alocada. Se não tiver, tenta alocar. Em seguida, ele tenta alocar os recursos do processo caso ele esteja requisitando algum. Caso a alocação de recursos ou de memória não tenha sido bem sucedida, o processo é movido para o fim da fila e incrementa-se o **temporizador**.

Se a memória e recursos do processo tiverem sido alocados, o processo recebe posse da CPU pelo período de um *quantum*. Após o fim do processamento, ele incrementa o temporizador, altera a idade do processo (recebe **-1**) e verifica se o tempo de processamento zerou. Caso tenha, o processo é excluído da fila e a sua memória é liberada. Caso contrário, ele é movido para o fim da fila. Por fim, **escalonadorUsuario** chama o método **promoverProcesso**.

3.2 Módulo de Processos

Um processo é simplesmente uma coleção de dados do processo, contando com os seguintes campos que foram extraídos do arquivo de entrada: **pid, idade, offset, tempoInicializacao, prioridade, tempoProcessador, blocoMem, reqImpressora, reqScanner, reqModem, codDisco**.

Além disso, esse módulo tem também um método chamado **precisaRecursos**, que verifica se o processo precisa alocar algum recurso.

3.3 Módulo de Filas

O módulo de filas é composto por duas estruturas: **processosReais** e **processosUsuario**. A primeira é apenas uma estrutura de dados do tipo *list*, estrutura da biblioteca padrão *list* do *C++*, que contém processos e será utilizada como uma fila. Os **processosUsuario**, por sua vez, são uma classe que contém três listas de processos que também serão usadas como filas. Cada uma dessas filas corresponde às prioridades dos processos que elas armazenarão, isto é, a **fila1** terá os processos de prioridade 1 e assim por diante.

O módulo de filas tem três métodos: **adicionarFilas**, **promoverProcesso** e **age**. O método **adicionarFilas** adiciona um processo a uma das filas de acordo com a sua prioridade e também o adiciona à fila de processos globais. **promoverProcesso** por sua vez varre todas as filas de prioridade e verifica se os processos contidos nelas estão velhos o suficiente para terem suas prioridades aumentadas. Por fim, **age** simplesmente percorre todas as filas e incrementa a variável idade de todos os processos.

3.4 Módulo de Memória

O módulo de memória conta basicamente com um *array* de inteiros com 1024 posições por padrão. Cada posição desse *array* corresponde a um bloco da memória, com seu conteúdo sendo igual ao **pid** do processo alocado, ou a **-1** caso o bloco esteja livre.

Além disso, o módulo considera que a região para processos em tempo real possui 64 blocos, sendo os 960 blocos restantes reservados para processos de usuário.

O módulo possui apenas dois métodos, **alocaBloco** e **liberaBloco**. **AlocaBloco** aloca uma porção contígua de memória para um processo, respeitando a região reservada para cada tipo de processo(tempo real ou usuário), retornando o índice do início do bloco de alocação. Caso não encontre blocos contíguos suficientes para o processo a alocação não é feita e o método retorna -1. **LiberaBloco** libera toda a região de memória anteriormente alocada para um processo.

3.5 Módulo de Recursos

O módulo de recursos é composto por variáveis do tipo inteiro que identificam o processo para o qual cada tipo de recurso está alocado: *scanner*, duas impressoras, modem e dois *sata*. As impressora e os satas são armazenados como um *array* de duas posições. Caso o recurso não esteja alocado, é armazenado **-1** na variável correspondente.

O módulo possui quatro métodos. O método **alocaRecurso** aloca o recurso de um tipo para um processo. Caso essa alocação não seja possível o método retorna -1. **checaRecurso** verifica se um processo, identificado pelo seu pid, possui algum recurso alocado, retornando *true* ou *false*. O método *liberaRecurso* libera os recursos de um determinado tipo que estejam alocados para um processo, enquanto o método **liberaTodos** funciona da mesma forma, mas liberando os recursos de todos os tipos que estejam alocados para um processo.

4 Dificuldades encontradas

5 Divisão do trabalho

References