# **Relatório de Simulador de Gerência de Processos**

Graduando Gabriel Caetano, 201710009

O presente relatório tem por objetivo descrever o funcionamento do simulador de gerência de processos no modelo de 5 estados usando múltiplas filas para lidar com diversos dispositivos de entrada e saída.

O projeto teve início pela implementação das filas, o qual foi utilizada uma estrutura comum de lista duplamente encadeada, em que existe a *struct QUEUE* e *struct NODE,* conforme está apresentada na Figura 1.

typedef struct QUEUE {

NODE \*head;

NODE \*tail;

int size;

} QUEUE; (a)

typedef struct NODE {

struct NODE \*prev;

struct NODE \*next;

PCB \*process;

} NODE; (b)

Figura 1 - Código das *structs* usadas na implementação da lista encadeada, (a) *struct QUEUE* e (b) *struct NODE.*

Na Figura 1 (a) podemos ver que a estrutura contém os ponteiros do primeiro e último membros, assim como o tamanho da fila. Logo na Figura 1 (b) a estrutura contém ponteiros para o próximo membro e membro anterior e do processo contido neste nó.

Para o funcionamento adequado da fila foram implementadas apenas 2 funções para acessar os valores da fila como pode ser visto na Figura 2.

void pushHead(QUEUE \*queue, NODE \*node) {

if (queue->size) {

queue->head->prev = node;

node->next = queue->head;

} else {

queue->tail = node;

}

queue->head = node;

queue->size++;

}

(a)

NODE \*popTail(QUEUE \*queue) {

NODE \*res = NULL;

if (queue->size > 1) {

queue->tail->prev->next = NULL;

res = queue->tail;

queue->tail = res->prev;

res->prev = NULL;

} else {

res = queue->head;

queue->head = queue->tail = NULL;

}

queue->size--;

return res;

} (b)

pushHead(terminated, popTail(running)); (c)

Figura 2 - Código das rotinas de acesso da fila, (a) pushHead(), (b) *popTail()* e (c) utilização*.*

A função mostrada na Figura 2 (a) insere um valor no início da fila, ao passo que a função mostrada na Figura 2 (b) remove e retorna um valor do fim da fila, permitindo que um valor possa ser transferido por exemplo da fila *running* para a fila *terminated* com a sintaxe demonstrada na Figura 2 (c).

Com a estrutura da fila criada, foi feita a implementação das seguintes filas:

* Fila processos, gerada a partir do arquivo lido pelo programa, para concentrar uma lista de todos os processos a serem executados;
* Fila *new*, na qual cada processo é inserido no tempo indicado pelo arquivo lido;
* Fila *ready,* onde ficam os processos que estão prontos para serem escalonados para o processador;
* Fila *running*, com tamanho limitado em 1 inicialmente, é onde fica o processo que será executado pelo processador a cada tempo, também foi usada uma fila neste estado para facilitar a alteração do código caso fosse necessário executar com mais processadores em paralelo;
* Fila *terminated*, para onde vão os processos que finalizaram sua execução; e
* Um array com uma fila para cada dispositivo indicado no arquivo lido, que será usada para gerenciar as chamadas de dispositivos dos processos e onde ficarão os processos bloqueados.

Após a implementação das filas foi feita a construção do *process* *control block* (PCB) com uma *struct* contendo as informações descritas na Tabela 1.

Tabela 1 - Lista de variáveis utilizados na criação da *struct PCB.*

| **Variáveis** | **Descrição** |
| --- | --- |
| *state* | Informação de estado atual do processo, utilizada para registro no output, atualizado cada vez que o processo muda de um estado (fila) para outro. |
| *start* | Tempo do processador em que o processo deve ser iniciado, inserindo na fila *new*, definido na leitura do arquivo de entrada. |
| *cpuTimes* | Array de tempos de CPU entre as chamadas de sistema definido na leitura do arquivo de entrada, limitado em trinta e dois, caso o arquivo de teste possua processos com mais fatias de CPU entre chamadas de sistema este valor deve ser alterado. |
| *cpuI* | Índice do tempo de CPU que está sendo executado atualmente, inicializado em zero e incrementado cada vez que ocorre uma chamada de sistema. |
| *devCalls* | Array de dispositivos na ordem em que serão chamados pelo processo, assim como *cpuTimes*, também limitado em trinta e dois valores, caso o processo possua mais do que trinta e dois chamadas ao sistema este valor deve ser alterado. |
| *devI* | Índice do próximo dispositivo a ser chamado, inicializado em zero e incrementado cada vez que ocorre uma chamada de dispositivo.. |
| *callsLeft* | Quantidade de chamadas a dispositivos que o processo ainda fará antes de encerrar, inicializado a partir do arquivo lido na entrada, após contar quantas chamadas de sistema haverá no total e decrementado a cada chamada feita. |
| *timeLeft* | Quantidade de unidades de tempo que o processo ainda tem para executar a tarefa atual, resetado cada vez que o processo entra em uma execução com tempo predefinido com o valor do tempo determinado pela execução em questão, sendo quatro no caso do processador e sendo o tempo de dispositivo quando o processo realiza uma chamada, esta variável é decrementada a cada iteração em que o processo se mantém em alguma execução temporizada. |
| totalCPU | Contador inicializado em zero, incrementado a cada iteração em que o processo está em estado *running.* |
| totalWait | Contador inicializado em 0, incrementado a cada iteração em que o processo está em estado *new, ready* ou *blocked.* |

Com a implementação das filas e do PCB foi possível realizar a implementação do simulador, a qual foi dividida em etapas.

A primeira sendo a leitura do arquivo de entrada, onde é conferida a formatação e a validade de cada informação enquanto é criada uma estrutura de *INPUT* contendo todas as informações lidas do arquivo de forma facilmente manipulável pelo sistema. Em caso de algum problema na leitura do arquivo uma mensagem de erro é emitida no console e o programa é encerrado. Caso o arquivo seja lido corretamente, uma mensagem de sucesso aparece no console e o programa executa normalmente até a sua conclusão.

Na próxima etapa ocorre a execução do programa que consiste inicialmente na criação das filas que serão usadas para o gerência dos processos, a partir daí o sistema entrará em um *loop* que será encerrado apenas quando todos os processos estiverem na fila *terminated*. Dentro deste *loop* primeiramente serão resolvidos os processos que encontram-se bloqueados em algum dos dispositivos. Seguido destes que estão entrando no sistema, para resolver o que está em execução. E por fim tratar as mudanças de estado automáticas, como mover do estado *new* para *ready,* e encerrar um processo que já executou sua fatia de tempo ou escolher um novo processo para ser executado.

Existem duas rotinas auxiliares para construir oarquivo de saída corretamente, uma contendo um contador que atualiza os valores de tempo total de CPU e espera de cada processo e adiciona uma linha contendo a informação do estado de cada processo ao fim de cada iteração, e outra que ao concluir o *loop* adiciona ao arquivo os tempos de cada processo e encerra todos os recursos usados durante a execução.

Após a execução, uma última etapa destrói os recursos criados durante a leitura do arquivo de entrada para garantir que não ocorra vazamento de memória no sistema.

Para a execução do simulador, existe uma variável global *TEST\_NUMBER* usada para selecionar facilmente o arquivo de entrada.

A fim de garantir um código de qualidade foi usada a opção *-Wall* durante a compilação e também foram feitos testes de execução com a ferramenta *valgrind*.

Além disso, foi criado um script em *python3* “*taskGenerator.py”* para gerar arquivos de teste facilmente para o sistema, sendo necessário apenas passar os valores máximos de cada parâmetro dentro do código.