Simulador de Escalonador de Processos: Relatório de Implementação

Henrique S. Pinheiro, Sávio O. Camacam

***Resumo*—Este relatório detalha o processo de implementação de um simulador para escalonamento de processos de sistemas operacionais. As descrições apresentadas nesse trabalho tratam do funcionamento de algoritmos específicos aplicados nos três principais modelos de sistemas operacionais para um ambiente de simulação controlado.**

***Palavras-chave*—Escalonamento, Sistemas Operacionais, Filas de Escalonamento, Processos, Chaveamento, Tabela de Processos.**

I. INTRODUÇÃO

Esse documento relata o percurso feito na implementação de módulos que foram necessários para executar cinco políticas de escalonamento usadas em Sistemas Operacionais em Lote e Sistemas Operacionais Interativos. As políticas implementadas foram: *Round Robin, SJF (Shortest Job First), FCFS (First Come, First Served), Múltiplas Filas (FP) e Random.* As implementações foram feitas com base num Modelo de Simulador previamente fornecido, com as Estruturas de dados definidas em módulos, onde o seu funcionamento geral e as estruturas usadas são descritas e explicadas em tópicos próprios. Pela necessidade de implementação das particularidades de cada política, foi reservado um espaço para descrever o funcionamento de cada uma delas de acordo as modificações necessárias. Módulos adicionados e a interface entre o modelo original são descritos logo em seguida e por fim são apresentadas nossas conclusões sobre o desempenho dessa atividade, a complexidade envolvida nos cálculos e sincronização do escalonamento entre múltiplas políticas e o aprendizado com a modularização do problema.

II. SOBRE A IMPORTÂNCIA DA SIMULAÇÃO

Como já citado na descrição do presente trabalho, à simulação é uma ferramenta extremamente importante para o desenvolvimento de novas técnicas e tecnologias em todas as áreas da ciência e engenharia.

A vantagem está no fato que a implementação de um simulador é geralmente mais simples que a implementação da técnica na prática, mas seus resultados permitem avaliar se o ganho esperado com a técnica supera seus esforços de implementação. Além disto, como a principal funcionalidade da simulação é coletar dados para a análise posterior, sua implementação não requer tanto cuidado com desempenho da máquina quanto uma implementação real.

Esse trabalho foi submetido como objeto de avaliação da disciplina de Sistemas Operacionais do curso de Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

III. O SIMULADOR

Em uma visão geral e simplificada, o simulador funciona com a execução de um programa em torno dos seguintes módulos:

- Módulo de Leitura de Arquivos;
- Módulo de Blocos de Controle de Processos;
- Módulo de Políticas de Escalonamento;
- Módulo de Eventos;

Esse grupo de módulos abriga as estruturas necessárias para a implementação das políticas.

O simulador se inicia com a leitura do arquivo de experimento passado como parâmetro para o simulador e partir dele, o arquivo de processos é carregado com os eventos que ocorrerão na sua descrição.

São carregadas também as informações a respeito de qual política de escalonamento deve ser usada e quais os parâmetros necessário.

Na inicialização do simulador todos os processos são instanciados numa lista de processos novos com as suas variáveis de contagem inicializadas adequadamente. São criadas também outras duas listas onde são mantidos os processos para a representação de seus estados possíveis, como a Lista de Processos Bloqueados (bloqueados), que representa os processos à espera de algum dado externo ou interrupção para voltar a fazer uma solicitação de CPU, e a Lista de Processos Prontos (prontos) representando os processos em espera para serem escalonados e usar os recursos da CPU. Com a política instanciada, arquivo de processos carregado e as listas representando os estados dos processos, o simulador segue sua execução verificando a necessidade de chaveamentos, seja por descrição do arquivo de processo ou por aplicação da política de escalonamento usada até que não exista mais processos que precisem de tempo de processamento.

No primeiro ciclo de execução, depois da entrada de todos os registros do arquivo de processos como novos, o primeiro processo está pronto para ser escalonado.

A primeira verificação é feita para saber se o tempo de entrada do primeiro processo na fila de novos é igual ao tempo do relógio contando naquele instante, significando que aquele

processo deve entrar na fila de prontos do simulador. A rotina de *novoProcesso* é chamada ainda dentro dessa verificação para que a política utilizada possa aplicar a lógica necessária em um evento de criação de processo.

Caso haja algum processo em execução, a cada tempo do relógio, seu tempo de execução é atualizado e uma verificação é feita para aquele processo afim de saber se há algum evento programado para aquele processo em determinado tempo, onde espera-se que seja ou um evento de Término ou um Evento de Bloqueio, que pode ser proveniente da própria lista de eventos do processo ou da aplicação de determinada política.

Em casos de um Evento de Término, a rotina *fimProcesso* da política em execução é chamada para que, caso necessário, a política possa tratar esse evento de acordo com seu funcionamento. O processo em execução tem algumas das suas variáveis de contagem atualizadas e em é tirado de execução e removido do simulador.

Em casos de um Evento de Bloqueio o processo tem sua variável de controle de tempo de bloqueio (tempoBloqueio) atualizada para um valor que representa por quanto tempo ele deverá permanecer bloqueado. O processo é tirado de execução e colocado na lista de bloqueados.

Após a verificação de eventos a rotina *tick* da política é chamada para que, caso necessário, a política possa implementar suas regras para quando um tempo do relógio do simulador passar.

Após o *tick* é feita uma atualização, através do decremento, do tempo de bloqueio dos processos que se encontram na lista de bloqueio e, caso algum processo esteja em um momento de desbloqueio, ou seja, *processo->tempoBloqueio <= 0*, o processo é removido da lista de bloqueados e inserido na lista de prontos onde poderá ser novamente escalonado de acordo com a política utilizada.

Na última etapa de um ciclo do simulador, é feito o incremento do tempo do relógio.

É feita uma verificação para o caso de não existir nenhum processo em execução e caso não esteja a rotina de *escalonar* da política em vigor é chamada e, caso exista um novo processo a ser executado nesse momento é feita a troca de contexto e caso não exista um novo processo a ser executado no momento o *tempo_ocioso*, que representa o tempo sem processamento, é incrementado.

O clico se repete continuamente até que não existam mais processos novos, processos prontos, processos bloqueados ou processos executando.

IV. Estruturas

Aqui estão descritas as estruturas de dados escolhidas para estruturar o simulador e facilitar a implementação e manutenção do código, assim como uma visão mais detalhada do simulador e seu fluxo de funcionamento.

A leitura do arquivo de entrada, ou arquivo de experimento, é feita em duas etapas e envolve duas estruturas do simulador, *experimento_t* e *politica_t*.

```
1. typedef struct experimento_t{
2.     char* nome_exp;
3.     char* arq_processos;
4.     char* arq_saida;
5.     politica_t* politica;
6. }experimento_t;
```
(1)

A estrutura de dados *experimento_t* contém as informações presentes no arquivo de experimento como o nome do arquivo de experimento, o nome do arquivo de processo que deverá ser usado para a simulação, o nome do arquivo de saída que conterá os resultados da simulação de acordo com as especificações e um ponteiro para uma estrutura de dados do tipo *politica_t* representado a política e as rotinas que serão executadas durante a simulação.

```
1. typedef struct politica_t{
2.     POLITICA_ESC politica;
3.     union{
4.         rr_t* rr;
5.         fp_t* fp;
6.         fcfs_t* fcfs;
7.         random_t* random;
8.     }param;
9.     bcp_t* (*escalonar)(struct politica_t*);
10.    void (*tick)(struct politica_t*);
11.    void (*novoProcesso)(struct politica_t*, bcp_t*);
12.    void (*fimProcesso)(struct politica_t*, bcp_t*);
13.    void (*desbloqueado)(struct politica_t*, bcp_t*);
14. }politica_t;
```
(2)

A estrutura de dados *politica_t* pode ser considerada a estrutura de maior importância no simulador.

Cada uma das variáveis presentes nessa estrutura são descritas na lista abaixo:

- **política:** uma enumeração para indicar o tipo de política que ela representa;
- **param:** uma union responsável por manter dados específicos de cada uma das políticas implementadas;
- **(*escalonar):** ponteiro de função que recebe como parâmetro uma referência para uma *politica_t* e, deve retornar uma referência para um *bcp_t*, representando o processo que deverá ser colocado em execução;
- **(*tick):** ponteiro de função que recebe como parâmetro uma referência para uma *politica_t*. Essa rotina é chamada com o incremento do relógio;
- **(*novoProcesso):** ponteiro de função que recebe como parâmetro uma referência para uma *politica_t* e um *bcp_t*. A referência para o *bcp_t* representa o novo processo que deve ser inserido no sistema e o papel da função é descrever de que maneira o novo processo deve ser inserido de acordo com a política implementada.
- **(*fimProcesso):** ponteiro de função que recebe como parâmetro uma referência para uma *politica_t* e um *bcp_t*. A referência para o *bcp_t* representa o processo que deve ser removido do sistema pois já foi

terminado e não precisará de mais processamento.

- **(*desbloqueado):** ponteiro de função que recebe como parâmetro uma referência para uma *politica_t* e um *bcp_t*. A referência para o *bcp_t* representa o processo que deve ser tirado de execução e colocado na lista de processos bloqueados. Essa função é chamada toda vez que um evento de bloqueio é detectado no loop de execução principal do simulador.

V. Implementação das Políticas

Em termos de hierarquia e prioridade de processos, em sistemas operacionais somos introduzidos a um princípio que trata da diferenciação entre mecanismo e política de escalonamento.

Uma vez que impreterivelmente, os processos são escalonados pelo núcleo, processos de usuários podem controlar como seus processos filhos serão escalonados a partir de políticas próprias que são passadas ao núcleo e executadas por ele.

A. *Round Robin*

Caracterizado para operar em sistemas operacionais interativos ou servidores, o Round Robin também conhecido como algoritmo por chaveamento circular, trata o processo atribuindo-lhe um intervalo de tempo para uso da CPU de tamanho fixo, o quantum.

```
1.  typedef struct rr_t{
2.      int quantum;
3.      bcpList_t* fifo;
4.      int* pos;
5.  }rr_t;
```
(3)

A estrutura *rr_t* foi utilizada para representar os parâmetros específicos da política Round Robin.

Essa estrutura contém: a quantidade do tempo de quantum que será dado para o processo em execução, uma lista de processos representando todos os processos carregados no sistema e um índice para indicar em qual posição, na lista de prontos, que o escalonador está.

No simulador essa política tem cinco assinaturas de funções para manipulação dos processos.

```
1.  static void RR_tick(struct politica_t *p);
2.  static void RR_novoProcesso(struct politica_t *p, bcp_t* novoProcesso);
3.  static bcp_t* RR_escalonar(struct politica_t *p);
4.  static void RR_fimProcesso(struct politica_t *p, bcp_t* processo);
5.  static politica_t* POLITICARR_criar(FILE* arqProcessos);
```
(4)

Como citado anteriormente, no loop principal há a chamada de execução da política selecionada no experimento – call-back – executada por uma função *tick* para cada política, e embora ela não tenha desempenhado nenhuma função nas demais políticas, nessa, porém, foi de suma importância por ser a responsável pela contabilização do *timeslice* de cada processo com base no quantum passado como parâmetro.

No Round-Robin, a cada contagem de tempo do relógio do processador, o processo que tem a atenção da CPU tem seu tempo de controle decrementado e nessa situação podem ocorrer duas situações: a primeira, um evento de bloqueio do processo em execução pode ocorrer e então o processo é interrompido e em seguida há um chaveamento para outro processo; na segunda situação, seu *timeslice* é encerrado e na função *RR_tick* o processo em execução é inserido na lista de prontos e é feito o chaveamento para um outro processo, seguindo a sequência, na lista de prontos.

Na função *RR_escalonar*, é retornado um *BCP (Bloco de Controle de Processos)* como o processo em execução no loop principal, e nessa função o programa verifica se a quantidade de blocos de processos é igual a zero, o que significa que não há nenhum processo carregado para aquela política, retornando assim NULL.

De outra forma, através de um índice que percorre todos os blocos no vetor de ponteiros de blocos do Round-robin, é verificado se o PID daquele bloco está na lista de bloqueados e sendo negativo esse resultado, é afirmativo que ele estará na lista de prontos e assim o processo é removido dessa lista e é retornado para o loop principal do programa e colocado em execução para contabilização dos eventos correspondentes e atualização do seu tempo de execução.

Para o call-back de desbloqueado, na Política Round-Robin, foi usado uma função Dummy, que não executa nenhuma instrução, para evitar o tratamento de casos especiais para quando a função existe ou não, tornando o código mais simples de ser implementado.

B. *SJF (Shortest Job First)*

Esse é um algoritmo que supõe previamente que são conhecidos todos os tempos de execução. Otimizado para sistemas em lote, o *Shortest Job First*, ou Tarefa Mais Curta Primeiro, trata da execução dos processos menores em primeira ordem, considerando que todos têm a mesma prioridade um processo mais curto não interfere em um mais lento e assim o tempo médio de retorno é menor.

A estrutura *sjf_t* foi utilizada para representar os parâmetros específicos da política SJF.

```
1.  typedef struct sjf_t{
2.          bcpList_t *lista;
3.  }sjf_t;
```
(5)

Essa estrutura contém apenas uma lista de processos que representa todos os processos que devem ser processados no sistema.

Como citado anteriormente, a política SJF necessita saber a priori o tempo total de processamento do processo para realizar o escalonamento de maneira apropriada e, devido a uma limitação da estrutura do arquivo de processos (*. proc), é necessário calcular o tempo total de cada processo no momento em que ele entra no sistema.

O cálculo é feito através da somatória do tempo dos eventos de Bloqueio e Termino de cada processo. Dessa maneira foi necessário a criação de uma nova variável na estrutura *bcp_t*, *tTotalProcesso*, que é responsável por guardar o tempo total de execução necessário para cada processo.

No simulador essa política tem quatro assinaturas de funções para manipulação dos processos.

```
1. static void SJF_novoProcesso(struct politica_t *p, bcp_t* novoProcesso);
2. static void SJF_fimProcesso(struct politica_t *p, bcp_t* processo);
3. static bcp_t* SJF_escalonar(struct politica_t *p);
4. static politica_t* POLITICASJF_criar(FILE* arqProcessos);
```
(6)

Quando a função *SJF_novoProcesso* é chamada o parâmetro *novoProcesso*, tem sua lista de eventos percorrida para o cálculo, previamente descrito, do tempo total de processo. Após o cálculo ser realizado o processo é então colocado na lista de novos para ser manipulado adequadamente pelo simulador.

Ao término de um processo, a função *SJF_fimProcesso* é chamada e o processo passado por parâmetro é removido do sistema pois sua execução já terminou.

No loop principal do simulador quando o call-back para escalonar é chamado, no caso da política SJF, a função *SJF_escalonar* é chamada. Sua responsabilidade é a de percorrer a lista de prontos, encontrar o processo que possui o menor tempo total de execução faltante para o término do processo e retornar o processo para o simulador onde ele será colocado em status de execução.

A função *POLITICASJF_criar* simplesmente cria a política e faz a atribuição correta dos call-backs. Os call-back de *tick* e *desbloqueado* utilizam as funções *DUMMY*, uma vez que não existem operações especificas para serem feitas quando algum desses eventos ocorre.

C. *FCFS (First Come, First Served)*

Provavelmente é o algoritmo mais simples na implementação do escalonamento, sendo basicamente uma fila de processos em que são escolhidos aqueles em sua ordem se chegada. Diferenciando do Round Robin em termos de tempo de execução, esse é um método em que o processo não tem um tempo fixo de uso da CPU podendo fazer uso dessa por quanto tempo seja necessário.

A estrutura fcfs_t foi utilizada para guardar os parâmetros dessa política.

```
1. typedef struct fcfs_t{
2.     bcpList_t* fifo;
3. }fcfs_t;
```
(7)

No simulador essa política possui quatro call-backs para gerencia-la.

```
1. static void FCFS_novoProcesso(struct politica_t *p, bcp_t* novoProcesso);
2. static void FCFS_fimProcesso(struct politica_t *p, bcp_t* processo);
3. static bcp_t* FCFS_escalonar(struct politica_t *p);
4. static politica_t* POLITICAFCFS_criar(FILE* arqProcessos);
```
(8)

Quando um novo processo é inserido no sistema a função *FCFS_novoProcesso* é chamada. Sua responsabilidade é a de inserir o processo na lista de processos novos do sistema.

Caso seja identificado um evento de termino a função *FCFS_fimProcesso* é chamada. Sua responsabilidade é a de retirar o processo do sistema já que seu processamento já foi concluído.

Na função de escalonamento para o *FCFS_escalonar*, é feita uma iteração entre todos os processos na lista de prontos procurando por aquele com menor tempo para entrada, ou seja, o processo que chegou primeiro ao sistema. O processo é então removido da lista de prontos e retornado para o simulador colocá-lo em execução.

A função *POLITICAFCFS_criar* simplesmente cria a política e faz a atribuição correta dos call-backs. Os call-back de tick e desbloqueado também utilizam as funções *DUMMY*.

D. *Múltiplas Filas*

Aprimorado para a resolução de um problema de chaveamento de processos entre a memória principal e o disco, esse método atribui tempos de execução referentes a uma classificação específica de prioridades dada a uma faixa de processos semelhantes.

A política de múltiplas filas, por fazer uso da chamada das políticas existentes agrupando processos em níveis prioridade foi a mais complexa, tanto em termos de escalonamento, como na criação da própria política.

```
1. typedef struct fp_t{
2.     int tam;
3.     struct politica_t** filas;
4. }fp_t;
```
(9)

A estrutura escolhida para guardar as variáveis especificas foi a *fp_t(9)*.

Essa estrutura contém, um ponteiro para um vetor de filas de políticas e o tamanho, ou quantidade, de filas que existem.
A política FP, dentro do simulador, utiliza quatro call-backs.

```
1. static void FP_novoProcesso(struct politica_t *p, bcp_t* novoProcesso);
2. static void FP_fimProcesso(struct politica_t *p, bcp_t* processo);
3. static bcp_t* FP_escalonar(struct politica_t *p);
4. static politica_t* POLITICAFP_criar(FILE* arqProcessos);
```
(10)

Quando um novo processo entra no sistema a função *FP_novoProcesso* é chamada. Sua responsabilidade é de encontrar, através dos dados do *bcp_t*, em qual faixa de prioridade o novo processo pertence. De posse do número da faixa de prioridade do processo, a fila de política pode ser indexada e a rotina de *novoProcesso* da própria política é chamada passando o novo processo como parâmetro.

Caso um evento de término ocorra, o mesmo é feito para a função *FP_fimProcesso*, diferenciando-se apenas por chamar a rotina de *fimProcesso* da política indexada.

Na criação da política, com a passagem do arquivo de experimentos como parâmetro, a função, *POLITICAFP_criar*, inicializa uma estrutura de bloco de controle de processo para Múltiplas Filas que contém um vetor de ponteiros de políticas, para que cada posição receba uma nova política criada a partir de cada linha encontrada no arquivo de Experimento.

Assim, é feita uma varredura desse arquivo, coletando as quarenta políticas indicadas. Em cada linha é feita a verificação do nome da política dentre as quatro demais e assim é acessado na estrutura de blocos própria para múltiplas filas, uma posição das políticas instanciadas para a criação da política correspondente chamando a criação da própria função já implementada.

Devido a modularização adotada com a passagem de um arquivo como parâmetro para a criação da política em específico, notamos que é efetivamente usado apenas na criação da política Round-robin a para as demais, é um

parâmetro totalmente dispensável e nesse quesito, a verificação de criação do Round-robin foi a única a ter um tratamento diferenciado devido a questão de leitura do arquivo. Como essa é a única política que espera um parâmetro do arquivo, é necessária sua abertura para leitura desses dados, então foi criado um arquivo temporário com a gravação do parâmetro e esse arquivo temporário foi passado para a criação da política.

Como essas políticas já são implementadas e para tanto, executam seu próprio escalonamento, fica a cargo do *FCFS_escalonar* a iteração entre processos a serem escalonados a partir de sua faixa de prioridades. Ao ser encontrar a primeira ocorrência daquela faixa, a política é retirada do vetor de ponteiro de políticas e sua própria rotina de escalonamento é chamada.

E. Random

Seu funcionamento é simples: um processo aleatório é escolhido entre os processos prontos para serem executados. No bloqueio, o processo é colocado na lista de *bloqueados* e um outro processo aleatório é executado. No desbloqueio, o processo é simplesmente colocado na lista de processos prontos.

A estrutura escolhida para guardar as informações especificas da política Random foi a *random_t(11).*

```
typedef struct random_t{
    bcpList_t* lista;
}random_t;
```
(11)

Essa estrutura contém apenas um ponteiro para a lista de processos presentes no sistema.

A manipulação dos processos pela política *Random* se dá pelo uso das funções cujas assinaturas são:

```
static void RANDOM_novoProcesso(struct politica_t *p, bcp_t* novoProcesso);
static void RANDOM_fimProcesso(struct politica_t *p, bcp_t* processo);
static bcp_t* RANDOM_escalonar(struct politica_t *p);
static politica_t* POLITICARANDOM_criar(FILE* arqProcessos);
```
(12)

Como não há mudanças nas funções triviais de cada política, a distinção entre elas fica a cargo da função *POLITICA_escalonar*. No caso da Política Random, com a escolha de um processo aleatório da sua lista de processos usamos a função *rand ()* da biblioteca <*time.h*> para gerar um valor aleatório no intervalo de 0 ao tamanho da lista de processos prontos. Dessa forma um processo aleatório será escolhido para ser escalonado e retornado ao simulador para entrar estado de execução.

VI. Novos Módulos

Esse tópico ficou reservado para descrição das alterações que foram feitas no esqueleto original do simulador fornecido pelo professor.

A. Programa Principal

No programa principal, *simulador.c*, foram criadas as variáveis responsáveis pela contagem dos eventos e informações que são necessária para a análise da política aplicada no simulador. Essas variáveis são: *tme, tmr, vazao e qtdProcExecutados.*

O Tempo Médio de Espera (tme) é um cálculo resultante da média de chaveamento dos processos numa política, calculado como o somatório dos tempos no relógio de entrada e reentrada dos processos dividido pela quantidade de processos. Esse valor é demonstrado ao final da execução do experimento.

O tempo de retorno (tmr) é estatisticamente o tempo médio do momento em que uma tarefa em lote é submetido até o momento em que ele é terminado. Ele indica quanto tempo, em média, o usuário tem de esperar pelo fim do trabalho. Esse valor está sendo calculado a cada evento de término do processo, o somatório acumulado dos processos com a diferença do tempo da última execução com o tempo da primeira execução daquele processo.

A vazão (vazao) é a quantidade média de tarefas que o sistema termina num intervalo de tempo, no nosso programa é dado pela média da quantidade de processos executados em um intervalo de 1000 tempos do relógio do simulador.

A quantidade de processos executados (qtdProcExecutados) guarda a contagem do número de processos executados até determinado tempo do relógio.

B. Logger

Para a criação do log final foi criada uma nova biblioteca *logger.h*, essa biblioteca é responsável por registrar os eventos importantes do simulador a medida em que eles acontecem.
A estrutura principal da biblioteca *logger.h* é o *Log*.

```
struct _log{
    FILE *arquivoLog;
    FILE *diagramaEventos;
    FILE *sequenciaTermino;
    FILE *header;
};
```
(13)

Ela contém somente ponteiros para arquivos: *arquivoLog*, *diagramaEventos*, *sequênciaTermino* e header.

O arquivo de Log (arquivoLog) é o arquivo de saída final que é gerado ao final da execução de um experimento. Nele estão contidas as informações de todos os outros arquivos.

O arquivo do Diagrama de Eventos (diagramaEventos) é responsável por guardar os eventos referentes a troca de contextos, bloqueios, desbloqueios, criações e término de processos.

O arquivo de sequência de término (sequenciaTermino) registra o PID dos processos a medida em que eles terminam.

O arquivo de cabeçalho (header) é responsável por manter as estatísticas calculadas do experimento como tme, tmr e vazão além de outras informações que, segundo a especificação do trabalho, devem estar presentes no arquivo final de saída.

Todos os eventos são gravados através da função *recordEvent*.

```
int recordEvent( Log *logger, char *content, LogType logtype );
```
(14)

Essa função recebe uma referência para um Log, uma string e um enum *LogType*. Parâmetro *LogType* indica em qual dos arquivos do Log um determinado evento deve ser gravado. A string *content* representa o conteúdo que será gravado no arquivo de log.

Quando o arquivo de log deve ser gerado a função *getLog* é chamada.

```
1.  void getLog( Log *logger, char *fname );
```
(15)

Essa função recebe como parâmetro uma referência de Log, e o nome do arquivo de saída. Durante a execução dessa rotina o arquivo de saída é criado e o conteúdo dos arquivos diagramaEventos, sequenciaTermino e header são transferidos para o arquivo de saída.

VII. Problemas Encontrados

Durante a execução do presente trabalho foram encontrados alguns problemas. A descrição desses problemas e as soluções, se encontradas, são descritas abaixo.

A. Overflow do Contador

1) Problema Encontrado

Durante a execução de um experimento com muitos processos o relógio do simulador, que é um inteiro de 64 bits, pode estourar e assim a contagem recomeça do zero.

2) Solução Possível

Uma solução possível seria a de usarmos uma variável com maior capacidade para guardar o tempo do relógio.

B. Redundância na Busca de Prontos

1) Problema Encontrado

Na interpretação do código, tomando por base a implementação já feita do algoritmo Round-robin, identificamos certa redundância na busca pelo processo a ser escalonado em duas listas sem chegarmos a um consenso do porque a busca se sucedeu dessa forma.

Ao ser realizada a iteração por todos os processos no vetor de ponteiros de blocos dessa política é tomado um bloco de cada vez dada a posição encontrada naquele momento e com esse bloco, com chamadas da função auxiliar *LISTA_BCP_buscar*, é passado o PID do processo, mas a ser buscado na lista de bloqueados, e só depois é feita a busca na lista de prontos, o que analisamos ser preciso que se houvesse algum processo a ser escalonado, esse já estaria na lista de prontos sem a necessidade de uma verificação a mais.

C. Impressões no Log/Terminal

1) Problema Encontrado

Durante a impressão dos eventos percebemos um overflow nos números do tipo uint64_t mesmo que o tipo suportasse o número em questão.

2) Solução Possível

Inclusão da biblioteca *<inttypes.h>* e troca da expressão “*%lld\n”* por “*%” PRIu64 “\n”* sendo que *PRIu64* é uma macro que lida adequadamente com números 64 bits sem representação de sinal.

VIII. Divisão de Tarefas

A implementação baseada em módulos facilita não só o entendimento do código como também o trabalho em equipe. Uma vez que as interfaces que serão usada estão bem definidas, pessoas distintas podem trabalhar em outros módulos ou em até em partes diferentes de um mesmo módulo.

Essa abstração possibilitou que os integrantes da equipe dividissem bem o trabalho com cada integrante implementando uma política diferente.

No final a modularização facilitou também a refatoração do código e as modificações necessárias puderam ser feitas sem maiores preocupações enquanto mantivermos as interfaces por onde os módulos interagem entre si.

IX. Conclusão

Além de evidenciar a importância da simulação seja para qual for a área da computação, este trabalho também mostrou a importância de se estruturar bem um programa. Ter as interfaces bem definidas e as funcionalidades modularizadas fez toda a diferença para a implementação e também para o entendimento das etapas envolvidas em um escalonador de processos.

É um fato que este é um simulador muito simplista em relação ao funcionamento de um escalonador de processos reais porém ele consegue fornece uma ideia muito boa de como os algoritmos mais consagrados funcionado e também nos permite testar novas ideias e compará-las na prática com outros algoritmos.

Referências

[1] Andrew. S. Tanenbaum, “Modern Operating Systems”, 4th ed., Pearson.
[2] M. Kerrisk, “The Linux Programming Interface”, No Starch Press.