Escalonamento de Processos[1]

Geovani Pedroso da Mata[1], Gabriel Negrão Silva[2]

Universidade Tecnológica Federal do Paraná – UTFPR

Bacharelado em Ciência da Computação

Campo Mourão, Paraná, Brasil

[1]geovanimata@alunos.utfpr.edu.br

[2]itsg_negrao@hotmail.com

Resumo

O artigo tem o objetivo de mostrar de forma bem resumida como é realizado o escalonamento de processos feito por um Sistema operacional e a implementação de um simulador com o objetivo de compreensão e analise de comportamento de alguns algoritmos de escalonamento de processos.

1. Introdução

O Escalonador é uma parte do Sistema Operacional que faz o chaveamento de processos de acordo com regras bem estabelecidas, ou seja, é sua responsabilidade decidir o momento em que cada processo tenha acesso a CPU, utilizando algoritmos de escalonamentos que estabeleça a lógica de tal decisão e avaliando o cenário em que o sistema está sendo utilizado.

Uma implementação de um escalonador de processos deve se levar em conta alguns critérios como justiça (cada processo ter acesso justo ao tempo da CPU), eficiência (garantir a ocupação total do tempo da CPU), minimizar o tempo de resposta, maximizar o número de serviços processados em um determinado tempo, etc.

2. Processos

Um sistema operacional é capaz de executar várias tarefas ao mesmo tempo, tanto as realizadas pela determinação do usuário, quanto a realizada pelo próprio sistema, e toda tarefa requer acesso ao meio físico que consequentemente leva um certo tempo de resposta, ou seja, demoraria muito tempo se para cada tarefa a CPU executasse a tarefa inteira para depois ir para a próxima, por isso a criação de processos consegue ser modelados de diferentes formas para gerar melhorias para o sistema.

Um exemplo prático é o google chrome que para cada aba é executado uma tarefa, ou seja, um processo, onde se determinada aba para de funcionar o resto continua funcionando.

2.1 Criação de Processos

A criação de um processo normalmente é consequência de alguns eventos, sendo eles o início do sistema, a execução de uma chamada de sistema de criação de processo por um processo em execução, uma requisição do próprio usuário e o início de uma tarefa em lote.

No início de um sistema, em geral, é criado vários processos sendo eles de primeiro plano, na qual há interação com o usuário, porém é executado pelo sistema operacional, um exemplo seria o antivírus, e o de segundo plano, que não apresentam interação com o usuário, mas que realizam outras funções, estes processos que ficam em segundo plano e realizam atividades como impressão, mensagens eletrônicas, entre outros, são denominados de *daemons*.

A execução de uma chamada de sistema de criação de processo por um processo em execução é quando um processo normalmente em segundo plano realiza chamadas de sistema para criar processos para ajudar a realização de determinada tarefa e a criação pelo próprio usuário é quando ele executa qualquer funcionalidade ou aplicativo do sistema operacional, que consequentemente gera um processo, além disso há criação em lote que ocorre em computadores de grande porte, pela qual o usuário pode submeter várias tarefas e quando o sistema determinar que possui recurso ele executada a próxima tarefa da fila.

Normalmente em todos os eventos um novo processo é criado, chamado de filho, por um processo já existente, chamado de pai, onde o processo pai executa uma chamada de sistema para cria um novo processo, e consequentemente determina direta ou indiretamente qual programa executar nele. Além disso, quando um processo é criado o processo pai e o filho, possuem diferentes endereços, ou seja, se algum dos dois forem alterados, a mudança não vai interferir no outro.

Um exemplo dos conceitos citados acima é a do terminal básico do UNIX como é possível analisar o Código 1, exposto logo a seguir:

Código 1: Terminal básico no Unix

```
#include <unist.h>
#include <stdio.h>
#define TRUE 1
int main(){
      int pid, status
       while (TRUE){
            type_prompt();
            read_command(comand,param
            eters);
//fork: Cria o clone do processo que
o chamou
     pid = fork();
     if (pid!=0){ //codido pai
          waitpid(-1, &status, 0);
     }
     else{//código do processo filho

           //execpv: Muda a imagem da
     memoria para poder executar um
     novo programa

     execvp(command, parameters);
return 0;
}
```

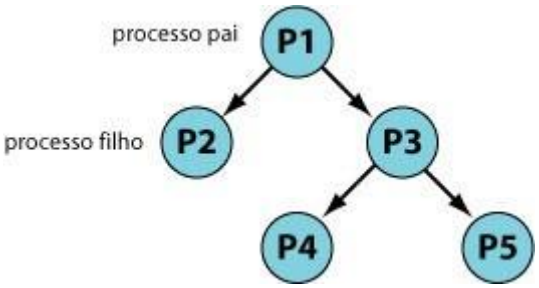

Figura 1: Hierarquia de processo

2.2 Término de Processos

O término de um processo é dado por algumas condições, sendo elas, saída normal, saída por erro, erro fatal e cancelamento por um outro processo, normalmente um processo acaba com a saída normal, ou seja, quando o processo já terminou sua tarefa e o sistema executa uma chamada para eliminar o processo. A saída por erro, pode ser dada por diversos problemas, como um erro encontrado no programa, referência à memória inexistente, entre outros. Já o erro fatal é consequência de uma chamada na qual os arquivos envolvidos diretamente não existem e assim é cancelado imediatamente, porém em programas com interface, é questionado ao usuário se ele quer tentar novamente realizar determinada tarefa ou cancelar, além disso, há o cancelamento por outro processo, ou seja, ocorre uma chamada dizendo para o outro processo terminar, por exemplo a chamada *kill* encontrada no sistema UNIX.

2.3 Estados de Processos

No momento em que um processo é criado, não significa necessariamente que o mesmo será executado imediatamente e que não dependerá de outros processos. Por exemplo: a saída do processo A pode ser a entrada do processo B. Além disso, mesmo que esse processo está

sendo executado, ele pode ser colocado em "espera" com a chegada de um processo com prioridade maior

(vai depender do algoritmo de escalonamento). É nesse cenário, que existem três principais estados que um processo pode assumir (de acordo com Tanenbaum): em execução, pronto e bloqueado, outras fontes literárias separam de uma diferente forma como executável, dormente, zumbi e parado. Não será levantado detalhes do último citado.

O estado em execução é o momento em que a CPU está realmente sendo usada pelo processo em questão. O estado pronto é onde o processo está pronto para ser executado, mas foi temporariamente parado para dar "lugar" para outro processo. Estado bloqueado é uma situação onde o processo fica incapaz de executar até ocorrer uma ação externa. A Figura 2 mostra com mais detalhes a dinâmica dos estados.

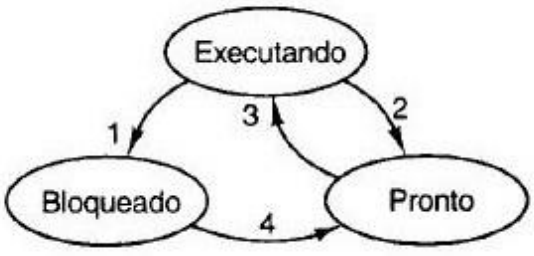

Figura 2: Estados de processo

Como é possível ver na Figura 2, é possível que ocorra quatro transições entre os estados. A primeira ocorre quando o sistema operacional decide que o processo não pode prosseguir. A transição dois e três são causadas pelo escalonador (que terá mais detalhes posteriormente no artigo. A dois acontece quando o escalonador percebe que o processo executou tempo suficiente. A três ocorre quando existe uma política de igualdade de execução entre os processos e o primeiro que já tinha sido executado volta a estar. A quarta transição consiste na dependência de um processo por algum valor de entrada.

2.4 Escalonamento

Tem como objetivo determinar dentre os processos que estão na memória qual processo será associado a CPU, quando esta estiver disponível. Um escalonamento pode ocorrer por quatro principais situações: quando um processo passa do estado executando para o estado espera (por exemplo, o processo solicita uma E/S), no momento em que o processo passa do estado executando para o pronto, passa do estado espera para pronto e quando o processo é finalizado.

O critério para escalonar não é genérico, vai depender do programador que desenvolverá o algoritmo responsável por esta função.

Dentre os critérios para realização de um chaveamento, são *First-Come First-Served* (FCFS), Round Robin (RR), Random e *Shortest Job First* (SJF). Cabe ao programador definir qual método utilizar, tornando a tomada de decisão por qual processo executar.

3. Simulação

A aplicação implementada a partir de um código base fornecido, tem como objetivo simular um escalonador de processos de um Sistema Operacional, utilizando políticas de escalonamento pré-definidas anteriormente.

3.1 Políticas de Escalonamento

As políticas de escalonamento são algoritmos de escalonamento de acordo com critérios escolhidos para o chaveamento de processos.

3.1.1 FCFS

O FCFS (First come, first served, em português, primeiro a chegar é o primeiro a ser servido), é uma política de escalonamento não preemptivo que entrega a CPU os processos por ordem de chegada, onde o primeiro processo que entra é executado até o final sem ser interrompido e quando um novo processo chega, ele entra em uma fila de espera.

3.1.1.1 Implementação

Na implementação do FCFS, levamos em conta algumas características, no bloqueio há o chaveamento de processos, caso haja algum processo pronto e em um desbloqueio, o processo mais antigo retoma a execução

Primeiro inicializamos a estrutura FCFS, fazendo o alocamento, ligando os call-backs à suas rotinas, e atualizando a política com os parâmetros do escalonador

Código 2: Inicializando as estruturas FCFS

```
politica_t * POLITICAFCFS_criar() {
    politica_t* p;
    fcfs_t* fcfs;

    p = malloc(sizeof (politica_t));

    p->politica = POL_FCFS;

    //Ligar os callbacks com as rotinas FCFS
    p->escalonar = FCFS_escalonar;
    p->tick = DUMMY_tick;
    p->novoProcesso = FCFS_novoProcesso;
    p->fimProcesso = FCFS_fimProcesso;
    p->desbloqueado = FCFS_desbloqueado;

    //Alocar a struct que contém os parâmetros para a política fcfs
    fcfs = malloc(sizeof (fcfs_t));

    //inicializar a estrutura de dados fcfs
    fcfs->fifo = LISTA_BCP_criar();

    //Atualizar a política com os parâmetros do escalonador
    p->param.fcfs = fcfs;

    return p;

}
```

Quando um processo chega, ele é inserido na fila do FCFS usando a função “LISTA_BCP_inserir”, como abaixo no código 2:

Código 3: Adicionando Processos FCFS

```
void FCFS_novoProcesso(struct politica_t *p, bcp_t* novoProcesso) {
    //quando um novo processo chega, ele é inserido na fila
    LISTA_BCP_inserir(p->param.fcfs->fifo, novoProcesso);
}
```

Para escalonar processo, e feito a chamada de uma função que usando critérios FCFS, retorna o próximo processo que deve ser executado.

Nesta função inicializamos um ponteiro para o processo que deverá ser retornado para a execução “bcp_t* ret”.

É feita uma verificação se a fila de processos está vazia, caso ela esteja o ponteiro de retorno recebe “null”, indicando que não há processos na lista.

Depois de verificado se a lista de processos está vazia, atribuímos a “ret” o endereço do processo na posição atual da fila, respeitando a politica FCFS, onde o primeiro que chego sempre será verificado. Após isso, fazemos uma verificação se o processo atual da fila está em estado de bloqueio, caso esteja, aumentamos a flag de controle “nBloqueados” e “ret” recebe “null”, e, assim sucessivamente percorremos o vetor com o uso da flag, enquanto a flag for menor que o tamanho da fila de processos, verificamos todas as posições.

Caso o processo da vez não esteja bloqueado, atribuímos a “ret” o endereço deste processo para que ao final da função ele possa ser retornado para ser escalonado. Segue abaixo o código 3 de acordo com as especificações acima.

Código 4: FCFS_escalonar

```
bcp_t* FCFS_escalonar(struct politica_t *p) {
    bcp_t* ret;
    int nBloqueados = 0;

    //Se não há processos na fila fcfs, retornar nenhum
    if (p->param.fcfs->fifo->tam == 0)
        return NULL;

    //testar todos os processos da fila fcfs a partir da posição atual
    while (nBloqueados < p->param.fcfs->fifo->tam) {
        ret = p->param.fcfs->fifo->data[nBloqueados];
        //verificar se o atual da fila fcfs está bloqueado
        if (LISTA_BCP_buscar(bloqueados, ret->pid) != LISTA_N_ENCONTRADO){
            //Se estiver, testar o próximo!
            nBloqueados++;
            ret = NULL;
        } else {
            //retornar o processo para ser executado!
            LISTA_BCP_remover(prontos, ret->pid);
            break;
        }
    }
    return ret;
}
```

Quando um processo termina, é necessário remove-lo da fila FCFS

Código 5: Removendo processos FCFS

```
void FCFS_fimProcesso(struct politica_t *p, bcp_t* processo) {
    //Quando um processo termina, removê-lo da fila fcfs
    LISTA_BCP_remover(p->param.fcfs->fifo, processo->pid);
}
```

3.1.2 Randon

A política Randon, tem como característica principal, escolher um processo aleatório na lista de processos prontos e executa-lo. No bloqueio, ele simplesmente coloca o processo na lista de bloqueados e escolhe outro aleatório para executar e no desbloqueio, o processo é inserido na lista de prontos.

3.1.2.1 Implementação

Assim como no FCFS e todos os outros, inicializamos as estruturas, ligamos call-backs e atualizando a política com os parâmetros do escalonador.

Código 6: Inicializando estruturas Randon

```
politica_t * POLITICARANDOM_criar() {
    politica_t* p;
    random_t* random;

    p = malloc(sizeof (politica_t));

    p->politica = POL_RANDOM;

    //Ligar os callbacks com as rotinas RANDOM
    p->escalonar = RANDOM_escalonar;
    p->tick = DUMMY_tick;
    p->novoProcesso = RANDOM_novoProcesso;
    p->fimProcesso = RANDOM_fimProcesso;
    p->desbloqueado = DUMMY_desbloqueado;

    //Alocar a struct que contém os parâmetros para a política RANDOM
    random = malloc(sizeof (random_t));

    //inicializar a estrutura de dados RANDOM
    random->fifo = LISTA_BCP_criar();

    //Atualizar a política com os parâmetros do escalonador
    p->param.random = random;

    return p;

}
```

Quando um processo chega, ele é inserido na lista de processos à serem escalonados.

Código 7: Adicionando processos Randon

```
void RANDOM_novoProcesso(struct politica_t *p, bcp_t* novoProcesso) {
    //quando um novo processo chega, ele é inserido na Lista
    LISTA_BCP_inserir(p->param.random->fifo, novoProcesso);
}
```

Para escalonar um processo utilizando a política Randon, utilizamos a chamada da função “RANDOM_escalonar”. Nesta função, novamente utilizaremos um ponteiro “ret” que retornará o processo que deverá ser executado (caso haja algum).

Primeiramente, verifica-se se a lista de processos não está vazia, caso ela esteja, retorna “null” e interrompe a função.

Depois de feita a verificação, é escolhido um processo da lista de processos aleatoriamente, é efetuado uma verificação de estado deste processo a fim de descobrir se este não está no estado de bloqueio. Caso o processo estiver bloqueado, incrementa a flag de controle “nBloqueados” e “ret” recebe “null”, assim com o while rodará novamente até achar um processo a ser executado ou chegar ao fim da lista.

Achado um processo à ser executado, “ret” é retirado da lista de prontos e retornado para a execução, como podemos ver no código abaixo no código 8.

Código 8: RANDON escalonar

```
bcp_t* RANDOM_escalonar(struct politica_t *p) {
    bcp_t* ret;
    int nBloqueados = 0;

    //Se não há processos na fila fcfs, retornar nenhum
    if (p->param.random->fifo->tam == 0)
        return NULL;

    //testar todos os processos da fila fcfs a partir da posição atual
    while (nBloqueados < p->param.random->fifo->tam) {
        srand((unsigned) time(NULL));
        ret = p->param.random->fifo->data[(rand() % p->param.random->fifo-
        //verificar se o processo escolhido está bloqueado
        if (LISTA_BCP_buscar(bloqueados, ret->pid) != LISTA_N_ENCONTRADO)
            //Se estiver, testar o próximo!
            nBloqueados++;
            ret = NULL;
        } else {
            //retornar o processo para ser executado!
            LISTA_BCP_remover(prontos, ret->pid);
            break;
        }
    }
    return ret;
}
```

Quando um processo chega ao fim, o escalonador chama a função “RANDON_fimProcesso” para retirar este processo da lista de processos à ser escalonado.

Código 9: RANDOM_fimProcesso

```
void RANDOM_fimProcesso(struct politica_t *p, bcp_t* processo) {
    //Quando um processo termina, removê-lo da lista
    LISTA_BCP_remover(p->param.random->fifo, processo->pid);
}
```

3.1.3 SJF

A política de escalonamento SJF (Shortest Job First, em português Processo mais curto primeiro), é um

Algoritmo de escalonamento em que os menores processos, ou seja, os processos que ocuparem a CPU por menos tempo, terão prioridade e serão executados primeiro.

3.1.3.1 Implementação

Primeiramente inicializamos as estruturas, ligamos os call-backs às suas respectivas rotinas, e atualizamos os parâmetros do escalonado.

Código 10: SJF criar

```
politica_t * POLITICASJF_criar() {
    politica_t* p;
    sjf_t* sjf;

    p = malloc(sizeof (politica_t));

    p->politica = POL_SJF;

    //Ligar os callbacks com as rotinas SJF
    p->escalonar = SJF_escalonar;
    p->tick = DUMMY_tick;
    p->novoProcesso = SJF_novoProcesso;
    p->fimProcesso = SJF_fimProcesso;
    p->desbloqueado = SJF_desbloqueado;

    //Alocar a struct que contém os parâmetros para a política sjf
    sjf = malloc(sizeof (sjf_t));

    //inicializar a estrutura de dados sjf
    sjf->fifo = LISTA_BCP_criar();

    //Atualizar a política com os parâmetros do escalonador
    p->param.sjf = sjf;

    return p;

}
```

Quando um processo chega, ele é inserido na lista processos.

Nesta política, precisamos saber o tempo total em que o processo x irá usar a CPU, para isso atribuímos a variável “timeSlice” do bcp do processo, o tempo total deste processo, a função SJF_CALC_TIME_PROCESSO chamada na função SJF_novoProcesso, faz essa atribuição, como podemos ver no código 11.

Código 11: SJF novo processo

```
void SJF_CALC_TIME_PROCESSO(bcp_t* novoProcesso) {
    novoProcesso->timeSlice = novoProcesso->eventos[novoProcesso->nEventos - 1]->tempo;
}

void SJF_novoProcesso(struct politica_t *p, bcp_t* novoProcesso) {
    //quando um novo processo chega, ele é inserido na fila round robin
    SJF_CALC_TIME_PROCESSO(novoProcesso);
    LISTA_BCP_inserir(p->param.sjf->fifo, novoProcesso);
}
```

Na função SJF_escalonar, precisamos percorrer todos os processos e buscar o que possui o menor tempo, então, criamos variáveis de controle "tempo" e "tempo_anterior = 999999999999999999" para ser usada nesta busca.

Primeiramente verificamos se a lista de processos não está vazia, caso esteja, retorna se "null", caso não esteja vazia, percorremos a lista, pegando processo por processo e verificando se o seu estado atual não é bloqueado, se não for, time recebe o seu tamanho total subtraindo o timeSlice com o seu tempo executado e comparando com o tempo_anterior, se o time deste processo for menor, tempo_anterior recebe o time para assim continuar a verificação processo à processo com o objetivo de achar o menor. A variável "proc" fica responsável por guardar a posição do menor processo desta lista e ao final "ret recebe o processo na posição proc e o retorna para ser executado.

Código 12: SJF escalonar

```
bcp_t* SJF_escalonar(struct politica_t *p) {
    bcp_t* ret;
    int i, proc = 0, nBloqueados = 0;
    uint64_t tempo = 0, tempo_anterior = 999999999999999999;

    //Se não há processos na fila SJF, retornar nenhum
    if (p->param.sjf->fifo->tam == 0)
        return NULL;

    //testar todos os processos da fila fcfs a partir da posição atual
    for (i = 0; i < p->param.sjf->fifo->tam; i++) {
        ret = p->param.sjf->fifo->data[i];
        //verificar se o atual da fila fcfs está bloqueado
        if (LISTA_BCP_buscar(bloqueados, ret->pid) != LISTA_N_ENCONTRADO) {
            //Se estiver, testar o próximo!
            nBloqueados++;
        } else {
            //retornar o processo para ser executado!
            tempo = p->param.sjf->fifo->data[i]->timeSlice -
            p->param.sjf->fifo->data[i]->tempoExecutado;

            if (tempo <= tempo_anterior) {
                tempo_anterior = tempo;
                proc = i;
            }
        }
    }
    if (nBloqueados == p->param.sjf->fifo->tam) {
        ret = NULL;
    } else {
        ret = p->param.sjf->fifo->data[proc];
        LISTA_BCP_remover(prontos, ret->pid);
    }
    return ret;
}
```

Quando um processo chega ao fim ele é retirado da lista de processos.

Código 13: SJF fim processo

```
void SJF_fimProcesso(struct politica_t *p, bcp_t* processo) {
    //Quando um processo termina, removê-lo da fila fcfs
    LISTA_BCP_remover(p->param.fcfs->fifo, processo->pid);
}
```

3.1.4 Filas de Prioridade (FP)

Fila de prioridade, é uma fila de processos com prioridade de 1-40, onde cada posição desta fila possui uma lista de processos com a mesma prioridade e a política que será usada para escalonar estes processos.

Então podemos generalizar um pouco, e chamar à grosso modo esta fila de prioridade de um tipo de gerenciador de escolha de políticas, ou seja, ele faz uma verificação de acordo com a prioridade e manda para o escalonador quais processos será e qual política será usada.

3.1.4.1 Implementação

Primeiramente inicializamos as estruturas e ligamos os call-backs as suas rotinas.

Como o FP só faz o gerenciamento, quando um processo chega, ele verifica qual a política dele e faz a chamada da política responsável para adicionar este processo.

```
void FP_novoProcesso(struct politica_t *p, bcp_t* novoProcesso) {
    if (p->param.fp->filas[novoProcesso->prioridade]->politica == POL_FCFS) {
        FCFS_novoProcesso(p->param.fp->filas[novoProcesso->prioridade], novoProcesso);
    } else if (p->param.fp->filas[novoProcesso->prioridade]->politica == POL_RANDOM) {
        RANDOM_novoProcesso(p->param.fp->filas[novoProcesso->prioridade], novoProcesso);
    } else if (p->param.fp->filas[novoProcesso->prioridade]->politica == POL_RR) {
        RR_novoProcesso(p->param.fp->filas[novoProcesso->prioridade], novoProcesso);
    } else if (p->param.fp->filas[novoProcesso->prioridade]->politica == POL_SJF) {
        SJF_novoProcesso(p->param.fp->filas[novoProcesso->prioridade], novoProcesso);
    }
}
```

Caso a política à ser usado seja Round-Robin, precisaremos da função Tick, para diminuir o tempo executado do processo

Código 14: FP criar

```
void FP_tick(struct politica_t *p) {
    if (executando) {
        if (p->param.fp->filas[executando->prioridade]->politica == POL_RR)
            RR_tick(p->param.fp->filas[executando->prioridade]);
        } else DUMMY_tick(p);
    }
}
```

Para escalonar, o FP_escalonar verifica qual a política será utilizada no escalonamento e faz a chamada do escalonar da política como já vimos acima.

Código 15: FP escalonar

```
bcp_t* FP_escalonar(struct politica_t *p) {
    int i;
    bcp_t* ret;
    for (i = 0; i < p->param.fp->faixa_max; i++) {
        if (p->param.fp->filas[i]->politica == POL_FCFS) {
            if (p->param.fp->filas[i]->param.fcfs->fifo->tam > 0) {
                ret = FCFS_escalonar(p->param.fp->filas[i]);
                break;
            } else ret = NULL;
        }
        else if (p->param.fp->filas[i]->politica == POL_RANDOM) {
            if (p->param.fp->filas[i]->param.random->fifo->tam > 0) {
                ret = RANDOM_escalonar(p->param.fp->filas[i]);
                break;
            } else ret = NULL;
        }
        else if (p->param.fp->filas[i]->politica == POL_RR) {
            if (p->param.fp->filas[i]->param.rr->fifo->tam > 0) {
                ret = RR_escalonar(p->param.fp->filas[i]);
                break;
            } else ret = NULL;
        }
        else if (p->param.fp->filas[i]->politica == POL_SJF) {
            if (p->param.fp->filas[i]->param.sjf->fifo->tam > 0) {
                ret = SJF_escalonar(p->param.fp->filas[i]);
                break;
            } else ret = NULL;
        }
    }
    return ret;
}
```

Quando acontece um desbloqueio, chamamos a função da política utilizada responsável por tratar esse problema.

Código 16: FP desbloquear

```
void FP_desbloqueado(struct politica_t *p, bcp_t* processoBloq) {
    if (p->param.fp->filas[processoBloq->prioridade]->politica == POL_FCFS) {
        FCFS_desbloqueado(p->param.fp->filas[processoBloq->prioridade], proces
    } else if (p->param.fp->filas[processoBloq->prioridade]->politica == POL_S
        SJF_desbloqueado(p->param.fp->filas[processoBloq->prioridade], process
    }
}
```

Quando um processo chega ao fim, novamente verificamos a política responsável e chamamos a função de fim de processo da política.

Código 17: FP fim processo

```
void FP_fimProcesso(struct politica_t *p, bcp_t* processo) {
    if (p->param.fp->filas[processo->prioridade]->politica == POL_FCFS) {
        FCFS_fimProcesso(p->param.fp->filas[processo->prioridade], processo);
    } else if (p->param.fp->filas[processo->prioridade]->politica == POL_RANDOM) {
        RANDOM_fimProcesso(p->param.fp->filas[processo->prioridade], processo);
    } else if (p->param.fp->filas[processo->prioridade]->politica == POL_RR) {
        RR_fimProcesso(p->param.fp->filas[processo->prioridade], processo);
    } else if (p->param.fp->filas[processo->prioridade]->politica == POL_SJF) {
        SJF_fimProcesso(p->param.fp->filas[processo->prioridade], processo);
    }
}
```

3.2 Saída

Na saída, devemos escrever em um arquivo de saída todas as informações do escalonamento dos processos, contendo as seguintes informações:

- Número de chaveamento
- Tempo médio de espera
- Tempo médio de retorno
- Vazão (quantidade de processos terminados a cada 1000 unidades de tempo)
- Sequência de termino dos processos
- Diagrama de execução

Tentando deixar mais simples a implementação de um código para criar este arquivo de saída, criamos um arquivo de cabeçalho (arq_saida.h), onde nele definimos as estruturas e funções que utilizamos nesta criação.

No arq_saida.h, criamos uma estrutura para utilizar no cálculo do TMR(tempo médio de retorno), onde salvamos o tempo da primeira e última execução.

Código 18: estrutura TMR

```
typedef struct TMR_processos {
    int PID;
    uint64_t tPrimeiraExec;
    uint64_t tUltimaExec;
} TMR_processos;
```

Criamos uma estrutura para os tipos de eventos de um processo, implementando os tipos de eventos, são eles criação, execução, termino, bloqueio, desbloqueio e quantum ex (quando acaba o tempo que ele recebeu para usar a CPU).

Código 19: Eventos

```
typedef enum EVENTO_SAIDA {
    EVT_CRIACAO,
    EVT_EXEC,
    EVT_TERMIN,
    EVT_QUANTUM_EX,
    EVT_BLOQ,
    EVT_DESBLOQ
} EVENTO_SAIDA;
```

Estrutura para salvar o diagrama de eventos, onde nele tem o pid do processo, os eventos que ele possui e o tempo de cada evento.

Código 20: Diagrama de eventos

```
typedef struct diagrama_evento {
    EVENTO_SAIDA evento;
    int PID;
    uint64_t TTT;
} diagrama_evento;
```

Estrutura que irá escrever no arquivo de saída as informações dos chaveamentos.

Código 21: atributos do arquivo de saída

```
typedef struct arq_saida_t {
    TMR_processos** tmr_processos;
    diagrama_evento** evento;
    float TME;
    uint64_t TMR;
    float VAZAO;
    int *TERMINO;
    int evento_n;
    int alocacao_evento_n;
    int chaveamentos_n;
    int termino_n;
    int tmr_n;
} arq_saida_t;
```

E por fim, os escopos das funções principais que usamos na criação do arquivo de saída.

Código 22: escopos das funções

```
arq_saida_t* CRIA_arq_saida(arq_processos_t* arq_processos);
void CRIA_ADC_evento(arq_saida_t* arq_saida,
 char* evento, uint64_t relogio, bcp_t* processo);

void GRAVAR_saida(arq_saida_t* saida, char* arq_saida, int nProcessos);
void CALCULA_tme();
void CALCULA_tmr();
void CALCULA_vazao();
```

Depois de ter definido as estruturas e o escopos das funções, criamos um novo arquivo arq_processos.c onde implementamos os comportamentos necessários para escrever no arquivo os dados necessários. Atribuindo à estrutura de atributos do arquivo de saída os seus respectivos valores buscando dos processos que foram escalonados e da politica utilizada.

Logo após preencher a estrutura, a função gravar saída fica responsável por escrever essas informações no arquivo de saída.

4 Conclusão

Com tudo isso, podemos concluir que, em um chaveamento de processos existe diversos algoritmos para poder escalonar estes processos e que cada algoritmo utiliza uma política diferente e com a implementação desta simulação podemos ver mais de perto como realmente é escrito esses algoritmos e como eles funcionam.