Análise Comportamental do Escalonador de Processos

Santa Maria, 19 de Novembro de 2017

Nome: Victor Dallagnol Bento
Matrícula: 201520835

Resumo: *O escalonador de processos em questão tratará de uma comparação referente a duas hipóteses de experimento. O grau de multiprogramação será alterado para que ocorra uma análise quanto a interferência dessa variável no desempenho do programa.*

1. Introdução

Os procedimentos analisados a seguir serão desenvolvidos com o intuito de verificar as mudanças nos dados de saída de um escalonador quando este tem grau de multiprogramação distinto. Este valor, passado como parâmetro através de um arquivo, coordenará o funcionamento de todo o sistema e serve como referência para inúmeros testes dentro do próprio programa.

2. Funcionamento do Código

O código criado está dividido em doze arquivos do tipo *.c* e um do tipo *.h*. O arquivo *sistema.h* possui as definições de variáveis e todas as estruturas necessárias para implementação do código, tanto as definidas como flags, quanto as variáveis de cada processo, instâncias das estruturas e filas criadas. O *inputs.c* possui as funções que recebem os valores lidos do arquivo de entrada (Tempo total de simulação, Grau de multiprogramação e Fatia de tempo para escalonamento Round Robin) e os atribui às suas respectivas variáveis instanciadas no *sistema.h*. Ademais, os arquivos *arrive.c, ready.c* e *io.c* contém as devidas funções específicas de cada fila, como inserção, remoção, tamanho, print, entre outras.

main.c():

Na função *main.c* foram setadas as flags globais para zero. Além disso, é chamada a função *PegaEntradas()* para receber os valores lidos do arquivo txt. São criadas as três filas (Arrive, Ready e I/O). É aberto um arquivo para que nele sejam gravados os dados de saída e o trace da execução. Em seguida, a parte principal que compõem a estrutura *main.c* é o laço `while` que vai coordenar toda a execução do código. Sua repetição ocorre até que o Tempo de Execução da Simulação seja atingido. É por esse fato que, após cada chamada de função dentro do laço, ocorre um teste com essa variável `TempoSimulado`. As funções chamadas serão vistas a seguir, separadamente. Contudo, vale ressaltar que a ordem da chamada dessas funções faz parte da lógica do programa, pois estas são totalmente dependentes entre si. Ademais, está inserido na função o `printf` dos dados relevantes para a verificação do andamento da simulação ao final da execução do `while`, junto com a escrita dos dados no arquivo de saída. Após isso, o arquivo é fechado. Todas essas especificações podem ser vistas na **Figura 1.**

```c
int main(int arg, char* argv[])
{
    fl.TempoSimulado = 0;                       // variável para controlar o tempo de simulação
    fl.TamanhoArrive = 0;                       // variável para controlar o tamanho da fila de chegada
    fl.j = 0;                                   // define a ID dos processos
    fl.CPUStatus = -1;                          // CPU Ociosa
    fl.ProcessosNoSistema = 0;                  // Nenhum processo no sistema
    fl.IOStatus = 0;                            // I/O ocioso
    fl.IO_completa = 0;
    fl.timeCPUUsada = 0;
    fl.timeIO = 0;
    fl.somaTimeExecProcess = 0;
    fl.PReady = 0;
    fl.PIO = 0;
    fl.TArrive = 0;
    fl.TReady = 0;
    fl.TIO = 0;

    PegaEntradas();                                 // chama função para pegar entradas

    Filas* chegada = cria_arrive();                 // cria fila de CHEGADA
    Filas* prontos = cria_ready();                  // cria fila de PRONTOS
    Filas* io = cria_io();                          // cria fila de I/O


    outputs = fopen("outputs.txt", "wt");               // Criação do Arquivo para saidas
    if (outputs == NULL)
    {
        printf("Problemas na CRIACAO do arquivo\n");
        return 0;
    }
    trace = fopen("trace.txt", "wt");               // Criação do Arquivo para Trade de execução dos Processos
    if (trace == NULL)
    {
        printf("Problemas na CRIACAO do arquivo\n");
        return 0;
    }

    while (fl.TempoSimulado < entrada.TempoSimulacao)       // while para simulação
    {
        CriarProcessos(chegada);                            // Cria processo na ARRIVE
        if(fl.TempoSimulado == entrada.TempoSimulacao)      // Testa se tempo de simulação já foi atingido
            break;
        print_arrive(chegada);
        if(fl.TempoSimulado == entrada.TempoSimulacao)
            break;
        confere_CPU(prontos);                               // Seta as flags para saber situação atual da CPU
        if(fl.TempoSimulado == entrada.TempoSimulacao)
            break;
        organizaFilas(chegada, prontos);                    // Organiza as filas conforme a situação da CPU
        if(fl.TempoSimulado == entrada.TempoSimulacao)
            break;
        insere_CPU();                                       // Coloca/Mantém processos na CPU
        if(fl.TempoSimulado == entrada.TempoSimulacao)
            break;
        fluxos(prontos, io);                                // Função para os fluxo que o processo pode seguir
        if(fl.TempoSimulado == entrada.TempoSimulacao)
            break;
        confere_IO(io);                                     // Seta as flags para saber a situação atual do I/O
        if(fl.TempoSimulado == entrada.TempoSimulacao)
            break;
        insere_IO(io, prontos);                             // Coloca processos da fila de I/O para o dispositivo I/O
        if(fl.TempoSimulado == entrada.TempoSimulacao)
            break;
    }
```

```c
    // Prints para ver saidas
    printf("\nPROCESSOS TERMINADOS: %d\n", fl.ProcessosCompletos);
    printf("\nOPERAÇÕES DE I/O COMPLETAS: %d\n", fl.IO_completa);
    printf("\nTEMPO DE CPU USADA: %.2f, %.2f(PORCENTO)\n", fl.timeCPUUsada, 100*(fl.timeCPUUsada/(fl.timeCPUUsada+fl.timeCPUOciosa)));
    printf("\nTEMPO DE CPU OCIOSA: %.2f, %.2f(PORCENTO)\n", fl.timeCPUOciosa, 100*(fl.timeCPUOciosa/(fl.timeCPUUsada+fl.timeCPUOciosa)));
    printf("\nTEMPO DE I/O USADO: %.2f, %.2f(PORCENTO)\n", fl.timeIO, 100*(float)(fl.timeIO/(fl.TempoSimulado)));
    printf("\nTEMPO MÉDIO DE ESPERA ENTRE PROCESSOS: %.2f\n", (fl.somaTimeExecProcess/fl.ProcessosCompletos));
    printf("\nTEMPO MÉDIO NA FILA ARRIVE: %.2f\n", (fl.Time_arrive/fl.ProcessoCriado));
    printf("\nTEMPO MÉDIO NA FILA READY: %.2f\n", (fl.Time_ready/fl.PReady));
    printf("\nTEMPO MÉDIO NA FILA IO: %.2f\n", (fl.Time_io/fl.PIO));
    printf("\nTAMANHO MÉDIO NA FILA ARRIVE: %.2f\n", (fl.TArrive/fl.ProcessoCriado));
    printf("\nTAMANHO MÉDIO NA FILA READY: %.2f\n", (fl.TReady/fl.PReady));
    printf("\nTAMANHO MÉDIO NA FILA IO: %.2f\n", (fl.TIO/fl.PIO));

    // Atribuição das saidas para arquivo
    fprintf (outputs, "PROCESSOS TERMINADOS: %d\n", fl.ProcessosCompletos);
    fprintf (outputs, "OPERAÇÕES DE I/O COMPLETAS: %d\n", fl.IO_completa);
    fprintf (outputs, "TEMPO DE CPU USADA: %.2f, %.2f(PORCENTO)\n", fl.timeCPUUsada, 100*(fl.timeCPUUsada/(fl.timeCPUUsada+fl.timeCPUOciosa)));
    fprintf (outputs, "TEMPO DE CPU OCIOSA: %.2f, %.2f(PORCENTO)\n", fl.timeCPUOciosa, 100*(fl.timeCPUOciosa/(fl.timeCPUUsada+fl.timeCPUOciosa)));
    fprintf (outputs, "TEMPO DE I/O USADO: %.2f, %.2f(PORCENTO)\n", fl.timeIO, 100*(float)(fl.timeIO/(fl.TempoSimulado)));
    fprintf (outputs, "TEMPO MÉDIO DE ESPERA ENTRE PROCESSOS: %.2f\n", (fl.somaTimeExecProcess/fl.ProcessosCompletos));
    fprintf (outputs, "TEMPO MÉDIO NA FILA ARRIVE: %.2f\n", (fl.Time_arrive/fl.ProcessoCriado));
    fprintf (outputs, "TEMPO MÉDIO NA FILA READY: %.2f\n", (fl.Time_ready/fl.PReady));
    fprintf (outputs, "TEMPO MÉDIO NA FILA IO: %.2f\n", (fl.Time_io/fl.PIO));
    fprintf (outputs, "TAMANHO MÉDIO NA FILA ARRIVE: %.2f\n", (fl.TArrive/fl.ProcessoCriado));
    fprintf (outputs, "TAMANHO MÉDIO NA FILA READY: %.2f\n", (fl.TReady/fl.PReady));
    fprintf (outputs, "TAMANHO MÉDIO NA FILA IO: %.2f\n", (fl.TIO/fl.PIO));

    fclose(outputs);
    fclose(trace);

    return 0;
}
```

Figura 1: Função *main.c.*

CriarProcessos():

A primeira função chamada pela *main.c* é a *CriarProcessos()*. Nela, o tempo de CPU ociosa é verificado e incrementado se necessário. Ademais, o tamanho da fila Arrive foi estabelecido com o valor limite sendo o do Grau de Multiprogramação do sistema. Dessa forma, no tempo 0, a fila é preenchida, e, conforme processos forem sendo retirados, são criados processos novos. A criação de um processo é feita pela chamada de *Processo* `teste`. O processo do tipo `teste` é inserido na fila Arrive e seu estado é definido como `NEW` assim como seus campos são inicializados, como sua ID, seu tempo de execução, seu tempo de solicitar I/O, entre outros. Flags para cálculos futuros são incrementadas. Ver **Figura 2.**

```c
void CriarProcessos(Filas* fila){
    if(fl.CPUStatus == -1){
        fl.timeCPUOciosa++;
    }
    while(fl.TamanhoArrive < entrada.ML){              // Define um tamanho máximo para fila ARRIVE
        Processo teste;                                // Cria um processo NULL
        if (insere_arrive(fila, teste)){               // Insere e seta os valores do processo na ARRIVE
            fl.ProcessoCriado++;                       // Nº de processos criados aumenta
        }else{
            break;                                     // Limite máximo da fila atingido
        }
        fl.TamanhoArrive++;                            // Aumenta o tamanho da rive por processo criado
        fl.TArrive += tamanho_arrive(fila);
        }
}
```

Figura 2: Função *CriarProcessos*.

confere_CPU():

Após o processo ser criado, a função chamada na execução do laço `while`, é a função *confere_CPU()*. Ela é responsável por setar as flags conforme o estado da CPU (ociosa ou ocupada). Essa análise de flags será feita posteriormente em outra função responsável por inserir processos na CPU. O código pode ser visto na **Figura 3**.

```c
void confere_CPU(Filas* r){                         // Recebe fila READY para saber se esta vazia

    if (fl.CPUStatus == -1)             // CPU ociosa
    {
        fl.removerChegada = 1;          // Processo deve sair da fila de chegada
        fl.inserirReady = 1;            // Processo removido da fila chegada deve ir p fila de prontos
        fl.fazerCPU = 1;                // Primeiro processo da fila de prontos deve ir para CPU
    }
    else                                // CPU ocupada
    {
        fl.removerChegada = 1;          // Processo deve sair da fila de chegada
        fl.inserirReady = 1;            // Processo removido da fila chegada deve ir p fila de prontos
        fl.fazerCPU = 0;                // flag em 0, indica que o processo nao pode acessar a cpu
    }
}
```

Figura 3: Função *confere_CPU.*

organizaFilas():

Seguindo o fluxo da execução no laço principal, a próxima função chamada após as flags serem setadas é a *organizaFilas().* Ela será responsável por reorganizar os processos nas filas. Primeiramente, ela confere o estado da CPU, e se esta estiver ociosa, o tempo de CPU ociosa é incrementado. Ademais, ela irá utilizar as flags setadas anteriormente em suas condições de execução. Caso a CPU esteja ocupada e o número de processos no sistema ainda não atingiu o Grau de Multiprogramação, o processo será removido da fila Arrive e inserido na fila Ready(estado é atualizado para `READY`), através de uma variável temporária `temp1`. São atualizadas flags de contadores utilizados posteriormente em cálculos. Porém, caso a CPU esteja ociosa e o Grau de Multiprogramação não tenha sido atingido, o processo também será removido da fila Arrive e inserido na fila Ready(variável `temp1`), sendo retirado da Ready logo em seguida (variável `temp2`). Flags para uso posterior também são setadas nesta etapa. A utilização de duas variáveis temporárias distintas se fez necessária pelo fato de a variável `temp2` se referir ao processo que entrará na CPU. Caso o processo permaneça lá por mais de um ciclo, essa variável não poderá ser sobrescrita. Caso fosse utilizada a `temp1` para isso, ela seria sobrescrita assim que um processo fosse removido da fila Arrive e a referência seria perdida. Ver **Figura 4**.

```c
void organizaFilas(Filas* c, Filas* p){
    if(fl.CPUStatus == -1){
        fl.timeCPUOciosa++;
    }
    temp1 = (Processo*) malloc(sizeof(Processo));               // alocação do processo

    if(fl.removerChegada == 1 && fl.inserirReady == 1 && fl.fazerCPU == 0)          // CPU esta ocupada e processo precisa ser removido da ARRIVE para READY
    {
        if (fl.ProcessosNoSistema < entrada.ML)                                     // Verifica se numero de processos no sistema foi atingido
        {
            *temp1 = remove_arrive(c);          // temporário recebe processo removido da fila de ARRIVE
            fl.TamanhoArrive--;
            fl.TempoSimulado++;
            temp1->waitingTime_arrive_saida = fl.TempoSimulado;                 // Tempo de saida da fila do Processo é atualizada
            temp1->waitingTime_arrive += temp1->waitingTime_arrive_saida - temp1->waitingTime_arrive_chegada;      // Tempo de espera do processo na fila ARRIVE
            fl.Time_arrive += temp1->waitingTime_arrive;        // Tempo de espera da fila ARRIVE é atualizado
            insere_ready(p, *temp1);            // temporario é inserido na fila READY
            fl.TReady += tamanho_ready(p);      // Pega tamanho da READY
            fl.PReady++;                        // Mais um processo na READY
            fl.ProcessosNoSistema++;             // Manutenção do grau de multiprogramação (ML)
           }
        print_ready(p);
    }
```

```
    else          // CPU desocupada
    {
        if (fl.ProcessosNoSistema < entrada.ML)        // Verifica se numero de processos no sistema foi atingido
        {
            *temp1 = remove_arrive(c);                 // temporario recebe processo removido da ARRIVE
            fl.TamanhoArrive--;
            fl.ProcessosNoSistema++;                   // Manutenção do grau de multiprogramação (ML)
            fl.TempoSimulado++;
            temp1->waitingTime_arrive_saida = fl.TempoSimulado;                // Tempo de saida da fila do Processo é atualizada
            temp1->waitingTime_arrive += temp1->waitingTime_arrive_saida - temp1->waitingTime_arrive_chegada;      // Tempo de espera do process
            fl.Time_arrive += temp1->waitingTime_arrive;        // Tempo de espera da fila ARRIVE é atualizado
            insere_ready(p, *temp1);                           // temporario é inserido na READY
            fl.TReady += tamanho_ready(p);      // Pega tamanho da READY
            fl.PReady++;                        // Mais um processo na READY
        }
            temp2 = (Processo*) malloc(sizeof(Processo));          // alocação dinamica do temporário2
            print_ready(p);
            *temp2 = remove_ready(p);                              // Temporario2 Recebe o processo retirado da READY pois o mesmo ira para CPU
            temp2->waitingTime_ready_saida = fl.TempoSimulado;      // Tempo de saida da fila do Processo é atualizada
            temp2->waitingTime_ready += temp2->waitingTime_ready_saida - temp2->waitingTime_ready_chegada;      // Tempo de espera do processo r
            fl.Time_ready =+ temp2->waitingTime_ready;        // Tempo de espera da fila READY é atualizado
            fl.auxiliar = 0;
        }
}
```

Figura 4: Função *organizaFilas*.

insere_CPU():

Em seguida, temos a função *insere_CPU(),* responsável pela execução do processo na CPU. Ao acessar a CPU, o seu estado muda para ocupado e o tempo de execução do processo assim como o tempo que o mesmo solicita I/O são decrementados e checados através de uma variável `auxiliar`, que posteriormente é comparada com o valor de entrada responsável pela fatia do Round Robin.

O estado do processo é atualizado (`RUNNING`) e dependendo de seu resultado, o processo pode continuar na CPU ou seguir para um dos três fluxos (`A`, `B` ou `C`), dependendo do valor de suas devidas flags, como indica a **Figura 5**. Se o seu tempo de execução na CPU não acabou, nem sua fatia de tempo e nem a solicitação do I/O, o processo deve permanecer na CPU(flags resetadas). Caso seu tempo de solicitação I/O termine antes de sua fatia de tempo, o processo segue para o fluxo C. De maneira contrária, quando seu tempo de execução termina antes de sua solicitação I/O, o processo segue para o fluxo A. Quando sua fatia de tempo na CPU acaba, e o processo não solicita I/O, nem termina seu tempo de execução, ele segue para o fluxo B.

```
void insere_CPU(){
    fl.CPUStatus = 1;                           // CPU esta ocupada
    temp2->timeExec--;                          // Decrementa fatia de tempo do processo
    temp2->solicitaIO--;                        // Decrementa Solicitação do I/O
    fl.auxiliar++;                              // Auxiliar é comparado com fatia RR
    fl.TempoSimulado++;
    fl.timeCPUUsada++;

    // Atualiza situação do processo
    strcpy(temp2->ciclo, "RUNNING");
    printf("{%d} P[%d] -> %s\n", fl.TempoSimulado, temp2->ID, temp2->ciclo);
    fprintf (trace, "{%d} P[%d] -> %s\n", fl.TempoSimulado, temp2->ID, temp2->ciclo);

    // Caso processo permaneça na CPU, flags são zeradas
    fl.B = 0;
    fl.C = 0;
    fl.A = 0;

//fluxo B Volta pra ready
    if(fl.auxiliar == entrada.FatiaTempo && temp2->timeExec > 0){          // Se tempo da fatia acabou mas o tempo do processo não
        fl.B = 1;
        fl.C = 0;
        fl.A = 0;

//fluxo A Sai do sistema
    }else if(temp2->timeExec == 0){                             // Se o tempo de execução acabou
        fl.B = 0;
        fl.C = 0;
        fl.A = 1;
```

```
//fluxo C Faz IO
    }else if(temp2->solicitaIO == 0){                    // Tempo de solicitação do I/O atingido
        fl.B = 0;
        fl.C = 1;
        fl.A = 0;
    }
}
```

Figura 5: Função *insere_CPU*.

fluxos():

Dependendo do fluxo definido na função anterior, a função *fluxos.c* seguirá comportamentos distintos. Caso a flag A esteja setada, significa que o processo terminou e sairá do sistema. Dessa forma, seu estado muda para `TERMINATED` e inúmeras flags são atualizadas. A variável `temp2` é liberada e são gravados no arquivo os valores referentes aos processos terminados.

Caso o fluxo a se seguir seja o B, ou seja, voltar para a fila Ready, o processo sai da CPU e é inserido na fila novamente (*insere_ready()*), voltando para o estado `READY`. Flags são setadas e a variável `temp2` é liberada da alocação.

Finalmente, caso o fluxo referido seja o C, significa que o processo solicitou I/O, e deve ir para a fila I/O. É chamada a função *insere_io()* e o processo passa para o estado `BLOCKED`. A variável `temp2` é liberada e flags são atualizadas. Caso nenhuma das flags (A, B ou C) seja compatível, ou seja, não é necessária nenhuma ação em *fluxos.c*, nada acontece e a execução continua normalmente. O esquema do código pode ser visto na **Figura 6**.

```c
void fluxos(Filas* p, Filas* i){

    if (fl.A)
    {
        fl.TempoSimulado++;
        strcpy(temp2->ciclo, "TERMINATED");                // Atualiza situação do processo
        printf("{%d} P[%d] -> %s\n", fl.TempoSimulado, temp2->ID, temp2->ciclo);
        fprintf (trace, "{%d} P[%d] -> %s\n", fl.TempoSimulado, temp2->ID, temp2->ciclo);   // salva no arquivo
        temp2->turnaroundSaida = fl.TempoSimulado;                 // Salva tempo em que processo terminou
        printf("*** TEMPO DE ESPERA DO PROCESSO[%d] = %d\n", temp2->ID, (temp2->waitingTime_arrive + temp2->waitingTime_ready + temp2->waitingTime_io));
        printf("*** PROCESSO: %d - TURNAROUND: %d\n", temp2->ID, temp2->turnaroundSaida - temp2->turnaroundEntrada);
        fprintf (outputs, "TEMPO DE ESPERA DO PROCESSO[%d] = %d\n", temp2->ID, (temp2->waitingTime_arrive + temp2->waitingTime_ready + temp2->waitingTime_io));
        fprintf (outputs, "PROCESSO: %d - TURNAROUND: %d\n", temp2->ID, temp2->turnaroundSaida - temp2->turnaroundEntrada);
        fl.somaTimeExecProcess += (temp2->waitingTime_arrive + temp2->waitingTime_ready + temp2->waitingTime_io);       //  Tempo total de espera do Processo
        free(temp2);                        // Libera temporário
        fl.ProcessosCompletos++;
        fl.ProcessosNoSistema--;
        fl.CPUStatus = -1;                      // CPU volta a ficar ociosa pois o processo acaba
        fl.timeCPUOciosa++;
        print_ready(p);
    }
    else if(fl.B)
    {
        fl.TempoSimulado++;
        insere_ready(p, *temp2);                // Temporario2 é inserido na fila READY
        fl.TReady += tamanho_ready(p);          // Pega tamanho da READY toda vez que um processo é inserido
        fl.PReady++;                            // Processo na READY aumenta
        print_ready(p);
        free(temp2);
        fl.CPUStatus = -1;                  // CPU fica ociosa pois o processo vai para READY
        fl.timeCPUOciosa++;
    }
    else if (fl.C)
    {
        fl.TempoSimulado++;
        insere_io(i, *temp2);               // Temporario2 é inserido na fila do I/O
        fl.TIO += tamanho_io(i);            // Pega tamanho da I/O toda vez que um processo é inserido
        fl.PIO++;                           // Processo na I/O aumenta
        print_io(i);
        free(temp2);
        fl.CPUStatus = -1;                  // CPU fica ociosa pois processo vai para fila do I/O
        fl.timeCPUOciosa++;
    }
}
```

Figura 6: Função *fluxos*.

confere_IO():

A função *confere_IO()* verifica se o I/O está ou não ocupado e se a fila do I/O está vazia ou não. Dependendo destas condições, algumas flags serão setadas, como mostra a **Figura 7**.

Caso o dispositivo I/O esteja desocupado e a fila I/O esteja vazia, não é preciso fazer I/O. Se o dispositivo I/O estiver vazio e a fila de I/O não estiver vazia, um processo precisa ser retirado da fila e enviado para fazer I/O. Se o I/O estiver sendo ocupado por um processo, não é preciso retirar nenhum processo da fila de I/O e o processo que está nele continua executando.

```
void confere_IO(Filas *i){                          // Recebe fila I/O para saber se esta vazia
    if (fl.IOStatus == 0 && io_vazia(i) == 1)                  // I/O Desocupado e Fila vazia
    {
        fl.removerIO = 0;               // Não remove da fila I/O
        fl.fazerIO = 0;                 // Não faz I/O
    }
    else if (fl.IOStatus == 0 && io_vazia(i) == 0)             // I/O desocupado e a fila não está vazia
    {
        fl.removerIO = 1;               // Processo precisa ser removido da fila de I/O
        fl.fazerIO = 1;                 // Faz I/O
    }
    else                            // I/O está ocupado
    {
        fl.removerIO = 0;           // Não remove da fila
        fl.fazerIO = 1;             // Continua fazendo I/O
    }
}
```

Figura 7: Função *confere_IO*.

insere_IO():

Por fim, a última função chamada no laço `while` é *insere_IO()*. Nessa função, o valor das flags setadas no *confere_IO()* serão utilizadas. É conferido o estado da CPU para incrementar ou não seu tempo ociosa. A flag verificada primeiramente é *fazerIO*. No `case 1`, é feito um teste para verificar a flag *removerIO*. Caso o valor seja 1, uma terceira variável temporária (`temp3`) é alocada para retirar o processo da fila I/O e colocá-lo no dispositivo de I/O (esse trecho de código simula um atendimento de I/O pois vai decrementando o tempo que o processo em questão leva no I/O). Outras flags serão setadas e o teste de atendimento do I/O é feito, para que, caso seu tempo no dispositivo tenha terminado, ele saia de `BLOCKED` e volte novamente para a fila de prontos (`READY`) e `temp3` é liberada, caso contrário, ele permanece no dispositivo por mais uma unidade de tempo.

Porém, se o valor de *removerIO* for 0, significa que o dispositivo está ocupado e não deve ser retirado nenhum processo da fila de I/O, apenas continuar a execução do processo contido na `temp3`. Se o processo terminar, retorna para a fila de prontos (READY).

Em `case 0`, não é necessário executar nada nessa função, pois não há processos na fila nem no dispositivo I/O. A descrição do código está na **Figura 8**. A função termina e o laço `while` recomeça.

```
void insere_IO(Filas* i, Filas* r){                  // alocação dinamica do Temporario3
    if(fl.CPUStatus == -1){
        fl.timeCPUOciosa++;
    }
    switch(fl.fazerIO)                      // Confere a flag para fazer ou não I/O
    {
    case 1:
        if(fl.removerIO == 1){              // Se tiver que remover da fila do I/O
            temp3 = (Processo*) malloc(sizeof(Processo));
            *temp3 = remove_io(i);          // Remove processo da fila I/O e salva no temp3
            temp3->waitingTime_io_saida = fl.TempoSimulado;         // Salva tempo de saiva da fila de I/O do Processo
            temp3->waitingTime_io += temp3->waitingTime_io_saida - temp3->waitingTime_io_chegada;       // Faz calculo do tempo de espera do Processo na fila I/O
            fl.Time_io += temp3->waitingTime_io;        // Atribui tempo de espera da fila I/O
            fl.IOStatus = 1;                // I/O esta ocupado
            fl.TempoSimulado++;
            strcpy(temp3->ciclo, "BLOCKED-DISP(I/O)");                  // Atualiza situação do processo
            printf("{%d} P[%d] -> %s\n", fl.TempoSimulado, temp3->ID, temp3->ciclo);
            fprintf (trace, "{%d} P[%d] -> %s\n", fl.TempoSimulado, temp3->ID, temp3->ciclo);   // salva no arquivo
            temp3->atendimentoIO--;
            fl.timeIO++;
            if(temp3->atendimentoIO == 0){          // Se atendimento do I/O acabar
                insere_ready(r, *temp3);            // Insere temp3 na READY
                fl.TReady += tamanho_ready(r);      // Pega tamanho da READY
                fl.PReady++;                        // Mais um processo inserido na READY
                free(temp3);
                fl.IOStatus = 0;                    // I/O fica desocupado
                fl.IO_completa++;
            }
        }else{                                  // Se não tiver que remover da fila do I/O
            fl.IOStatus = 1;                        // I/O esta ocupado
            fl.TempoSimulado++;
            temp3->atendimentoIO--;
            fl.timeIO++;
            strcpy(temp3->ciclo, "BLOCKED-DISP(I/O)");              // Atualiza situação do processo
            printf("{%d} P[%d] -> %s\n", fl.TempoSimulado, temp3->ID, temp3->ciclo);
            fprintf (trace, "{%d} P[%d] -> %s\n", fl.TempoSimulado, temp3->ID, temp3->ciclo);
            if(temp3->atendimentoIO == 0){          // Se atendimento do I/O acabar
                insere_ready(r, *temp3);            // Insere temp3 na READY
                fl.TReady += tamanho_ready(r);      // Pega o tamanho da READY
                fl.PReady++;                        // Mais um processo inserido na READY
                free(temp3);
                fl.IOStatus = 0;                    // I/O fica desocupado
                fl.IO_completa++;
            }
        }
        break;
    case 0:
        break;
    }
}
```

Figura 8: Função *insere_IO*.

3. Experimento e Análise

3.1 Grau de Multiprogramação 15, Fatia de Tempo 8, Tempo de Simulação 500 :

Para os referidos valores de entrada e um tempo de execução de 500, simulou-se o programa com o intuito de analisar-se os valores obtidos. O número total de processos terminados durante a execução foram de 20 processos. Esse dado foi obtido com base na flag de *ProcessosCompletos* que é incrementada na execução do fluxo A, ou seja, quando um processo sai do sistema.

Os processos que solicitaram I/O e tiveram sua execução concluída, foram 24 processos. Esses valores são obtidos pela flag *IO_completa* que é incrementada dentro da condição de verificação do tempo de atendimento do I/O.

O tempo de CPU utilizada na execução de processos foi de 215 (48.31%), enquanto que o tempo em que ela estava ociosa foi de 230 (51.69%). O tempo de I/O utilizado foi de 193 (38.6%). Para o primeiro caso foi incrementada a flag *timeCPUUsada* logo que o processo entra na CPU ou quando ele continua dentro da CPU. Como o tempo que ela está

ociosa é todo o tempo em que ela não está executando um processo, a flag *timeCPUOciosa* é incrementada toda vez que o tempo de execução é incrementado e a condição de *CPUOciosa* é verdadeira. Para o I/O, a flag *timeIO* é incrementada toda vez que um processo entra ou permanece no dispositivo I/O.

Já o tempo médio de espera entre processos foi calculado para os processos que saíram do sistema. Quando um processo entrava na fila Arrive, o tempo de simulação era salvo em *waitingTime_arrive_chegada* e quando ele saia da fila, esse mesmo tempo era salvo em *waitingTime_arrive_saida*. Assim, a variável *waitingTime_arrive* era incrementada com o valor de *waitingTime_arrive_saida* - *waitingTime_arrive_chegada*, ou seja, o tempo que o processo ficou na fila esperando.

Para a fila Ready a ideia é similar. Quando um processo entrava na fila Ready, o tempo de simulação era salvo em *waitingTime_ready_chegada* e quando ele saia da fila, esse mesmo tempo era salvo em *waitingTime_ready_saida*. Assim, a variável *waitingTime_ready* era incrementada com o valor de *waitingTime_ready_saida* - *waitingTime_ready_chegada*, ou seja, o tempo que o processo ficou na fila esperando nesta fila.

Por fim, para a fila de I/O, a ideia segue sendo a mesma. Quando um processo entrava na fila I/O, o tempo de simulação era salvo em *waitingTime_io_chegada* e quando ele saia da fila, esse mesmo tempo era salvo em *waitingTime_io_saida*. Assim, a variável *waitingTime_io* era incrementada com o valor de *waitingTime_io_saida* - *waitingTime_io_chegada*, ou seja, o tempo que o processo ficou na fila esperando.

Como o item é o tempo médio de espera do processo, para um processo **P,** o tempo de espera dele é a soma da espera nas três filas (*waitingTime_arrive* + *waitingTime_ready* + *waitingTime_io).* O valor obtido foi de 208.95.

O tempo médio de espera nas três filas foi calculado a partir da mesma ideia, porém levou-se em conta todos os processos criados, não apenas os que saíram do sistema. Assim, acrescentava-se o valor do cálculo feito anteriormente para cada fila (saída - chegada) em uma outra variável global, para que ao fim da execução cada fila tenha um valor total de espera de processos. Dessa forma, as variáveis *Time_arrive*, *Time_ready* e *Time_io* representam os valores de espera e as variáveis *ProcessoCriado*, *PReady* e *PIO* representam a quantidade de processos inseridos em cada fila. Somando -se os tempos e dividindo pelo número de processos inseridos tem -se o tempo médio de espera que, nesse caso, foi de 91.50 (Arrive), 2.63 (Ready) e 68.46 (I/O). A falta de exatidão nestes valores é devido ao fato de que nem todos os processos que são inseridos nas filas saem até o fim da execução, porém são contabilizados na quantidade de processos que entram na fila. Logo, o número do divisor pode ser maior do que o real, o que gera um resultado um pouco menor do que o esperado.

Ainda, é necessário o cálculo do tamanho médio de cada fila. Para tanto, existem funções de verificar o tamanho da fila em cada uma delas (*tamanho_arrive*, *tamanho_ready* e *tamanho_io*). Essas funções foram chamadas todas as vezes em que se inseriu um processo em uma das filas. O valor que retornasse como resultado foi salvo em uma variável global para cada fila (*TArrive*, *TReady* e *TIO*). Assim, ao fim da execução do laço `while`, foi dividido o tamanho da fila pelo número de processos que nela foram inseridos. Logo, obteve-se o valor do tamanho médio para cada uma delas: 12.9 (Arrive), 8.93 (Ready) e 4.38 (I/O).

Levando em consideração o Grau de Multiprogramação do sistema (15), percebe-se que mais processos podem entrar no sistema e competir pela CPU ou I/O.

O fato de a fatia de tempo ser 8 significa que os processos têm grande chance de acabar sua execução, a não ser que peçam I/O antes disso, pois nesse caso perdem a CPU e devem voltar para a fila Ready (somente após a execução de seu I/O) que já vai estar com muitos processos inseridos, uma vez que sua política é FIFO.

A fila de I/O também comporta vários processos, como pode ser percebido no decorrer da simulação. Sua política é a prioridade do processo, sendo que o processo que tiver o menor número possui a prioridade mais alta. Isso pode fazer com que um processo não saia imediatamente da fila, mesmo que tenha sido inserido nela a muito tempo.

Ademais, dos 50 processos criados, 20 foram terminados. Isso significa que os processos estão demorando para sair do sistema. Observa-se os processos **P0** ao **P3**:

O primeiro a sair é o processo **P1** com tempo de espera de 5 e turnaround 10. Isso quer dizer que era um processo rápido e não pediu I/O. O segundo processo a sair foi o **P3** com tempo de espera de 22 e turnaround de 30. Percebe-se que **P0** e **P2** ainda estão no sistema. O próximo a sair é o processo **P0**, com tempo de espera 35 e turnaround 75. Ou seja, o primeiro processo criado saiu do sistema após 88 unidades de tempo. Isso significa que ele foi um processo longo e/ou teve muitas solicitações I/O.

É notável também o tempo de CPU ociosa (51.69%). O nível de paralelismo nesse caso não está suprindo a boa execução dos processos, pois se essa ociosidade fosse menor, haveriam mais processos terminados do que apenas 20. Em termos de comparação *CPU-bound* e *IO-bound*, os valores são bem próximos (215 e 193, respectivamente). Logo, não é possível uma classificação de especificidade.

3.2 Grau de Multiprogramação 5, Fatia de Tempo 8, Tempo de Simulação 500:

Após a primeira simulação, manteve-se o mesmo Tempo de Simulação e a Fatia de Tempo, alterando-se apenas o grau de multiprogramação para 5. Percebe-se que o número de processos que saem do sistema tem um acréscimo de 6 processos, sendo que o novo valor da flag para processos concluídos foi 26.

Também ocorreu uma diminuição de no número de processos que solicitaram I/O e tiveram sua operação concluída. A flag *IO_completa,* que controla o número de processos que tiveram a operação concluída, é incrementada, como visto anteriormente e possui o valor de 23 processos.

Pelo incremento da flag *timeCPUUsada* e *timeCPUOciosa* obedecendo suas devidas condições como visto anteriormente, o tempo total de CPU para execução dos processos foi de 474, e destes, a CPU foi utilizada 226 ou 47.68%. O tempo que a CPU ficou ociosa, sem executar, foi de 248 ou 52.32%. Através da flag *timeIO,* obtém-se o tempo em que o dispositivo I/O foi utilizado, esse tempo foi de 182 ou 36.40%.

Para o tempo médio de espera entre processos foram utilizadas 9 flags (*waitingTime_arrive_chegada*, *waitingTime_arrive_saida*, *waitingTime_arrive*, *waitingTime_ready_chegada, waitingTime_ready_saida, waitingTime_ready, waitingTime_io_chegada, waitingTime_io_saida, waitingTime_io ,Time_arrive, Time_ready e Time_io*) e o número de processos completos. Cada uma das flags é específica de cada fila,

e são atualizadas toda vez que os processos entram e saem delas. O cálculo específico do tempo de espera de cada fila encontra-se na análise anterior, quando foi utilizado um ML de 15. A soma das três flags (*waitingTime_arrive* + *waitingTime_ready* + *waitingTime_io*) dividida pela flag *ProcessosCompletos,* que indica o número de processos que foram completos, representa o tempo de espera para médio entre processos. Utilizando um ML de 5, obteve-se um tempo de espera de 118.04.

Para o cálculo do tempo médio de espera nas três filas levou-se em conta todos os processos criados. Para isso foi preciso utilizar três variáveis(*ProcessoCriado, PReady* e *PIO*) para a quantidade de processos inseridos em cada fila e três variáveis (*Time_arrive, Time_ready* e *Time_io*) para os valores de espera de cada fila, para que no fim da execução cada fila tenha um valor total de espera de processos, como visto anteriormente. Para o cálculo do tempo de espera de cada fila utilizou-se o mesmo cálculo da análise anterior, obtendo-se um valor de espera de 59.77, 0.55 e 17.40 para as filas Arrive, Ready e I/O, respectivamente. Existe uma margem de erro nestes valores de espera de cada fila, visto que a contabilidade é feita quando processos entram e saem das filas, e quando a simulação acaba ainda existem processos nas três filas.

Para o cálculo do tamanho de cada fila, utilizou-se funções que foram implementadas para cada fila (*tamanho_arrive, tamanho_ready* e *tamanho_io*). Toda vez que um processo é inserido em uma das filas, ocorre a chamada da função e o valor do tamanho é salvo nas flags (*TArrive, TReady* e *TIO*) específicas de cada fila. O valor destas flags foi dividido pelo número de processos que entraram nelas(*ProcessoCriado, PReady* e *PIO*). O tamanho de cada uma das filas foi de 4.71 (Arrive), 3.22(Ready) e 1.64(I/O).

Analisando o Grau de Multiprogramação (5), percebe-se que a quantidade de processos no sistema diminui, comparado com a análise anterior, o que nos leva a conclusão de que há menos competição entre os recursos e os processos que estão no sistema tem mais chance de executar até o fim, aumentando o número de processos concluídos.

Considerando que o tempo de execução e o tempo de solicitação para o uso do dispositivo I/O são gerados randomicamente, o processo que tiver a fatia de tempo menor do que o tempo de solicitar I/O tem mais chance de executar, caso contrário ele volta para o final da fila Ready (ordem FIFO). Por estar com o Grau de Multiprogramação menor do que na análise anterior, este mesmo processo tem mais chance de voltar a executar a CPU e ser finalizado.

Percebe-se durante a execução, que a fila de I/O chega a ficar com 4 processos, o que indica que todos os processos do sistema solicitam I/O. A política da fila de I/O é inserção por prioridade. Ao ser criado o processo recebe uma prioridade (também gerada randomicamente), e através dela é alocado na fila de I/O, quanto menor o seu número de prioridade mais alta ela é. Isso pode fazer com que um processo fique esperando na fila indefinidamente caso tenha uma prioridade muito baixa (número alto).

Foram criados 34 processos, e 26 foram terminados, o que indica que a 76.47% dos processo são terminados. Observa-se os processos **P0** ao **P9**:

O primeiro processo a terminar é o **P1**, com tempo de espera de 5 e turnaround de 10. Ele Terminou rápido por possuir um tempo de execução menor do que a fatia de RR e menor do que o seu tempo de solicitar I/O. O segundo processo a sair é o **P3** com tempo de espera de 22 e turnaround de 27. **P0** e **P2** continuam no sistema, executando na CPU e no dispositivo I/O. O próximo a terminar é o processo **P0**, com tempo de espera de 15 e

turnaround de 51. O primeiro processo criado saiu do sistema com 51 unidades de tempo, pode-se notar que ele tinha um baixo tempo de execução e um baixo tempo de solicitar I/O, porém seu tempo de atendimento de I/O era longo. Os processos **P4, P5** e **P6** estão no sistema.

O próximo processo a terminar é o **P2** com tempo de espera de 32 e tempo de turnaround de 74. Pode-se perceber que seu tempo e execução é maior do que o seu tempo de solicitar I/O, e seu tempo de I/O é grande, pois durante grande parte de sua execução ele fica executando o dispositivo I/O. Após isso retorna para a fila de Ready e por possuir um ML baixo, logo volta a usar a CPU e é finalizado. Neste mesmo tempo **P7** já está no sistema.

É notável também o tempo de CPU ociosa (52.32%). O nível de paralelismo nesse caso não está suprindo a boa execução dos processos, pois se essa ociosidade fosse menor, haveriam mais processos terminados do que apenas 26. O processo teve uma semelhança no tempo de CPU usada e do dispositivo I/O usado, sendo 215 e 193 respectivamente.

4. Conclusão

A partir da análise de ambos os tipos de execução e os dados obtidos para cada um, conclui-se que o Grau de Multiprogramação menor tem melhor execução para um mesmo intervalo de tempo e mesma fatia de tempo na CPU. O número de processos terminados foi maior, isso porque existem menos processos no sistema para competirem pelos recursos e os que são inseridos geralmente acabam sendo finalizados.

Ademais, o tempo médio de espera entre os processos se reduziu, bem como as esperas nas filas. Isso é devido ao fato de existirem menos processos sendo criados (uma vez que a criação de processos considera o grau de multiprogramação como parâmetro) e, consequentemente, há menos “congestionamento” nas filas.

Ainda, a diminuição do tamanho das filas é uma consequência direta da redução do grau de multiprogramação, uma vez que menos processos podem entrar no sistema, causando uma redução de processos nas filas Ready e I/O,além da Arrive.