

RELATÓRIO DE ATIVIDADES  
TRABALHO 2 - SISTEMAS OPERACIONAIS  
Luan Tiago Streck e Enzo Santin da Silveira

1. Introdução.

O presente relatório tem como objetivo apresentar todos os detalhes acerca da implementação do Trabalho 2, que foca em adicionar ao Sistema Operacional o Gerenciamento de Processos e Escalonamento.

Na etapa anterior, o simulador operava em um modo monotarefa, executando um único programa por vez, com técnicas ineficientes de entrada/saída, sem bloqueio de processo e sem escalonamento.

Assim, no trabalho 2, com a orientação do professor, o objetivo foi transformar o ambiente em um sistema operacional multiprogramável, gerenciando a execução de múltiplos processos. As principais implementações tratadas neste trabalho incluem: Gerenciamento de Processos, Chamadas de Sistema, Eliminação de Espera Ocupada e Escalonamento de CPU.

No fim, é possível analisar as métricas obtidas durante a execução do programa na função de relatório interno. Com isso, é possível notar as diferenças na execução de acordo com as adições de funcionalidades.

## 2. Desenvolvimento prático.

### 2.1. Parte I.

Para começar a transformar o simulador em um SO multiprogramável, o primeiro passo foi implementar o Bloco de Controle de Processos e a Tabela de Processos. Antes, o estado da CPU era salvo em variáveis soltas na estrutura `so_t`. Agora, foi criada a estrutura `processo_t`, que possui todas as informações necessárias sobre um determinado programa. Com isso, outras variáveis também foram adicionadas, como o PID, que é um identificador do processo, uma variável que armazena o estado do processo e variáveis para E/S (entradas/saídas).

```
// a estrutura com as informações de um processo (Bloco do Controle de Processos)
typedef struct {
    int pid; // identificador do processo
    processo_estado_t estado; // estado atual do processo
    processo_bloqueio_t tipo_bloqueio; // motivo pelo qual está bloqueado

    // contexto da CPU (registradores salvos)
    int regA;
    int regX;
    int regPC;
    int regERRO;

    // informações de E/S
    dispositivo_id_t disp_entrada; // dispositivo de entrada do processo
    dispositivo_id_t disp_saida; // dispositivo de saída do processo
} processo_t;
```

Imagem 1- Estrutura `processo_t` criada.

Para gerenciar as múltiplas instâncias da estrutura `processo_t`, foi criada a Tabela de Processos, implementada com um vetor estático de tamanho fixo, definido no início do código (utilizamos `MAX_PROCESSOS=10`). O SO lida com essa tabela procurando espaços livres, analisando o estado dos processos ou acessando o índice para criar um novo processo ou manipular o atual.

As funções `so_salva_estado_da_cpu` e `so_despacha` tratam as interrupções e utilizam a Tabela de Processos implementadas, salvando os registradores em endereços específicos e copiando para esse Bloco de Controle de Processos (BCP) e pegando os valores do BCP e escrevendo de volta na memória, respectivamente.

```

static void so_salva_estado_da_cpu(so_t *self)
{
    if (self->processo_atual_idx == -1) { // nenhum processo estava executando
        return;
    }
    // obtém um ponteiro para o BCP do processo que estava executando
    processo_t *p = &self->tabela_processos[self->processo_atual_idx];
    // lê o estado da CPU que foi salvo na memória pela interrupção
    int pc, a, erro, x;
    if (mem_le(self->mem, CPU_END_PC, &pc) != ERR_OK ||
        mem_le(self->mem, CPU_END_A, &a) != ERR_OK ||
        mem_le(self->mem, CPU_END_erro, &erro) != ERR_OK ||
        mem_le(self->mem, 59, &x) != ERR_OK) { // X salvo pelo trata_int.asm
        console_printf("SO: erro na leitura dos registradores ao salvar contexto.");
        self->erro_interno = true;
        return;
    }
    // salva os valores no BCP do processo
    p->regPC = pc;
    p->regA = a;
    p->regERRO = erro;
    p->regX = x;
}

```

Imagem 2- Implementação da função que salva o estado da CPU.

```

static int so_despacha(so_t *self){
    if (self->processo_atual_idx == -1) {
        return 1; // Retorna 1 para o assembly, que fará a CPU parar (PARA)
    }
    processo_t *p = &self->tabela_processos[self->processo_atual_idx];
    // escreve o estado do processo na memória, de onde a CPU irá restaurá-lo
    if (mem_escreve(self->mem, CPU_END_PC, p->regPC) != ERR_OK ||
        mem_escreve(self->mem, CPU_END_A, p->regA) != ERR_OK ||
        mem_escreve(self->mem, CPU_END_erro, p->regERRO) != ERR_OK ||
        mem_escreve(self->mem, 59, p->regX) != ERR_OK) {
        console_printf("SO: erro na escrita dos registradores ao despachar processo.");
        self->erro_interno = true;
        return 1; // Para a CPU em caso de erro
    }
    // marca o processo como executando
    p->estado = EXECUTANDO;
    // se houver algum erro interno no SO, para a CPU
    if (self->erro_interno) {
        return 1;
    }
    return 0; // Retorna 0 para o assembly, que fará a CPU retornar da interrupção (RETI)
}

```

Imagem 3- Implementação da função de despachar processo do SO.

Também é necessário criar um processo para o init, que acontece na função `so_trata_reset`. O arquivo `init.maq` é carregado na memória e vai para o primeiro slot na tabela de processos. A estrutura do processo é preenchida, com o estado colocado como PRONTO.

As funções para criação e morte de processos também foram implementadas. Elas foram implementadas do zero, visto que a versão anterior não tinha suporte a isso. No `so_chamada_cria_proc`, o processo pai solicita a criação de um filho. O SO procura por um processo na tabela que esteja TERMINADO, caso não achar, retorna erro. Ele lê o nome do programa na memória do pai e aplica ao

filho, caso seja possível, e então carrega o programa do filho na memória. Após isso, atribui um PID e terminais de E/S. O `so_chamada_mata_proc` mata o processo atual ou outro, dependendo do PID que é passado. Assim, ele libera entrada na tabela para uso futuro, mudando o estado do processo morto para `TERMINADO`.

```
static void so_chamada_cria_proc(so_t *self){
    // o processo que está chamando a criação é o processo pai
    processo_t *pai = &self->tabela_processos[self->processo_atual_idx];
    //acha um slot livre na tabela de processos
    int novo_idx = -1;
    for (int i = 0; i < MAX_PROCESSOS; i++) {
        if (self->tabela_processos[i].estado == TERMINADO) {
            novo_idx = i;
            break;
        }
    }

    //cria o programa ....

    //preenche o bloco do novo processo
    processo_t *novo = &self->tabela_processos[novo_idx];
    novo->pid = self->proximo_pid++;
    novo->estado = PRONTO;
    novo->tipo_bloqueio = BLOQUEIO_NENHUM;
    novo->regPC = ender_carga;
    novo->regA = 0;
    novo->regX = 0;
    novo->regERRO = ERR_OK;
    novo->pid_esperado = -1;

    // ...continua o programa configurando terminais

    // retorna o PID do novo processo no registrador A do pai
    pai->regA = novo->pid;
    console_printf("SO: Processo '%s' criado com PID %d.", nome_prog, novo->pid);
}
```

Imagem 4- Implementação da função de criação de processos (trechos menos importantes foram omitidos para economizar espaço).

```
static void so_chamada_mata_proc(so_t *self){
    processo_t *chamador = &self->tabela_processos[self->processo_atual_idx];
    int pid_alvo = chamador->regX; // O PID q deve ser morto ta no registrador X

    //Muda o estado do processo para TERMINADO
    processo_t *alvo = &self->tabela_processos[idx_alvo];
    int pid_morto = alvo->pid; //guarda o PID antes de o invalidar
    alvo->estado = TERMINADO;
    alvo->pid = -1; //Libera o PID

    console_printf("SO: Processo com PID %d terminado.", pid_alvo);

    //Retorna 0 (sucesso) no registrador A do processo chamador
    chamador->regA = 0;
}
```

Imagem 5- Implementação da função de matar processos (trechos menos importantes foram omitidos para economizar espaço).

## 2.2. Parte II.

Na parte dois, foi necessário implementar um bloqueio de processos, visto que, na parte I, se um processo não está morto, ele pode executar. Assim, deve-se implementar o bloqueio de processos e eliminar a espera ocupada na E/S.

Para isso, foram implementadas as funcionalidades nas funções `so_chamada_le` e `so_chamada_escr`. Antes, eles consumiam a CPU enquanto esperavam, em um loop, por uma condição de dados do teclado. Agora, o SO consulta o estado do dispositivo, se estiver pronto, a atividade é realizada na hora. Se ele estiver indisponível, o processo altera o seu estado para BLOQUEADO, seu motivo é registrado, o índice do processo é definido como -1 e o escalonador deve escolher outro processo útil. Assim, o processo não mais entra em loop.

```
static void so_chamada_le(so_t *self){
    // [...] recupera o processo e verifica se o dispositivo de entrada ta pronto

    if (estado != 0) {
        // Dispositivo pronto, realiza a leitura imediatamente
        int dado;
        if (es_le(self->es, teclado, &dado) != ERR_OK) {
            console_printf("SO: problema no acesso ao teclado do processo %d", p->pid);
            self->erro_interno = true;
            p->regA = -1; // Retorna erro
            return;
        }
        p->regA = dado; // Coloca o dado lido no registrador A do processo
    } else {
        // Dispositivo não está pronto, bloqueia o processo
        console_printf("SO: Processo %d bloqueado esperando por entrada.", p->pid);
        p->estado = BLOQUEADO;
        p->tipo_bloqueio = BLOQUEIO_LE;
        // Força o escalonador a escolher outro processo
        self->processo_atual_idx = -1;
    }
}
```

Imagem 5- Implementação da função de bloqueio por leitura (trechos menos importantes foram omitidos para economizar espaço).

A implementação do `so_chamada_escr` é muito semelhante, só muda que o dado vem do `regX` e o tipo de bloqueio é `BLOQUEIO_ESCR`.

Os processos bloqueados devem ser desbloqueados se possível, assim, para isso a função `so_trata_pendencias` foi implementada. Ela é executada a cada interrupção, ela percorre a Tabela de Processos, procurando os processos bloqueados e o seu respectivo motivo. Se for bloqueio de leitura, ele vê se agora há dados, realiza a leitura, muda o estado do processo para PRONTO e insere o processo na fila de prontos. Se for bloqueio de escrita, ele verifica se a tela está livre, e se sim, ele escreve e desbloqueia.

```

static void so_trata_pendencias(so_t *self){
    for (int i = 0; i < MAX_PROCESSOS; i++) {
        processo_t *p = &self->tabela_processos[i];

        if (p->estado == BLOQUEADO) {
            // O processo está bloqueado, verifica o motivo
            if (p->tipo_bloqueio == BLOQUEIO_LE) {
                // [...] trecho que verifica o teclado
                if (estado_dev != 0) { //o dispositivo está pronto
                    p->tipo_bloqueio = BLOQUEIO_NENHUM;
                    console_printf("SO: Processo %d desbloqueado apos leitura.", p->pid);
                }
            } else if (p->tipo_bloqueio == BLOQUEIO_ESCR) {
                // [...] trecho que verifica a tela
                if (estado_dev != 0) { //o disp está pronto
                    p->regA = 0; // Retorna 0 (sucesso)
                    p->estado = PRONTO; // Desbloqueia o processo
                    p->tipo_bloqueio = BLOQUEIO_NENHUM;
                    console_printf("SO: Processo %d desbloqueado apos escrita.", p->pid);
                }
            }
        }
    }
}
}

```

Imagem 6- Implementação da função de tratamento de pendências, verificando o tipo de bloqueio (trechos menos importantes foram omitidos para economizar espaço).

Às vezes, os processos podem esperar por outros processos, além de hardware. Para isso, foi implementada a função `so_chamada_espera_proc`, ela coloca o processo chamador no estado de BLOQUEADO, com tipo BLOQUEIO\_ESPERA (diferente de leitura e escrita) e salva o PID do processo alvo em `pid_esperado`. Para ser desbloqueado, existem duas alternativas, uma é diretamente na função de tratar as pendências, que tem justamente esse objetivo, e a outra é na função de morte de processos. Na primeira, a função verifica se o `pid_esperado` existe, se não existir, ela desbloqueia. Na segunda, a função verifica se algum processo estava esperando pelo PID do processo que morreu, se sim, ele desbloqueia.

```

static void so_chamada_espera_proc(so_t *self) {
    //[...] trecho para validacoes
    chamador->estado = BLOQUEADO;
    chamador->tipo_bloqueio = BLOQUEIO_ESPERA;
    chamador->pid_esperado = pid_alvo;
    //força o escalonador a escolher outro processo
    self->processo_atual_idx = -1;
}

static void so_trata_pendencias(so_t *self) {
    //[...] verificacoes anteriores
    else if (p->tipo_bloqueio == BLOQUEIO_ESPERA) {
        // Verifica se o processo esperado ainda existe na tabela
        bool esperado_terminou = true;
        for (int j = 0; j < MAX_PROCESSOS; j++) {
            if (self->tabela_processos[j].pid == p->pid_esperado) {
                esperado_terminou = false; // Ainda existe
                break;
            }
        }
        if (esperado_terminou) {
            // Processo morreu, desbloqueia o chamador
            p->estado = PRONTO;
            p->tipo_bloqueio = BLOQUEIO_NENHUM;
            //[...]
        }
    }
}

```

Imagem 7- Implementação da função de bloqueio por espera e a adição dessa condição de bloqueio na função de tratamento de pendências (trechos menos importantes foram omitidos para economizar espaço).

### 2.3. Parte III.

Para a terceira e última parte do T2, foi necessário implementar dois escalonadores, um sendo o escalonador circular preemptivo round-robin e o outro um escalonador de prioridade. Eles irão estabelecer o critério de decisão para escolher quais processos ocuparão a CPU, considerando que haja vários processos no estado PRONTO.

Vale destacar que, dentro do código, é possível escolher o escalonador mudando manualmente o #define no início, na parte comentada que explica. Como os escalonadores não funcionam do mesmo modo, em alguns lugares do código existem #if e #endif, que são condicionais que verificam em tempo de compilação qual o escalonador a ser usado (isso acontece, por exemplo, para adicionar um processo na fila de prontos, se for o Round Robin).

As implementações dos escalonadores não foram exibidas com imagens no relatório pois elas se encontram espalhadas por todo o código, não em um trecho, como seria uma função. Por isso, apenas sua explicação de aplicação foi descrita e é possível entender o seu funcionamento a partir dela.

### 2.3.1. Escalonador Round Robin.

Esse é um algoritmo projetado para sistemas com tempo compartilhado, distribuindo o tempo de CPU de forma justa (por isso circular).

No código do nosso SO, foi utilizada uma Fila de Prontos, implementada como um vetor circular, onde novos e desbloqueados processos vão para o fim da fila e o escalonador sempre despacha o processo que está no início da fila. Para a implementação desse escalonador, foi necessário utilizar uma nova variável: o quantum. Ele basicamente é o tempo que um processo tem o direito de usar para executar. Se acabar o quantum de um processo, ele é colocado no final da fila, isso é a preempção.

No nosso código, cada vez que um processo é despachado, ele recebe uma quantia de tempo, nesse caso, um valor para o quantum. A cada interrupção de relógio, o contador é decrementado. Quando ele chegar a zero, acontece a preempção, o processo é forçado a sair da CPU, seus dados são salvos e ele vai para o final da fila de PRONTOS.

### 2.3.2. Escalonador de Prioridade.

O segundo escalonador é baseado em prioridades, onde processos com diferentes comportamentos são tratados de forma diferente. Nesse caso, os processos que usam pouco a CPU e bastante E/S são favorecidos. No código, a fórmula para o cálculo da variável de prioridade recalculada para quando o processo deixa a CPU foi dada pelo professor:  $prio = (prio + usocpu/quantum)/2$ . Nesse caso, quanto menor o valor da prioridade, maior é a prioridade do processo.

Para esse escalonador, a preempção pode ocorrer se um processo de maior prioridade ficar pronto, além da preempção por esgotar o quantum. Assim, o escalonador verifica a cada ciclo se existe alguém na fila com prioridade superior ao processo atual para executar.

### 2.3.3. Métricas.

Como orientado, o SO, ao final da execução, deve apresentar algumas métricas coletadas durante a execução dos programas sobre eles, para ser possível analisar o desempenho com diferentes configurações. Assim, diversas variáveis foram adicionadas na estrutura `processo_t`, que marcam número de vezes em determinado estado, tempo em determinado estado, entre outros.

As métricas são atualizadas em diferentes pontos durante a execução do processo, como registrando o tempo\_criacao e tempo\_termino em so\_chamada\_cria\_proc e so\_chamada\_mata\_proc, respectivamente, incrementando vezes\_bloqueado quando necessário, entre outros.

No final da execução, a função criada so\_gera\_relatório é chamada na main.c, antes do SO ser destruído, e então ela imprime no console, que fica salvo no log\_do\_console. Assim, é possível analisar as mudanças em desempenho para diferentes configurações do sistema.

```
typedef struct {
    // [...] PARTE 1 - T1

    // PARTE 3 - T2
    float prioridade;           // Prioridade do processo
    int tempo_inicio_execucao;  // tempo de início de execução do processo

    // -----METRICAS-----
    int tempo_criacao;
    int tempo_termino;
    int num_preempcoes;
    //entrada nos estados
    int vezes_pronto;
    int vezes_bloqueado;
    int vezes_executando;
    //cont de tempo nos estados
    long tempo_total_pronto;
    long tempo_total_bloqueado;
    long tempo_total_executando;
    int tempo_entrou_no_estado_atual;
    //tempo medio das respostas
    long soma_tempo_resposta;
    int n_respostas;
    int tempo_desbloqueio;
} processo_t;
```

Imagem 8- Adição das métricas e variáveis necessárias para a implementação da Parte III.

### 3. Análise de Resultados.

A análise de resultados vai ser interpretando os valores das métricas obtidas na função `so_gera_relatório`, que estão disponíveis no `log_do_console`.

Os testes foram realizados em duas configurações para cada escalonador, variando o valor de quantum.

#### 3.1. Escalonador Round Robin.

```
--- RELATORIO FINAL DO SISTEMA ---

[Metricas Globais]
- Tempo total de execucao: 72535 instrucoes
- Numero total de processos criados: 3
- Tempo em que a CPU ficou ociosa: 45750 instrucoes
- Numero total de preempcoes: 198
- Numero de interrupcoes por tipo:
  > IRQ 0 (Reset): 1 vezes
  > IRQ 2 (Chamada de sistema): 462 vezes
  > IRQ 3 (E/S: relógio): 1364 vezes

[Metricas por Processo]

>> Processo P1:
  - Tempo de retorno: 23722 instrucoes
  - Numero de preempcoes: 125
  - Entradas em PRONTO: 349, BLOQUEADO: 8, EXECUTANDO: 341
  - Tempo total em PRONTO: 12023, BLOQUEADO: 628, EXECUTANDO: 11071
  - Tempo medio de resposta: 1.88 instrucoes

>> Processo P2:
  - Tempo de retorno: 13185 instrucoes
  - Numero de preempcoes: 35
  - Entradas em PRONTO: 199, BLOQUEADO: 13, EXECUTANDO: 186
  - Tempo total em PRONTO: 7686, BLOQUEADO: 786, EXECUTANDO: 4713
  - Tempo medio de resposta: 94.77 instrucoes

>> Processo P3:
  - Tempo de retorno: 22795 instrucoes
  - Numero de preempcoes: 29
  - Entradas em PRONTO: 413, BLOQUEADO: 95, EXECUTANDO: 318
  - Tempo total em PRONTO: 10425, BLOQUEADO: 4993, EXECUTANDO: 7377
  - Tempo medio de resposta: 46.75 instrucoes
```

Imagem 9- Relatório de execução do Round Robin com quantum sendo 2.

```

--- RELATORIO FINAL DO SISTEMA ---

[Metricas Globais]
- Tempo total de execucao: 72551 instrucoes
- Numero total de processos criados: 3
- Tempo em que a CPU ficou ociosa: 46950 instrucoes
- Numero total de preempcoes: 70
- Numero de interrupcoes por tipo:
  > IRQ 0 (Reset): 1 vezes
  > IRQ 2 (Chamada de sistema): 462 vezes
  > IRQ 3 (E/S: relógio): 1364 vezes

[Metricas por Processo]

>> Processo P1:
  - Tempo de retorno: 21148 instrucoes
  - Numero de preempcoes: 45
  - Entradas em PRONTO: 322, BLOQUEADO: 6, EXECUTANDO: 316
  - Tempo total em PRONTO: 9848, BLOQUEADO: 429, EXECUTANDO: 10871
  - Tempo medio de resposta: 17.33 instrucoes

>> Processo P2:
  - Tempo de retorno: 15372 instrucoes
  - Numero de preempcoes: 15
  - Entradas em PRONTO: 200, BLOQUEADO: 13, EXECUTANDO: 187
  - Tempo total em PRONTO: 9854, BLOQUEADO: 797, EXECUTANDO: 4721
  - Tempo medio de resposta: 270.46 instrucoes

>> Processo P3:
  - Tempo de retorno: 26952 instrucoes
  - Numero de preempcoes: 7
  - Entradas em PRONTO: 416, BLOQUEADO: 97, EXECUTANDO: 319
  - Tempo total em PRONTO: 13385, BLOQUEADO: 6182, EXECUTANDO: 7385
  - Tempo medio de resposta: 93.74 instrucoes

--- FIM DO RELATORIO ---

```

Imagem 10- Relatório de execução do Round Robin com quantum sendo 5.

Com esses relatórios, é possível ver como o escalonador Round Robin se comporta quando o quantum é incrementado. Nesse caso, o incremento do quantum fez com que menos processos fossem expulsos da cpu, por isso, diminuiu o valor de preempções. Os valores de entrada em PRONTO estão sempre maiores que o valor de preempções, que está correto.

### 3.2. Escalonador de Prioridade.

```
--- RELATORIO FINAL DO SISTEMA ---

[Metricas Globais]
- Tempo total de execucao: 60323 instrucoes
- Numero total de processos criados: 3
- Tempo em que a CPU ficou ociosa: 34150 instrucoes
- Numero total de preempcoes: 217
- Numero de interrupcoes por tipo:
  > IRQ 0 (Reset): 1 vezes
  > IRQ 2 (Chamada de sistema): 462 vezes
  > IRQ 3 (E/S: relógio): 1135 vezes

[Metricas por Processo]

>> Processo P1:
  - Tempo de retorno: 23623 instrucoes
  - Numero de preempcoes: 149
  - Entradas em PRONTO: 353, BLOQUEADO: 7, EXECUTANDO: 346
  - Tempo total em PRONTO: 12008, BLOQUEADO: 504, EXECUTANDO: 11111
  - Tempo medio de resposta: 34.00 instrucoes

>> Processo P2:
  - Tempo de retorno: 22815 instrucoes
  - Numero de preempcoes: 40
  - Entradas em PRONTO: 212, BLOQUEADO: 20, EXECUTANDO: 192
  - Tempo total em PRONTO: 16693, BLOQUEADO: 1361, EXECUTANDO: 4761
  - Tempo medio de resposta: 11.15 instrucoes

>> Processo P3:
  - Tempo de retorno: 19781 instrucoes
  - Numero de preempcoes: 22
  - Entradas em PRONTO: 437, BLOQUEADO: 127, EXECUTANDO: 310
  - Tempo total em PRONTO: 6133, BLOQUEADO: 6335, EXECUTANDO: 7313
  - Tempo medio de resposta: 18.39 instrucoes

--- FIM DO RELATORIO ---
```

Imagem 11- Relatório de execução do Round Robin com quantum sendo 3.

```

--- RELATORIO FINAL DO SISTEMA ---

[Metricas Globais]
- Tempo total de execucao: 65399 instrucoes
- Numero total de processos criados: 3
- Tempo em que a CPU ficou ociosa: 38900 instrucoes
- Numero total de preempcoes: 175
- Numero de interrupcoes por tipo:
  > IRQ 0 (Reset): 1 vezes
  > IRQ 2 (Chamada de sistema): 462 vezes
  > IRQ 3 (E/S: relógio): 1230 vezes

[Metricas por Processo]

>> Processo P1:
  - Tempo de retorno: 23645 instrucoes
  - Numero de preempcoes: 128
  - Entradas em PRONTO: 346, BLOQUEADO: 8, EXECUTANDO: 338
  - Tempo total em PRONTO: 12051, BLOQUEADO: 547, EXECUTANDO: 11047
  - Tempo medio de resposta: 10.75 instrucoes

>> Processo P2:
  - Tempo de retorno: 16944 instrucoes
  - Numero de preempcoes: 29
  - Entradas em PRONTO: 221, BLOQUEADO: 17, EXECUTANDO: 204
  - Tempo total em PRONTO: 11036, BLOQUEADO: 1051, EXECUTANDO: 4857
  - Tempo medio de resposta: 28.47 instrucoes

>> Processo P3:
  - Tempo de retorno: 22347 instrucoes
  - Numero de preempcoes: 15
  - Entradas em PRONTO: 433, BLOQUEADO: 127, EXECUTANDO: 306
  - Tempo total em PRONTO: 8630, BLOQUEADO: 6436, EXECUTANDO: 7281
  - Tempo medio de resposta: 19.71 instrucoes

```

Imagem 12- Relatório de execução do Round Robin com quantum sendo 5.

Com esses relatórios, é possível ver como o escalonador de prioridade se comporta quando o quantum é incrementado. Nesse caso, o aumento do quantum fez com que houvesse, também, menos preempções. Também pode-se ver como muda o comportamento de certas variáveis com a implementação de um escalonador que decide a ordem de forma diferente.

#### 4. Conclusão.

O desenvolvimento deste trabalho permitiu evoluir de um simulador que operava em monotarefa para um Sistema Operacional multiprogramável, que gerencia seus processos de forma eficiente, por meio de uma lógica de controle de processos.

Uma das coisas mais importantes que foram feitas foi a implementação do mecanismo de bloqueio de processos, após a implementação da Tabela de Processos e do Bloco de Controle de Processos, que substituiu a espera ocupada. Com isso, foi possível obter um uso mais eficiente da CPU, suspendendo processos que aumentavam o desperdício por tempo ocioso, como acontecia na versão anterior.

Além disso, a implementação de dois escalonadores, Round Robin e de Prioridade, permitiram ver como diferentes lógicas de escolha afetam o comportamento do sistema. Os testes, analisando as métricas do sistema, mostraram que o Escalonador de Prioridade favorece os processos de E/S, dando maior importância a eles, logo, tempo de resposta menor. Enquanto isso, o Round Robin oferece uma distribuição de tempo mais justa, mas sem diferenciar urgência. A preempção foi fundamental para garantir a fluidez do sistema, forçando processos a saírem da CPU depois de estourarem um limite de tempo.

Assim, agora o SO consegue gerenciar a concorrência de processos utilizando diferentes escalonadores, a serem definidos de acordo com a preferência para a prática ou para o determinado teste. O sistema consegue reagir dinamicamente de acordo com a demanda de processos, seguindo uma lógica real e implementada.