UNIVERSIDADE FEDERAL
DO RIO DE JANEIRO

Trabalho 1 - SIMULAÇÃO DE ESCALONAMENTO DE PROCESSOS

Relatório - Grupo 3

Curso: Sistemas Operacionais
Período: 2020.1 (Remoto)
Professora: Valéria Menezes Bastos

Alunos - DRE:
Ângelo Augusto Costa Ribeiro Daumas - 115046541
José Vitor Hisse - 113065484
Renan Mendanha Alvarino - 118055604

Resumo: *Este relatório detalha os objetivos buscados e as premissas seguidas para o desenvolvimento de um simulador de gerenciamento de processos, além de documentar sua modelagem, implementação e saída no terminal.*

31/01/2021
DCC - UFRJ
Rio de Janeiro - RJ - Brasil

SUMÁRIO

1. INTRODUÇÃO

O objetivo deste trabalho é a simulação do gerenciamento de processos realizado em um sistema operacional. As principais metas são: a legibilidade do código, a separação de conceitos[1] e a fidelidade ao funcionamento real de computadores.

O trabalho é desenvolvido em C e implementa o algoritmo de seleção *Round Robin* com *Feedback*, como foi visto durante as aulas de Sistemas Operacionais. Todas as referências às variáveis e funções presentes no código do simulador estarão em **negrito e sublinhadas**. No Apêndice A há uma tabela contendo a relação dos principais elementos de código citados. As referências aos arquivos de código fonte estarão em *itálico* e possuem sufixos *.h* ou *.c*. A relação de arquivos do projeto pode ser encontrada no Apêndice B.

Neste relatório primeiramente será abordada a modelagem do programa e o processo pelo qual o grupo passou para realizá-lo. Depois, será abordado o código do programa e suas interfaces e funcionalidades. No quarto capítulo está documentada a saída do programa e como pode ser lida. Finalmente, antes das considerações finais, haverá uma explicação de como compilar e rodar o programa.

1.1. PREMISSAS

De acordo com as discussões internas do grupo, as premissas escolhidas foram as seguintes:

1. Limite de 20 processos ativos simultaneamente;
2. Fatia máxima de 3 unidades de tempo para cada processo;
3. Opção para gerar os processos de forma predeterminada (para motivos de teste) ou aleatória (com duração indeterminada de simulação);
4. Tempo de duração das operações nos dispositivos: 4 u.t. no disco, 7 u.t. na fita, 20 u.t. na impressora;
5. Um valor numérico menor na prioridade de um processo indica natureza mais prioritária.

Também foi decidido que esses valores deviam ser parametrizados em constantes no código fonte, de forma a documentá-los e tornar mais fácil sua alteração, seja para motivos de teste ou no caso do abandono dessas premissas por novas durante a elaboração do trabalho.

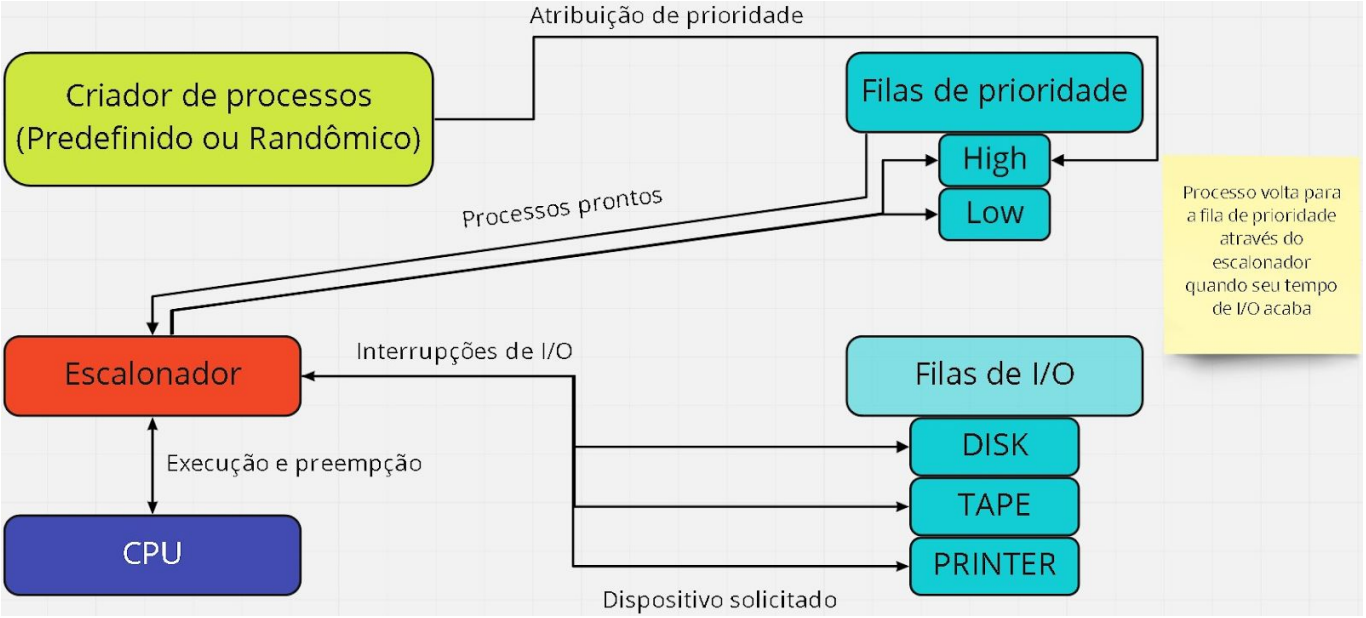
As durações das operações foram escolhidas de forma a ordená-las crescentemente espelhando a realidade. A fatia de tempo escolhida foi uma conciliação entre: valores muito grandes (que iriam se aproximar de uma estratégia FCFS) e valores muito pequenos (que não representam bem o fato de que programas realizam muitas instruções em cada time slice alocado).

[1] Princípio utilizado na computação que visa a preocupação com cada aspecto do código de forma modular.

2. MODELAGEM

Para guiar a construção do simulador, foi criado um diagrama que representa os seus elementos mais importantes e o fluxo de um processo entre cada um deles. Esse diagrama foi atualizado à medida que novos requisitos eram percebidos e sua versão final está ilustrada na Figura 1.

Figura 1 – Diagrama usado para o simulador



Como auxílio para a concepção do simulador, foi criado um protótipo escrito na linguagem *Python*, que está disposto no Apêndice D. A utilização de uma linguagem de alto nível permitiu ao grupo implementar o simulador sem preocupações como implantar as próprias estruturas de dados ou lidar com o gerenciamento de memória alocada. Foi na fase de prototipagem que a decisão de modularizar o programa foi feita, pois foram percebidos quais eram os componentes básicos necessários para o simulador e como eles poderiam ser delimitados e associados.

Para representar a passagem de tempo ou tick do processador, será utilizado um valor inteiro guardado na variável **CPUtime**. A passagem de uma unidade de tempo equivale a incrementar em 1 esse valor. Cada execução do laço principal do simulador implica a passagem de uma unidade de tempo. Quando um programa inicia uma operação de E/S[2], não há passagem de unidade de tempo (o tempo de processamento necessário na UCP[3] é desconsiderado).

Para representar um processo, foi criada a *struct* **Process**. Cada programa terá uma fila de instruções, que serão executadas em ordem. Cada instrução representa o conjunto de operações que um

[2] Entrada/Saída
[3] Unidade Central de Processamento

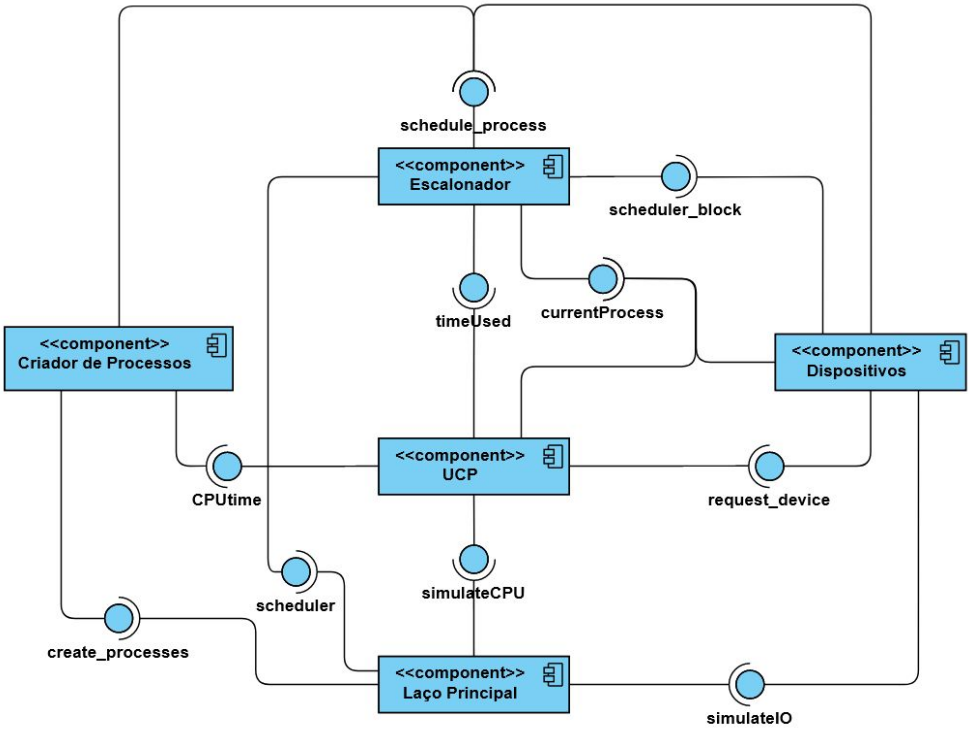
programa real estaria executando na UCP durante uma unidade de tempo. Há um total de 4 instruções: uma para cada dispositivo de E/S e a instrução **CPU**. As instruções de E/S (**DISK**, **TAPE**, **PRINTER**) indicam que o programa irá pedir acesso a um dispositivo, portanto será bloqueado enquanto espera o resultado da operação. A instrução **CPU** indica que o programa irá realizar apenas processamento na memória principal ou nos registradores.

Essa representação foi escolhida pois é fiel à realidade, na qual processos são instâncias de programas e portanto possuem um código fonte, que seria retratado na simulação como a fila de instruções. Por exemplo, um programa poderia solicitar processamento de 5 instruções de UCP, aguardar 3 ticks de tempo para uma leitura em disco e executar mais 3 instruções finais de processamento na UCP.

As filas de processos e também de instruções foram implementadas usando a *struct* **Queue**, declarada no arquivo *queue.h* e definida em *queue.c*. Para não ser necessária a utilização de múltiplas *structs* diferentes, as filas guardam números inteiros, que podem representar tanto o PID de um processo quanto uma instrução. Como a implementação interna das filas não é relevante para o funcionamento do simulador, ela não será abordada neste relatório.

Os diferentes componentes do programa irão se comunicar por meio de variáveis e funções globais definidas nas interfaces dos arquivos *.h*. As relações de uso e realização de interfaces foram documentadas usando um diagrama de componentes no padrão UML, disposto na Figura 2. A seção principal 3 a seguir irá detalhar essas interfaces e suas implementações.

Figura 2 – Diagrama de componentes do simulador



3. SIMULADOR

O simulador foi construído de forma modular, isto é, cada componente foi desenvolvido individualmente. Cada componente conhece apenas a API[4] pública dos outros componentes, cujo funcionamento interno é escondido dos outros arquivos (utilizando o modificador *static* do C). Isso tornou o desenvolvimento do trabalho mais flexível, pois cada componente pode ser refatorado ou substituído sem causar conflitos com o funcionamento dos outros componentes.

O simulador foi então dividido em 4 etapas que seriam implementadas individualmente:.

a) **Criação de processos:** simula a chegada de novos processos no sistema, seja pelo comando de um usuário ou por tarefas pré-programadas;
b) **Escalonador:** implementa a funcionalidade do escalonador: determinar o programa a ser executado no próximo ciclo da UCP;
c) **CPU:** simula o processamento da UCP;
d) **E/S:** simula o funcionamento dos dispositivos de E/S. Cada dispositivo é capaz de atender apenas 1 processo de cada vez.

Essas etapas são implementadas por funções, de forma que o funcionamento do simulador pode ser entendido observando apenas o laço principal do programa (ilustrado na Figura 3), enquanto os detalhes da implementação podem ser explorados nos arquivos que definem essas funções.

Figura 3 – Código-fonte do laço principal do simulador

```
while (True){
    create_processes();  // Passo 1: Simular a chegada de novos processos.

    scheduler();    // Passo 2: Simular a execução do escalonador.

    simulateCPU();  // Passo 3: Simular o processamento na CPU.

    simulateIO();   // Passo 4: Simular o processamento nos dispositivos de E/S.
}
```

As seções secundárias a seguir detalham os membros da API declarada por cada parte do simulador que são relevantes para o entendimento da sua funcionalidade. Funções relacionadas à geração de saída e à inicialização do programa estão omitidas. Como a simulação da UCP é implementada no arquivo principal e não apresenta nenhuma interface para ser usada em outros arquivos, a seção 3.3 se refere à implementação dessa fase do simulador, não à sua API.

[4] Interface de Programação da Aplicação

As seções terciárias documentam a implementação das interfaces citadas nas secundárias.

3.1. CRIAÇÃO DE PROCESSOS

A API para criação de processos é declarada no arquivo *creator.h* e é realizada pela função **create_processes**, que é a primeira função a ser chamada em cada iteração do laço principal do simulador. A outra função importante desse arquivo é **has_incoming_processes**, que é usada para determinar se o laço principal deve continuar. Ela pode ser de duas formas: pré-definida (implementada no arquivo *creator.c)* ou aleatória (implementada no arquivo *creatorrandom.c*).

3.1.1 Implementação predeterminada

O algoritmo para essa implementação é simples: os processos são guardados no array **future_processes** e suas instruções são definidas em **initalize_processes**. Os processos devem estar no array em ordem crescente de instante de criação. A função **create_processes** executa um laço enquanto o número de processos criados até o momento for menor que o tamanho de **future_processes**. Quando um processo com tempo de início igual ao **CPUtime** é encontrado, esse processo é criado e escalonado. Quando um processo com tempo maior que o **CPUtime** é encontrado, a função retorna o controle para quem a chamou. A relação de processos criados é ilustrada no Quadro 1.

Quadro 1 – Processos predeterminados

Processo	Tempo de Início	Instruções
P1	0	3xCPU - 1xDISK - 10xCPU
P2	5	5xCPU
P3	5	1xCPU - 1xTAPE - 4xCPU

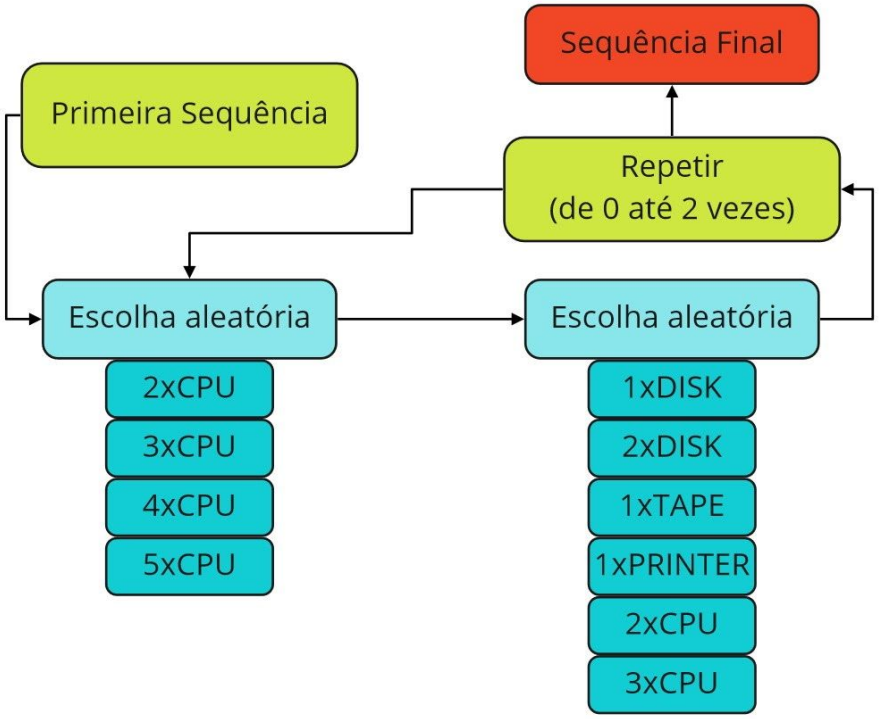
3.1.2 Implementação aleatória

Este algoritmo já é mais sofisticado: os processos são gerados dinamicamente a cada chamada de **create_processes**. Cada chamada pode criar zero, um, ou dois processos, com distribuição uniforme na escolha desse número. As instruções de cada processo são uma série de sequências geradas por um mesmo algoritmo. A cada processo são adicionadas de 1 a 3 sequências de instruções. O último passo é

garantir que a última instrução do processo seja **CPU**. Isso é feito pois nenhum dos exemplos vistos em aula tinham pedidos de E/S no último instante de vida do processo.

A geração das sequências de instruções possui dois passos. Primeiro, são adicionadas de 2 a 5 instruções **CPU**. Depois é adicionada uma das opções seguintes: 1 **DISK**, 2 **DISK**, 1 **TAPE**, 1 **PRINTER**, 2 **CPU**, 3 **CPU**. Esse algoritmo foi desenvolvido com o intuito de garantir uma presença maior de instruções **CPU** nos processos e para possibilitar a criação de processos sem nenhuma instrução de E/S, mas mantendo que a maior parte dos processos tenha uma instrução desse tipo. A Figura 4 ilustra uma representação gráfica desse algoritmo.

Figura 4 – Algoritmo para criação randômica de sequências de instruções



A variável **MAX_INSTRUCTIONS**, cujo valor padrão é 20, indica qual o tamanho máximo das filas de instruções dos processos criados. Se a sequência final gerada for maior que esse valor, as instruções adicionadas após a fila ficar cheia são ignoradas, de acordo com a interface da *struct* **Queue**.

3.2. ESCALONADOR

O escalonador é implementado nos arquivos *scheduler.c* e *scheduler.h*. O arquivo *.h* não especifica o tipo de escalonador usado, apenas define as operações que o escalonador deve implementar, como o bloqueio e escalonamento de processos. Também define o número de prioridades existentes. A implementação e funcionamento interno do escalonador, inclusive o algoritmo e estratégia usados, são determinados pelo arquivo *.c*. Assim seria possível criar arquivos com estratégias distintas que seriam

compatíveis com o mesmo *scheduler.h.* Os principais membros da API exposta pelo escalonador podem ser encontrados no Quadro 2.

Quadro 2 – Principais membros da API do escalonador

Membro	Descrição
currentProcess	um ponteiro para **Process** que indica qual processo deve ser executado pela UCP.
timeUsed	uma variável inteira que deve ser incrementada pela CPU a cada tick, representa o tempo que UCP gastou com o processo atual ou sem executar nenhum processos.
scheduler_block()	função usada para bloquear o **currentProcess** atual e selecionar automaticamente um novo.
schedule_process(Process* p)	marca um processo como pronto e o coloca na fila de processos prontos referente à sua prioridade. Porém, se ele não tiver mais instruções, o processo é terminado.
scheduler()	função que implementa a etapa do escalonador no laço principal do simulador.

O escalonador, portanto, também tem o papel de marcar processos como terminados e bloqueados.

3.2.1 Implementação Round Robin com Feedback

Como especificado na seção 1, a estratégia utilizada é de *Round Robin* com *Feedback*. Isto significa que haverá a preempção de processos baseada em fatias de tempo e também serão atribuídas prioridades a cada processo, que serão levadas em conta na escolha do próximo processo a ser executado na UCP.

Para a implementação desse algoritmo, a solução escolhida foi criar uma fila de processos prontos distinta para cada nível de prioridade. Dentro dessas filas, a seleção de processos será feita utilizando o *Round Robin*.

3.3. UNIDADE CENTRAL DE PROCESSAMENTO (UCP)

A simulação da unidade central de processamento é realizada pela função **simulateCPU**, que é definida no arquivo *main.c* e chamada dentro do laço principal do simulador. O papel da UCP consiste em ler e lidar com as instruções dos processos, além de realizar a incrementação da variável **CPUtime**.

Quando é encontrada uma instrução de E/S, é enviado um pedido para o referido dispositivo. Isto leva ao bloqueio do processo e à definição de um novo **currentProcess** pelo escalonador. Como explicado na seção 2, o tempo de processamento das instruções que pedem uma operação de E/S é desconsiderado. Assim, **CPUtime** só será incrementado quando for encontrado um processo cuja próxima

instrução seja **CPU**; ou quando não houver mais processos prontos para executar. Esse comportamento é implementado usando o laço descrito na Figura 5.

Figura 5 – Laço que processa instruções de E/S

```
int instruction;
while (currentProcess && (instruction = queue_pop(currentProcess->instructions))
!= CPU){
    request_device(instruction);
}
```

3.4. DISPOSITIVOS DE E/S

A simulação dos dispositivos de E/S é implementada nos arquivos *devices.c* e *devices.h*. Essa etapa consiste em simular a passagem de tempo para os dispositivos, durante o qual eles estariam processando os pedidos enviados a eles. As principais funções da API são: **request_device**, que deve ser chamada pela UCP quando um processo requisita um dispositivo, e **simulateIO**, chamada no laço principal.

3.4.1 Implementação dos dispositivos

Cada tipo de operação de E/S tem uma duração específica, que é rastreada usando números inteiros. O funcionamento do dispositivo em si não é simulado ao nível da UCP, na qual as instruções do processo são executadas sequencialmente: a operação do dispositivo é representada apenas pela passagem de tempo.

Cada dispositivo possui uma variável contadora que rastreia a duração remanescente da operação atual. Todo tick, se houver processos na fila de espera do dispositivos, essa variável é diminuída. Quando essa variável atinge zero, a operação é considerada completa e o primeiro processo da fila de espera é removido. Esse comportamento é implementado pela função **simulate_device**, que retorna o PID do processo removido da fila quando uma operação acaba (ou zero se não houve fim de operação).

A ação de função **simulateIO** então é simples: ele irá iterar por todos os dispositivos e chamar a função **simulate_device**. Se um PID positivo for retornado, significa que um processo terminou o seu pedido de E/S, logo esse processo pode ser retornado para a fila de processos prontos por meio da função **schedule_process** do escalonador.

4. SAÍDA DO PROGRAMA

A saída do programa contém 3 aspectos: *eventos de processos*, *estado da UCP* e *composição das filas*. Estes dois últimos são impressos em todo tick, enquanto os eventos são impressos apenas quando ocorrem. Na Figura 6 está exposto um tick teórico no qual todos os eventos possíveis ocorreram.

Figura 6 – Exemplo de saída para um tick com a ocorrência de todos os eventos

```
 1 [Tick 54]  DISK: FINISHED (pid 3)
 2 [Tick 54]  TAPE: FINISHED (pid 4)
 3 [Tick 54]  PRINTER: FINISHED (pid 5)
 4 [Tick 54]  New process: pid 1
 5 (CCCDCCCTCCCCPC)
 6 [Tick 54]  SCHEDULER: TERMINATE (pid 2)
 7 [Tick 54]  SCHEDULER: PREEMPT (pid 1)
 8 [Tick 54]  CPU: REQUEST DISK (pid 16)
 9 [Tick 54]  CPU: RUNNING (pid 15) - t=1
10 [Tick 54]  Ready Queue (HIGH): { 7 8 }
11 [Tick 54]  Ready Queue (LOW): { 9 10 }
12 [Tick 54]  I/O Queue (DISK): { 11 }
13 [Tick 54]  I/O Queue (TAPE): { 12 13 }
   [Tick 54]  I/O Queue (PRINTER): { 14 }
```

Os números em vermelho na Figura 6 representam a identificação da linha e não fazem parte da saída do programa. Cada linha da saída é prefixada pelo tick atual, ou seja, o tempo total de execução da UCP naquele momento. Este dado é encontrado entre colchetes e separado da informação específica da linha por um caractere de tabulação.

As subseções a seguir explicam a formatação para cada tipo de linha de saída gerada. No Apêndice C pode ser encontrada em íntegra a saída do programa para o conjunto exemplo de processos, cujas instruções e tempo de entrada podem ser encontrados no Quadro 1 (seção 3.1.1).

4.1. EVENTOS DE PROCESSOS

A seguir serão listados os possíveis eventos que podem ser imprimidos na saída (entre parêntesis, os números das linhas da Figura 6 onde esses eventos aparecem): *operação de E/S terminada* (1, 2, 3); *programa terminado* (5); *programa preemptivo* (6); *requisição de dispositivo de E/S* (7); *criação de novo processo* (4). Nesse último caso, o ID do processo é indicado depois dos dois pontos e as instruções desse novo processo são exibidas entre parêntesis. Nos outros casos, o ID do processo é imprimido entre parêntesis.

As instruções de cada processo recém-criado são formatadas como uma sequência de letras, sem nenhum separador entre elas, na qual cada letra representa uma instrução. As equivalências entre cada letra e sua instrução podem ser encontradas no Quadro 3.

Quadro 3 – Letras usadas na exibição de instruções.

Instrução	Letra
CPU	C
DISK	D
TAPE	T
PRINTER	P

Caso seja conveniente obter a saída do programa sem incluir os eventos de processos, é possível alterar o macro **SHOW_EVENTS** (definido em *main.c*) para zero. Se isso for feito, apenas os eventos de criação de um novo processo serão impressos na tela.

4.2. ESTADO DA UCP

A UCP possui dois estados: *RUNNING* e *IDLE*. O primeiro ocorre quando **currentProcess** não é nulo e o ID desse processo é indicado entre parênteses na saída. O segundo ocorre quando **currentProcess** é nulo, ou seja, não há processo sendo executado na UCP. Em ambos os casos, o tempo de duração do estado atual é impresso no fim da linha, após um hífen. Para o estado *RUNNING*, esse tempo é zerado toda vez que um novo processo é escolhido pelo escalonador.

Na Figura 6, a linha 8 é referente ao estado da UCP e indica que ela está ocupada. Um exemplo de saída para uma UCP ociosa há 7 unidades de tempo é dado na Figura 7.

Figura 7 – Exemplo de saída para um tick com UCP ociosa

```
[Tick 101]  CPU: IDLE - t=7
```

4.3. COMPOSIÇÃO DAS FILAS DE PROCESSOS

Como últimas linhas de cada tick, são impressas as filas de processos, tanto as de processos prontos quanto as de processos bloqueados esperando uma operação de E/S. Os PIDs dos processos aparecem entre chaves na ordem em que entraram na fila, separados por um espaço em branco. A designação de cada fila (nível de prioridade ou nome do dispositivo) é reproduzida entre parêntesis.

O comportamento padrão é imprimir todas as filas incondicionalmente. Entretanto, é possível alterar o valor do macro **SHOW_EMPTY_QUEUES** (definido em *main.c*) para zero. Se isso for feito, apenas as filas que não estiverem vazias serão impressas.

4.4. TABELA DE PROCESSOS

Para demonstrar a corretude dos estados assumidos por cada processo e também para proporcionar uma visão alternativa da simulação, é dada ao usuário a opção de imprimir a tabela de processos. Isso ocorre apenas quando o programa é executado no modo interativo (mais informações na seção 5).

As informações impressas em cada tabela são, em ordem de aparecimento: PID do processos, tempo de início dos processos, estado dos processos, instruções remanescentes dos processos. Processos com estado *terminado* não possuem mais instruções. Processos com estado *criado* não aparecem na tabela, pois são escalonados pelo escalonador e logo assumem o estado de *pronto*. A Figura 7 ilustra um exemplo de tabela onde os processos assumiram todos os estados possíveis. A formatação das instruções segue o mesmo padrão descrito na seção 4.1.

Figura 7 – Exemplo de uma tabela de processos

```
ID   START    STATE  INSTRUCTIONS
 1       0    Ready  CCDDCCCCCCCCCCCCC
 2       1    Ready  CCPCCCCTC
 3       2    Block  CCPCCCCTC
 4       2      Run  DCCCCCC
 5       3    Ready  CCCCCCCPCCCC
 6       3    Termi
 7       4    Ready  CCCCCC
 8       6    Ready  CCCCCDCCCTC
 9       6    Ready  CCCTCCCCCDDC
10       7    Ready  CCCCC
11       7    Ready  CCCCTC
12       8    Ready  CCCCCDDCCCCC
```

5. EXECUTANDO O PROGRAMA

O programa pode ser compilado em duas formas, referentes à forma de criação dos processos: predeterminada ou aleatória. Na primeira, é utilizado um exemplo conjunto de programas descrito na seção 3.1.1. Na segunda, a criação de processos não tem fim determinado: processos são continuamente gerados utilizando conjuntos de instruções aleatórios. Esta é a opção padrão e também torna o programa interativo, permitindo ao usuário decidir em que momento o próximo tick deve ser executado.

Os processos gerados aleatoriamente seguem uma estratégia de distribuição de instruções que busca inserir instruções de E/S entre um número maior de instruções **CPU**. Essa estratégia foi descrita na seção 3.1.2.

Para compilar o programa gerando processos aleatórios, é usado o comando *make* no terminal. Para obter processos predeterminados, é usado o comando *make example*. No Quadro 4 é feita uma comparação entre ambos os comandos.

Quadro 4 – Comparação entre os comandos para compilação do programa

Comando	Executável gerado	Modo de Execução	Criação de Processos
make	*simulador*	interativo	aleatória
make example	*example*	automático	predeterminada

No modo interativo, é necessário que o usuário aperte a tecla *ENTER* para prosseguir com a simulação. Dependendo da configuração do sistema do usuário, é possível segurar a tecla *ENTER* na posição abaixada para avançar rapidamente. Esse modo também oferece ao usuário alguns comandos, que estão listados no Quadro 5. Para realizar um comando, basta digitar o caractere indicado. Na maior parte dos terminais, é por padrão necessário teclar *ENTER* para que o programa possa ler os caracteres. Nesse caso, o próximo tick será também gerado, exceto se o comando dado foi de encerrar o programa.

Quadro 5 – Comandos do modo interativo do programa

Caracteres	Efeito causado
t ou T	Imprimir a tabela de processos. Um exemplo é dado na seção 4.4 deste relatório.
p ou P	Encerrar a execução do programa.

6. CONSIDERAÇÕES FINAIS

A realização deste trabalho foi de grande valor acadêmico para os integrantes do grupo, pois a simulação dos conceitos aprendidos na sala de aula é uma ótima maneira de se obter uma perspectiva alternativa desses conceitos e ajudar no seu entendimento.

Destaca-se também como o trabalho foi uma oportunidade para a exerção de conhecimentos multidisciplinares:

a) programação em linguagens de baixo nível (no uso de C);
b) fundamentos da engenharia de software (na modelagem, modularização e prototipagem);
c) arquitetura de computadores (na simulação de diferentes partes do sistema como a UCP e os dispositivos de E/S);
d) formatação e escrita de trabalhos técnicos (no relatório);
e) *soft skills* (para se comunicar e se organizar no trabalho em grupo e na gravação do vídeo).

Quanto às metas que foram colocadas no início do trabalho, pode-se considerar que foram atingidas de modo satisfatório:

a) **A legibilidade do código** é auxiliada por comentários explicativos para esclarecer a função de cada elemento, nomes adequados para variáveis e número de linhas aceitável para cada arquivo;
b) **A separação de conceitos** foi bem-sucedida. Isso pôde ser demonstrado pela dupla implementação do criador de processos: a modularização do código permitiu implementações em arquivos *.c* distintos, mantendo o mesmo arquivo *.h*. Ou seja, componentes podem ser substituídos sem afetar o funcionamento do resto do programa;
c) **A fidelidade** da simulação é enriquecida pelo uso de uma lista de instruções para os processos, que foi uma forma interessante de remeter ao código fonte de programas reais.

Por fim, o gerenciamento de processos é um dos aspectos elementares dos sistemas operacionais e o grande responsável pelo surgimento de sistemas multitarefa, sem os quais a utilidade dos computadores modernos seria extremamente reduzida. Fica então clara a importância desse conceito e os méritos de estudá-lo em qualquer curso de Bacharelado em Ciência da Computação.

7. BIBLIOGRAFIA

Para a realização deste trabalho foram referenciados os materiais de apoio disponíveis na página Classroom do curso de Sistemas Operacionais no período de 2020/1. Mais especificamente, foram utilizados os slides da Aula 3 (Processos) e Aula 4 (Escalonamento de Processos).

A composição das filas de instruções dos processos seguiu os moldes dos exemplos vistos na Lista 2 disponibilizada no curso, nos quais os processos possuíam poucas instruções de E/S intercaladas com vasto uso de tempo da UCP.

APÊNDICE A – PRINCIPAIS ELEMENTOS DE CÓDIGO CITADOS NO TEXTO

Nome	Descrição
Process	Estrutura que guarda as informações de um processo específico, incluindo sua fila de instruções, identificador de processo, prioridade e estado.
Queue	Estrutura de fila a qual podemos executar operações para guardar, remover e obter dados da mesma.
currentProcess	Processo que será ou foi executado pela CPU.
timeUsed	Guarda o tempo consumido pelo processo que está sendo executado.
scheduler_block	Bloqueia o processo em execução e indica o próximo processo da fila de prontos a ser executado.
schedule_process	Posiciona um processo na fila de prontos referente à sua prioridade. Se o processo não tiver instruções, é terminado.
scheduler	Fase 2 no laço principal do simulador. Lida com a organização dos processos em filas de pronto e de I/O, indicando o próximo processo a ser executado caso o processo anterior tenha sofrido preempção ou terminado.
CPUtime	Tempo decorrido na CPU desde a inicialização do simulador.
simulateCPU	Fase 3 no laço principal do simulador. Simula a execução de processos na CPU, requisitando operações de E/S ou aumentando o tempo usado pelo processo em um ciclo RR.
create_processes	Função chamada no laço principal, responsável por criar novos processos.
has_incoming_processes	Retorna true se houver o criador de processos ainda estiver ativo.
request_device	Função chamada pela CPU para requisitar E/S para o **currentProcess**.
simulateIO	Função chamada no laço principal, responsável por simular a passagem de tempo nos dispositivos de E/S.
CPU	Instrução que simula operação apenas nos registradores ou memória.
DISK	Instrução para requisitar o dispositivo I/O de disco presente no sistema. Também representa o ID desse dispositivo para funções de *device.h*.
TAPE	Instrução para requisitar o dispositivo I/O de fita presente no sistema. Também representa o ID desse dispositivo para funções de *device.h*.
PRINTER	Instrução para requisitar o dispositivo I/O de impressão presente no sistema. Também representa o ID desse dispositivo para funções de *device.h*.

APÊNDICE B – RELAÇÃO DOS ARQUIVOS DE CÓDIGO FONTE

Arquivos Auxiliares

process.h	Arquivo que define a estrutura e ações sobre um processo, incluindo também a tabela de processos e a sua contagem, além de suas possíveis instruções e estados.
process.c	Apresenta a implementação de process.h.
queue.h	Arquivo que contém a definição da API que será utilizada para filas no projeto.
queue.c	Apresenta a implementação de queue.h.
.replit	Indica a linguagem sendo usada e a chamada para o makefile do aplicativo no ambiente de execução remoto.

Arquivos do Simulador

creator.h	Arquivo que contém a definição das funções usadas para criar os processos.
creator.c	Apresenta a implementação de creator.h para um número de processos finitos e predeterminados, facilitando a análise do resultado gerado.
creatorrandom.c	Contém a implementação de creator.h, de forma que a criação dos processos é feita de forma aleatória e repondo processos que foram terminados. Nesta implementação, é possível definir um tempo finito ou infinito pelo qual a CPU irá executar processos.
devices.h	Arquivo que contém a definição das funções e atributos na simulação de dispositivos de entrada e saída do sistema.
devices.c	Apresenta a implementação de devices.h.
scheduler.h	Apresenta a definição de todas as funções e informações do escalonador de processos, incluindo o Quantum de um ciclo RR e filas de prioridade.
scheduler.c	Apresenta a implementação de scheduler.h.
main.c	Arquivo responsável por inicializar o sistema e simular o seu loop de funcionamento, também inclui informações cruciais para o funcionamento do programa, como o tempo total de execução da CPU. Além disso, também está presente neste arquivo a implementação de funções para fornecer saídas no console com as informações relevantes acerca do estado do sistema.
output.h	Define as funções de output para o console que são implementadas em main.c.

APÊNDICE C – SAÍDA DO CONJUNTO DE PROGRAMAS EXEMPLO

```
[Tick 0]    New process: pid 1 (CCCDCCCCCCCCCC)
[Tick 0]    CPU: RUNNING (pid 1) - t=1
[Tick 0]    Ready Queue (HIGH): { }
[Tick 0]    Ready Queue (LOW): { }
[Tick 0]    I/O Queue (DISK): { }
[Tick 0]    I/O Queue (TAPE): { }
[Tick 0]    I/O Queue (PRINTER): { }

[Tick 1]    CPU: RUNNING (pid 1) - t=2
[Tick 1]    Ready Queue (HIGH): { }
[Tick 1]    Ready Queue (LOW): { }
[Tick 1]    I/O Queue (DISK): { }
[Tick 1]    I/O Queue (TAPE): { }
[Tick 1]    I/O Queue (PRINTER): { }

[Tick 2]    CPU: RUNNING (pid 1) - t=3
[Tick 2]    Ready Queue (HIGH): { }
[Tick 2]    Ready Queue (LOW): { }
[Tick 2]    I/O Queue (DISK): { }
[Tick 2]    I/O Queue (TAPE): { }
[Tick 2]    I/O Queue (PRINTER): { }

[Tick 3]    SCHEDULER: PREEMPT (pid 1)
[Tick 3]    CPU: REQUEST DISK (pid 1)
[Tick 3]    CPU: IDLE - t=1
[Tick 3]    Ready Queue (HIGH): { }
[Tick 3]    Ready Queue (LOW): { }
[Tick 3]    I/O Queue (DISK): { 1 }
[Tick 3]    I/O Queue (TAPE): { }
[Tick 3]    I/O Queue (PRINTER): { }

[Tick 4]    CPU: IDLE - t=1
[Tick 4]    Ready Queue (HIGH): { }
[Tick 4]    Ready Queue (LOW): { }
[Tick 4]    I/O Queue (DISK): { 1 }
[Tick 4]    I/O Queue (TAPE): { }
[Tick 4]    I/O Queue (PRINTER): { }

[Tick 5]    New process: pid 2 (CCCCC)
[Tick 5]    New process: pid 3 (CTCCCC)
[Tick 5]    CPU: RUNNING (pid 2) - t=1
[Tick 5]    Ready Queue (HIGH): { 3 }
[Tick 5]    Ready Queue (LOW): { }
[Tick 5]    I/O Queue (DISK): { 1 }
[Tick 5]    I/O Queue (TAPE): { }
[Tick 5]    I/O Queue (PRINTER): { }

[Tick 6]    CPU: RUNNING (pid 2) - t=2
[Tick 6]    Ready Queue (HIGH): { 3 }
[Tick 6]    Ready Queue (LOW): { }
[Tick 6]    I/O Queue (DISK): { 1 }
[Tick 6]    I/O Queue (TAPE): { }
[Tick 6]    I/O Queue (PRINTER): { }

[Tick 7]    DISK: FINISHED (pid 1)
[Tick 7]    CPU: RUNNING (pid 2) - t=3
[Tick 7]    Ready Queue (HIGH): { 3 }
[Tick 7]    Ready Queue (LOW): { 1 }
[Tick 7]    I/O Queue (DISK): { }
[Tick 7]    I/O Queue (TAPE): { }
[Tick 7]    I/O Queue (PRINTER): { }

[Tick 8]    SCHEDULER: PREEMPT (pid 2)
[Tick 8]    CPU: RUNNING (pid 3) - t=1
[Tick 8]    Ready Queue (HIGH): { }
[Tick 8]    Ready Queue (LOW): { 1 2 }
[Tick 8]    I/O Queue (DISK): { }
[Tick 8]    I/O Queue (TAPE): { }
[Tick 8]    I/O Queue (PRINTER): { }

[Tick 9]    CPU: REQUEST TAPE (pid 3)
[Tick 9]    CPU: RUNNING (pid 1) - t=1
[Tick 9]    Ready Queue (HIGH): { }
[Tick 9]    Ready Queue (LOW): { 2 }
[Tick 9]    I/O Queue (DISK): { }
[Tick 9]    I/O Queue (TAPE): { 3 }
[Tick 9]    I/O Queue (PRINTER): { }

[Tick 10]   CPU: RUNNING (pid 1) - t=2
[Tick 10]   Ready Queue (HIGH): { }
[Tick 10]   Ready Queue (LOW): { 2 }
[Tick 10]   I/O Queue (DISK): { }
[Tick 10]   I/O Queue (TAPE): { 3 }
[Tick 10]   I/O Queue (PRINTER): { }

[Tick 11]   CPU: RUNNING (pid 1) - t=3
[Tick 11]   Ready Queue (HIGH): { }
[Tick 11]   Ready Queue (LOW): { 2 }
[Tick 11]   I/O Queue (DISK): { }
[Tick 11]   I/O Queue (TAPE): { 3 }
[Tick 11]   I/O Queue (PRINTER): { }

[Tick 12]   SCHEDULER: PREEMPT (pid 1)
[Tick 12]   CPU: RUNNING (pid 2) - t=1
[Tick 12]   Ready Queue (HIGH): { }
[Tick 12]   Ready Queue (LOW): { 1 }
[Tick 12]   I/O Queue (DISK): { }
[Tick 12]   I/O Queue (TAPE): { 3 }
```

```
[Tick 12]    I/O Queue (PRINTER): { }

[Tick 13]    CPU: RUNNING (pid 2) - t=2
[Tick 13]    Ready Queue (HIGH): { }
[Tick 13]    Ready Queue (LOW): { 1 }
[Tick 13]    I/O Queue (DISK): { }
[Tick 13]    I/O Queue (TAPE): { 3 }
[Tick 13]    I/O Queue (PRINTER): { }

[Tick 14]    SCHEDULER: TERMINATE (pid 2)
[Tick 14]    CPU: RUNNING (pid 1) - t=1
[Tick 14]    Ready Queue (HIGH): { }
[Tick 14]    Ready Queue (LOW): { }
[Tick 14]    I/O Queue (DISK): { }
[Tick 14]    I/O Queue (TAPE): { 3 }
[Tick 14]    I/O Queue (PRINTER): { }

[Tick 15]    CPU: RUNNING (pid 1) - t=2
[Tick 15]    Ready Queue (HIGH): { }
[Tick 15]    Ready Queue (LOW): { }
[Tick 15]    I/O Queue (DISK): { }
[Tick 15]    I/O Queue (TAPE): { 3 }
[Tick 15]    I/O Queue (PRINTER): { }

[Tick 16]    TAPE: FINISHED (pid 3)
[Tick 16]    CPU: RUNNING (pid 1) - t=3
[Tick 16]    Ready Queue (HIGH): { 3 }
[Tick 16]    Ready Queue (LOW): { }
[Tick 16]    I/O Queue (DISK): { }
[Tick 16]    I/O Queue (TAPE): { }
[Tick 16]    I/O Queue (PRINTER): { }

[Tick 17]    SCHEDULER: PREEMPT (pid 1)
[Tick 17]    CPU: RUNNING (pid 3) - t=1
[Tick 17]    Ready Queue (HIGH): { }
[Tick 17]    Ready Queue (LOW): { 1 }
[Tick 17]    I/O Queue (DISK): { }
[Tick 17]    I/O Queue (TAPE): { }
[Tick 17]    I/O Queue (PRINTER): { }

[Tick 18]    CPU: RUNNING (pid 3) - t=2
[Tick 18]    Ready Queue (HIGH): { }
[Tick 18]    Ready Queue (LOW): { 1 }
[Tick 18]    I/O Queue (DISK): { }
[Tick 18]    I/O Queue (TAPE): { }
[Tick 18]    I/O Queue (PRINTER): { }

[Tick 19]    CPU: RUNNING (pid 3) - t=3
[Tick 19]    Ready Queue (HIGH): { }
[Tick 19]    Ready Queue (LOW): { 1 }
[Tick 19]    I/O Queue (DISK): { }
[Tick 19]    I/O Queue (TAPE): { }
[Tick 19]    I/O Queue (PRINTER): { }

[Tick 20]    SCHEDULER: PREEMPT (pid 3)
[Tick 20]    CPU: RUNNING (pid 1) - t=1
[Tick 20]    Ready Queue (HIGH): { }
[Tick 20]    Ready Queue (LOW): { 3 }
[Tick 20]    I/O Queue (DISK): { }
[Tick 20]    I/O Queue (TAPE): { }
[Tick 20]    I/O Queue (PRINTER): { }

[Tick 21]    CPU: RUNNING (pid 1) - t=2
[Tick 21]    Ready Queue (HIGH): { }
[Tick 21]    Ready Queue (LOW): { 3 }
[Tick 21]    I/O Queue (DISK): { }
[Tick 21]    I/O Queue (TAPE): { }
[Tick 21]    I/O Queue (PRINTER): { }

[Tick 22]    CPU: RUNNING (pid 1) - t=3
[Tick 22]    Ready Queue (HIGH): { }
[Tick 22]    Ready Queue (LOW): { 3 }
[Tick 22]    I/O Queue (DISK): { }
[Tick 22]    I/O Queue (TAPE): { }
[Tick 22]    I/O Queue (PRINTER): { }

[Tick 23]    SCHEDULER: PREEMPT (pid 1)
[Tick 23]    CPU: RUNNING (pid 3) - t=1
[Tick 23]    Ready Queue (HIGH): { }
[Tick 23]    Ready Queue (LOW): { 1 }
[Tick 23]    I/O Queue (DISK): { }
[Tick 23]    I/O Queue (TAPE): { }
[Tick 23]    I/O Queue (PRINTER): { }

[Tick 24]    SCHEDULER: TERMINATE (pid 3)
[Tick 24]    CPU: RUNNING (pid 1) - t=1
[Tick 24]    Ready Queue (HIGH): { }
[Tick 24]    Ready Queue (LOW): { }
[Tick 24]    I/O Queue (DISK): { }
[Tick 24]    I/O Queue (TAPE): { }
[Tick 24]    I/O Queue (PRINTER): { }

[Tick 25]    SCHEDULER: TERMINATE (pid 1)
[Tick 25]    CPU: IDLE - t=1
[Tick 25]    Ready Queue (HIGH): { }
[Tick 25]    Ready Queue (LOW): { }
[Tick 25]    I/O Queue (DISK): { }
[Tick 25]    I/O Queue (TAPE): { }
[Tick 25]    I/O Queue (PRINTER): { }
```

APÊNDICE D – PROTÓTIPO DO SIMULADOR EM PYTHON

```
from collections import deque

# Essas constantes representam as instruções que um processo pode realizar.
CPU, DISK, TAPE, PRINTER = range(4)

class Process:
    "Cada processo terá um tempo de início e uma lista de instruções."

    def __init__(self, start, instructions):
        self.start = start
        self.instructions  = instructions

        self.id = self.IOcounter = None


CPUtime = 0  # Rastreador global de tempo

# Lista de processos (ordenada de acordo com o tempo de início
processList = [
    Process(0, deque([DISK] + [CPU]*10)),
    Process(5, deque([CPU]*2 + [DISK] + [CPU]*13))
]

# Filas de processos prontos (0: alta prioridade | 1: baixa prioridade)
queues = [deque(), deque()]


ID_TEST = 0  # solução temporária para gerar ID's de processos.

def createProcesses():
    "Essa função itera pela lista de processos, buscando aqueles que podem ser iniciados."
    global ID_TEST
    i = 0
    for process in processList:
        if CPUtime == process.start:
            queues[0].appendleft(process)
            print("new process")
            print(queues)
            ID_TEST += 1
            process.id = ID_TEST
            i += 1
        else:
            break

    if i > 0:
        del processList[0:i]

currentProcess = None  # Processo atualmente executado na CPU.
```

```
timeUsed = 0           # Tempo usado pelo processo atual.

DISK_LIST = deque()

def simulateIO():
    "Deve ser rodada a cada tick da simulação. Simula a operação dos dispositivos de E/S."
    if DISK_LIST:
        DISK_LIST[-1].IOcounter -= 1
        if DISK_LIST[-1].IOcounter == 0:
            queues[0].appendleft(DISK_LIST.pop())

def getNextProcess():
    global currentProcess, timeUsed

    for queue in queues:
        if queue:
            currentProcess = queue.pop()
            timeUsed = 0
            return
    currentProcess = None

def scheduler():
    if currentProcess == None:
        getNextProcess()
    else:
        if not currentProcess.instructions:
            getNextProcess()

        elif timeUsed == 3:
            queues[1].appendleft(currentProcess)
            getNextProcess()

    while currentProcess and currentProcess.instructions.popleft() != CPU:
            DISK_LIST.appendleft(currentProcess)
            currentProcess.IOcounter = 7
            getNextProcess()

def main():
    global currentProcess, CPUtime, timeUsed

    while True:
        createProcesses()
        scheduler()

        CPUtime += 1
        if currentProcess:
            print(CPUtime, ':' , currentProcess.id)
            timeUsed += 1

        simulateIO()
```