Universidade Federal do Espírito Santo
Departamento de Informática
Trabalho Prático
Sistemas Operacionais (INF15980)

Implementação de um Mini-Kernel *Multithread* com suporte a Escalonamento por FCFS, RR e Prioridade (Preemptiva)

---

1. Objetivo Geral

Este trabalho tem como objetivo o desenvolvimento um sistema de escalonamento de processos multithread em linguagem C, com suporte a múltiplas políticas de escalonamento (First Come First Served - FCFS, Round-Robin - RR e Prioridade Preemptiva), utilizando threads do padrão POSIX (*lib. pthread*) para a representação e simulação de execução de múltiplos fluxos de execução por processo. O Linux implementa a modelagem 1:1 de threads, ou seja, cada thread é enxergada pelo Kernel e escalonada como tal. O comportamento do sistema deve respeitar este paradigma.

O trabalho é projetado para consolidar os seguintes conceitos:

- Escalonamento e sincronização de processos e threads
- Controle de concorrência com mutexes e variáveis de condição
- Estruturas de dados de controle de processos (Bloco de Controle de Processos - BCP) e threads (Task Control Block - TCB)
- Implementação de políticas de escalonamento reais - FCFS, RR e Prioridade Preemptiva
- Utilização de filas de execução e simulação de tempo de execução

2. Visão Geral do Mini-Kernel

Seu sistema deve implementar um escalonador de processos *multithread* em C, simulando diferentes políticas de escalonamento (FCFS, Round Robin e Prioridade) em um ambiente *concorrente*. Cada processo do sistema é representado por uma estrutura com suas características (PID, duração, prioridade, tempo de chegada, número de threads, etc.) e pode conter múltiplas threads executando simultaneamente — todas sincronizadas por meio de variáveis de condição e mutexes.

As principais funcionalidades que o sistema deve implementar, de modo geral, são:

- Emular a criação e execução concorrente de processos reais com múltiplas threads
- Controlar a fila de prontos (ready queue) e aplicar a política de escalonamento escolhida
- Realizar a sincronização e preempção entre as execuções dos processos
- Registrar logs de eventos de escalonamento para posterior análise

2.1 Inicialização dos processos e threads

A execução do sistema é iniciada pela leitura de um conjunto de dados de entrada, referentes a descrição dos processos que serão criados, armazenados e escalonados pelo escalonador. Uma lista de processos é preenchida com os dados fornecidos pelo usuário, incluindo:

- PID (identificador único) - deve-se recuperar esta informação na leitura do processo (não confundir com getpid())
- Duração total de execução (em milissegundos)
- Prioridade (quanto menor o número, maior a prioridade)
- Quantidade de threads que o processo irá utilizar
- Tempo de chegada (em milissegundos, relativo ao início da execução)

Cada processo é armazenado como uma estrutura BCP, contendo, além dos dados mencionados:

- Um mutex exclusivo para controlar acesso concorrente às suas variáveis internas
- Uma variável de condição que será utilizada para sinalizar às threads quando elas podem executar
- Um vetor com os identificadores de suas threads

- Uma variável de controle do tempo de execução restante para a execução do processo como um todo
- O estado do processo (podendo ser READY, RUNNING ou FINISHED)

2.2. Simulação de chegada e criação de threads

Após a inicialização dos processos, o sistema inicia a simulação do tempo de chegada:

- Deve-se aguardar ativamente até que o tempo atual (em milissegundos desde o início) alcance o tempo de chegada do processo - iniciando as threads associadas ao processo. A prioridade das threads descende da prioridade dos processos.

- **Observação importante:** as threads são criadas, mas não começam a "trabalhar" imediatamente. Elas ficam adormecidas, esperando pela variável de condição que indica o momento de execução, controlado pelo escalonador.

2.3. Execução dos processos pelas threads

Para cada thread associada a um processo, o seguinte pipeline é executado:

- Verifica-se, dentro de um loop, se o processo está no estado RUNNING
- Caso não esteja, ela bloqueia-se na variável de condição, aguardando sinal do escalonador
- Quando liberada, a thread simula a execução por um pequeno intervalo de tempo (500ms, por exemplo), utilizando *usleep* - simula de fato a execução da thread (não se esqueça de descontar este valor em algum lugar!!!)
- Após esse tempo, deve-se reduzir o tempo restante do processo de forma segura (pense nisso!!!)
- Se o tempo restante chega a zero, a thread (ou a primeira que detectar isso) muda o estado do processo para FINISHED e sinaliza as outras para encerrar.
- As threads têm 500ms para executarem até que uma nova consulta por outra thread/processo seja realizada (lembrando que isto depende da política de escalonamento).

As threads terminam sua execução de forma coordenada com base no estado global do processo.

2.4. Funcionamento do escalonador

O sistema inicia, em paralelo, uma thread escalonadora, que é responsável por:

- Monitorar a fila de prontos, onde estão os processos prontos para execução.
- Selecionar, de acordo com a política escolhida (FCFS, Round Robin ou Prioridade), o próximo processo que deve ser executado.
- Atualizar o estado do processo
- Sinalizar todas as threads do processo com a variável de condição para que comecem a execução
- Aguardar o término ou o tempo de quantum, dependendo da política

Durante a execução:

- Em FCFS, o escalonador espera o processo terminar completamente antes de passar ao próximo
- Em Round Robin, ele cede a CPU após um tempo fixo (quantum - 500ms), reinserindo o processo na fila caso ainda tenha tempo restante.
- Na política de prioridade, ele verifica frequentemente se um processo de prioridade maior chegou, e em caso afirmativo, preempção ocorre, reinserindo o processo atual na fila (***somente são retirados processos de execução - são preemptados - após o término do quantum***).

A fila de prontos PODE SER implementada como uma fila circular protegida por mutex, com suporte a:

- Inserção no final
- Remoção do início
- Remoção de elementos arbitrários (para preempção por prioridade)

2.5. Sincronização entre processos e escalonador

Para evitar problemas de concorrência, o sistema utiliza:

- Mutexes de processo: para proteger leitura e escrita nos campos remaining_time, state, etc.
- Mutex da fila de prontos: para garantir que apenas uma thread (escalonador ou produtor de processos) manipule a fila simultaneamente
- Variáveis de condição:
    - scheduler_condition_variable: permite que o escalonador durma quando a fila está vazia, acordando quando novos processos chegam

    - pcb->condition_variable: usada para sinalizar às threads do processo que elas podem prosseguir (quando ele entra em RUNNING)

2.6. Finalização da execução

O escalonador monitora constantemente se:

- A fila de prontos está vazia
- Todos os processos já chegaram (controlado por uma variável booleana global)
- Nenhum processo está em execução atualmente

Quando essas três condições são verdadeiras, ele termina. Em seguida, o processo principal:

- Aguarda o encerramento de todas as threads de todos os processos (com join)
- Salva o conteúdo do *buffer* de log em um arquivo (*escalonador_log_minikernel.txt*), que registra eventos como:
    - Qual processo foi executado
    - Quantum utilizados
    - Mudanças de estado
    - Finalizações de processo

2.7 Potenciais Variáveis e Estruturas

Variável	Funcionalidade
PCB *pcb_list	Lista de todos os processos do sistema
pthread_t *thread_ids	Identificadores das threads associadas a um processo
process_len	Duração total de execução do processo (ms)
remaining_time	Tempo restante de execução
start_time	Tempo de chegada do processo ao sistema
priority	Nível de prioridade (1 = mais alta)
state	Estado atual: READY, RUNNING, FINISHED
runqueue	Fila de prontos circular
generator_done	Sinaliza que todos os processos foram criados e enfileirados
current_process	Processo atualmente em execução
quantum	Quantum usado no Round Robin (ms)
scheduler_type	Define política de escalonamento
log_buffer	Armazena mensagens de log durante a execução

2.8 Estrutura BCP

A estrutura BCP representa o bloco de controle de processo. Cada instância descreve um processo abstrato do sistema, contendo tanto suas informações estáticas (definidas na criação) quanto dinâmicas (modificadas durante a execução).

```
typedef struct {
  int pid;
  int process_len;
  int remaining_time;
  int priority;
  int num_threads;
  int start_time;

  ProcessState state;

  pthread_mutex_t mutex;
  pthread_cond_t cv;
  pthread_t *thread_ids;

} PCB;
```

2.8.1 Descrição do BCP

A estrutura BCP contém diversos atributos fundamentais para representar o estado e o comportamento de um processo no sistema.

- O campo pid (identificador do processo) é um valor inteiro único e serve para identificação em *logs*, mensagens de debug e no escalonador. Ele não é alterado após a criação do processo.
- O campo process_len representa a duração total do processo, em milissegundos. Este valor é definido na entrada e permanece constante durante a execução. Serve de base para o cálculo e manutenção do campo remaining_time.
- O campo remaining_time indica quanto tempo de execução ainda falta para que o processo seja finalizado. Ele é decrementado pelas threads durante a execução. Quando chega a 0, o processo é considerado finalizado (estado FINISHED). Por ser acessado por múltiplas threads, a manipulação deste campo deve ser **segura e atômica.**
- A priority define a prioridade do processo, com valores de 1 (maior prioridade) a 5 (menor). Este campo é utilizado exclusivamente em algoritmos de

escalonamento baseados em prioridade e não é modificado após a criação do processo.

- O campo num_threads indica quantas threads fazem parte do processo. Este número é definido na entrada do usuário e determina quantas threads devem ser aguardadas com pthread_join ao final da execução.
- O start_time especifica o momento (em milissegundos desde o início da execução) em que o processo deve entrar no sistema. As threads só são criadas quando o tempo atual atinge ou ultrapassa este valor. Este campo é consultado pelo escalonador para determinar se o processo já está disponível para execução.
- O campo state representa o estado atual do processo: READY, RUNNING ou FINISHED. O estado controla o fluxo das threads: elas só executam quando o estado é RUNNING. A transição de estados ocorre de forma controlada pelo escalonador: de READY para RUNNING quando o processo é escalonado, de RUNNING para FINISHED quando remaining_time <= 0, e de RUNNING para READY em políticas com preempção (como Round Robin). Este campo também é protegido por mutex.
- O mutex é utilizado para garantir acesso exclusivo a campos sensíveis do processo. Sempre que uma thread ou o escalonador precisa acessar ou modificar essas variáveis, o mutex é travado, evitando condições de corrida.
- A variável de condição cv permite que as threads esperem passivamente até que o processo esteja liberado para execução. Enquanto o estado não for RUNNING, as threads se bloqueiam nesta variável. Quando o processo entra em execução, o escalonador utiliza broadcast para acordar as threads. A cv também é usada para sinalizar o término do processo.
- O vetor *thread_ids guarda os identificadores das threads do processo, facilitando a utilização de pthread_join() para garantir o término de todas as threads no final da execução. Esse vetor é preenchido durante a criação das threads.

2.8.2 Interação entre os Atributos

Durante a execução típica do sistema:

- O start_time determina o momento em que as threads de um processo serão criadas.
- Após a criação, elas se bloqueiam na variável de condição cv até que o estado do processo seja alterado para RUNNING.
- O escalonador então escolhe o processo, altera seu estado para RUNNING e sinaliza as threads para começarem a executar.

- As threads acordam, travam o mutex, e decrementam o remaining_time.
- Quando remaining_time chega a zero, uma thread define o estado como FINISHED e sinaliza as demais.
- O escalonador, ao detectar o fim do processo, o remove da fila de execução.
- Por fim, o vetor thread_ids é utilizado para realizar join nas threads e garantir que todas tenham finalizado corretamente.

A estrutura BCP é o núcleo da simulação do sistema, representando cada processo de forma completa e realista. Ela encapsula desde a identidade e carga de trabalho do processo até sua prioridade e estado de execução. Além disso, oferece mecanismos seguros de sincronização concorrente, fundamentais para a simulação correta do escalonamento. O manuseio correto dos campos críticos como remaining_time, state, mutex e cv é essencial para evitar problemas de concorrência, inconsistências ou deadlocks. Com isso, a simulação consegue representar com fidelidade o comportamento de sistemas operacionais reais.

2.9 A Estrutura TCB

A estrutura TCB representa o bloco de controle de threads. Cada instância descreve uma thread relacionada a um processo específico contendo suas informações de contexto para escalonamento no sistema.

```
typedef struct {
  PCB* pcb;
  int thread_index;
} TCB;
```

2.9.1 Descrição do TCB

A estrutura TCB é utilizada para passar informações específicas a cada thread criada dentro de um processo, e contém dois campos principais:

- Um ponteiro que referencia o processo (BCP) ao qual a thread pertence, permitindo que a thread tenha acesso às informações compartilhadas do processo, como tempo restante, estado de execução e mecanismos de sincronização (mutexes e variáveis de condição).
- Um inteiro *thread_index* que identifica a posição ou número da thread dentro da lista de threads do processo, possibilitando que cada thread tenha um identificador único no contexto do processo.

Essa estrutura é alocada dinamicamente para cada thread criada e passada como argumento para a função do processo, permitindo que cada thread saiba a qual processo pertence e qual é seu índice local.

2.10 Comportamento esperado

- Preempção e concorrência: seu escalonador deve lidar corretamente com a mudança de estados (READY, RUNNING, FINISHED) dos processos, o que exige muita atenção à ordem de travamento/destravamento dos mutexes.

- Condicionalidade por prioridade: a política por prioridade precisa avaliar *constantemente* a fila, identificando se processos mais importantes chegaram para preemptar os que estão em execução corrente.

- Gerenciamento de tempo (em milissegundos): como não há temporizador real, o sistema simula tempo com *usleep()* e *gettimeofday()*.

- Funcionamento geral do Mini-Kernel:
    - A thread principal é o ponto de partida da aplicação e tem como objetivo configurar, inicializar e executar a simulação de escalonamento de processos com múltiplas threads. Ela deve controlar o conjunto de threads criadas na leitura de cada processo (por meio de BCPs e TCBs) definido pelo usuário. Para cada processo, são lidos (i) duração total de execução em milissegundos, (ii) prioridade (sendo 1 a mais alta e 5 a mais baixa), (iii) número de threads que o processo conterá e (iv) o tempo de chegada em milissegundos. Com base nestas informações, cada processo é devidamente configurado: seu tempo restante é definido com base na duração, o estado inicial é ajustado para READY, e são inicializados os mecanismos de sincronização como mutexes e variáveis de condição, que serão usados para controlar a execução concorrente das threads.
    - Após a configuração dos processos, o usuário escolhe qual algoritmo de escalonamento será utilizado: FCFS (First Come, First Served), Round Robin ou Prioridade Preemptiva. Com base nessa escolha, o escalonador é configurado, definindo parâmetros relacionados a implementação como o tamanho máximo da fila de processos prontos, o tempo de quantum para o Round Robin (exemplo: 500 ms), e o tipo de escalonador selecionado.
    - Deve-se criar as threads de cada processo, respeitando o tempo de chegada de cada um. Assim que o tempo de chegada é alcançado, o

processo tem suas threads iniciadas, sendo que cada uma referencia o BCP de seu processo, e seu índice local. Os primeiros processos são inseridos na fila de prontos.

- A lógica de escalonamento escolhida deve ser coordenada de forma independente (sugestão, um novo processo ou thread dedicada), controlando quais processos são colocados em execução ao longo da simulação (fica fácil aguardar o fim de cada thread de cada processo dessa forma, certo?).
- Após o escalonador concluir sua tarefa, o programa garante que todas as threads dos processos sejam finalizadas corretamente. Por fim, o conteúdo do log, que foi gerado durante a execução (incluindo eventos como início e término de processos), é salvo em um arquivo chamado escalonador_log_minikernel.txt. A função encerra sem imprimir nada na tela (vide seção 4.3).

3. Sua Tarefa

A sua tarefa será implementar o Mini-Kernel especificado que lê processos, cria threads, os escalona adequadamente de acordo com a política de escalonamento definida pelo usuário e simula suas execuções. Devem ser implementadas DUAS (2) versões do Mini-Kernel, uma simulando a execução dos processos em um sistema monoprocessador, e outra simulando a execução dos processos em um sistema multiprocessado, com **dois (2)** processadores. Para tal, deve-se seguir os seguintes passos:

- Receber da entrada padrão as informações de processos:
    - Duração (ms)
    - Prioridade (1–5)
    - Número de threads
    - Tempo de chegada (ms)
- Simular a execução dos processos com múltiplas threads
- Escalonar os processos com base em três políticas:
    - FCFS (First Come First Serve)
    - Round-Robin (quantum fixo)
    - Prioridade Preemptiva
- Manter uma fila de processos prontos (ready queue)
- Registrar todos os eventos relevantes em um log de execução
- Implementar sincronização adequada entre threads do processo e o escalonador
- Encerrar corretamente todas as threads e liberar memória alocada

4. Entrada e Saída

4.1. Entrada

Todas as informações serão dadas em um arquivo de entrada. Este arquivo vai conter a quantidade *n* de processos a serem lidos, todos os processos p_i (com *i* variando de 0 até *n-1*) e suas respectivas informações. A primeira linha do arquivo terá *n*. Para as demais *n* entradas, serão lidos:

- Duração do processo p_i (em milissegundos)
- Prioridade do processo p_i (de 1 a 5)
- Quantidade de threads que compõem o processo p_i
- Tempo de chegada do processo (em milissegundos)

Por fim, a última linha indica qual a política de escalonamento a ser adota: 1 para FCFS, 2 para RR e 3 para Prioridade Preemptiva.

Importante: As threads de um processo herdam sua prioridade, e o tempo de execução das threads é a divisão do tempo de duração do processo (lido pelo terminal).

A seguir, o conteúdo de um arquivo de entrada para o Mini-Kernel:

3 // *n de processos*
1000 // *duração do processo*
3 // *prioridade do processo*
2 // *número de threads do processo*
0 // *tempo de chegada do processo*
2000
1
1
200
1500
2
1
100
1 // *política de escalonamento*

4.2. Saída

A saída do trabalho deverá ser salva em um arquivo, também de texto, com as informações de execução dos processos. O arquivo deve ter o nome “log_execucao_minikernel.txt”, não importando qual política de escalonamento tenha

sido adotada. Para o exemplo de entrada na seção 4.2, tem-se o seguinte resultado esperado, no modo monoprocessaror:

[FCFS] Executando processo PID 1
[FCFS] Executando processo PID 2
[FCFS] Executando processo PID 3
Escalonador terminou execução de todos processos

4.3. Execução do trabalho

Para testar seu trabalho, o professor executará comandos seguindo o seguinte padrão.

tar -xzvf <nome_arquivo>.tar.gz
make <versao>
./trabSO <nome_arquivo_entrada>

É extremamente importante que vocês sigam esse padrão. Seu programa não deve solicitar a entrada de nenhum valor e também não deve imprimir nada na tela.

Por exemplo, se o nome do arquivo recebido for 2004209608.tar.gz, os dados de entrada estiverem em “entrada.txt”, o professor executará para a versão monoprocessador do Mini-Kernel:

tar -xzvf 2004209608.tar.gz
make monoprocessador
./trabSO entrada.txt

Para a versão multiprocessador, será executado:

tar -xzvf 2004209608.tar.gz
make multiprocessador
./trabSO entrada.txt

5. Detalhes de Implementação

A seguir, alguns detalhes, comentários e dicas sobre a implementação. Muita atenção aos usuários do Sistema Operacional Windows.

- O trabalho deve ser implementado em C. A versão do C a ser considerada é a presente nos Computadores do LabGrad (Ubuntu).
- Seu programa deve ser, obrigatoriamente, compilado com o utilitário make. Crie um arquivo Makefile que gera como executável para o seu programa um arquivo de nome *trabSO*.
- Ao longo do desenvolvimento do trabalho, **certifique-se que o seu código não está vazando memória testando-o com o valgrind**. Não espere terminar o código para usar o valgrind, incorpore-o no seu ciclo de desenvolvimento. Ele é uma ferramenta excelente para se detectar erros sutis de acesso à memória que são muito comuns em C. Idealmente o seu programa deve sempre executar sem nenhum erro no valgrind.

6. Regras para o desenvolvimento do trabalho

- Data da Entrega: O trabalho deve ser entregue até as 23:59h do dia 26/08/2025. Não serão aceitos trabalhos após essa data.
- Grupo: O trabalho pode ser feito em duplas.
- Como entregar: Pela atividade criada no Classroom. É encorajada a entraga pelo GitHub - seu grupo pode criar um repositório privado, compartilhar com o professor e colocar os arquivos fontes lá. Ainda, a dupla pode utilizar o README para dar detalhes da implementação e decisões para a modelagem do trabalho. Este tipo de entrega agrada muito o professor. A submissão deve incluir todos os arquivos de código e um Makefile, como especificado anteriormente. Somente uma pessoa do grupo deve enviar o trabalho no Classroom. Coloque a matrícula de todos integrantes do grupo (separadas por vírgula) no nome do arquivo do trabalho.
- Recomendações: Modularize o seu código adequadamente. Crie códigos claros e organizados. Utilize um estilo de programação consistente. Comente o seu código extensivamente. Não deixe para começar o trabalho na última hora.

7. Avaliação

Assim como especificado no plano de ensino, o trabalho vale 10 pontos.

- A parte de implementação será avaliada de acordo com a fração e tipos de casos de teste que seu trabalho for capaz de resolver de forma correta. Serão disponibilizados casos de teste públicos e as respectivas saídas para o modo *monoprocessador*. As saídas esperadas para o modo multiprocessador não serão disponibilizadas e significam 35% da nota do trabalho.
- Trabalhos com erros de compilação receberão nota zero.
- Trabalhos que gerem *segmentation fault* para algum dos casos de teste disponibilizados no Classroom serão severamente penalizados na nota.
- Trabalhos com *memory leak* (vazamento de memória) sofrerão desconto na nota.
- O uso da primitiva goto implica em nota zero. Modularize bem o código e utilize variáveis globais somente quando ESTRITAMENTE necessário.
- Organização do código e comentários valem nota. Trabalhos confusos e sem explicação sofrerão desconto na nota.
- Caso seja detectada cópia (entre alunos ou da Internet), todos os envolvidos receberão nota zero. Caso as pessoas envolvidas em suspeita de cópia discordem da nota, amplo direito de argumentação e defesa será concedido. Neste caso, as regras estabelecidas nas resoluções da UFES serão seguidas.
- A critério do professor, poderão ser realizadas entrevistas com os alunos, sobre o conteúdo do trabalho entregue. Caso algum aluno seja convocado para uma entrevista, a nota do trabalho será dependente do desempenho na entrevista. (Vide item sobre cópia, acima.)
- O uso da *lib pthread* para criação e controle de threads é obrigatório, bem como o uso de mutexes para sincronização. Trabalhar com tempo em milissegundos, utilizando usleep, gettimeofday, e outras funções que manipulem tempo.
- Cuidado com condições de corrida e deadlocks. Penalização SEVERA na nota.
- O código deve ser portável (Linux/Unix).