PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

INF 1316
Relatório de Atividade - Trabalho 1

2210872 Felipe Antelo Machado de Oliveira
2112171 Pedro Henrique de Oliveira Valentim Gomes

Observação Inicial: Com o objetivo de garantir maior sincronia e robustez na comunicação entre o interpretador e o escalonador, o grupo julgou importante introduzir o módulo “executor.c”. Este arquivo atua como processo principal do sistema, sendo responsável por inicializar tanto o interpretador quanto o escalonador como subprocessos, além de gerenciar a criação da FIFO para comunicação entre eles.
Essa abordagem substitui versões anteriores em que o escalonador era executado de forma isolada. Agora, o fluxo correto de execução consiste em compilar todos os módulos e, posteriormente, rodar o programa a partir do “executor.c”. Isso assegura o correto funcionamento do ambiente multi-processos e evita possíveis condições de corrida ou problemas de sincronização entre pai e filhos.
O grupo optou por adicionar essa nota ao começo do relatório apenas para justificar o uso do módulo e, também, alertar leitores da necessidade de rodar o **“executor.c”** ao invés do usual **“escalonador.c”**.

Objetivo:
Para o trabalho 1 - Escalonador e interpretador, a dupla deve criar um código capaz de, ao ler um arquivo texto que descreve processos de diferentes tempo e diferentes tempos de uso de CPU, simular, de acordo com os princípios de ambiente UNIX, como funciona sistemas de preempção com time slices e prioridades diversas. Para entender melhor o trabalho, devemos desmembrar a tarefa em diferentes partes, sendo elas:

- **Interpretador:** Arquivo em c que lê um arquivo exec.txt, linha a linha, parseando os comandos.
- **Escalonador:** Programa que recebe ordens do interpretador e gerencia a execução dos processos, controlando também a sua política de ordem de prioridade, sendo elas de acordo com os tipos REAL-TIME, PRIORIDADE e ROUND-ROBIN (prioridades nessa ordem)
- **Programas de teste**: processos simples em C, conforme o enunciado passado pelo professor (alguns são CPU-bound, outros possuem tempo controlado).
- **Relatório em PDF:** O relatório explicativo do código em questão que está

sendo lido agora. O relatório deve conter informações bases, além de observações e conclusões.

Estrutura do Programa :

Interpretador.c que lê o exec.txt linha por linha funciona via um loop principal que envia para o escalonador e espera 1 segundo. Isso garante que os processos sejam escalonados um a um, simulando a chegada gradual dos processos ao sistema.

Escalonador.c que gerencia a criação, pausa, retomada e término dos processos filhos e é responsável por aplicar as políticas de escalonamento. Funciona através de um loop infinito, onde continuamente lê do FIFO (com fgets). Ao receber um novo comando, faz o fork e executa o processo correspondente (via execlp). O processo é imediatamente parado (SIGSTOP) para aguardar o escalonamento. Depois ele escolhe o processo seguinte a executar segundo a hierarquia REAL-TIME > PRIORIDADE > ROUND-ROBIN, usando laços e verificações para selecionar o processo correto em cada "tick" (UT). Para tal, faz controle por sinais SIGCONT e SIGSTOP.

Executor.c é responsável por iniciar o sistema, isso é, criar o FIFO, gerar processos filhos para escalonador e interpretador e esperar que ambos terminem. Ele funciona através da criação do nosso FIFO, ou seja, garante que a pipe nomeada esteja pronta para ser usada. Após isso, realiza um fork( ) entre processos em que um filho roda o escalonador e outro filho roda o interpretador. Por fim, através de waitpid( ), o processo pai espera a finalização dos filhos.

Temos também os demais processos (px.c) que simulam um processo CPU-bound contínuo, apenas "dormindo" em um loop infinito ou ao menos é o que parece ao vê-los de forma isolada. Ao observar o código como um todo, percebemos que cada processo filho é tratado de um modo pelo resto do código de acordo com o que o nosso “exec.txt” determina no seu modo e configurações gerais (tempo inicial, tempo total necessário etc). Utiliza um simples loop while(1) com sleep(1) para permanecer ativo.

Solução do programa:

Buscando uma explicação clara, trarei abaixo uma explicação geral de funcionamento de cada programa no contexto do projeto.

Funcionamento Geral

Executor:

- Cria FIFO se não existir
- Inicializa dois processos filhos: um para o escalonador e outro para o interpretador.

Interpretador:

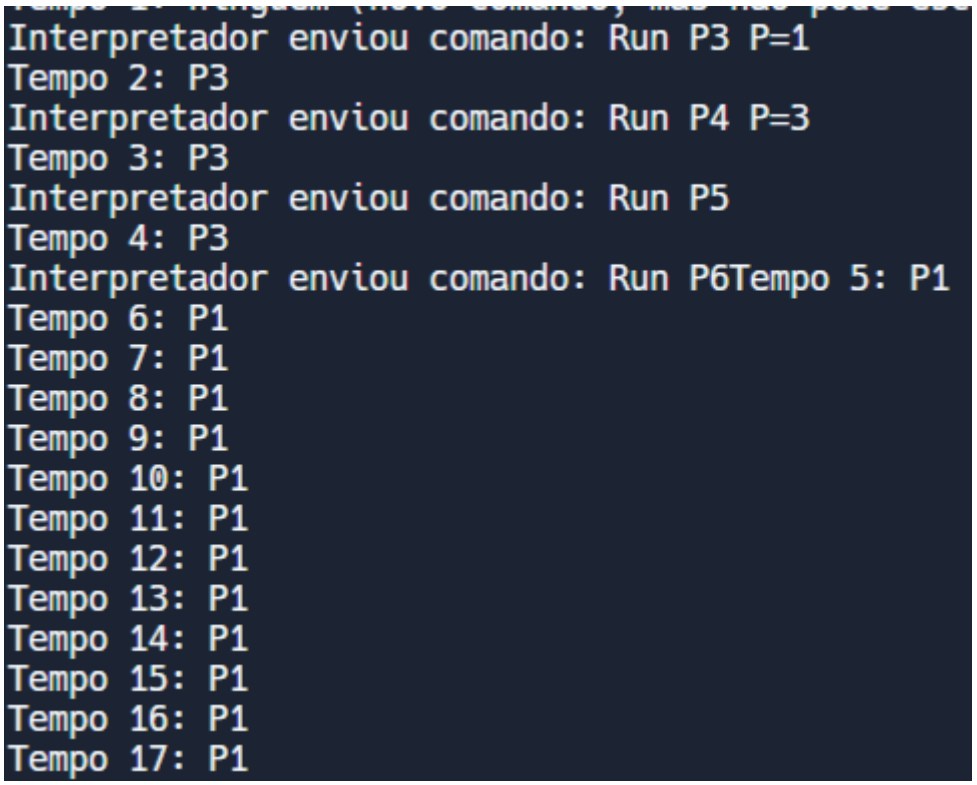
- Lê o arquivo exec.txt linha a linha, cada linha contendo a especificação de um processo a ser executado.
- Para cada linha lida, envia o comando para o FIFO (fifo_escalonador) e aguarda 1 segundo antes de enviar o próximo comando, garantindo que os processos sejam enviados um a um.

Escalonador:

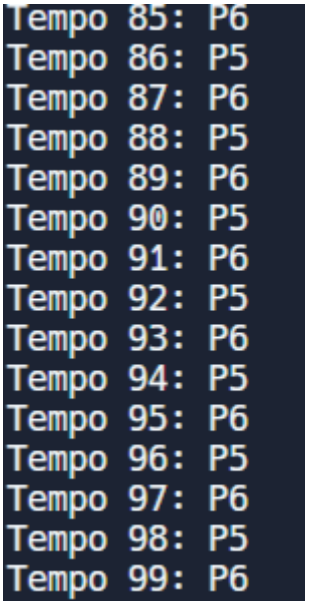
- Lê continuamente os comandos vindos do FIFO.
- Ao receber um comando, identifica o tipo de escalonamento (PRIORIDADE, ROUND-ROBIN ou REAL-TIME) e cria o processo correspondente, armazenando suas informações em uma estrutura de dados.
- Controla a execução dos processos conforme a política:
    - REAL-TIME: Tem prioridade máxima. O escalonador verifica se é o momento de iniciar a execução e por quanto tempo o processo deve rodar. Conflitos de agendamento são verificados conforme exigido no enunciado.
    - PRIORIDADE: Executa processos finitos (3 unidades de tempo cada) de acordo com a prioridade (menor valor = maior prioridade).
    - ROUND-ROBIN: Executa processos CPU-bound em fatias de tempo, em esquema de rodízio.
- Utiliza sinais para pausar e retomar processos conforme a política e o tempo de execução, garantindo a alternância correta entre eles.
- A cada unidade de tempo, imprime na tela qual processo está executando ou se está aguardando novo comando, facilitando a análise do comportamento do escalonador.

Processos de Teste (ex: P1.c):

- São programas simples que simulam processos reais a serem escalonados (neste exemplo, loops infinitos que dormem 1 segundo por iteração).
- Como pode ser visto abaixo, o código realiza uma iniciação e segue os prints de acordo com as ordens de prioridade



- Depois de terminar os processos gerais, o código roda processos específicos de Round-Robin, uma vez que esses não tem duração limite



Observação e conclusões:

- Hierarquia de Políticas (diferentes tipos) foi aplicada com sucesso.
- Comunicação entre processos realizada pelo nosso FIFO (pipe) funciona
- Preempção é existente e funcional através do uso de sinais (SIGSTOP, SIGCONT) para suspender e retomar processos filhos, simulando um ambiente real de escalonamento multitarefa.

O grupo acredita, pelos resultados obtidos dos diferentes testes e por tudo que aprendemos da matéria, que o código final está completo e funcional. Ao todo, o maior problema que encontramos no trabalho foi sincronizar de forma eficiente o interpretador e o escalonador (no caso do nosso grupo, ainda havia um executor). Em certos casos, havia um gargalo na transferência de mensagens (sinais) que causava atrasos no prints que não seriam ideias. Em outros casos, também ocorriam erros de ordenação, nos quais a falta de sincronização causava com que um processo na fila fosse pulado, atrasando sua execução total ou tirando o processo da fila por completo. Para consertar isso, o grupo precisou de, após diversos testes e debugs, introduzir variáveis específicas para acompanhar a execução dos processos atuais de forma a tomar atitudes e ordens entre escalonador e interpretador que fossem, ao mesmo tempo, mais rápidas e imediatas somente após variáveis que indicassem se ele havia sido "lido" ou não.

O trabalho como um todo girou ao redor de saber como utilizar corretamente sinais para controlar qual processo filho estaria atuando e transferir as informações para o pai (executor) para que ele pudesse tomar as ações necessárias. Também tivemos de tomar cuidado com questões de memória compartilhada, a fim de garantir que não houvesse nenhuma violação de segurança ou perda de dados ao longo do código. Entretanto, após aplicar técnicas similares às de labs anteriores, isso deixou de ser um problema. Outra questão crucial foi manter as prioridades entre tipos e garantir que o programa só seria substituído após determinado time-slice ou, naturalmente, após seu fim de execução. Para conseguir tal, foram necessárias diferentes refatorações para modificar o formato do loop do código e a sua ordem de tarefas.

Portanto, apesar de todas as dificuldades iniciais, o trabalho final funcionou da forma que foi planejado. Acredito que, se questionado, a única coisa que eu mudaria seria introduzir prints mais claros para o usuário, especialmente visando aqueles que não compreendem muito bem o que está sendo tratado no trabalho. Entretanto, como ainda assim ficou um print que evidencia o processo e resultado, não acredito que foi um fator faltante para o trabalho.

TimeLine (Resumida):

- P1 - 5~24
- P2 - 30~34
- P3 - 2~3,
- P4 - 25~27
- P5 - 28 ~Alterna P6 infinito
- P6 - 29 ~Alterna P5 infinito

TimeLine (Total):

Tempo 0: ninguém
Tempo 1: ninguém
Tempo 2: P3

Tempo 3: P3
Tempo 4: P3
Tempo 5: P1
Tempo 6: P1
Tempo 7: P1
Tempo 8: P1
Tempo 9: P1
Tempo 10: P1
Tempo 11: P1
Tempo 12: P1
Tempo 13: P1
Tempo 14: P1
Tempo 15: P1
Tempo 16: P1
Tempo 17: P1
Tempo 18: P1
Tempo 19: P1
Tempo 20: P1
Tempo 21: P1
Tempo 22: P1
Tempo 23: P1
Tempo 24: P1
Tempo 25: P4
Tempo 26: P4
Tempo 27: P4
Tempo 28: P5
Tempo 29: P6
Tempo 30: P2
Tempo 31: P2
Tempo 32: P2
Tempo 33: P2
Tempo 34: P2
Tempo 35: P5
Tempo 36: P6
Tempo 37: P5
Tempo 38: P6
Tempo 39: P5
Tempo 40: P6
Tempo 41: P5
Tempo 42: P6
Tempo 43: P5
Tempo 44: P6
Tempo 45: P5
Tempo 46: P6

Tempo 47: P5
Tempo 48: P6
Tempo 49: P5
Tempo 50: P6
Tempo 51: P5
Tempo 52: P6
Tempo 53: P5
Tempo 54: P6
Tempo 55: P5
Tempo 56: P6
Tempo 57: P5
Tempo 58: P6
Tempo 59: P5
Tempo 60: P6
Tempo 61: P5
Tempo 62: P6
Tempo 63: P5
Tempo 64: P6
Tempo 65: P1
Tempo 66: P1
Tempo 67: P1
Tempo 68: P1
Tempo 69: P1
Tempo 70: P1
Tempo 71: P1
Tempo 72: P1
Tempo 73: P1
Tempo 74: P1
Tempo 75: P1
Tempo 76: P1
Tempo 77: P1
Tempo 78: P1
Tempo 79: P1
Tempo 80: P1
Tempo 81: P1
Tempo 82: P1
Tempo 83: P1
Tempo 84: P1
Tempo 85: P5
Tempo 86: P6
Tempo 87: P5
Tempo 88: P6
Tempo 89: P5
Tempo 90: P2

Tempo 91: P2
Tempo 92: P2
Tempo 93: P2
Tempo 94: P2
Tempo 95: P6
Tempo 96: P5
Tempo 97: P6
Tempo 98: P5
Tempo 99: P6
Tempo 100: P5
Tempo 101: P6
Tempo 102: P5
Tempo 103: P6
Tempo 104: P5
Tempo 105: P6
Tempo 106: P5
Tempo 107: P6
Tempo 108: P5
Tempo 109: P6
Tempo 110: P5
Tempo 111: P6
Tempo 112: P5
Tempo 113: P6
Tempo 114: P5
Tempo 115: P6
Tempo 116: P5
Tempo 117: P6
Tempo 118: P5
Tempo 119: P6
Tempo 120: P5