shell e processos

Nathan Benedetto Proença
8941276
Victor Sena Molero
8941317

arquitetura do shell

O shell necessita gerenciar três tarefas:

1. leitura do comando
2. execução do comando
3. tratamento do retorno do comando

Essas tarefas são executadas num laço enquanto o usuário não terminar o shell.

leitura do comando

A leitura do comando é gerenciada pela biblioteca `readline`.

Contudo, ela só gerencia a leitura da string. Ainda é necessário extrair as informações para prosseguir com a execução. Para isso, criamos funções próprias.

A `build_prompt_string` gera o prompt para o usuário.

A `count_arguments` separa o que foi lido nas partes necessárias para execução.

A `build_call` reune o que é necessário para prosseguir com a execução.

execução do comando

A execução do comando foi resolvida com um `if` que trata os diferentes tipos de comando, além de lidar com o possível `fork`.

Do código em si, o mais distante do visto em aula foi o gerenciamento de erros, ou melhor, a entrega da mensagem devolvida pelo processo filho ao usuário do shell.

tratamento do retorno do comando

Algo a ser resolvido caso a caso.

Na grande maioria das chamadas implementadas, basta checar o retorno das chamadas do sistema.

No entanto, quando um programa externo era executado em um `fork`, envolvia esperar o resultado com `waitpid`.

arquitetura do simulador

Como explicitado no enunciado, foi implementado uma thread por processo a ser simulado.

Assim, o centro de toda a simulação são as `tasks`, estruturas que representam os processos.

Guardamos estas estruturas em espaço global através do projeto, para poder fazer acesso e gerenciamento delas de diversos contextos.

papel e implementação da `task`

`task` foi a abstração criada para os processos a serem simulados.

Além disso, é a conexão entre cada uma das threads com o processo que ela representa.

Cada thread roda independente e acessa apenas a informação pertinente a si.

Cada thread fica responsável em garantir a consistência do modelo de processo simulado, tanto atualizando as informações sobre o tempo de execução quanto terminando a thread quando necessário.

Esta parte do código, que é o que de fato é paralelizado no nosso projeto, ficou por conta da função `process_runner`, que é passada como argumento para o `pthread_create`.

utilização do `pthread_mutex`

Como consequência da independência das informações de cada processo em `tasks` distintas, o uso de mutex no projeto se tornou bem menor.

De fato, o único caso no qual um mutex se fez necessário foi para modificar as variáveis que indicavam se havia ou não alguma `task` rodando, para possibilitar nosso modelo de simulador com uma única CPU.

arquitetura geral do simulador

Fixado o significado de uma `task` no projeto, ficou fácil implementar o restante.

A função `main` nada mais faz do que ler a entrada e criar os `task_obj` necessários.

Dali para frente, todo o trabalho é gerenciado pelas funções que implementam o algoritmo de escalonamento.

algoritmos de escalonamento

Apesar dos três algoritmos serem distintos, há na implementação destes diversas semelhanças.

A principal é que a mesma função que executa os algoritmos de escalonamento também processam e adicionam as tarefas quando necessário.

Isso é feito percorrendo um vetor de `task_obj`, sob a hipótese de que os tempos de chegada dos processos estão ordenados.

Encapsulamos numa função `_assign` a parte de escolher o próximo processo a ser executado, sempre que necessário.

first-come first-served

Sem grandes surpresas, o algoritmo mais simples teve a implementação mais simples.

Como as `tasks` já estavam num vetor ordenado pelo tempo de chegada, criado pela `main`, o trabalho já começou praticamente feito.

Para implementar a fila foi apenas necessário manter os dois ponteiros, indicando o começo e fim da fila.

shortest remaining time next

Aqui grande parte da sujeira foi resolvida com a implementação de uma heap, responsável por manter o processo com *shortest remaining time* acessível em tempo constante.

Como neste modelo processos podem voltar para a fila, foi necessário uma checagem extra para ver se o processo tinha ou não uma thread.

preempção

O diferencial do *shortest remaining time next* no nosso EP é por ser o único a de fato depender de preempção.

Como o acesso ao trabalho rodando é simples, a checagem da necessidade de preempção se tornava simples.

Além disso, quando necessário, a implementação se reduz a chamar `task_stop` no processo rodando, e deixar a `heap` garantir que o processo correto será executado na sequência.

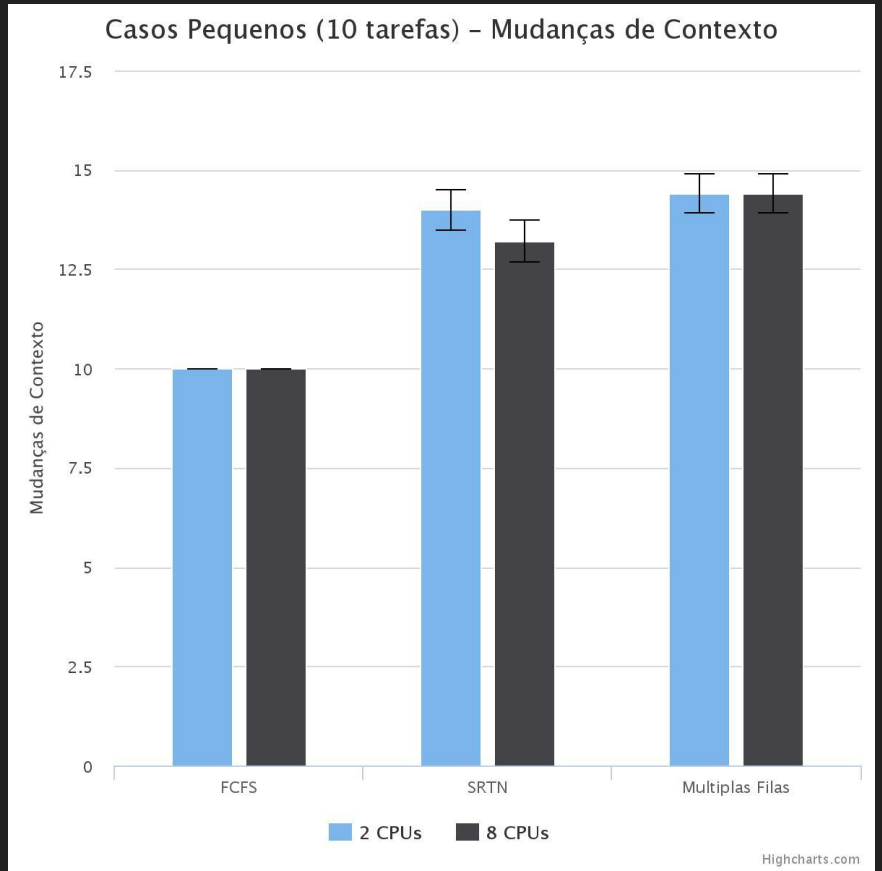
escalonamento com múltiplas filas

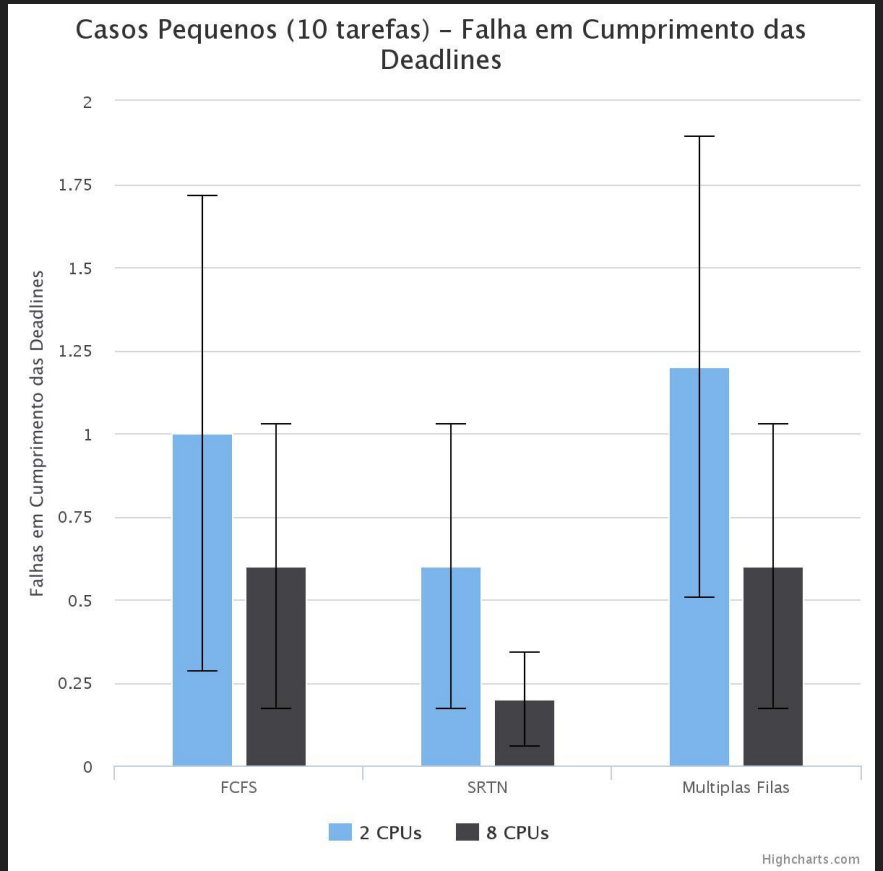
Aqui tomamos a decisão de implementar uma abstração para as múltiplas filas, ao invés de aproveitar o vetor de `task_obj`.

Como no *shortest remaining time*, o relevante da implementação está na estrutura de dados. Aqui, uma fila que guardava o quanto de *quantum* cada processo tinha direito na próxima execução.

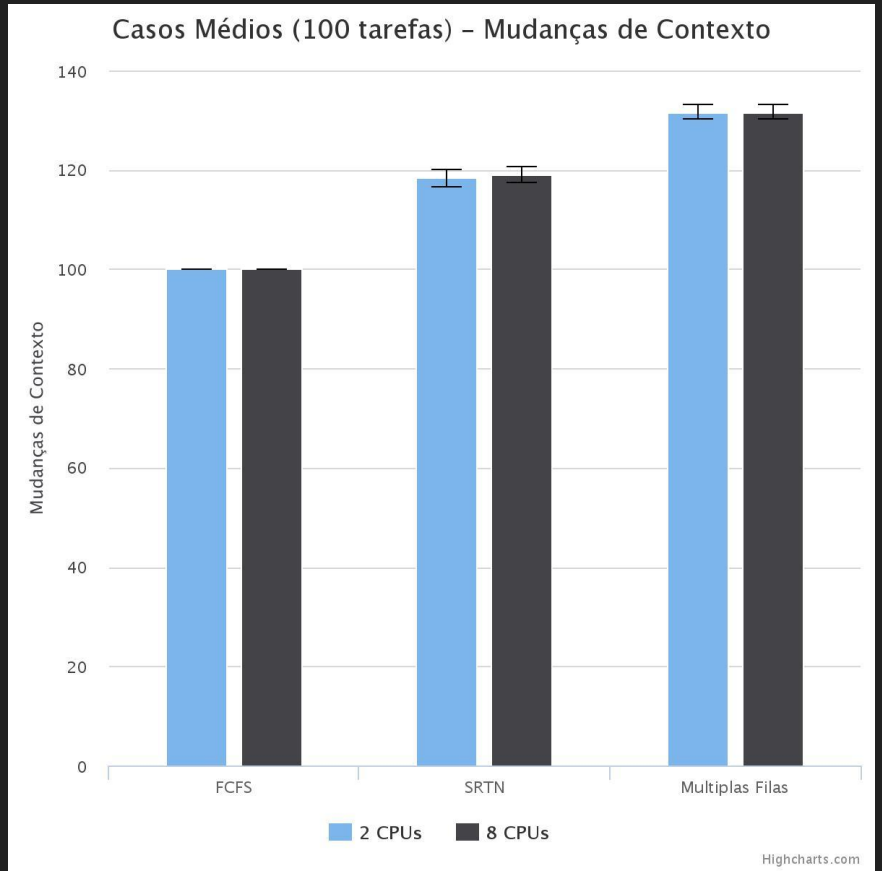
Isto foi feito com duas filas paralelas, uma de ponteiros para as `tasks`, e outra com as “prioridades” em si.

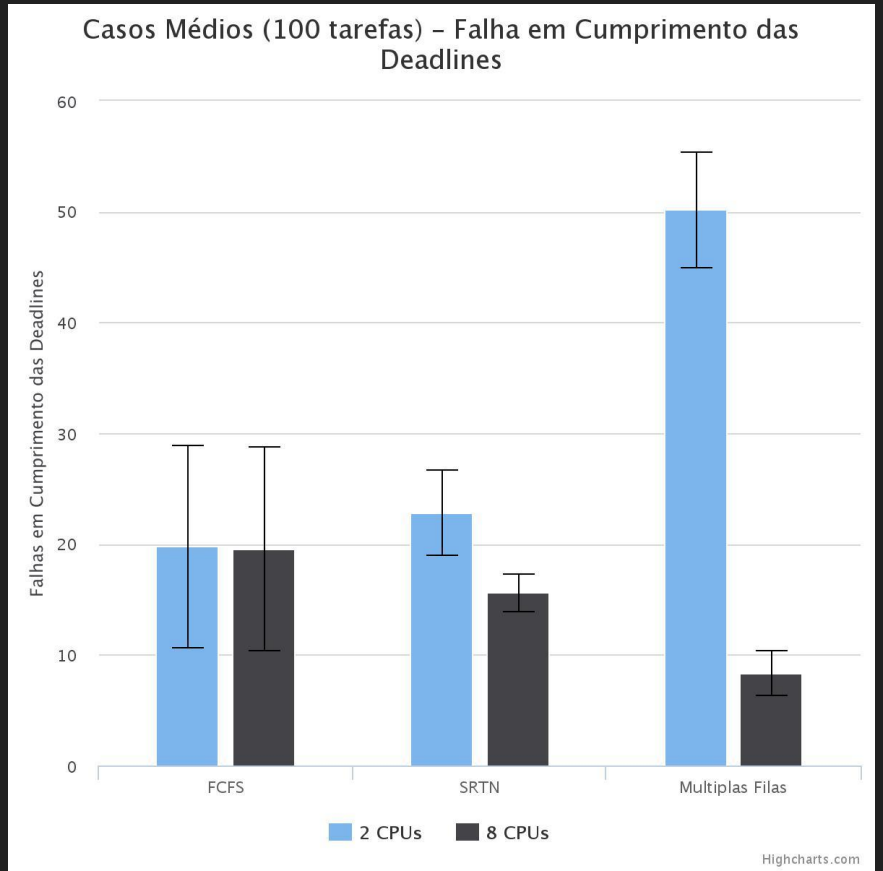
testes - casos pequenos



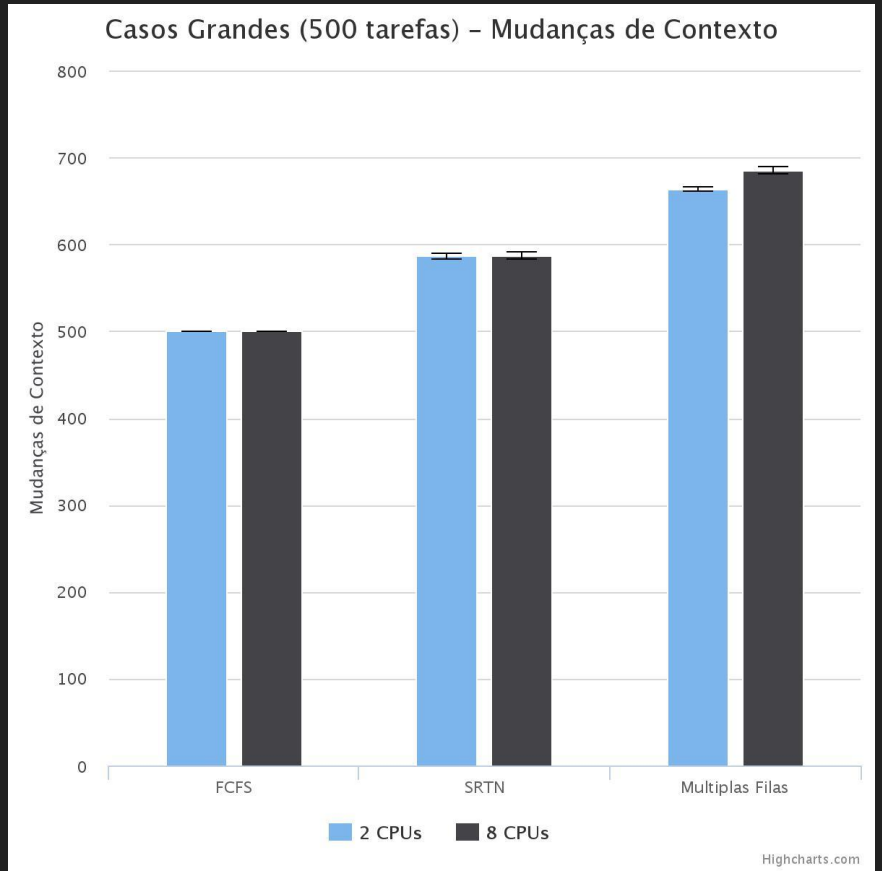


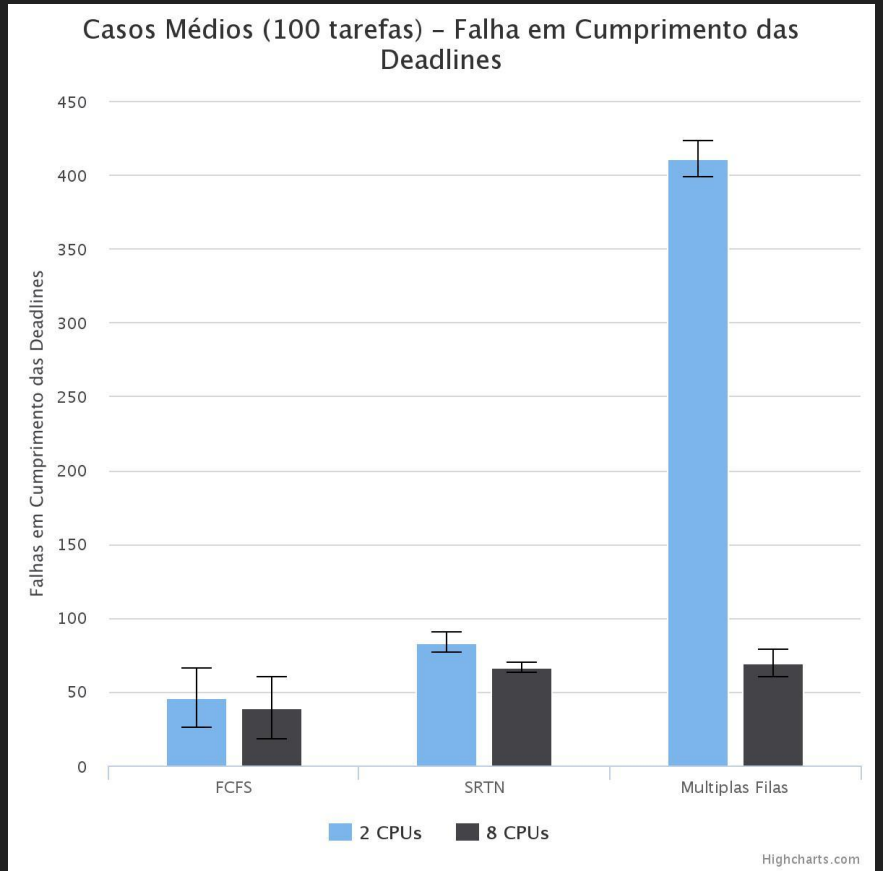
testes - casos médios





testes - casos grandes





dados dos testes

Os testes foram realizados em uma máquina com duas CPUs e em outra máquina com oito CPUs.

Os valores mensurados são os pedidos no enunciado: quantidade de mudanças de contexto e falhas no comprimento de deadlines.

Para cada um desses dados, há uma lista com o intervalo de confiança da média.

Estes dados estão organizados por escalonador, máquina e conjunto de testes, e aparecem nos próximos slides.

quantidade de mudanças de contexto (2 CPUs)

testes pequenos

- ➔ fcfs entre 10.000000 e 10.000000
- ➔ srtn entre 13.493930 e 14.506070
- ➔ Múltiplas filas entre 13.914595 e 14.885405

testes médios

- ➔ fcfs entre 100.000000 e 100.000000
- ➔ srtn entre 116.623702 e 120.176298
- ➔ Múltiplas filas entre 130.157923 e 133.042077

testes grandes

- ➔ fcfs entre 500.000000 e 500.000000
- ➔ srtn entre 583.131243 e 589.668757
- ➔ Múltiplas filas entre 660.531874 e 666.268126

quantidade de mudanças de contexto (8 CPUs)

testes pequenos

- ➔ fcfs entre 10.000000 e 10.000000
- ➔ srtn entre 12.674077 e 13.725923
- ➔ Múltiplas filas entre 13.914595 e 14.885405

testes médios

- ➔ fcfs entre 100.000000 e 100.000000
- ➔ srtn entre 117.448419 e 120.551581
- ➔ Múltiplas filas entre 130.157923 e 133.042077

testes grandes

- ➔ fcfs entre entre 500.000000 e 500.000000
- ➔ srtn entre 582.181714 e 591.418286
- ➔ Múltiplas filas entre 680.466691 e 689.933309

quantidade deadlines não respeitados (2 CPUs)

testes pequenos

- ➔ fcfs entre 0.284309 e 1.715691
- ➔ srtn entre 0.170586 e 1.029414
- ➔ Múltiplas filas entre 0.506112 e 1.893888

testes médios

- ➔ fcfs entre 10.658748 e 28.941252
- ➔ srtn entre 18.963203 e 26.636797
- ➔ Múltiplas filas entre 44.997563 e 55.402437

testes grandes

- ➔ fcfs entre entre 26.202383 e 65.797617
- ➔ srtn entre 77.022650 e 90.177350
- ➔ Múltiplas filas entre 398.466212 e 423.533788

quantidade deadlines não respeitados (8 CPUs)

testes pequenos

- ➔ fcfs entre 0.170586 e 1.029414
- ➔ srtn entre 0.056862 e 0.343138
- ➔ Múltiplas filas entre 0.170586 e 1.029414

testes médios

- ➔ fcfs entre 10.405114 e 28.794886
- ➔ srtn entre 13.927671 e 17.272329
- ➔ Múltiplas filas entre 6.368144 e 10.431856

testes grandes

- ➔ fcfs entre 18.542183 e 59.857817
- ➔ srtn entre 62.787123 e 69.612877
- ➔ Múltiplas filas entre 60.047548 e 78.352452