Ep1 de Sistemas Operacionais

Alunos: Ígor Barberino e Luis Vitor Zerkowski
NUSP: 11221689 e 9837201
Disciplina: Sistemas operacionais (MAC-0422)
Professor: Daniel Batista

Introdução

Este documento relata, de maneira breve, todo o processo de trabalho envolvido na construção do exercício programa desenvolvido pela dupla. Aqui serão descritos os mais diversos detalhes de implementação do código, bem como explicadas todas as escolhas de projeto e os resultados obtidos nas experimentações. Este relatório, portanto, subdivide-se em dois principais segmentos: o projeto, envolvendo código e escolhas de implementação, e o laboratório, mostrando experimentos realizados e o que fora alcançado.

Parte I - O Projeto

BCC Shell (bccsh)

Nesta primeira etapa do projeto, nos foi pedido o desenvolvimento de um shell que, através do uso direto de chamadas de sistema (*syscalls*) em C, implementaria alguns poucos comandos d'um típico *shell* do Linux. Além dos tais comandos, naturalmente foi necessário que fizéssemos alguma interface com o usuário para a devida utilização de nosso próprio *shell*. Nos próximos parágrafos, pois, essas duas etapas de codificação serão brevemente descritas.

Para a implementação da interface, seguimos o padrão sugerido pelo enunciado. Utilizamos, desse modo, as funções *getenv("USER")* e *getcwd(string, sizeof(string))* para imprimir em todo início de iteração com o usuário uma *string* nos moldes "*{lui@/home/lui/Documents/IME/2020.2/SO/SO_rep/EP_1}*" - exemplo real tirado do computador de um dos integrantes do grupo. Falando agora sobre as iterações de escrita pelo usuário e leitura pelo programa, utilizamos a função *readline("")* para obter a linha digitada no *prompt* e a função *add_history(string)* para guardar essa linha no histórico e permitir seu acesso através das seta para cima no teclado. Após a leitura da entrada do usuário, construímos um *parser* com a função *strsep(&string, " ")* para *tokenizar* a linha digitada e poder executar os comandos solicitados.

Já para os comandos, fizemos algumas comparações com os *tokens* obtidos na etapa anterior para checar qual instrução fora passada pelo usuário e, para qualquer uma delas, criamos um processo filho com a função *fork()* e, através da função *waitpid(-1, NULL, 0)* fazemos com o que o processo pai - o próprio *shell* - espere a execução do comando de seu filho. Segue agora a lista de todas as funções utilizadas para cada um dos comandos:

1. *mkdir diretorio:*
    a. *mkdir(diretorio, 0777)*

2. *kill -9 pid:*
    a. *kill(pid, abs(atoi(-9)));*

3. *ln -s arquivo link:*
    a. *symlink(arquivo, link);*

1. */usr/bin/du -hs .char* aux [] = {"/usr/bin/du", "-hs", ".", NULL}*
    a. *execve(“/usr/bin/du”, aux, NULL)*

2. */usr/bin/traceroute www.google.com.br*
    a. *char* aux [] = {"/usr/bin/traceroute", "www.google.com.br”, NULL}*
    b. *execve(“/usr/bin/traceroute”, aux, NULL)*
3. */ep1 #escalonador arquivo_trace arquivo_simulacao (d)*
    a. *char* aux [] = {"./ep1", #escalonador, arquivo_trace, arquivo_simulacao, NULL}*
    b. *(char* aux [] = {"./ep1", #escalonador, arquivo_trace, arquivo_simulacao, d, NULL})*
    c. *execve(“./ep1”, aux, NULL)*

Finalizada a execução de um comando, o *shell* volta ao início de todo esse processo, imprimindo novamente a linha padrão do *prompt* e dando início a um novo ciclo de iterações.

O Escalonador (ep1)

Nesta segunda etapa, devíamos implementar um simulador de processos, programa esse que deveria escalonar processos de um arquivo de simulação - *trace.txt* - através de um método escolhido entre três pelo usuário: *First Come First Served (FCFS), Shortest Remaining Time Next (SRTN), Round Robin (RR)*. Nos próximos parágrafos, pois, a implementação de cada um desses escalonadores será dissecada para melhor entendimento do código por trás de todos eles.

Antes de mergulhar no estudo de cada um dos escalonadores individualmente, é possível estudar os elementos comuns a todos os códigos. Os recursos compartilhados não são tantos e, prendendo-se um pouco mais aos de maior destaque, podemos resumi-los a: contadores, estrutura de leitura dos processos e *threads*. Segue uma breve explicação de cada um desses elementos:

1. Contadores: De maneira bastante natural, os contadores aparecem como recurso comum a todos os escalonadores por sua importância na etapa de medições e testes. São variáveis como *tempo* - conta o tempo real de simulação desde a entrada do primeiro processo - ou *contador_contexto* - conta a quantidade de mudanças de contexto de um certo escalonador para um certo arquivo de *trace* - que aparecem nesta categoria e, de maneira bastante autoexplicativa, ajudam a garantir funcionamento correto do programa. Além dessas duas variáveis, temos também recursos como *cumpre_deadline* - conta a quantidade de processos terminados cuja deadline foi cumprida pelo escalonador - que ajudam na análise posterior da eficiência de cada um dos escalonadores, bem como na confecção dos gráficos do laboratório.

2. Leitura de processos: Apesar de suas múltiplas implementações, uma para cada escalonador, a lógica por trás da leitura dos processos é bastante simples e comum a todos esses escalonadores. O arquivo de *trace* é lido linha por linha e cada um dos processos é adicionado a uma grande fila - caso não seja uma linha vazia como tipicamente são as últimas linhas de um arquivo qualquer - que será posteriormente utilizada para simular a chegada de cada um desses processos no escalonador propriamente dito.

1. *Threads*: De maneira bastante análoga ao caso anterior, as *threads* também apresentam múltiplas implementações por uma questão de detalhes majoritariamente relacionados a impressões na *stderr* - solicitação do parâmetro opcional *d*. As *threads* criadas por todos os escalonadores, portanto, seguem o seguinte ciclo:

- Inicializa-se uma variável auxiliar que irá receber a unidade básica de cada um dos escalonadores - processos para os escalonadores *FCFS* e *SRTN* e nós de uma lista ligada para *RR*.

    A *thread* entra num laço que roda até o *dt* do processo tornar-se zero ou até o escalonador responsável impor uma mudança.

- Através da função *pthread_mutex_lock(&vetor_mutex[id_processo])*, o controle da *thread* a ser processada é feito. É neste segmento de código que o escalonador controla o processamento das *threads* - relacionadas de maneira única com os processos -, liberando para rodar apenas aquela cujo *id_processo* fora escolhido através de um dos métodos de escalonamento - *SRTN* ou *RR*. Neste segmento, faz-se importante o destaque ao funcionamento dos semáforos. O vetor de semáforos do tipo *mutex*, portanto, é utilizado aqui para garantir alternância entre uma unidade de tempo na *thread* selecionada e uma breve iteração no escalonador, uma vez que alguns atributos do processo em execução passam a ser críticos, e também para assegurar que apenas a *thread* selecionada pelo escalonador irá ser processada, já que a própria *thread* é responsável por “fechar” seu semáforo. Nota-se, ainda, que esse segmento de código não é utilizado para o método *FCFS* por simples falta de necessidade, uma vez que o processo selecionado por esse escalonador será processado ininterruptamente até o final.

- A *thread* agora apenas simula a execução do processo, decrementando uma unidade de tempo no atributo *dt* desse tal processo, utilizando a função *sleep(1)* para garantir a unidade de tempos em segundos exigida no enunciado, incrementando em uma unidade o tempo geral de simulação e, por fim, rodando um laço contador que fará milhões de somas para garantir a fácil verificação do uso de algum núcleo da máquina.
- Como última etapa, a *thread* libera o escalonador para voltar a rodar através da função *pthread_mutex_unlock(&mutex_escalonador)*. Nota-se, ainda, que a *thread* não irá prosseguir normalmente na próxima iteração do laço porque ela mesmo “fechou” seu semáforo. Além disso, percebe-se, por fim, que essa etapa novamente não consta nas *threads* do escalonador *FCFS* pelos mesmos motivos de falta de necessidade do uso de semáforos.

FCFS

Para esse escalonador, a ideia do projeto foi bastante simples e pode facilmente ser resumida a uma fila. Implementamos, portanto, uma fila num vetor redimensionável que armazena todos os processos lidos de *trace.txt* em ordem de chegada. Com essa fila em mãos, o procedimento segue um ciclo simples e bastante claro:

a. O próximo da fila será removido da mesma e terá uma *thread* criada para si.
b. O escalonador utiliza a função *pthread_join(thread, NULL)* para esperar a *thread* criada ser processada antes de continuar seu trabalho.
c. A *thread* criada será processada até que o processo seja finalizado - tenha sido processada por *dt* segundos.
d. O escalonador, então, volta a trabalhar e escreve os devidos dados do processo recém finalizado no arquivo de simulação. Em seguida, checa três fenômenos antes de voltar para o início do ciclo: se a fila não está vazia, se o processo que estava sendo rodado anteriormente já acabou de fato, e se o próximo da fila já está pronto para entrar - *t0* menor ou igual ao tempo de simulação. Se a fila estiver vazia, o laço de iterações é quebrado e o número de mudanças de contexto é escrito no arquivo de simulação - aqui será igual ao número de processos, uma vez que as únicas mudanças de contexto são as entradas das novas *threads*. Se a fila não estiver vazia e as outras condições também forem devidamente checadas, voltamos ao início do ciclo.

STRN

O escalonador, nesse caso, foi implementado principalmente através de um *heap de mínimo* (*min heap)* organizado a partir do dt de cada processo. Esse *heap* foi implementado num vetor onde a posição 0 é vazia e a posição 1 contém o processo com menor dt entre todos os que devem ser executados. Depois de criar uma *thread* para cada processo e trancá-las com semáforos específicos de cada processo, o escalonamento, então, segue da seguinte maneira:

a. Se o *min heap* está vazio, o escalonador desbloqueia seu semáforo que virá em seguida.
b. O semáforo do escalonador se tranca.
c. Se o *heap* está vazio, o escalonador espera um segundo para que novos processos cheguem.
d. Se a *thread* que está sendo executada acabar, ela é removida do *heap* e dá lugar para execução do próximo processo com menor dt.
e. Se processos novos chegaram, o escalonador insere esses processos no *heap*, de maneira que suas posições dependerão novamente do valor de seus dt.
f. Caso o processo que ocupa a posição de raiz do *min heap* tenha mudado, é contabilizado uma mudança de contexto pelo escalonador.
g. Se não há mais nenhum processo para ser executado, o loop de escalonagem é quebrado.
h. Se há processos no *heap*, o escalonador destranca o semáforo do processo da primeira posição.
i. Nesse momento, o processo com menor *dt* começa a ser executado. Ele tranca o seu semáforo, faz a sua devida operação e destranca o semáforo do escalonador.
j. A execução do loop do escalonador recomeça

Round-Robin

O Round-Robin foi implementado utilizando como base uma lista ligada circular. O nó *lista_first* contém o primeiro processo a ser executado e o nó *lista_last* contém o último. Novamente é criada uma thread para cada processo e o semáforo de todas elas é trancado, começando o escalonamento:

a. Se a lista está vazia, o semáforo do escalonador é destrancado.
b. O escalonador tranca seu semáforo.
c. Se a lista está vazia, o escalonador dorme por 1 segundo, esperando para que novos processos cheguem.
d. Se a lista não está vazia e o processo da primeira posição terminou de executar, esse processo é removido da lista e o processo que estava na segunda posição vai para a primeira.
e. Se a lista não está vazia e o processo da primeira posição já executou por um tempo igual ou maior ao quantum, ele é reposicionado para o final da lista e o processo que estava na segunda posição passa para a primeira.
f. O escalonador retira da fila todos os processos que tem tempo de chegada menor ou igual ao tempo atual, ou seja, os processos que já chegaram. Esses processos são, então, inseridos na lista na última posição.
g. Se a fila de processos futuros está vazia e a lista está vazia, não há mais processos a serem executados. Assim atribui-se *NULL* a *lista_first*, sinalizando para que o loop termine.
h. Se a listas não está vazia, o semáforo do processo do nó apontado por *lista_first* é desbloqueado. Se além disso o *lista_first* passou a apontar para outro nó durante esse *loop* do escalonador, é considerado que houve uma mudança de contexto.
i. É executada então a thread do processo da primeira posição da lista. Ele tranca o seu próprio semáforo, faz suas operações e destranca o semáforo do escalonador.
j. A execução do *loop* do escalonador recomeça.

Observações finais da etapa

Para finalizar esta etapa do guia, faz-se necessária a observação sobre a contagem de mudanças de contexto. Como não fora explicitado o método de tal contagem, consideramos mudança de contexto tanto as preempções quanto o início de processamento de uma *thread* quando não existia nenhuma outra em execução. Todo escalonador, portanto, faz pelo menos tantas mudanças de contexto quanto existem processos no arquivo de simulação.

Parte II - O Laboratório

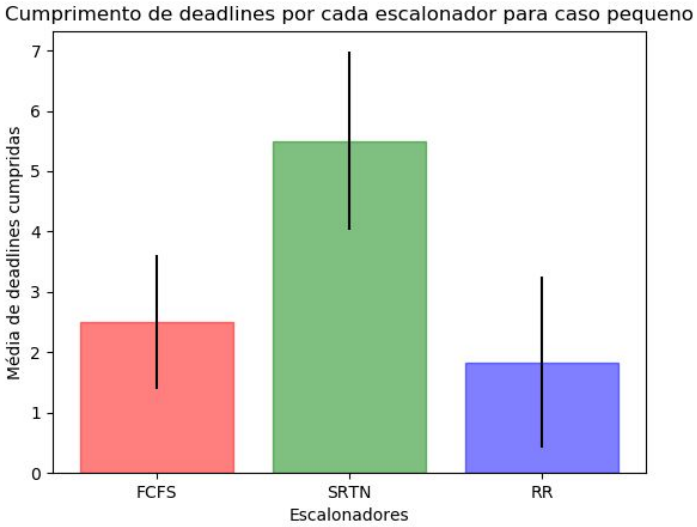

Gráfico de barras ilustrando a média de processos finalizados cujas *deadlines* foram cumpridas para cada um dos escalonadores. Nesse experimento, foram utilizados arquivos de *trace* com 10 processos.

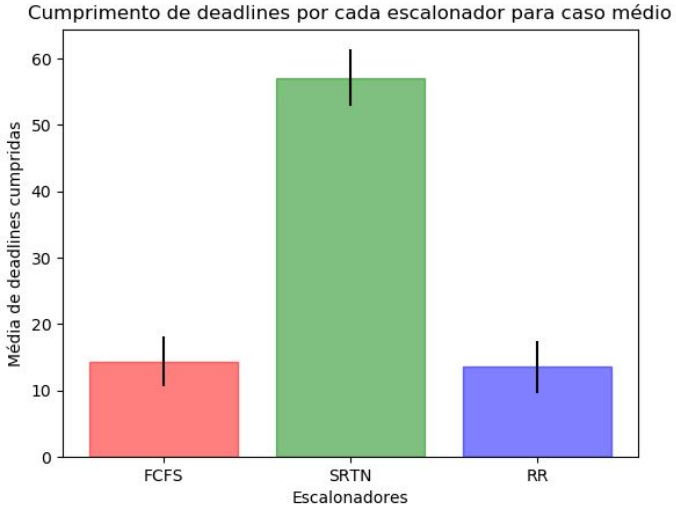

Gráfico de barras ilustrando a média de processos finalizados cujas *deadlines* foram cumpridas para cada um dos escalonadores. Nesse experimento, foram utilizados arquivos de *trace* com 100 processos.

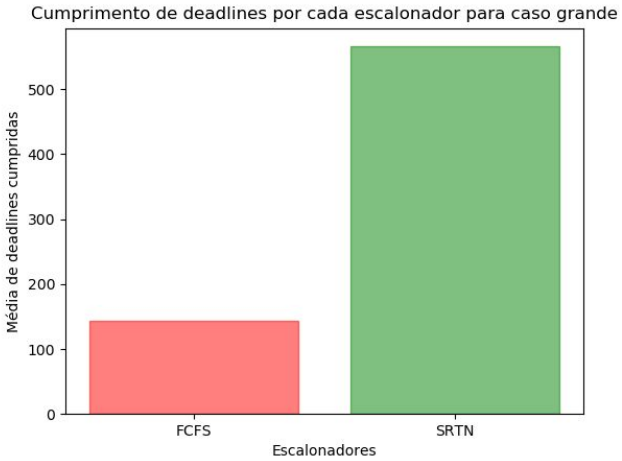

Gráfico de barras ilustrando a média de processos finalizados cujas *deadlines* foram cumpridas dois dos escalonadores. Nesse experimento, foram utilizados arquivos de *trace* com 1000 processos.

Intervalo de confiança do cumprimento da deadline

First-Come First-Served:

10 processos - {1,81 , 3,19}

100 processos - {13,63 , 15,10}

Shortest Remaining Time Next:

10 processos - {4,58 , 6,42}

100 processos - {56,26 , 57,94}

Round-Robin:

10 processos - {0,95 , 2,70}

100 processos - {12,79 , 14,35}

Em questão de amplitude intervalo de confiança do cumprimento das deadlines, não há grandes diferenças observadas entre os escalonadores.

Observações da deadline

Observando os gráficos de média de cumprimento de deadlines, é possível enxergar uma similaridade na eficiência do *First-Come First-Served* e do *Round-Robin* nesse quesito, já que nenhum dos dois foca nesse aspecto. O *FCFS* é apenas uma primeira implementação mais simples de um escalonador, não pensada para aplicações mais delicadas. Serve principalmente para aplicações mais sequenciais sem muita interação com usuários, podendo ser usado para supercomputadores por exemplo. Já o escalonador *RR*, é pensado para circunstâncias de muita interação com o usuário. Seu objetivo é tentar dividir o tempo de uso do processador proporcionalmente entre todos os processos a serem executados, sendo tipicamente utilizado nos computadores pessoais. Apesar das propostas bastante divergentes entre si, é possível notar em ambos a incompatibilidade com o preciosismo de terminar mais processos dentro de suas deadlines. O *SRTN*, por fim, apresenta dados muito melhores sobre cumprimento de deadlines. Já que sua implementação foca em executar os processo que estão mais próximos de acabar, o escalonador consequentemente acaba sendo muito melhor em respeitar as deadlines dos processos.

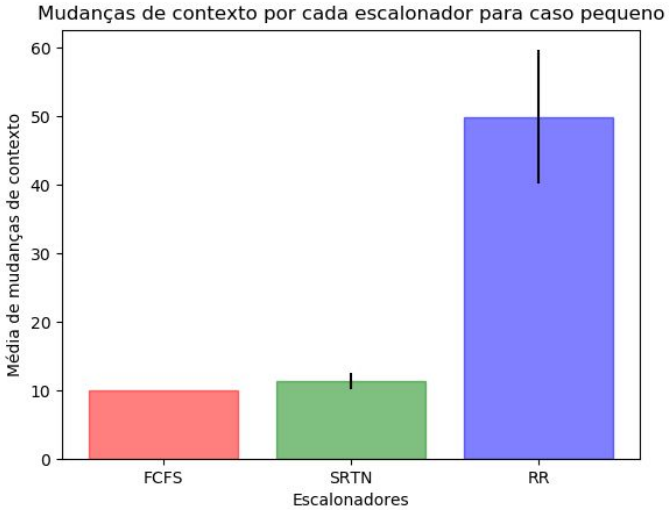

Gráfico de barras ilustrando a média mudanças de contexto para cada um dos escalonadores. Nesse experimento, foram utilizados arquivos de *trace* com 10 processos.

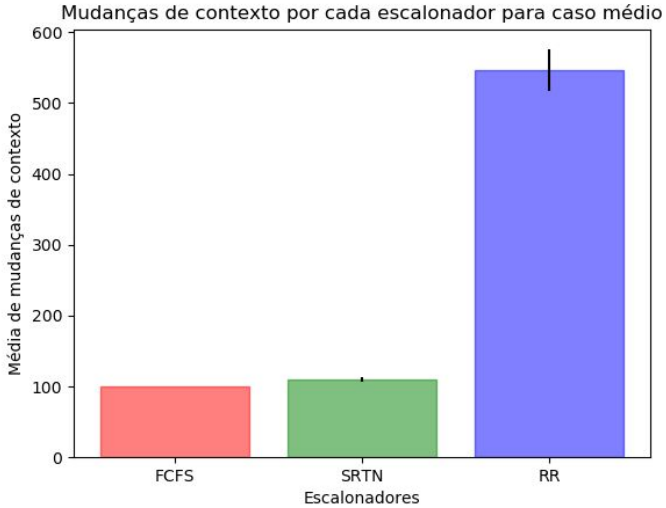

Gráfico de barras ilustrando a média mudanças de contexto para cada um dos escalonadores. Nesse experimento, foram utilizados arquivos de *trace* com 100 processos.

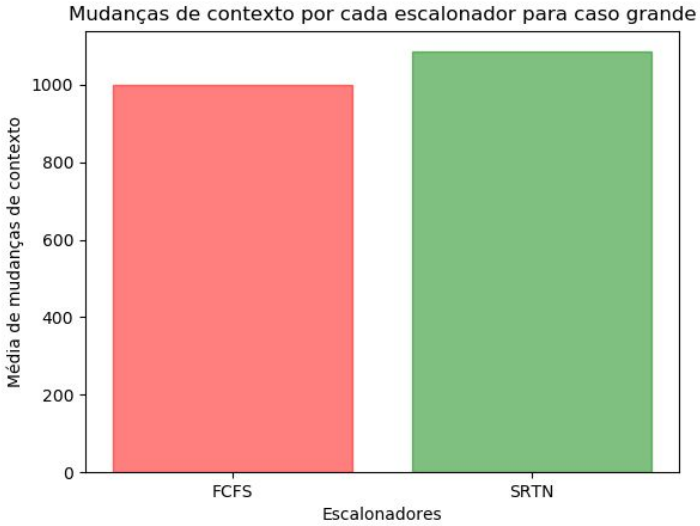

Gráfico de barras ilustrando a média mudanças de contexto para dois dos escalonadores. Nesse experimento, foram utilizados arquivos de *trace* com 1000 processos.

Intervalo de confiança para as mudanças de contexto

First-Come First-Served:

10 processos - {10, 10}

100 processos - {100, 100}

Shortest Remaining Time Next:

10 processos - {10,69 , 12,11}

100 processos - {109,47 , 110,67}

Round-Robin:

10 processos - {43,88 , 55,98}

100 processos - {540,85 , 552,21}

É possível observar que o intervalo de confiança do Round-Robin tem amplitude consideravelmente maior do que o dos outros escalonadores. Isso aconteceu devido a seu grande desvio padrão, gerado porque as mudanças de contexto no round-robin dependem muito do tempo de execução de cada um de seus processos, gerando a difícil previsibilidade da quantidade de vezes que essas mudanças acontecerão.

Observações finais da etapa

Escalabilidade

Para além desses testes, foi realizado um teste com cinco mil processos em todos os três escalonadores. O processamento demorou cerca de sete horas e meia para cada escalonador. Os escalonadores *FCFS* e *SRTN* não tiveram nenhum problema e mostraram-se perfeitamente escaláveis. O terceiro escalonador, no entanto, apresentou dificuldades para escalar em ordem de grandeza e tipicamente rodava apenas simulações de cerca de setecentos e cinquenta processos antes de chegar a um *segmentation fault*. Por esse motivo, inclusive, os gráficos para simulações com mil processos não contém uma barra para o *RR*.

Intervalo de confiança

Por fim, vale ressaltar que não foram feitos os cálculos para intervalo de confiança para as simulações grandes - de cerca de mil processos - por falta de múltiplos experimentos. Devido ao longo tempo de execução do programa para este caso - por volta de uma hora e meia para cada simulação - rodamos os códigos dos dois escalonadores, *FCFS e SRTN*, apenas uma vez.

Porque não foram feitos gráficos específicos para computadores com números de cores diferentes

As duas pessoas da dupla só tinham computadores com 4 cores para realizar os testes. Por essa razão, os testes foram feitos usando 3 máquinas diferentes, mas todas com 4 núcleos. Devido a isso, não foram feitas diferenciações nos gráficos entre as máquinas.