Exercício Programa 1

Escalonador de processos

Lucas Paiolla Forastiere e Marcos Siolin Martins

IME-USP

05 de outubro de 2020

Arquitetura do Shell

O shell segue a arquitetura sugerida em aula, tendo sido implementado apenas no arquivo `bccsh.c`.

Conta com um loop principal em que lê um comando por iteração e executa esse comando internamente por meio de chamadas de sistema ou realiza a invocação externa do binário informado até que um sinal `EOF` seja emitido pelo usuário (pressionar as teclas `CTRL+D`).

Arquitetura do Shell

Além disso algumas decisões de projetos foram tomadas, entre elas:

- A função `read_command(command, parameters)` recebe o comando digitado pelo usuário, usando a função `readline()`, e devolve o comando na variável `command` e os parâmetros na variável `parameters`. Por definição, `parameters[0] = command`;
- A função `readline()` aloca a memória necessária. Guardamos seu retorno no histórico usando `add_history()` e tratamos seu retorno substituindo espaços em branco pelo caractere '\0' e mudando os ponteiros de `parameters` para o começo de cada parâmetro na string.

Arquitetura do Shell

- Todos os arrays que não são alocados por funções externas são alocados estaticamente com valor máximo definido por diretivas `#define`:
  - `CUR_DIR_SIZE`: tamanho máximo do nome do diretório;
  - `PROMPT_SIZE`: tamanho máximo da string exibida no prompt;
  - `MAX_PARAMETERS`: quantidade máxima de parâmetros.
- Por fim, implementamos a chamada de sistema `mkdir` passando como parâmetro a constante `S_IRWXU` que dá ao usuário todas as permissões sobre aquele diretório.

Implementação dos Escalonadores

Inicialmente, todas as threads que eventualmente chegarão no sistema são carregadas do arquivo de entrada, criadas e ficam bloqueadas até que o escalonador lhes dê a permissão de rodar.

Para fazer esse gerenciamento das threads, existe um array de mutex (chamado `mutex`) em que cada mutex está associado a uma thread. Se o mutex está liberado, então a thread pode rodar. Caso contrário, a thread fica bloqueada. Usamos os mutex da biblioteca `pthread` para fazer esse gerenciamento, juntamente com as funções `pthread_lock` e `pthread_unlock`.

Implementação dos Escalonadores

A variável inteira chamada `semaforo` possui valor igual à thread que está em execução no momento (decidimos que apenas uma thread executaria por vez).

Essa variável é gerenciada pela função `setSemaforo(value)`, que recebe o valor da thread que executará e bloqueia a antiga thread para liberar a nova (caso o valor passado seja -1, então isso indica que nenhuma thread está em execução no momento).

Implementação dos Escalonadores

Decidimos também criar uma `struct` para os processos, para armazenar algumas informações de cada processo. Como o tempo em que ela terminou de executar e as propriedades informadas no arquivo de trace.

Esses processos ficam em um array chamado `processos` que possui tamanho máximo igual a `nmax`. Assumimos que o número máximo de processos é 1000, mas deixamos `nmax` como 1024 para ter uma folga.

Todas as variáveis e funções de uso amplo foram colocadas no arquivo `util.h` e cada escalonador foi implementado em um arquivo próprio.

Implementação dos Escalonadores

Por fim, o arquivo `ep1.c` possui a função `main` e a função `busy`, que é a função executada por cada uma das threads.

O consumo de CPU realizado por ela advém da função `sched_getcpu()`, que retorna a CPU atual em que a thread está executando. Nos nossos testes, esse uso foi de 100% do núcleo para os escalonadores `FCFS` e `SRTN`. No `round robin` a ocorrência massiva de preempções dificulta a visualização de qual CPU está sendo usada.

Implementação dos Escalonadores - Tempo

Sobre o tempo da simulação, o próprio escalonador controla o tempo passado através do uso da função `usleep(t)` que coloca o escalonador para "dormir" por (pelo menos) t microssegundos. Quando o escalonador "acorda" se passaram pelo menos t microssegundos e assumimos que exatamente t microssegundos se passaram (o erro cometido por `usleep` é pequeno de acordo com a documentação).

A variável global `cur_time` controla quantos segundos na simulação se passaram. Ela começa com valor igual ao t_0 do primeiro processo (avançamos a simulação para o ponto em que o primeiro processo chega).

Implementação dos Escalonadores - FCFS

No FCFS, aproveitamos a própria fila de processos carregada na entrada para simular a ordem dos processos.

Usamos uma variável `atual` para controlar o índice da thread que está executando no momento e vamos atualizando o tempo que ela ficou executando conforma a simulação avança.

Além disso, também mantemos uma variável `prox` para dizer qual é o próximo processo que vai chegar na simulação. Enquando o tempo atual da simulação é igual ao t_0 de `prox` indicamos a chegada dele e incrementamos a variável.

Implementação dos Escalonadores - SRTN

No SRTN, usamos um vetor `fila` que guarda os índices de cada thread.

A variável `prox` tem o mesmo papel que no FCFS, assim como `atual`.

A variável `ini` aponta para o começo da fila e, por definição, `fila[ini-1]` é o processo que está executando no momento.

A variável `fim` aponta para a última posição não ocupada da fila e, por definição, `fila[fim]` é sempre igual a -1.

Além disso, esse escalonador conta com a função `insere_na_fila`, que insere o processo que acabou de chegar no seu lugar apropriado na fila (ordenado pelo tempo restante de execução) e atualiza os valores de `ini` e `fim`.

Implementação dos Escalonadores - Round Robin

No Round Robin, a fila de processos carregada na entrada é reaproveitada para simular a ordem dos processos.

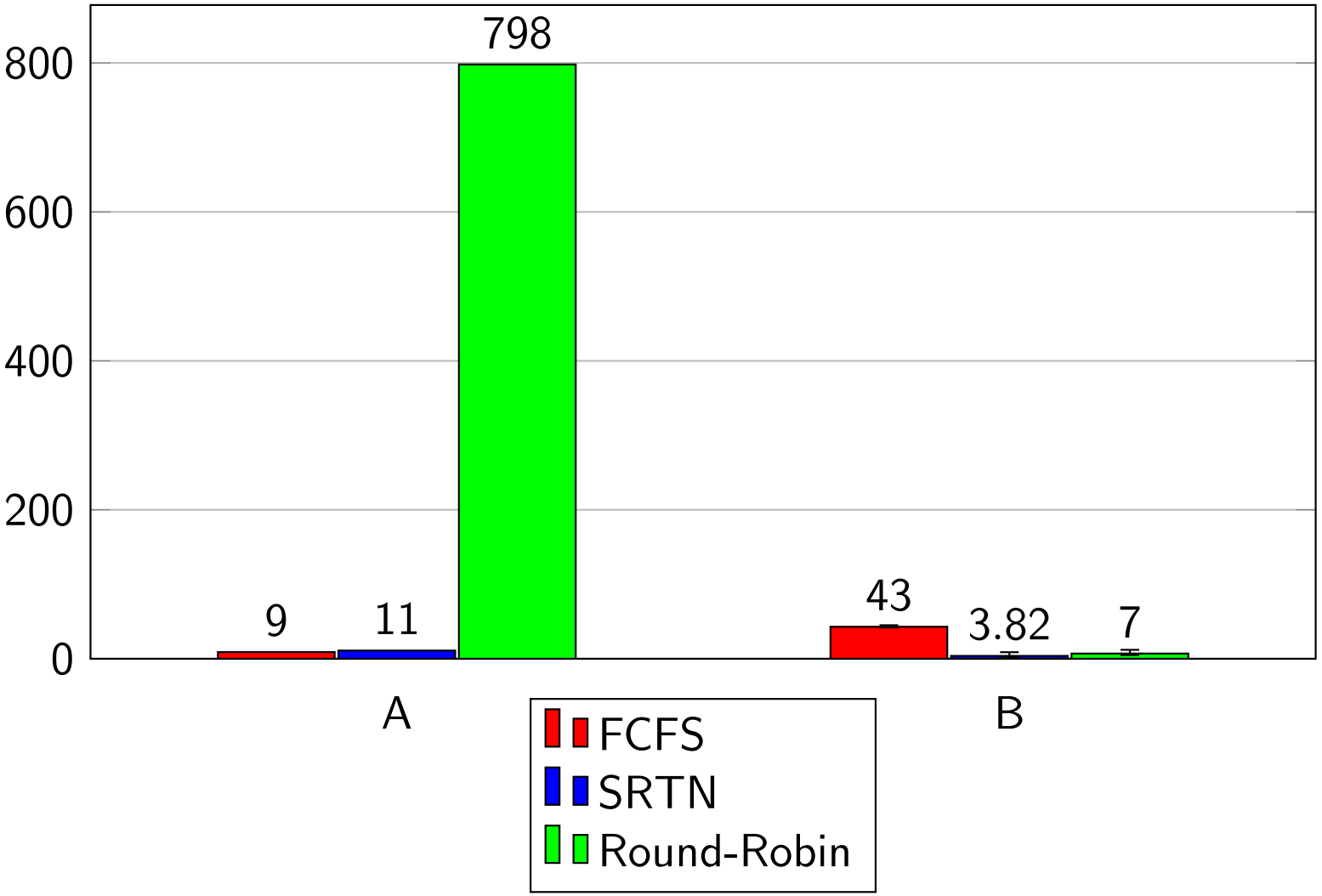
As variáveis `atual` e `prox` cumprem o mesmo papel que cumpriam nos outros dois escalonadores, enquanto a variável `tempo_dormindo` controla quantos microssegundos além do segundo atual já se passaram.

O `quantum` é definido em um `#define` e é dado em microssegundos. Testamos para `quantum = 0.05s`.

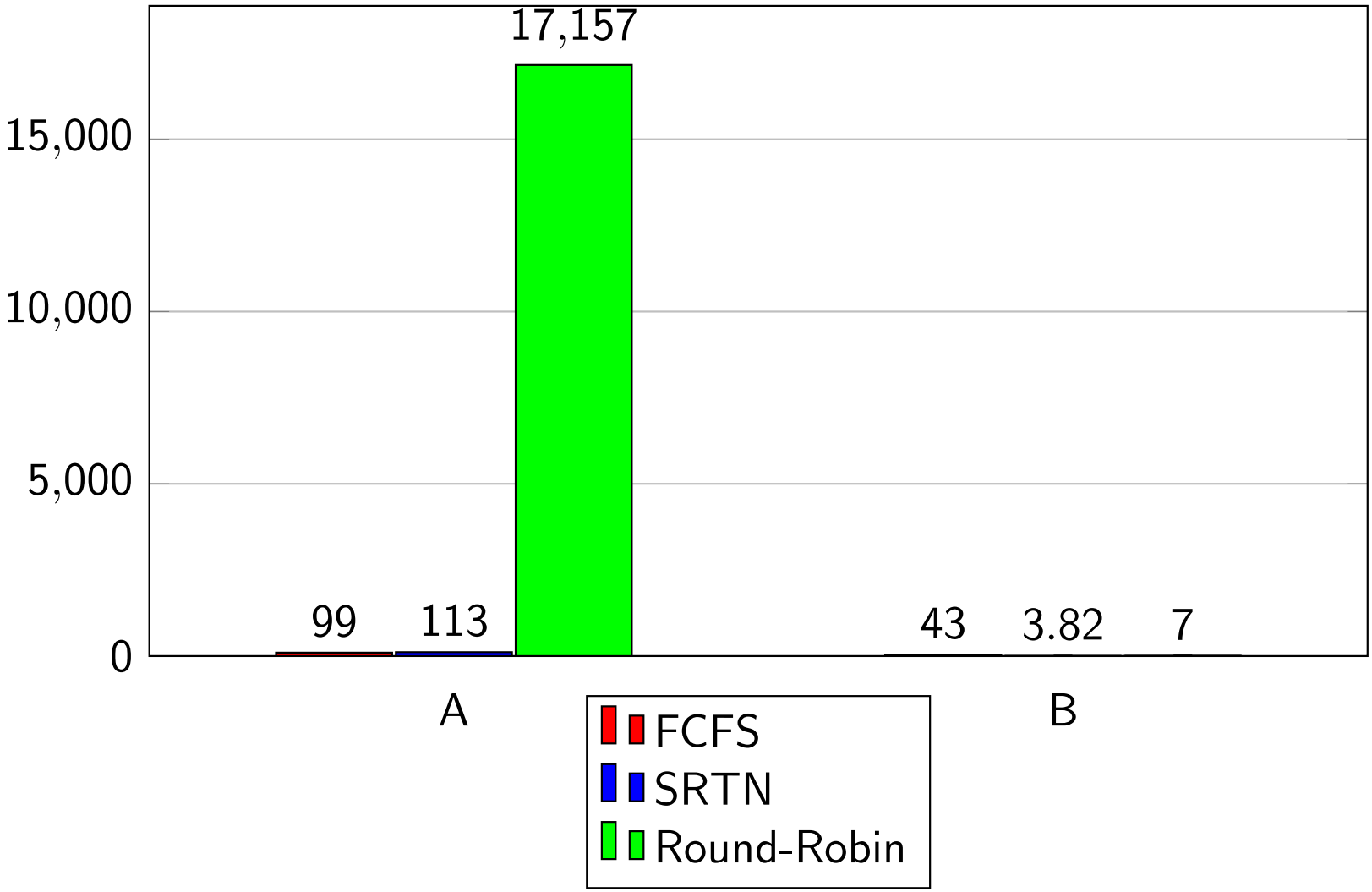
A variável `minimo` assegura que o escalonador não dormirá por `quantum` se o tempo para o processo terminar é menor que isso.

A variável `todos_terminaram` controla se todos os processos já encerraram sua execução.

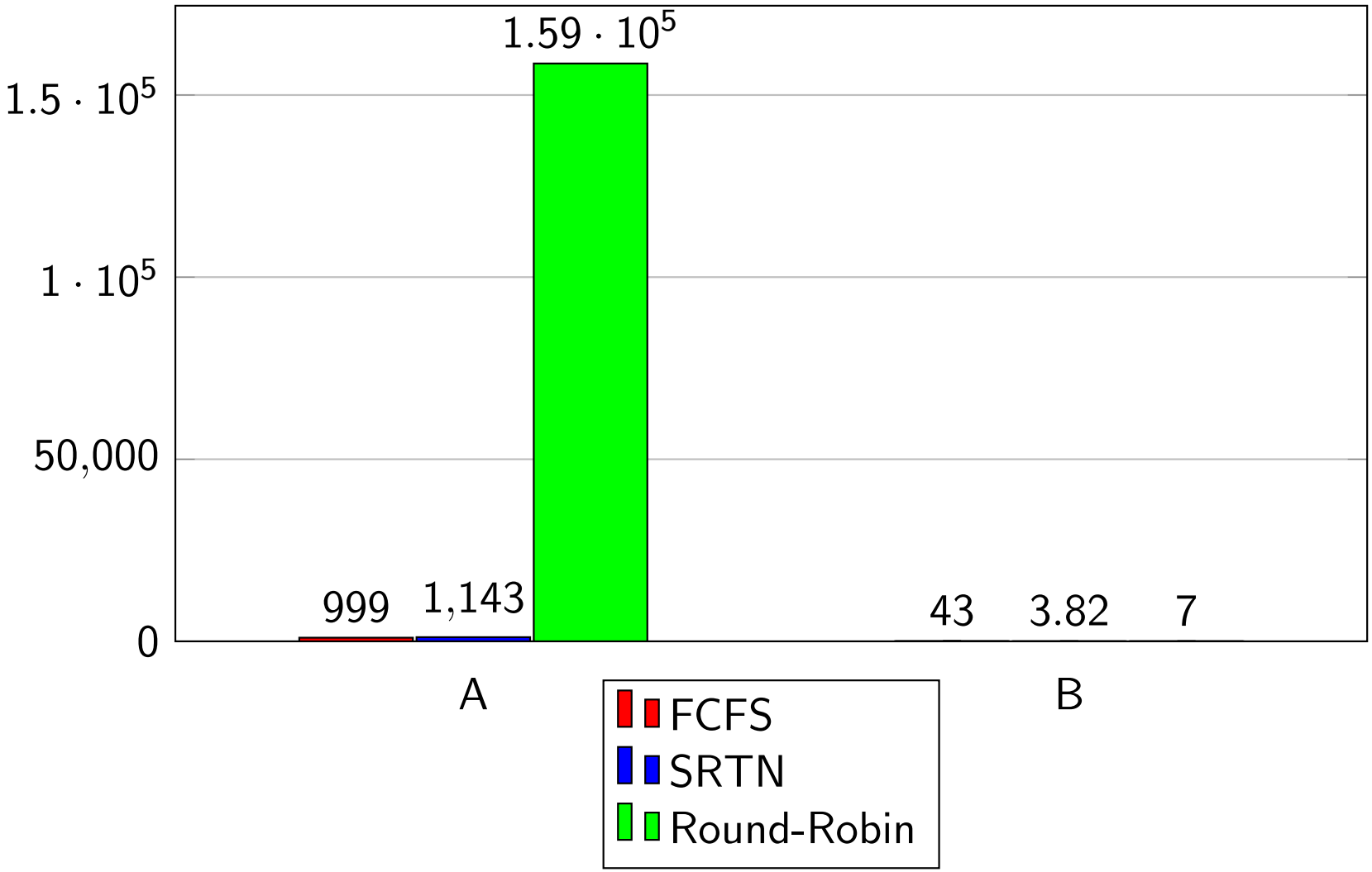
Arquivo de Trace: 10 processos - Mudanças de contexto



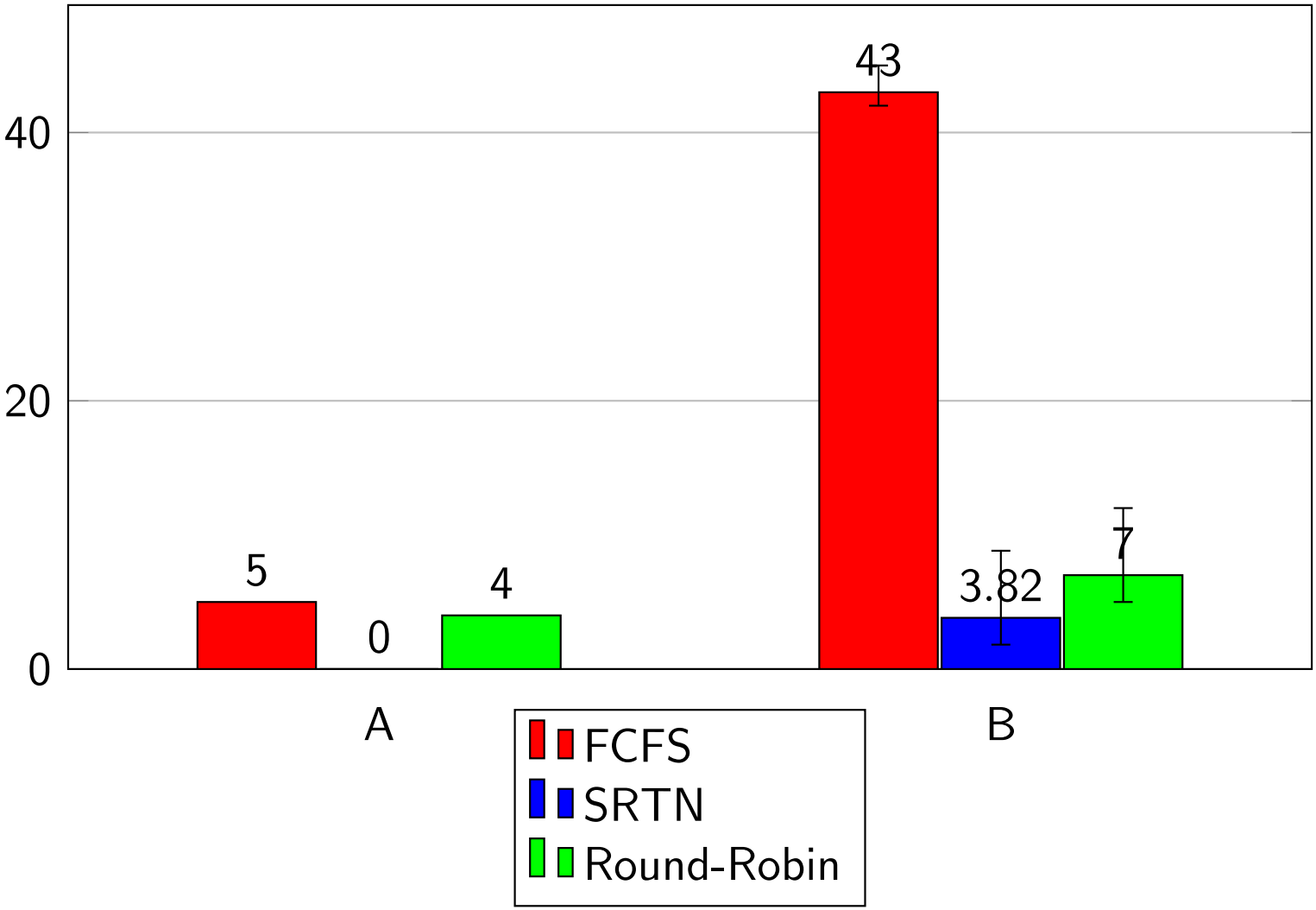
Arquivo de Trace: 100 processos - Mudanças de contexto



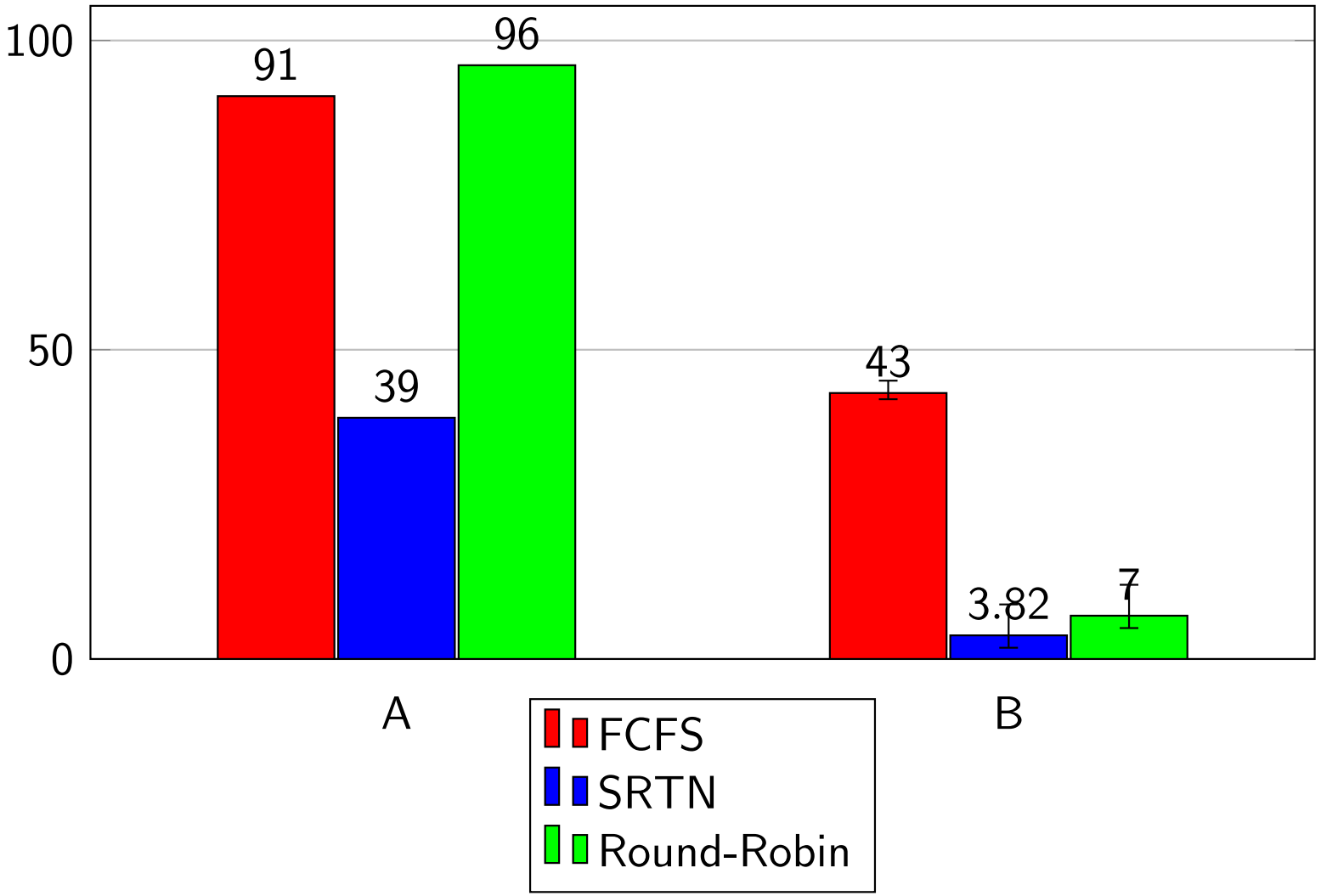
Arquivo de Trace: 1000 processos - Mudanças de contexto



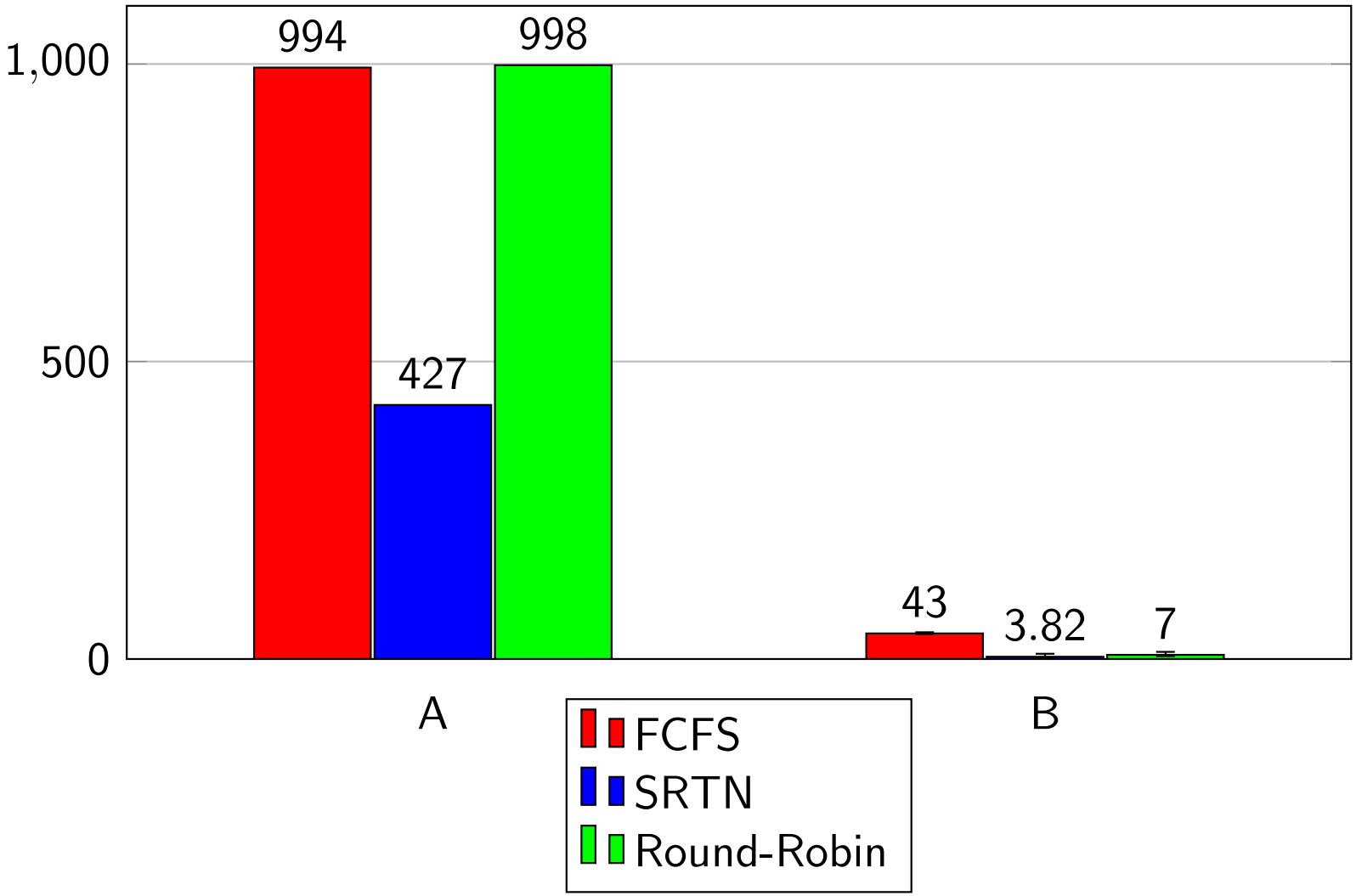
Arquivo de Trace: 10 processos - Deadlines



Arquivo de Trace: 100 processos - Deadlines



Arquivo de Trace: 1000 processos - Deadlines



Conclusões dos experimentos