EP1 - MAC0422

2º semestre 2020

Dupla:

Luciano Rodrigues Saraiva Leão (11221817)

Davi de Menezes Pereira (11221988)

ARQUITETURA DO SHELL BCCSH

O nosso shell é composto por um loop infinito, que espera a entrada do usuário com a função readline() da bilbioteca GNU readline, e após o comando e os parâmetros serem lidos, temos uma sequência de if e elses que dependendo do comando podem:

- Criar um diretório a partir da syscall mkdir.
- Matar um processo executando a syscall kill.
- Criar uma ligação simbólica com a syscall symlink.
- Executar um processo usando a syscall execve.

Simulador

O simulador é um programa em .c que recebe como parâmetros na linha de comando o que foi especificado no enunciado (./ep1 <escalonador> <arquivo_trace> <arquivo_saida> <parametro opcional d>), abre os arquivos de entrada e de saída, manda para o escalonador escolhido e depois fecha esses arquivos.

Para cada escalonador existe uma variável MAXN, que Está como 10000. Ou seja, o nosso código só suporta Até 10000 threads, e além disso só testamos até 1000. Portanto, para testar para valores maiores é necessário Alterar essa constante.

Escalonadores

Os escalonadores foram feitos como três bibliotecas diferentes, cada um com seus arquivos .c e .h.

Algumas das estruturas utilizadas nos escalonadores são idênticas, mas mesmo assim decidimos criar cada uma delas em cada arquivo, mesmo que repetida (como é o caso das structs processo_fcfs e processo_srtn) por motivos de teste e adaptação do código na hora de debugar.

Escalonadores

De um modo geral, todos seguem uma mesma estrutura. No .h de cada escalonador tem um protótipo de uma função do tipo
<escalonador>(arquivo_de_entrada, arquivo_de_saida, parametro_opcional_d)
que é a função principal, e uma struct para os processos.

No .c também temos algumas semelhanças em cada um deles. Temos um vetor de processos para armazenar todos os processos e temos um vetor de prontos que vai armazenando os processos que estão prontos, por exemplo.

Escalonadores

Na execução também seguimos a mesma ideia em cada escalonador: primeiramente lemos todos as linhas do arquivo trace e guardamos todos os processos num vetor de processos.

Depois disso entramos num loop: enquanto ainda existirem processos que ainda não ficaram prontos e enquanto ainda existirem processos prontos esperando serem executados realizamos o loop.
No inicio de cada loop checamos se algum processo acabou de finalizar, logo depois checamos quem ficou pronto e passamos para a fila de prontos, e finalmente executamos o algoritmos de cada escalonador para decidir quem irá usar a CPU.

Ao final de cada loop, dormimos os escalonador por um segundo e fazemos tudo novamente ou saímos do loop, esperamos as threads acabarem caso alguma esteja rodando (embora teoricamente não estejam mais), e saímos do escalonador.

Tempo

No desenvolvimento do EP1, um dos maiores desafios foi a sincronização. Testamos algumas bibliotecas e diferentes funções para controlar o tempo, como clock(), time() da biblioteca <time.h> e gettimeofday() da <sys/time.h>,além das funções do tipo sleep.

Porém tivemos muitos problemas para sincronizar, já que, por exemplo, a função time() às vezes causava uma imprecisão de quase um segundo e a clock() tinha problemas com o sleep.

No caso de usar só sleeps, não gostamos pois o consumo da CPU por parte das threads estava muitos baixo e para consertar isso teríamos que alterar muita coisa.

Tempo

Outro fator que dificultou é que, apesar de estarmos considerando uma CPU só para as threads, pelo escalonador gastar pouco tempo e CPU em relação às threads, consideramos que ele rodaria ""em paralelo"" em relação as elas. Então quando consideramos 1 segundo no loop do escalonador, na verdade ele executou 1 + épsilon segundos.

No final, conseguimos ajustar para ficar sincronizado usando as funções time() ou gettimeofday() e sleep() dependendo do caso, para os testes que fizemos.

Threads

Para pausar e voltar a executar as threads usamos o sugerido em: https://stackoverflow.com/questions/1606400/how-to-sleep-or-pause-a-pthread-in-c-on-linux

Considerando que na thread temos:

```
pthread_mutex_lock(&lock);
    while(!play) { /* We're paused */
        pthread_cond_wait(&cond, &lock); /* Wait for play signal */
    }
pthread_mutex_unlock(&lock);
```

Threads

Para pausar:

```
pthread_mutex_lock(&lock);
play = 0;
pthread_mutex_unlock(&lock);
```

Para dar play:

```
pthread_mutex_lock(&lock);
play = 1;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&lock);
```

E para achar em qual CPU a thread está ou liberou, usamos sched_getcpu() da biblioteca <sched.h>.

ESCALONADORES FCFS E SRTN

FCFS:
Tiramos um processo da fila de prontos e executamos ele até o final. Quando ele acaba, executamos o próximo pronto em ordem de chegada.

SRTN:
Deixamos os processos na fila de prontos, ordenados pelo dt, de modo
que o primeiro da fila é o que está sendo executado, só quando ele acaba de
executar tiramos ele da fila.

ESCALONADOR ROUND ROBIN

O quantum mínimo escolhido foi 1, pois equivale a duração mínima de um processo, já que estávamos considerando apenas números naturais.

Na nossa implementação, usamos uma fila normal e toda vez que um processo chegava, ele era colocado no final da fila e a cada segundo, o primeiro processo da fila (se houver) é retirado da 1ª posição e somente quando acabar o período equivalente ao quantum ele é colocado de volta na fila.

TESTES

Os testes foram gerados aleatoriamente, como arquivos de 5, 10, 20 e 100 processos por causa do tempo que tínhamos disponível.

Para cada processos fizemos os cálculos:

t0 += rand() % CONST_T0;

dt = rand() % MAX_DT + 1

X = (num_processos - (rand() % num_processos)) / (5 + rand() %

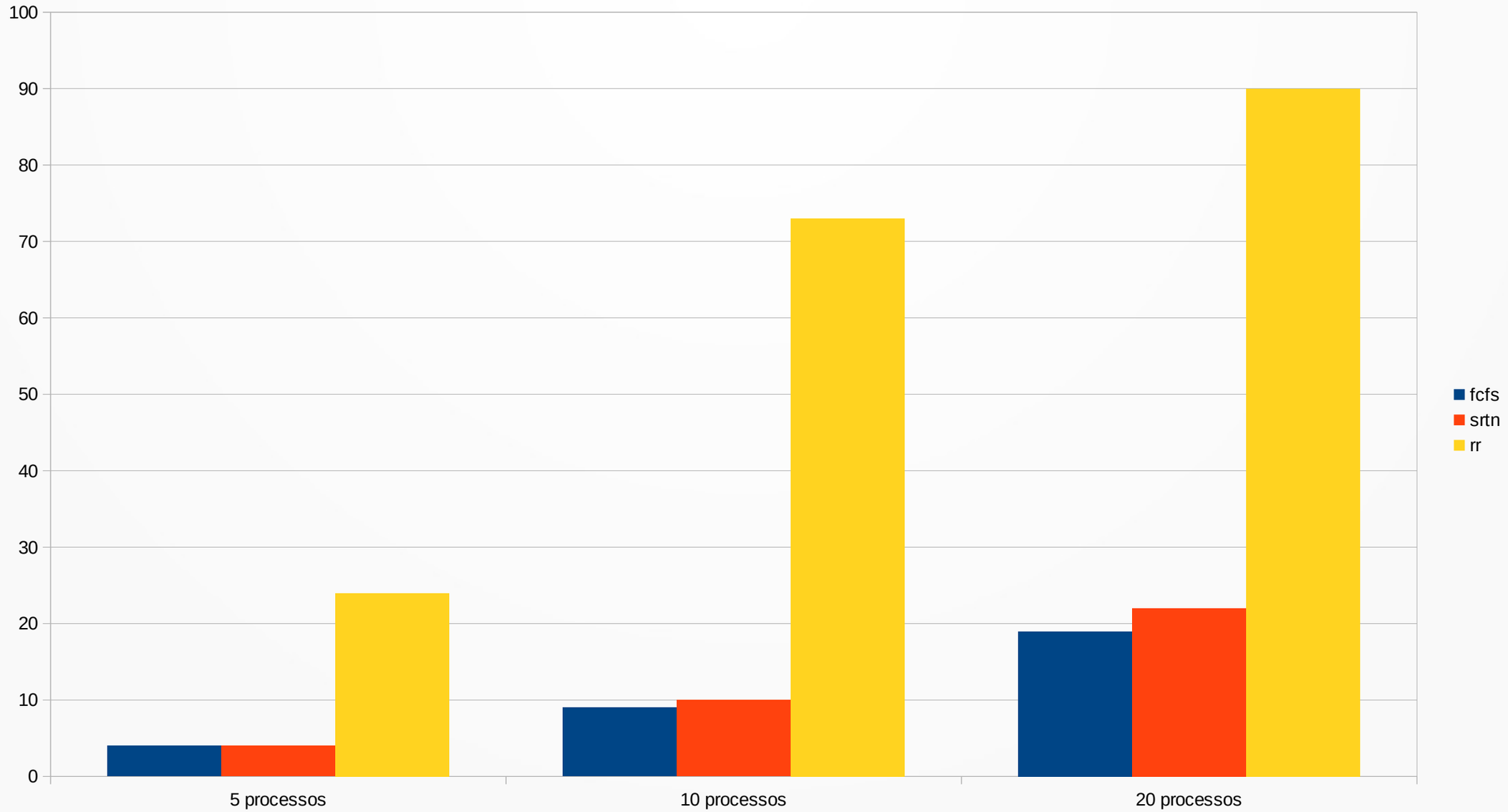
deadline = t0 + dt + 1 + (rand() % MAX_DT) *(x);

COM CONST_T0 = 5; MAX_DT = 9; CONST_DEADLINE = 50

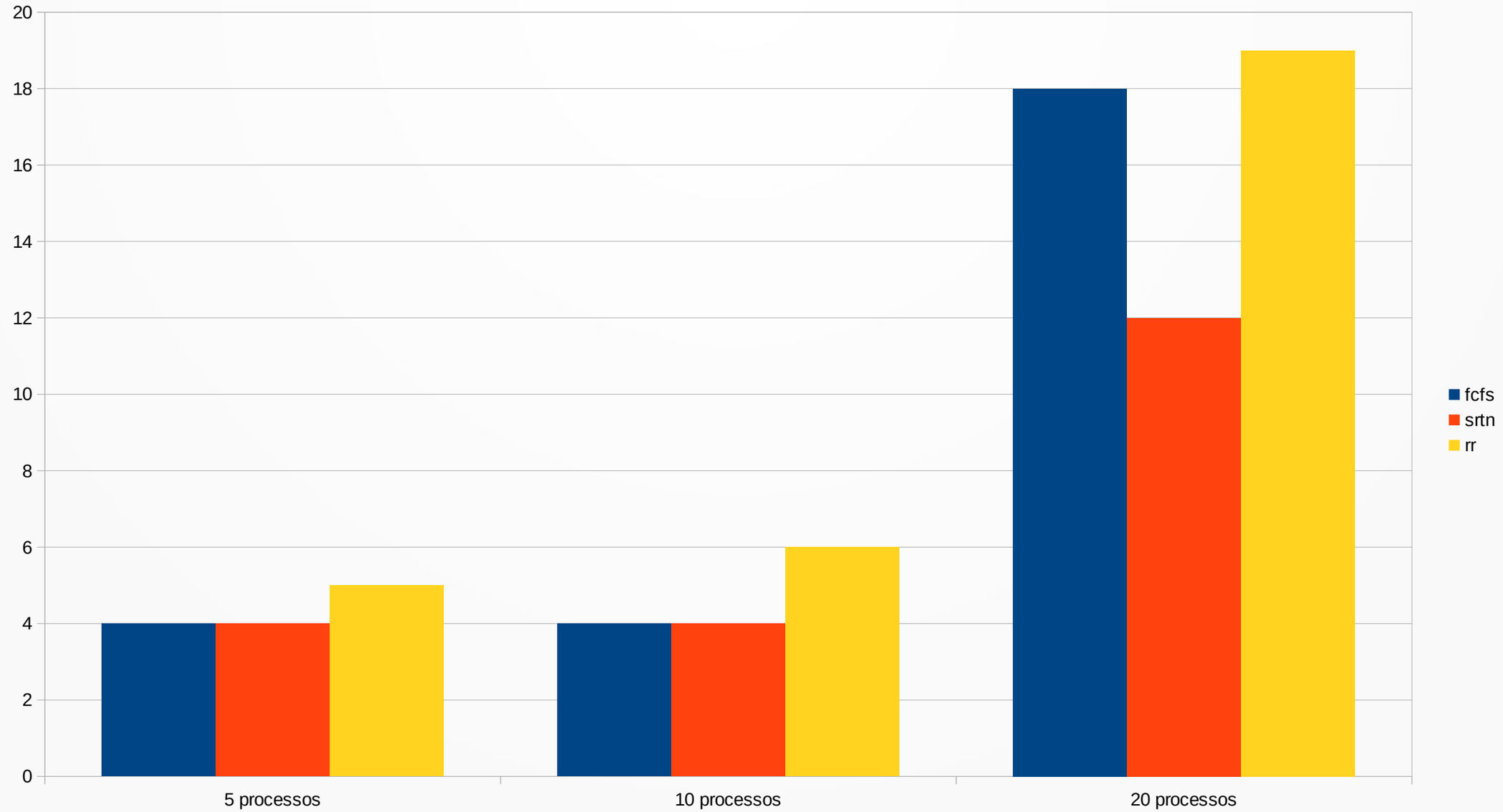
TESTES

Além disso, rodamos em duas máquinas diferentes, 30 testes Para cada tipo de arquivo.

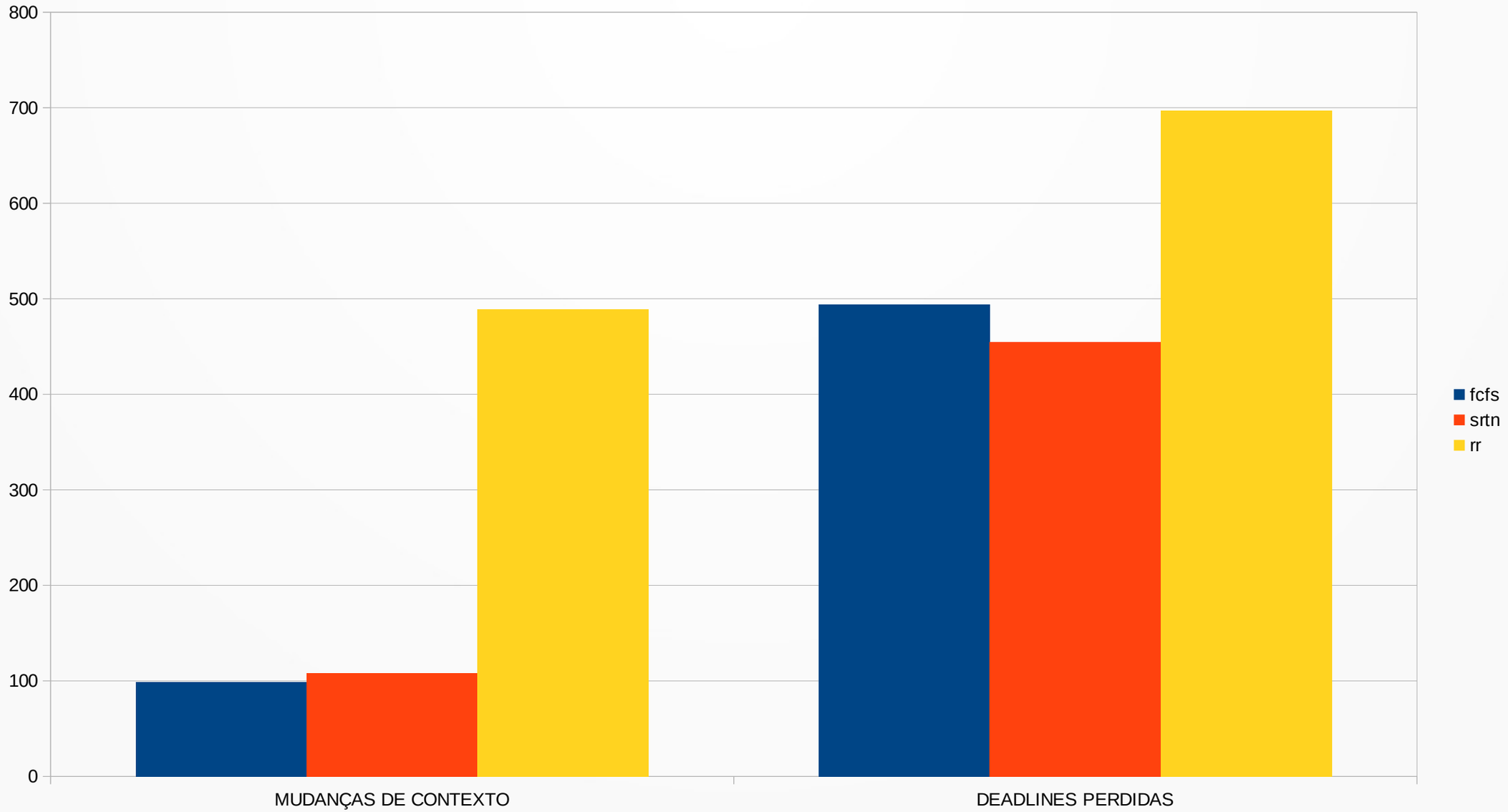
Mudanças de contexto



Deadlines não cumpridas



Teste com 100 processos



RESULTADOS

O resultado esperado, como comentado em sala, era que O escalonador Round Robin obtivesse o melhor resultado. Porém Não foi isso que aconteceu no nosso caso, primeiramente por Causa das constantes e cálculos escolhidos para gerar os testes, Mas também, e talvez principalmente, por causa do valor do QUANTUM que escolhemos. Não tivemos tempo de adaptar O nosso código para suportar valores de QUANTUM menores do Que um.

Além disso, percebemos que o uso da função time() deixou os Nossos algoritmos um pouco imprecisos em alguns casos, o que Levou a um desvio padrão maior do que zero.

Por fim, os outros resultados ocorreram dentro do esperado.