Process Simulator

João Marco Maciel da Silva 7577598
Renan Fichberg 7991131

Arquitetura do Shell

Arquitetura do Shell

1. *getcwd* do *unistd.h* para pegar o diretório atual, rodado automaticamente
2. *using_history* do *readline/history.h* para pegar o histórico dos comando
3. *readline* do *readline/readline.h* para pegar o comando do usuário
4. Testa de comando em comando qual foi o comando digitado

Um comando extra chamado *exit* apenas sair do *shell.*

Arquitetura do Shell: */bin/ls -1*

1. Verifica se o comando está correto (só aceita /bin/ls -1);
2. Se sim faz um fork;
    a. Original (pai) espera o filho terminar;
    b. Filho executa o *ls* com um *execve* do *unistd.h.*

Arquitetura do Shell: *pwd*

PWD

1. Imprime o diretório atual. O diretório atual é descoberto com uma chamada a getcwd, da unistd.h.

Arquitetura do Shell: *cd*

1. Filtra o argumento do cd, ignorando espaços desnecessários e interpretando alguns caracteres especiais, como ‘\0’ e o ‘\’. O argumento do cd pode ser tanto um caminho absoluto quanto um caminho relativo ao diretório atual (qual dos dois é determinado pelo primeiro caracter do argumento. Se for ‘/’, é absoluto, caso contrário, relativo).
2. Se de (1) obtivermos um caminho relativo, é concatenado ao final do diretório atual, descoberto por uma chamada automática a getcwd (mencionada no primeiro slide), o argumento já validado por (1). A nova string é passada de argumento para a função chdir, do unistd.h, que fará a mudança de diretório.
3. Se de (1) obtivermos um caminho absoluto, o próprio caminho absoluto é usado de argumento para a função chdir.

Arquitetura do Shell: histórico e readline

- No *shell* apertar para cima ou para baixo pode navegar pelo histórico e editar o texto para poder reutilizar graças ao *using_history*, o *history_expand*, o *add_history* e o readline das bibliotecas *readline/history.h e readline/readline.h*;
- Comando extra *show*:
  a. verifica se o comando está correto;
  b. pega o histórico com o comando *history_list* do *readline/history.h*;
  c. Imprime um por linha os comandos usados.

Arquitetura do Shell: *./ep1*

1. Verifica se o comando está correto e filtra as opções, removendo espaços desnecessários.
2. se estiver correto faz um fork;
   a. Original (pai) espera o filho terminar;
   b. Filho invoca *./ep1* com os argumentos usando *execve*.

Escalonadores

./ep1

1. Verifica se os parâmetros estão corretos;
2. Aloca a informação sobre as *threads* na memória e lê do arquivo as informações sobre eles;
3. Chama o escalonador passando a informação sobre as *threads* com a função *do_nome_do_escalonador*.

Informação sobre as Threads

Nossas threads estão associadas a uma struct com as informação de cada

- pthread_t thread;
- boolean coordinator;
- float arrival;
- boolean arrived;
- char name[64];
- float duration;
- float remaining;
- float deadline;
- float finish_cpu_time;
- float finish_elapsed_time;
- int priority;
- boolean working;
- boolean failed;
- boolean done;
- sem_t next_stage;
- pthread_mutex_t mutex;

Coordenador

Usa a mesma estrutura dos processos mais alguns campos

No *array* de processos para n processos o coordenador está no índice n e os outros processos de 0 a n-1

- unsigned int total;
- struct process *process;
- unsigned int *context_changes;

Obs: o coordenador é uma thread especial que sabe da existência das demais threads e pode acessar seus campos (em especial, os booleanos).

Visão geral do Fluxo

- Inicia todas as threads e deixa-as dormindo num semáforo (cada thread tem o seu) antes de começar a simulação;
- A n-ésima thread é a coordenadora;
- Sempre que um thread não está com a CPU, ela volta a dormir no seu semáforo até que o coordenador sinalize.
- Uma thread vai buscar uma CPU até que ela cumpra toda a duração do processo.
- Quando todas as threads cumprirem suas respectivas durações, a simulação acaba.

Processos

1. Um processo enquanto não estiver completo:
    a. Espera o coordenador assinalar uma CPU para o processo;
    b. Executa a tarefa:
        i. Pega o tempo agora;
        ii. Entra num loop de busy waiting onde:
            1. Pega o tempo novamente;
            2. Compara com o tempo do inicio;
            3. Atualiza o tempo dele;
        iii. Se não for FCFS nem SJF verifica se ainda é a vez, se não for mais a vez cai fora;
        iv. Verifica se estourou o tempo, se sim marca a flag failed;
2. Ao terminar, o processo muda a sua variável booleana “done”, registra o tempo de termino em relação ao início da simulação e encerra.

Coordenador - First-Come First-Served (FCFS)

1. Conta os processadores;
2. Setaffinity de todas os processos nas CPUs;
3. Inicializa os *cores*;
4. Pega o tempo do começo da simulação;
5. Entra num loop até acabarem as threads:
    a. Verifica o tempo e atualiza quem chegou;
    b. Entre os que chegaram e não estão assinalados pega o que chegou primeiro;
    c. Se tem *core* disponível assinala ele para o processo;
    d. Atualiza o contador dos que terminaram;
    e. Atualiza o número de *cores* livres;
6. Verifica o tempo e assinala os tempos do fim da simulação;

Coordenador - Shortest Job First (SJF)

1. Conta os processadores;
2. Setaffinity de todas os processos nas CPUs;
3. Inicializa os *cores*;
4. Pega o tempo do começo da simulação;
5. Entra num loop até acabarem as threads:
    a. Verifica o tempo e atualiza quem chegou;
    b. Entre os que chegaram e não estão assinalados pega o que tem duração mais curta;
    c. Se tem *core* disponível assinala ele para o processo;
    d. Atualiza o contador dos que terminaram;
    e. Atualiza o número de *cores* livres;
6. Verifica o tempo e assinala os tempos do fim da simulação;

Coordenador - Shortest Remaining Time Next (SRTN)

1. Conta os processadores;
2. Setaffinity de todas os processos nas CPUs;
3. Inicializa os *cores*;
4. Pega o tempo do começo da simulação;
5. Entra num loop até acabarem as threads:
    a. Verifica o tempo e atualiza quem chegou;
    b. Entre os que chegaram e não estão assinalados pega o que tem tempo restante mais curto;
    c. Se não tem *core* disponível verifica se tem algum processo fora da CPU que tenha um tempo de execução menor que um dos que estão trabalhando para fazer a troca;
    d. Se tem *core* disponível assinala ele para o processo;
    e. Atualiza o contador dos que terminaram;
    f. Atualiza o número de *cores* livres;
6. Verifica o tempo e assinala os tempos do fim da simulação;

Coordenador - Round-Robin (RR) - Parte 1

1. Conta os processadores;
2. Setaffinity de todas os processos nas CPUs;
3. Inicializa os *cores*;
4. Cria uma lista ligada circular dos processos que já chegaram, na ordem que eles chegaram;
5. Calcula o *quantum* (= tempo que cada processo terá de uso de CPU);
    a. no nosso caso usamos a raiz quadrada da soma das durações;
6. Pega o tempo do começo da simulação;

Coordenador - Round-Robin (RR) - Parte 2

1. Entra num loop até acabarem as threads:
    a. Se tem processos não assinalados e todos os *cores* estão ocupados tenta liberar algum;
        i. Só é liberado se algum processo já ficou tempo suficiente processando ou
        ii. Se um processo que está com a CPU terminou sua tarefa antes do quantum.
    b. Verifica se algum novo processo chegou:
        i. Se chegou adiciona ele no final da lista ligada dos processos;
    c. Pega o próximo processo da fila;
    d. Assinala um core livre para o processo;
    e. Atualiza o contador dos que terminaram (removendo da lista);
    f. Atualiza o número de *cores* livres;
2. Verifica o tempo e assinala os tempos do fim da simulação;

Coordenador - Escalonamento com prioridade (PS)

Similar ao RR, mas a fila é construída considerando a prioridade ao invés da ordem de chegada. Para selecionar o próximo processo, diferente do RR, o ponteiro sempre começa do inicio da fila, enquanto no RR o ponteiro sempre começa do próximo da fila, que vem imediatamente depois do último a pegar a CPU. A maneira como o quantum é calculado também é diferente.

Coordenador - Escalonamento em tempo real com deadlines rígidos (EDF)

1. Conta os processadores;
2. Setaffinity de todas os processos nas CPUs;
3. Inicializa os *cores*;
4. Pega o tempo do começo da simulação;
5. Entra num loop até acabarem as threads:
    a. Verifica o tempo e atualiza quem chegou;
    b. Entre os que chegaram e não estão assinalados pega o que tem deadline menor;
    c. Se não tem *core* disponível verifica se tem algum processo fora da CPU que tenha um deadline menor que um dos que estão trabalhando para fazer a troca;
    d. Se tem *core* disponível assinala ele para o processo;
    e. Atualiza o contador dos que terminaram;
    f. Atualiza o número de *cores* livres;
6. Verifica o tempo e assinala os tempos do fim da simulação;

Resultados dos Experimentos

Máquinas usadas

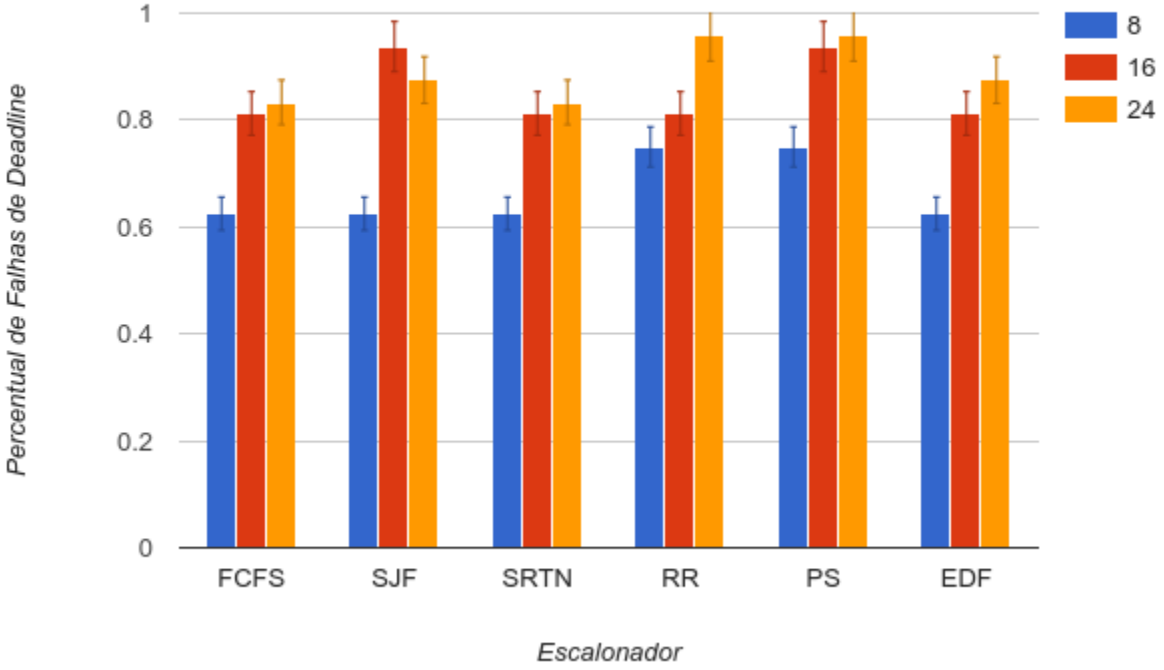
Máquina 2 cores:

- Intel Core(TM)2 Duo T8300
- Processadores: 2
- Clock: 2.4GHz
- Cache L1: 64kB
- Cache L2: 3072kB
- Cache L3: N/A
- RAM: 2x2GB DDR2

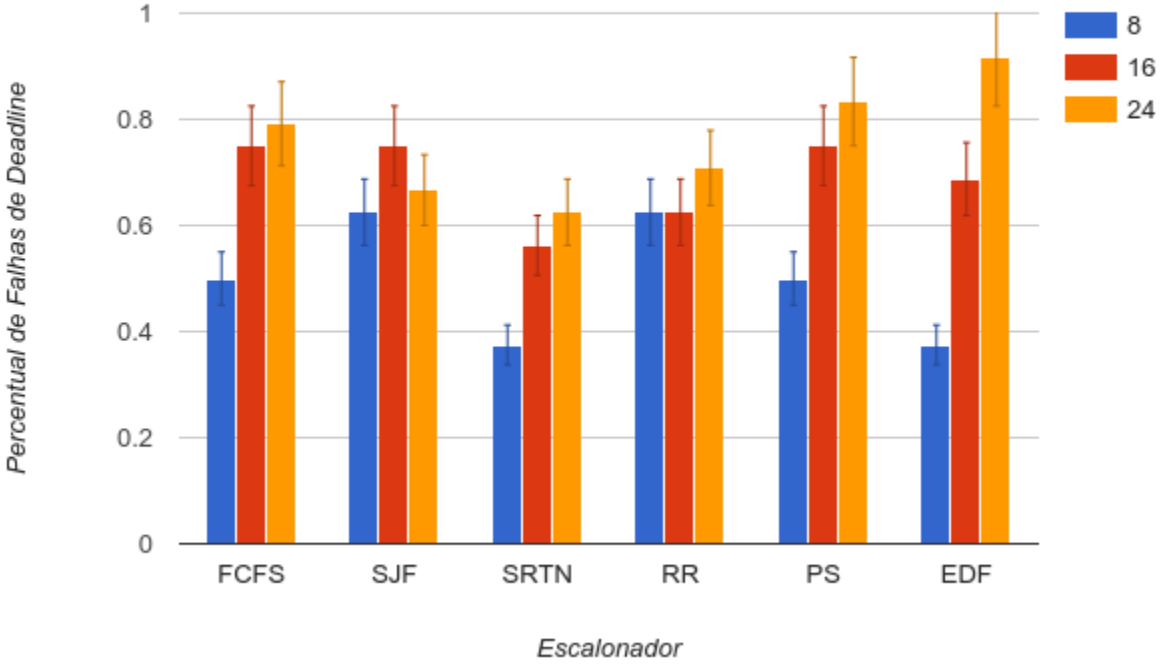
Máquina 4 cores:

- Intel Core 2 Quad i5-3450
- Processadores: 4
- Clock: 3.10GHz
- Cache L1: 64kB
- Cache L2: 256kB
- Cache L3: 6144kB
- RAM: 1x8GB DDR3

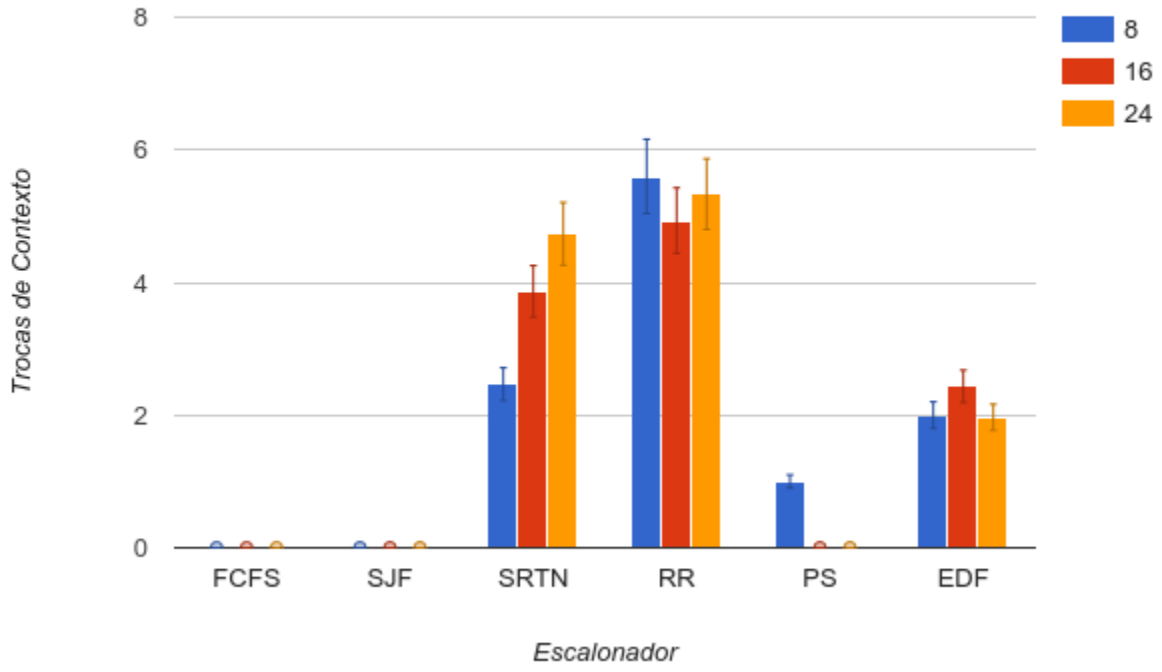
MÁQUINA 1: Deadlines Por Quantidade de Processos



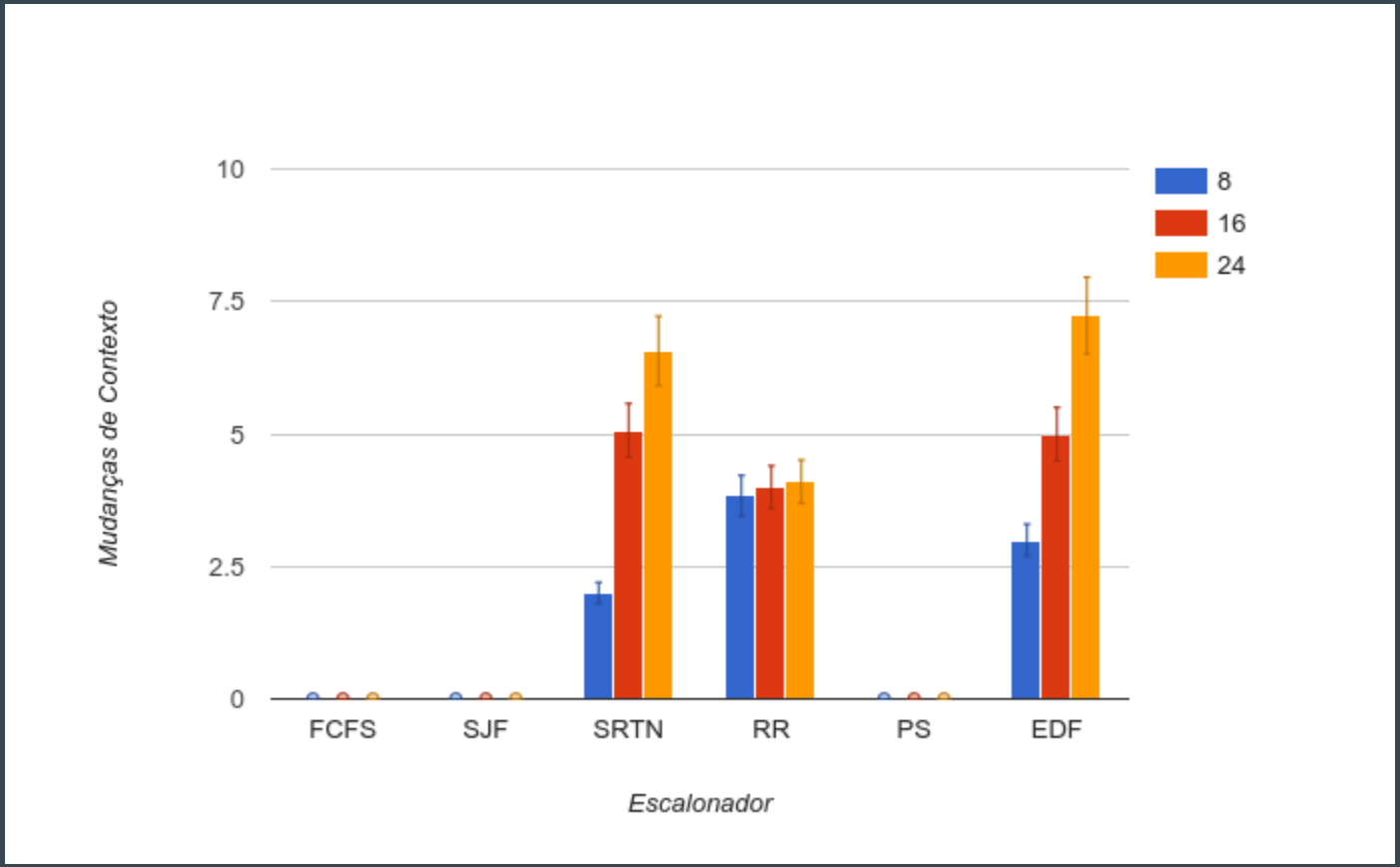
MÁQUINA 2: Deadlines Por Quantidade de Processos



MÁQUINA 1: Mudanças de Contexto



MÁQUINA 2: Mudanças de Contexto



Justiça

Pela forma que fizemos os 6 algoritmos de escalonamento, podemos afirmar que:

1. FCFS: por considerar a ordem de chegada dos processos, garante que, em algum momento, o processo que está na fila irá executar, portanto é justo.
2. SJF: apesar de não preemptivo, o algoritmo tem um alto potencial de *starvation*, uma vez que ele escalona sempre os processos de duração menor. Injusto.
3. SRTN: a versão preemptiva do SJF implementada no EP também tem o mesmo problema de *starvation* do SJF.
4. RR: este algoritmo é justo exatamente por dividir o tempo de CPU entre processos sem considerar as informações particulares destes. Não há *starvation*.
5. PS: este algoritmo, do jeito que foi implementado no Simulador de Processos, não é justo, pois os processos de maior prioridade sempre revezaram entre eles o uso das CPUs..
6. EDF: Este algoritmo, apesar de ser implementado de forma similar ao SRTN, pode ser justo se considerar que os processos que chegam estão configurados com o mesmo relógio do simulador. Se os tempos que chegarem forem arbitrários (o que não faria sentido para usar o EDF), daí possivelmente ele seria injusto. O que garante a justiça, nesse caso, é o mesmo motivo que garante para o FCFS. O tempo vai passando, e em algum momento o deadline de alguém que já esperou terá de ser o mais favorável para assumir uma CPU.

Considerações

Decidimos calcular os *quantums* do RR e do PS ao invés de usarmos um valor fixo pois nos nossos testes, nossos valores iniciais estavam atrasando consideravelmente a simulação. Com os novos métodos para definição do *quantum* que implementamos, houve o seguinte *tradeoff* (que detectamos com os nossos testes):

1. Redução do número de mudanças de contexto
2. Aumento do número de deadlines finalizados a tempo
3. Diminuição do tempo total de simulação

Obviamente que com isso, os eventos de troca de contexto tornaram-se mais raros, mas considerado o item 2, houve um ganho de eficiência (que está relacionado com o item 3).