Trabalho3

Victor Putrich, Lucca Dornelles e Guilherme Santos

Introdução

Este relatório visa apresentar e avaliar os resultados das execuções de um programa multithread e seus diferentes escalonamentos de acordo com a mudança de política de escalonamento selecionada, além da mudança em ambientes de processamento multi-core e single-core.

O objetivo deste trabalho é a criação de um programa multithread com um número específico de threads que escrevem caracteres em um buffer global. A escrita no buffer é sincronizada, de forma que não ocorra sobrescrita de uma posição, pois assim é possível verificar o escalonamento das threads durante a execução através da leitura do conteúdo do buffer. Para facilitar a análise, ao concluir a execução o programa mostra a leitura do buffer e a contagem de quantas vezes cada caráter foi escalonado.

Escalonadores

No nosso programa existe um grupo de escalonadores que podem ser utilizados, estes estão divididos em *Completely Fair Scheduler* (CFS) e *Real-time scheduler*. Dentro dessas duas classes, existem diferentes políticas disponibilizadas pelo sistema operacional que podem ser explicitamente definidas ao abrir um thread na execução do programa. Algumas políticas ainda tem disponível um intervalo de prioridade que varia de política para política. O escalonador usa a política aliada à prioridade *sched_priority* para tomar a decisão de qual thread deve ganhar o uso da cpu.

CFS

As três políticas CSF disponibilizadas para uso no sistemas Linux não tem diferenciação através do parâmetro *sched_priority*, portanto todos são passados com valor 0.

- **SCHED_OTHER**: Esta é a política default do Linux. Não existe uma prioridade que pode ser definida de forma estática para os processos, ao invés disso as threads que forem usar essa política, tem um valor de prioridade dinâmica chamado "nice value". O objetivo desse atributo é garantir que todas as threads possam vir a "progredir" de forma justa. O valor é ajustado para cada "quantum" que as threads estão em estado *ready*.

- **SCHED_BATCH**: Semelhante à política anterior, o escalonador de *batch* também usa o atributo dinâmico *nice value* para buscar balancear a execução das threads.

A diferença dessa política é que ela penaliza tarefas que envolvam serem "despertadas". Portanto, é interessante usá-la em tarefas que não sejam tão interativas com o usuário. Tarefas que buscam um escalonamento mais determinístico que não seja dependente de interação, podem se beneficiar desta política.

- **SCHED_IDLE**: Nesta política não há prioridade estática ou dinâmica, tarefas que usam essa política tem como objetivo ser de baixíssima prioridade.

Real-time

Aquelas threads que usam as políticas real-time fazem uso do *sched_priority* que tem o intervalo de 1 a 99. Sendo 1 o valor mínimo e 99 o valor máximo. A diferença entre as duas políticas não está entre threads com diferentes prioridades, a thread com maior prioridade sempre vai ganhar acesso à cpu, preemptando a thread corrente. A mudança entre elas está na decisão do que fazer entre threads que têm a mesma prioridade e em que posição na fila adicionar novas threads com diferentes prioridades. Importante comentar que threads que tiverem usando políticas real-time sempre terão preferência sobre a políticas CFS.

- **SCHED_FIFO:** A política *First in first out*(FIFO) basicamente adiciona os processos a serem executados por ordem de chegada e são alocados da mesma forma. Uma thread que for preempitada por outra thread de prioridade maior, permanecerá na frente da fila e retornará à execução assim que a thread de maior prioridade for bloqueada. A thread que for bloqueada ao voltar para o estado *runnable*, volta para o final da fila. Além disso, threads de maior preferência vão "furar" a fila até a posição onde exista uma thread de maior prioridade à sua frente.

- **SCHED_RR**: *Round-robin* é uma política mais "justa" se compararmos com a FIFO, sua política é menos suscetível a ocorrência de *starvation*. Diferente da política anterior que roda seus processos até que estes ou sejam preemptados, ou fiquem bloqueados, a round-robin atribui um "tempo de execução" para cada thread, chamado de *quantum*. Quando um processo atinge o limite do seu *quantum* é posto no final da fila, de acordo com sua prioridade.

- **SCHED_DEADLINE** A política em questão é usado para o cumprimento das chamadas *sporadic tasks* que é composta por um conjunto de *jobs*. São executados um *job* por vez, e cada tem um tempo de *deadline* que seria o momento que o programa a ser executado, caso já não tenha realizado o seu trabalho, acaba descartando todo trabalho feito até então. Cada *job* a ser executado é composto por duas características, a *relative deadline* e o tempo de execução. A chamada *absolute deadline* é a soma destes dois valores. Essa política ajusta os tempos de *deadline* conforme a demanda individual de cada task, buscando balanceá-las para evitar starvation.

Implementação

Argumentos de Entrada

Para executarmos a nossa aplicação precisamos informar primeiro o número de threads usadas e o tamanho da memória, além disso, para cada thread precisamos explicitamente definir qual a política, podendo ser qualquer uma das listadas anteriormente e a sua respectiva prioridade que varia de política para política. Podemos verificar o formato da entrada logo abaixo:

```
  ./setpriority <NUMBER_THREADS> <MEM_SIZE> <POLICY 0> <PRIORITY 0> ...
<POLICY N> <PRIORITY N>
```

Podemos rodar por exemplo 4 threads com a política round-robin, todas com prioridade mínima e com memória de tamanho 100.000 da seguinte forma:

```
  ./setpriority 4 100000 SCHED_RR 1 SCHED_RR 1 SCHED_RR 1 SCHED_RR 1
SCHED_RR 1 SCHED_RR 1
```

Função Run

Para que pudéssemos acompanhar as políticas de escalonamento competindo pelo controle da CPU para escrever a sua letra na memória, definimos dentro da função "run" o método de acesso da memória que pode ser vista por todas as threads do programa. Primeiramente, para que todas as threads concorram pelo acesso à cpu de forma justa, no início da função usamos um método de sincronização via *mutex* que será explicado posteriormente, isto faz com que todas as threads iniciem o processamento de fato juntas. Cada thread recebe uma letra que é passada pelo parâmetro *void data* da função. Este parâmetro é um valor de deslocamento que é incrementado ao caráter inicial do alfabeto. Como o programa foi escrito em C, basta somarmos a 'a' o valor de offset e teremos o caractere da thread correspondente.

```
  char letter = 'a'+(int)data;
```

A parte principal do programa consiste num while que roda enquanto a memória não estiver preenchida por completo, isso é feito comparando o tamanho máximo de memória *SIZE_BUF* com um contador que serve para sempre apontar para a próxima posição de

memória a ser escrita *index_mem*. O while ocorre enquanto o index for menor ou igual ao tamanho máximo de memória.

Controle de Acesso

Utilizamos dois objetos mutex para coordenar o funcionamento das threads.

- **simLock** trava qualquer thread que tente rodar antes que todas sejam inicializadas, fazendo com que todas as threads comecem simultaneamente.

- **lock** É utilizado dentro das threads para impedir que uma thread rode enquanto outra esteja na zona crítica (que é a integridade da thread após sua inicialização). Enquanto uma thread está escrevendo no buffer todas as outras esperam e só após o processo de escrita acabar que o scheduler pode trocar de thread.

Perceba que *index_mem* e a memória em si são consultadas e escritas por múltiplas threads, para evitar que exista Condição de Corrida no nosso programa, adicionamos uma restrição de acesso durante a escrita em memória e incremento da variável *index_mem*. Dessa forma evitamos por exemplo que duas threads acabam escrevendo na mesma posição de memória ou incrementando o index de forma duplicada, fazendo com que ele incremente somente uma vez ao invés de duas.

Podemos ver a seguir a implementação da função por completo:

```
void *run(void *data)
{
	//iniciar todas threads "simultaneamente"
	pthread_mutex_lock(&simLock);
	pthread_mutex_unlock(&simLock);

	char d = 'a'+(int)data;
	while (index_mem <= SIZE_BUF){
    // zona crítica
		pthread_mutex_lock(&lock);
		mem[index_mem] = d;
		index_mem+=1;
		pthread_mutex_unlock(&lock);
	}
	pthread_mutex_unlock(&lock);
	return 0;
}
```

Testes

Para os testes rodados, primeiro foi definido um critério de análise das políticas. Como temos uma gama de escalonadores e podemos usar diferentes métodos em diferentes threads, vamos definir primeiro situações interessantes para avaliarmos:

- **Single-core:** Para que possamos verificar o comportamento puro de cada política, devemos executá-las em single-core para um número pequeno de threads. Dessa maneira, se usarmos, por exemplo, uma mesma política real-time com threads diferentes, porém variando o valor de prioridade, poderemos verificar a organização da fila das threads, assim como a execução da mesma e troca de contexto. Nesse caso, executará somente a thread com maior prioridade, independente se for FIFO ou RR.

- **Multi-core:** Usando mais de uma cpu na execução do programa, podemos avaliar como o escalonado atribui para cada cpu os processos a partir da sua política e prioridade. Dessa vez, tomando o exemplo anterior, a thread de maior prioridade deve ficar ocupando uma das cpu o tempo todo, enquanto as outras threads que tenham prioridade inferior e igual, devem disputar o uso da cpu disponível.

- **Política Igual:** Vamos testar também testes com múltiplas threads executando a mesma política, poderemos verificar se o escalonamento condiz com a descrição do comportamento de cada política conforme descrito anteriormente.

- **Classes Diferentes:** Veremos como o escalonador se comporta quando rodamos threads que estão usam alguma das políticas da classe CFS, assim como políticas de classe real-time. Poderemos acompanhar o escalonamento destas usando 1, 2 ou + cpus.

Teste 1

A tabela abaixo descreve a configuração usada para este teste, avaliamos o comportamento de 4 escalonados CSF Other.

CPU	MEM(kb)	Threads	th1	th2	th3	th4
1	1000000	4	OTHER 0	OTHER 0	OTHER 0	OTHER 0

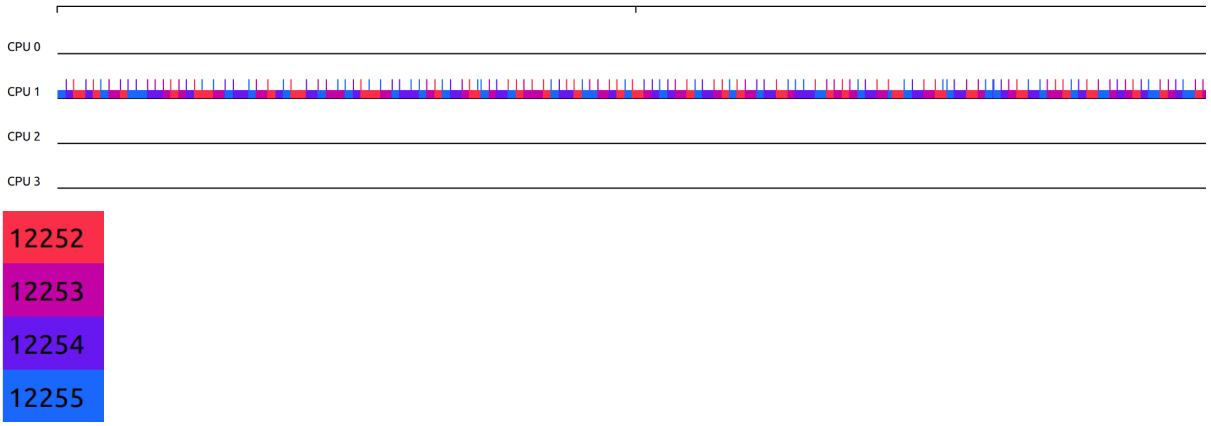
O resultado da execução foi:

Contagem letras:

- (PID 12252) a: 249288KB

- (PID 12253) b: 250688KB
- (PID 12254) c: 251130KB
- (PID 12255) d: 248892KB

Podemos verificar que tanto a saída textual, apresentada acima, como a saída gráfica de execução na ferramenta kernelshark mostra o comportamento do escalonador com uma distribuição muito parelha entre todas as tasks executadas. Todas tiveram um tempo de uso de cpu parelho, apresentando uma média de 249999KB por thread e desvio padrão 1077.



Teste 2

Neste teste foram utilizadas 3 threads com prioridades iguais e escalonadores batch, junto a elas, atribuímos à thread 4 a política IDLE.

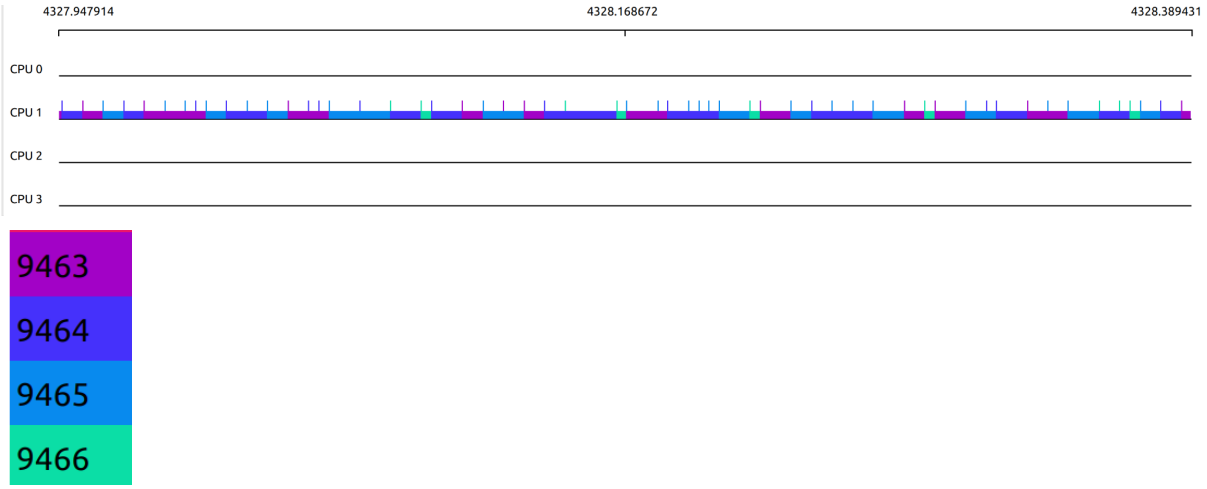
CPU	MEM(kb)	Threads	th1	th2	th3	th4
1	1000000	4	BATCH 0	BATCH 0	BATCH 0	IDLE 0

Contagem letras:

- (PID 9463) a: 326150KB
- (PID 9464) b: 339542KB
- (PID 9465) c: 331073KB
- (PID 9466) d: 3233KB

Acreditamos que para este teste, o batch deve ter comportamento semelhante à política do SCHED_OTHER, pela execução do programa não tratar ou esperar nenhum tipo de interrupção. A discrepância da quarta thread com as anteriores se dá pela sua política ser de baixa prioridade, indicando que aquela thread não está executando no momento um trabalho que exija controle da cpu.

Como esperado, as três primeiras threads executaram trabalhos semelhantes, a média de trabalho executado pelas 3 foi de 332255KB com desvio padrão 6773. A quarta thread quando comparada com as outras três realizou aproximadamente 1% do trabalho



Teste 3

A configuração do teste 3 se encontra na tabela abaixo:

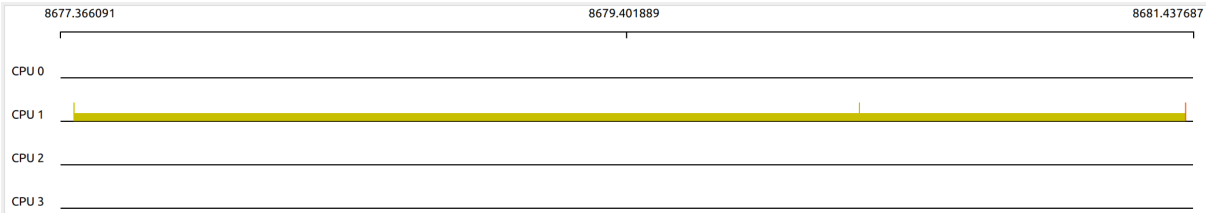
CPU	MEM(kb)	Threads	th1	th2	th3	th4
1	200000	4	FIFO 1	FIFO 1	FIFO 1	FIFO 1

As informações de execução das threads:

Contagem letras:

- (PID 5964) a: 200000KB
- (PID 5965) b: 0KB
- (PID 5966) c: 0KB
- (PID 5967) d: 0KB

O terceiro teste utiliza a política FIFO que roda o primeiro processo até ele acabar, for interrompido, ou preemptado por outro com prioridade maior. Como as prioridades são iguais, só o primeiro processo conseguiu rodar, o que é o esperado.



Teste 4

O objetivo do teste 4 é avaliar o comportamento da política round-robin para processos de mesma prioridade:

CPU	MEM(kb)	Threads	th1	th2	th3	th4
1	200000	4	SCHED_RR 1	SCHED_RR 1	SCHED_RR 1	SCHED_RR 1

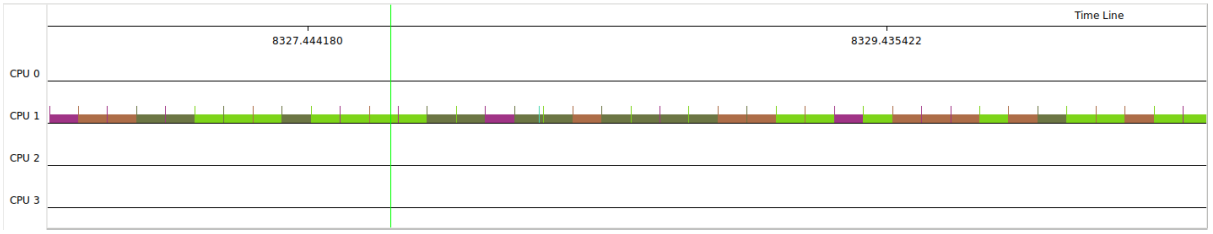
Os resultados obtidos dos testes foram:

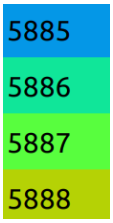
Contagem letras:

- (PID 5885) a: 12421KB
- (PID 5886) b: 50875KB
- (PID 5887) c: 75794KB
- (PID 5888) d: 60909KB

No teste do Round Robin foram utilizadas 4 threads com prioridade igual. Aqui, conseguimos comparar a diferença das duas políticas de mesma classe, RR e FIFO. Na política Round-Robin, conseguimos ver que o escalonador designa a execução da cpu de forma a respeitar o *quantum*. Um comportamento que fica fácil de identificar quando visualizamos o log de trocas de contexto no wireshark é que muitas vezes o *quantum* de uma task acaba enquanto a thread está com o lock do mutex. Ou seja está na zona crítica de execução, isso faz com que a thread que seria a próxima a ser executada pelo escalonador não consiga de fato acesso à memória. O escalonador então vai se movendo pela fila até chegar no final dela, dando a execução novamente à thread que está com o lock.

A média de acesso por thread foi de 49999KB com desvio-padrão de 27063. O desvio padrão é relativamente alto já que estamos testando uma política que busca distribuir igualitariamente o uso da cpu, acreditamos que existam duas explicações para isso: A primeira como comentado anteriormente, devido a nossa aplicação ter zona crítica, acaba alterando a ordem de execução da política que tenta passar a cpu para a próxima thread da fila mas não consegue. A segunda situação é o tamanho de memória que talvez fosse insuficiente para que o balanceamento da política fique visível. Note que este balanceamento é feito pelo primeiro problema relatado.





Teste 5

No teste 5 passamos a configurar a execução do programa para que seja executada em 2 cpus.

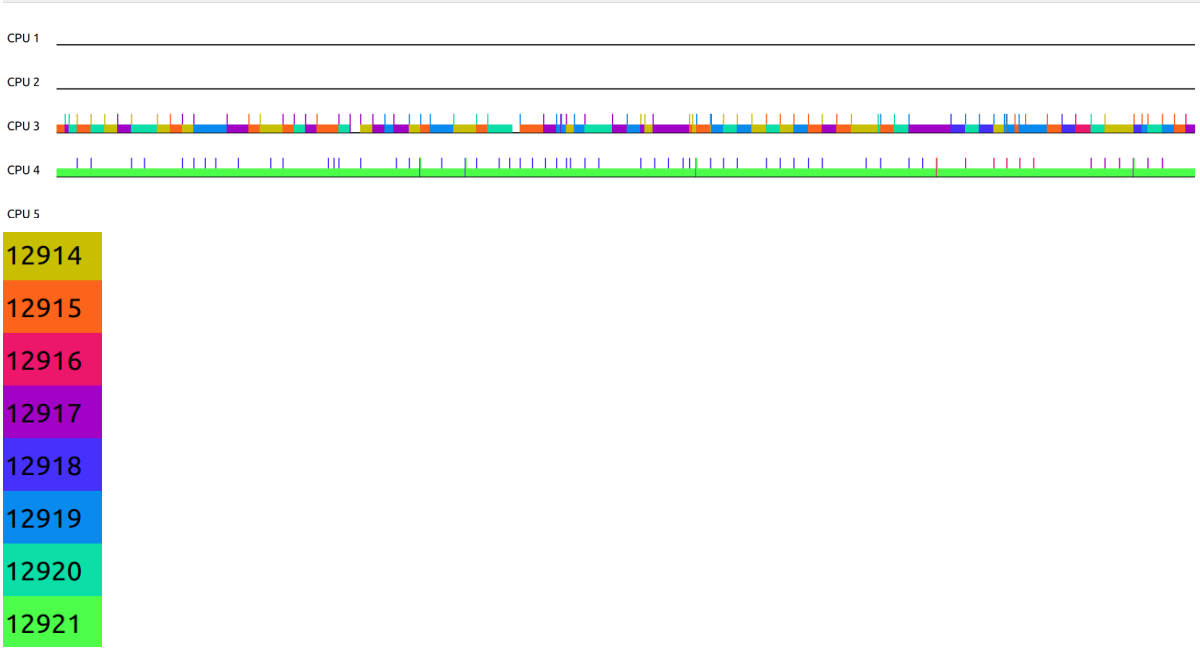
CPU	MEM(kb)	Threads	th1	th2	th3	th4	th5	th6	th7	th8
2	100000	8	OTHER 0	OTHER 0	OTHER 0	OTHER 0	OTHER 0	OTHER 0	OTHER 0	RR 1

O objetivo desse teste é verificar o que ocorre quando existem threads de diferentes classes (CFS e real-time)

Contagem letras:

- (PID 12914) a: 9581KB
- (PID 12915) b: 9488KB
- (PID 12916) c: 9483KB
- (PID 12917) d: 9449KB
- (PID 12918) e: 9265KB
- (PID 12919) f: 9614KB
- (PID 12920) g: 8945KB
- (PID 12921) h: 34169KB

O algoritmo other procura balancear as threads pelo *nice value*, o que gera uma contagem de letras ainda mais parelha e intercalada para os primeiros 7 processos, enquanto a última thread toma conta da segunda cpu e fica com ela ao longo de toda execução.



Em alguns momentos tivemos uma certa "instabilidade" na distribuição da primeira cpu, aparecendo inclusive alguns espaços em branco. Estes espaços são ocupados por processos fora da execução da nossa aplicação. Usamos um filtro para que apareçam somente tasks que nos interessam para este trabalho.

Teste 6

O teste 6 foi feito com o objetivo de avaliarmos o comportamento de um grande número de threads com diferentes políticas e prioridades:

CPU	MEM(kb)	Threads	th1	th2	th3	th4	th5	th6	th7	th8
3	200000	8	OTHER 0	OTHER 0	RR 1	FIFO 1	RR 1	RR 1	RR 1	RR 10

Usamos 3 cpus para a execução de 8 threads. Uma delas tem prioridade 10, quatro outras têm prioridade 1 e ainda temos 2 threads CFS.

Contagem letras:

- (PID 13858) a: 364KB
- (PID 13859) b: 345KB
- (PID 13860) c: 26962KB
- (PID 13861) d: 27097KB
- (PID 13862) e: 26859KB
- (PID 13863) f: 26711KB
- (PID 13864) g: 26804KB

- (PID 13865) h: 64854KB

Surpreendentemente, todas threads conseguiram acesso à alguma das cpus. A thread 8 (13865) como tinha maior prioridade, recebeu praticamente uma cpu para sí. As cpus restantes foram disputadas principalmente pelas threads *real-time* com prioridade 1 (threads 3,4,5,6 e 7). Curiosamente, as threads 1 e 2 da classe CFS também tiveram seu tempo, mesmo que ínfimo para executar algum trabalho. Se aproximarmos o gráfico, podemos ver que por curtos momentos a thread 1 e 2 assumem a cpu, no momento que as threads *real-time* trocam de contexto.