Scheduler Isolation

Grupo 5

- Maria Eloisa
- Tiago A. Fontana
- William Kraemer

The didactic version of OpenEPOS implements thread scheduling (or CPU scheduling) as a property of the process management mechanisms. This is actually the way most OS implement it, but in essence, scheduling policies and mechanisms could be reused for disc, network, and other system resources. EPOS original motivation to isolate the scheduler was the possibility to implement it in hardware and thus reduce jitter for hard real-time tasks (actually threads, but that's how RT people name them)[1]. However, this refactoring has proven a very effective way to practice advanced software development techniques and get a insight of System-level Desgin.

To do

Before refactoring EPOS to isolate the scheduler, inspect your current implementation and answare the following questions:

1. What are the policies implemented by the original multilevel scheduler? What is the inter-level policy? And what is the intra-level?
2. How is it that EPOS implements a multilevel scheduler with a single Ready Queue?
3. Could you implement other policies using the same strategy?

After answering the question, refactor the scheduler out of class Thread to create a new Scheduler class. Queues such as Ready must now be whitin Scheduler. Preserve the original polices.

Respondendo primeiramente as perguntas feitas para realização dos exercícios:

1. Quais as políticas implementadas pelo escalonador multinivel original? Qual a política inter-level? E a intra-level?

Inter-level: O escalonador atualmente guarda as threads em uma fila organizada pela prioridade definida na classe thread:

```
// Thread Priority
typedef int Priority;
enum {
        MAIN   = 0,
        HIGH   = 1,
        NORMAL = (unsigned(1) << (sizeof(int) * 8 - 1)) - 4,
        LOW    = (unsigned(1) << (sizeof(int) * 8 - 1)) - 3,
        IDLE   = (unsigned(1) << (sizeof(int) * 8 - 1)) - 2
};

// Thread Queue
typedef Ordered_Queue<Thread, Priority> Queue;
```

Esta fila é organizada em ordem crescente de valor numérico de prioridade. Da forma como é implementado atualmente, enquanto tivermos threads de alta prioridade, estas serão alocadas sempre no início da fila e threads menos prioritárias não executam nunca (*starvation*).

Intra-level: Dentro de um mesmo nível é escolhido o tipo de escalonamento. Em caso da preemptividade estar ativa, a política escolhida é um *round-robin* (RR). Ao fim do quantum, a thread que estava rodando é escalonada para o final da fila e a nova cabeça da fila é chamada para rodar. Caso não estiva com a preemptividade ativada, a política é *first come first served* (FCFS). A primeira thread criada executa o tempo que for necessário e só é chamada uma nova thread ao final da sua execução ou quando precisar de algum recurso que esteja bloqueado. É então colocada ao final da fila e deverá esperar as outras threads terminarem de rodar para que volte a execução.

```
protected:
        static const bool preemptive = Traits<Thread>::preemptive;
```

2. Como o EPOS implementa um escalonador multinível com uma única fila *ready*?

A fila *ready* é organizada sempre que uma nova thread é inserida na Ordered_List:

```
void insert(Element * e) {
        if(empty())
```

```
                insert_first(e);
        else {
                Element * next, * prev;
                for(next = head(), prev = 0; (next->rank() <= e->rank()) &&
        next->next(); prev = next, next = next->next())
                        if(relative)
                                e->rank(e->rank() - next->rank());
                if(next->rank() <= e->rank()) {
                        if(relative)
                                e->rank(e->rank() - next->rank());
                        insert_tail(e);
                } else
                        if(!prev) {
                                if(relative)
                                        next->rank(next->rank() - e->rank());
                                insert_head(e);
                        } else {
                                if(relative)
                                        next->rank(next->rank() - e->rank());
                                Base::insert(e, prev, next);
                        }
        }
}
```

De forma bem simplista, a Ordered_Queue funciona como se tivéssemos várias pequenas filas dentro de uma fila principal e as threads vão sendo alocadas de acordo com seu rank dentro destas filas. Esta simulação garante que não seja necessário buscar informações sobre o estado das listas (se estão vazias ou não) para que uma próxima thread seja escalonada. No entanto este processo faz com que threads de baixa prioridade fiquem muito tempo na fila esperando para poder executar enquanto forem criadas threads de prioridade maior.

Uma abordagem para que esse tipo de situação não ocorra, no caso uma thread não sofra *starvation*, seria utilizar técnicas aprendidas anteriormente em Sistemas Operacionais I como envelhecimento, onde a prioridade de uma thread aumenta a medida que o tempo passa e esta thread não é chamada pelo escalonador.

3. Você poderia implementar outras políticas usando esta estratégia?

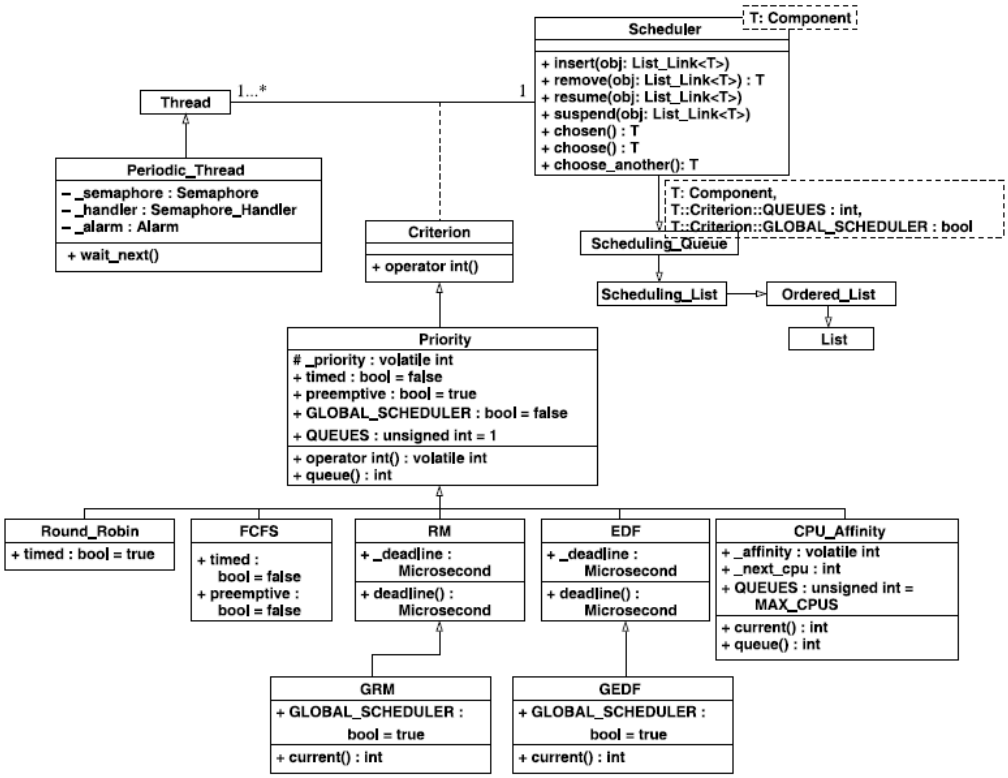
Sim. Outras políticas de escalonamento possíveis seriam:

● Shortest Job First: Processos menores são escalonados primeiro. Apenas a ordem de escalonamento seria invertida.

● Loteria: Cada processo recebe prioridades(fichas) numeradas aleatórias. A CPU escalona a thread cuja ficha foi sorteada com mais alta prioridade e esta tem direito a usar a CPU.

Implementação:

A ideia de implementação é separar as funções de thread das funções de escalonamento. Para implementar as funções de escalonamento precisamos criar uma classe scheduler, onde serão isoladas as funcionalidades e estruturas do escalonador. Para possibilitar o isolamento da política de escalomento, necessita-se criar abstrações de critérios e prioridades de uma política desejada. Assim conseguimos desacoplar implementações de distintas politicas do escalonador propriamente dito, como mostra o diagrma UML abaixo:



Priority

A classe associativa Criterion permite o desacoplamento da politica de escalonamento do escalonador, ao exigir a sobreescrita do operador int(), permite a

implementação de qualquer politica de escalonamento para uma versão em fila única, pois ao concentrar todos os dados necessários para a ordenação da fila em um único inteiro, permite a ordenação com fila ordenada.

Uma das restrições, devido à classe associativa Criterion, é que todos os critérios de escalonamento, ou políticas, devem definir um operador int, garantindo assim o retorno correto de escalonamento para ordenação da fila de escalonamento. Isto é feito com o uso de um namespace para declarar as políticas:

```
namespace Scheduling_Criteria {
	class Priority {
	public:
		enum {
			MAIN   = 0,
			HIGH   = 1,
			NORMAL = (unsigned(1) << (sizeof(int) * 8 - 1)) - 4,
			LOW    = (unsigned(1) << (sizeof(int) * 8 - 1)) - 3,
			IDLE   = (unsigned(1) << (sizeof(int) * 8 - 1)) - 2
		};

		static const bool timed = false;
		static const bool preemptive = true;
	public:
		Priority(int p = NORMAL): _priority(p) {}
		operator const volatile int() const volatile { return _priority; }
	protected:
		volatile int _priority;
	};

	class RR: public Priority {
	public:
		enum {
			MAIN   = 0,
			NORMAL = 1,
			IDLE   = (unsigned(1) << (sizeof(int) * 8 - 1)) - 2
		};
		static const bool timed = true;
		static const bool preemptive = true;
	public:
		RR(int p = NORMAL): Priority(p) {}
```

```
        };

        class FCFS: public Priority {
        public:
                enum {
                        MAIN   = 0,
                        NORMAL = 1,
                        IDLE   = (unsigned(1) << (sizeof(int) * 8 - 1)) - 2
                };

                static const bool timed = false;
                static const bool preemptive = false;

        public:
                FCFS(int p = NORMAL) : Priority((p == IDLE) ? IDLE : TSC::time_stamp()) {}
        };
}
```

De forma simplista, namespaces são usados a fim de que não haja “colisões” no uso de nomes no código. No caso, a Priority declarada dentro do namespace como uma classe, pode vir a ser uma função ou um método em outro namespace. Isso também indica que qualquer código declarado dentro de um namespace pertence ao namespace e precisará ser chamado sempre que se queira definir algo sobre ele.

Em Priority são declarados os enumeradores que servirão de retorno do critério de escalonamento, extraídos do header thread.h. São declarados também padrões relativos à política de escalonamento (se será utilizado quantum fixo para preempção e se a política adota ou não preempção).

Caso não seja dado um valor de prioridade da thread é assumido o valor “NORMAL” de prioridade a ela no construtor. E como cada thread terá sua _priority, faz-se necessário declará-la volátil.

Um fator importante vem do valor que será passado para Criterion:

```
        operator const volatile int() const volatile { return _priority; }
```

A primeira const nos diz que o valor de retorno é constante, não podemos mudá-lo. O segundo const, após o int() significa que esta função não pode alterar a variável e não pode chamar uma função não constante. O primeiro volatile nos diz que este objeto está ligado a recursos externos e, por isso, quando for escrito sobre ele, não

se deve reordenar a escrita com relação a outras leituras ou escritas sobre este mesmo operador. O segundo volatile indica que a função só pode chamar outras funções voláteis, que no caso seria:

```
volatile int _priority;
```

São declaradas também duas classes de políticas de escalonamento, contendo enums com menos valores, visto que não há necessidade de mais prioridades que as definidas.

No caso, a prioridade MAIN é a maior prioridade que uma thread pode ter, será escalonada como a primeira na fila de threads. IDLE sempre ficará no fim da fila e as threads subsequentes criadas sempre terão a mesma prioridade para RR e FCFS.

Em FCFS é adicionado um timestamp para a thread para que o sistema saiba em qual momento ela foi criada e, assim, seja adicionada na fila de acordo com este valor. IDLE sempre será a última da fila.

Scheduling_Queue

Seguindo a estrutura criada no EPOS, a Scheduling_Queue é uma especialização da Scheduling_List existente no header *list.h*. A Scheduling_Queue usa template pois permite que, com isso, seja possível criar uma fila de escalonamento com qualquer tipo de objeto do sistema que o usuário queira, desde que este objeto possua um critério de escalonamento, para que este critério seja usado como rank para posicionamento na fila.

```
template <typename T, typename R = typename T::Criterion>
class Scheduling_Queue: public Scheduling_List<T> {};
```

Scheduler

O Scheduler nada mais é que uma lista e, por este motivo, estende a classe Scheduling_Queue, sendo necessário apenas o objeto como parâmetro para que se possa executar o escalonamento.

São definidos dentro escalonador (typedef) a Scheduling_Queue, a Scheduling_List, o Criterion e a Queue para que seja possível utilizar e redefinir os métodos das listas dentro do próprio Scheduler, caso contrário isso não é possível. É criado um método schedulables para que seja possível saber quantas threads estão na lista.

```
template <typename T>
class Scheduler: public Scheduling_Queue<T> {
private:
	typedef Scheduling_Queue<T> Base;
public:
	typedef typename T::Criterion Criterion;
	typedef Scheduling_List<T, Criterion> Queue;
	typedef typename Queue::Element Element;
public:
	Scheduler() {}
	unsigned int schedulables() { return Base::size(); }
	T * volatile chosen() {
	//	return const_cast<T * volatile>((Base::chosen()) ? Base::chosen()->object()
: 0);
		return const_cast<T * volatile>(Base::chosen()->object());
	}

	void insert(T * obj) {

		Base::insert(obj->link());
	}

	 T * remove(T * obj) {

		return Base::remove(obj->link()) ? obj : 0;
	}

	void suspend(T * obj) {

		Base::remove(obj->link());
	}

	void resume(T * obj) {

		Base::insert(obj->link());
	}

	T * choose() {

		T * obj = Base::choose()->object();

		return obj;
```

```
        }

        T * choose_another() {

                T * obj = Base::choose_another()->object();

                return obj;
        }

        T * choose(T * obj) {

                if(!Base::choose(obj->link()))
                        obj = 0;

                return obj;
        }
};
```

Com estas questões esclarecidas, é necessário realizar alterações agora na Thread para que use apenas o escalonador e não mais filas próprias.

Types

Em types.h foi necessário adicionar a informação relativa ao scheduling_criteria, visto que sem essa informação, o sistema não adiciona suas informações na execução do EPOS:

```
        template <typename> class Scheduler;
        namespace Scheduling_Criteria {
                class Priority;
                class FCFS;
                class RR;
        };
```

Traits

Em traits.h é feita a remoção do boolean que define a preempção do escalonador, visto que isto agora fica a cargo do valor booleano definido dentro da política de escalonamento escolhida. O nome dado para o critério escolhido de escalonamento é Criterion.

```
typedef Scheduling_Criteria::RR Criterion;
```

Thread

Em thread.h, o escalonador vira uma classe amiga para que possa ter acesso a atributos da thread que de outra forma não teria acesso.

Outra alteração foi feita quanto ao estado de preemptividade. Essa informação virá diretamente do Criterion, um novo trait de thread. Por estar definido o Scheduling_Criteria::RR, a thread sabe qual valor de preemptive virá como constante.

```
class Thread {
        friend class Init_First;
        friend class Scheduler<Thread>;
        friend class Synchronizer_Common;
        friend class Alarm;
        friend class Task;
protected:
        static const bool preemptive = Traits<Thread>::Criterion::preemptive;
```

A forma de chamada das prioridades também muda. Agora é chamada a prioridade pelo namespace e o enum de prioridade da thread dependerá do enum definido em prioridade e da política de escalonamento utilizada.

```
typedef Scheduling_Criteria::Priority Priority;
typedef Traits<Thread>::Criterion Criterion;
enum {
        HIGH    = Criterion::HIGH,
        NORMAL  = Criterion::NORMAL,
        LOW     = Criterion::LOW,
        MAIN    = Criterion::MAIN,
        IDLE    = Criterion::IDLE
};

struct Configuration {
        Configuration(const State & s = READY, const Criterion c = NORMAL, unsigned
int ss = STACK_SIZE)
         : state(s), priority(c), stack_size(ss) {}

         State state;
         Criterion priority;
         unsigned int stack_size;
```

```
    };

typedef Ordered_Queue<Thread, Criterion, Scheduler<Thread>::Element> Queue;
```

A fila ordenada passa a receber o critério (rank) e o elemento a ser ordenado na lista como informação adicional, visto a nova estrutura da lista ordenada.

```
static Scheduler<Thread> _scheduler;
static Thread * volatile running() { return _scheduler.chosen(); }
```

Foi criado também um escalonador em thread.h e foi alterado o método running para que o escalonador pegue a partir do método chosen a cabeça da fila, no caso, a thread que está rodando.

Thread_init

Em thread_init.cc a ordem como é iniciado o relógio do escalonador é invertida (ela é colocada antes da criação das threads). Isso se deve ao fato de que a criação das threads pode gerar um reescalonamento das threads e, com isso, gerar um reset do timer, visto que o tempo calculado do quantum precisa do timer e este é resetado sempre que uma nova thread é chamada para executar.

```
if(Criterion::timed)
        _timer = new (SYSTEM) Scheduler_Timer(QUANTUM, time_slicer);
```