Universidade Federal de Santa Catarina
Centro Tecnológico
Departamento de Informática e Estatística
INE5424 - Sistemas Operacionais II
Professor: Antônio Augusto Medeiros Fröhlich
Estagiário de Docência: Mateus Krepsky Ludwich
Grupo: 01
Alunos: Alisson Granemann Abreu matrícula: 11100854
Maria Eloísa Costa matrícula: 10200630
Osvaldo Edmundo Schwerz da Rocha matrícula: 10103132

E1: Blocking Thread Synchronization

A proposta do trabalho é alterar a aplicação philosopher's dinner e a classe synchronizer.h a fim de que, ao fazer a chamada sleep sobre uma thread, a mesma não execute um yield, mas seja suspensa, até que uma chamada wakeup seja efetuada para que a thread retome sua execução.

A primeira alteração solicitada foi a alteração da mesa dos filósofos de mutex para semáforo, alterando também lock() e unlock para p() e v() respectivamente. Houve também a adição de um delay antes que as threads dos filósofos começassem a execução. Essa alteração foi feita antes do join(), no entanto, mesmo efetuando a alteração, a aplicação não apresentou erros, como acreditamos que fosse o esperado.

Linha 91 → `Delay init(1000000);`

A segunda alteração solicitada estava na classe synchronizer, para que as threads não mais utilizassem o método yield na chamada do método sleep, mas que fosse encontrada uma outra solução. Primeiro de tudo, deve-se pensar na questão de acesso a sessão crítica. Alguns critérios devem ser atendidos ao trabalharmos com esta abordagem:

Espera limitada: Todas as threads devem, em algum momento da execução, ter acesso ao recurso solicitado;

Exclusão mútua: somente uma thread pode ter acesso a seção crítica por vez;

Independência de outras tarefas: threads que não queiram acesso a seção crítica não devem impedir que as threads que queiram tenham acesso.

A garantia que mais de uma thread não vai acessar os recursos do synchronizer é feita pelo uso das operações atômicas disponibilizadas: begin_atomic() e end_atomic(). Estas foram colocadas na classe semáforo, antes de chamar os métodos sleep() e wakeup() (em vermelho).

```
void Semaphore::p()
{
      db<Synchronizer>(TRC)   <<   "Semaphore::p(this="   <<   this   <<
",value=" << _value << ")" << endl;
```

```
            begin_atomic();
            fdec(_value);
            if(_value < 0)
                  sleep();
            else
                  end_atomic();
      }

      void Semaphore::v()
      {
            db<Synchronizer>(TRC)   <<   "Semaphore::v(this="   <<   this   <<
",value=" << _value << ")" << endl;

            begin_atomic();
            finc(_value);
            if(_value < 1)
                  wakeup();
            else
                  end_atomic();
      }
```

O uso da chamada suspend() em sleep() (azul) e resume() em wakeup() e wakeup_all (verde) fazem o tratamento das threads com o uso de filas, garantindo que as threads da fila _suspended serão "acordadas" na ordem de espera da fila e colocadas nesta ordem posteriormente na fila _ready para voltarem à execução.

```
      void sleep()
      {
            Thread::running()->suspend();
      } // implicit unlock()
      void wakeup()
      {
            if(!Thread::_suspended.empty())
            {
                  Thread::_suspended.head()->object()->resume();
            }
      }
      void wakeup_all()
      {
            while(!Thread::_suspended.empty())
            {
                  Thread::_suspended.head()->object()->resume();
            }
      }
```

Não foi implementado um wakeup aleatório ou utilizado talvez o método search para pegar uma thread específica da fila _suspended, pois desta forma não seria garantida justiça de acesso à seção crítica, nem a ausência de starvation. Pegando-se sempre a thread que está como cabeça da fila, garante-se que todas as threads em algum momento executarão.

Na classe semáforo, foi feita uma alteração no método p(), visto que não há mais o uso do método yield() para fazer as threads "dormirem". Quando era utilizado o método yield(), as

threads não eram suspensas, passavam apenas do estado running para ready. Por causa disso, o valor precisava ser verificado toda vez. Pra que isso não ocorresse mais, o método foi alterado de while para apenas um if:

```
void Semaphore::p()
{
	db<Synchronizer>(TRC)	<<	"Semaphore::p(this="	<<	this	<<
",value=" << _value << ")" << endl;

	begin_atomic();
	fdec(_value);
	if(_value < 0)
		sleep();
	else
		end_atomic();
}
```

Na classe Synchronizer também foi feito um destrutor para o mesmo, visto que o semáforo é um tipo de Synchronizer_Common e este não possuía destrutor explícito. Todos os recursos precisam ser liberados ao final da execução do programa para que não fiquem recursos "pendurados" no sistema, ocupando espaço de memória indefinidamente. O Synchronizer apenas acorda todas as threads que estão ainda, de alguma forma, presas na fila _suspended para que terminem sua execução e sejam finalizadas.

```
~Synchronizer_Common()
{
	begin_atomic();
	wakeup_all();
	end_atomic();
}
```

Na tentativa de resolver o problema, foi pensado na criação de uma nova lista na classe Synchronizer, para armazenar as threads que seriam suspensas e, posteriormente acordadas. Porém após análise mais detalhada do código, vimos que não havia necessidade, visto que a classe thread já disponibilizava de duas filas de controle, sendo mais fácil utilizá-las para a resolução.