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

E2: Blocking Thread Joining

Para este novo exercício, foi proposto a alteração do método join para que não mais fosse usado o método yield, mas que a thread fosse suspensa até a thread a qual foi solicitado join terminasse de executar.

Para tanto, existem duas formas de tentar resolver o exercício:

1 - Criar uma nova fila para cada Thread, Queue _joining, alocando as threads que tentem join() nesta fila e assim sabermos quais threads estão esperando para poderem executar.

2 - Criar um atributo Thread * _join_wait que armazena referências para a thread pela qual possa se estar esperando.

O primeiro caso utiliza mais memória para armazenamento das filas criadas e necessita de um grande gerenciamento destas. Há na necessidade de varredura das listas sempre que uma thread é deletada ou finaliza normalmente no sistema. Já o segundo caso diminui o consumo de memória, porém precisa de maiores cuidados quanto a referências e a threads que estejam esperando por uma thread finalizada.

Fizemos os dois casos como forma de experimento. A ideia principal é similar nos dois casos. Se fôssemos escolher um dos dois casos em um sistema operacional, optaríamos pelo caso dois, pela questão do uso de memória.

CASO 1: Uso de Filas

Foi criado um atributo novo em Thread.h com o nome de _joining (em vermelho). Esta fila será responsável por armazenar threads que queiram dar join em uma thread específica.

```
protected:
    char * _stack;
    Context * volatile _context;
    volatile State _state;
    Queue * _waiting;
    Queue::Element _link;
    Queue _joining;

    static Scheduler_Timer * _timer;
```

O segundo passo é a alteração do método join em Thread.cc. Algumas considerações precisam ser verificadas antes de mais nada. Uma thread não deve em nenhum momento tentar dar join nela mesma (em vermelho), pois isso fará com que o sistema fique "pendurado" esperando a thread voltar a rodar. Além disso, não será permitido que uma thread tente fazer mais de um join em outra thread (em azul), para não nos perdermos o controle de referências de uma thread para outra, nem deixarmos, por algum descuido, alguma referência nula em algum momento no sistema. Garantindo essas duas condições, troca-se o while por um if, visto que não utilizaremos mais o método yield e a verificação não precisará mais ser feita indefinidamente. Se uma Thread A tenta chamar join() em uma Thread B, ela verifica se a B já terminou de executar. Se sim, nada é feito. Caso contrário, A será inserida nesta fila _joining e será suspensa até que B termine de executar (em verde).

```
int Thread::join()
{
    lock();
```

```
    db<Thread>(TRC) << "Thread::join(this=" << this << ",state=" <<
_state << ")" << endl;

        assert(_running != this);
        assert(!_joining);

        if(_state != FINISHING) {
            //yield(); // implicit unlock() Old code
              _joining.insert(&_running->_link);
              _running->suspend();
        }

        unlock();

    return *reinterpret_cast<int *>(_stack);
}
```

Outros dois cuidados precisam ser tomados a partir deste ponto. Quando uma thread terminar de executar, seja porque finalizou, seja porque foi deletada, todas as threads que tinham dado join precisam voltar a sua execução normal. Para tanto, duas outras alterações foram feitas. Uma no destrutor da thread e uma no método exit.

No destrutor, foi feito uma verificação da fila da respectiva thread. Caso a fila não esteja vazia, a referência será removida da lista de _joining e _suspended (visto que ela estará nesta fila também quando for chamado o método suspend() em join()), seu estado será retornado como pronta para executar e será inserida na fila de threads prontas, para ser chamada pelo escalonador quando puder.

```
while(!_joining.empty()) {
        Thread * t = _joining.remove()->object();
        _suspended.remove(&t->_link);
        t->_state = READY;
        _ready.insert(&t->_link);
}
```

Já no método exit, como queremos trabalhar com o _joining específico de uma dada thread, cria-se uma fila com a referência da fila da thread _running e é com ela que trabalhamos. O mesmo que foi feito no destrutor da classe é feito no método exit (em vermelho), antes da verificação da lista de prontos, visto que posteriormente é feito um tratamento com esta fila e uma nova thread é chamada para executar logo em seguida.

```
        void Thread::exit(int status)
        {
            lock();

            db<Thread>(TRC) << "Thread::wakeup_all(running=" << running() << ")"
<< endl;

            while(_ready.empty() && !_suspended.empty())
                idle(); // implicit unlock();

            lock();

            Queue * queue = &_running->_joining;

            while(!queue->empty()){
                 Thread * t = queue->remove()->object();
                 _suspended.remove(&t->_link);
                 t->_state = READY;
                 _ready.insert(&t->_link);
            }

            if(!_ready.empty()) {
                Thread * prev = _running;
                prev->_state = FINISHING;
                *reinterpret_cast<int *>(prev->_stack) = status;

                _running = _ready.remove()->object();
                _running->_state = RUNNING;

                dispatch(prev, _running);
```

```
            } else {
                db<Thread>(WRN) << "The last thread in the system has exited!"
<< endl;
                if(reboot) {
                    db<Thread>(WRN) << "Rebooting the machine ..." << endl;
                    Machine::reboot();
                } else {
                    db<Thread>(WRN) << "Halting the CPU ..." << endl;
                    CPU::halt();
                }
            }

            unlock();
        }
```

CASO 2: Uso de um ponteiro Thread

Criamos um atributo novo na classe Thread.h com o nome Thread * _join_wait (em vermelho). Foi utilizado volatile pois várias threads utilizarão este atributo e não queremos que essa referência seja perdida por possíveis alterações.

```
    protected:
        char * _stack;
        Context * volatile _context;
        volatile State _state;
        Queue * _waiting;
        Queue::Element _link;
        Thread * volatile _join_wait;

        static Scheduler_Timer * _timer;
```

O segundo passo é a alteração do método join em Thread.cc, de forma similar ao que foi feito pro caso 1. As mesmas considerações precisam ser levantadas aqui. Uma thread não deve em nenhum momento tentar dar join nela mesma (em vermelho), assim como não será permitido que uma thread tente fazer mais de um join em outra thread (em azul). Garantindo essas duas condições, troca-se o while por um if, visto que não utilizaremos mais o método yield e a

verificação não precisará mais ser feita indefinidamente. Se uma Thread A tenta chamar join() em uma Thread B, ela verifica se a B já terminou de executar. Se sim, nada é feito. Caso contrário, A apontará para o endereço de B e será suspensa (em verde).

```
int Thread::join()
{
    lock();

    db<Thread>(TRC) << "Thread::join(this=" << this << ",state=" << _state << ")" << endl;
       assert(_running != this);
       assert(!_join_wait);

       if(_state != FINISHING) {
           //yield(); // implicit unlock()
             _running->_join_wait = this;
             _running->suspend();
       }

       unlock();

    return *reinterpret_cast<int *>(_stack);
}
```

Os mesmos cuidados com finalização ou destruição de thread foram tomados para esta solução também.

No destrutor, foi feito uma verificação da referência existente entre a thread e a join. Caso a referência ainda exista, será feito o tratamento para remover a thread joined de dentro da fila de suspensas, alteração do seu estado para ready e inserção da thread joined na fila de prontas para que o escalonador a chame quando puder para que execute.

```
if(_running->_join_wait) {
       Thread * t = _running->_join_wait;
       _suspended.remove(&t->_link);
       t->_state = READY;
```

```
        _ready.insert(&t->_link);
        _running->_join_wait = 0;
    }
```

Para o método exit foi feito o mesmo tratamento do destrutor da classe (em vermelho), antes da verificação da lista de prontos, visto que posteriormente é feito um tratamento com esta fila e uma nova thread é chamada para executar logo em seguida.

```
void Thread::exit(int status)
{
    lock();

    db<Thread>(TRC) << "Thread::wakeup_all(running=" << running() << ")"
<< endl;

    while(_ready.empty() && !_suspended.empty())
        idle(); // implicit unlock();

    lock();

    if(_running->_join_wait) {
        Thread * t = _running->_join_wait;
        _suspended.remove(&t->_link);
        t->_state = READY;
        _ready.insert(&t->_link);
        _running->_join_wait = 0;
    }

    if(!_ready.empty()) {
        Thread * prev = _running;
        prev->_state = FINISHING;
        *reinterpret_cast<int *>(prev->_stack) = status;

        _running = _ready.remove()->object();
        _running->_state = RUNNING;

        dispatch(prev, _running);
```

```
    } else {
        db<Thread>(WRN) << "The last thread in the system has exited!"
<< endl;
        if(reboot) {
            db<Thread>(WRN) << "Rebooting the machine ..." << endl;
            Machine::reboot();
        } else {
            db<Thread>(WRN) << "Halting the CPU ..." << endl;
            CPU::halt();
        }
    }

    unlock();
}
```