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

E3: Idle Thread

To do

Modify the implementation of idle so it becomes a thread that is only scheduled when there are no other threads ready to run. This will eliminate bugs such as having the timer handler undesirably waking up suspended threads.

Done

Afim de que o sistema não utilizasse mais o timer como recurso para que as threads em suspenso ou na fila de pronto fossem novamente escalonadas pelo sistema, foi inicialmente criado uma thread idle (em vermelho) em thread_init.cc:

```
void Thread::init()
{
    int (* entry)() = reinterpret_cast<int (*)()>(__epos_app_entry);

    db<Init, Thread>(TRC) << "Thread::init(entry=" << reinterpret_cast<void
*>(entry) << ")" << endl;
```

```
        _running = new (kmalloc(sizeof(Thread))) Thread(Configuration(RUNNING,
NORMAL), entry);
        new (kmalloc(sizeof(Thread))) Thread(Configuration(READY, IDLE), &idle);

    if(preemptive)
        _timer = new (kmalloc(sizeof(Scheduler_Timer)))
Scheduler_Timer(QUANTUM, time_slicer);

    db<Init, Thread>(INF) << "Dispatching the first thread: " << _running <<
endl;

    This_Thread::not_booting();

    _running->_context->load();
}
```

A thread _running é criada antes da thread idle para que não haja escalonamento implícito da thread idle devido a falta de threads rodando no sistema. A thread idle foi criada com valor &idle no lugar de entry, visto que somente esta thread agora chamará o método idle.

Foi criada uma nova prioridade para a thread idle. Colocamos um valor maior que 32, pois permite que novas prioridades sejam definidas posteriormente, caso necessário.

```
// Thread Priority
    typedef unsigned int Priority;
    enum {
        HIGH = 0,
        NORMAL = 15,
        LOW = 31,
        IDLE = 42
    };
```

Além disso, alguns métodos no arquivo thread.cc tiveram de ser alterados visto que a verificação na fila de prontos não mais precisaria ser feita, já que idle estaria sempre pronta para rodar, caso nenhuma outra thread estiver disponível no sistema.

No construtor foi adicionada uma verificação quanto às threads que são adicionadas ao sistema (em vermelho). Caso o escalonador seja preemptivo, o estado da thread seja "pronto" e, muito importante, a thread não tenha o rank IDLE (prioridade), haverá reescalonamento das threads. Esta verificação garante que mesmo a thread idle seja criada em um sistema com escalonamento preemptivo, ela não será chamada para rodar no lugar da thread main.

```
void Thread::constructor(const Log_Addr & entry, unsigned int stack_size)
{
    db<Thread>(TRC) << "Thread(entry=" << entry
                    << ",state=" << _state
                    << ",priority=" << _link.rank()
                    << ",stack={b=" << _stack
                    << ",s=" << stack_size
                    << "},context={b=" << _context
                    << "," << *_context << "}) => " << this << endl;

    switch(_state) {
        case RUNNING: break;
        case SUSPENDED: _suspended.insert(&_link); break;
        default: _ready.insert(&_link);
    }

      if(preemptive && (_state == READY) && (_link.rank() != IDLE))
            reschedule();
      else
            unlock();
}
```

Nos métodos em que havia verificação da fila de threads prontas, o código foi comentado para que não mais fosse chamada a verificação, como acontece em suspend, yield, sleep e exit (em vermelho). Em suspend, a fila de pronto nunca estará vazia, então o método idle não precisa ser chamado.

```
void Thread::suspend()
{
```

```
    lock();

    db<Thread>(TRC) << "Thread::suspend(this=" << this << ")" << endl;

    if(_running != this)
        _ready.remove(this);

    _state = SUSPENDED;
    _suspended.insert(&_link);

    if(_running == this) {
       // while(_ready.empty())
       //     idle();

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

        dispatch(this, _running);
    }

    unlock();
}
```

Em yield, há o escalonamento de threads caso a fila de ready não esteja vazio. Como a fila nunca fica vazia, esta verificação não precisa mais ser efetuada.

```
void Thread::yield()
{
    lock();

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

    //if(!_ready.empty()) {
        Thread * prev = _running;
        prev->_state = READY;
        _ready.insert(&prev->_link);
```

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

            dispatch(prev, _running);
        //} else
        //    idle();

        unlock();
    }
```

Em sleep, assim como em suspend, a fila de prontos nunca ficará vazia, então a verificação não precisa mais ser efetuada.

```
    void Thread::sleep(Queue * q)
    {
        db<Thread>(TRC) << "Thread::sleep(running=" << running() << ",q=" << q <<
")" << endl;

        // lock() must be called before entering this method
        assert(locked());

        //while(_ready.empty())
        //    idle();

        Thread * prev = running();
        prev->_state = WAITING;
        prev->_waiting = q;
        q->insert(&prev->_link);

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

        dispatch(prev, _running);

        unlock();
```

```
}
```

Em exit foi necessário um cuidado maior, visto que a thread idle não chama o método exit. Por isso, parte do código em exit foi removido (a verificação relativa à fila de prontos e todo o tratamento para desligamento da CPU para economia de energia, mantendo-se somente o reescalonamento das threads remanescentes no sistema. Este código de desligamento foi realocado para o método idle, visto que a CPU só será realmente desligada se não tivermos mais threads ativas.

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

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

    Thread * prev = _running;
    prev->_state = FINISHING;
    *reinterpret_cast<int *>(prev->_stack) = status;

      if(prev->_joining) {
        prev->_joining->resume();
        prev->_joining = 0;
    }

    lock();

    //if(_ready.empty()) {
    //    if(!_suspended.empty()) {
    //        while(_ready.empty())
    //            idle(); // implicit unlock();
    //        lock();
    //    } else { // _ready.empty() && _suspended.empty()
    //             db<Thread>(WRN) << "The last thread in the system has
exited!\n";
    //        if(reboot) {
    //            db<Thread>(WRN) << "Rebooting the machine ...\n";
```

```
        //            Machine::reboot();
        //        } else {
        //            db<Thread>(WRN) << "Halting the CPU ...\n";
        //            CPU::halt();
        //        }
        //    }
        //} else {
            _running = _ready.remove()->object();
            _running->_state = RUNNING;

            dispatch(prev, _running);
        //}

        unlock();
    }
```

No método idle, se a fila de suspensos não estiver vazia, não é possível desligar o sistema. Enquanto isso for verdade e somente a thread idle estiver na fila de threads prontas, as interrupções da CPU são reativadas e é colocada em modo de economia de energia. Se a fila de suspensas estiver vazia, muito possivelmente todas as threads terminaram de rodar. Pode-se optar por reboot da máquina ou colocar a CPU para dormir. Se a CPU for colocada para dormir, ainda há chance de novas threads entrarem no sistema. Este é o único caso onde a fila de prontos é verificada novamente, para garantir que a thread idle não está mais sozinha no sistema e, com isso, a nova thread é chamada a partir do método yield.

```
int Thread::idle()
{
	while(true) {
		db<Thread>(TRC) << "Thread::idle()" << endl;
		CPU::int_disable();

		if(!_suspended.empty()) {
			db<Thread>(INF) << "There are no runnable threads at the moment!" << endl;
			CPU::int_enable();
```

```
                db<Thread>(INF) << "Halting the CPU ..." << endl;
                CPU::halt();
            } else {
                db<Thread>(WRN) << "The last thread in the system has
exited!\n";
                if(reboot) {
                   db<Thread>(WRN) << "Rebooting the machine ...\n";
                   Machine::reboot();
                } else {
                   db<Thread>(WRN) << "Halting the CPU ...\n";
                   CPU::halt();
                   if (_ready.size() > 1) {
                      yield();
                   }
                }
            }
        }

        return 0;
    }
```