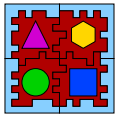


Embedded Parallel Operating System

Coding Journey through OS Design
–from co-routines to a multicore kernel–

Prof. Antônio Augusto Fröhlich, Ph.D.

UFSC / LISHA

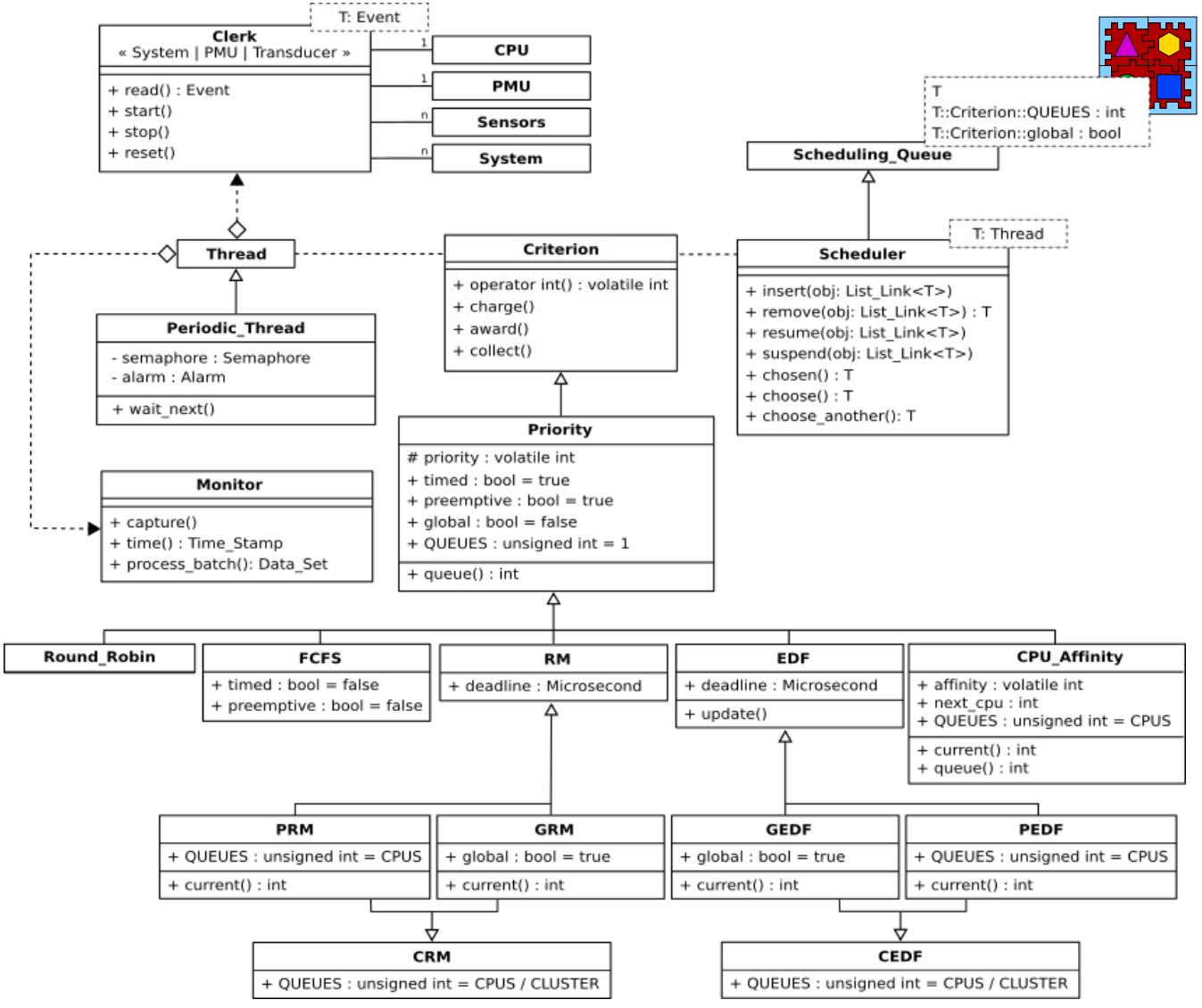
July 12, 2021

Scheduling

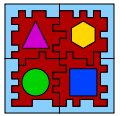
Decoupling Scheduling from Thread

Flexible architecture

- Framework
- Powerful List
- Policies mapped into ordering



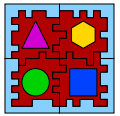
```
template<typename T, typename R = Rank>
class Ranked
{
public:
    typedef T Object_Type;
    typedef R Rank_Type;
    typedef Ranked Element;

public:
    Ranked(const T * o, const R & r = 0):
        _object(o), _rank(r) {}

    T * object() const {
        return const_cast<T *>(_object); }

    const R & rank() const { return _rank; }
    const R & key() const { return _rank; }
    void rank(const R & r) { _rank = r; }
    int promote(const R & n = 1) {
        _rank -= n; return _rank; }
    int demote(const R & n = 1) {
    _rank += n; return _rank; }

private:
    const T * _object;
    R _rank;
};
```



```
template<typename T, typename R = Rank>
class Doubly_Linked_Ordered
{
public:
    typedef T Object_Type;
    typedef Rank Rank_Type;
    typedef Doubly_Linked_Ordered Element;

public:
    Doubly_Linked_Ordered(const T * o,
        const R & r = 0):
    _object(o), _rank(r), _prev(0), _next(0) {}

    T * object() const {
        return const_cast<T *>(_object); }

    Element * prev() const { return _prev; }
    Element * next() const { return _next; }
    void prev(Element * e) { _prev = e; }
    void next(Element * e) { _next = e; }

    const R & rank() const { return _rank; }
    void rank(const R & r) { _rank = r; }

private:
    const T * _object;
    R _rank;
    Element * _prev;
    Element * _next;
};
```

```
template<typename T, typename R = Rank>
class Ranked
{
public:
    typedef T Object_Type;
    typedef R Rank_Type;
    typedef Ranked Element;

public:
    Ranked(const T * o, const R & r = 0):
        _object(o), _rank(r) {}

    T * object() const {
        return const_cast<T *>(_object); }

    const R & rank() const { return _rank; }
    const R & key() const { return _rank; }
    void rank(const R & r) { _rank = r; }
    int promote(const R & n = 1) {
        _rank -= n; return _rank; }
    int demote(const R & n = 1) {
    _rank += n; return _rank; }

private:
    const T * _object;
    R _rank;
};
```

list order



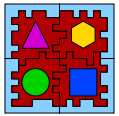
```
template<typename T, typename R = Rank>
cla
{
public:
    typedef T Object_Type;
    typedef Rank Rank_Type;
    typedef Doubly_Linked_Ordered Element;

public:
    Doubly_Linked_Ordered(const T * o,
        const R & r = 0):
    _object(o), _rank(r), _prev(0), _next(0) {}

    T * object() const {
        return const_cast<T *>(_object); }

    Element * prev() const { return _prev; }
    Element * next() const { return _next; }
    void prev(Element * e) { _prev = e; }
    void next(Element * e) { _next = e; }

    const R & rank() const { return _rank; }
    void rank(const R & r) { _rank = r; }

private:
    const T * _object;
    R _rank;
    Element * _prev;
    Element * _next;
};
```

list ordering

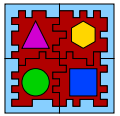
```
{
public:
    typedef T Object_Type;
    typedef R Rank_Type;
    typedef Ranked Element;

public:
    Ranked(const T * o, const R & r = 0):
        _object(o), _rank(r) {}

    T * object() const {
        return const_cast<T *>(_object); }

    const R & rank() const { return _rank; }
    const R & key() const { return _rank; }
    void rank(const R & r) { _rank = r; }
    int promote(const R & n = 1) {
        _rank -= n; return _rank; }
    int demote(const R & n = 1) {
    _rank += n; return _rank; }

private:
    const T * _object;
    R _rank;
};
```

Rank>

```
template<typename T, typename R = Rank>
class Doubly_Linked_Ordered
{
public:
    typedef T Object_Type;
    typedef Rank Rank_Type;
    typedef Doubly_Linked_Ordered Element;

public:
    Doubly_Linked_Ordered(const T * o,
        const R & r = 0):
    _object(o), _rank(r), _prev(0), _next(0) {}

    T * object() const {
        return const_cast<T *>(_object); }

    Element * prev() const { return _prev; }
    Element * next() const { return _next; }
    void prev(Element * e) { _prev = e; }
    void next(Element * e) { _next = e; }

    const R & rank() const { return _rank; }
    void rank(const R & r) { _rank = r; }

private:
    const T * _object;
    R _rank;
    Element * _prev;
    Element * _next;
};
```



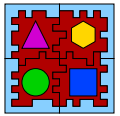
```
class Priority
{
public:
    enum : int {
        MAIN   = 0,
        HIGH   = 1,
        NORMAL = (unsigned(1)<<(sizeof(int)*8-1))-3,
        LOW    = (unsigned(1)<<(sizeof(int)*8-1))-2,
        IDLE   = (unsigned(1)<<(sizeof(int)*8-1))-1
    };

    enum {
        PERIODIC    = HIGH,
        APERIODIC   = NORMAL,
        SPORADIC    = NORMAL
    };

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

public:
    template <typename ... Tn>
    Priority(int p = NORMAL, Tn & ... an):
        _priority(p) {}
    operator const volatile int() const volatile {
        return _priority; }

    protected:
        volatile int _priority;
    };
```



```
class RR: public Priority
{
public:
    static const bool timed = true;
    static const bool dynamic = false;
    static const bool preemptive = true;

public:
    template <typename ... Tn>
    RR(int p = NORMAL, Tn & ... an): Priority(p) {}
};
```

```
class FCFS: public Priority
{
public:
    static const bool timed = false;
    static const bool dynamic = false;
    static const bool preemptive = false;

public:
    template <typename ... Tn>
    FCFS(int p = NORMAL, Tn & ...an): Priority(p){}
};
```

Priority-based polices



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

    enum {
        PERIODIC    = HIGH,
        APERIODIC   = NORMAL,
        SPORADIC    = NORMAL
    };

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

public:
    template <typename ... Tn>
    Priority(int p = NORMAL, Tn & ... an):
        _priority(p) {}
    operator const volatile int() const volatile {
        return _priority; }

    protected:
        volatile int _priority;
    };
```

```
class
{
public:
    static const bool timed = true;
    static const bool dynamic = false;
    static const bool preemptive = true;

public:
    template <typename ... Tn>
    RR(int p = NORMAL, Tn & ... an): Priority(p) {}
};
```

```
class FCFS: public Priority
{
public:
    static const bool timed = false;
    static const bool dynamic = false;
    static const bool preemptive = false;

public:
    template <typename ... Tn>
    FCFS(int p = NORMAL, Tn & ...an): Priority(p){}
};
```

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

    enum {
        PERIODIC    = HIGH,
        APERIODIC   = NORMAL,
        SPORADIC    = NORMAL
    };

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

public:
    template <typename ... Tn>
    Priority(int p = NORMAL, Tn & ... an):
        _priority(p) {}
    operator const volatile int() const volatile {
        return _priority; }

    protected:
        volatile int _priority;
    };
```

priorities

```
class RR: public Priority
{
public:
    static const
    static const bool dynamic = false;
    static const bool preemptive = true;

public:
    template <typename ... Tn>
    RR(int p = NORMAL, Tn & ... an): Priority(p) {}
};
```

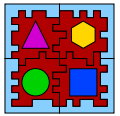
```
class FCFS: public Priority
{
public:
    static const bool timed = false;
    static const bool dynamic = false;
    static const bool preemptive = false;

public:
    template <typename ... Tn>
    FCFS(int p = NORMAL, Tn & ...an): Priority(p){}
};
```

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

    enum {
        PERIODIC    = HIGH,
        APERIODIC   = NORMAL,
        SPORADIC    = NORMAL
    };

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

public:
    template <typename ... Tn>
    Priority(int p = NORMAL, Tn & ... an):
        _priority(p) {}
    operator const volatile int() const volatile {
        return _priority; }

    protected:
        volatile int _priority;
    };
```

```
class RR: public Priority
{
public:
    static const bool timed = true;
    static const bool dynamic = false;
    static const bool preemptive = true;

public:
    template <typename ... Tn>
                                    .. an): Priority(p) {}
};
```

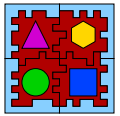
policy types

```
class FCFS: public Priority
{
public:
    static const bool timed = false;
    static const bool dynamic = false;
    static const bool preemptive = false;

public:
    template <typename ... Tn>
    FCFS(int p = NORMAL, Tn & ...an): Priority(p){}
};
```



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

    enum {
        PERIODIC    = HIGH,
        APERIODIC   = NORMAL,
        SPORADIC    = NORMAL
    };

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

public:
    template <typename ... Tn>
    Priority(int p = NORMAL, Tn & ... an):
        _priority(p) {}
    operator const volatile int() const volatile {
        return _priority; }

    protected:
        volatile int _priority;
    };
```

```
class RR: public Priority
{
public:
    static const bool timed = true;
    static const bool dynamic = false;
    static const bool preemptive = true;

public:
    template <typename ... Tn>
    RR(int p = NORMAL, Tn & ... an): Priority(p) {}
};
```

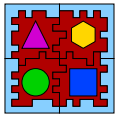


policy traits

```
class FCFS:
{
public:
    static const bool timed = false;
    static const bool dynamic = false;
    static const bool preemptive = false;

public:
    template <typename ... Tn>
    FCFS(int p = NORMAL, Tn & ...an): Priority(p){}
};
```

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

    enum {
        PERIODIC    = HIGH,
        APERIODIC   = NORMAL,
        SPORADIC    = NORMAL
    };

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

public:
    template <typename ... Tn>
    Priority(int p = NORMAL, Tn & ... an):
        _priority(p) {}
    operator const volatile int() const volatile {
        return _priority; }

    protected:
        volatile int _priority;
    };
```

```
class RR: public Priority
{
public:
    static const bool timed = true;
    static const bool dynamic = false;
    static const bool preemptive = true;

public:
    template <typename ... Tn>
    RR(int p = NORMAL, Tn & ... an): Priority(p) {}
};
```



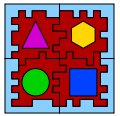
```
class FCFS: public Priority
{
public:
    static const bool timed = false;
    static const bool dynamic = false;
    static const bool preemptive = false;

public:
    tem
    FCF
};
```

scheduling criteria must define operator int() with the semantics of returning the desired order of a given object within the scheduling list (or, alternatively, declare all the ordering operators)

Round-Robin

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

    enum {
        PERIODIC    = HIGH,
        APERIODIC   = NORMAL,
        SPORADIC    = NORMAL
    };

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

public:
    template <typename ... Tn>
    Priority(int p = NORMAL, Tn & ... an):
        _priority(p) {}
    operator const volatile int() const volatile {
        return _priority; }

    protected:
        volatile int _priority;
    };
```

```
class RR: public Priority
{
public:
    static const bool timed = true;
    static const bool dynamic = false;
    static const bool preemptive = true;

public:
    template <typename ... Tn>
    RR(int p = NORMAL, Tn & ... an): Priority(p) {}
};
```



```
class FCFS: public Priority
{
public:
    static const bool timed = false;
    static const bool dynamic = false;
    static const bool preemptive = false;

public:
    template <typename ... Tn>
    FCFS(int p = NORMAL, Tn & ...an): Priority(p){}
};
```

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

    enum {
        PERIODIC    = HIGH,
        APERIODIC   = NORMAL,
        SPORADIC    = NORMAL
    };

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

public:
    template <typename ... Tn>
    Priority(int p = NORMAL, Tn & ... an):
        _priority(p) {}
    operator const volatile int() const volatile {
        return _priority; }

    protected:
        volatile int _priority;
    };
```

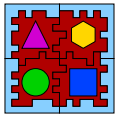
FCFS

```
class RR: public Priority
{
public:
    static const bool timed = true;
    static const bool dynamic = false;
    static const bool preemptive = true;

public:
    template <typename ... Tn>
    RR(int p = NORMAL, Tn & ... an): Priority(p) {}
};
```

```
class FCFS: public Priority
{
public:
    static const bool timed = false;
    static const bool dynamic = false;
    static const bool preemptive = false;

public:
    template <typename ... Tn>
    FCFS(int p = NORMAL, Tn & ...an): Priority(p){}
};
```

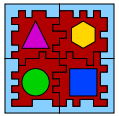
```
template<typename T, typename R = typename T::Criterion>
class Scheduling_Queue: public Scheduling_List<T> {};

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()->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() { return Base::choose()->object(); }
    T * choose_another() { return Base::choose_another()->object(); }
    T * choose(T * obj) { return Base::choose(obj->link()) ? obj : 0; }
};
```

Scheduling_Queue is the heart of EPOS scheduler



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

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()->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() { return Base::choose()->object(); }
    T * choose_another() { return Base::choose_another()->object(); }
    T * choose(T * obj) { return Base::choose(obj->link()) ? obj : 0; }
};
```

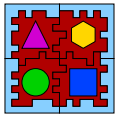
Criterion defines ordering



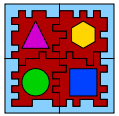
```
template<typename T, typename R = typename T::Criterion>
class Scheduling_Queue: public Scheduling_List<T> {};

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()->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() { return Base::choose()->object(); }
    T * choose_another() { return Base::choose_another()->object(); }
    T * choose(T * obj) { return Base::choose(obj->link()) ? obj : 0; }
};
```

define the queue (instead of importing it)

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

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()->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() { return Base::choose()->object(); }
    T * choose_another() { return Base::choose_another()->object(); }
    T * choose(T * obj) { return Base::choose(obj->link()) ? obj : 0; }
};
```

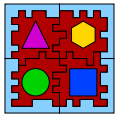


T must provide link()



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

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()->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() { return Base::choose()->object(); }
    T * choose_another() { return Base::choose_another()->object(); }
    T * choose(T * obj) { return Base::choose(obj->link()) ? obj : 0; }
};
```

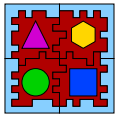
suspend() and resume() are key for hardware implementations



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

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()->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() { return Base::choose()->object(); }
    T * choose_another() { return Base::choose_another()->object(); }
    T * choose(T * obj) { return Base::choose(obj->link()) ? obj : 0; }
};
```

policy based on queue ordering

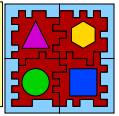
```
class Thread
{
    friend class Scheduler<Thread>;

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

public:
    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
    };

    typedef Ordered_Queue<Thread, Criterion,
            Scheduler<Thread>::Element> Queue;
protected:
    Criterion & criterion() {
        return const_cast<Criterion &>(_link.rank()); }
    Queue::Element * link() { return &_link; }

    static Thread * volatile running() {
        return _scheduler.chosen(); }
    static void dispatch(Thread * prev, Thread * next,
                         bool charge = true);
```



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

    static volatile unsigned int _thread_count;
    static Scheduler_Timer * _timer;
    static Scheduler<Thread> _scheduler;
};

void Thread::constructor_prologue(...)
{
    lock();
    _thread_count++;
    _scheduler.insert(this);
    _stack = new (SYSTEM) char[stack_size];
}

void Thread::constructor_epilogue(...)
{
    if((_state != READY) && (_state != RUNNING))
        _scheduler.suspend(this);
    if(preemptive && (_state == READY) &&
        (_link.rank() != IDLE))
        reschedule();
    else
        unlock();
}
```

```
class Thread
{
    friend class Scheduler<Thread>;

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

public:
    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
    };

    typedef Ordered_Queue<Thread, Criterion,
            Scheduler<Thread>::Element> Queue;
protected:
    Criterion & criterion() {
        return const_cast<Criterion &>(_link.rank()); }
    Queue::Element * link() { return &_link; }

    static Thread * volatile running() {
        return _scheduler.chosen(); }
    static void dispatch(Thread * prev, Thread * next,
                         bool charge = true);
```

isolated scheduler

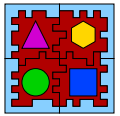


```
protected
    char
    Conte
    volatile State _state;
    Queue * _waiting;
    Thread * volatile _joining;
    Queue::Element _link;

    static volatile unsigned int _thread_count;
    static Scheduler_Timer * _timer;
    static Scheduler<Thread> _scheduler;
};

void Thread::constructor_prologue(...)
{
    lock();
    _thread_count++;
    _scheduler.insert(this);
    _stack = new (SYSTEM) char[stack_size];
}

void Thread::constructor_epilogue(...)
{
    if((_state != READY) && (_state != RUNNING))
        _scheduler.suspend(this);
    if(preemptive && (_state == READY) &&
        (_link.rank() != IDLE))
        reschedule();
    else
        unlock();
}
```

```
class Thread
{
    friend class Scheduler<Thread>;

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

public:
    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
    };

    typedef Ordered_Queue<Thread, Criterion,
            Scheduler<Thread>::Element> Queue;
protected:
    Criterion & criterion() {
        return const_cast<Criterion &>(_link.rank()); }
    Queue::Element * link() { return &_link; }

    static Thread * volatile running() {
        return _scheduler.chosen(); }
    static void dispatch(Thread * prev, Thread * next,
                         bool charge = true);
```

```
protected:
    char * _stack;
    Context * volatile _context;
    volatile State _state;
    Queue *  waiting;
    Thre
    Queue

    static volatile unsigned int _thread_count;
    static Scheduler_Timer * _timer;
    static Scheduler<Thread> _scheduler;
};

void Thread::constructor_prologue(...)
{
    lock();
    _thread_count++;
    _scheduler.insert(this);
    _stack = new (SYSTEM) char[stack_size];
}

void Thread::constructor_epilogue(...)
{
    if((_state != READY) && (_state != RUNNING))
        _scheduler.suspend(this);
    if(preemptive && (_state == READY) &&
        (_link.rank() != IDLE))
        reschedule();
    else
        unlock();
}
```

Criterion defines priorities and policy Traits



```
class Thread
{
    friend class Scheduler<Thread>;

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

public:
    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
    };

    typedef Ordered_Queue<Thread, Criterion,
            Scheduler<Thread>::Element> Queue;
protected:
    Criterion & criterion() {
        return const_cast<Criterion &>(_link.rank()); }
    Queue::Element * link() { return &_link; }

    static Thread * volatile running() {
        return _scheduler.chosen(); }
    static void dispatch(Thread * prev, Thread * next,
                         bool charge = true);
```

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

    static volatile unsigned int _thread_count;
    static Scheduler_Timer * _timer;
    static Scheduler<Thread> _scheduler;
};

void Thread::constructor_prologue(...)
{
    lock()
    _threa
    _sched
    _stack = new (SYSTEM) char[stack_size];
}

void Thread::constructor_epilogue(...)
{
    if((_state != READY) && (_state != RUNNING))
        _scheduler.suspend(this);
    if(preemptive && (_state == READY) &&
       (_link.rank() != IDLE))
        reschedule();
    else
        unlock();
}
```

Queue for uses other than scheduling, with Thread and Criterion, but Element imported for interoperability

```
class Thread
{
    friend class Scheduler<Thread>;

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

public:
    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
    };

    typedef Ordered_Queue<Thread, Criterion,
            Scheduler<Thread>::Element> Queue;
protected:
    Criterion & criterion() {
        return const_cast<Criterion &>(_link.rank()); }
    Queue::Element * link() { return &_link; }

    static Thread * volatile running() {
        return _scheduler.chosen(); }
    static void dispatch(Thread * prev, Thread * next,
                         bool charge = true);
```

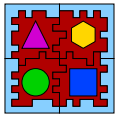
```
protected:
    char * _stack;
    Context * volatile _context;
    volatile State _state;
    Queue * _waiting;
    Thread * volatile _joining;
    Queue::Element _link;

    static volatile unsigned int _thread_count;
    static Scheduler_Timer * _timer;
    static Scheduler<Thread> _scheduler;
};

void Thread::constructor_prologue(...)
{
    lock();
    _thread_count++;
    _scheduler.insert(this);
    _stack = new (SYSTEM) char[stack_size];
}

void Thread::constructor_epilogue(...)
{
    if((_s                                  UNNING))
        _s
    if(pre                                  &
        (_link.rank() != IDLE))
        reschedule();
    else
        unlock();
}
```

getters for the scheduler

```
class Thread
{
    friend class Scheduler<Thread>;

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

public:
    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
    };

    typedef Ordered_Queue<Thread, Criterion,
            Scheduler<Thread>::Element> Queue;
protected:
    Criterion & criterion() {
        return const_cast<Criterion &>(_link.rank()); }
    Queue::Element * link() { return &_link; }

    static Thread * volatile running() {
        return _scheduler.chosen(); }
    static void dispatch(Thread * prev, Thread * next,
                         bool charge = true);
```

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

    static volatile unsigned int _thread_count;
    static Scheduler_Timer * _timer;
    static Scheduler<Thread> _scheduler;
};

void Thread::constructor_prologue(...)
{
    lock();
    _thread_count++;
    _scheduler.insert(this);
    _stack = new (SYSTEM) char[stack_size];
}

void T                                  .)
{
    if((_state != READY) && (_state != RUNNING))
        _scheduler.suspend(this);
    if(preemptive && (_state == READY) &&
        (_link.rank() != IDLE))
        reschedule();
    else
        unlock();
}
```

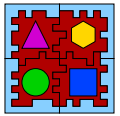
_running is now the head of the scheduling list



```
class Thread
{
    friend class Scheduler<Thread>;

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

public:
    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
    };

    typedef Ordered_Queue<Thread, Criterion,
            Scheduler<Thread>::Element> Queue;
protected:
    Criterion & criterion() {
        return const_cast<Criterion &>(_link.rank()); }
    Queue::Element * link() { return &_link; }

    static Thread * volatile running() {
        return _scheduler.chosen(); }
    static void dispatch(Thread * prev, Thread * next,
                         bool charge = true);
```

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

    static volatile unsigned int _thread_count;
    static Scheduler_Timer * _timer;
    static Scheduler<Thread> _scheduler;
};

void Thread::constructor_prologue(...)
{
    lock();
    _thread_count++;
    _scheduler.insert(this);
    _stack = new (SYSTEM) char[stack_size];
}

void Thread::constructor_epilogue(...)
{
    if((_state != READY) && (_state != RUNNING))
        _scheduler.suspend(this);
    if(preemptive && (_state == READY) &&
        (_li
        resc
    else
        unlock();
}
```

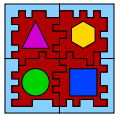
dispatch() now features charging control for pass()



```
class Thread
{
    friend class Scheduler<Thread>;

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

public:
    typed                                   ty Priority;

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

    typedef Ordered_Queue<Thread, Criterion,
            Scheduler<Thread>::Element> Queue;
protected:
    Criterion & criterion() {
        return const_cast<Criterion &>(_link.rank()); }
    Queue::Element * link() { return &_link; }

    static Thread * volatile running() {
        return _scheduler.chosen(); }
    static void dispatch(Thread * prev, Thread * next,
                         bool charge = true);
```

not to be confused with Scheduler::schedulables()

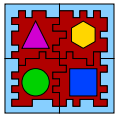
```
protected:
    char * _stack;
    Context * volatile _context;
    volatile State _state;
    Queue * _waiting;
    Thread * volatile _joining;
    Queue::Element _link;

    static volatile unsigned int _thread_count;
    static Scheduler_Timer * _timer;
    static Scheduler<Thread> _scheduler;
};

void Thread::constructor_prologue(...)
{
    lock();
    _thread_count++;
    _scheduler.insert(this);
    _stack = new (SYSTEM) char[stack_size];
}

void Thread::constructor_epilogue(...)
{
    if((_state != READY) && (_state != RUNNING))
        _scheduler.suspend(this);
    if(preemptive && (_state == READY) &&
        (_link.rank() != IDLE))
        reschedule();
    else
        unlock();
}
```



```
class Thread
{
    friend class Scheduler<Thread>;

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

public:
    typedef Scheduling_Criteria::Priority Priority;

    typedef Traits<Thread>::Criterion Criterion;
    enum {
        MAIN    = Criterion::MAIN,
        IDLE    = Criterion::IDLE
    };

    typedef Ordered_Queue<Thread, Criterion,
            Scheduler<Thread>::Element> Queue;
protected:
    Criterion & criterion() {
        return const_cast<Criterion &>(_link.rank()); }
    Queue::Element * link() { return &_link; }

    static Thread * volatile running() {
        return _scheduler.chosen(); }
    static void dispatch(Thread * prev, Thread * next,
                         bool charge = true);
```

always register the Thread with the Scheduler

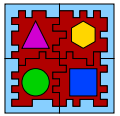
```
protected:
    char * _stack;
    Context * volatile _context;
    volatile State _state;
    Queue * _waiting;
    Thread * volatile _joining;
    Queue::Element _link;

    static volatile unsigned int _thread_count;
    static Scheduler_Timer * _timer;
    static Scheduler<Thread> _scheduler;
};

void Thread::constructor_prologue(...)
{
    lock();
    _thread_count++;
    _scheduler.insert(this);
    _stack = new (SYSTEM) char[stack_size];
}

void Thread::constructor_epilogue(...)
{
    if((_state != READY) && (_state != RUNNING))
        _scheduler.suspend(this);
    if(preemptive && (_state == READY) &&
        (_link.rank() != IDLE))
        reschedule();
    else
        unlock();
}
```

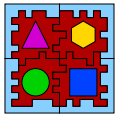


```
class Thread
{
    friend class Scheduler<Thread>;

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

public:
    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
    };

    typedef Ordered_Queue<Thread, Criterion,
                          ment> Queue;

    Criterion & criterion() {
        return const_cast<Criterion &>(_link.rank()); }
    Queue::Element * link() { return &_link; }

    static Thread * volatile running() {
        return _scheduler.chosen(); }
    static void dispatch(Thread * prev, Thread * next,
                         bool charge = true);
```

suspend the Thread if not READY or RUNNING

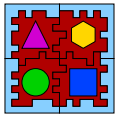
```
protected:
    char * _stack;
    Context * volatile _context;
    volatile State _state;
    Queue * _waiting;
    Thread * volatile _joining;
    Queue::Element _link;

    static volatile unsigned int _thread_count;
    static Scheduler_Timer * _timer;
    static Scheduler<Thread> _scheduler;
};

void Thread::constructor_prologue(...)
{
    lock();
    _thread_count++;
    _scheduler.insert(this);
    _stack = new (SYSTEM) char[stack_size];
}

void Thread::constructor_epilogue(...)
{
    if((_state != READY) && (_state != RUNNING))
        _scheduler.suspend(this);
    if(preemptive && (_state == READY) &&
       (_link.rank() != IDLE))
        reschedule();
    else
        unlock();
}
```





```
Thread::~Thread()
{
    lock();
    assert(_state != RUNNING);
    switch(_state) {
    case RUNNING:
        exit(-1);
        break;
    case READY:
        _scheduler.remove(this);
        _thread_count--;
        break;
    case SUSPENDED:
        _scheduler.resume(this);
        _scheduler.remove(this);
        _thread_count--;
        break;
    case WAITING:
        _waiting->remove(this);
        _scheduler.resume(this);
        _scheduler.remove(this);
        _thread_count--;
        break;
    case FINISHING:
        break;
    }
    if(_joining)
        _joining->resume();
    unlock();
    delete _stack;
}
```

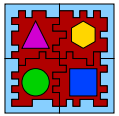
```
Thread::~Thread()
{
    lock();
    assert(_state != RUNNING);
    switch(_state) {
    case RUNNING:
        exit(-1);
        break;
    case READY:
        _scheduler.remove(this);
        _thread_count--;
        break;
    case SUSPENDED:
        _scheduler.resume(this);
        _scheduler.remove(this);
        _thread_count--;
        break;
    case WAITING:
        _waiting->remove(this);
        _scheduler.resume(this);
        _scheduler.remove(this);
        _thread_count--;
        break;
    case FINISHING:
        break;
    }
    if(_joining)
        _joining->resume();
    unlock();
    delete _stack;
}
```



resume() before remove()
(won't start running)

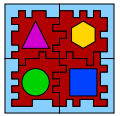
```
Thread::~Thread()
{
    lock();
    assert(_state != RUNNING);
    switch(_state) {
    case RUNNING:
        exit(-1);
        break;
    case READY:
        _scheduler.remove(this);
        _thread_count--;
        break;
    case SUSPENDED:
        _scheduler.resume(this);
        _scheduler.remove(this);
        _thread_count--;
        break;
    case WAITING:
        _waiting->remove(this);
        _scheduler.resume(this);
        _scheduler.remove(this);
        _thread_count--;
        break;
    case FINISHING:
        break;
    }
    if(_joining)
        _joining->resume();
    unlock();
    delete _stack;
}
```

be careful with multiple queues!



```
Thread::~Thread()
{
    lock();
    assert(_state != RUNNING);
    switch(_state) {
    case RUNNING:
        exit(-1);
        break;
    case READY:
        _scheduler.remove(this);
        _thread_count--;
        break;
    case SUSPENDED:
        _scheduler.resume(this);
        _scheduler.remove(this);
        _thread_count--;
        break;
    case WAITING:
        _waiting->remove(this);
        _scheduler.resume(this);
        _scheduler.remove(this);
        _thread_count--;
        break;
    case FINISHING:
        break;
    }
    if(_joining)
        _joining->resume();
    unlock();
    delete _stack;
}
```

case *RUNNING*: // For switch completion only: the running thread would have deleted itself! Stack wouldn't have been released!



```
void Thread::priority(const Priority & c)
{
    lock();

    _link.rank(Criterion(c));

    if(_state != RUNNING) {
        _scheduler.remove(this);
        _scheduler.insert(this);
    }
    if(preemptive)
        reschedule();
    else
        unlock();
}

void Thread::pass()
{
    lock();

    Thread * prev = running();
    Thread * next = _scheduler.choose(this);

    if(next)
        dispatch(prev, next, false);
    else {
        db<Thread>(WRN) << "Thread::pass => thread
            (" << this << ") not ready!" << endl;
        unlock();
    }
}
```

```
void Thread::suspend(bool locked)
{
    if(!locked)
        lock();
    Thread * prev = running();
    _state = SUSPENDED;
    _scheduler.suspend(this);
    Thread * next = running();
    dispatch(prev, next);
}

void Thread::resume()
{
    lock();
    if(_state == SUSPENDED) {
        _state = READY;
        _scheduler.resume(this);
        if(preemptive)
            reschedule();
        else
            unlock();
    } else {
        db<Thread>(WRN) << "Resume called for
            unsuspended object!" << endl;
        unlock();
    }
}
```

```
void Thread::priority(const Priority & c)
{
    lock();

    _link.rank(Criterion(c));

    if(_state != RUNNING) {
        _scheduler.remove(this);
        _scheduler.insert(this);
    }
    if(preemptive)
        reschedule();
    else
        unlock();
}

void Thread::pass()
{
    lock();

    Thread * prev = running();
    Thread * next = _scheduler.choose(this);

    if(next)
        dispatch(prev, next, false);
    else {
        db<Thread>(WRN) << "Thread::pass => thread
            (" << this << ") not ready!" << endl;
        unlock();
    }
}
```

```
void Thread::suspend(bool locked)
{
    if(!l
        l
    Thread * prev = running();
    _state = SUSPENDED;
    _scheduler.suspend(this);
    Thread * next = running();
    dispatch(prev, next);
}

void Thread::resume()
{
    lock();
    if(_state == SUSPENDED) {
        _state = READY;
        _scheduler.resume(this);
        if(preemptive)
            reschedule();
        else
            unlock();
    } else {
        db<Thread>(WRN) << "Resume called for
            unsuspended object!" << endl;
        unlock();
    }
}
```

mind the conversions!

```
void Thread::priority(const Priority & c)
{
    lock();

    _link.rank(Criterion(c));

    if(_state != RUNNING) {
        _scheduler.remove(this);
        _scheduler.insert(this);
    }
    if(preemptive)
        reschedule();
    else
        unlock();
}

void Thread::pass()
{
    lock();

    Thread * prev = running();
    Thread * next = _scheduler.choose(this);

    if(next)
        dispatch(prev, next, false);
    else {
        db<Thread>(WRN) << "Thread::pass => thread
            (" << this << ") not ready!" << endl;
        unlock();
    }
}
```

```
void Thread::suspend(bool locked)
{
    if(!locked)
        lock();
    Thread * prev = running();
    _state = SUSPENDED;
    _sche
    Threa
    dispa
}

void Thread::resume()
{
    lock();
    if(_state == SUSPENDED) {
        _state = READY;
        _scheduler.resume(this);
        if(preemptive)
            reschedule();
        else
            unlock();
    } else {
        db<Thread>(WRN) << "Resume called for
            unsuspended object!" << endl;
        unlock();
    }
}
```

reorder the scheduling queue

```
void Thread::priority(const Priority & c)
{
    lock();

    _link.rank(Criterion(c));

    if(_state != RUNNING) {
        _scheduler.remove(this);
        _scheduler.insert(this);
    }
    if(preemptive)
        reschedule();
    else
        unlock();
}

void Thread::pass()
{
    lock();

    Thread * prev = running();
    Thread * next = _scheduler.choose(this);

    if(next)
        dispatch(prev, next, false);
    else {
        db<Thread>(WRN) << "Thread::pass => thread
            (" << this << ") not ready!" << endl;
        unlock();
    }
}
```

```
void Thread::suspend(bool locked)
{
    if(!locked)
        lock();
    Thread * prev = running();
    _state = SUSPENDED;
    _scheduler.suspend(this);
    Thread * next = running();
    dispatch(prev, next);
}

void Thread::resume()
{
    lock();
    if(_state == SUSPENDED) {
        _state = READY;
        _scheduler.resume(this);
        if(preemptive)
            reschedule();
        el
            unlock();
    } else {
        db<Thread>(WRN) << "Resume called for
            unsuspended object!" << endl;
        unlock();
    }
}
```

null if cannot be chosen

```
void Thread::priority(const Priority & c)
{
    lock();

    _link.rank(Criterion(c));

    if(_state != RUNNING) {
        _scheduler.remove(this);
        _scheduler.insert(this);
    }
    if(preemptive)
        reschedule();
    else
        unlock();
}

void Thread::pass()
{
    lock();

    Thread * prev = running();
    Thread * next = _scheduler.choose(this);

    if(next)
        dispatch(prev, next, false);
    else {
        db<Thread>(WRN) << "Thread::pass => thread
            (" << this << ") not ready!" << endl;
        unlock();
    }
}
```

```
void Thread::suspend(bool locked)
{
    if(!locked)
        lock();
    Thread * prev = running();
    _state = SUSPENDED;
    _scheduler.suspend(this);
    Thread * next = running();
    dispatch(prev, next);
}

void Thread::resume()
{
    lock();
    if(_state == SUSPENDED) {
        _state = READY;
        _scheduler.resume(this);
        if(preemptive)
            reschedule();
        else
            unlock();
    } else {
        db<Thread>(WRN) << "Resume called for
            unsuspended object!" << endl;
    }
}
```

don't charge the passing Thread (charge target)

```
void Thread::priority(const Priority & c)
{
    lock();

    _scheduler.insert(this);
    }
    if(preemptive)
        reschedule();
    else
        unlock();
}

void Thread::pass()
{
    lock();

    Thread * prev = running();
    Thread * next = _scheduler.choose(this);

    if(next)
        dispatch(prev, next, false);
    else {
        db<Thread>(WRN) << "Thread::pass => thread
            (" << this << ") not ready!" << endl;
        unlock();
    }
}
```

who holds a pointer to the previously running thread in a software implementation?
(suspend() == remove())

```
void Thread::suspend(bool locked)
{
    if(!locked)
        lock();
    Thread * prev = running();
    _state = SUSPENDED;
    _scheduler.suspend(this);
    Thread * next = running();
    dispatch(prev, next);
}

void Thread::resume()
{
    lock();
    if(_state == SUSPENDED) {
        _state = READY;
        _scheduler.resume(this);
        if(preemptive)
            reschedule();
        else
            unlock();
    } else {
        db<Thread>(WRN) << "Resume called for
            unsuspended object!" << endl;
        unlock();
    }
}
```

```
void Thread::priority(const Priority & c)
{
    lock();

    _link.rank(Criterion(c));

    if(_state != RUNNING) {
        _scheduler.remove(this);
        _scheduler.insert(this);
    }
    if(preemptive)
```

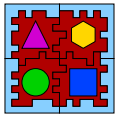
where does “this” come from?

```
}

void Thread::pass()
{
    lock();

    Thread * prev = running();
    Thread * next = _scheduler.choose(this);

    if(next)
        dispatch(prev, next, false);
    else {
        db<Thread>(WRN) << "Thread::pass => thread
            (" << this << ") not ready!" << endl;
        unlock();
    }
}
```

```
void Thread::suspend(bool locked)
{
    if(!locked)
        lock();
    Thread * prev = running();
    _state = SUSPENDED;
    _scheduler.suspend(this);
    Thread * next = running();
    dispatch(prev, next);
}

void Thread::resume()
{
    lock();
    if(_state == SUSPENDED) {
        _state = READY;
        _scheduler.resume(this);
        if(preemptive)
            reschedule();
        else
            unlock();
    } else {
        db<Thread>(WRN) << "Resume called for
            unsuspended object!" << endl;
        unlock();
    }
}
```

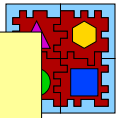
```
void Thread::yield()
{
    lock();
    Thread * prev = running();
    Thread * next = _scheduler.choose_another();
    dispatch(prev, next);
}

void Thread::reschedule()
{
    assert(locked());
    Thread * prev = running();
    Thread * next = _scheduler.choose();
    dispatch(prev, next);
}

void Thread::dispatch(Thread * prev, Thread * next,
                      bool charge)
{
    if(charge) {
        if(Criterion::timed)
            _timer->reset();
    }
    if(prev != next) {
        if(prev->_state == RUNNING)
            prev->_state = READY;
        next->_state = RUNNING;
        CPU::switch_context(const_cast<Context **>(&prev->_context), next->_context);
    }
    unlock();
}
```

```
class Init_First
{
public:
    Init_First() {
        if(!Traits<System>::multithread) {
            CPU::int_enable();
            return;
        }

        Thread::self()->_context->load();
    }
};
```



```
void Thread::yield()
{
    lock();
    Thread * prev = running();
    Thread * next = _scheduler.choose_another();
    dispatch(prev, next);
}

void Thread::reschedule()
{
    assert(locked());
    Thread * prev = running();
    Thread * next = _scheduler.choose();
    dispatch(prev, next);
}

void Thread::dispatch(Thread * prev, Thread * next,
                      bool charge)
{
    if(charge) {
        if(Criterion::timed)
            _timer->reset();
    }
    if(prev != next) {
        if(prev->_state == RUNNING)
            prev->_state = READY;
        next->_state = RUNNING;
        CPU::switch_context(const_cast<Context **>(&prev->_context), next->_context);
    }
    unlock();
}
```

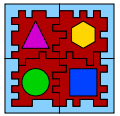
```
class Init_First
{
public:
    Init_Fir
        if(!Traits<System>::multithread) {
            CPU::int_enable();
            return;
        }

        Thread::self()->_context->load();
    }
};
```



prev might be equal to next!

```
void Thread::yield()
{
    lock();
    Thread * prev = running();
    Thread * next = _scheduler.choose_another();
    dispatch(prev, next);
}

void Thread::reschedule()
{
    assert(locked());
    Thread * prev = running();
    Thread * next = _scheduler.choose();
    dispatch(prev, next);
}

void Thread::dispatch(Thread * prev, Thread * next,
                      bool charge)
{
    if(charge) {
        if(Criterion::timed)
            _timer->reset();
    }
    if(prev != next) {
        if(prev->_state == RUNNING)
            prev->_state = READY;
        next->_state = RUNNING;
        CPU::switch_context(const_cast<Context **>(&prev->_context), next->_context);
    }
    unlock();
}
```

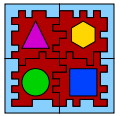
```
class Init_First
{
public:
    Init_First() {
        if(!Traits<System>::multithread) {
            CPU::int_enable();
            return;
        }

        Thread::self()->_context->load();
    }
};
```



more elaborated charges can come here

```
void Thread::yield()
{
    lock();
    Thread * prev = running();
    Thread * next = _scheduler.choose_another();
    dispatch(prev, next);
}

void Thread::reschedule()
{
    assert(locked());
    Thread * prev = running();
    Thread * next = _scheduler.choose();
    dispatch(prev, next);
}

void Thread::dispatch(Thread * prev, Thread * next,
                      bool charge)
{
    if(charge) {
        if(Criterion::timed)
            _timer->reset();
    }
    if(prev != next) {
        if(prev->_state == RUNNING)
            prev->_state = READY;
        next->_state = RUNNING;
        CPU::switch_context(const_cast<Context **>(&prev->_context), next->_context);
    }
    unlock();
}
```

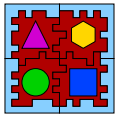
```
class Init_First
{
public:
    Init_First() {
        if(!Traits<System>::multithread) {
            CPU::int_enable();
            return;
        }

        Thread::self()->_context->load();
    }
};
```



will actually dispatch?

```
void Thread::yield()
{
    lock();
    Thread * prev = running();
    Thread * next = _scheduler.choose_another();
    dispatch(prev, next);
}

void Thread::reschedule()
{
    assert(locked());
    Thread * prev = running();
    Thread * next = _scheduler.choose();
    dispatch(prev, next);
}

void Thread::dispatch(Thread * prev, Thread * next,
                      bool charge)
{
    if(charge) {
        if(Criterion::timed)
            _timer->reset();
    }
    if(prev != next) {
        if(prev->_state == RUNNING)
            prev->_state = READY;
        next->_state = RUNNING;
        CPU::switch_context(const_cast<Context **>(&prev->_context), next->_context);
    }
    unlock();
}
```

```
class Init_First
{
public:
    Init_First() {
        if(!Traits<System>::multithread) {
            CPU::int_enable();
            return;
        }

        Thread::self()->_context->load();
    }
};
```

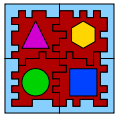


then, adjust states

CRTs are run before main()

```
void Thread::yield()
{
    Thread * prev = running();
    Thread * next = _scheduler.choose_another();
    dispatch(prev, next);
}

void Thread::reschedule()
{
    assert(locked());
    Thread * prev = running();
    Thread * next = _scheduler.choose();
    dispatch(prev, next);
}

void Thread::dispatch(Thread * prev, Thread * next,
                      bool charge)
{
    if(charge) {
        if(Criterion::timed)
            _timer->reset();
    }
    if(prev != next) {
        if(prev->_state == RUNNING)
            prev->_state = READY;
        next->_state = RUNNING;
        CPU::switch_context(const_cast<Context **>(&prev->_context), next->_context);
    }
    unlock();
}
```

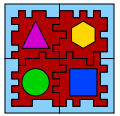
```
class Init_First
{
public:
    Init_First() {
        if(!Traits<System>::multithread) {
            CPU::int_enable();
            return;
        }

        Thread::self()->_context->load();
    }
};
```



```
void Thread::yield()
{
    lock();
    Thread * prev = running();
    Thread * next = _scheduler.choose_another();
    dispatch(prev, next);
}

void
{
    Thread * prev = running();
    Thread * next = _scheduler.choose();
    dispatch(prev, next);
}

void Thread::dispatch(Thread * prev, Thread * next,
                      bool charge)
{
    if(charge) {
        if(Criterion::timed)
            _timer->reset();
    }
    if(prev != next) {
        if(prev->_state == RUNNING)
            prev->_state = READY;
        next->_state = RUNNING;
        CPU::switch_context(const_cast<Context **>(&prev->_context), next->_context);
    }
    unlock();
}
```

is it safe to call self() here?
(running is set by the Scheduler at each insert())

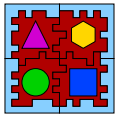
```
class Init_First
{
public:
    Init_First() {
        if(!Traits<System>::multithread) {
            CPU::int_enable();
            return;
        }

        Thread::self()->_context->load();
    }
};
```



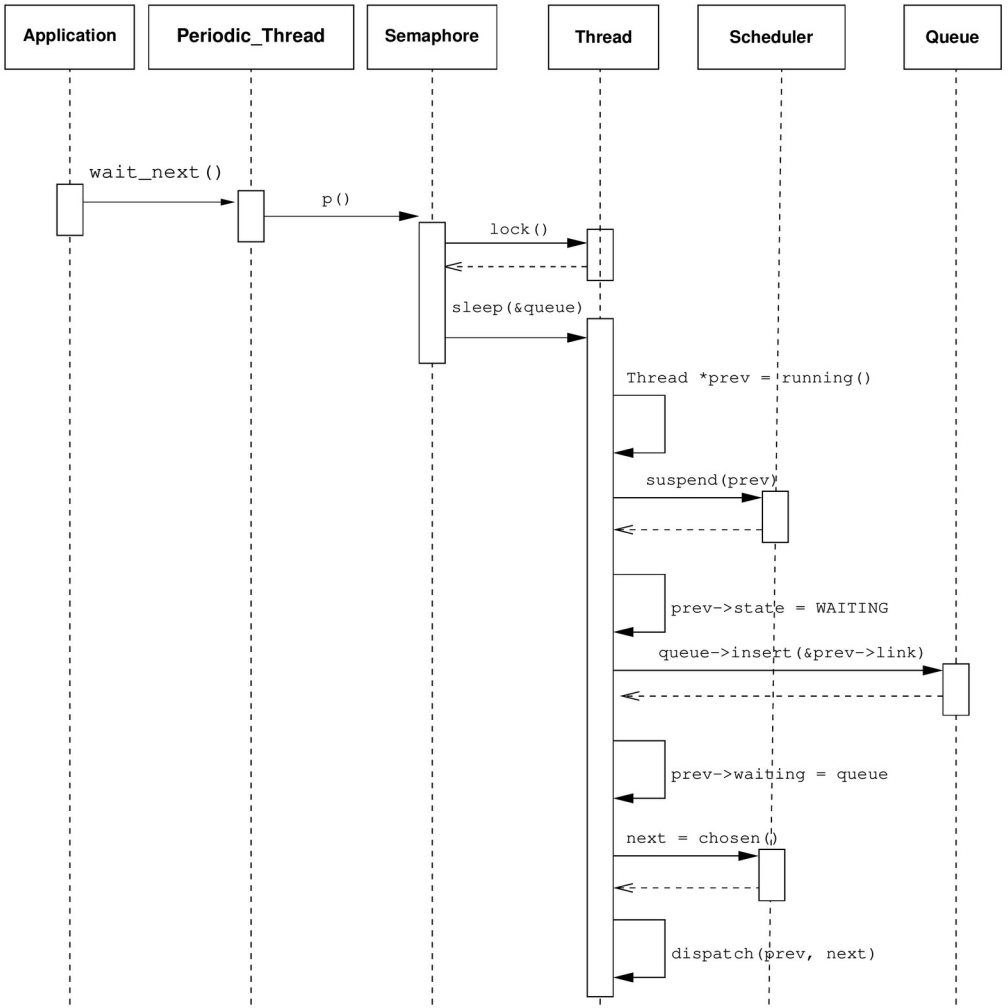
```
void Thread::yield()
{
    lock();
    Thread * prev = running();
    Thread * next = _scheduler.choose_another();
    dispatch(prev, next);
}

void Thread::reschedule()
{
    assert(locked());
    Thread * prev = running();
    Thread * next = _scheduler.choose();
    dispatch(prev, next);
}

void Thread::dispatch(Thread * prev, Thread * next,
                      bool charge)
{
    if(charge) {
        if(Criterion::timed)
            _timer->reset();
    }
    if(prev != next) {
        if(prev->_state == RUNNING)
            prev->_state = READY;
        next->_state = RUNNING;
        CPU::switch_context(const_cast<Context **>(&prev->_context), next->_context);
    }
    unlock();
}
```

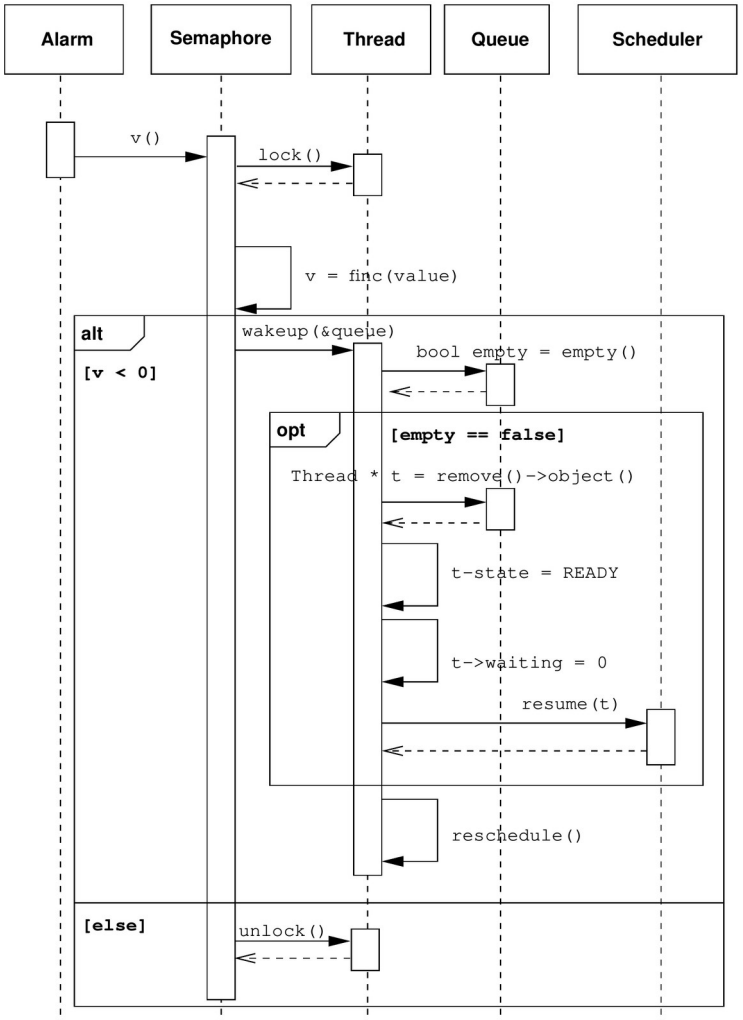
no other context to switch from
(what was then running so far?)

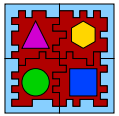
```
class Init_First
{
public:
    Init_First() {
        if(!Traits<System>::multithread) {
            CPU::int_enable();
            return;
        }

        Thread::self()->_context->load();
    }
};
```



sleep() and wakeup()







Embedded Parallel Operating System

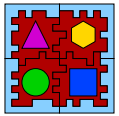
Coding Journey through OS Design
–from co-routines to a multicore kernel–

Prof. Antônio Augusto Fröhlich, Ph.D.

UFSC / LISHA

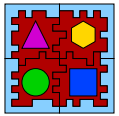
September 30, 2020

Memory Management



Explore the MMU

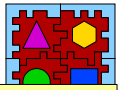
- Address spaces
- Segments (not to be confused with segmentation)
- Multiple heaps
- Exported with placement new



```
class Heap: private Grouping_List<char>
{
    static const bool typed = Traits<System>::multiheap;

public:
    using Grouping_List<char>::empty;
    using Grouping_List<char>::size;

    Heap() {}
    Heap(void * addr, unsigned int bytes) { free(addr, bytes); }

    void * alloc(unsigned int bytes) {
        if(!bytes)
            return 0;
        if(!Traits<CPU>::unaligned_memory_access)
            while((bytes % sizeof(void *)))
                ++bytes;
        if(typed)
            bytes += sizeof(void *);  // add room for heap pointer
        bytes += sizeof(int);         // add room for size
        if(bytes < sizeof(Element))
            bytes = sizeof(Element);
        Element * e = search_decrementing(bytes);
        if(!e) {
            out_of_memory();
            return 0;
        }
        int * addr = reinterpret_cast<int *>(e->object() + e->size());
        if(typed)
            *addr++ = reinterpret_cast<int>(this);
        *addr++ = bytes;
        return addr;
    }
```

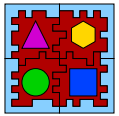
```
class Heap: private Grouping_List<char>
{
    static const bool typed = Traits<System>::multiheap;

public:
    using Grouping_List<char>::empty;
    using Grouping_List<char>::size;

    Heap() {}
    Heap(void * addr, unsigned int bytes) { free(addr, bytes); }

    void * alloc(unsigned int bytes) {
        if(!bytes)
            return 0;
        if(!Traits<CPU>::unaligned_memory_access)
            while((bytes % sizeof(void *)))
                ++bytes;
        if(typed)
            bytes += sizeof(void *);  // add room for heap pointer
        bytes += sizeof(int);         // add room for size
        if(bytes < sizeof(Element))
            bytes = sizeof(Element);
        Element * e = search_decrementing(bytes);
        if(!e) {
            out_of_memory();
            return 0;
        }
        int * addr = reinterpret_cast<int *>(e->object() + e->size());
        if(typed)
            *addr++ = reinterpret_cast<int>(this);
        *addr++ = bytes;
        return addr;
    }
```

buddy allocator



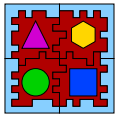
configurable feature

```
class Heap: private Grouping_List<char>
{
    static const bool typed = Traits<System>::multiheap;

public:
    using Grouping_List<char>::empty;
    using Grouping_List<char>::size;

    Heap() {}
    Heap(void * addr, unsigned int bytes) { free(addr, bytes); }

    void * alloc(unsigned int bytes) {
        if(!bytes)
            return 0;
        if(!Traits<CPU>::unaligned_memory_access)
            while((bytes % sizeof(void *)))
                ++bytes;
        if(typed)
            bytes += sizeof(void *);  // add room for heap pointer
        bytes += sizeof(int);         // add room for size
        if(bytes < sizeof(Element))
            bytes = sizeof(Element);
        Element * e = search_decrementing(bytes);
        if(!e) {
            out_of_memory();
            return 0;
        }
        int * addr = reinterpret_cast<int *>(e->object() + e->size());
        if(typed)
            *addr++ = reinterpret_cast<int>(this);
        *addr++ = bytes;
        return addr;
    }
```



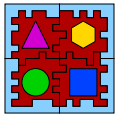
```
class Heap: private Grouping_List<char>
{
    static const bool typed = Traits<System>::multiheap;

public:
    using Grouping_List<char>::empty;
    using Grouping_List<char>::size;

    Heap() {}
    Heap(void * addr, unsigned int bytes) { free(addr, bytes); }

    void * alloc(unsigned int bytes) {
        if(!bytes)
            return 0;
        if(!Traits<CPU>::unaligned_memory_access)
            while((bytes % sizeof(void *)))
                ++bytes;
        if(typed)
            bytes += sizeof(void *);  // add room for heap pointer
        bytes += sizeof(int);         // add room for size
        if(bytes < sizeof(Element))
            bytes = sizeof(Element);
        Element * e = search_decrementing(bytes);
        if(!e) {
            out_of_memory();
            return 0;
        }
        int * addr = reinterpret_cast<int *>(e->object() + e->size());
        if(typed)
            *addr++ = reinterpret_cast<int>(this);
        *addr++ = bytes;
        return addr;
    }
```

heap populated with free()



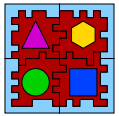
```
class Heap: private Grouping_List<char>
{
    static const bool typed = Traits<System>::multiheap;

public:
    using Grouping_List<char>::empty;
    using Grouping_List<char>::size;

    Heap() {}
    Heap(void * addr, unsigned int bytes) { free(addr, bytes); }

    void * alloc(unsigned int bytes) {
        if(!bytes)
            return 0;
        if(!Traits<CPU>::unaligned_memory_access)
            while((bytes % sizeof(void *)))
                ++bytes;
        if(typed)
            bytes += sizeof(void *);  // add room for heap pointer
        bytes += sizeof(int);         // add room for size
        if(bytes < sizeof(Element))
            bytes = sizeof(Element);
        Element * e = search_decrementing(bytes);
        if(!e) {
            out_of_memory();
            return 0;
        }
        int * addr = reinterpret_cast<int *>(e->object() + e->size());
        if(typed)
            *addr++ = reinterpret_cast<int>(this);
        *addr++ = bytes;
        return addr;
    }
```

optimization or requirement (e.g. MIPS)



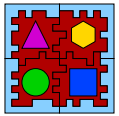
```
class Heap: private Grouping_List<char>
{
    static const bool typed = Traits<System>::multiheap;

public:
    using Grouping_List<char>::empty;
    using Grouping_List<char>::size;

    Heap() {}
    Heap(void * addr, unsigned int bytes) { free(addr, bytes); }

    void * alloc(unsigned int bytes) {
        if(!bytes)
            return 0;
        if(!Traits<CPU>::unaligned_memory_access)
            while((bytes % sizeof(void *)))
                ++bytes;
        if(typed)
            bytes += sizeof(void *);  // add room for heap pointer
        bytes += sizeof(int);         // add room for size
        if(bytes < sizeof(Element))
            bytes = sizeof(Element);
        Element * e = search_decrementing(bytes);
        if(!e) {
            out_of_memory();
            return 0;
        }
        int * addr = reinterpret_cast<int *>(e->object() + e->size());
        if(typed)
            *addr++ = reinterpret_cast<int>(this);
        *addr++ = bytes;
        return addr;
    }
```

metainfo inside allocated unit



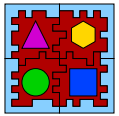
```
class Heap: private Grouping_List<char>
{
    static const bool typed = Traits<System>::multiheap;

public:
    using Grouping_List<char>::empty;
    using Grouping_List<char>::size;

    Heap() {}
    Heap(void * addr, unsigned int bytes) { free(addr, bytes); }

    void * alloc(unsigned int bytes) {
        if(!bytes)
            return 0;
        if(!Traits<CPU>::unaligned_memory_access)
            while((bytes % sizeof(void *)))
                ++bytes;
        if(typed)
            bytes += sizeof(void *);  // add room for heap pointer
        bytes += sizeof(int);         // add room for size
        if(bytes < sizeof(Element))
            bytes = sizeof(Element);
        Element * e = search_decrementing(bytes);
        if(!e) {
            out_of_memory();
            return 0;
        }
        int * addr = reinterpret_cast<int *>(e->object() + e->size());
        if(typed)
            *addr++ = reinterpret_cast<int>(this);
        *addr++ = bytes;
        return addr;
    }
```

buddy alloc



```
class Heap: private Grouping_List<char>
{
    static const bool typed = Traits<System>::multiheap;

public:
    using Grouping_List<char>::empty;
    using Grouping_List<char>::size;

    Heap() {}
    Heap(void * addr, unsigned int bytes) { free(addr, bytes); }

    void * alloc(unsigned int bytes) {
        if(!bytes)
            return 0;
        if(!Traits<CPU>::unaligned_memory_access)
            while((bytes % sizeof(void *)))
                ++bytes;
        if(typed)
            bytes += sizeof(void *);  // add room for heap pointer
        bytes += sizeof(int);         // add room for size
        if(bytes < sizeof(Element))
            bytes = sizeof(Element);
        Element * e = search_decrementing(bytes);
        if(!e) {
            out_of_memory();
            return 0;
        }
        int * addr = reinterpret_cast<int *>(e->object() + e->size());
        if(typed)
            *addr++ = reinterpret_cast<int>(this);
        *addr++ = bytes;
        return addr;
    }
```

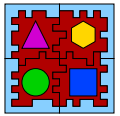
the famous one!

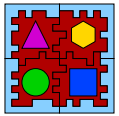


```
    void free(void * ptr, unsigned int bytes) {
        if(ptr && (bytes >= sizeof(Element))) {
            Element * e = new (ptr) Element(reinterpret_cast<char *>(ptr), bytes);
            Element * m1, * m2;
            insert_merging(e, &m1, &m2);
        }
    }

    static void typed_free(void * ptr) {
        int * addr = reinterpret_cast<int *>(ptr);
        unsigned int bytes = *--addr;
        Heap * heap = reinterpret_cast<Heap *>(*--addr);
        heap->free(addr, bytes);
    }

    static void untyped_free(Heap * heap, void * ptr) {
        int * addr = reinterpret_cast<int *>(ptr);
        unsigned int bytes = *--addr;
        heap->free(addr, bytes);
    }

private:
    void out_of_memory();
};
```

```
    void free(void * ptr, unsigned int bytes) {
        if(ptr && (bytes >= sizeof(Element))) {
            Element * e = new (ptr) Element(reinterpret_cast<char *>(ptr), bytes);
            Element * m1, * m2;
            insert_merging(e, &m1, &m2);
        }
    }

    static void typed_free(void * ptr) {
        int * addr = reinterpret_cast<int *>(ptr);
        unsigned int bytes = *--addr;
        Heap * heap = reinterpret_cast<Heap *>(*--addr);
        heap->free(addr, bytes);
    }

    static void untyped_free(Heap * heap, void * ptr) {
        int * addr = reinterpret_cast<int *>(ptr);
        unsigned int bytes = *--addr;
        heap->free(addr, bytes);
    }

private:
    void out_of_memory();
};
```

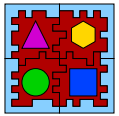
List::Element in-place
(size ensured by alloc)





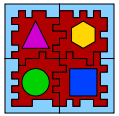
```
    void free(void * ptr, unsigned int bytes) {
        if(ptr && (bytes >= sizeof(Element))) {
            Element * e = new (ptr) Element(reinterpret_cast<char *>(ptr), bytes);
            Element * m1, * m2;
            insert_merging(e, &m1, &m2);
        }
    }

    static void typed_free(void * ptr) {
        int * addr = reinterpret_cast<int *>(ptr);
        unsigned int bytes = *--addr;
        Heap * heap = reinterpret_cast<Heap *>(*--addr);
        heap->free(addr, bytes);
    }

    static void untyped_free(Heap * heap, void * ptr) {
        int * addr = reinterpret_cast<int *>(ptr);
        unsigned int bytes = *--addr;
        heap->free(addr, bytes);
    }

private:
    void out_of_memory();
};
```

buddy deallocator



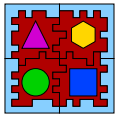
```
    void free(void * ptr, unsigned int bytes) {
        if(ptr && (bytes >= sizeof(Element))) {
            Element * e = new (ptr) Element(reinterpret_cast<char *>(ptr), bytes);
            Element * m1, * m2;
            insert_merging(e, &m1, &m2);
        }
    }

    static void typed_free(void * ptr) {
        int * addr = reinterpret_cast<int *>(ptr);
        unsigned int bytes = *--addr;
        Heap * heap = reinterpret_cast<Heap *>(*--addr);
        heap->free(addr, bytes);
    }

    static void untyped_free(Heap * heap, void * ptr) {
        int * addr = reinterpret_cast<int *>(ptr);
        unsigned int bytes = *--addr;
        heap->free(addr, bytes);
    }

private:
    void out_of_memory();
};
```

metainfo inside allocated unit



```
    void free(void * ptr, unsigned int bytes) {
        if(ptr && (bytes >= sizeof(Element))) {
            Element * e = new (ptr) Element(reinterpret_cast<char *>(ptr), bytes);
            Element * m1, * m2;
            insert_merging(e, &m1, &m2);
        }
    }

    static void typed_free(void * ptr) {
        int * addr = reinterpret_cast<int *>(ptr);
        unsigned int bytes = *--addr;
        Heap * heap = reinterpret_cast<Heap *>(*--addr);
        heap->free(addr, bytes);
    }

    static void untyped_free(Heap * heap, void * ptr) {
        int * addr = reinterpret_cast<int *>(ptr);
        unsigned int bytes = *--addr;
        heap->free(addr, bytes);
    }

private:
    void out_of_memory();
};
```

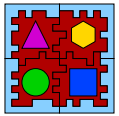
typed and untyped free()



```
class Application
{
    friend void * ::malloc(size_t);
    friend void ::free(void *);
  :
private:
    static char _preheap[sizeof(Heap)];
    static Heap * _heap;
};

class System
{
    friend void * ::malloc(size_t);
    friend void ::free(void *);
    friend void * ::operator new(size_t, const EPOS::System_Allocator &);
    friend void * ::operator new[](size_t, const EPOS::System_Allocator &);
    friend void ::operator delete(void *);
    friend void ::operator delete[](void *);
  :
private:
    static char _preheap[(Traits<System>::multiheap ? sizeof(Segment) : 0) + sizeof(Heap)];
    static Segment * _heap_segment;
    static Heap * _heap;
};
```

```
class Application
{
    friend void * ::malloc(size_t);
    friend void ::free(void *);
  :
private:
    static char _preheap[sizeof(Heap)];
    static Heap * _heap;
};

class System
{
    friend void * ::malloc(size_t);
    friend void ::free(void *);
    friend void * ::operator new(size_t, const EPOS::System_Allocator &);
    friend void * ::operator new[](size_t, const EPOS::System_Allocator &);
    friend void ::operator delete(void *);
    friend void ::operator delete[](void *);
  :
private:
    static char _preheap[(Traits<System>::multiheap ? sizeof(Segment) : 0) + sizeof(Heap)];
    static Segment * _heap_segment;
    static Heap * _heap;
};
```

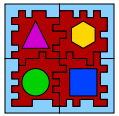
malloc() and free() use the application's heap



```
class Application
{
    friend void * ::malloc(size_t);
    friend void ::free(void *);
  :
private:
    static char _preheap[sizeof(Heap)];
    static Heap * _heap;
};
```

proto-heap
(cannot new Heap)
(in data segment)

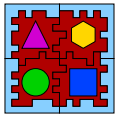
```
class System
{
    friend void * ::malloc(size_t);
    friend void ::free(void *);
    friend void * ::operator new(size_t, const EPOS::System_Allocator &);
    friend void * ::operator new[](size_t, const EPOS::System_Allocator &);
    friend void ::operator delete(void *);
    friend void ::operator delete[](void *);
  :
private:
    static char _preheap[(Traits<System>::multiheap ? sizeof(Segment) : 0) + sizeof(Heap)];
    static Segment * _heap_segment;
    static Heap * _heap;
};
```



```
class Application
{
    friend void * ::malloc(size_t);
    friend void ::free(void *);
  :
private:
    static char _preheap[sizeof(Heap)];
    static Heap * _heap;
};

class System
{
    friend void * ::malloc(size_t);
    friend void ::free(void *);
    friend void * ::operator new(size_t, const EPOS::System_Allocator &);
    friend void * ::operator new[](size_t, const EPOS::System_Allocator &);
    friend void ::operator delete(void *);
    friend void ::operator delete[](void *);
  :
private:
    static char _preheap[(Traits<System>::multiheap ? sizeof(Segment) : 0) + sizeof(Heap)];
    static Segment * _heap_segment;
    static Heap * _heap;
};
```

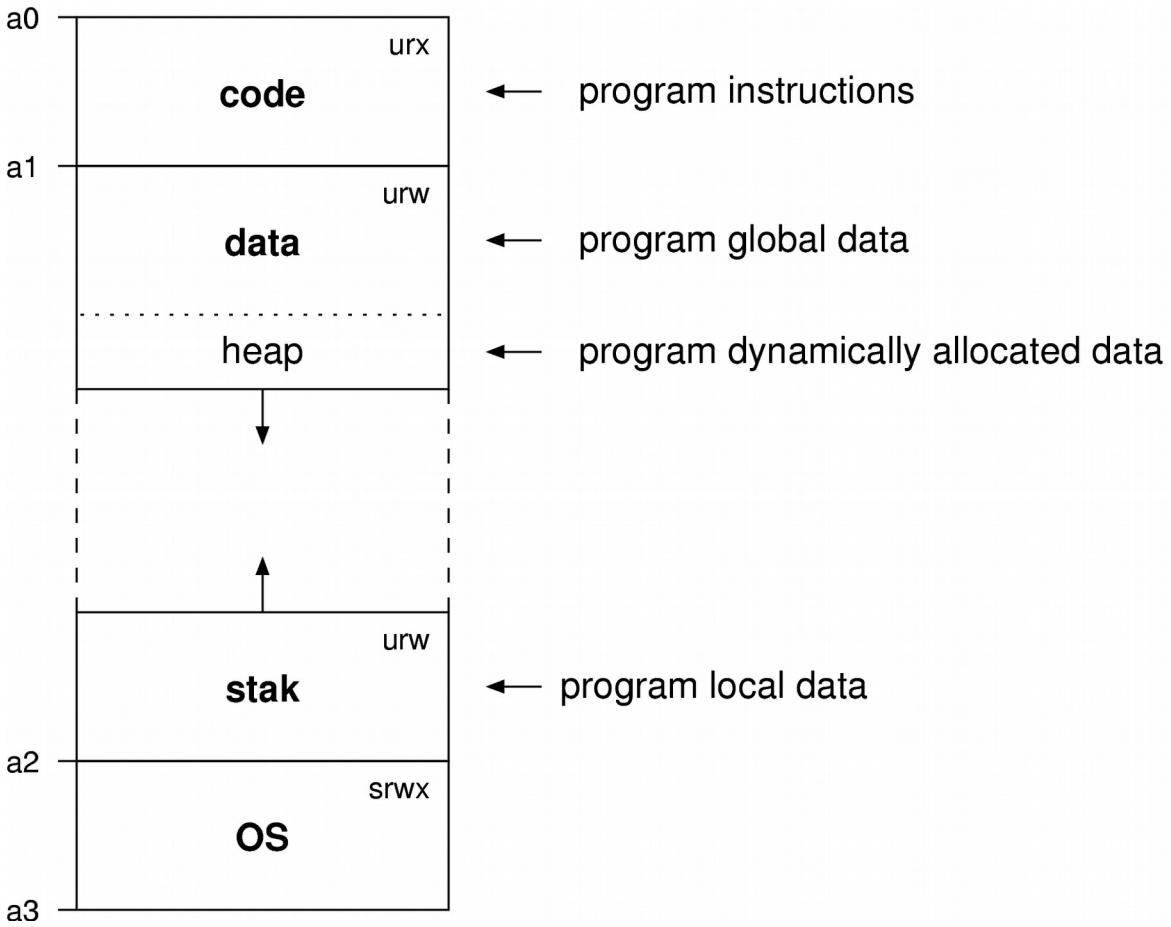
system allocator based on placement new operator

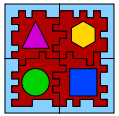
```
class Application
{
    friend void * ::malloc(size_t);
    friend void ::free(void *);
  :
private:
    static char _preheap[sizeof(Heap)];
    static Heap * _heap;
};

class System
{
    friend void * ::malloc(size_t);
    friend void ::free(void *);
    friend void * ::operator new(size_t, const EPOS::System_Allocator &);
    friend void * ::operator new[](size_t, const EPOS::System_Allocator &);
    friend void ::operator delete(void *);
    friend void ::operator delete[](void *);
  :
private:
    static char _preheap[(Traits<System>::multiheap ? sizeof(Segment) : 0) + sizeof(Heap)];
    static Segment * _heap_segment;
    static Heap * _heap;
};
```



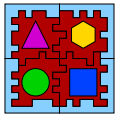
proto-heap
(heap in separate segment)

Unix Memory Model



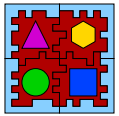


```
class Init_System
{
private:
    static const unsigned int HEAP_SIZE = Traits<System>::HEAP_SIZE;

public:
    Init_System() {
      :
        if(Traits<System>::multiheap) {
            Segment * tmp = reinterpret_cast<Segment *>(&System::_preheap[0]);
            System::_heap_segment = new (tmp) Segment(HEAP_SIZE, WHITE, Segment::Flags::SYS);
            System::_heap = new (&System::_preheap[sizeof(Segment)])
                Heap(Address_Space(MMU::current()).attach(System::_heap_segment, Memory_Map::SYS_HEAP),
                    System::_heap_segment->size());
        } else
            System::_heap = new (&System::_preheap[0]) Heap(MMU::alloc(MMU::pages(HEAP_SIZE)), HEAP_SIZE);
      :
    }
};

// SETUP
  :
    if(Traits<System>::multiheap) { // Application heap in data segment
        si->lm.app_data_size = MMU::align_page(si->lm.app_data_size);
        si->lm.app_stack = si->lm.app_data + si->lm.app_data_size;
        si->lm.app_data_size += MMU::align_page(Traits<Application>::STACK_SIZE);
        si->lm.app_heap = si->lm.app_data + si->lm.app_data_size;
        si->lm.app_data_size += MMU::align_page(Traits<Application>::HEAP_SIZE);
    }
  :
```



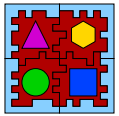
```
class Init_System
{
private:
    static const unsigned int HEAP_SIZE = Traits<System>::HEAP_SIZE;

public:
    Init_System() {
      :
        if(Traits<System>::multiheap) {
            Segment * tmp = reinterpret_cast<Segment *>(&System::_preheap[0]);
            System::_heap_segment = new (tmp) Segment(HEAP_SIZE, WHITE, Segment::Flags::SYS);
            System::_heap = new (&System::_preheap[sizeof(Segment)])
                Heap(Address_Space(MMU::current()).attach(System::_heap_segment, Memory_Map::SYS_HEAP),
                    System::_heap_segment->size());
        } else
            System::_heap = new (&System::_preheap[0]) Heap(MMU::alloc(MMU::pages(HEAP_SIZE)), HEAP_SIZE);
      :
    }
};

// SETUP
  :
    if(Traits<System>::multiheap) { // Application heap in data segment
        si->lm.app_data_size = MMU::align_page(si->lm.app_data_size);
        si->lm.app_stack = si->lm.app_data + si->lm.app_data_size;
        si->lm.app_data_size += MMU::align_page(Traits<Application>::STACK_SIZE);
        si->lm.app_heap = si->lm.app_data + si->lm.app_data_size;
        si->lm.app_data_size += MMU::align_page(Traits<Application>::HEAP_SIZE);
    }
  :
```

system's heap in a private segment



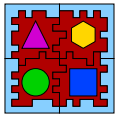
```
class Init_System
{
private:
    static const unsigned int HEAP_SIZE = Traits<System>::HEAP_SIZE;

public:
    Init_System() {
      :
        if(Traits<System>::multiheap) {
            Segment * tmp = reinterpret_cast<Segment *>(&System::_pr
            System::_heap_segment = new (tmp) Segment(HEAP_SIZE, WHITE, Segment::Flags::SYS);
            System::_heap = new (&System::_preheap[sizeof(Segment)])
                Heap(Address_Space(MMU::current()).attach(System::_heap_segment, Memory_Map::SYS_HEAP),
                    System::_heap_segment->size());
        } else
            System::_heap = new (&System::_preheap[0]) Heap(MMU::alloc(MMU::pages(HEAP_SIZE)), HEAP_SIZE);
      :
    }
};

// SETUP
  :
    if(Traits<System>::multiheap) { // Application heap in data segment
        si->lm.app_data_size = MMU::align_page(si->lm.app_data_size);
        si->lm.app_stack = si->lm.app_data + si->lm.app_data_size;
        si->lm.app_data_size += MMU::align_page(Traits<Application>::STACK_SIZE);
        si->lm.app_heap = si->lm.app_data + si->lm.app_data_size;
        si->lm.app_data_size += MMU::align_page(Traits<Application>::HEAP_SIZE);
    }
  :
```

remember: a Segment must be attached to an Address_Space!



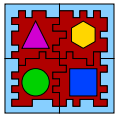
```
class Init_System
{
private:
    static const unsigned int HEAP_SIZE = Traits<System>::HEAP_SIZE;

public:
    Init_System() {
      :
        if(Traits<System>::multiheap) {
            Segment * tmp = reinterpret_cast<Segment *>(&System::_preheap[0]);
            System::_heap_segment = new (tmp) Segment(HEAP_SIZE, WHITE, Segment::Flags::SYS);
            System::_heap = new (&System::_preheap[sizeof(Segment)])
                Heap(Address_Space(MMU::current()).attach(System::_heap_segment, Memory_Map::SYS_HEAP),
                    System::_heap_segment->size());
        } else
            System::_heap = new (&System::_preheap[0]) Heap(MMU::alloc(MMU::pages(HEAP_SIZE)), HEAP_SIZE);
      :
    }
};

// SETUP
  :
    if(Traits<System>::multiheap) { // Application heap in data segment
        si->lm.app_data_size = MMU::align_page(si->lm.app_data_size);
        si->lm.app_stack = si->lm.app_data + si->lm.app_data_size;
        si->lm.app_data_size += MMU::align_page(Traits<Application>::STACK_SIZE);
        si->lm.app_heap = si->lm.app_data + si->lm.app_data_size;
        si->lm.app_data_size += MMU::align_page(Traits<Application>::HEAP_SIZE);
    }
  :
```

get a reference to the current address space (arranged by SETUP)



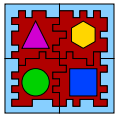
```
class Init_System
{
private:
    static const unsigned int HEAP_SIZE = Traits<System>::HEAP_SIZE;

public:
    Init_System() {
      :
        if(Traits<System>::multiheap) {
            Segment * tmp = reinterpret_cast<Segment *>(&System::_preheap[0]);
            System::_heap_segment = new (tmp) Segment(HEAP_SIZE, WHITE, Segment::Flags::SYS);
            System::_heap = new (&System::_preheap[sizeof(Segment)])
                Heap(Address_Space(MMU::current()).attach(System::_heap_segment, Memory_Map::SYS_HEAP),
                    System::_heap_segment->size());
        } else
            System::_heap = new (&System::_preheap[0]) Heap(MMU::al                                    );
      :
    }
};

// SETUP
  :
    if(Traits<System>::multiheap) { // Application heap in data segment
        si->lm.app_data_size = MMU::align_page(si->lm.app_data_size);
        si->lm.app_stack = si->lm.app_data + si->lm.app_data_size;
        si->lm.app_data_size += MMU::align_page(Traits<Application>::STACK_SIZE);
        si->lm.app_heap = si->lm.app_data + si->lm.app_data_size;
        si->lm.app_data_size += MMU::align_page(Traits<Application>::HEAP_SIZE);
    }
  :
```

populate the heap with the Segment's memory



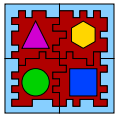
```
class Init_System
{
private:
    static const unsigned int HEAP_SIZE = Traits<System>::HEAP_SIZE;

public:
    Init_System() {
      :
        if(Traits<System>::multiheap) {
            Segment * tmp = reinterpret_cast<Segment *>(&System::_preheap[0]);
            System::_heap_segment = new (tmp) Segment(HEAP_SIZE, WHITE, Segment::Flags::SYS);
            System::_heap = new (&System::_preheap[sizeof(Segment)])
                Heap(Address_Space(MMU::current()).attach(System::_heap_segment, Memory_Map::SYS_HEAP),
                    System::_heap_segment->size());
        } else
            System::_heap = new (&System::_preheap[0]) Heap(MMU::alloc(MMU::pages(HEAP_SIZE)), HEAP_SIZE);
      :
    }
};

// SETUP
  :
    if(Traits<System>::multiheap) { // Application heap in data segment
        si->lm.app_data_size = MMU::align_page(si->lm.app_data_size);
        si->lm.app_stack = si->lm.app_data + si->lm.app_data_size;
        si->lm.app_data_size += MMU::align_page(Traits<Application>::STACK_SIZE);
        si->lm.app_heap = si->lm.app_data + si->lm.app_data_size;
        si->lm.app_data_size += MMU::align_page(Traits<Application>::HEAP_SIZE);
    }
  :
```

another nice usage for the placement new, isn't it?



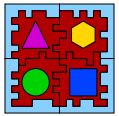
```
class Init_System
{
private:
    static const unsigned int HEAP_SIZE = Traits<System>::HEAP_SIZE;

public:
    Init_System() {
      :
        if(Traits<System>::multiheap) {
            Segment * tmp = reinterpret_cast<Segment *>(&System::_preheap[0]);
            System::_heap_segment = new (tmp) Segment(HEAP_SIZE, WHITE, Segment::Flags::SYS);
            System::_heap = new (&System::_preheap[sizeof(Segment)])
                Heap(Address_Space(MMU::current()).attach(System::_heap_segment, Memory_Map::SYS_HEAP),
                    System::_heap_segment->size());
        } else
            System::_heap = new (&System::_preheap[0]) Heap(MMU::alloc(MMU::pages(HEAP_SIZE)), HEAP_SIZE);
      :
    }
};

// SETUP
  :
    if(Traits<System>::multiheap) { // Application heap in data segment
        si->lm.app_data_size = MMU::align_page(si->lm.app_data_size);
        si->lm.app_stack = si->lm.app_data + si->lm.app_data_size;
        si->lm.app_data_size += MMU::align_page(Traits<Application>::STACK_SIZE);
        si->lm.app_heap = si->lm.app_data + si->lm.app_data_size;
        si->lm.app_data_size += MMU::align_page(Traits<Application>::HEAP_SIZE);
    }
  :
```

SETUP arranges a memory model for single-task configs

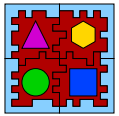


```
extern "C"
{
    // Standard C Library allocators
    inline void * malloc(size_t bytes) {
        __USING_SYS;
        if(Traits<System>::multiheap)
            return Application::_heap->alloc(bytes);
        else
            return System::_heap->alloc(bytes);
    }

    inline void * calloc(size_t n, unsigned int bytes) {
        void * ptr = malloc(n * bytes);
        memset(ptr, 0, n * bytes);
        return ptr;
    }

    inline void free(void * ptr) {
        __USING_SYS;
        if(Traits<System>::multiheap)
            Heap::typed_free(ptr);
        else
            Heap::untyped_free(System::_heap, ptr);
    }
}
```

```
extern "C"
{
    // Standard C Library allocators
    inline void * malloc(size_t bytes) {
        __USING_SYS;
        if(Traits<System>::multiheap)
            return Application::_heap->alloc(bytes);
        else
            return System::_heap->alloc(bytes);
    }

    inline void * calloc(size_t n, unsigned int bytes) {
        void * ptr = malloc(n * bytes);
        memset(ptr, 0, n * bytes);
        return ptr;
    }

    inline void free(void * ptr) {
        __USING_SYS;
        if(Traits<System>::multiheap)
            Heap::typed_free(ptr);
        else
            Heap::untyped_free(System::_heap, ptr);
    }
}
```

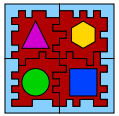


single-heap uses System::heap



```
extern "C"
{
    // Standard C Library allocators
    inline void * malloc(size_t bytes) {
        __USING_SYS;
        if(Traits<System>::multiheap)
            return Application::_heap->alloc(bytes);
        else
            return System::_heap->alloc(bytes);
    }

    inline void * calloc(size_t n, unsigned int bytes) {
        void * ptr = malloc(n * bytes);
        memset(ptr, 0, n * bytes);
        return ptr;
    }

    inline void free(void * ptr) {
        __USING_SYS;
        if(Traits<System>::multiheap)
            Heap::typed_free(ptr);
        else
            Heap::untyped_free(System::_heap, ptr);
    }
}
```

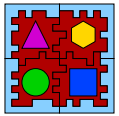
originally, delete took only ptr, so the type must be memorized



```
extern "C" { char _end; }

class Init_Application
{
private:
    static const unsigned int HEAP_SIZE = Traits<Application>::HEAP_SIZE;
    static const unsigned int STACK_SIZE = Traits<Application>::STACK_SIZE;

public:
    Init_Application() {
        if(Traits<System>::multiheap) {
            char * heap = MMU::align_page(&_end);
            if(Traits<Build>::MODE != Traits<Build>::KERNEL)
                heap += MMU::align_page(Traits<Application>::STACK_SIZE);
            Application::_heap = new (&Application::_preheap[0]) Heap(heap, HEAP_SIZE);
        } else
            for(unsigned int frames = MMU::allocable(); frames; frames = MMU::allocable())
                System::_heap->free(MMU::alloc(frames), frames * sizeof(MMU::Page));
        db<Init>(INF) << "done!" << endl;
    }
};
```

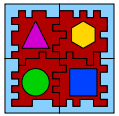
```
extern "C" { char _end; }

class Init_Application
{
private:
    static const unsigned int HEAP_SIZE = Traits<Application>::HEAP_SIZE;
    static const unsigned int STACK_SIZE = Traits<Application>::STACK_SIZE;

public:
    Init_Application() {
        if(Traits<System>::multiheap) {
            char * heap = MMU::align_page(&_end);
            if(Traits<Build>::MODE != Traits<Build>::KERNEL)
                heap += MMU::align_page(Traits<Application>::STACK_SIZE);
            Application::_heap = new (&Application::_preheap[0]) Heap(heap, HEAP_SIZE);
        } else
            for(unsigned int frames = MMU::allocable(); frames; frames = MMU::allocable())
                System::_heap->free(MMU::alloc(frames), frames * sizeof(MMU::Page));
        db<Init>(INF) << "done!" << endl;
    }
};
```

symbol defined by the compiler to indicate the end of the data segment



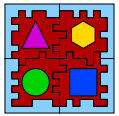
```
extern "C" { char _end; }

class Init_Application
{
private:
    static const unsigned int HEAP_SIZE = Traits<Application>::HEAP_SI
    static const unsigned int STACK_SIZE = Traits<Application>::STACK_SIZE;

public:
    Init_Application() {
        if(Traits<System>::multiheap) {
            char * heap = MMU::align_page(&_end);
            if(Traits<Build>::MODE != Traits<Build>::KERNEL)
                heap += MMU::align_page(Traits<Application>::STACK_SIZE);
            Application::_heap = new (&Application::_preheap[0]) Heap(heap, HEAP_SIZE);
        } else
            for(unsigned int frames = MMU::allocable(); frames; frames = MMU::allocable())
                System::_heap->free(MMU::alloc(frames), frames * sizeof(MMU::Page));
        db<Init>(INF) << "done!" << endl;
    }
};
```

heap in data segment arranged by SETUP



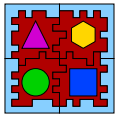
```
extern "C" { char _end; }

class Init_Application
{
private:
    static const unsigned int HEAP_SIZE = Traits<Application>::HEAP_SIZE;
    static const unsigned int STACK_SIZE = Traits<Application>::STACK_SIZE;

public:
    Init_Application() {
        if(Traits<System>::multiheap) {
            char * heap = MMU::align_page(&_end);
            if(Traits<Build>::MODE != Traits<Build>::KERNEL)
                heap += MMU::align_page(Traits<Application>::STACK_SIZE);
            Application::_heap = new (&Application::_preheap[0]) Heap(heap, HEAP_SIZE);
        } else
            for(unsigned int frames = MMU::allocable(); frames; frames = MMU::allocable())
                System::_heap->free(MMU::alloc(frames), frames * sizeof(MMU::Page));
        db<Init>(INF) << "done!" << endl;
    }
};
```

if not a kernel, then use the stack allocated by SETUP, otherwise make it part of the heap



```
extern "C" { char _end; }

class Init_Application
{
private:
    static const unsigned int HEAP_SIZE = Traits<Application>::HEAP_SIZE;
    static const unsigned int STACK_SIZE = Traits<Application>::STACK_SIZE;

public:
    Init_Application() {
        if(Traits<System>::multiheap) {
            char * heap = MMU::align_page(&_end);
            if(Traits<Build>::MODE != Traits<Build>::KERNEL)
                heap += MMU::align_page(Traits<Application>::STACK_SIZE);
            Application::_heap = new (&Application::_preheap[0]) Heap(heap, HEAP_SIZE);
        } else
            for(unsigned int frames = MMU::allocable(); frames; frames = MMU::allocable())
                System::_heap->free(MMU::alloc(frames), frames * sizeof(MMU::Page));
        db<Init>(INF) << "done!" << endl;
    }
};
```

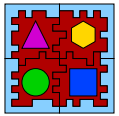
if !multiheap, put all free memory in the application s heap



```
// new allocators in types.h
void * operator new(size_t, const EPOS::System_Allocator &);
void * operator new[](size_t, const EPOS::System_Allocator &);

void * operator new(size_t, const EPOS::Scratchpad_Allocator &);
void * operator new[](size_t, const EPOS::Scratchpad_Allocator &);

// C++ dynamic memory allocators and deallocators
inline void * operator new(size_t bytes) {
    return malloc(bytes);
}

inline void * operator new[](size_t bytes) {
    return malloc(bytes);
}

inline void * operator new(size_t bytes, const EPOS::System_Allocator & allocator) {
    return _SYS::System::_heap->alloc(bytes);
}

inline void * operator new[](size_t bytes, const EPOS::System_Allocator & allocator) {
    return _SYS::System::_heap->alloc(bytes);
}

void operator delete(void * ptr);
void operator delete[](void * ptr);
void operator delete(void * ptr, size_t bytes);
void operator delete[](void * ptr, size_t bytes);
```

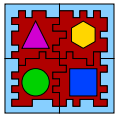
```
// new allocators in types.h
void * operator new(size_t, const EPOS::System_Allocator &);
void * operator new[](size_t, const EPOS::System_Allocator &);

void * operator new(size_t, const EPOS::Scratchpad_Allocator &);
void * operator new[](size_t, const EPOS::Scratchpad_Allocator &);

// C++ dynamic memory allocators and deallocators
inline void * operator new(size_t bytes) {
    return malloc(bytes);
}

inline void * operator new[](size_t bytes) {
    return malloc(bytes);
}

inline void * operator new(size_t bytes, const EPOS::System_Allocator & allocator) {
    return _SYS::System::_heap->alloc(bytes);
}

inline void * operator new[](size_t bytes, const EPOS::System_Allocator & allocator) {
    return _SYS::System::_heap->alloc(bytes);
}

void operator delete(void * ptr);
void operator delete[](void * ptr);
void operator delete(void * ptr, size_t bytes);
void operator delete[](void * ptr, size_t bytes);
```

placement new's second parameter is free (yet): overloading



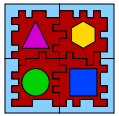
```
// new allocators in types.h
void * operator new(size_t, const EPOS::System_Allocator &);
void * operator new[](size_t, const EPOS::System_Allocator &);

void * operator new(size_t, const EPOS::Scratchpad_Allocator &);
void * operator new[](size_t, const EPOS::Scratchpad_Allocator &);

// C++ dynamic memory allocators and deallocators
inline void * operator new(size_t bytes) {
    return malloc(bytes);
}

inline void * operator new[](size_t bytes) {
    return malloc(bytes);
}

inline void * operator new(size_t bytes, const EPOS::System_Allocator & allocator) {
    return _SYS::System::_heap->alloc(bytes);
}

inline void * operator new[](size_t bytes, const EPOS::System_Allocator & allocator) {
    return _SYS::System::_heap->alloc(bytes);
}

void operator delete(void * ptr);
void operator delete[](void * ptr);
void operator delete(void * ptr, size_t bytes);
void operator delete[](void * ptr, size_t bytes);
```

applications get the plain new operator



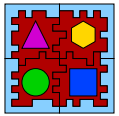
```
// new allocators in types.h
void * operator new(size_t, const EPOS::System_Allocator &);
void * operator new[](size_t, const EPOS::System_Allocator &);

void * operator new(size_t, const EPOS::Scratchpad_Allocator &);
void * operator new[](size_t, const EPOS::Scratchpad_Allocator &);

// C++ dynamic memory allocators and deallocators
inline void * operator new(size_t bytes) {
    return malloc(bytes);
}

inline void * operator new[](size_t bytes) {
    return malloc(bytes);
}

inline void * operator new(size_t bytes, const EPOS::System_Allocator
    return _SYS::System::_heap->alloc(bytes);
}

inline void * operator new[](size_t bytes, const EPOS::System_Allocator & allocator) {
    return _SYS::System::_heap->alloc(bytes);
}

void operator delete(void * ptr);
void operator delete[](void * ptr);
void operator delete(void * ptr, size_t bytes);
void operator delete[](void * ptr, size_t bytes);
```

system uses a typed one (SYSTEM)



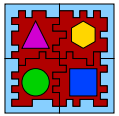
```
// new allocators in types.h
void * operator new(size_t, const EPOS::System_Allocator &);
void * operator new[](size_t, const EPOS::System_Allocator &);

void * operator new(size_t, const EPOS::Scratchpad_Allocator &);
void * operator new[](size_t, const EPOS::Scratchpad_Allocator &);

// C++ dynamic memory allocators and deallocators
inline void * operator new(size_t bytes) {
    return malloc(bytes);
}

inline void * operator new[](size_t bytes) {
    return malloc(bytes);
}

inline void * operator new(size_t bytes, const EPOS::System_Allocator & allocator) {
    return _SYS::System::_heap->alloc(bytes);
}

inline void * operator new[](size_t bytes, const EPOS::System_Allocator & allocator) {
    return _SYS::System::_heap->alloc(bytes);
}

void operator delete(void * ptr);
void operator delete[](void * ptr);
void operator delete(void * ptr, size_t bytes);
void operator delete[](void * ptr, size_t bytes);
```

ordinary delete



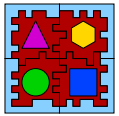
```
// new allocators in types.h
void * operator new(size_t, const EPOS::System_Allocator &);
void * operator new[](size_t, const EPOS::System_Allocator &);

void * operator new(size_t, const EPOS::Scratchpad_Allocator &);
void * operator new[](size_t, const EPOS::Scratchpad_Allocator &);

// C++ dynamic memory allocators and deallocators
inline void * operator new(size_t bytes) {
    return malloc(bytes);
}

inline void * operator new[](size_t bytes) {
    return malloc(bytes);
}

inline void * operator new(size_t bytes, const EPOS::System_Allocator & allocator) {
    return _SYS::System::_heap->alloc(bytes);
}

inline void * operator new[](size_t bytes, const EPOS::System_Allocator & allocator) {
    return _SYS::System::_heap->alloc(bytes);
}

void operator delete(void * ptr);
void operator delete[](void * ptr);
void operator delete(void * ptr, size_t bytes);
void operator delete[](void * ptr, size_t bytes);
```

delete cannot be declared inline because of virtual destructors

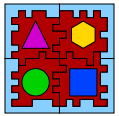


```
// new allocators in types.h
void * operator new(size_t, const EPOS::System_Allocator &);
void * operator new[](size_t, const EPOS::System_Allocator &);

void * operator new(size_t, const EPOS::Scratchpad_Allocator &);
void * operator new[](size_t, const EPOS::Scratchpad_Allocator &);

// C++ dynamic memory allocators and deallocators
inline void * operator new(size_t bytes) {
    return malloc(bytes);
}

inline void * operator new[](size_t bytes) {
    return malloc(bytes);
}

inline void * operator new(size_t bytes, const EPOS::System_Allocator & allocator) {
    return _SYS::System::_heap->alloc(bytes);
}

inline void * operator new[](size_t bytes, const EPOS::System_Allocator & allocator) {
    return _SYS::System::_heap->alloc(bytes);
}

void operator delete(void * ptr);
void operator delete[](void * ptr);
void operator delete(void * ptr, size_t bytes);
void operator delete[](void * ptr, size_t bytes);
```

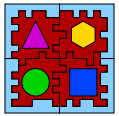
delete cannot be declared inline because of virtual destructors



```
// no more kmalloc!
_timer = new (SYSTEM) Scheduler_Timer(QUANTUM, time_slicer);

_stack = new (SYSTEM) char[stack_size];
delete _stack;

Thread::_running = new (SYSTEM) Thread(Thread::Configuration(Thread::RUNNING,
                                       Thread::MAIN),
                                       reinterpret_cast<int (*)()>(__epos_app_entry));
new (SYSTEM) Thread(Thread::Configuration(Thread::READY, Thread::IDLE), &Thread::idle);
```



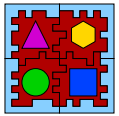
```
// no more kmalloc!
_timer = new (SYSTEM) Scheduler_Timer(QUANTUM, time_slicer);

_stack = new (SYSTEM) char[stack_size];
delete _stack;

Thread::_running = new (SYSTEM) Thread(Thread::Configuration(Thread::RUNNING,
                                       Thread::MAIN),
                                       reinterpret_cast<int (*)()>(__epos_app_entry));
new (SYSTEM) Thread(Thread::Configuration(Thread::READY, Thread::IDLE), &Thread::idle);
```

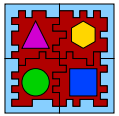


```
// no more kmalloc!
_timer = new (SYSTEM) Scheduler_Timer(QUANTUM, time_slicer);

_stack = new (SYSTEM) char[stack_size];
delete _stack;

Thread::_running = new (SYSTEM) Thread(Thread::Configuration(Thread::RUNNING,
                                       Thread::MAIN),
                                       reinterpret_cast<int (*)()>(__epos_app_entry));
new (SYSTEM) Thread(Thread::Configuration(Thread::READY, Thread::IDLE), &Thread::idle);
```

```
// no more kmalloc!
_timer = new (SYSTEM) Scheduler_Timer(QUANTUM, time_slicer);

_stack = new (SYSTEM) char[stack_size];
delete _stack;

Thread::_running = new (SYSTEM) Thread(Thread::Configuration(Thread::RUNNING,
                                       Thread::MAIN),
                                       reinterpret_cast<int (*)()>(__epos_app_entry));
new (SYSTEM) Thread(Thread::Configuration(Thread::READY, Thread::IDLE), &Thread::idle);
```

no more kmalloc()
typed new() instead!





Embedded Parallel Operating System

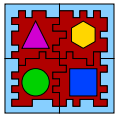
Coding Journey through OS Design
–from co-routines to a multicore kernel–

Prof. Antônio Augusto Fröhlich, Ph.D.

UFSC / LISHA

September 30, 2020

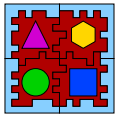
True Idle Waiting: Time Management



Calling Thread::yield() whenever a condition is not met isn’t actually “busy-waiting”, but we now can easily block and unblock threads to achieve a “full idle-waiting” system

Our timer tick handler in Alarm is not re-entrant

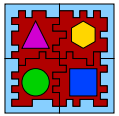
- -If a handler executes for longer than a tick, events may be lost

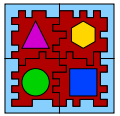


```
class Alarm
{
private:
    typedef Timer::Tick Tick;
    typedef Relative_Queue<Alarm, Tick> Queue;

public:
    typedef TSC::Hertz Hertz;
    typedef RTC::Microsecond Microsecond;
    enum { INFINITE = RTC::INFINITE };

public:
    Alarm(const Microsecond & time, Handler * handler, int times = 1);
    ~Alarm();

    const Microsecond & period() const { return _time; }
    static Hertz frequency() { return _timer->frequency(); }

    static void delay(const Microsecond & time);

private:
    static Microsecond timer_period() {
        return 1000000 / frequency();
    }
    static Tick ticks(const Microsecond & time) {
        return (time + timer_period() / 2) / timer_period();
    }

    static void lock() { Thread::lock(); }
    static void unlock() { Thread::unlock(); }

    static void handler(const IC::Interrupt_Id & i);
```

```
class Alarm
{
private:
    typedef Timer::Tick Tick;
    typedef Relative_Queue<Alarm, Tick> Queue;

public:
    typedef TSC::Hertz Hertz;
    typedef RTC::Microsecond Microsecond;
    enum { INFINITE = RTC::INFINITE };

public:
    Alarm(const Microsecond & time, Handler * handler, int times = 1);
    ~Alarm();

    const Microsecond & period() const { return _time; }
    static Hertz frequency() { return _timer->frequency(); }

    static void delay(const Microsecond & time);

private:
    static Microsecond timer_period() {
        return 1000000 / frequency();
    }
    static Tick ticks(const Microsecond & time) {
        return (time + timer_period() / 2) / timer_period();
    }

    static void lock() { Thread::lock(); }
    static void unlock() { Thread::unlock(); }

    static void handler(const IC::Interrupt_Id & i);
```

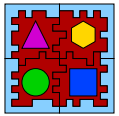
tick-based timer





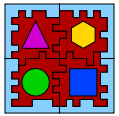
```
class Alarm
{
private:
    typedef Timer::Tick Tick;
    typedef Relative_Queue<Alarm, Tick> Queue;

public:
    typedef TSC::Hertz Hertz;
    typedef RTC::Microsecond Microsecond;
    enum { INFINITE = RTC::INFINITE };

public:
    Alarm(const Microsecond & time, Handler * handler, int times = 1);
    ~Alarm();

    const Microsecond & period() const { return _time; }
    static Hertz frequency() { return _timer->frequency(); }

    static void delay(const Microsecond & time);

private:
    static Microsecond timer_period() {
        return 1000000 / frequency();
    }
    static Tick ticks(const Microsecond & time) {
        return (time + timer_period() / 2) / timer_period();
    }

    static void lock() { Thread::lock(); }
    static void unlock() { Thread::unlock(); }

    static void handler(const IC::Interrupt_Id & i);
```

Relative_Queue is an ordered queue in which each element's rank is an offset to the previous



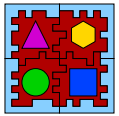
```
class Alarm
{
private:
    typedef Timer::Tick Tick;
    typedef Relative_Queue<Alarm, Tick> Queue;

public:
    typedef TSC::Hertz Hertz;
    typedef RTC::Microsecond Microsecond;
    enum { INFINITE = RTC::INFINITE };

public:
    Alarm(const Microsecond & time, Handler * handler, int times = 1);
    ~Alarm();

    const Microsecond & period() const { return _time; }
    static Hertz frequency() { return _timer->frequency(); }

    static void delay(const Microsecond & time);

private:
    static Microsecond timer_period() {
        return 1000000 / frequency();
    }
    static Tick ticks(const Microsecond & time) {
        return (time + timer_period() / 2) / timer_period();
    }

    static void lock() { Thread::lock(); }
    static void unlock() { Thread::unlock(); }

    static void handler(const IC::Interrupt_Id & i);
```

import, don't redefine!



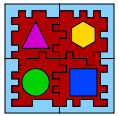
```
class Alarm
{
private:
    typedef Timer::Tick Tick;
    typedef Relative_Queue<Alarm, Tick> Queue;

public:
    typedef TSC::Hertz Hertz;
    typedef RTC::Microsecond Microsecond;
    enum { INFINITE = RTC::INFINITE };

public:
    Alarm(const Microsecond & time, Handler * handler, int times = 1);
    ~Alarm();

    const Microsecond & period() const { return _time; }
    static Hertz frequency() { return _timer->frequency(); }

    static void delay(const Microsecond & time);

private:
    static Microsecond timer_period() {
        return 1000000 / frequency();
    }
    static Tick ticks(const Microsecond & time) {
        return (time + timer_period() / 2) / timer_period();
    }

    static void lock() { Thread::lock(); }
    static void unlock() { Thread::unlock(); }

    static void handler(const IC::Interrupt_Id & i);
```

periodically invoked handler



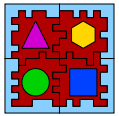
```
class Alarm
{
private:
    typedef Timer::Tick Tick;
    typedef Relative_Queue<Alarm, Tick> Queue;

public:
    typedef TSC::Hertz Hertz;
    typedef RTC::Microsecond Microsecond;
    enum { INFINITE = RTC::INFINITE };

public:
    Alarm(const Microsecond & time, Handler * handler, int times = 1);
    ~Alarm();

    const Microsecond & period() const { return _time; }
    static Hertz frequency() { return _timer->frequency(); }

    static void delay(const Microsecond & time);

private:
    static Microsecond timer_period() {
        return 1000000 / frequency();
    }
    static Tick ticks(const Microsecond & time) {
        return (time + timer_period() / 2) / timer_period();
    }

    static void lock() { Thread::lock(); }
    static void unlock() { Thread::unlock(); }

    static void handler(const IC::Interrupt_Id & i);
```

function to wait for a given time



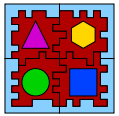
```
class Alarm
{
private:
    typedef Timer::Tick Tick;
    typedef Relative_Queue<Alarm, Tick> Queue;

public:
    typedef TSC::Hertz Hertz;
    typedef RTC::Microsecond Microsecond;
    enum { INFINITE = RTC::INFINITE };

public:
    Alarm(const Microsecond & time, Handler * handler, int times = 1);
    ~Alarm();

    const Microsecond & period() const { return _time; }
    static Hertz frequency() { return _timer->frequency(); }

    static void delay(const Microsecond & time);

private:
    static Microsecond timer_period() {
        return 1000000 / frequency();
    }
    static Tick ticks(const Microsecond & time) {
        return (time + timer_period() / 2) / timer_period();
    }

    static void lock() { Thread::lock(); }
    static void unlock() { Thread::unlock(); }

    static void handler(const IC::Interrupt_Id & i);
```

preemption control

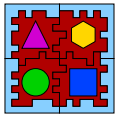


```
class Alarm
{
private:
    typedef Timer::Tick Tick;
    typedef Relative_Queue<Alarm, Tick> Queue;

public:
    typedef TSC::Hertz Hertz;
    typedef RTC::Microsecond Microsecond;
    enum { INFINITE = RTC::INFINITE };

public:
    Alarm(const Microsecond & time, Handler * handler, int times = 1);
    ~Alarm();

    const Microsecond & period() const { return _time; }
    static Hertz frequency() { return _timer->frequency(); }

    static void delay(const Microsecond & time);

private:
    static Microsecond timer_period() {
        return 1000000 / frequency();
    }
    static Tick ticks(const Microsecond & time) {
        return (time + timer_period() / 2) / timer_period();
    }

    static void lock() { Thread::lock(); }
    static void unlock() { Thread::unlock(); }

    static void handler(const IC::Interrupt_Id & i);
```

(peseudo) interrupt handler

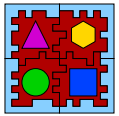


```
private:
    Microsecond _time;
    Handler * _handler;
    int _times;
    Tick _ticks;
    Queue::Element _link;

    static Alarm_Timer * _timer;
    static volatile Tick _elapsed;
    static Queue _request;
};

Alarm::Alarm(const Microsecond & time, Handler * handler, unsigned int times)
: _time(time), _handler(handler), _times(times), _ticks(ticks(time)), _link(this, _ticks)
{
    lock();

    db<Alarm>(TRC) << "Alarm(t=" << time << ",tk=" << _ticks
        << ",h=" << reinterpret_cast<void *>(handler)
        << ",x=" << times << ") => " << this << endl;

    if(_ticks) {
        _request.insert(&_link);
        unlock();
    } else {
        unlock();
        (*handler)();
    }
}
```



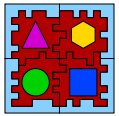
```
private:
    Microsecond _time;
    Handler * _handler;
    int _times;
    Tick _ticks;
    Queue::Element _link;

    static Alarm_Timer * _timer;
    static volatile Tick _elapsed;
    static Queue _request;
};

Alarm::Alarm(const Microsecond & time, Handler * handler, unsigned int times)
: _time(time), _handler(handler), _times(times), _ticks(ticks(time)), _link(this, _ticks)
{
    lock();

    db<Alarm>(TRC) << "Alarm(t=" << time << ",tk=" << _ticks
        << ",h=" << reinterpret_cast<void *>(handler)
        << ",x=" << times << ") => " << this << endl;

    if(_ticks) {
        _request.insert(&_link);
        unlock();
    } else {
        unlock();
        (*handler)();
    }
}
```

tick counter (overflow?)



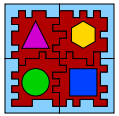
```
private:
    Microsecond _time;
    Handler * _handler;
    int _times;
    Tick _ticks;
    Queue::Element _link;

    static Alarm_Timer * _timer;
    static volatile Tick _elapsed;
    static Queue _request;
};

Alarm::Alarm(const Microsecond & time, Handler * handler, unsigned int times)
: _time(time), _handler(handler), _times(times), _ticks(ticks(time)), _link(this, _ticks)
{
    lock();

    db<Alarm>(TRC) << "Alarm(t=" << time << ",tk=" << _ticks
        << ",h=" << reinterpret_cast<void *>(handler)
        << ",x=" << times << ") => " << this << endl;

    if(_ticks) {
        _request.insert(&_link);
        unlock();
    } else {
        unlock();
        (*handler)();
    }
}
```

insert pending event in the queue if > tick



```
private:
    Microsecond _time;
    Handler * _handler;
    int _times;
    Tick _ticks;
    Queue::Element _link;

    static Alarm_Timer * _timer;
    static volatile Tick _elapsed;
    static Queue _request;
};

Alarm::Alarm(const Microsecond & time, Handler * handler, unsigned int times)
: _time(time), _handler(handler), _times(times), _ticks(ticks(time)), _link(this, _ticks)
{
    lock();

    db<Alarm>(TRC) << "Alarm(t=" << time << ",tk=" << _ticks
        << ",h=" << reinterpret_cast<void *>(handler)
        << ",x=" << times << ") => " << this << endl;

    if(_ticks) {
        _request.insert(&_link);
        unlock();
    } else {
        unlock();
        (*handler)();
    }
}
```

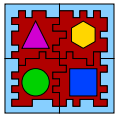
dispatch handler immediately otherwise



```
Alarm::~Alarm()
{
    lock();
    _request.remove(this);
    unlock();
}


void Timer::int_handler(const Interrupt_Id & i)
{
    if(_channels[ALARM]) {
        _channels[ALARM]->_current[0] = _channels[ALARM]->_initial;
        _channels[ALARM]->_handler(i);
    }
  :
}
```

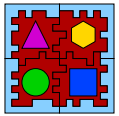
```
Alarm::~Alarm()
{
    lock();
    _request.remove(this);
    unlock();
}


void Timer::int_handler(const Interrupt_Id & i)
{
    if(_channels[ALARM]) {
        _channels[ALARM]->_current[0] = _channels[ALARM]->_initial;
        _channels[ALARM]->_handler(i);
    }
  :
}
```

destructor just remove the alarm from the queue



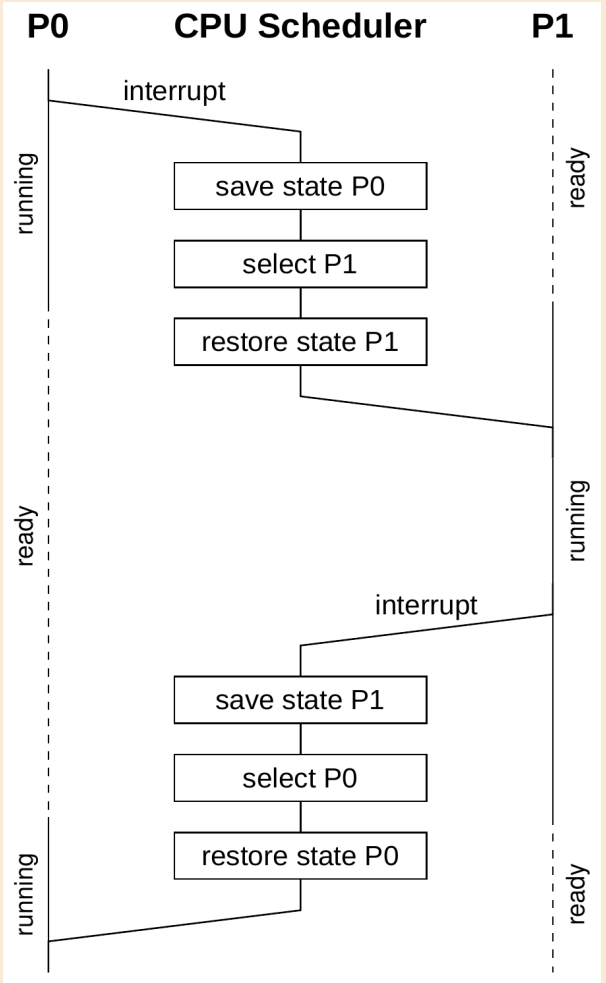
```
Alarm::~Alarm()
{
    lock();
    _request.remove(this);
    unlock();
}


void Timer::int_handler(const Interrupt_Id & i)
{
    if(_channels[ALARM]) {
        _channels[ALARM]->_current[0] = _channels[ALARM]->_initial;
        _channels[ALARM]->_handler(i);
    }
  :
}
```

real interrupt handler for PC, which multiplex logical timers

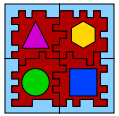


```
Alarm::~Alarm()
{
    lock();
    _request.remove(this);
    unlock();
}


void Timer::int_handler(const Interrupt_Id & i)
{
    if(_channels[ALARM]) {
        _channels[ALARM]->_current[0] = _channels[ALARM]->_initial;
        _channels[ALARM]->_handler(i);
    }
  :
}
```

Time-triggered Reschedule



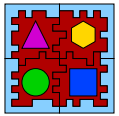

```
_Id & i)

_channels[ALARM]->_initial;
```

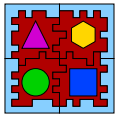


```
// Current (bad) handler
void Alarm::handler(const IC::Interrupt_Id & i)
{
    static Tick next_tick;
    static Handler * next_handler;

    lock();

    _elapsed++;

    if(next_tick)
        next_tick--;
    if(!next_tick) {
        if(next_handler)
            (*next_handler)();
        if(_request.empty())
            next_handler = 0;
        else {
            Queue::Element * e = _request.remove();
            Alarm * alarm = e->object();
            next_tick = alarm->_ticks;
            next_handler = alarm->_handler;
            if(alarm->_times != INFINITE)
                alarm->_times--;
            if(alarm->_times) {
                e->rank(alarm->_ticks);
                _request.insert(e);
            }
        }
    }
    unlock();
}
```

```
// Current (bad) handler
void Alarm::handler(const IC::Interrupt_Id & i)
{
    static Tick next_tick;
    static Handler * next_handler;

    lock();

    _elapsed++;

    if(next_tick)
        next_tick--;
    if(!next_tick) {
        if(next_handler)
            (*next_handler)();
        if(_request.empty())
            next_handler = 0;
        else {
            Queue::Element * e = _request.remove();
            Alarm * alarm = e->object();
            next_tick = alarm->_ticks;
            next_handler = alarm->_handler;
            if(alarm->_times != INFINITE)
                alarm->_times--;
            if(alarm->_times) {
                e->rank(alarm->_ticks);
                _request.insert(e);
            }
        }
    }
    unlock();
}
```

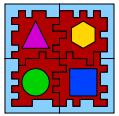
static variables!





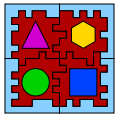
```
// Current (bad) handler
void Alarm::handler(const IC::Interrupt_Id & i)
{
    static Tick next_tick;
    static Handler * next_handler;

    lock();

    _elapsed++;

    if(next_tick)
        next_tick--;
    if(!next_tick) {
        if(next_handler)
            (*next_handler)();
        if(_request.empty())
            next_handler = 0;
        else {
            Queue::Element * e = _request.remove();
            Alarm * alarm = e->object();
            next_tick = alarm->_ticks;
            next_handler = alarm->_handler;
            if(alarm->_times != INFINITE)
                alarm->_times--;
            if(alarm->_times) {
                e->rank(alarm->_ticks);
                _request.insert(e);
            }
        }
    }
    unlock();
}
```

tick counter increment



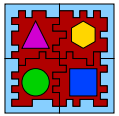
```
// Current (bad) handler
void Alarm::handler(const IC::Interrupt_Id & i)
{
    static Tick next_tick;
    static Handler * next_handler;

    lock();

    _elapsed++;

    if(next_tick)
        next_tick--;
    if(!next_tick) {
        if(next_handler)
            (*next_handler)();
        if(_request.empty())
            next_handler = 0;
        else {
            Queue::Element * e = _request.remove();
            Alarm * alarm = e->object();
            next_tick = alarm->_ticks;
            next_handler = alarm->_handler;
            if(alarm->_times != INFINITE)
                alarm->_times--;
            if(alarm->_times) {
                e->rank(alarm->_ticks);
                _request.insert(e);
            }
        }
    }
    unlock();
}
```

down stepping until handler is called
what happens if next_handler has an infinite loop?



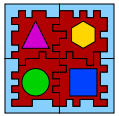
```
// Current (bad) handler
void Alarm::handler(const IC::Interrupt_Id & i)
{
    static Tick next_tick;
    static Handler * next_handler;

    lock();

    _elapsed++;

    if(next_tick)
        next_tick--;
    if(!next_tick) {
        if(next_handler)
            (*next_handler)();
        if(_request.empty())
            next_handler = 0;
        else {
            Queue::Element * e = _request.remove();
            Alarm * alarm = e->object();
            next_tick = alarm->_ticks;
            next_handler = alarm->_handler;
            if(alarm->_times != INFINITE)
                alarm->_times--;
            if(alarm->_times) {
                e->rank(alarm->_ticks);
                _request.insert(e);
            }
        }
    }
    unlock();
}
```

next alarm handling



```
// Current (bad) handler
void Alarm::handler(const IC::Interrupt_Id & i)
{
    static Tick next_tick;
    static Handler * next_handler;

    lock();

    _elapsed++;

    if(next_tick)
        next_tick--;
    if(!next_tick) {
        if(next_handler)
            (*next_handler)();
        if(_request.empty())
            next_handler = 0;
        else {
            Queue::Element * e = _request.remove();
            Alarm * alarm = e->object();
            next_tick = alarm->_ticks;
            next_handler = alarm->_handler;
            if(alarm->_times != INFINITE)
                alarm->_times--;
            if(alarm->_times) {
                e->rank(alarm->_ticks);
                _request.insert(e);
            }
        }
    }
    unlock();
}
```

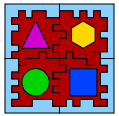
periodic alarm handling

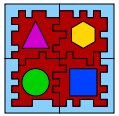


```
// new handler
void Alarm::handler(const IC::Interrupt_Id & i)
{
    lock();

    _elapsed++;

    Alarm * alarm = 0;

    if(!_request.empty()) {
        if(_request.head()->promote() <= 0) {
            Queue::Element * e = _request.remove();
            alarm = e->object();
            if(alarm->_times != INFINITE)
                alarm->_times--;
            if(alarm->_times) {
                e->rank(alarm->_ticks);
                _request.insert(e);
            }
        }
    }

    unlock();

    if(alarm)
        (*alarm->_handler)();
}
```

```
// new handler
void Alarm::handler(const IC::Interrupt_Id & i)
{
    lock();

    _elapsed++;

    Alarm * alarm = 0;

    if(!_request.empty()) {
        if(_request.head()->promote() <= 0) {
            Queue::Element * e = _request.remove();
            alarm = e->object();
            if(alarm->_times != INFINITE)
                alarm->_times--;
            if(alarm->_times) {
                e->rank(alarm->_ticks);
                _request.insert(e);
            }
        }
    }

    unlock();

    if(alarm)
        (*alarm->_handler)();
}
```

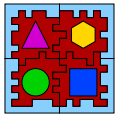
first reentrance rule of thumb: local context only!





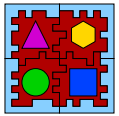
```
// new handler
void Alarm::handler(const IC::Interrupt_Id & i)
{
    lock();

    _elapsed++;

    Alarm * alarm = 0;

    if(!_request.empty()) {
        if(_request.head()->promote() <= 0) {
            Queue::Element * e = _request.remove();
            alarm = e->object();
            if(alarm->_times != INFINITE)
                alarm->_times--;
            if(alarm->_times) {
                e->rank(alarm->_ticks);
                _request.insert(e);
            }
        }
    }

    unlock();

    if(alarm)
        (*alarm->_handler)();
}
```

replacing the “if" by a "while" is tempting, but recovering the lock and dispatching the handler is troublesome if the Alarm gets destroyed in between



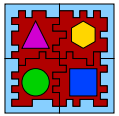
```
// new handler
void Alarm::handler(const IC::Interrupt_Id & i)
{
    lock();

    _elapsed++;

    Alarm * alarm = 0;

    if(!_request.empty()) {
        if(_request.head()->promote() <= 0) {
            Queue::Element * e = _request.remove();
            alarm = e->object();
            if(alarm->_times != INFINITE)
                alarm->_times--;
            if(alarm->_times) {
                e->rank(alarm->_ticks);
                _request.insert(e);
            }
        }
    }

    unlock();

    if(alarm)
        (*alarm->_handler)();
}
```

rank can be negative whenever multiple handlers get created for the same time tick



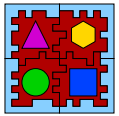
```
// new handler
void Alarm::handler(const IC::Interrupt_Id & i)
{
    lock();

    _elapsed++;

    Alarm * alarm = 0;

    if(!_request.empty()) {
        if(_request.head()->promote() <= 0) {
            Queue::Element * e = _request.remove();
            alarm = e->object();
            if(alarm->_times != INFINITE)
                alarm->_times--;
            if(alarm->_times) {
                e->rank(alarm->_ticks);
                _request.insert(e);
            }
        }
    }

    unlock();

    if(alarm)
        (*alarm->_handler)();
}
```

handling periodic alarms



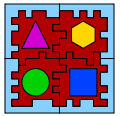
```
// new handler
void Alarm::handler(const IC::Interrupt_Id & i)
{
    lock();

    _elapsed++;

    Alarm * alarm = 0;

    if(!_request.empty()) {
        if(_request.head()->promote() <= 0) {
            Queue::Element * e = _request.remove();
            alarm = e->object();
            if(alarm->_times != INFINITE)
                alarm->_times--;
            if(alarm->_times) {
                e->rank(alarm->_ticks);
                _request.insert(e);
            }
        }
    }

    unlock();

    if(alarm)
        (*alarm->_handler)();
}
```

second reentance rule of thumb:
release locks before dispatching!



```
// new handler
void Alarm::handler(const IC::Interrupt_Id & i)
{
    lock();

    _elapsed++;

    Alarm * alarm = 0;

    if(!_request.empty()) {
        if(_request.head()->promote() <= 0) {
            Queue::Element * e = _request.remove();
            alarm = e->object();
            if(alarm->_times != INFINITE)
                alarm->_times--;
            if(alarm->_times) {
                e->rank(alarm->_ticks);
                _request.insert(e);
            }
        }
    }

    unlock();

    if(alarm)
        (*alarm->_handler)();
}
```

```
class Handler
{
public:
    // A handler function
    typedef void (Function)();

public:
    Handler() {}
    virtual ~Handler() {}

    virtual void operator()() = 0;
};

class Function_Handler: public Handler
{
public:
    Function_Handler(Function * h): _handler(h) {}
    ~Function_Handler() {}

    void operator()() { _handler(); }

private:
    Function * _handler;
};
```

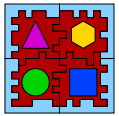


```
template<typename T>
class Functor_Handler: public Handler
{
public:
    typedef void (Functor)(T *);

public:
    Functor_Handler(Functor * h, T * p):
        _handler(h), _ptr(p) {}
    ~Functor_Handler() {}

    void operator()() { _handler(_ptr); }

private:
    Functor * _handler;
    T * _ptr;
};

class Thread_Handler : public Handler
{
public:
    Thread_Handler(Thread * h) : _handler(h) {}
    ~Thread_Handler() {}

    void operator()() { _handler->resume(); }

private:
    Thread * _handler;
};
```

```
class Handler
{
public:
    // A handler function
    typedef void (Function)();

public:
    Handler() {}
    virtual ~Handler() {}

    virtual void operator()() = 0;
};

class Function_Handler: public Handler
{
public:
    Function_Handler(Function * h): _handler(h) {}
    ~Function_Handler() {}

    void operator()() { _handler(); }

private:
    Function * _handler;
};
```

first polymorphic class in our design (abstract, indeed)

(don't try do yourself what the compiler can do better)

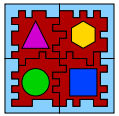


```
        Functor_Handler(Functor * h, T * p):
            _handler(h), _ptr(p) {}
        ~Functor_Handler() {}

        void operator()() { _handler(_ptr); }

    private:
        Functor * _handler;
        T * _ptr;
    };

    class Thread_Handler : public Handler
    {
    public:
        Thread_Handler(Thread * h) : _handler(h) {}
        ~Thread_Handler() {}

        void operator()() { _handler->resume(); }

    private:
        Thread * _handler;
    };
```

```
class Handler
{
public:
    // A handler function
    typedef void (Function)();

public:
    Handler() {}
    virtual ~Handler() {}

    virtual void operator()() = 0;
};

class Function_Handler: public Handler
{
public:
    Function_Handler(Function * h): _handler(h) {}
    ~Function_Handler() {}

    void operator()() { _handler(); }

private:
    Function * _handler;
};
```

overload of the call operator

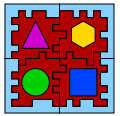
```
template<typename T>
class Functor_Handler: public Handler
{
public:
    typedef void (Functor)(T *);

public:
    Functor_Handler(Functor * h, T * p):

    void operator()() { _handler(_ptr); }

private:
    Functor * _handler;
    T * _ptr;
};

class Thread_Handler : public Handler
{
public:
    Thread_Handler(Thread * h) : _handler(h) {}
    ~Thread_Handler() {}

    void operator()() { _handler->resume(); }

private:
    Thread * _handler;
};
```



```
class Handler
{
public:
    // A handler function
    typedef void (Function)();

public:
    Handler() {}
    virtual ~Handler() {}

    virtual void operator()() = 0;
};

class Function_Handler: public Handler
{
public:
    Function_Handler(Function * h): _handler(h) {}
    ~Function_Handler() {}

    void operator()() { _handler(); }

private:
    Function * _handler;
};
```

```
template<typename T>
class Functor_Handler: public Handler
{
public:
    typedef void (Functor)(T *);

public:
    Functor_Handler(Functor * h, T * p):
        _handler(h), _ptr(p) {}
    ~Functor_Handler() {}

    void operator()() { _handler(_ptr); }

priv

};

class Thread_Handler : public Handler
{
public:
    Thread_Handler(Thread * h) : _handler(h) {}
    ~Thread_Handler() {}

    void operator()() { _handler->resume(); }

private:
    Thread * _handler;
};
```

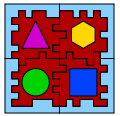
a simple indirect function call



a traditional Functor
(should be variadic?)

```
    typedef void (Function)();

public:
    Handler() {}
    virtual ~Handler() {}

    virtual void operator()() = 0;
};

class Function_Handler: public Handler
{
public:
    Function_Handler(Function * h): _handler(h) {}
    ~Function_Handler() {}

    void operator()() { _handler(); }

private:
    Function * _handler;
};
```

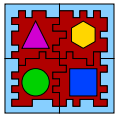
```
template<typename T>
class Functor_Handler: public Handler
{
public:
    typedef void (Functor)(T *);

public:
    Functor_Handler(Functor * h, T * p):
        _handler(h), _ptr(p) {}
    ~Functor_Handler() {}

    void operator()() { _handler(_ptr); }

private:
    Functor * _handler;
    T * _ptr;
};

class Thread_Handler : public Handler
{
public:
    Thread_Handler(Thread * h) : _handler(h) {}
    ~Thread_Handler() {}

    void operator()() { _handler->resume(); }

private:
    Thread * _handler;
};
```



```
class Handler
{
public:
    // A handler function
    typedef void (Function)();

public:
    Handler() {}
    virtual ~Handler() {}

    virtual void operator()() = 0;
};

class Function_Handler: public Handler
{
public:
    Function_Handler(Function * h): _handler(h) {}
    ~Function_Handler() {}

    void operator()() { _handler(); }
```

handler calls Thread::resume()

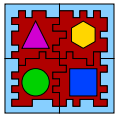
```
template<typename T>
class Functor_Handler: public Handler
{
public:
    typedef void (Functor)(T *);

public:
    Functor_Handler(Functor * h, T * p):
        _handler(h), _ptr(p) {}
    ~Functor_Handler() {}

    void operator()() { _handler(_ptr); }

private:
    Functor * _handler;
    T * _ptr;
};

class Thread_Handler : public Handler
{
public:
    Thread_Handler(Thread * h) : _handler(h) {}
    ~Thread_Handler() {}

    void operator()() { _handler->resume(); }

private:
    Thread * _handler;
};
```



```
class Handler
{
public:
    // A handler function
    typedef void (Function)();

public:
    Handler() {}
    virtual ~Handler() {}

    virtual void operator()() = 0;
};

class Function_Handler: public Handler
{
public:
    Function_Handler(Function * h): _handler(h) {}
    ~Function_Handler() {}

    void operator()() { _handler(); }

private:
    Function * _handler;
};
```

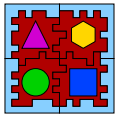
programmer calls Thread::suspend()

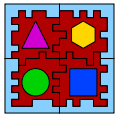
```
template<typename T>
class Functor_Handler: public Handler
{
public:
    typedef void (Functor)(T *);

public:
    Functor_Handler(Functor * h, T * p):
        _handler(h), _ptr(p) {}
    ~Functor_Handler() {}

    void operator()() { _handler(_ptr); }

private:
    Functor * _handler;
    T * _ptr;
};

class Thread_Handler : public Handler
{
public:
    Thread_Handler(Thread * h) : _handler(h) {}
    ~Thread_Handler() {}

    void operator()() { _handler->resume(); }

private:
    Thread * _handler;
};
```



```
class Mutex_Handler: public Handler
{
public:
    Mutex_Handler(Mutex * h) : _handler(h) {}
    ~Mutex_Handler() {}

    void operator()() { _handler->unlock(); }

private:
    Mutex * _handler;
};


class Semaphore_Handler: public Handler
{
public:
    Semaphore_Handler(Semaphore * h) : _handler(h)
{}
    ~Semaphore_Handler() {}

    void operator()() { _handler->v(); }

private:
    Semaphore * _handler;
};
```

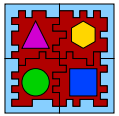
handler calls Mutex::unlock()





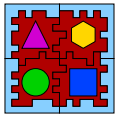
```
class Mutex_Handler: public Handler
{
public:
    Mutex_Handler(Mutex * h) : _handler(h) {}
    ~Mutex_Handler() {}

    void operator()() { _handler->unlock(); }

private:
    Mutex * _handler;
};


class Semaphore_Handler: public Handler
{
public:
    Semaphore_Handler(Semaphore * h) : _handler(h)
{}
    ~Semaphore_Handler() {}

    void operator()() { _handler->v(); }

private:
    Semaphore * _handler;
};
```

programmer calls Mutex::lock()



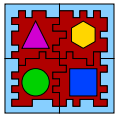
```
class Mutex_Handler: public Handler
{
public:
    Mutex_Handler(Mutex * h) : _handler(h) {}
    ~Mutex_Handler() {}

    void operator()() { _handler->unlock(); }

private:
    Mutex * _handler;
};


class Semaphore_Handler: public Handler
{
public:
    Semaphore_Handler(Semaphore * h) : _handler(h)
{}
    ~Semaphore_Handler() {}

    void operator()() { _handler->v(); }

private:
    Semaphore * _handler;
};
```

handler calls Semaphore::v()

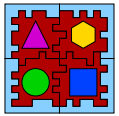


```
class Mutex_Handler: public Handler
{
public:
    Mutex_Handler(Mutex * h) : _handler(h) {}
    ~Mutex_Handler() {}

    void operator()() { _handler->unlock(); }

private:
    Mutex * _handler;
};


class Semaphore_Handler: public Handler
{
public:
    Semaphore_Handler(Semaphore * h) : _handler(h)
{}
    ~Semaphore_Handler() {}

    void operator()() { _handler->v(); }

private:
    Semaphore * _handler;
};
```

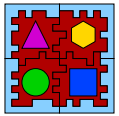
programmer calls Semaphore::p()



```
class Mutex_Handler: public Handler
{
public:
    Mutex_Handler(Mutex * h) : _handler(h) {}
    ~Mutex_Handler() {}

    void operator()() { _handler->unlock(); }

private:
    Mutex * _handler;
};


class Semaphore_Handler: public Handler
{
public:
    Semaphore_Handler(Semaphore * h) : _handler(h)
{}
    ~Semaphore_Handler() {}

    void operator()() { _handler->v(); }

private:
    Semaphore * _handler;
};
```



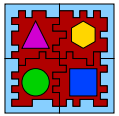
```
// current handler
void Alarm::delay(const Microsecond & time)
{
    db<Alarm>(TRC) << "Alarm::delay(time="
                   << time << ")" << endl;

    Tick t = _elapsed + ticks(time);

    while(_elapsed < t);
}


// new handler
void Alarm::delay(const Microsecond & time)
{
    db<Alarm>(TRC) << "Alarm::delay(time="
                   << time << ")" << endl;

    Semaphore semaphore(0);
    Semaphore_Handler handler(&semaphore);
    Alarm alarm(time, &handler, 1);
    semaphore.p();
}
```



```
// current handler
void Alarm::delay(const Microsecond & time)
{
    db<Alarm>(TRC) << "Alarm::delay(time="
                   << time << ")" << endl;

    Tick t = _elapsed + ticks(time);

    while(_elapsed < t);
}
```

busy-waiting!

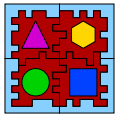
```
// new handler
void Alarm::delay(const Microsecond & time)
{
    db<Alarm>(TRC) << "Alarm::delay(time="
                   << time << ")" << endl;

    Semaphore semaphore(0);
    Semaphore_Handler handler(&semaphore);
    Alarm alarm(time, &handler, 1);
    semaphore.p();
}
```



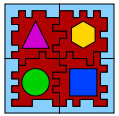
```
// current handler
void Alarm::delay(const Microsecond & time)
{
    db<Alarm>(TRC) << "Alarm::delay(time="
                   << time << ")" << endl;

    Tick t = _elapsed + ticks(time);

    while(_elapsed < t);
}


// new handler
void Alarm::delay(const Microsecond & time)
{
    db<Alarm>(TRC) << "Alarm::delay(time="
                   << time << ")" << endl;

    Semaphore semaphore(0);
    Semaphore_Handler handler(&semaphore);
    Alarm alarm(time, &handler, 1);
    semaphore.p();
}
```

Semaphore has memory!!!

```
// current handler
void Alarm::delay(const Microsecond & time)
{
    db<Alarm>(TRC) << "Alarm::delay(time="
                   << time << ")" << endl;

    Tick t = _elapsed + ticks(time);

    while(_elapsed < t);
}


// new handler
void Alarm::delay(const Microsecond & time)
{
    db<Alarm>(TRC) << "Alarm::delay(time="
                   << time << ")" << endl;

    Semaphore semaphore(0);
    Semaphore_Handler handler(&semaphore);
    Alarm alarm(time, &handler, 1);
    semaphore.p();
}
```



if time < tick, then call v()



Embedded Parallel Operating System

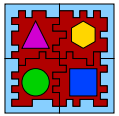
Coding Journey through OS Design
–from co-routines to a multicore kernel–

Prof. Antônio Augusto Fröhlich, Ph.D.

UFSC / LISHA

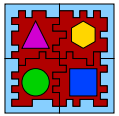
September 22, 2020

What about preemption?

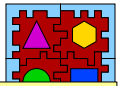


We have now an OS that does blocking synchronization
- But Threads only leave the CPU when they block or finish
- Many scheduling policies imply in preempting the CPU

Let’s make our system preemptive!

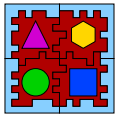


```
template<> struct Traits<Thread>: public Traits<void>
{
    static const bool preemptive = true;
    static const bool trace_idle = hysterically_debugged;
    static const unsigned int QUANTUM = 10000; // us
};

void Thread::constructor_epilogue(const Log_Addr & entry, unsigned int stack_size)
{
    assert((_state != WAITING) && (_state != FINISHING)); // Invalid states

    switch(_state) {
        case RUNNING: assert(entry == __epos_app_entry); break;
        case READY: _ready.insert(&_link); break;
        case SUSPENDED: _suspended.insert(&_link); break;
        case WAITING: break;   // Invalid state, for switch completion only
        case FINISHING: break; // Invalid state, for switch completion only
    }

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



preemption is now a feature controlled by programmers

```
template<> struct Traits<Thread>: public Traits<void>
{
    static const bool preemptive = true;
    static const bool trace_idle = hysterically_debugged;
    static const unsigned int QUANTUM = 10000; // us
};

void Thread::constructor_epilogue(const Log_Addr & entry, unsigned int stack_size)
{
    assert((_state != WAITING) && (_state != FINISHING)); // Invalid states

    switch(_state) {
        case RUNNING: assert(entry == __epos_app_entry); break;
        case READY: _ready.insert(&_link); break;
        case SUSPENDED: _suspended.insert(&_link); break;
        case WAITING: break;   // Invalid state, for switch completion only
        case FINISHING: break; // Invalid state, for switch completion only
    }

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

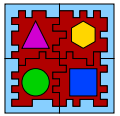


```
template<> struct Traits<Thread>: public Traits<void>
{
    static const bool preemptive = true;
    static const bool trace_idle = hysterically_debugged;
    static const unsigned int QUANTUM = 10000; // us
};

void Thread::constructor_epilogue(const Log_Addr & entry, unsigned int stack_size)
{
    assert((_state != WAITING) && (_state != FINISHING)); // Invalid states

    switch(_state) {
        case RUNNING: assert(entry == __epos_app_entry); break;
        case READY: _ready.insert(&_link); break;
        case SUSPENDED: _suspended.insert(&_link); break;
        case WAITING: break;   // Invalid state, for switch completion only
        case FINISHING: break; // Invalid state, for switch completion only
    }

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

reschedule at Thread creation, observing
state == READY and priority != IDLE

```
void Thread::resume(bool unpreemptive)
{
    lock();

    if(_state == SUSPENDED) {
        _suspended.remove(this);
        _state = READY;
        _ready.insert(&_link);

        if(preemptive && !unpreemptive)
            reschedule();
        else
            unlock();
    } else {
        db<Thread>(WRN) << "Resume called for
unsuspended object!" << endl;

        unlock();
    }
}

void Thread::exit(int status)
{
            :
    if(prev->_joining) {
        Thread * joining = prev->_joining;
        prev->_joining = 0;
        joining->resume(true);
        lock();
    }
            :
}
```

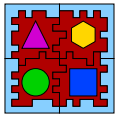
```
void Thread::wakeup(Queue * q)
{
  // lock() must be called before entering this method
  assert(locked());

  if(!q->empty()) {
    Thread * t = q->remove()->object();
    t->_state = READY;
    t->_waiting = 0;
    _ready.insert(&t->_link);

    if(preemptive)
      reschedule();
    else
      unlock();
  } else
    unlock();
}
```



```
void Thread::resume(bool unpreemptive)
{
    lock();

    if(_state == SUSPENDED) {
        _suspended.remove(this);
        _state = READY;
        _ready.insert(&_link);

        if(preemptive && !unpreemptive)
            reschedule();
        else
            unlock();
    } else {
        db<Thread>(WRN) << "Resume called for
unsuspended object!" << endl;

        unlock();
    }
}

void Thread::exit(int status)
{
            :
    if(prev->_joining) {
        Thread * joining = prev->_joining;
        prev->_joining = 0;
        joining->resume(true);
        lock();
    }
            :
}
```

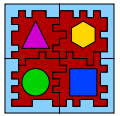
```
void Thread::wakeup(Queue * q)
{
  // lock() must be called before entering this method
  assert(locked());

  if(!q->empty()) {
    Thread * t = q->remove()->object();
    t->_state = READY;
    t->_waiting = 0;
    _ready.insert(&t->_link);

    if(preemptive)
      reschedule();
    else
      unlock();
  } else
    unlock();
}
```

reschedule at resume()



```
void Thread::resume(bool unpreemptive)
{
    lock();

    if(_state == SUSPENDED) {
        _suspended.remove(this);
        _state = READY;
        _ready.insert(&_link);

        if(preemptive && !unpreemptive)
            reschedule();
        else
            unlock();
    } else {
        db<Thread>(WRN) << "Resume called for
unsuspended object!" << endl;

        unlock();
    }
}

void Thread::exit(int status)
{
            :
    if(prev->_joining) {
        Thread * joining = prev->_joining;
        prev->_joining = 0;
        joining->resume(true);
        lock();
    }
            :
}
```

```
void Thread::wakeup(Queue * q)
{
  // lock() must be called before entering this method
  assert(locked());

  if(!q->empty()) {
    Thread * t = q->remove()->object();
    t->_state = READY;
    t->_waiting = 0;
    _ready.insert(&t->_link);

    if(preemptive)
      reschedule();
    else
      unlock();
  } else
    unlock();
}
```

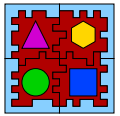


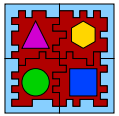
prevent preemption here, since exit will always cause a reschedule

```
void Thread::resume(bool unpreemptive)
{
    lock();

    if(_state == SUSPENDED) {
        suspended.remove(this);

        if(preemptive && !unpreemptive)
            reschedule();
        else
            unlock();
    } else {
        db<Thread>(WRN) << "Resume called for
unsuspended object!" << endl;

        unlock();
    }
}

void Thread::exit(int status)
{
            :
    if(prev->_joining) {
        Thread * joining = prev->_joining;
        prev->_joining = 0;
        joining->resume(true);
        lock();
    }
            :
}
```

reschedule at wakeup()

```
void Thread::wakeup(Queue * q)
{
   // lock() must be called before entering this method
   assert(locked());

   if(!q->empty()) {
      Thread * t = q->remove()->object();
      t->_state = READY;
      t->_waiting = 0;
      _ready.insert(&t->_link);

      if(preemptive)
         reschedule();
      else
         unlock();
   } else
      unlock();
}
```



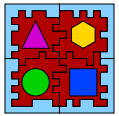


```
void Thread::wakeup_all(Queue * q)
{
    db<Thread>(TRC) << "Thread::wakeup_all(running=" << running() << ",q=" << q << ")" << endl;

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

    if(!q->empty()) {
        while(!q->empty()) {
            Thread * t = q->remove()->object();
            t->_state = READY;
            t->_waiting = 0;
            _ready.insert(&t->_link);
        }

        if(preemptive)
            reschedule();
        else
            unlock();
    } else
        unlock();
}
```

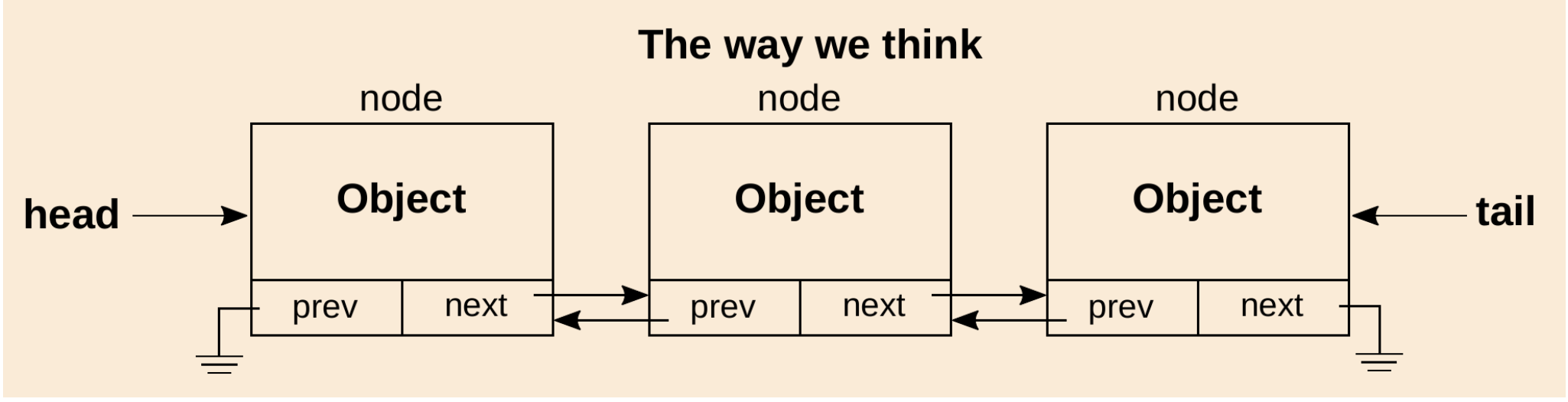
```
void Thread::wakeup_all(Queue * q)
{
    db<Thread>(TRC) << "Thread::wakeup_all(running=" << running() << ",q=" << q << ")" << endl;

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

    if(!q->empty()) {
        while(!q->empty()) {
            Thread * t = q->remove()->object();
            t->_state = READY;
            t->_waiting = 0;
            _ready.insert(&t->_link);
        }

        if(preemptive)
            reschedule();
        else
            unlock();
    } else
        unlock();
}
```

wake up everybody before calling reschedule()

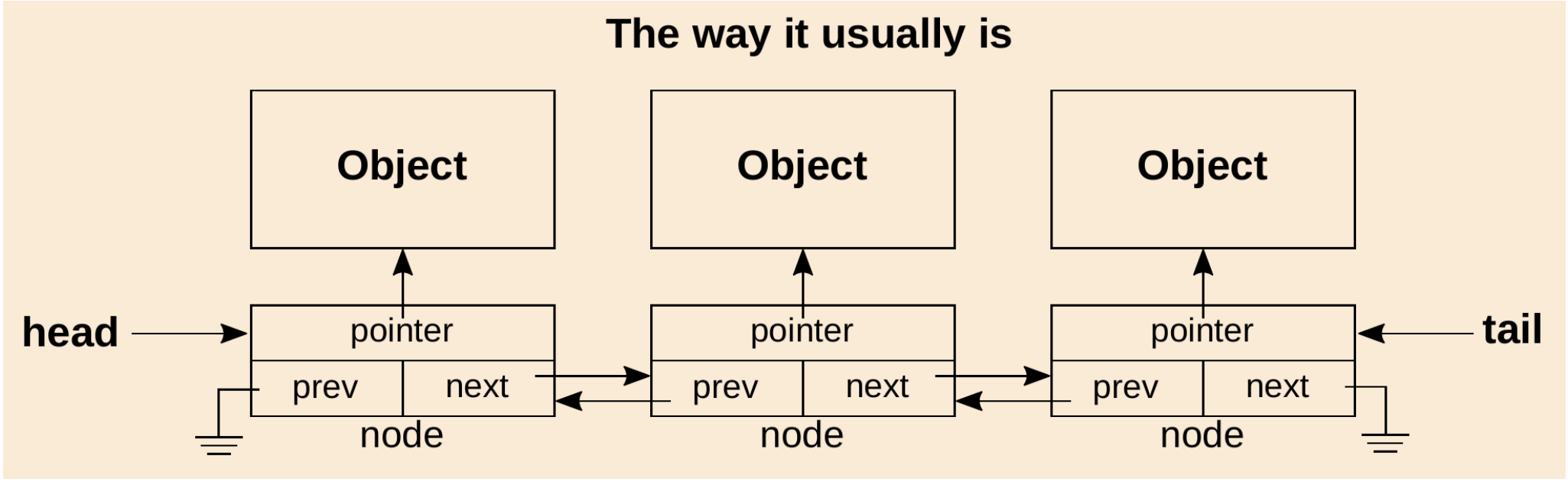
About Lists at System-level



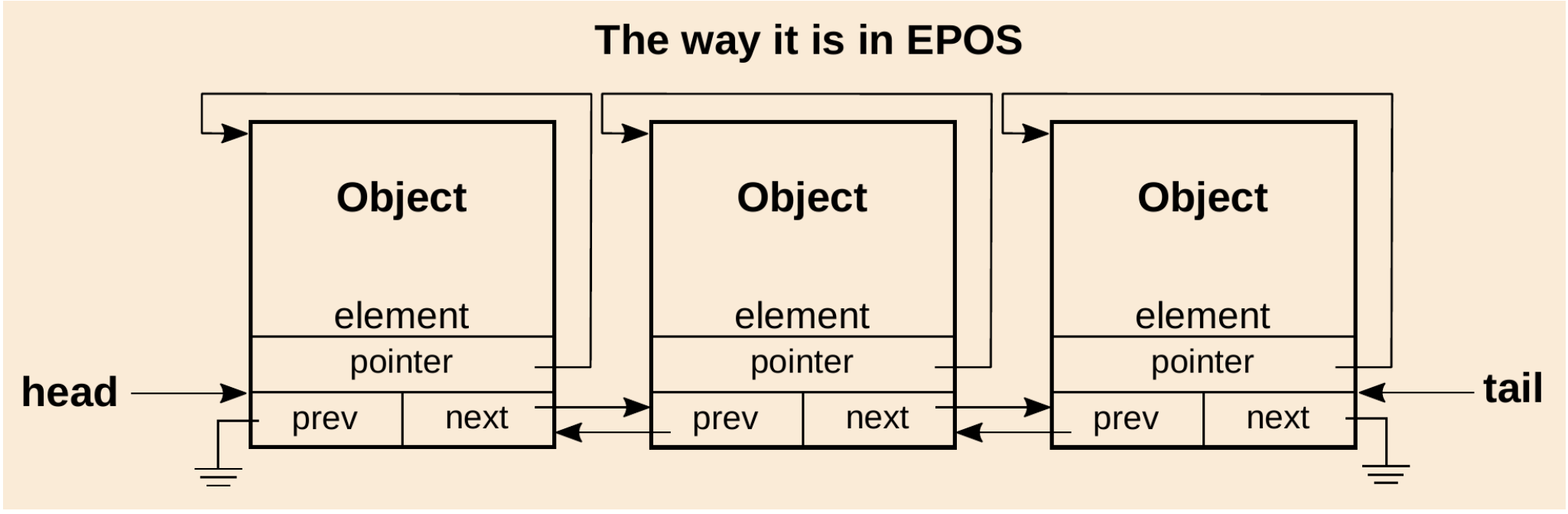
About Lists at System-level

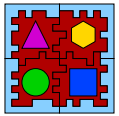


About Lists at System-level



About Lists at System-level





Embedded Parallel Operating System

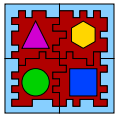
Coding Journey through OS Design
–from co-routines to a multicore kernel–

Prof. Antônio Augusto Fröhlich, Ph.D.

UFSC / LISHA

September 30, 2020

Doing nothing well!



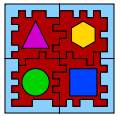
Our idle() function can be improved!

It is being explicitly called all around

Making it a thread, would also eliminate many test for READY being empty

We need a idle Thread that **does nothing well**!

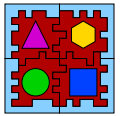
```
class Thread
{
public:
    // Thread Priority
    typedef unsigned int Priority;
    enum {
        MAIN   = 0,
        HIGH   = 1,
        NORMAL =(unsigned(1)<<(sizeof(int)*8-1))-4,
        LOW    =(unsigned(1)<<(sizeof(int)*8-1))-3,
        IDLE   =(unsigned(1)<<(sizeof(int)*8-1))-2
    };

protected:
    static volatile unsigned int _thread_count;
};

void Thread::constructor_prologue(unsigned int
stack_size)
{
    lock();
    _thread_count++;
    _stack = reinterpret_cast<char *>(
             kmalloc(stack_size));
}
```



```
Thread::~Thread()
{
    lock();
    // The running thread cannot delete itself!
    assert(_state != RUNNING);
    switch(_state) {
    case RUNNING:  // For switch completion only:
the running thread would have deleted itself! Stack
wouldn't have been released!
        exit(-1);
        break;
    case READY:
        _ready.remove(this);
        _thread_count--;
        break;
    case SUSPENDED:
        _suspended.remove(this);
        _thread_count--;
        break;
    case WAITING:
        _waiting->remove(this);
        _thread_count--;
        break;
    case FINISHING: // Already called exit()
        break;
    }
    if(_joining)
        _joining->resume();
    unlock();
    kfree(_stack);
}
```

```
class Thread
{
public:
    // Thread Priority
    typedef unsigned 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
    };

protected:
    static volatile unsigned int _thread_count;
};

void Thread::constructor_prologue(unsigned int
stack_size)
{
    lock();
    _thread_count++;
    _stack = reinterpret_cast<char *>(
             kmalloc(stack_size));
}
```

rearranging priorities and including IDLE



```
Th
{
    // The running thread cannot delete itself!
    assert(_state != RUNNING);
    switch(_state) {
    case RUNNING:  // For switch completion only:
the running thread would have deleted itself! Stack
wouldn't have been released!
        exit(-1);
        break;
    case READY:
        _ready.remove(this);
        _thread_count--;
        break;
    case SUSPENDED:
        _suspended.remove(this);
        _thread_count--;
        break;
    case WAITING:
        _waiting->remove(this);
        _thread_count--;
        break;
    case FINISHING: // Already called exit()
        break;
    }
    if(_joining)
        _joining->resume();
    unlock();
    kfree(_stack);
}
```

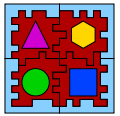
```
class Thread
{
public:
    // Thread Priority
    typedef unsigned int Priority;
    enum {
        MAIN   = 0,
        HIGH   = 1,
        NORMAL =(unsigned(1)<<(sizeof(int)*8-1))-4,
        LOW    =(unsigned(1)<<(sizeof(int)*8-1))-3,
        IDLE   =(unsigned(1)<<(sizeof(int)*8-1))-2
    };

protected:
    static volatile unsigned int _thread_count;
};

void Thread::constructor_prologue(unsigned int
stack_size)
{
    lock();
    _thread_count++;
    _stack = reinterpret_cast<char *>(
             kmalloc(stack_size));
}
```

Thread counting to decide when it's time to reboot or halt the system

```
Thread::~Thread()
{
    lock();
    // The running thread cannot delete itself!
    assert( state != RUNNING);

                                          n only:
th                                        lf! Stack
wouldn't have been released!
        exit(-1);
        break;
    case READY:
        _ready.remove(this);
        _thread_count--;
        break;
    case SUSPENDED:
        _suspended.remove(this);
        _thread_count--;
        break;
    case WAITING:
        _waiting->remove(this);
        _thread_count--;
        break;
    case FINISHING: // Already called exit()
        break;
    }
    if(_joining)
        _joining->resume();
    unlock();
    kfree(_stack);
}
```

a running Thread cannot delete itself!
how could it finish the delete operation?

```
    typedef unsigned 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
    };

protected:
    static volatile unsigned int _thread_count;
};

void Thread::constructor_prologue(unsigned int
stack_size)
{
    lock();
    _thread_count++;
    _stack = reinterpret_cast<char *>(
             kmalloc(stack_size));
}
```

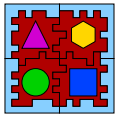


```
Thread::~Thread()
{
    lock();
    // The running thread cannot delete itself!
    assert(_state != RUNNING);
    switch(_state) {
    case RUNNING:  // For switch completion only:
the running thread would have deleted itself! Stack
wouldn't have been released!
        exit(-1);
        break;
    case READY:
        _ready.remove(this);
        _thread_count--;
        break;
    case SUSPENDED:
        _suspended.remove(this);
        _thread_count--;
        break;
    case WAITING:
        _waiting->remove(this);
        _thread_count--;
        break;
    case FINISHING: // Already called exit()
        break;
    }
    if(_joining)
        _joining->resume();
    unlock();
    kfree(_stack);
}
```

```
class Thread
{
public:
    // Thread Priority
    typedef unsigned 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
    };
prote
    static volatile unsigned int _thread_count;
};

void Thread::constructor_prologue(unsigned int
stack_size)
{
    lock();
    _thread_count++;
    _stack = reinterpret_cast<char *>(
             kmalloc(stack_size));
}
```

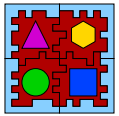
the “expected” case

```
Thread::~Thread()
{
    lock();
    // The running thread cannot delete itself!
    assert(_state != RUNNING);
    switch(_state) {
    case RUNNING:  // For switch completion only:
the running thread would have deleted itself! Stack
wouldn't have been released!
        exit(-1);
        break;
    case READY:
        _ready.remove(this);
        _thread_count--;
        break;
    case SUSPENDED:
        _suspended.remove(this);
        _thread_count--;
        break;
    case WAITING:
        _waiting->remove(this);
        _thread_count--;
        break;
    case FINISHING: // Already called exit()
        break;
    }
    if(_joining)
        _joining->resume();
    unlock();
    kfree(_stack);
}
```



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

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

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

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

        dispatch(this, _running);
    }

    unlock();
}

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

    Thread * prev = _running;
    prev->_state = READY;
    _ready.insert(&prev->_link);
    _running = _ready.remove()->object();
    _running->_state = RUNNING;

    dispatch(prev, _running);

    unlock();
}
```

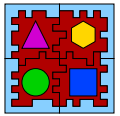
```
void Thread::exit(int status)
{
    lock();

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

    if(prev->_joining) {
        Thread * joining = prev->_joining;
        prev->_joining = 0;
        joining->resume(); // implicit unlock()
        lock();
    }

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

    dispatch(prev, _running);

    unlock();
}
```



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

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

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

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

        dispatch(this, _running);
    }

    unlock();
}

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

    Thread * prev = _running;
    prev->_state = READY;
    _ready.insert(&prev->_link);
    _running = _ready.remove()->object();
    _running->_state = RUNNING;

    dispatch(prev, _running);

    unlock();
}
```

no more _ready.empty() tests!

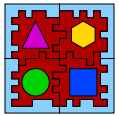
```
    Thread * prev = _running;
    prev->_state = FINISHING;
    *reinterpret_cast<int *>(prev→_stack)
        = status;
    _thread_count--;

    if(prev->_joining) {
        Thread * joining = prev->_joining;
        prev->_joining = 0;
        joining->resume(); // implicit unlock()
        lock();
    }

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

    dispatch(prev, _running);

    unlock();
}
```



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

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

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

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

        dispatch(this, _running);
    }

    unlock();
}

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

    Thread * prev = _running;
    prev->_state = READY;
    _ready.insert(&prev->_link);
    _running = _ready.remove()->object();
    _running->_state = RUNNING;

    dispatch(prev, _running);

    unlock();
}
```

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

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

    if(prev->_joining) {
        Thread * joining = prev->_joining;
        prev->_joining = 0;
        joining->resume(); // implicit unlock()
        lock();
    }

                                          t();

    unlock();
}
```

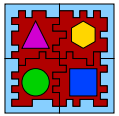
no more _ready.empty() tests!



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

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

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

    if
                                        ();
        dispatch(this, _running);
    }

    unlock();
}

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

    Thread * prev = _running;
    prev->_state = READY;
    _ready.insert(&prev->_link);
    _running = _ready.remove()->object();
    _running->_state = RUNNING;

    dispatch(prev, _running);

    unlock();
}
```

no more _ready.empty() tests!
last Thread now handled by idle

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

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

    if(prev->_joining) {
        Thread * joining = prev->_joining;
        prev->_joining = 0;
        joining->resume(); // implicit unlock()
        lock();
    }

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

    dispatch(prev, _running);

    unlock();
}
```



```
int Thread::idle()
{
    while(_thread_count > 1) {
        if(Traits<Thread>::trace_idle)
            db<Thread>(TRC) << "Thread::idle(this="
                << running() << ")" << endl;
        CPU::int_enable();
        CPU::halt();
    }

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

    return 0;
}


extern "C" { void __epos_app_entry(); }
```

```
void Thread::init()
{
    db<Init,Thread>(TRC) << "Thread::init()" <<endl;

    Thread::_running =
        new (kmalloc(sizeof(Thread))) Thread(
            Thread::Configuration(Thread::RUNNING,
                Thread::MAIN), __epos_app_entry));

    new (kmalloc(sizeof(Thread))) Thread(
        Thread::Configuration(Thread::READY,
            Thread::IDLE), &Thread::idle);

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

    This_Thread::not_booting();
}

class Init_First
{
public:
    Init_First() {
        db<Init>(TRC) << "Init_First()" << endl;
        if(!Traits<System>::multithread) {
            CPU::int_enable();
            return;
        }

        Thread::running()->_context->load();
    }
};
```

```
int Thread::idle()
{
    while(_thread_count > 1) {
        if(Traits<Thread>::trace_idle)
            db<Thread>(TRC) << "Thread::idle(this="
                << running() << ")" << endl;
        CPU::int_enable();
        CPU::halt();
    }

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

    return 0;
}


extern "C" { void __epos_app_entry(); }
```

idle loop until there is only one thread in the system (i.e. idle)

```
void Thread::init()
{
    db<Init>(INF) << "Thread::init()" << endl;

    Thread::_running =
        new (kmalloc(sizeof(Thread))) Thread(
            Thread::Configuration(Thread::RUNNING,
                Thread::MAIN), __epos_app_entry));

    new (kmalloc(sizeof(Thread))) Thread(
        Thread::Configuration(Thread::READY,
            Thread::IDLE), &Thread::idle);

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

    This_Thread::not_booting();
}

class Init_First
{
public:
    Init_First() {
        db<Init>(TRC) << "Init_First()" << endl;
        if(!Traits<System>::multithread) {
            CPU::int_enable();
            return;
        }

        Thread::running()->_context->load();
    }
};
```

```
int Thread::idle()
{
    while(_thread_count > 1) {
        if(Traits<Thread>::trace_idle)
            db<Thread>(TRC) << "Thread::idle(this="
                << running() << ")" << endl;
        CPU::int_enable();
        CPU::halt();
    }

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

    return 0;
}


extern "C" { void __epos_app_entry(); }
```

last Thread exit handling

```
void Thread::init()
{
    db<Init,Thread>(TRC) << "Thread::init()" <<endl;

    Thread::_running =
        new (kmalloc(sizeof(Thread))) Thread(
            Thread::Configuration(Thread::RUNNING,
                                  ));

    new (k
        Thread::Configuration(Thread::READY,
            Thread::IDLE), &Thread::idle);

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

    This_Thread::not_booting();
}

class Init_First
{
public:
    Init_First() {
        db<Init>(TRC) << "Init_First()" << endl;
        if(!Traits<System>::multithread) {
            CPU::int_enable();
            return;
        }

        Thread::running()->_context->load();
    }
};
```

```
int Thread::idle()
{
    while(_thread_count > 1) {
        if(Traits<Thread>::trace_idle)
            db<Thread>(TRC) << "Thread::idle(this="
                << running() << ")" << endl;
        CPU::int_enable();
        CPU::halt();
    }

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

    return 0;
}


extern "C" { void __epos_app_entry(); }
```

```
void Thread::init()
{
    db<Init,Thread>(TRC) << "Thread::init()" <<endl;

    Thread::_running =
        new (kmalloc(sizeof(Thread))) Thread(
            Thread::Configuration(Thread::RUNNING,
                Thread::MAIN), __epos_app_entry));

    new (kmalloc(sizeof(Thread))) Thread(
        Thread::Configuration(Thread::READY,
            Thread::IDLE), &Thread::idle);

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

    This_Thread::not_booting();
}

class Init_First
{
public:
    I

        Thread::running()->_context->load();
    }
};
```

If EPOS is a library, then adjust the application entry point to __epos_app_entry, which will directly call main(). In this case, _init will have already been called before Init_Application to construct MAIN's global objects.

```
int Thread::idle()
{
    while(_thread_count > 1) {
        if(Traits<Thread>::trace_idle)
            d::idle(this="
                < endl;
        ();
    }

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

    return 0;
}


extern "C" { void __epos_app_entry(); }
```

create the main Thread with state=RUNNING and priority = MAIN

```
void Thread::init()
{
    db<Init,Thread>(TRC) << "Thread::init()" <<endl;

    Thread::_running =
        new (kmalloc(sizeof(Thread))) Thread(
            Thread::Configuration(Thread::RUNNING,
                Thread::MAIN), __epos_app_entry));

    new (kmalloc(sizeof(Thread))) Thread(
        Thread::Configuration(Thread::READY,
            Thread::IDLE), &Thread::idle);

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

    This_Thread::not_booting();
}

class Init_First
{
public:
    Init_First() {
        db<Init>(TRC) << "Init_First()" << endl;
        if(!Traits<System>::multithread) {
            CPU::int_enable();
            return;
        }

        Thread::running()->_context->load();
    }
};
```

```
int Thread::idle()
{
    while(_thread_count > 1) {
        if(Traits<Thread>::trace_idle)
            db<Thread>(TRC) << "Thread::idle(this="
                << running() << ")" << endl;
        CPU::int_enable();
        CPU::halt();

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

    return 0;
}


extern "C" { void __epos_app_entry(); }
```

create the idle Thread with
state = READY and priority = IDLE

```
void Thread::init()
{
    db<Init,Thread>(TRC) << "Thread::init()" <<endl;

    Thread::_running =
        new (kmalloc(sizeof(Thread))) Thread(
            Thread::Configuration(Thread::RUNNING,
                Thread::MAIN), __epos_app_entry));

    new (kmalloc(sizeof(Thread))) Thread(
        Thread::Configuration(Thread::READY,
            Thread::IDLE), &Thread::idle);

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

    This_Thread::not_booting();
}

class Init_First
{
public:
    Init_First() {
        db<Init>(TRC) << "Init_First()" << endl;
        if(!Traits<System>::multithread) {
            CPU::int_enable();
            return;
        }

        Thread::running()->_context->load();
    }
};
```

```
int Thread::idle()
{
    while(_thread_count > 1) {
        if(Traits<Thread>::trace_idle)
            db<Thread>(TRC) << "Thread::idle(this="
                << running() << ")" << endl;
        CPU::int_enable();
        CPU::halt();
    }

    CPU::int_disable();
```

install a time slicer for time-sharing algorithms

```
            << "Rebooting the machine ..." << endl;
        Machine::reboot();
    } else {
        db<Thread>(WRN)
            << "Halting the machine ..." << endl;
        CPU::halt();
    }

    return 0;
}


extern "C" { void __epos_app_entry(); }
```

```
void Thread::init()
{
    db<Init,Thread>(TRC) << "Thread::init()" <<endl;

    Thread::_running =
        new (kmalloc(sizeof(Thread))) Thread(
            Thread::Configuration(Thread::RUNNING,
                Thread::MAIN), __epos_app_entry));

    new (kmalloc(sizeof(Thread))) Thread(
        Thread::Configuration(Thread::READY,
            Thread::IDLE), &Thread::idle);

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

    This_Thread::not_booting();
}

class Init_First
{
public:
    Init_First() {
        db<Init>(TRC) << "Init_First()" << endl;
        if(!Traits<System>::multithread) {
            CPU::int_enable();
            return;
        }

        Thread::running()->_context->load();
    }
};
```

```cpp
int Thread::idle()
{
    while(_thread_count > 1) {
        if(Traits<Thread>::trace_idle)
            db<Thread>(TRC) << "Thread::idle(this="
                << running() << ")" << endl;
        CPU::int_enable();
        CPU::halt();
    }

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

    return 0;
}


extern "C" { void __epos_app_entry(); }
```

transition from CPU-based locking to thread-based locking

```cpp
void Thread::init()
{
    db<Init,Thread>(TRC) << "Thread::init()" <<endl;

    Thread::_running =
        new (kmalloc(sizeof(Thread))) Thread(
            Thread::Configuration(Thread::RUNNING,
                Thread::MAIN), __epos_app_entry));

    new (kmalloc(sizeof(Thread))) Thread(
        Thread::Configuration(Thread::READY,
            Thread::IDLE), &Thread::idle);

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

    This_Thread::not_booting();
}

class Init_First
{
public:
    Init_First() {
        db<Init>(TRC) << "Init_First()" << endl;
        if(!Traits<System>::multithread) {
            CPU::int_enable();
            return;
        }

        Thread::running()->_context->load();
    }
};
```

```
int Thread::idle()
{
    while(_thread_count > 1) {
        if(Traits<Thread>::trace_idle)
            db<Thread>(TRC) << "Thread::idle(this="
                << running() << ")" << endl;
        CPU::int_enable();
        CPU::halt();
    }

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

simple, single-core scenarios might not even need multithreading!

```
extern "C" { void __epos_app_entry(); }
```

```
void Thread::init()
{
    db<Init,Thread>(TRC) << "Thread::init()" <<endl;

    Thread::_running =
        new (kmalloc(sizeof(Thread))) Thread(
            Thread::Configuration(Thread::RUNNING,
                Thread::MAIN), __epos_app_entry));

    new (kmalloc(sizeof(Thread))) Thread(
        Thread::Configuration(Thread::READY,
            Thread::IDLE), &Thread::idle);

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

    This_Thread::not_booting();
}

class Init_First
{
public:
    Init_First() {
        db<Init>(TRC) << "Init_First()" << endl;
        if(!Traits<System>::multithread) {
            CPU::int_enable();
            return;
        }

        Thread::running()->_context->load();
    }
};
```

```
int Thread::idle()
{
    while(_thread_count > 1) {
        if(Traits<Thread>::trace_idle)
            db<Thread>(TRC) << "Thread::idle(this="
                << running() << ")" << endl;
        CPU::int_enable();
        CPU::halt();
    }

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

loading the first (main) Thread's context, causing it to start executing

```
}


extern "C" { void __epos_app_entry(); }
```

```
void Thread::init()
{
    db<Init,Thread>(TRC) << "Thread::init()" <<endl;

    Thread::_running =
        new (kmalloc(sizeof(Thread))) Thread(
            Thread::Configuration(Thread::RUNNING,
                Thread::MAIN), __epos_app_entry));

    new (kmalloc(sizeof(Thread))) Thread(
        Thread::Configuration(Thread::READY,
            Thread::IDLE), &Thread::idle);

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

    This_Thread::not_booting();
}

class Init_First
{
public:
    Init_First() {
        db<Init>(TRC) << "Init_First()" << endl;
        if(!Traits<System>::multithread) {
            CPU::int_enable();
            return;
        }

        Thread::running()->_context->load();
    }
};
```

```
int Thread::idle()
{
    while(_thread_count > 1) {
        if(Traits<Thread>::trace_idle)
            db<Thread>(TRC) << "Thread::idle(this="
                << running() << ")" << endl;
        CPU::int_enable();
        CPU::halt();
    }

    CPU::int_disable();
    db<Thread>(WRN)
        << "The last thread has exited!" << endl;
    if(reboot) {
                                          he ..." << endl;


                                          ..." << endl;

}

ex                                          ; }
```

Context::load()

From the perspective of the CPU, a process is already running on the CPU since it was switched on. Short after, the stack pointer was set and the process, which already had an implicit address space, code and data, got a stack. The OS, however, is not aware of this. So we cannot “switch” context. We must load the first context.

```
void Thread::init()
{
    db<Init,Thread>(TRC) << "Thread::init()" <<endl;

    Thread::_running =
        new (kmalloc(sizeof(Thread))) Thread(
            Thread::Configuration(Thread::RUNNING,
                Thread::MAIN), __epos_app_entry));

    new (kmalloc(sizeof(Thread))) Thread(
        Thread::Configuration(Thread::READY,
            Thread::IDLE), &Thread::idle);

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

    This_Thread::not_booting();
}

class Init_First
{
public:
    Init_First() {
        db<Init>(TRC) << "Init_First()" << endl;
        if(!Traits<System>::multithread) {
            CPU::int_enable();
            return;
        }

        Thread::running()->_context->load();
    }
};
```

```
int Thread::idle()
{
    while(_thread_count > 1) {
        if(Traits<Thread>::trace_idle)
            db<Thread>(TRC) << "Thread::idle(this="
                << running() << ")" << endl;
        CPU::int_enable();
        CPU::halt();
    }

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

    return 0;
}


extern "C" { void __epos_app_entry(); }
```

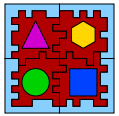
```
void Thread::init()
{
    db<Init,Thread>(TRC) << "Thread::init()" <<endl;

    Thread::_running =
        new (kmalloc(sizeof(Thread))) Thread(
            Thread::Configuration(Thread::RUNNING,
                Thread::MAIN), __epos_app_entry));

    new (kmalloc(sizeof(Thread))) Thread(
        Thread::Configuration(Thread::READY,
            Thread::IDLE), &Thread::idle);

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

    This_Thread::not_booting();
}

class Init_First
{
public:
    Init_First() {
        db<Init>(TRC) << "Init_First()" << endl;
        if(!Traits<System>::multithread) {
            CPU::int_enable();
            return;
        }

        Thread::running()->_context->load();
    }
};
```

```
int Thread::idle()
{
    while(_thread_count > 1) {
        if(Traits<Thread>::trace_idle)
            db<Thread>(TRC) << "Thread::idle(this="
                << running() << ")" << endl;
        CPU::int_enable();
        CPU::halt();
```

Who holds a pointer to idle?

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

    return 0;
}


extern "C" { void __epos_app_entry(); }
```

```
void Thread::init()
{
    db<Init,Thread>(TRC) << "Thread::init()" <<endl;

    Thread::_running =
        new (kmalloc(sizeof(Thread))) Thread(
            Thread::Configuration(Thread::RUNNING,
                Thread::MAIN), __epos_app_entry));

    new (kmalloc(sizeof(Thread))) Thread(
        Thread::Configuration(Thread::READY,
            Thread::IDLE), &Thread::idle);

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

    This_Thread::not_booting();
}

class Init_First
{
public:
    Init_First() {
        db<Init>(TRC) << "Init_First()" << endl;
        if(!Traits<System>::multithread) {
            CPU::int_enable();
            return;
        }

        Thread::running()->_context->load();
    }
};
```



Embedded Parallel Operating System

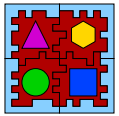
Coding Journey through OS Design
–from co-routines to a multicore kernel–

Prof. Antônio Augusto Fröhlich, Ph.D.

UFSC / LISHA

September 30, 2020

Blocking Synchronization



We will now start to change the OS so it does **blocking synchronization** instead of co-routines.

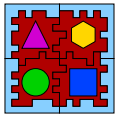
Let’s replace yield() calls for waiting queues operations.

Let’s first focus on the Synchronizers and then on Thread::join().

Let’s leave Alarm::delay() for later on!

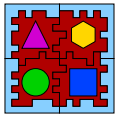
```
class Synchronizer_Common
{
protected:
    typedef Thread::Queue Queue;

protected:
    Synchronizer_Common() {}
    ~Synchronizer_Common() {
        begin_atomic();
        wakeup_all();
    }

    // Atomic operations
    bool tsl(volatile bool & lock) {
        return CPU::tsl(lock); }
    int finc(volatile int & number) {
        return CPU::finc(number); }
    int fdec(volatile int & number) {
        return CPU::fdec(number); }

    // Thread operations
    void begin_atomic() { Thread::lock(); }
    void end_atomic() { Thread::unlock(); }

    void sleep() { Thread::sleep(&_queue); }
    void wakeup() { Thread::wakeup(&_queue); }
    void wakeup_all() {
        Thread::wakeup_all(&_queue); }

protected:
    Queue _queue;
};
```

```
class Thread
{
        :
protected:
    static void sleep(Queue * q);
    static void wakeup(Queue * q);
    static void wakeup_all(Queue * q);

        :

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

        :
};
```



```
class Synchronizer_Common
{
protected:
    typedef Thread::Queue Queue;

protected:
    Synchronizer_Common() {}
    ~Synchronizer_Common() {
        begin_atomic();
        wakeup_all();
    }

    // Atomic operations
    bool tsl(volatile bool & lock) {
        return CPU::tsl(lock); }
    int finc(volatile int & number) {
        return CPU::finc(number); }
    int fdec(volatile int & number) {
        return CPU::fdec(number); }

    // Thread operations
    void begin_atomic() { Thread::lock(); }
    void end_atomic() { Thread::unlock(); }

    void sleep() { Thread::sleep(&_queue); }
    void wakeup() { Thread::wakeup(&_queue); }
    void wakeup_all() {
        Thread::wakeup_all(&_queue); }

protected:
    Queue _queue;
};
```

import Queue from Thread, honoring ordering

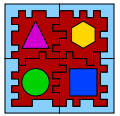


```
protected:
    static void sleep(Queue * q);
    static void wakeup(Queue * q);
    static void wakeup_all(Queue * q);

        :

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

        :
};
```

```
class Synchronizer_Common
{
protected:
    typedef Thread::Queue Queue;

protected:
    Synchronizer_Common() {}
    ~Synchronizer_Common() {
        begin_atomic();
        wakeup_all();
    }

    // Atomic operations
    bool tsl(volatile bool & lock) {
        return CPU::tsl(lock); }
    int finc(volatile int & number) {
        return CPU::finc(number); }
    int fdec(volatile int & number) {
        return CPU::fdec(number); }

    // Thread operations
    void begin_atomic() { Thread::lock(); }
    void end_atomic() { Thread::unlock(); }

    void sleep() { Thread::sleep(&_queue); }
    void wakeup() { Thread::wakeup(&_queue); }
    void wakeup_all() {
        Thread::wakeup_all(&_queue); }

protected:
    Queue _queue;
};
```

release blocked threads if deleted

```
class Thread
{
        :
protected:
    static void sleep(Queue * q);
    static void wakeup(Queue * q);
                                eue * q);
        :

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

        :
};
```



```
class Synchronizer_Common
{
protected:
    typedef Thread::Queue Queue;

protected:
    Synchronizer_Common() {}
    ~Synchronizer_Common() {
        begin_atomic();
        wakeup_all();
    }

    // Atomic operations
    bool tsl(volatile bool & lock) {
        return CPU::tsl(lock); }
    int finc(volatile int & number) {
        return CPU::finc(number); }
    int fdec(volatile int & number) {
        return CPU::fdec(number); }

    // Thread operations
    void begin_atomic() { Thread::lock(); }
    void end_atomic() { Thread::unlock(); }

    void sleep() { Thread::sleep(&_queue); }
    void wakeup() { Thread::wakeup(&_queue); }
    void wakeup_all() {
        Thread::wakeup_all(&_queue); }

protected:
    Queue _queue;
};
```

```
class Thread
{
        :
protected:
    static void sleep(Queue * q);
    static void wakeup(Queue * q);
    static void wakeup_all(Queue * q);

        :

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

        :
};
```

forward blocking operations to Thread

```
class Synchronizer_Common
{
protected:
    typedef Thread::Queue Queue;

protected:
    Synchronizer_Common() {}
    ~Synchronizer_Common() {
        begin_atomic();
        wakeup_all();
    }

    // Atomic operations
    bool tsl(volatile bool & lock) {
        return CPU::tsl(lock); }
    int finc(volatile int & number) {
        return CPU::finc(number); }
    int fdec(volatile int & number) {
        return CPU::fdec(number); }

    // Thread operations
    void begin_atomic() { Thread::lock(); }
    void end_atomic() { Thread::unlock(); }

    void sleep() { Thread::sleep(&_queue); }
    void wakeup() { Thread::wakeup(&_queue); }
    void wakeup_all() {
        Thread::wakeup_all(&_queue); }

protected:
    Queue _queue;
};
```

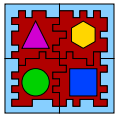
```
class Thread
{
        :
protected:
    static void sleep(Queue * q);
    static void wakeup(Queue * q);
    static void wakeup_all(Queue * q);

        :

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

        :
};
```

a queue of blocked Threads

```
class Synchronizer_Common
{
protected:
    typedef Thread::Queue Queue;

protected
    Synch
    ~Synchronizer_Common() {
        begin_atomic();
        wakeup_all();
    }

    // Atomic operations
    bool tsl(volatile bool & lock) {
        return CPU::tsl(lock); }
    int finc(volatile int & number) {
        return CPU::finc(number); }
    int fdec(volatile int & number) {
        return CPU::fdec(number); }

    // Thread operations
    void begin_atomic() { Thread::lock(); }
    void end_atomic() { Thread::unlock(); }

    void sleep() { Thread::sleep(&_queue); }
    void wakeup() { Thread::wakeup(&_queue); }
    void wakeup_all() {
        Thread::wakeup_all(&_queue); }

protected:
    Queue _queue;
};
```

takes a reference to a waiting queue

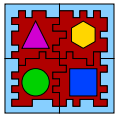
```
class Thread
{
        :
protected:
    static void sleep(Queue * q);
    static void wakeup(Queue * q);
    static void wakeup_all(Queue * q);

        :

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

        :
};
```

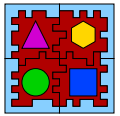


```
class Synchronizer_Common
{
protected:
    typedef Thread::Queue Queue;

protected:
    Synchronizer_Common() {}
    ~Synchronizer_Common() {
        begin_atomic();
        wakeup_all();
    }

    // A
    bool
    int
        return CPU::finc(number); }
    int fdec(volatile int & number) {
        return CPU::fdec(number); }

    // Thread operations
    void begin_atomic() { Thread::lock(); }
    void end_atomic() { Thread::unlock(); }

    void sleep() { Thread::sleep(&_queue); }
    void wakeup() { Thread::wakeup(&_queue); }
    void wakeup_all() {
        Thread::wakeup_all(&_queue); }

protected:
    Queue _queue;
};
```

a reference to the queue currently waiting on

```
class Thread
{
        :
protected:
    static void sleep(Queue * q);
    static void wakeup(Queue * q);
    static void wakeup_all(Queue * q);

        :

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

        :
};
```

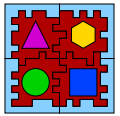




```
void Thread::sleep(Queue * q)
{
    db<Thread>(TRC) << "Thread::sleep(running="
                    << running() << ",q="
                    << q << ")" << endl;

    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();
}

void Thread::wakeup(Queue * q)
{
    db<Thread>(TRC) << "Thread::wakeup(running="
                    << running() << ",q="
                    << q << ")" << endl;
    assert(locked());

    if(!q->empty()) {
        Thread * t = q->remove()->object();
        t->_state = READY;
        t->_waiting = 0;
        _ready.insert(&t->_link);
    }
    unlock();
}


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

    assert(locked());

    while(!q->empty()) {
        Thread * t = q->remove()->object();
        t->_state = READY;
        t->_waiting = 0;
        _ready.insert(&t->_link);
    }

    unlock();
}
```

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

    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();
}

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

callers must have locked before, thus ensuring atomicity



```
        Thread * t = q->remove()->object();
        t->_state = READY;
        t->_waiting = 0;
        _ready.insert(&t->_link);
    }
    unlock();
}


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

    assert(locked());

    while(!q->empty()) {
        Thread * t = q->remove()->object();
        t->_state = READY;
        t->_waiting = 0;
        _ready.insert(&t->_link);
    }

    unlock();
}
```

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

    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();
}

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

#assert

```
#ifdef NDEBUG
#define assert(condition) ((void)0)
#else
#define assert(condition) /*implementation defined*/
#endif
```

If NDEBUG is defined then assert is a **model checking construct**

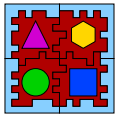
If NDEBUG is not defined, then assert checks the `condition` and outputs implementation-specific diagnostic information if it does not hold. In POSIX, it prints on the standard error output and calls std::abort.

```
    assert(locked());

    while(!q->empty()) {
        Thread * t = q->remove()->object();
        t->_state = READY;
        t->_waiting = 0;
        _ready.insert(&t->_link);
    }

    unlock();
}
```

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

    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();
}

void Thread::wakeup(Queue * q)
{
    db<Thread>(TRC) << "Thread::wakeup(running="
                    << running() << ",q="
                    << q << ")" << endl;
    assert(locked());

    if(!q->empty()) {
        Thread * t = q->remove()->object();
        t->_state = READY;
        t->_waiting = 0;
        _ready.insert(&t->_link);
    }

    unlock();
}
```

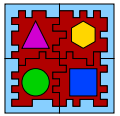
similar to suspend(), but holding a reference to the queue

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

    assert(locked());

    while(!q->empty()) {
        Thread * t = q->remove()->object();
        t->_state = READY;
        t->_waiting = 0;
        _ready.insert(&t->_link);
    }

    unlock();
}
```



similar to resume(), inserting threads in the READY queue

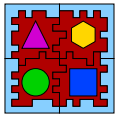
```
void Thr
{
    db<T                                    ning="
                    << running() << ",q="
                    << q << ")" << endl;

    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();
}

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

    assert(locked());

    if(!q->empty()) {
        Thread * t = q->remove()->object();
        t->_state = READY;
        t->_waiting = 0;
        _ready.insert(&t->_link);
    }
    unlock();
}


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

    assert(locked());

    while(!q->empty()) {
        Thread * t = q->remove()->object();
        t->_state = READY;
        t->_waiting = 0;
        _ready.insert(&t->_link);
    }

    unlock();
}
```



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

    assert(locked());

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

    Thread * prev = running();
```

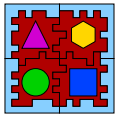
releases several threads at once (for Condition::broadcast())

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

    dispatch(prev, _running);
    unlock();
}

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

    assert(locked());

    if(!q->empty()) {
        Thread * t = q->remove()->object();
        t->_state = READY;
        t->_waiting = 0;
        _ready.insert(&t->_link);
    }
    unlock();
}


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

    assert(locked());

    while(!q->empty()) {
        Thread * t = q->remove()->object();
        t->_state = READY;
        t->_waiting = 0;
        _ready.insert(&t->_link);
    }

    unlock();
}
```

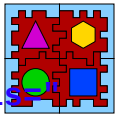


```
void Thread::sleep(Queue * q)
{
    db<Thread>(TRC) << "Thread::sleep(running="
                    << running() << ",q="
                    << q << ")" << endl;

    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();
}

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

    assert(locked());

    if(!q->empty()) {
        Thread * t = q->remove()->object();
        t->_state = READY;
        t->_waiting = 0;
        _ready.insert(&t->_link);
    }
    unlock();
}


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

    assert(locked());

    while(!q->empty()) {
        Thread * t = q->remove()->object();
        t->_state = READY;
        t->_waiting = 0;
        _ready.insert(&t->_link);
    }

    unlock();
}
```

there are many queue operations here that can get involved in race conditions!



```
void Mutex::lock()
{
    db<Synchronizer>(TRC) << "Mutex::lock(this="
                          << this << ")" << endl;

    begin_atomic();
    if(tsl(_locked))
        sleep(); // implicit end_atomic()
    else
        end_atomic();
}


void Mutex::unlock()
{
    db<Synchronizer>(TRC) << "Mutex::unlock(this="
                          << this << ")" << endl;

    begin_atomic();
    if(_queue.empty()) {
        _locked = false;
        end_atomic();
    } else
        wakeup(); // implicit end_atomic()
}
```

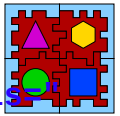


```
void Semaphore::p()
{
    db<Synchronizer>(TRC) << "Semaphore::p(this="
                          << this << ",value="
                          << _value << ")" << endl;

    begin_atomic();
    if(fdec(_value) < 1)
        sleep(); // implicit end_atomic()
    else
        end_atomic();
}


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

    begin_atomic();
    if(finc(_value) < 0)
        wakeup();  // implicit end_atomic()
    else
        end_atomic();
}
```

```
void Mutex::lock()
{
    db<Synchronizer>(TRC) << "Mutex::lock(this="
                          << this << ")" << endl;

    begin_atomic();
    if(tsl(_locked))
        sleep(); // implicit end_atomic()
    else
        end_atomic();
}
```

```
void Mutex::unlock()
{
    db<Synchronizer>(TRC) << "Mutex::unlock(this="
                          << this << ")" << endl;

    begin_atomic();
    if(_queue.empty()) {
        _locked = false;
        end_atomic();
    } else
        wakeup(); // implicit end_atomic()
}
```

loop removed



```
                                    re::p(this
                                    << this << ",value="
                                    << _value << ")" << endl;

    begin_atomic();
    if(fdec(_value) < 1)
        sleep(); // implicit end_atomic()
    else
        end_atomic();
}
```

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

    begin_atomic();
    if(finc(_value) < 0)
        wakeup();  // implicit end_atomic()
    else
        end_atomic();
}
```

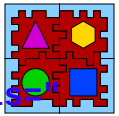
```
void Mutex::lock()
{
    db<Synchronizer>(TRC) << "Mutex::lock(this="
                          << this << ")" << endl;

    begin_atomic();
    if(tsl(_locked))
        sleep(); // implicit end_atomic()
    else
        end_atomic();
}
```

```
void Mutex::unlock()
{
    db<Synchronizer>(TRC) << "Mutex::unlock(this="
                          << this << ")" << endl;

    begin_atomic();
    if(_queue.empty()) {
        _locked = false;
        end_atomic();
    } else
        wakeup(); // implicit end_atomic()
}
```

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

    begin_atomic();
    if(fdec(_value) < 1)
        sleep(); // implicit end_atomic()
    else
        end_atomic();
}
```

only unlock when the queue is empty, thus preventing race conditions

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

    begin_atomic();
    if(finc(_value) < 0)
        wakeup();  // implicit end_atomic()
    else
        end_atomic();
}
```

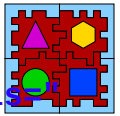


```
void Mutex::lock()
{
    db<Synchron                                          ;

    begin_atomic();
    if(tsl(_locked))
        sleep(); // implicit end_atomic()
    else
        end_atomic();
}
```

loop removed

```
void Mutex::unlock()
{
    db<Synchronizer>(TRC) << "Mutex::unlock(this="
                          << this << ")" << endl;

    begin_atomic();
    if(_queue.empty()) {
        _locked = false;
        end_atomic();
    } else
        wakeup(); // implicit end_atomic()
}
```

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

    begin_atomic();
    if(fdec(_value) < 1)
        sleep(); // implicit end_atomic()
    else
        end_atomic();
}
```

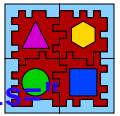


```
void Semaphore::v()
{
    db<Synchronizer>(TRC) << "Semaphore::v(this="
                          << this << ",value="
                          << _value << ")" << endl;

    begin_atomic();
    if(finc(_value) < 0)
        wakeup();  // implicit end_atomic()
    else
        end_atomic();
}
```

```
void Mutex::lock()
{
    db<Synchronizer>(TRC) << "Mutex::lock(this="
                          << this << ")" << endl;

    begin_atomic();
    if(tsl(_locked))
        )
}
```

ensures ordering

```
void Mutex::unlock()
{
    db<Synchronizer>(TRC) << "Mutex::unlock(this="
                          << this << ")" << endl;

    begin_atomic();
    if(_queue.empty()) {
        _locked = false;
        end_atomic();
    } else
        wakeup(); // implicit end_atomic()
}
```



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

    begin_atomic();
    if(fdec(_value) < 1)
        sleep(); // implicit end_atomic()
    else
        end_atomic();
}
```

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

    begin_atomic();
    if(finc(_value) < 0)
        wakeup();  // implicit end_atomic()
    else
        end_atomic();
}
```

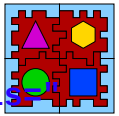
```
void Mutex::lock()
{
    db<Synchronizer>(TRC) << "Mutex::lock(this="
                          << this << ")" << endl;

    begin_atomic();
    if(tsl(_locked))
        sleep(); // implicit end_atomic()
    else
        end_atomic();
}
```

```
void Mutex::unlock()
{
    db<Synchroniz
    begin_atomic();
    if(_queue.empty()) {
        _locked = false;
        end_atomic();
    } else
        wakeup(); // implicit end_atomic()
}
```

compares finc()'s result
(not _value)

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

    begin_atomic();
    if(fdec(_value) < 1)
        sleep(); // implicit end_atomic()
    else
        end_atomic();
}
```

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

    begin_atomic();
    if(finc(_value) < 0)
        wakeup();  // implicit end_atomic()
    else
        end_atomic();
}
```

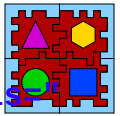


```
void Mutex::lock()
{
    db<Synchronizer>(TRC) << "Mutex::lock(this="
                          << this << ")" << endl;

    begin_atomic();
    if(tsl(_locked))
        sleep(); // implicit end_atomic()
    else
        end_atomic();
}
```

```
void Mutex::unlock()
{
    db<Synchronizer>(TRC) << "Mutex::unlock(this="
                          << this << ")" << endl;

    begin_atomic();
    if(_queue.empty()) {
        _locked = false;
        end_atomic();
    } else
        w
}
```

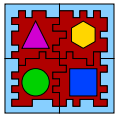
are CPU::finc() and Thread:lock() enough to ensure queues won't be corrupted?

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

    begin_atomic();
    if(fdec(_value) < 1)
        sleep(); // implicit end_atomic()
    else
        end_atomic();
}
```



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

    begin_atomic();
    if(finc(_value) < 0)
        wakeup();  // implicit end_atomic()
    else
        end_atomic();
}
```

```
class Thread
{
      :

protected:
    static void sleep(Queue * q);
    static void wakeup(Queue * q);
    static void wakeup_all(Queue * q);

      :

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

      :
};

template<typename ... Tn>
inline Thread::Thread(int (* entry)(Tn ...), Tn ... an)
: _state(READY), _waiting(0), _joining(0),
  _link(this, NORMAL) {
    constructor_prologue(STACK_SIZE);
    _context = CPU::init_stack(0, _stack + STACK_SIZE,
                               &__exit, entry, an ...);

    constructor_epilogue(entry, STACK_SIZE);
}
```

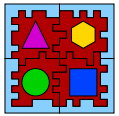


```
Thread::~Thread()
{
    lock();

    db<Thread>(TRC) << "~Thread(this=" << this
                    << ",state=" << _state
                    << ",priority="
                    << _link.rank()
                    << ",stack={b="
                    << reinterpret_cast<void *>
                       (_stack)
                    << ",context={b=" << _context
                    << "," << *_context << "})"
                    << endl;

    _ready.remove(this);
    _suspended.remove(this);

    if(_waiting)
        _waiting->remove(this);

    if(_joining)
        _joining->resume();

    unlock();

    kfree(_stack);
}
```

```
class Thread
{
      :

protected:
    static void sleep(Queue * q);
    static void wakeup(Queue * q);
    static void wakeup_all(Queue * q);

      :

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

      :
};

template<typename ... Tn>
inline Thread::Thread(int (* entry)(Tn ...), Tn ... an)
: _state(READY), _waiting(0), _joining(0),
  _link(this, NORMAL) {
    constructor_prologue(STACK_SIZE);
    _context = CPU::init_stack(0, _stack + STACK_SIZE,
                               &__exit, entry, an ...);

    constructor_epilogue(entry, STACK_SIZE);
}
```

adding a pointer to an eventual joiner (a single one; no queue)

```
Thread::~Thread()
{
    lock();

                                    this=" << this
                                     << _state
                                    y="
                                << _link.rank()
                                << ",stack={b="
                                << reinterpret_cast<void *>
                                   (_stack)
                                << ",context={b=" << _context
                                << "," << *_context << "})"
                                << endl;

    _ready.remove(this);
    _suspended.remove(this);

    if(_waiting)
        _waiting->remove(this);

    if(_joining)
        _joining->resume();

    unlock();

    kfree(_stack);
}
```



```
class Thread
{
      :

protected:
    static void sleep(Queue * q);
    static void wakeup(Queue * q);
    static void wakeup_all(Queue * q);

      :

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

      :
};

template<typename ... Tn>
inline Thread::Thread(int (* entry)(Tn ...), Tn ... an)
: _state(READY), _waiting(0), _joining(0),
  _link(this, NORMAL) {
    constructor_prologue(STACK_SIZE);
    _context = CPU::init_stack(0, _stack + STACK_SIZE,
                               &__exit, entry, an ...);

    constructor_epilogue(entry, STACK_SIZE);
}
```

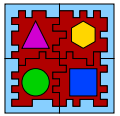
initialize pointers with null (0)

```
Thread::~Thread()
{
    lock();

    db<Thread>(TRC) << "~Thread(this=" << this
                    << ",state=" << _state
                    << ",priority="
                    << _link.rank()
                    << ",stack={b="
                                          cast<void *>
                                          =" << _context
                                          << "})"
                    << endl;

    _ready.remove(this);
    _suspended.remove(this);

    if(_waiting)
        _waiting->remove(this);

    if(_joining)
        _joining->resume();

    unlock();

    kfree(_stack);
}
```

```
class Thread
{
      :

protected:
    static void sleep(Queue * q);
    static void wakeup(Queue * q);
    static void wakeup_all(Queue * q);

      :

protected:
    char * _stack;
    Context * volatile _context;
    volatile State _state;
    Queue * _waiting;
};

template<typename ... Tn>
inline Thread::Thread(int (* entry)(Tn ...), Tn ... an)
: _state(READY), _waiting(0), _joining(0),
  _link(this, NORMAL) {
    constructor_prologue(STACK_SIZE);
    _context = CPU::init_stack(0, _stack + STACK_SIZE,
                               &__exit, entry, an ...);

    constructor_epilogue(entry, STACK_SIZE);
}
```

if the deleted thread was waiting in a Synchronizer's queue, remove it

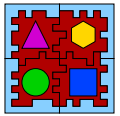
```
Thread::~Thread()
{
    lock();

    db<Thread>(TRC) << "~Thread(this=" << this
                    << ",state=" << _state
                    << ",priority="
                    << _link.rank()
                    << ",stack={b="
                    << reinterpret_cast<void *>
                       (_stack)
                    << ",context={b=" << _context
                    << "," << *_context << "})"
                    << endl;

    _ready.remove(this);
    _suspended.remove(this);

    if(_waiting)
        _waiting->remove(this);

    if(_joining)
        _joining->resume();

    unlock();

    kfree(_stack);
}
```



```
class Thread
{
      :

protected:
    static void sleep(Queue * q);
    static void wakeup(Queue * q);
    static void wakeup_all(Queue * q);

      :

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

      :
};

templa
inline                                    ...), Tn ... an)
: _sta                                    0),
  _link(this, NORMAL) {
    constructor_prologue(STACK_SIZE);
    _context = CPU::init_stack(0, _stack + STACK_SIZE,
                               &__exit, entry, an ...);

    constructor_epilogue(entry, STACK_SIZE);
}
```

if someone was waiting for this thread termination, resume it

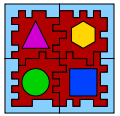
```
Thread::~Thread()
{
    lock();

    db<Thread>(TRC) << "~Thread(this=" << this
                    << ",state=" << _state
                    << ",priority="
                    << _link.rank()
                    << ",stack={b="
                    << reinterpret_cast<void *>
                       (_stack)
                    << ",context={b=" << _context
                    << "," << *_context << "})"
                    << endl;

    _ready.remove(this);
    _suspended.remove(this);

    if(_waiting)
        _waiting->remove(this);

    if(_joining)
        _joining->resume();

    unlock();

    kfree(_stack);
}
```



```
class Thread
{
      :

protected:
    static void sleep(Queue * q);
    static void wakeup(Queue * q);
    static void wakeup_all(Queue * q);

      :

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

      :
};

templa
inline                                    ...), Tn ... an)
: _sta                                    0),
  _link(this, NORMAL) {
    constructor_prologue(STACK_SIZE);
    _context = CPU::init_stack(0, _stack + STACK_SIZE,
                               &__exit, entry, an ...);

    constructor_epilogue(entry, STACK_SIZE);
}
```

if someone was waiting for this thread termination, resume it

```
Thread::~Thread()
{
    lock();

    db<Thread>(TRC) << "~Thread(this=" << this
                    << ",state=" << _state
                    << ",priority="
                    << _link.rank()
                    << ",stack={b="
                    << reinterpret_cast<void *>
                       (_stack)
                    << ",context={b=" << _context
                    << "," << *_context << "})"
                    << endl;

    _ready.remove(this);
    _suspended.remove(this);

    if(_waiting)
        _waiting->remove(this);

    if(_joining)
        _joining->resume();

    unlock();

    kfree(_stack);
}
```



```
class Thread
{
      :

protected:
    static void sleep(Queue * q);
    static void wakeup(Queue * q);
    static void wakeup_all(Queue * q);

      :

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

      :
};

templa
inline                                          ...), Tn ... an)
: _sta                                          0),
  _link(this, NORMAL) {
    constructor_prologue(STACK_SIZE);
    _context = CPU::init_stack(0, _stack + STACK_SIZE,
                               &__exit, entry, an ...);

    constructor_epilogue(entry, STACK_SIZE);
}
```

if someone was waiting for this thread termination, resume it

```
    _suspended.remove(this);

    if(_waiting)
        _waiting->remove(this);

    if(_joining)
        _joining->resume();

    unlock();

    kfree(_stack);
}
```

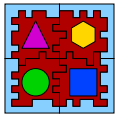
Process, Task and Thread

A **process** is a program in execution. It has an **address space**, mapping the memory where the program's **code** and **data** are stored, one or more **threads**, and a set of OS **resources**.

A **thread** is an execution flow of a process. It has a **state**, a **context**, and a **stack**. It also has some sort of **id**.

A task can be:
- The passive part of a process (i.e. address space, code and data);
- A synonym for thread.

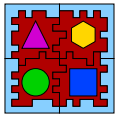
```
int Thread::join()
{
    lock();

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

    // Precondition: no Thread::self()->join()
    assert(running() != this);

    // Precondition: a single joiner
    assert(!_joining);

    if(_state != FINISHING) {
        _joining = running();
        _joining->suspend();
    } else
        unlock();

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



```
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();
}
```

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

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

    // Precondition: no Thread::self()->join()
    assert(running() != this);

    // Precondition: a single joiner
    assert(!_joining);

    if(_state != FINISHING) {
        _joining = running();
        _joining->suspend();
    } else
        unlock();

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

precondition: a Thread cannot join itself

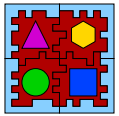


```
    db<Thread>(TRC) << "Thread::suspend(this="
                    << this << ")" << endl;

    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();
}
```

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

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

    // Precondition: no Thread::self()->join()
    assert(running() != this);

    // Precondition: a single joiner
    assert(!_joining);

    if(_state != FINISHING) {
        _joining = running();
        _joining->suspend();
    } else
        unlock();

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

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

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

    _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();
}
```

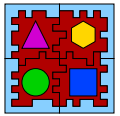
precondition: only one joiner a time



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

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

    // Precondition: no Thread::self()->join()
    assert(running() != this);

    // Precondition: a single joiner
    assert(!_joining);

    if(_state != FINISHING) {
        _joining = running();
        _joining->suspend();
    } else
        unlock();

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

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

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

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

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

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

        dispatch(this, _running);
    }

    unlock();
}
```

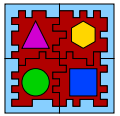
if the Thread hasn't finished yet, suspend the caller



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

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

    // Precondition: no Thread::self()->join()
    assert(running() != this);

    // Precondition: a single joiner
    assert(!_joining);

    if(_state != FINISHING) {
        _joining = running();
        _joining->suspend();
    } else
        unlock();

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

```
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();

        dispatch(this, _running);
    }

    unlock();
}
```

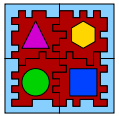
returns to the caller what the Thread returned when finished



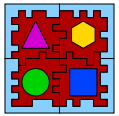
```
int Thread::join()
{
    lock();

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

    // Precondition: no Thread::self()->join()
    assert(running() != this);

    // Precondition: a single joiner
    assert(!_j

    if(_state
        _joining = running();
        _joining->suspend();
    } else
        unlock();

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

wait for a Thread to become READY before actually suspending

```
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();
}
```



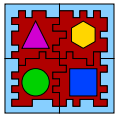
```
void Thread::exit(int status)
{
    lock();

    db<Thread>(TRC) << "Thread::exit(status="
                    << status << ") [running="
                    << running() << "]" << endl;

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

    if(prev->_joining) {
        Thread * joining = prev->_joining;
        prev->_joining = 0;
        joining->resume(); // implicit unlock()
        lock();
    }

    if(_ready.empty()) {
        if(!_suspended.empty()) {
            while(_ready.empty())
                idle(); // implicit unlock();
            lock();
        } 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();
            }
        }
    } else {
        _running = _ready.remove()->object();
        _running->_state = RUNNING;

        dispatch(prev, _running);
    }

    unlock();
}
```



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

    db<Thread>(TRC) << "Thread::exit(status="
                    << status << ") [running="
                    << running() << "]" << endl;

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

    if(prev->_joining) {
        Thread * joining = prev->_joining;
        prev->_joining = 0;
        joining->resume(); // implicit unlock()
        lock();
    }
```

set state to FINISHING and save the return value

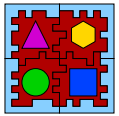
```
    if(_ready.empty()) {
        if(!_suspended.empty()) {
            while(_ready.empty())
                                    it unlock();

                                    he 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();
            }
        }
    } else {
        _running = _ready.remove()->object();
        _running->_state = RUNNING;

        dispatch(prev, _running);
    }

    unlock();
}
```



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

    db<Thread>(TRC) << "Thread::exit(status="
                    << status << ") [running="
                    << running() << "]" << endl;

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

    if(prev->_joining) {
        Thread * joining = prev->_joining;
        prev->_joining = 0;
        joining->resume(); // implicit unlock()
        lock();
    }
```

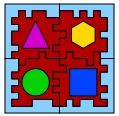
if someone was waiting for this Thread to exit, wake him up

```
    if(_ready.empty()) {
        if(!_suspended.empty()) {
            while(_ready.empty())
                idle(); // implicit unlock();
            lock();
        } else {
            db<Thread>(WRN) << "The last thread in"
                << "the system has exited!"
                << endl;
                << "Rebooting the machine …"
                << endl;
            Machine::reboot();
        } else {
            db<Thread>(WRN)
              << "Halting the CPU ..." << endl;
            CPU::halt();
        }
    }
} else {
    _running = _ready.remove()->object();
    _running->_state = RUNNING;

    dispatch(prev, _running);
}

unlock();
}
```



wait for a Thread to become READY before actually exiting

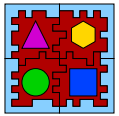
```
void Thread::exit(int status)
{
    db<Thread>(TRC) << "Thread::exit(status="
                    << status << ") [running="
                    << running() << "]" << endl;

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

    if(prev->_joining) {
        Thread * joining = prev->_joining;
        prev->_joining = 0;
        joining->resume(); // implicit unlock()
        lock();
    }

    if(_ready.empty()) {
        if(!_suspended.empty()) {
            while(_ready.empty())
                idle(); // implicit unlock();
            lock();
        } 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();
            }
        }
    } else {
        _running = _ready.remove()->object();
        _running->_state = RUNNING;

        dispatch(prev, _running);
    }

    unlock();
}
```

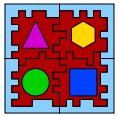


```
void Thread::exit(int status)
{
    lock();

    db<Thread>(TRC) << "Thread::exit(status="
                    << status << ") [running="
                                          << endl;
    prev->_state = FINISHING;
    *reinterpret_cast<int *>(prev->_stack) =
        status;

    if(prev->_joining) {
        Thread * joining = prev->_joining;
        prev->_joining = 0;
        joining->resume(); // implicit unlock()
        lock();
    }
```

if both READY and SUSPENDED queues are empty, then ...

```
    if(_ready.empty()) {
        if(!_suspended.empty()) {
            while(_ready.empty())
                idle(); // implicit unlock();
            lock();
        } 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();
            }
        }
    } else {
        _running = _ready.remove()->object();
        _running->_state = RUNNING;

        dispatch(prev, _running);
    }

    unlock();
}
```





Embedded Parallel Operating System

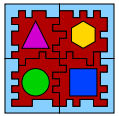
Coding Journey through OS Design
–from co-routines to a multicore kernel–

Prof. Antônio Augusto Fröhlich, Ph.D.

UFSC / LISHA

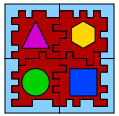
September 30, 2020

General Pointers



- Site
  https://epos.lisha.ufsc.br
- Documentation
  https://epos.lisha.ufsc.br/EPOS+2+User+Guide
- Git
  https://gitlab.lisha.ufsc.br/epos/epos/tree/master
- Cross-compilers
  https://epos.lisha.ufsc.br/EPOS+Software

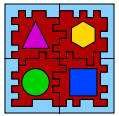
Co-routines: the minimal OS



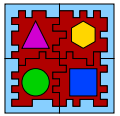
“Co-routines are computer program components that generalize subroutines for non-preemptive multitasking, by allowing execution to be suspended and resumed. Co-routines are well-suited for implementing familiar program components such as cooperative tasks, exceptions, event loops, iterators, infinite lists and pipes.

According to Donald Knuth, Melvin Conway coined the term co-routine in 1958 when he applied it to construction of an assembly program.”

(Wikipedia)



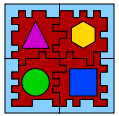
```
// EPOS Semaphore Component Test Program

#include <machine/display.h>
#include <time.h>
#include <synchronizer.h>
#include <process.h>

using namespace EPOS;

const int iterations = 10;

Mutex table;

Thread * phil[5];
Semaphore * chopstick[5];

OStream cout;

int philosopher(int n, int l, int c)
{
    int first = (n < 4)? n : 0;
    int second = (n < 4)? n + 1 : 4;

    for(int i = iterations; i > 0; i--) {

        table.lock();
        Display::position(l, c);
        cout << "thinking";
        table.unlock();

        Delay thinking(2000000);

        // get first chopstick
        chopstick[first]→p();
        // get second chopstick
        chopstick[second]->p();

        table.lock();
        Display::position(l, c);
        cout << " eating ";
        table.unlock();

        Delay eating(1000000);

        // release first chopstick
        chopstick[first]->v();
        // release second chopstick
        chopstick[second]->v();
    }

    table.lock();
    Display::position(l, c);
    cout << "  done  ";
    table.unlock();

    return iterations;
}
```



```
// EPOS Semaphore Component Test Program

#include <machine/display.h>
#include <time.h>
#include <synchronizer.h>
#include <process.h>

using namespace EPOS;

const int iterations = 10;

Mutex table;

Thread * phil[5];
Semaphore * chopstick[5];

OStream cout;

int philosopher(int n, int l, int c)
{
    int first = (n < 4)? n : 0;
    int second = (n < 4)? n + 1 : 4;

    for(int i = iterations; i > 0; i--) {

        table.lock();
        Display::position(l, c);
        cout << "thinking";
        table.unlock();
```

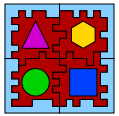
API inclusion

```
                                    ng(2000000);

                                     chopstick
                                    chopstick[first]→p();
                                    // get second chopstick
                                    chopstick[second]->p();

                                    table.lock();
                                    Display::position(l, c);
                                    cout << " eating ";
                                    table.unlock();

                                    Delay eating(1000000);

                                    // release first chopstick
                                    chopstick[first]->v();
                                    // release second chopstick
                                    chopstick[second]->v();
                                }

                                table.lock();
                                Display::position(l, c);
                                cout << "  done  ";
                                table.unlock();

                                return iterations;
                            }
```



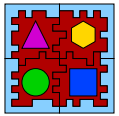
```
// EPOS Semaphore Component Test Program

#include <machine/display.h>
#include <time.h>
#include <synchronizer.h>
#include <process.h>

using namespace EPOS;

const int iterations = 10;

Mutex table;

Thread * phil[5];
Semaphore * chopstick[5];

OStream cout;

int philosopher(int n, int l, int c)
{
    int first = (n < 4)? n : 0;
    int second = (n < 4)? n + 1 : 4;

    for(int i = iterations; i > 0; i--) {

        table.lock();
        Display::position(l, c);
        cout << "thinking";
        table.unlock();

        Delay thinking(2000000);

        // get first chopstick
        chopstick[first]→p();
        // get second chopstick
        chopstick[second]->p();

        table.lock();
        Display::position(l, c);
        cout << " eating ";
        table.unlock();

        Delay eating(1000000);

        // release first chopstick
        chopstick[first]->v();
        // release second chopstick
        chopstick[second]->v();
    }

    table.lock();
    Display::position(l, c);
    cout << "  done  ";
    table.unlock();

    return iterations;
}
```

namespace import



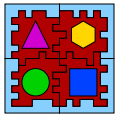
```
// EPOS Semaphore Component Test Program

#include <machine/display.h>
#include <time.h>
#include <synchronizer.h>
#include <process.h>

using namespace EPOS;

const int iterations = 10;

Mutex table;

Thread * phil[5];
Semaphore * chopstick[5];

OStream cout;

int philosopher(int n, int l, int c)
{
    int first = (n < 4)? n : 0;
    int second = (n < 4)? n + 1 : 4;

    for(int i = iterations; i > 0; i--) {

        table.lock();
        Display::position(l, c);
        cout << "thinking";
        table.unlock();

        Delay thinking(2000000);

        // get first chopstick
        chopstick[first]→p();
        // get second chopstick
        chopstick[second]->p();

        table.lock();
        Display::position(l, c);
        cout << "  eating  ";
        table.unlock();

        Delay eating(1000000);

        // release first chopstick
        chopstick[first]->v();
        // release second chopstick
        chopstick[second]->v();
    }

    table.lock();
    Display::position(l, c);
    cout << "  done  ";
    table.unlock();

    return iterations;
}
```

system objects can be statically allocated
(constructor called before main())



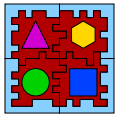
```
// EPOS Semaphore Component Test Program

#include <machine/display.h>
#include <time.h>
#include <synchronizer.h>
#include <process.h>

using namespace EPOS;

const int iterations = 10;

Mutex table;

Thread * phil[5];
Semaphore * chopstick[5];

OStream cout;

int philosopher(int n, int l, int c)
{
    int first = (n < 4)? n : 0;
    int second = (n < 4)? n + 1 : 4;

    for(int i = iterations; i > 0; i--) {

        table.lock();
        Display::position(l, c);
        cout << "thinking";
        table.unlock();

        Delay thinking(2000000);

        // get first chopstick
        chopstick[first]→p();
        // get second chopstick
        chopstick[second]->p();

        table.lock();
        Display::position(l, c);
        cout << " eating ";
        nlock();

        ating(1000000);

        // release first chopstick
        chopstick[first]->v();
        // release second chopstick
        chopstick[second]->v();
    }

    table.lock();
    Display::position(l, c);
    cout << "  done  ";
    table.unlock();

    return iterations;
}
```

or dynamically



```
// EPOS Semaphore Component Test Program

#include <machine/display.h>
#include <time.h>
#include <synchronizer.h>
#include <process.h>

using namespace EPOS;

const int iterations = 10;

Mutex table;

Thread * phil[5];
Semaphore * chopstick[5];

OStream cout;

int philosopher(int n, int l, int c)
{
    int first = (n < 4)? n : 0;
    int second = (n < 4)? n + 1 : 4;

    for(int i = iterations; i > 0; i--) {

        table.lock();
        Display::position(l, c);
        cout << "thinking";
        table.unlock();

        Delay thinking(2000000);

        // get first chopstick
        chopstick[first]→p();
        // get second chopstick
        chopstick[second]->p();

        table.lock();
        Display::position(l, c);
        cout << " eating ";
        table.unlock();

        Delay eating(1000000);

        // release first chopstick
        chopstick[first]->v();
        // release second chopstick
        chopstick[second]->v();
    }

    table.lock();
    Display::position(l, c);
    cout << "  done  ";
    table.unlock();

    return iterations;
}
```

cout not created by default



```
// EPOS Semaphore Component Test Program

#include <machine/display.h>
#include <time.h>
#include <synchronizer.h>
#include <process.h>

using namespace EPOS;

const int iterations = 10;

Mutex table;

Thread * phil[5];
Semaphore * chopstick[5];

OStream cout;

int philosopher(int n, int l, int c)
{
    int first = (n < 4)? n : 0;
    int second = (n < 4)? n + 1 : 4;

    for(int i = iterations; i > 0; i--) {

        table.lock();
        Display::position(l, c);
        cout << "thinking";
        table.unlock();

        Delay thinking(2000000);

        // get first chopstick
        chopstick[first]→p();
        // get second chopstick
        chopstick[second]->p();

        table.lock();
        Display::position(l, c);
        cout << " eating ";
        table.unlock();

        Delay eating(1000000);

        // release first chopstick
        chopstick[first]->v();
                                                    stick
                                                    ;

        table.lock();
        Display::position(l, c);
        cout << "  done  ";
        table.unlock();

        return iterations;
    }
```

any function can become a thread

Lamport's deadlock-free solution to the Dining Philosophers Problem

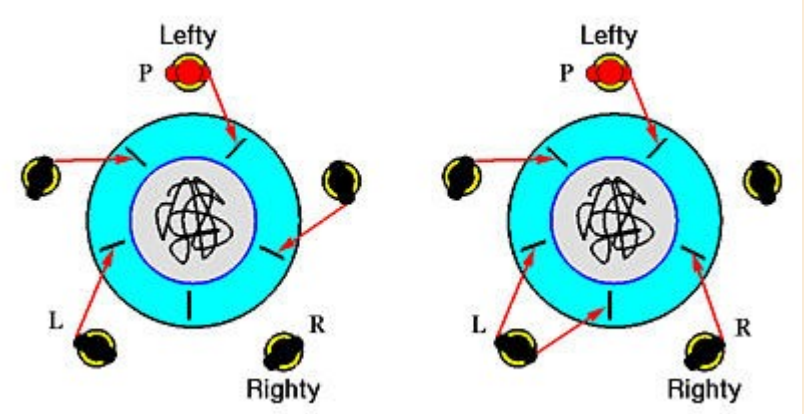
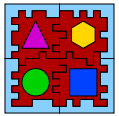


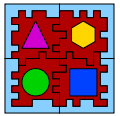
Figure by Dr. C.-K. Shene

```
int philosopher(int n, int l, int c)
{
    int first = (n < 4)? n : 0;
    int second = (n < 4)? n + 1 : 4;

    for(int i = iterations; i > 0; i--) {

        table.lock();
        Display::position(l, c);
        cout << "thinking";
        table.unlock();

        Delay thinking(2000000);

        // get first chopstick
        chopstick[first]→p();
        // get second chopstick
        chopstick[second]->p();

        table.lock();
        Display::position(l, c);
        cout << " eating ";
        table.unlock();

        Delay eating(1000000);

        // release first chopstick
        chopstick[first]->v();
        // release second chopstick
        chopstick[second]->v();
    }

    table.lock();
    Display::position(l, c);
    cout << "  done  ";
    table.unlock();

    return iterations;
}
```



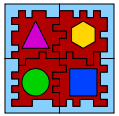
```
// EPOS Semaphore Component Test Program

#include <machine/display.h>
#include <time.h>
#include <synchronizer.h>
#include <process.h>

using namespace EPOS;

const int iterations = 10;

Mutex table;

Thread * phil[5];
Semaphore * chopstick[5];

OStream cout;

int philosopher(int n, int l, int c)
{
    int first = (n < 4)? n : 0;
    int second = (n < 4)? n + 1 : 4;

    for(int i = iterations; i > 0; i--) {

        table.lock();
        Display::position(l, c);
        cout << "thinking";
        table.unlock();

        Delay thinking(2000000);

        // get first chopstick
        chopstick[first]→p();
        // get second chopstick
        chopstick[second]->p();

        table.lock();
        Display::position(l, c);
        cout << " eating ";
        table.unlock();

        Delay eating(1000000);

        // release first chopstick
        chopstick[first]->v();
        // release second chopstick
        chopstick[second]->v();
    }

    table.lock();
    Display::position(l, c);
    cout << "  done  ";
    k();

    ations;
    }
```

atomic printing



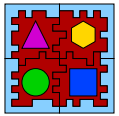
wait (in µs)

```
// EPOS Semaphore Component Test Program

#in
#in
#include <synchronizer.h>
#include <process.h>

using namespace EPOS;

const int iterations = 10;

Mutex table;

Thread * phil[5];
Semaphore * chopstick[5];

OStream cout;

int philosopher(int n, int l, int c)
{
    int first = (n < 4)? n : 0;
    int second = (n < 4)? n + 1 : 4;

    for(int i = iterations; i > 0; i--) {

        table.lock();
        Display::position(l, c);
        cout << "thinking";
        table.unlock();

        Delay thinking(2000000);

        // get first chopstick
        chopstick[first]→p();
        // get second chopstick
        chopstick[second]->p();

        table.lock();
        Display::position(l, c);
        cout << " eating ";
        table.unlock();

        Delay eating(1000000);

        // release first chopstick
        chopstick[first]->v();
        // release second chopstick
        chopstick[second]->v();
    }

    table.lock();
    Display::position(l, c);
    cout << "  done  ";
    table.unlock();

    return iterations;
}
```

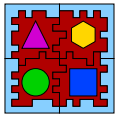


```
// EPOS Semaphore Component Test Program

#include <machine/display.h>
#include <time.h>
#include <synchronizer.h>
#include <process.h>

using namespace EPOS;

const int iterations = 10;

Mutex table;

Thread * phil[5];
Semaphore * chopstick[5];

OStream cout;

int philosopher(int n, int l, int c)
{
    int first = (n < 4) ? n : 0;
    int second = (n < 4) ? n + 1 : 4;

    for(int i = iterations; i > 0; i--) {

        table.lock();
        Display::position(l, c);
        cout << "thinking";
        table.unlock();

        Delay thinking(2000000);

        // get first chopstick
        chopstick[first]→p();
        // get second chopstick
        chopstick[second]->p();

        table.lock();
        Display::position(l, c);
        cout << " eating ";
        table.unlock();

        Delay eating(1000000);

        // release first chopstick
        chopstick[first]->v();
        // release second chopstick
        chopstick[second]->v();
    }

    table.lock();
    Display::position(l, c);
    cout << "  done  ";
    table.unlock();

    return iterations;
}
```

implicit **exit()** if a thread

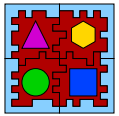


```
int main()
{
    table.lock();
    Display::clear();
    Display::position(0, 0);
    cout << "The Philosopher's Dinner:" << endl;

    for(int i = 0; i < 5; i++)
        chopstick[i] = new Semaphore;

    phil[0] = new Thread(&philosopher, 0,  5, 32);
    phil[1] = new Thread(&philosopher, 1, 10, 44);
    phil[2] = new Thread(&philosopher, 2, 16, 39);
    phil[3] = new Thread(&philosopher, 3, 16, 24);
    phil[4] = new Thread(&philosopher, 4, 10, 20);

    cout << "Philosophers are alive and hungry!"
         << endl;

    Display::position(7, 44);
    cout << '/';
    Display::position(13, 44);
    cout << '\\';
    Display::position(16, 35);
    cout << '|';
    Display::position(13, 27);
    cout << '/';
    Display::position(7, 27);
    cout << '\\';
    Display::position(19, 0);

    cout << "The dinner is served ..." << endl;
    table.unlock();

    for(int i = 0; i < 5; i++) {
        int ret = phil[i]->join();
        table.lock();
        Display::position(20 + i, 0);
        cout << "Philosopher " << i << " ate "
             << ret << " times " << endl;
        table.unlock();
    }

    for(int i = 0; i < 5; i++)
        delete chopstick[i];
    for(int i = 0; i < 5; i++)
        delete phil[i];

    cout << "The end!" << endl;

    return 0;
}
```



semaphore as a critical section guard (initialized with 1)

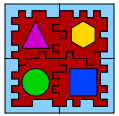
```
int main()
{
    table.lock();
    Display::clear();
    Display::position(0, 0);
    cout << "The Philosopher's Dinner:" << endl;

    for(int i = 0; i < 5; i++)
        chopstick[i] = new Semaphore;

    phil[0] = new Thread(&philosopher, 0,  5, 32);
    phil[1] = new Thread(&philosopher, 1, 10, 44);
    phil[2] = new Thread(&philosopher, 2, 16, 39);
    phil[3] = new Thread(&philosopher, 3, 16, 24);
    phil[4] = new Thread(&philosopher, 4, 10, 20);

    cout << "Philosophers are alive and hungry!"
         << endl;

    Display::position(7, 44);
    cout << '/';
    Display::position(13, 44);
    cout << '\\';
    Display::position(16, 35);
    cout << '|';
    Display::position(13, 27);
    cout << '/';
    Display::position(7, 27);
    cout << '\\';
    Display::position(19, 0);
    cout << "The dinner is served ..." << endl;
    table.unlock();

    for(int i = 0; i < 5; i++) {
        int ret = phil[i]->join();
        table.lock();
        Display::position(20 + i, 0);
        cout << "Philosopher " << i << " ate "
             << ret << " times " << endl;
        table.unlock();
    }

    for(int i = 0; i < 5; i++)
        delete chopstick[i];
    for(int i = 0; i < 5; i++)
        delete phil[i];

    cout << "The end!" << endl;

    return 0;
}
```



```
int main()
{
    table.lock();
    Display::clear();
    Display::position(0, 0);
    cout << "The Philosopher's Dinner:" << endl;

    for(int i = 0; i < 5; i++)
        chopstick[i] = new Semaphore;

    phil[0] = new Thread(&philosopher, 0,  5, 32);
    phil[1] = new Thread(&philosopher, 1, 10, 44);
    phil[2] = new Thread(&philosopher, 2, 16, 39);
    phil[3] = new Thread(&philosopher, 3, 16, 24);
    phil[4] = new Thread(&philosopher, 4, 10, 20);

    cout << "Philosophers are alive and hungry!"
         << endl;

    Display::position(7, 44);
    cout << '/';
    Display::position(13, 44);
    cout << '\\';
    Display::position(16, 35);
    cout << '|';
    Display::position(13, 27);
    cout << '/';
    Display::position(7, 27);
    cout << '\\';
    Display::position(19, 0);
```

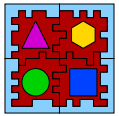
```
    cout << "The dinner is served ..." << endl;
    table.unlock();

    for(int i = 0; i < 5; i++) {
        int ret
        table.l
        Display
        cout <<
             << ret << " times " << endl;
        table.unlock();
    }

    for(int i = 0; i < 5; i++)
        delete chopstick[i];
    for(int i = 0; i < 5; i++)
        delete phil[i];

    cout << "The end!" << endl;

    return 0;
}
```

5 threads from a single function with different parameters



join waits for threads to finish

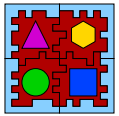
```
int main()
{
    table.lock();
    Display::clear();
    Display::position(0, 0);
    cout << "The Philosopher:" << endl;
    chopstick[i] = new Semaphore;

    phil[0] = new Thread(&philosopher, 0,  5, 32);
    phil[1] = new Thread(&philosopher, 1, 10, 44);
    phil[2] = new Thread(&philosopher, 2, 16, 39);
    phil[3] = new Thread(&philosopher, 3, 16, 24);
    phil[4] = new Thread(&philosopher, 4, 10, 20);

    cout << "Philosophers are alive and hungry!"
         << endl;

    Display::position(7, 44);
    cout << '/';
    Display::position(13, 44);
    cout << '\\';
    Display::position(16, 35);
    cout << '|';
    Display::position(13, 27);
    cout << '/';
    Display::position(7, 27);
    cout << '\\';
    Display::position(19, 0);
    cout << "The dinner is served ..." << endl;
    table.unlock();

    for(int i = 0; i < 5; i++) {
        int ret = phil[i]->join();
        table.lock();
        Display::position(20 + i, 0);
        cout << "Philosopher " << i << " ate "
             << ret << " times " << endl;
        table.unlock();
    }

    for(int i = 0; i < 5; i++)
        delete chopstick[i];
    for(int i = 0; i < 5; i++)
        delete phil[i];

    cout << "The end!" << endl;

    return 0;
}
```



```
int main()
{
    table.lock();
    Display::clear();
    Display::position(0, 0);
    cout << "The Philosopher's Dinner:" << endl;

    for(int i = 0; i < 5; i++)
        chopstick[i] = new Semaphore;

    phil[0] = new Thread(&philosopher, 0,  5, 32);
    phil[1] = new Thread(&philosopher, 1, 10, 44);
    phil[2] = new Thread(&philosopher, 2, 16, 39);
    phil[3] = new Thread(&philosopher, 3, 16, 24);
    phil[4] = new Thread(&philosopher, 4, 10, 20);

    cout << "Philosophers are alive and hungry!"
         << endl;

    Display::position(7, 44);
    cout << '/';
    Display::position(13, 44);
    cout << '\\';
    Display::position(16, 35);
    cout << '|';
    Display::position(13, 27);
    cout << '/';
    Display::position(7, 27);
    cout << '\\';
    Display::position(19, 0);
```

delete what you create, even system objects

```
    cout << "The dinner is served ..." << endl;
    table.unlock();

    for(int i = 0; i < 5; i++) {
        int ret = phil[i]->join();
        table.lock();
        Display::position(20 + i, 0);
        cout << "Philosopher " << i << " ate "
             << ret << " times " << endl;
        table.unlock();
    }

    for(int i = 0; i < 5; i++)
        delete chopstick[i];
    for(int i = 0; i < 5; i++)
        delete phil[i];

    cout << "The end!" << endl;

    return 0;
}
```

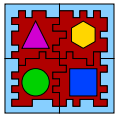


```
int main()
{
    table.lock();
    Display::clear();
    Display::position(0, 0);
    cout << "The Philosopher's Dinner:" << endl;

    for(int i = 0; i < 5; i++)
        chopstick[i] = new Semaphore;

    phil[0] = new Thread(&philosopher, 0,  5, 32);
    phil[1] = new Thread(&philosopher, 1, 10, 44);
    phil[2] = new Thread(&philosopher, 2, 16, 39);
    phil[3] = new Thread(&philosopher, 3, 16, 24);
    phil[4] = new Thread(&philosopher, 4, 10, 20);

    cout << "Philosophers are alive and hungry!"
         << endl;

    cout << '\\';
    Display::position(16, 35);
    cout << '|';
    Display::position(13, 27);
    cout << '/';
    Display::position(7, 27);
    cout << '\\';
    Display::position(19, 0);

    cout << "The dinner is served ..." << endl;
    table.unlock();

    for(int i = 0; i < 5; i++) {
        int ret = phil[i]->join();
        table.lock();
        Display::position(20 + i, 0);
        cout << "Philosopher " << i << " ate "
             << ret << " times " << endl;
        table.unlock();
    }

    for(int i = 0; i < 5; i++)
        delete chopstick[i];
    for(int i = 0; i < 5; i++)
        delete phil[i];

    cout << "The end!" << endl;

    return 0;
}
```

implicit **exit()**
+ globals' **destructors**

```
// EPOS Synchronizer Component Test Program

#include <time.h>
#include <synchronizer.h>
#include <process.h>

using namespace EPOS;

const int iterations = 100;

OStream cout;

const int BUF_SIZE = 16;
char buffer[BUF_SIZE];
Semaphore empty(BUF_SIZE);
Semaphore full(0);

int consumer()
{
    int out = 0;
    for(int i = 0; i < iterations; i++) {
        full.p();
        cout << "C<-" << buffer[out] << "\t";
        out = (out + 1) % BUF_SIZE;
        Alarm::delay(100000);
        empty.v();
    }

    return 0;
}
```

```
int main()
{
    cout << "Producer x Consumer" << endl;

    Thread * cons = new Thread(&consumer);

    // producer
    int in = 0;
    for(int i = 0; i < iterations; i++) {
        empty.p();
        Alarm::delay(100000);
        buffer[in] = 'a' + in;
        cout << "P->" << buffer[in] << "\t";
        in = (in + 1) % BUF_SIZE;
        full.v();
    }

    cons->join();

    cout << "The end!" << endl;

    delete cons;

    return 0;
}
```



Producer x Consumer

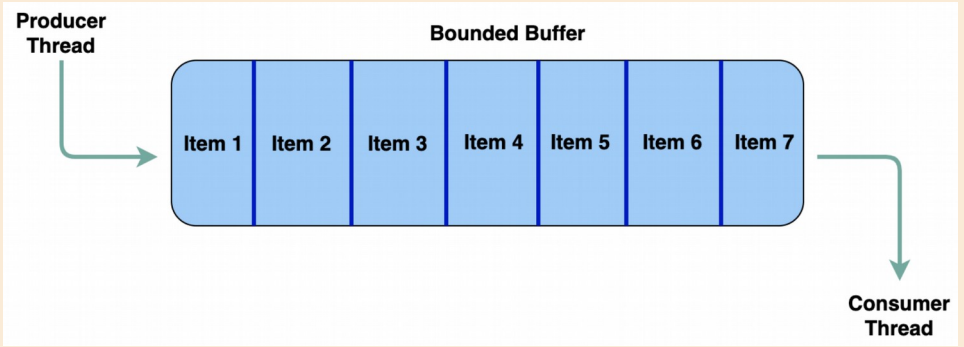
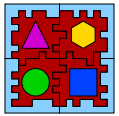
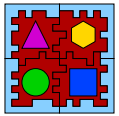


Figure by educative.io

```
Semaphore empty(BUF_SIZE);
Semaphore full(0);

int consumer()
{
    int out = 0;
    for(int i = 0; i < iterations; i++) {
        full.p();
        cout << "C<-" << buffer[out] << "\t";
        out = (out + 1) % BUF_SIZE;
        Alarm::delay(100000);
        empty.v();
    }

    return 0;
}
```

```
int main()
{
    cout << "Producer x Consumer" << endl;

    Thread * cons = new Thread(&consumer);

    // producer
    int in = 0;
    for(int i = 0; i < iterations; i++) {
        empty.p();
        Alarm::delay(100000);
        buffer[in] = 'a' + in;
        cout << "P->" << buffer[in] << "\t";
        in = (in + 1) % BUF_SIZE;
        full.v();
    }

    cons->join();

    cout << "The end!" << endl;

    delete cons;

    return 0;
}
```





```
// EPOS Synchronizer Component Test Program

#include <time.h>
#include <synchronizer.h>
#include <process.h>

using namespace EPOS;

const int iterations = 100;

OStream cout;

const int BUF_SIZE = 16;
char buffer[BUF_SIZE];
Semaphore empty(BUF_SIZE);
Semaphore full(0);

int consumer()
{
    int out = 0;
    for(int i = 0; i < iterations; i++) {
        full.p();
        cout << "C<-" << buffer[out] << "\t";
        out = (out + 1) % BUF_SIZE;
        Alarm::delay(100000);
        empty.v();
    }

    return 0;
}
```

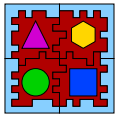
semaphores as atomic resource counters

```
int main()
{
    cout << "Producer x Consumer" << endl;

    Thread * cons = new Thread(&consumer);

    // producer
    int in = 0;
    for(int i = 0; i < iterations; i++) {
        empty.p();
        Alarm::delay(100000);
                                'a' + in;
                                << buffer[in] << "\t";
                                % BUF_SIZE;
                            full.v();
    }

    cons->join();

    cout << "The end!" << endl;

    delete cons;

    return 0;
}
```

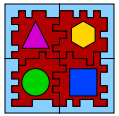
```
// EPOS Synchronizer Component Test Program

#include <time.h>
#include <synchronizer.h>
#include <process.h>

using namespace EPOS;

const int iterations = 100;

OStream cout;

const int BUF_SIZE = 16;
char buffer[BUF_SIZE];
Semaphore empty(BUF_SIZE);
Semaphore full(0);

int consumer()
{
    int out = 0;
    for(int i = 0; i < iterations; i++) {
        full.p();
        cout << "C<-" << buffer[out] << "\t";
        out = (out + 1) % BUF_SIZE;
        Alarm::delay(100000);
        empty.v();
    }

    return 0;
}
```

consumer
consumes from buffer

```
int main()
{
    cout << "Producer x Consumer" << endl;

    Thread * cons = new Thread(&consumer);

    // producer
    int in = 0;
    for(int i = 0; i < iterations; i++) {
        empty.p();
        Alarm::delay(100000);
        buffer[in] = 'a' + in;
        cout << "P->" << buffer[in] << "\t";
        in = (in + 1) % BUF_SIZE;
        full.v();
    }

    << endl;

    delete cons;

    return 0;
}
```



producer produces into buffer



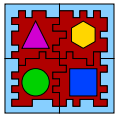
```
st Program
#include <synchronizer.h>
#include <process.h>

using namespace EPOS;

const int iterations = 100;

OStream cout;

const int BUF_SIZE = 16;
char buffer[BUF_SIZE];
Semaphore empty(BUF_SIZE);
Semaphore full(0);

int consumer()
{
    int out = 0;
    for(int i = 0; i < iterations; i++) {
        full.p();
        cout << "C<-" << buffer[out] << "\t";
        out = (out + 1) % BUF_SIZE;
        Alarm::delay(100000);
        empty.v();
    }

    return 0;
}
```

```
int main()
{
    cout << "Producer x Consumer" << endl;

    Thread * cons = new Thread(&consumer);

    // producer
    int in = 0;
    for(int i = 0; i < iterations; i++) {
        empty.p();
        Alarm::delay(100000);
        buffer[in] = 'a' + in;
        cout << "P->" << buffer[in] << "\t";
        in = (in + 1) % BUF_SIZE;
        full.v();
    }

    cons->join();

    cout << "The end!" << endl;

    delete cons;

    return 0;
}
```

```
// EPOS Synchronizer Component Test Program

#include <time.h>
#include <synchronizer.h>
#include <process.h>

using namespace EPOS;

const int iterations = 100;

OStream cout;

const int BUF_SIZE = 16;
char buffer[BUF_SIZE];

int consumer()
{
    int out = 0;
    for(int i = 0; i < iterations; i++) {
        full.p();
        cout << "C<-" << buffer[out] << "\t";
        out = (out + 1) % BUF_SIZE;
        Alarm::delay(100000);
        empty.v();
    }

    return 0;
}
```

join waits for threads to finish

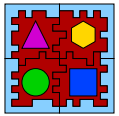
```
int main()
{
    cout << "Producer x Consumer" << endl;

    Thread * cons = new Thread(&consumer);

    // producer
    int in = 0;
    for(int i = 0; i < iterations; i++) {
        empty.p();
        Alarm::delay(100000);
        buffer[in] = 'a' + in;
        cout << "P->" << buffer[in] << "\t";
        in = (in + 1) % BUF_SIZE;
        full.v();
    }

    cons->join();

    cout << "The end!" << endl;

    delete cons;

    return 0;
}
```



```
// EPOS Synchronizer Component Test Program

#include <time.h>
#include <synchronizer.h>
#include <process.h>

using namespace EPOS;

const int iterations = 100;

OStream cout;

const int BUF_SIZE = 16;
char buffer[BUF_SIZE];
Semaphore empty(BUF_SIZE);
Semaphore full(0);

    for(int i = 0; i < iterations; i++) {
        full.p();
        cout << "C<-" << buffer[out] << "\t";
        out = (out + 1) % BUF_SIZE;
        Alarm::delay(100000);
        empty.v();
    }

    return 0;
}
```

delete what you create, even system objects

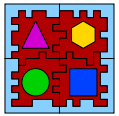
```
int main()
{
    cout << "Producer x Consumer" << endl;

    Thread * cons = new Thread(&consumer);

    // producer
    int in = 0;
    for(int i = 0; i < iterations; i++) {
        empty.p();
        Alarm::delay(100000);
        buffer[in] = 'a' + in;
        cout << "P->" << buffer[in] << "\t";
        in = (in + 1) % BUF_SIZE;
        full.v();
    }

    cons->join();

    cout << "The end!" << endl;

    delete cons;

    return 0;
}
```

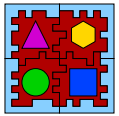


```
// EPOS Synchronizer Component Test Program

#include <time.h>
#include <synchronizer.h>
#include <process.h>

using namespace EPOS;

const int iterations = 100;

OStream cout;

const int BUF_SIZE = 16;
char buffer[BUF_SIZE];
Semaphore empty(BUF_SIZE);
Semaphore full(0);

int consumer()
{
    int out = 0;
    for(int i = 0; i < iterations; i++) {
        full.p();
        ut] << "\t";
        E;
        empty.v();
    }

    return 0;
}
```

implicit **exit()**
+ globals' **destructors**

```
int main()
{
    cout << "Producer x Consumer" << endl;

    Thread * cons = new Thread(&consumer);

    // producer
    int in = 0;
    for(int i = 0; i < iterations; i++) {
        empty.p();
        Alarm::delay(100000);
        buffer[in] = 'a' + in;
        cout << "P->" << buffer[in] << "\t";
        in = (in + 1) % BUF_SIZE;
        full.v();
    }

    cons->join();

    cout << "The end!" << endl;

    delete cons;

    return 0;
}
```

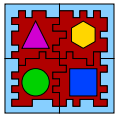




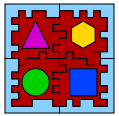
```
// EPOS Thread Component Declarations

#ifndef __process_h
#define __process_h

#include <architecture.h>
#include <utility/queue.h>
#include <utility/handler.h>
#include <machine/common.h>
#include <machine/timer.h>

extern "C" { void __exit(); }

__BEGIN_SYS

class Thread
{
    friend class Init_First;
    friend class Init_System;
    friend class Synchronizer_Common;
    friend class Alarm;
    friend class System;

protected:
    static const bool reboot =
        Traits<System>::reboot;
    static const unsigned int QUANTUM =
        Traits<Thread>::QUANTUM;
    static const unsigned int STACK_SIZE =
        Traits<Application>::STACK_SIZE;

    typedef CPU::Log_Addr Log_Addr;
    typedef CPU::Context Context;

public:
    // Thread State
    enum State { RUNNING, READY, SUSPENDED,
                 WAITING, FINISHING };

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

    // Thread Configuration
    struct Configuration {
        Configuration(const State & s = READY,
                      const Priority & p = NORMAL,
                      unsigned int ss = STACK_SIZE)
        : state(s), priority(p), stack_size(ss) {}

        State state;
        Priority priority;
        unsigned int stack_size;
    };

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

```
// EPOS Thread Component Declarations

#ifndef __process_h
#define __process_h

#include <architecture.h>
#include <utility/queue.h>
#include <utility/handler.h>
#include <machine/common.h>
#include <machine/timer.h>

extern "C" { void __exit(); }

__BEGIN_SYS

class Thread
{
    friend class Init_First;
    friend class Init_System;
    friend class Synchronizer_Common;
    friend class Alarm;
    friend class System;

protected:
    static const bool reboot =
        Traits<System>::reboot;
    static const unsigned int QUANTUM =
        Traits<Thread>::QUANTUM;
    static const unsigned int STACK_SIZE =
        Traits<Application>::STACK_SIZE;
```

header file's #include **guard**

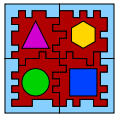


```
public:
    // Thread State
    enum State { RUNNING, READY, SUSPENDED,
                 WAITING, FINISHING };

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

    // Thread Configuration
    struct Configuration {
        Configuration(const State & s = READY,
                      const Priority & p = NORMAL,
                      unsigned int ss = STACK_SIZE)
        : state(s), priority(p), stack_size(ss) {}

        State state;
        Priority priority;
        unsigned int stack_size;
    };

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



```
// EPOS Thread Component Declarations

#ifndef __process_h
#define __process_h

#include <architecture.h>
#include <utility/queue.h>
#include <utility/handler.h>
#include <machine/common.h>
#include <machine/timer.h>

extern "C" { void __exit(); }

__BEGIN_SYS

class Thread
{
    friend class Init_First;
    friend class Init_System;
    friend class Synchronizer_Common;
    friend class Alarm;
    friend class System;

protected:
    static const bool reboot =
        Traits<System>::reboot;
    static const unsigned int QUANTUM =
        Traits<Thread>::QUANTUM;
    static const unsigned int STACK_SIZE =
        Traits<Application>::STACK_SIZE;

    typedef CPU::Log_Addr Log_Addr;
    typedef CPU::Context Context;

public:
    // Thread State
    enum State { RUNNING, READY, SUSPENDED,
                 WAITING, FINISHING };

    // Thread Priority
                                    ty;

        NORMAL = 15,
        LOW = 31
    };

    // Thread Configuration
    struct Configuration {
        Configuration(const State & s = READY,
                      const Priority & p = NORMAL,
                      unsigned int ss = STACK_SIZE)
        : state(s), priority(p), stack_size(ss) {}

        State state;
        Priority priority;
        unsigned int stack_size;
    };

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

helper function with **C linkage**

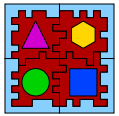


```
// EPOS Thread Component Declarations

#ifndef __process_h
#define __process_h

#include <architecture.h>
#include <utility/queue.h>
#include <utility/handler.h>
#include <machine/common.h>
#include <machine/timer.h>

extern "C" { void __exit(); }

__BEGIN_SYS

class Thread
{
    friend class Init_First;
    friend class Init_System;
    friend class Synchronizer_Common;
    friend class Alarm;
    friend class System;

protected:
    static const bool reboot =
        Traits<System>::reboot;
    static const unsigned int QUANTUM =
        Traits<Thread>::QUANTUM;
    static const unsigned int STACK_SIZE =
        Traits<Application>::STACK_SIZE;

    typedef CPU::Log_Addr Log_Addr;
    typedef CPU::Context Context;

public:
    // Thread State
    enum State { RUNNING, READY, SUSPENDED,
                 WAITING, FINISHING };

    // Thread Priority
    typedef unsigned int Priority;
    enum {
    };

    // Thread Configuration
    struct Configuration {
        Configuration(const State & s = READY,
                      const Priority & p = NORMAL,
                      unsigned int ss = STACK_SIZE)
        : state(s), priority(p), stack_size(ss) {}

        State state;
        Priority priority;
        unsigned int stack_size;
    };

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

namespaces: API, UTIL, SYS

```
// EPOS Thread Component Declarations

#ifndef __process_h
#define __process_h

#include <architecture.h>
#include <utility/queue.h>
#include <utility/handler.h>
#include <machine/common.h>
#include <machine/timer.h>

extern "C" { void __exit(); }

__BEGIN_SYS

class Thread
{
    friend class Init_First;
    friend class Init_System;
    friend class Synchronizer_Common;
    friend class Alarm;
    friend class System;

protected:
    static const bool reboot =
        Traits<System>::reboot;
    static const unsigned int QUANTUM =
        Traits<Thread>::QUANTUM;
    static const unsigned int STACK_SIZE
        Traits<Application>::STACK_SIZE;
```

```
class Class
{
    friend class …. ;

private:
    flag … ;
    CONSTANT … ;
    Type …. ;

public:
    CONSTANT … ;
    Types …. ;

public:
    constructor();
    destructor();
    method();

    static method();

private:
    method();
    static method();

    static void init();

private:
    attribute;

    static attribute;
}
```

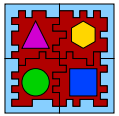


```
SPENDED,
 };

 s = READY,
y & p = NORMAL,
ss = STACK_SIZE)
ack_size(ss) {}

riority> Queue;
```

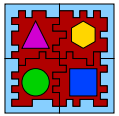
```
// EPOS Thread Component Declarations

#ifndef __process_h
#define __process_h

#include <architecture.h>
#include <utility/queue.h>
#include <utility/handler.h>
#include <machine/common.h>
#include <machine/timer.h>

extern "C" { void __exit(); }

__BEGIN_SYS

class Thread
{
    friend class Init_First;
    friend class Init_System;
    friend class Synchronizer_Common;
    friend class Alarm;
    friend class System;

protected:
    static const bool reboot =
        Traits<System>::reboot;
    static const unsigned int QUANTUM =
        Traits<Thread>::QUANTUM;
    static const unsigned int STACK_SIZE
        Traits<Application>::STACK_SIZE;
```

```
class Class
{
    friend class …. ;

private:
    flag … ;
    CONSTANT … ;
    Type …. ;

public:
    CONSTANT … ;
    Types …. ;

public:
    constructor();
    destructor();
    method();

    static method();

private:
    method();
    static method();

    static void init();

private:
    attribute;

    static attribute;
}
```

INTERFACE

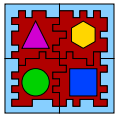


```
SPENDED,
};

s = READY,
y & p = NORMAL,
ss = STACK_SIZE)
ack_size(ss) {}

riority> Queue;
```



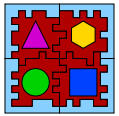
```
// EPOS Thread Component Declarations

#ifndef __process_h
#define __process_h

#include <architecture.h>
#include <utility/queue.h>
#include <utility/handler.h>
#include <machine/common.h>
#include <machine/timer.h>

extern "C" { void __exit(); }

__BEGIN_SYS

class Thread
{
    friend class Init_First;
    friend class Init_System;
    friend class Synchronizer_Common;
    friend class Alarm;
    friend class System;

protected:
    static const bool reboot =
        Traits<System>::reboot;
    static const unsigned int QUANTUM =
        Traits<Thread>::QUANTUM;
    static const unsigned int STACK_SIZE =
        Traits<Application>::STACK_SIZE;

    typedef CPU::Log_Addr Log_Addr;
    typedef CPU::Context Context;

public:
    // Thread State
    enum State { RUNNING, READY, SUSPENDED,
                 WAITING, FINISHING };

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

                                  = READY,
                    const Priority & p = NORMAL,
                    unsigned int ss = STACK_SIZE)
        : state(s), priority(p), stack_size(ss) {}

    State state;
    Priority priority;
    unsigned int stack_size;
};

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

friend classes to prevent encapsulation breaking



```
// EPOS Thread Component Declarations

#ifndef __process_h
#define __process_h

#include <architecture.h>
#include <utility/queue.h>
#include <utility/handler.h>
#include <machine/common.h>
#include <machine/timer.h>

extern "C" { void __exit(); }

__BEGIN_SYS

class Thread
{
    friend class Init_First;
    friend class Init_System;
    friend class Synchronizer_Common;
    friend class Alarm;
    friend class System;

protected:
    static const bool reboot =
        Traits<System>::reboot;
    static const unsigned int QUANTUM =
        Traits<Thread>::QUANTUM;
    static const unsigned int STACK_SIZE =
        Traits<Application>::STACK_SIZE;

    typedef CPU::Log_Addr Log_Addr;
    typedef CPU::Context Context;

public:
    // Thread State
    enum State { RUNNING, READY, SUSPENDED,
                 WAITING, FINISHING };

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

    // Thread Configuration
    struct Configuration {
        Configuration(const State & s = READY,
                      const Priority & p = NORMAL,
                      unsigned int ss = STACK_SIZE)
                                    _size(ss) {}

        Priority priority;
        unsigned int stack_size;
    };

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

static const int instead of #define

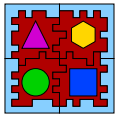


```
// EPOS Thread Component Declarations

#ifndef __process_h
#define __process_h

#include <architecture.h>
#include <utility/queue.h>
#include <utility/handler.h>
#include <machine/common.h>
#include <machine/timer.h>

extern "C" { void __exit(); }

__BEGIN_SYS

class Thread
{
    friend class Init_First;
    friend class Init_System;
    friend class Synchronizer_Common;
    friend class Alarm;
    friend class System;

protected:
    static const bool reboot =
        Traits<System>::reboot;
    static const unsigned int QUANTUM =
        Traits<Thread>::QUANTUM;
    static const unsigned int STACK_SIZE =
        Traits<Application>::STACK_SIZE;

    typedef CPU::Log_Addr Log_Addr;
    typedef CPU::Context Context;

public:
    // Thread State
    enum State { RUNNING, READY, SUSPENDED,
                 WAITING, FINISHING };

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

    // Thread Configuration
    struct Configuration {
        Configuration(const State & s = READY,
                      const Priority & p = NORMAL,
                      unsigned int ss = STACK_SIZE)
        : state(s), priority(p), stack_size(ss) {}

        Sta
        Pri
        uns
    };

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

Traits<> like in the STL (Stepanov)

imports

```
#define __process_h

#include <architecture.h>
#include <utility/queue.h>
#include <utility/handler.h>
#include <machine/common.h>
#include <machine/timer.h>

extern "C" { void __exit(); }

__BEGIN_SYS

class Thread
{
    friend class Init_First;
    friend class Init_System;
    friend class Synchronizer_Common;
    friend class Alarm;
    friend class System;

protected:
    static const bool reboot =
        Traits<System>::reboot;
    static const unsigned int QUANTUM =
        Traits<Thread>::QUANTUM;
    static const unsigned int STACK_SIZE =
        Traits<Application>::STACK_SIZE;
```

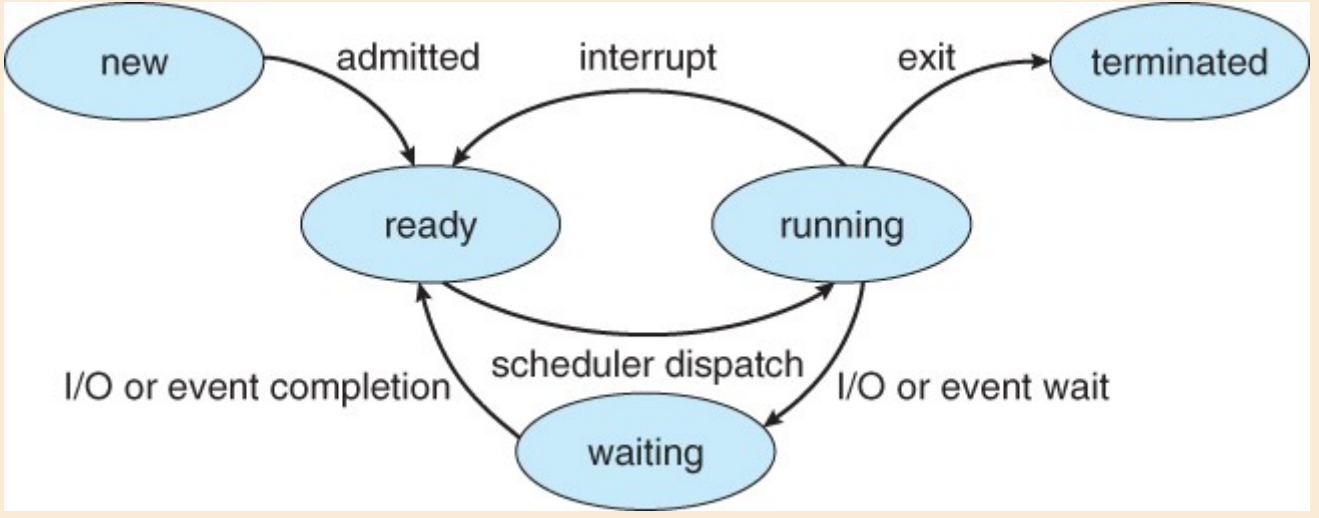


```
    typedef CPU::Log_Addr Log_Addr;
    typedef CPU::Context Context;

public:
    // Thread State
    enum State { RUNNING, READY, SUSPENDED,
                 WAITING, FINISHING };

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

    // Thread Configuration
    struct Configuration {
        Configuration(const State & s = READY,
                      const Priority & p = NORMAL,
                      unsigned int ss = STACK_SIZE)
        : state(s), priority(p), stack_size(ss) {}

        State state;
        Priority priority;
        unsigned int stack_size;
    };

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

```
// EPOS Thread Component Declarations

#ifndef __process_h
#define __process_h

#include <architecture.h>
#include <utility/queue.h>
#include <utility/handler.h>
#include <machine/common.h>
#include <machine/timer.h>

extern "C" { void __exit(); }
```

```
    typedef CPU::Log_Addr Log_Addr;
    typedef CPU::Context Context;

public:
    // Thread State
    enum State { RUNNING, READY, SUSPENDED,
                 WAITING, FINISHING };

    // Thread Priority
    typedef unsigned int Priority;
    enum {
        HIGH = 0,
```

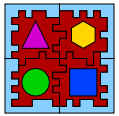
States as in Silberschatz

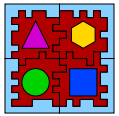



```
uration
tion {
n(const State & s = READY,
  const Priority & p = NORMAL,
  unsigned int ss = STACK_SIZE)
priority(p), stack_size(ss) {}


prity;
 stack_size;


Queue<Thread, Priority> Queue;
```

```
// EPOS Thread Component Declarations

#ifndef __process_h
#define __process_h

#include <architecture.h>
#include <utility/queue.h>

extern "C" { void __exit(); }

__BEGIN_SYS

class Thread
{
    friend class Init_First;
    friend class Init_System;
    friend class Synchronizer_Common;
    friend class Alarm;
    friend class System;

protected:
    static const bool reboot =
        Traits<System>::reboot;
    static const unsigned int QUANTUM =
        Traits<Thread>::QUANTUM;
    static const unsigned int STACK_SIZE =
        Traits<Application>::STACK_SIZE;

    typedef CPU::Log_Addr Log_Addr;
    typedef CPU::Context Context;

public:
    // Thread State
    enum State { RUNNING, READY, SUSPENDED,
                 WAITING, FINISHING };

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

    // Thread Configuration
    struct Configuration {
        Configuration(const State & s = READY,
                      const Priority & p = NORMAL,
                      unsigned int ss = STACK_SIZE)
        : state(s), priority(p), stack_size(ss) {}

        State state;
        Priority priority;
        unsigned int stack_size;
    };

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

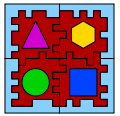
unsigned int + anonymous enum for standard operations





```
// EPOS Thread Component Declarations

#ifndef __process_h
#define __process_h

#include <architecture.h>
#include <utility/queue.h>
#include <utility/handler.h>
#include <machine/common.h>
#include <machine/timer.h>

extern "C" { void __exit(); }

__BEGIN_SYS

class Thread
{
    friend class Init_First;
    friend class Init_System;
    friend class Synchronizer_Common;
    friend class Alarm;
    friend class System;

protected:
    static const bool reboot =
        Traits<System>::reboot;
    static const unsigned int QUANTUM =
        Traits<Thread>::QUANTUM;
    static const unsigned int STACK_SIZE =
        Traits<Thread>::STACK_SIZE;

    typedef CPU::Log_Addr Log_Addr;
    typedef CPU::Context Context;

public:
    // Thread State
    enum State { RUNNING, READY, SUSPENDED,
                 WAITING, FINISHING };

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

    // Thread Configuration
    struct Configuration {
        Configuration(const State & s = READY,
                      const Priority & p = NORMAL,
                      unsigned int ss = STACK_SIZE)
        : state(s), priority(p), stack_size(ss) {}

        State state;
        Priority priority;
        unsigned int stack_size;
    };

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

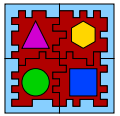
READY queue



```
// EPOS Synchronizer Components

#ifndef __synchronizer_h
#define __synchronizer_h

#include <architecture.h>
#include <utility/handler.h>
#include <process.h>

__BEGIN_SYS

class Synchronizer_Common
{
protected:
    Synchronizer_Common() {}

    // Atomic operations
    bool tsl(volatile bool & lock) {
        return CPU::tsl(lock); }
    int finc(volatile int & number) {
        return CPU::finc(number); }
    int fdec(volatile int & number) {
        return CPU::fdec(number); }

    // Thread operations
    void begin_atomic() { Thread::lock(); }
    void end_atomic() { Thread::unlock(); }

    void sleep() { Thread::yield(); }
    void wakeup() { end_atomic(); }
    void wakeup_all() { end_atomic(); }
};

class Mutex: protected Synchronizer_Common
{
public:
    Mutex();
    ~Mutex();

    void lock();
    void unlock();

private:
    volatile bool _locked;
};

class Semaphore: protected Synchronizer_Common
{
public:
    Semaphore(int v = 1);
    ~Semaphore();

    void p();
    void v();

private:
    volatile int _value;
};

__END_SYS

#endif
```

```
// EPOS Synchronizer Components

#ifndef __synchronizer_h
#define __synchronizer_h

#include <architecture.h>
#include <utility/handler.h>
#include <process.h>

__BEGIN_SYS

class Synchronizer_Common
{
protected:
    Synchronizer_Common() {}

    // Atomic operations
    bool tsl(volatile bool & lock) {
        return CPU::tsl(lock); }
    int finc(volatile int & number) {
        return CPU::finc(number); }
    int fdec(volatile int & number) {
        return CPU::fdec(number); }

    // Thread operations
    void begin_atomic() { Thread::lock(); }
    void end_atomic() { Thread::unlock(); }

    void sleep() { Thread::yield(); }
    void wakeup() { end_atomic(); }
    void wakeup_all() { end_atomic(); }
};
```

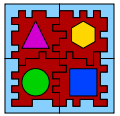
common packages are reused by a family of abstractions
(protected constructor)



```
class Mutex: protected Synchronizer_Common
{
pu
    void unlock();

private:
    volatile bool _locked;
};

class Semaphore: protected Synchronizer_Common
{
public:
    Semaphore(int v = 1);
    ~Semaphore();

    void p();
    void v();

private:
    volatile int _value;
};

__END_SYS

#endif
```

```
// EPOS Synchronizer Components

#ifndef __synchronizer_h
#define __synchronizer_h

#include <architecture.h>
#include <utility/handler.h>
#include <process.h>

__BEGIN_SYS

class Synchronizer_Common
{
protected:
    Synchronizer_Common() {}

    // Atomic operations
    bool tsl(volatile bool & lock) {
        return CPU::tsl(lock); }
    int finc(volatile int & number) {
        return CPU::finc(number); }
    int fdec(volatile int & number) {
        return CPU::fdec(number); }

    // Thread operations
    void begin_atomic() { Thread::lock(); }
    void end_atomic() { Thread::unlock(); }

    void sleep() { Thread::yield(); }
    void wakeup() { end_atomic(); }
    void wakeup_all() { end_atomic(); }
};
```

```
class Mutex: protected Synchronizer_Common
{
public:
    Mutex();
    ~Mutex();

    void lock();
    void unlock();

private:
    volatile bool _locked;
};

class Se
{
public:
    Sema
    ~Semaphore();

    void p();
    void v();

private:
    volatile int _value;
};

__END_SYS

#endif
```

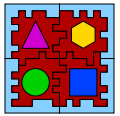


common packages are reused by a family of abstractions
(private inheritance)

```
// EPOS Synchronizer Components

#ifndef __synchronizer_h
#define __synchronizer_h

#include <architecture.h>
#include <utility/handler.h>
#include <process.h>

__BEGIN_SYS

class Synchronizer_Common
{
protected:
    Synchronizer_Common() {}

    // Atomic operations
    bool tsl(volatile bool & lock) {
        return CPU::tsl(lock); }
    int finc(volatile int & number) {
        return CPU::finc(number); }
    int fdec(volatile int & number) {
        return CPU::fdec(number); }

    // Thread operations
    void begin_atomic() { Thread::lock(); }
    void end_atomic() { Thread::unlock(); }

    void sleep() { Thread::yield(); }
    void wakeup() { end_atomic(); }
    void wakeup_all() { end_atomic(); }
};
```

```
class Mutex: protected Synchronizer_Common
{
public:
    Mutex();
    ~Mutex();

    void lock();
    void unlock();

private:
    volatile
};

class Semaphore: protected Synchronizer_Common
{
public:
    Semaphore(int v = 1);
    ~Semaphore();

    void p();
    void v();

private:
    volatile int _value;
};

__END_SYS

#endif
```

atomic operations provided by the architecture



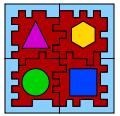
```
// EPOS Synchronizer Components

#ifndef __synchronizer_h
#define __synchronizer_h

#include <architecture.h>
#include <utility/handler.h>
#include <process.h>

__BEGIN_SYS

class Synchronizer_Common
{
protected:
    Synchronizer_Common() {}

    // Atomic operations
    bool tsl(volatile bool & lock) {
        return CPU::tsl(lock); }
    int finc(volatile int & number) {
        return CPU::finc(number); }
    int fdec(volatile int & number) {
        return CPU::fdec(number); }

    // Thread operations
    void begin_atomic() { Thread::lock(); }
    void end_atomic() { Thread::unlock(); }

    void sleep() { Thread::yield(); }
    void wakeup() { end_atomic(); }
    void wakeup_all() { end_atomic(); }
};
```

```
class Mutex: protected Synchronizer_Common
{
public:
    Mutex();
    ~Mutex();

    void lock();
    void unlock();

private:
    volatile bool _locked;
};

class Semaphore: protected Synchronizer_Common
{
public:
    Semaphore(int v = 1);
    ~Semaphore();

    void p();
    void v();

private:
    volatile int _value;
};

__END_SYS

#endif
```

preemption prevention operations provided by thread

```
// EPOS Synchronizer Components

#ifndef __synchronizer_h
#define __synchronizer_h

#include <architecture.h>
#include <utility/handler.h>
#include <process.h>

__BEGIN_SYS

class Synchronizer_Common
{
protected:
    Synchronizer_Common() {}

    // Atomic operations
    bool tsl(volatile bool & lock) {
        return CPU::tsl(lock); }
    int finc(volatile int & number) {
        return CPU::finc(number); }
    int fdec(volatile int & number) {
        return CPU::fdec(number); }

    // Thread operations
    void begin_atomic() { Thread::lock(); }
    void end_atomic() { Thread::unlock(); }

    void sleep() { Thread::yield(); }
    void wakeup() { end_atomic(); }
    void wakeup_all() { end_atomic(); }
};
```

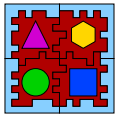
```
class Mutex: protected Synchronizer_Common
{
public:
    Mutex();
    ~Mutex();

    Semaphore(int v = 1);
    ~Semaphore();

    void p();
    void v();

private:
    volatile int _value;
};

__END_SYS

#endif
```



Friend Classes

`Thread::lock()`, which disables preemptive scheduling operations is clearly not part of the interface of class **`Thread`**. It is therefore declared as a private member function and the **`Synchronizar_Common`** class is declared a friend of class **`Thread`**.

classic Mutex with a volatile lock



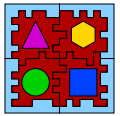
```
#define __synchronizer_h

#include <architecture.h>
#include <utility/handler.h>
#include <process.h>

__BEGIN_SYS

class Synchronizer_Common
{
protected:
    Synchronizer_Common() {}

    // Atomic operations
    bool tsl(volatile bool & lock) {
        return CPU::tsl(lock); }
    int finc(volatile int & number) {
        return CPU::finc(number); }
    int fdec(volatile int & number) {
        return CPU::fdec(number); }

    // Thread operations
    void begin_atomic() { Thread::lock(); }
    void end_atomic() { Thread::unlock(); }

    void sleep() { Thread::yield(); }
    void wakeup() { end_atomic(); }
    void wakeup_all() { end_atomic(); }
};

class Mutex: protected Synchronizer_Common
{
public:
    Mutex();
    ~Mutex();

    void lock();
    void unlock();

private:
    volatile bool _locked;
};

class Semaphore: protected Synchronizer_Common
{
public:
    Semaphore(int v = 1);
    ~Semaphore();

    void p();
    void v();

private:
    volatile int _value;
};

__END_SYS

#endif
```

```
// EPOS Synchronizer Components

#ifndef __synchronizer_h
#define __synchronizer_h

#include <architecture.h>
#include <utility/handler.h>
#include <process.h>
```

Djikstra's Semaphorewith a volatile int

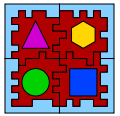
```
{
protected:
    Synchronizer_Common() {}

    // Atomic operations
    bool tsl(volatile bool & lock) {
        return CPU::tsl(lock); }
    int finc(volatile int & number) {
        return CPU::finc(number); }
    int fdec(volatile int & number) {
        return CPU::fdec(number); }

    // Thread operations
    void begin_atomic() { Thread::lock(); }
    void end_atomic() { Thread::unlock(); }

    void sleep() { Thread::yield(); }
    void wakeup() { end_atomic(); }
    void wakeup_all() { end_atomic(); }
};
```

```
class Mutex: protected Synchronizer_Common
{
public:
    Mutex();
    ~Mutex();

    void lock();
    void unlock();

private:
    volatile bool _locked;
};

class Semaphore: protected Synchronizer_Common
{
public:
    Semaphore(int v = 1);
    ~Semaphore();

    void p();
    void v();

private:
    volatile int _value;
};

__END_SYS

#endif
```



```
// EPOS Semaphore Component Implementation

#include <synchronizer.h>

__BEGIN_SYS

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

Semaphore::~Semaphore()
{
    db<Synchronizer>(TRC) << "~Semaphore(this="
                          << this << ")" << endl;
}

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

    begin_atomic();
    while(_value < 1)
        sleep();  // implicit end_atomic()
    fdec(_value);
    end_atomic();
}
```

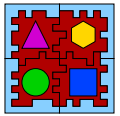
```
void Semaphore::v()
{
    db<Synchronizer>(TRC)
        << "Semaphore::v(this=" << this
        << ",value="
        << _value << ")" << endl;

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

__END_SYS
```



```
// EPOS Semaphore Component Implementation

#include <synchronizer.h>

__BEGIN_SYS

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

Semaphore::~Semaphore()
{
    db<Synchronizer>(TRC) << "~Semaphore(this="
                          << this << ")" << endl;
}

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

    begin_atomic();
    while(_value < 1)
        sleep();  // implicit end_atomic()
    fdec(_value);
    end_atomic();
}
```

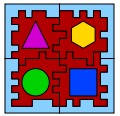
initialization

```
                          << "Semaphore::v(this=" << this
                          << ",value="
                          << _value << ")" << endl;

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

__END_SYS
```



```
// EPOS Semaphore Component Implementation

#include <synchronizer.h>

__BEGIN_SYS

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

Semaphore::~Semaphore()
{
    db<Synchronizer>(TRC) << "~Semaphore(this="
                          << this << ")" << endl;
}

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

    begin_atomic();
    while(_value < 1)
        sleep();  // implicit end_atomic()
    fdec(_value);
    end_atomic();
}
```

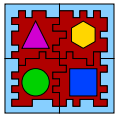
```
void Semaphore::v()
{
    db<Synchronizer>(TRC)
        << "Semaphore::v(this=" << this
        << ",value="
        << _value << ")" << endl;

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

__END_SYS
```



disable preemption via Thread::lock()

```
// EPOS Semaphore Component Implementation

#include <synchronizer.h>

__BEGIN_SYS

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

Semaphore::~Semaphore()
{
    db<Synchronizer>(TRC) << "~Semaphore(this="
                          << this << ")" << endl;
}

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

    begin_atomic();
    while(_value < 1)
        sleep();  // implicit end_atomic()
    fdec(_value);
    end_atomic();
}
```

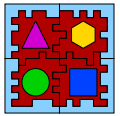
```
void Semaphore::v()
{
    db<Synchronizer>(TRC)
        << "Semaphore::v(this=" << this
        << ",value="
        << _value << ")" << endl;

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

__END_SYS
```



atomic decrement

```
// EPOS Semaphore Component Implementation

#include <synchronizer.h>

__BEGIN_SYS

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

Semaphore::~Semaphore()
{
    db<Synchronizer>(TRC) << "~Semaphore(this="
                          << this << ")" << endl;
}

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

    begin_atomic();
    while(_value < 1)
        sleep();  // implicit end_atomic()
    fdec(_value);
    end_atomic();
}
```

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

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

__END_SYS
```

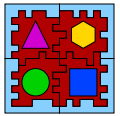


```
void sleep() { Thread::yield(); }
```

```
// EPOS Semaphore Component Implementation

#include <synchronizer.h>

__BEGIN_SYS

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

Semaphore::~Semaphore()
{
    db<Synchronizer>(TRC) << "~Semaphore(this="
                          << this << ")" << endl;
}

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

    begin_atomic();
    while(_value < 1)
        sleep();  // implicit end_atomic()
    fdec(_value);
    end_atomic();
}
```

atomic increment

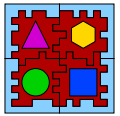
```
void Semaphore::v()
{
    db<Synchronizer>(TRC)
        << "Semaphore::v(this=" << this
        << ",value="
        << _value << ")" << endl;

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

__END_SYS
```

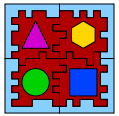


```
// EPOS Semaphore Component Implementation

#include <synchronizer.h>

__BEGIN_SYS

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

Semaphore::~Semaphore()
{
    db<Synchronizer>(TRC) << "~Semaphore(this="
                          << this << ")" << endl;
}

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

    begin_atomic();
    while(_value < 1)
        sleep();  // implicit end_atomic()
    fdec(_value);
    end_atomic();
}
```

```
void wakeup() { end_atomic(); }
```

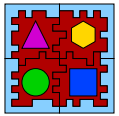
```
void Semaphore::v()
{
    db<Synchronizer>(TRC)
        << "Semaphore::v(this=" << this
        << ",value="
        << _value << ")" << endl;

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

__END_SYS
```





```
// EPOS Mutex Component Implementation

#include <synchronizer.h>

__BEGIN_SYS

Mutex::Mutex(): _locked(false)
{
    db<Synchronizer>(TRC) << "Mutex() => "
                          << this << endl;
}

Mutex::~Mutex()
{
    db<Synchronizer>(TRC) << "~Mutex(this="
                          << this << ")" << endl;
}

void Mutex::lock()
{
    db<Synchronizer>(TRC) << "Mutex::lock(this="
                          << this << ")" << endl;

    begin_atomic();
    if(tsl(_locked))
        while(tsl(_locked))
            sleep(); // implicit end_atomic()
    else
        end_atomic();
}

void Mutex::unlock()
{
    db<Synchronizer>(TRC)
        << "Mutex::unlock(this=" << this
        << ")" << endl;

    begin_atomic();
    _locked = false;
    wakeup(); // implicit end_atomic()
}

__END_SYS
```

initialization

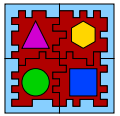


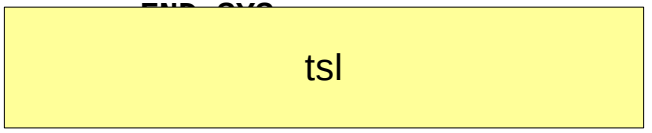
```
// EPOS Mutex Component Implementation

#include <synchronizer.h>

__BEGIN_SYS

Mutex::Mutex(): _locked(false)
{
    db<Synchronizer>(TRC) << "Mutex() => "
                          << this << endl;
}

Mutex::~Mutex()
{
    db<Synchronizer>(TRC) << "~Mutex(this="
                          << this << ")" << endl;
}

void Mutex::lock()
{
    db<Synchronizer>(TRC) << "Mutex::lock(this="
                          << this << ")" << endl;

    begin_atomic();
    if(tsl(_locked))
        while(tsl(_locked))
            sleep(); // implicit end_atomic()
    else
        end_atomic();
}
```

```
                          << "Mutex::unlock(this=" << this
                          << ")" << endl;

    begin_atomic();
    _locked = false;
    wakeup(); // implicit end_atomic()
}

__END_SYS
```

```
// EPOS Mutex Component Implementation

#include <synchronizer.h>

__BEGIN_SYS

Mutex::Mutex(): _locked(false)
{
    db<Synchronizer>(TRC) << "Mutex() => "
                          << this << endl;
}

Mutex::~Mutex()
{
    db<Synchronizer>(TRC) << "~Mutex(this="
                          << this << ")" << endl;
}

void Mutex::lock()
{
    db<Synchronizer>(TRC) << "Mutex::lock(this="
                          << this << ")" << endl;

    begin_atomic();
    if(tsl(_locked))
        while(tsl(_locked))
            sleep(); // implicit end_atomic()
    else
        end_atomic();
}
```

```
void Mutex::unlock()
{
    db<Synchronizer>(TRC)
        << "Mutex::unlock(this=" << this
        << ")" << endl;

    begin_atomic();
    _locked = false;
    wakeup(); // implicit end_atomic()
}
```

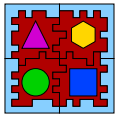




```
// EPOS Mutex Component Implementation

#include <synchronizer.h>

__BEGIN_SYS

Mutex::Mutex(): _locked(false)
{
    db<Synchronizer>(TRC) << "Mutex() => "
                          << this << endl;
}

Mutex::~Mutex()
{
    db<Synchronizer>(TRC) << "~Mutex(this="
                          << this << ")" << endl;
}

void Mutex::lock()
{
    db<Synchronizer>(TRC) << "Mutex::lock(this="
                          << this << ")" << endl;

    begin_atomic();
    if(tsl(_locked))
        while(tsl(_locked))
            sleep(); // implicit end_atomic()
    else
        end_atomic();
}
```

```
void sleep() { Thread::yield(); }
```

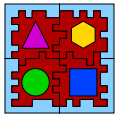
```
void Mutex::unlock()
{
    db<Synchronizer>(TRC)
        << "Mutex::unlock(this=" << this
        << ")" << endl;

    begin_atomic();
    _locked = false;
    wakeup(); // implicit end_atomic()
}

__END_SYS
```



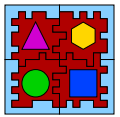
```
// EPOS Mutex Component Implementation

#include <synchronizer.h>

__BEGIN_SYS

Mutex::Mutex(): _locked(false)
{
    db<Synchronizer>(TRC) << "Mutex() =>
                          << this << endl;
}

Mutex::~Mutex()
{
    db<Synchronizer>(TRC) << "~Mutex(this="
                          << this << ")" << endl;
}

void Mutex::lock()
{
    db<Synchronizer>(TRC) << "Mutex::lock(this="
                          << this << ")" << endl;

    begin_atomic();
    if(tsl(_locked))
        while(tsl(_locked))
            sleep(); // implicit end_atomic()
    else
        end_atomic();
}
```

```
void wakeup() { end_atomic(); }
```

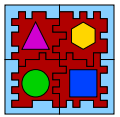
```
void Mutex::unlock()
{
    db<Synchronizer>(TRC)
        << "Mutex::unlock(this=" << this
        << ")" << endl;

    begin_atomic();
    _locked = false;
    wakeup(); // implicit end_atomic()
}

__END_SYS
```



```
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(); // implicit unlock()

    unlock();
}

void Thread::dispatch(Thread * prev, Thread * next)
{
    if(prev != next) {
        assert(prev->_state != RUNNING);
        assert(next->_state == RUNNING);

        db<Thread>(TRC) << "Thread::dispatch(prev="
                        << prev << ",next="
                        << next << ")" << endl;

        db<Thread>(INF) << "prev={" << prev
                        << ",ctx="
                        << *prev→_context
                        << "}" << endl;

        db<Thread>(INF) << "next={" << next
                        << ",ctx=" << *next→_context
                        << "}" << endl;

        CPU::switch_context(&prev->_context,
                            next->_context);
    }

    unlock();
}

int Thread::idle()
{
    db<Thread>(TRC) << "Thread::idle()" << endl;

    db<Thread>(INF)
    << "There are no runnable threads at the moment!"
    << endl;
    db<Thread>(INF) << "Halting the CPU ..." << endl;

    unlock();
    CPU::halt();

    return 0;
}
```



yields the CPU by moving running into READY and removing its head



```
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(); // implicit unlock()

    unlock();
}

void Thread::dispatch(Thread * prev, Thread * next)
{
    if(prev != next) {
        assert(prev->_state != RUNNING);
        assert(next->_state == RUNNING);

        db<Thread>(TRC) << "Thread::dispatch(prev="
                        << prev << ",next="
                        << next << ")" << endl;
```

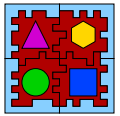
```
' << prev

context
endl;
```

```
        db<Thread>(INF) << "next={" << next
                        << ",ctx=" << *next→_context
                        << "}" << endl;

        CPU::switch_context(&prev->_context,
                            next->_context);
    }

    unlock();
}

int Thread::idle()
{
    db<Thread>(TRC) << "Thread::idle()" << endl;

    db<Thread>(INF)
    << "There are no runnable threads at the moment!"
    << endl;
    db<Thread>(INF) << "Halting the CPU ..." << endl;

    unlock();
    CPU::halt();

    return 0;
}
```

```
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(); // implicit unlock()

    unlock();
}

void Thread::dispatch(Thread * prev, Thread * next)
{
    if(prev != next) {
        assert(prev->_state != RUNNING);
        assert(next->_state == RUNNING);

        db<Thread>(TRC) << "Thread::dispatch(prev="
                        << prev << ",next="
                        << next << ")" << endl;
```

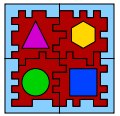
dispatches the new running

```
        db<Thread>(INF) << "prev={" << prev
                        << ",ctx="
                        << *prev->_context
                        << "}" << endl;
                        < "next={" << next
                        < ",ctx=" << *next->_context
                        << "}" << endl;

        CPU::switch_context(&prev->_context,
                            next->_context);
    }

    unlock();
}

int Thread::idle()
{
    db<Thread>(TRC) << "Thread::idle()" << endl;

    db<Thread>(INF)
    << "There are no runnable threads at the moment!"
    << endl;
    db<Thread>(INF) << "Halting the CPU ..." << endl;

    unlock();
    CPU::halt();

    return 0;
}
```



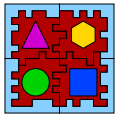
```
void Thread::yield()
{
    lock();

    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(); // implicit unlock()

    unlock();
}

void Thread::dispatch(Thread * prev, Thread * next)
{
    if(prev != next) {
        assert(prev->_state != RUNNING);
        assert(next->_state == RUNNING);

        db<Thread>(TRC) << "Thread::dispatch(prev="
                        << prev << ",next="
                        << next << ")" << endl;

        db<Thread>(INF) << "prev={" << prev
                        << ",ctx="
                        << *prev->_context
                        << "}" << endl;

        db<Thread>(INF) << "next={" << next
                        << ",ctx=" << *next->_context
                        << "}" << endl;

        CPU::switch_context(&prev->_context,
                            next->_context);
    }

    unlock();
}

int Thread::idle()
{
    db<Thread>(TRC) << "Thread::idle()" << endl;

    db<Thread>(INF)
    << "There are no runnable threads at the moment!"
    << endl;
    db<Thread>(INF) << "Halting the CPU ..." << endl;

    unlock();
    CPU::halt();

    return 0;
}
```

or goes into idle if READY is empty



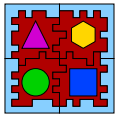
```
void Thread::yield()
{
    lock();

    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(); // implicit unlock()

    unlock();
}

void Thread::dispatch(Thread * prev, Thread * next)
{
    if(prev != next) {
        assert(prev->_state != RUNNING);
        assert(next->_state == RUNNING);

        db<Thread>(TRC) << "Thread::dispatch(prev="
                        << prev << ",next="
                        << next << ")" << endl;

        db<Thread>(INF) << "prev={" << prev
                        << ",ctx="
                        << *prev→_context
                        << "}" << endl;

        db<Thread>(INF) << "next={" << next
                        << ",ctx=" << *next→_context
                        << "}" << endl;

        CPU::switch_context(&prev->_context,
                            next->_context);
    }

    unlock();
}

    db<Thread>(TRC) << "Thread::idle()" << endl;

    db<Thread>(INF)
    << "There are no runnable threads at the moment!"
    << endl;
    db<Thread>(INF) << "Halting the CPU ..." << endl;

    unlock();
    CPU::halt();

    return 0;
}
```

assert states are correct (see #assert definition)



```
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(); // implicit unlock()

    unlock();
}

void Thread::dispatch(Thread * prev, Thread * next)
{
    if(prev != next) {
        assert(prev->_state != RUNNING);
        assert(next->_state == RUNNING);

        db<Thread>(TRC) << "Thread::dispatch(prev="
                        << prev << ",next="
                        << next << ")" << endl;

        db<Thread>(INF) << "prev={" << prev
                        << ",ctx="
                        << *prev->_context
                        << "}" << endl;

        db<Thread>(INF) << "next={" << next
                        << ",ctx=" << *next->_context
                        << "}" << endl;

        CPU::switch_context(&prev->_context,
                            next->_context);
    }

    unlock();
}

int Thread::idle()
{
    db<Thread>(TRC) << "Thread::idle()" << endl;

    db<Thread>(INF)
    << "There are no runnable threads at the moment!"
    << endl;
    db<Thread>(INF) << "Halting the CPU ..." << endl;

    unlock();
    CPU::halt();

    return 0;
}
```

fine-grain debug control
<Class>(LEVEL)



```
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;
```

CPU::halt()

Most CPUs support one or more low-power modes, including a mode in which only registers are preserved and the interrupt signal is monitored. A halted CPU returns to normal operation whenever an interrupt happens. Some CPU support selecting which interrupts are enabled during halt. Returning to normal operation usually takes several CPU cycles, but halting saves a lot of energy.

```
        db<Thread>(INF) << "prev={" << prev
                        << ",ctx="
                        << *prev→_context
                        << "}" << endl;

        db<Thread>(INF) << "next={" << next
                        << ",ctx=" << *next→_context
                        << "}" << endl;

        CPU::switch_context(&prev->_context,
                            next->_context);
    }

    unlock();
}

int Thread::idle()
{
    db<Thread>(TRC) << "Thread::idle()" << endl;

    db<Thread>(INF)
    << "There are no runnable threads at the moment!"
    << endl;
    db<Thread>(INF) << "Halting the CPU ..." << endl;

    unlock();
    CPU::halt();

    return 0;
}
```