Concurrency in a Nutshell

Dirk Vermeir

http://tinf2.vub.ac.be/~dvermeir
Vakgroep Informatica
Vrije Universiteit Brussel, VUB

December 10, 2009

Multiprocessing

$\boxed{\text{Process = Running Program}}$

- In a multiprocessing system, several processes may be executing at the same time (executing the same or different programs) on behalf of one or more users.
- Such concurrency may be
  - real, e.g. if there is more than 1 CPU in the system, or
  - perceived, e.g. when the OS allocates the CPU to the different processes in turn (e.g. "round robin" using a time slice of a few milliseconds).
- Check by running the ps −efl command on Linux.

Processes

- Each process is independent, with its own address space, program counter etc.
- The OS should provice IPC (Inter-Process Communication) facilities for processes to communicate.
- …

IPC Examples

- Unix pipes feed the output of one process to the input of another: `who | wc –l` creates two concurrently running processes executing who and wc, the output of who serves as input for wc –l,
- A process can wait for another process to finish.
- See `manuals/uintro/uintro.html` (from home page) for more info.

Threads

A thread is like a process: many threads execute concurrently, but

- a process can consist of many threads,
- all threads of a process share its address space (e.g. global data),
- each thread has its own program counter (and stack),
- since threads belonging to the same process share its address space, threads can exchange data using e.g. shared global variables,
- the main thread is started by the OS (function `::main(int, char**)` in C++),
- threads may start other threads.

The dvthread library

```
class Thread {
  public:
    Thread(bool del_at_end=false); // Does not start thread.
    virtual ~Thread(); // Only after main() finished!
    // Start executing main.
    int start() throw (std::runtime_error);

    // Function executed by thread.
    virtual int main() throw ();

    // Wait for this thread to finish.
    int join() throw ();
    // alternative for join: you *must* do one or the other
    Thread& detach();

    // Who am i? Only works for Thread objects!
    static Thread* self() throw (std::runtime_error);
};
```

Typical Usage

```
class MyThread: public Thread {
  public:
    MyThread(..); // Constructor may have parameters
                  // with specific info for this thread.
    ..
    int main() throw () { /* code to be executed by thread */ }
  private: .. // e.g. data to be used by MyThread
}

MyThread t1(..);
AnotherThread t2(..);
..
t2.start(); // Start t2, it will execute t2.main() once, then die.
t1.start(); // Start t1, it will execute t1.main() once, then die.
.. // do lots of other stuff
t1.join(); // Main thread waits for t1 to finish.
t2.join(); // Main thread waits for t2 to finish.
```

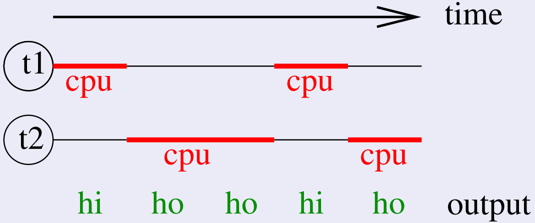
Example

```
#include <dvthread/thread.h>

class MyThread: public Dv::Thread::Thread {
  public:
    MyThread(const std::string& msg): msg_(msg) {}
    int main() throw () {
      for (int i=0; i<5; ++i) {
        sleep(1); std::cout << msg_ << std::endl;
      }
    }
  private:
    std::string msg_;
};

int
main() {
  MyThread t1("hi"); MyThread t2("ho");
  t2.start(); t1.start();
  t1.join(); t2.join(); // wait for threads to finish
}
```

Example Ouput

```
hi
ho
ho
hi
...
```

On a single-CPU system

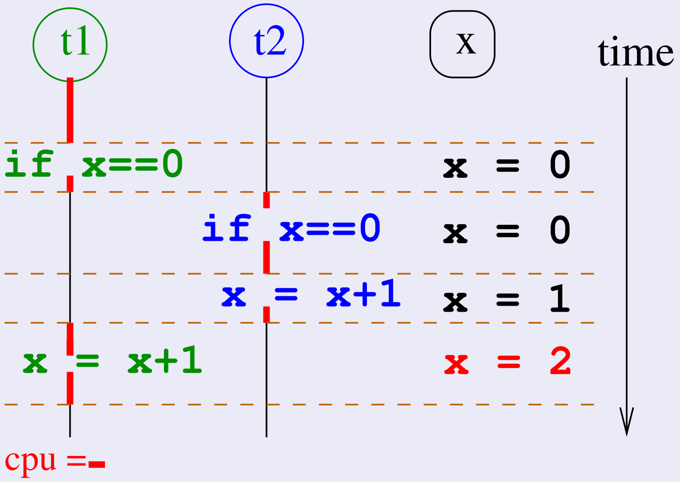


Example Problems with Shared Data

```
class MyThread {
  public: ..
    int main() throw () {
      if (x == 0) // Increment x if it is 0.
        x = x + 1;
    }
};

int x(0); // Shared data that can be accessed by all threads.
// (Think of x as the availability of a seat on a flight.)
MyThread t1; // E.g. travel agent 1.
MyThread t2; // E.g. travel agent 2.
t1.start(); t2.start();
t1.join(); t2.join();
cout << x << endl;
```

Problem Execution

Example



Critical Section

Only one thread should be able to execute a critical section of code at any one time.

```
class MyThread {
  public: ..
    int main() throw () {
      ..
      // START CRITICAL SECTION
      if (x == 0) // Increment x if it is 0.
        x = x + 1;
      // END CRITICAL SECTION
      ..
    }
};
```

Semaphores

A semaphore can be thought of as a non-negative integer variable s with two operations:

semaphore operations

- **P(s)** performs an atomic (uninterruptible) decrement: $--$s but only if s is nonzero. If s == 0, P(s) waits until it is nonzero and then does the substraction.
- **V(s)** performs an atomic (uninterruptible) increment: ++s

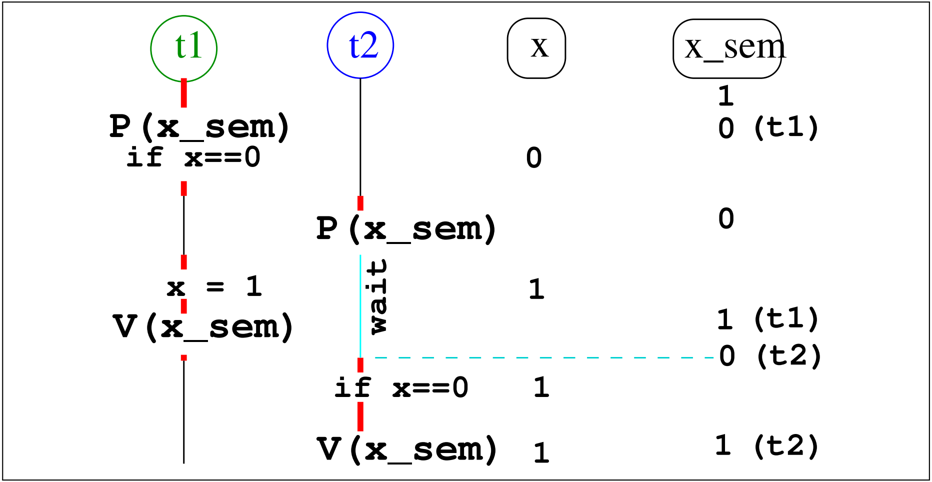
no busy wait

If a thread tries to do P(s) while s == 0, the thread will be put to sleep by the OS. It will be woken up when s > 0.

Semaphores Protect Critical Sections

```
// A mutual exclusion semaphore is initialized to 1, it
// can be used to protect a critical section so that
// only one thread can execute the section at any one time.
Semaphore x_sem(1); // Mutex semaphore.

class MyThread {
  public: ..
    int main() throw () {
      P(x_sem);
      // START CRITICAL SECTION
      if (x == 0) // Increment x if it is 0.
        x = x + 1;
      // END CRITICAL SECTION
      V(x_sem);
    }
};
```

Example Execution with Semaphore



Semaphore Implementation

- Semaphores are implemented by the operating system (OS).
- Each semaphore is associated with a queue of blocked threads that are waiting for the semaphore (which is 0) to become positive.
- A blocked thread does not get the CPU until it becomes "runnable".
- Runnable threads get the CPU, every now and then.

Semaphore Implementation Pseudocode

```
std::map<Semaphore,QueueOfThread> queues;

P(Semaphore s) { // OS code, cannot be interrupted
  if (s == 0)
    append thread to queues[s];
  else
    --s;
}

V(Semaphore s) { // OS code, cannot be interrupted
  if (queue[s].size() > 0)
    make thread in head of queues[s] runnable
  else
    ++s;
}
```

Monitor

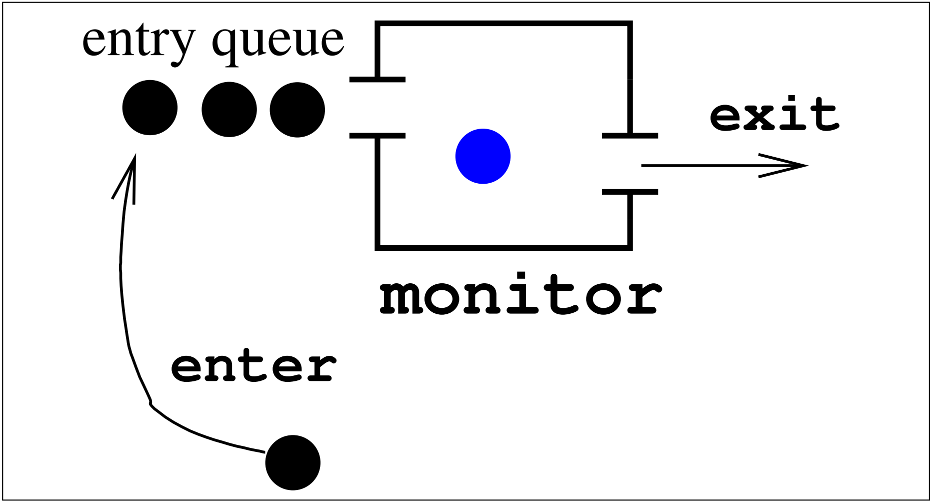
- Semaphores are rather low level.
- A Monitor is more powerful (but can be implemented using semaphores).
- A simple Monitor is like a critical section: only one thread can be in a monitor at any one time.

Monitor Operations

m.enter() Enter m if not occupied, otherwise wait until m is empty, then enter.

m.exit() Leave the monitor m.

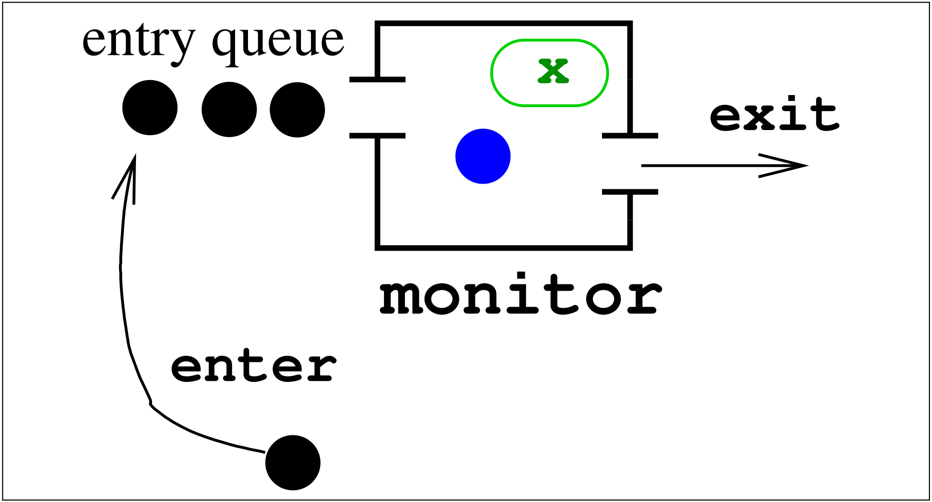
A Simple Monitor



Monitors in dvthread

```
class Monitor {
  public:
    // Constructor; n_conditions will be explained further on.
    Monitor(const std::string& name, size_t n_conditions);

    // Enter the monitor, wait if monitor is occupied.
    void enter() throw (std::runtime_error);

    // Exit the monitor.
    void exit() throw (std::runtime_error);
};
```

Monitors to Protect Data



Protecting Data Using dvthread

```
class Seat: private Monitor {
  public:
    Seat(): Monitor("Seat"), x_(0) {}
    bool reserve() {
      bool ok(false);
      enter(); // enter this monitor
      if (( ok = (x == 0) ))
        ++x;
      exit(); // exit this monitor
      return ok;
    }
  private:
    int x_; // == 0 if seat is still free
};
```

Protecting Data using dvthread

```
class TravelAgent {
  public:
    int main() throw () { .. seat.reserve() .. }
};

Seat seat;
TravelAgent a1;
TravelAgent a2;
a1.start(); a2.start();
a1.join(); a2.join();
```

More Convenience with dvthread

Locks

```
class Lock {
public: // Constructor enters m, destructor exits.
  Lock(Monitor& m): m_(m) { m_.enter(); }
  ~Lock() { m_.exit(); }
private:
  Monitor& m_;
};
```

```
class Seat: private Monitor {
  public:
    Seat(): Monitor("Seat"), x_(0) {}
    bool reserve() {
      Lock lock(*this); // Enter *this, exit when destroyed
      return ( x == 0 ? ++x, true : false);
    }
  private:
    int x_; // == 0 if seat is still free
};
```

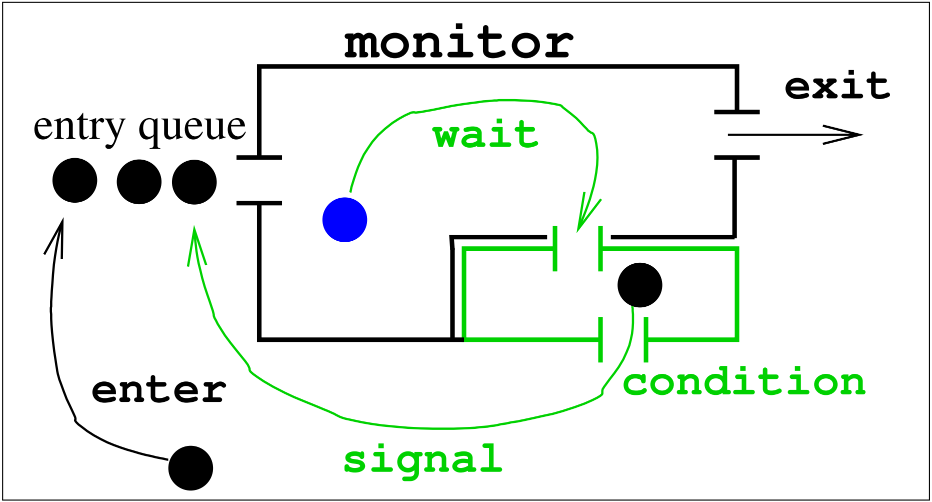
Even More Convenience with dvthread

```
// Doubtful advantage, really.
#define SYNCHRONIZED Lock lock(*this);

class Seat: private Monitor {
  public:
    Seat(): Monitor("Seat"), x_(0) {}
    bool reserve() { SYNCHRONIZED
      return ( x == 0 ? ++x, true : false);
    }
  private:
    int x_; // == 0 if seat is still free
};
```

Monitors with Conditions

- A Monitor may have a number of associated conditions.
- The thread occupying the monitor may
  - wait(c): on a condition c, causing the thread to leave the monitor for a special condition-queue associated with c.
  - signal(c): on a condition c, causing a thread in the queue of c (if any) to be transferred to the head of the normal entry-queue of the monitor.
  - wait(c,t): on a condition c with timeout t. It is like wait(c) but if the thread is not signaled within t millisecs, it will automatically rejoin the head of the entry queue.

Monitors with Conditions



Monitors with Conditions in dvthread

```
class Monitor {
  public:
    // Monitor has n_cond conditions 0 .. n_cond-1
    Monitor(const std::string& name, size_t n_cond);
    // Enter the monitor, wait if monitor is occupied.
    void enter() throw (std::runtime_error);
    // Exit the monitor.
    void exit() throw (std::runtime_error);

    // Wait for a condition.
    void wait(size_t condition) throw (std::runtime_error)
    // Wait for a condition for at most timeout millisecs.
    void wait(size_t condition, size_t timeout) throw (std::runti
    // Signal a condition.
    void signal(size_t condition) const throw (std::runtime_error
};
```

Example: Readers and Writers

Two kinds of concurrent threads acces a shared buffer with a limited capacity.

- Writer threads put items in the buffer.
- Reader threads retrieve items in the buffer.

Example

Interface to (multiple) printer spooling system. Writers put files to be printed. Readers are printer controllers that take files from the pool and actually print them.

Readers and Writers – Buffer 1/2

```
// A buffer with controlled access.
class Buffer: private Dv::Thread::Monitor {
  public:
    // A buffer is a Monitor with 2 conditions.
    Buffer() throw (std::runtime_error): Monitor("buffer",2), n_it
    // Names for conditions.
    enum {
      OK_TO_GET = 0, // Signal if buffer not empty.
      OK_TO_PUT = 1  // Signal if buffer not full.
    };

    void put(int i) throw (std::runtime_error) { SYNCHRONIZED
      while (n_items_==MAX)
        if (! wait(OK_TO_PUT,2000) ) // Wait at most 2 secs.
          throw std::runtime_error("Buffer::put() timed out");
      data_[n_items_++] = i; // Actually put the item.
      signal(OK_TO_GET);
    }
```

Readers and Writers – Buffer 2/2

```
  int get() throw (std::runtime_error) { SYNCHRONIZED
    while (n_items_==0)
      if (!wait(OK_TO_GET, 2000))
        throw std::runtime_error("Buffer::get() timed out");
    int tmp = data_[--n_items_];
    signal(OK_TO_PUT);
    return tmp;
  }
  private:
    enum { MAX = 3 }; // Buffer capacity.
    int n_items_; // Number of items in the buffer.
    int data_[MAX]; // Actual store for items.
};
```

Readers and Writers – Reader

```
// A reader tries to retrieve items from a buffer.
class Reader: public Dv::Thread::Thread {
  public:
    // N is the number of items to retrieve.
    Reader(Buffer& buf, size_t n): Thread(), buffer_(buf), n_(n) {
    virtual int main() throw () {
      try {
        for (unsigned int i=0;(i<n_);++i)
          buffer_.get();
      }
      catch (exception& e) {
        cerr << e.what() << endl;
      }
    }
   // Wait for this thread to finish before detroying it.
   ~Reader() { join(); }
  private:
   Buffer&  buffer_; // From where items will be retrieved.
   size_t  n_; // Number of items to retrieve.
};
```

Readers and Writers – Writer

```
class Writer: public Dv::Thread::Thread {
  public:
    // N is the number of items to put.
    Writer(Buffer& buf, unsigned int n): Thread(), buffer_(buf), n
    int main() throw () {
      try {
        for (unsigned int i=0; (i<n_);++i)
          buffer_.put(i);
      }
      catch (std::exception& e) {
        std::cerr << e.what() << std::endl;
      }
    }
    // Join this thread before destroying it.
    ~Writer() { join(); }
  private:
    Buffer& buffer_; // To which items will be written.
    unsigned int n_; // Number of items to write.
};
```

Readers and Writers – Sample Main

```
int
main(int,char**) {
  try {
    Buffer      buf; // buffer containing items
    Reader      r(buf,10); // takes items from mbuffer
    Writer      w(buf,15); // writes items to buffer

    r.start();
    w.start();
    // Destructors of r, w will join() them.
  }
  catch (exception& e) {
    cerr << e.what() << endl;
    return 1;
  }
  return 0;
}
```