Concurrency

Comp314
March 12, 2009
Devin Grady

Let's start at the beginning

- Concurrency is: Doing more than one task at the same time (logically – not always physically).
- 2 main reasons to want this:
  - Some subset of faster tasks have to wait for some subset of slow tasks to complete (responsiveness)
  - Entire problem is so large that it is too slow on a single processor (performance)

Responsiveness examples

- Networking
  - 1 execution thread (ET) per connection.
  - If one thread is really slow and has a lot to do, it won't cause unfair delays in the other threads.
- GUI
  - Event based programming (only 1 actual ET)
  - Register a set of events, wait for them to happen.
  - What happens if an event handler wants to send a 100MiB file to Tunisia over packet shortwave radio?

Performance examples

- These are usually more obvious:
  - Large matrix operations
  - Game tree exploration
  - Physical simulations
    - Protein folding
    - Combustion modeling
- ...

Event compression

- Screen refresh rate?
- Mouse update rate?
- Naïve program:

```
Mouse_Event_Handler(float x,
float y, float z)
{
   SetCursorPosition(x,y,z);
   RedrawScreen();
}
```

Wait. 1ET concurrency?

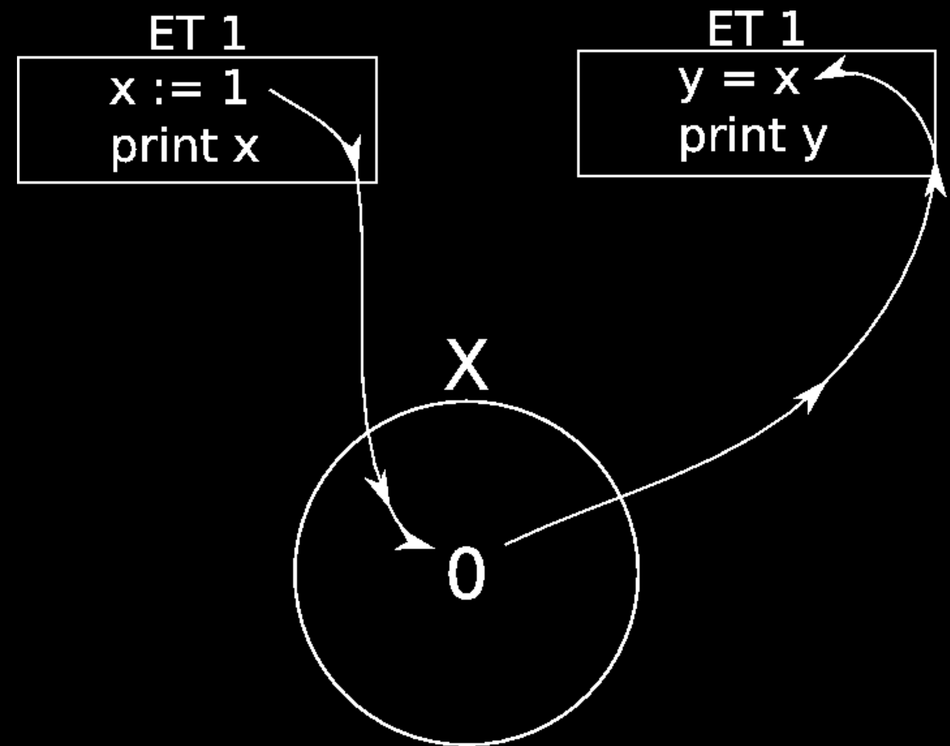
- Well, yes. It makes sense in the GUI / responsiveness cases even on 1 ET.
- In addition, it's easy to program.
- Basically, if that's all you need, then it can provide the benefits without the hassle.
- These days, everyone also wants to “leverage the multi-core phenomenon adroitly” so they use multiple ET's.
- Historical: Netscape 1.0 had event dispatch.

OK, so we want multiple ET's.

- There are many concurrency models.
- That's for later.
- We're just going to concentrate on the fork-join model.
  - fork() returns true/false indicating child/parent.
  - join() waits for threads to come back together.

Why is this hard?

- DATA CONSISTENCY!
- One thread reads, one thread writes.
  - What is the value that gets read?
- Insertion/Deletion in linked-lists.
  - What's the issue here?
- ...



Preemption

- Can have preemptive and non-preemptive concurrency.
- Non-preemptive requires an explicit yield() call.
  - Solves data consistency issues
  - Can't actually run the ET's in parallel on multiple processors though.
  - Other ET's can starve if not enough yields.
- Preemptive is timer-based
  - We can run on multiple processors.
  - No yield() calls but has the consistency issues.

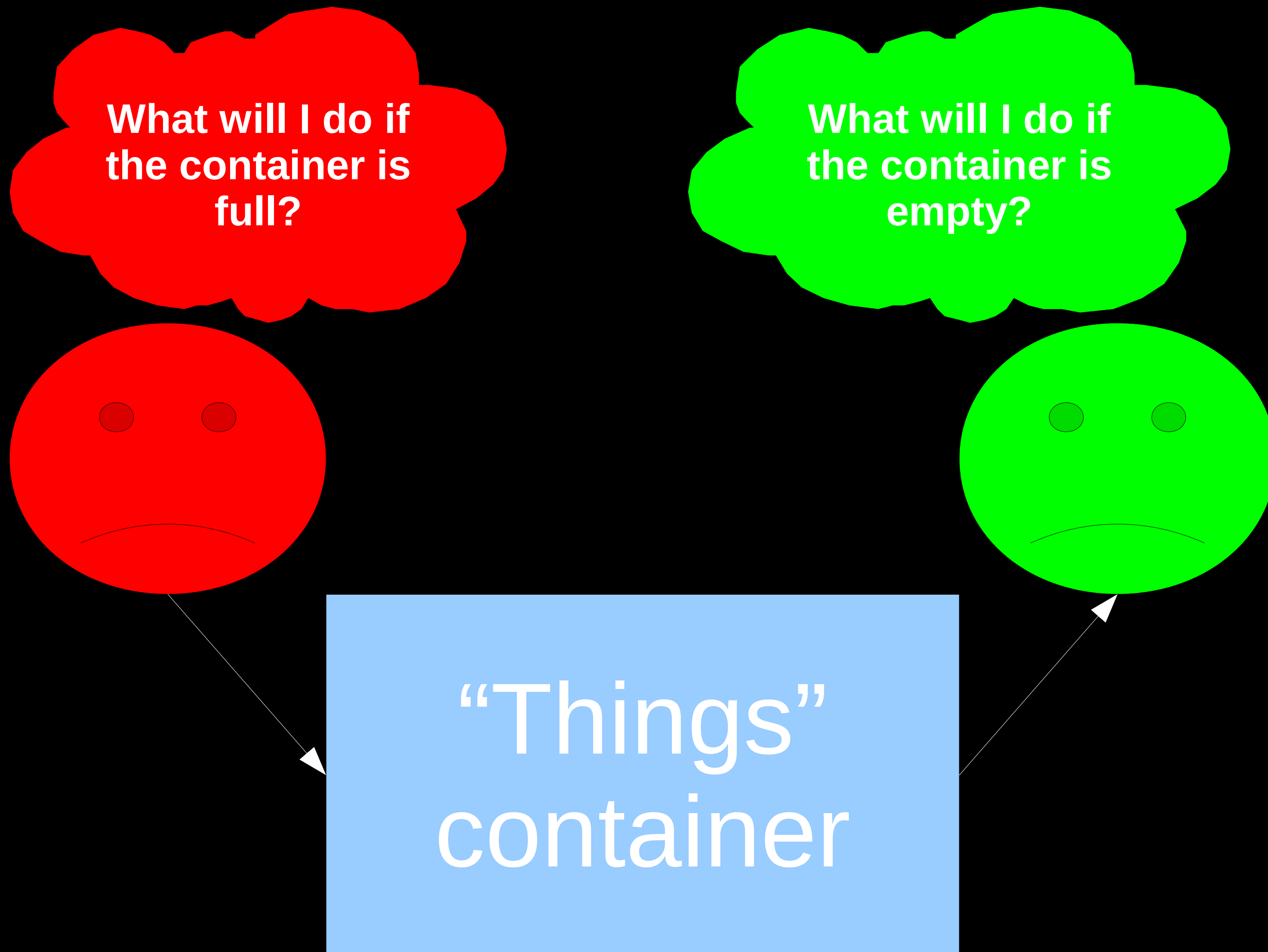
Consistency solutions (for preemption)

- Two common techniques:
  - Locking
    - Acquire a lock before accessing a shared structure, release when done.
    - Acquiring will block until the lock is available.
  - Semaphores
    - up(), down() methods and a counter i.
    - If i > 1 and down() is called, then i--.
      - Else block
    - up() i++ and wake somebody if i is now > 1.

Producer - Consumer

- Thread 1:
  - store(new thing);
- Thread 2:
  - retrieve(something);
  - use(something);
- Both are in loops of the form
  - while(not_done)

Producer – Consumer diagram



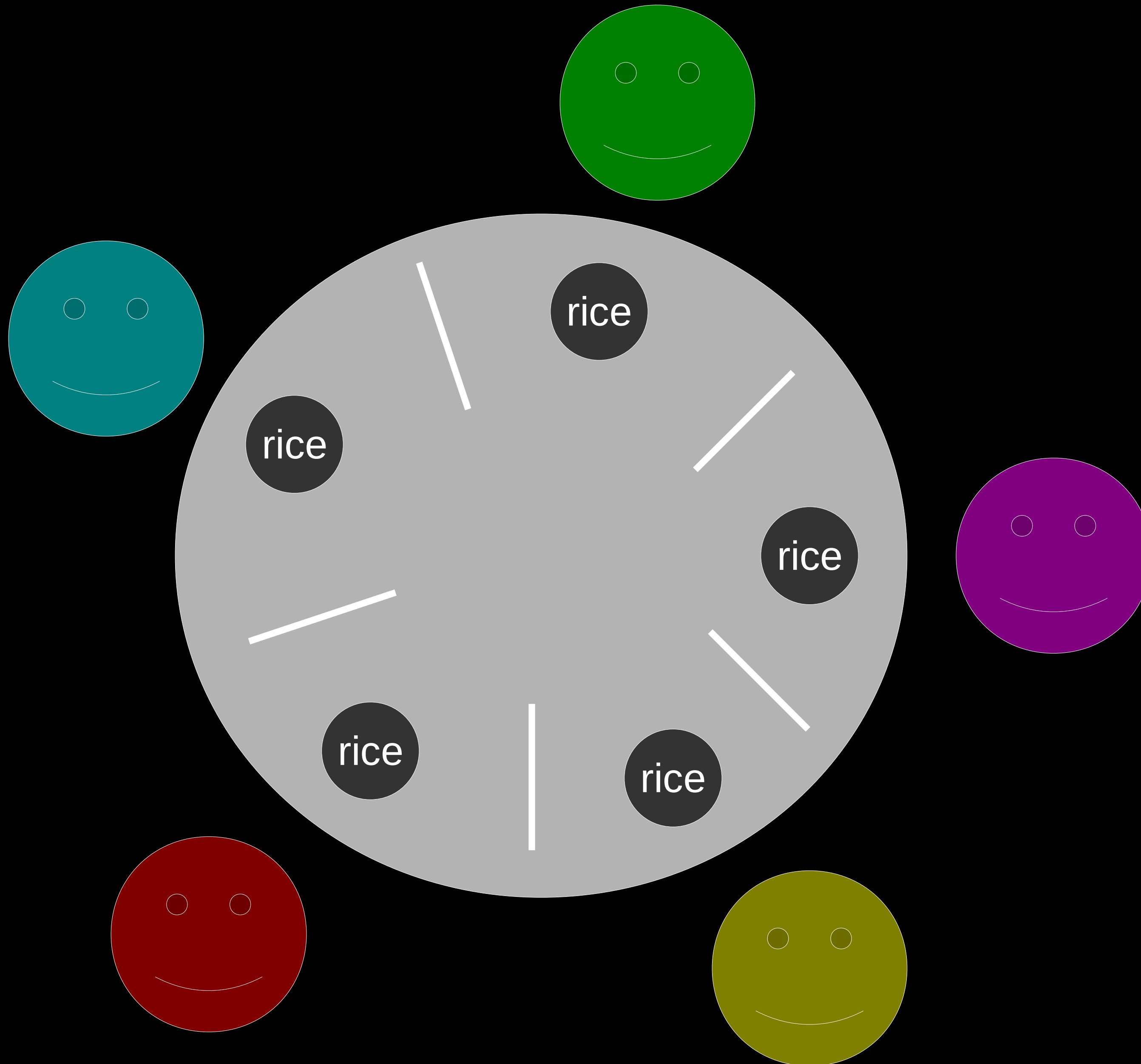
Producer – Consumer solution

- Non-preemptive
  - Producer will yield when some things are produced
  - Consumer will yield when things are all consumed
- Preemptive
  - Need semaphores so that the consumer will block when nothing is ready.
  - Could use locks as well – how?
  - Why is this much better than just looping?

Dining Philosophers problem

- 5 philosophers are eating from 5 bowls of rice using 5 chopsticks.
- Each philosopher thus has a right chopstick and a left chopstick.
- A philosopher can either eat or think. They don't talk. Or make eye contact. It's really annoying.

Dining Philosophers diagram



Dining Philosophers solution

- Have a lock called something like this:
  - WHOLE_BIG_TABLE_LOCK
  - Less than ideal – you only get 1 philosopher eating
- Have a strict ordering on the locks:
  - Only allowed to even attempt to lock 2 if you have 1 locked.
  - Put down in reverse order.
- This avoids what we call *Deadlock*.

Deadlock

- Happens when a graph of lock dependencies has a cycle in it.
- Lots of work is put into tools that can detect this sort of thing.
- It will only happen sometimes, so can be hard to detect manually.
- Careful design to avoid the possibility can be really important.

How are locks actually implemented?

- I'm sure you were just dying to know.
- You could do it by having a small section of code that disables interrupts and thus can't be preempted.
  - Requires kernel-level code
  - Doesn't work on multiprocessor environments
- Atomic test-and-set
  - Works on multiprocessors
  - Requires this special atomic operation

Atomic Test-and-Set

```
if(*address == old-value)
{
    *address == new-value;
    return true;
}
else
{
    return false;
}
```

Locks in a test-and-set world

```
lock.acquire() {
    while (!test-and-set()) {
        lock.queue.add(currentThread);
            // note: queue needs its own lock
        currentThread.sleep();
    }
}

lock.release() {
    // assume you've already got the lock

    Thread t = lock.queue.remove(); // may return null
    t.wakeUp();
}
```

Recap

- Single-thread event dispatch
  - Can think concurrent thoughts without the headaches.
- Multiple threads
  - Preemption
  - Data consistency
  - Resource contention
  - Locks and Semaphores

That's all folks!

Next time:

How to actually do this in Java and maintain consistency on multiprocessors