CSL373: Lecture 5
Deadlocks (no process runnable)
+
Scheduling (> 1 process runnable)

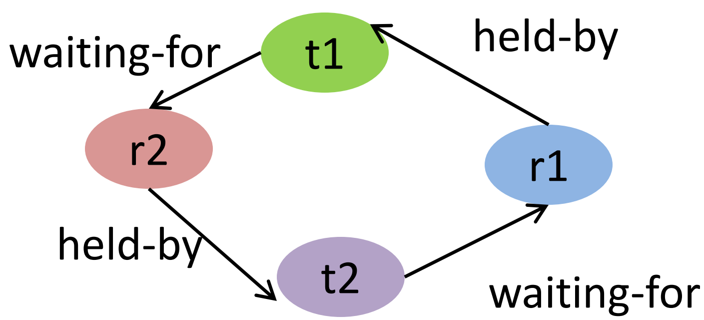
Past & Present

- Have looked at two constraints:
  - Mutual exclusion constraint between two events is a requirement that they do not overlap in time
    - » Enforced using scheduling, locks, semaphores, monitors
  - Precedence constraint between two events is a requirement that one completes before the other
    - » (usually) enforced using scheduling or semaphores
- Synchronization primitive ordering:
  - Atomic instructions can implement locks, locks can implement semaphores (lock + integer counter) or monitors (one implicit lock), and vice versa (of course)
- Today:
  - Deadlock: what to do when many threads = no progress
  - Scheduling: what to do when many threads want progress

Deadlock

- Graphically, caused by a directed cycle in inter-thread dependencies

  e.g., T1 holds resource R1 and is waiting on R2, T2 holds R2 and is waiting on R1



e.g., r1 = disk, r2 = printer

No progress possible

Even simple cases can be non-trivial to diagnose

An example

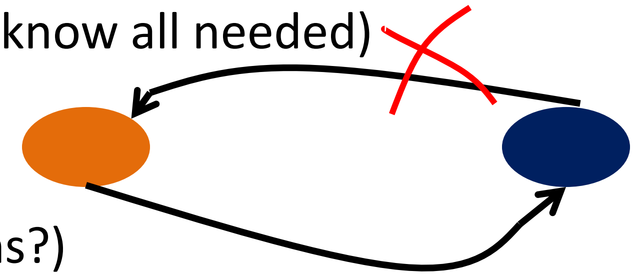
Lock l1, l2

```
void p() {
    l1->Acquire();
    l2->Acquire();
    <ops on shared state>
    l2->Release();
    l1->Release();
}
```

```
void q() {
    l2->Acquire();
    l1->Acquire();
    <ops on shared state>
    l1->Release();
    l2->Release();
}
```

Deadlock Prevention: Eliminate one condition

- Problem: limited access
  - Solutions: Buy more resources, split into pieces, or split the usage of a resource temporally to make it appear "infinite" in number
- Problem: Non-preemption
  - Solution: create copies or virtualize
  - Threads: each thread has it's own copy of registers = no lock
  - Physical memory: virtualized with VM, can take physical page away and give to another process!
- Problem: Hold + wait
  - Solution: acquire resources "all at once"
  - (wait on many without locking any, must know all needed)
- Problem: Circularity
  - Possible Solutions:
    - Single lock for entire system: (problems?)
    - Partial ordering of resources (next)

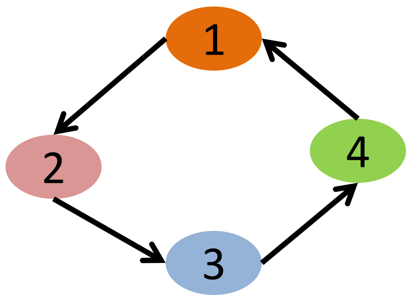


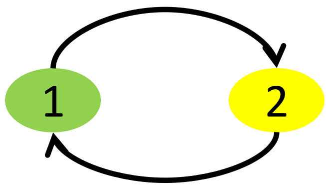
Partial orders: simple deadlock control

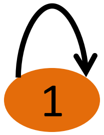
- Order resources (lock1, lock2, ...)
- Acquire resources in strictly increasing (or decreasing) order
- Intuition:

  number all nodes in a graph

  to form a cycle, there has to be at least one edge from high to low number (or to same node)







What if you need to acquire locks in different orders?

- Use trylocks()
- If trylock fails, release all previously held locks and reacquire them in correct order. Need to make sure that the state is *sane* when releasing locks.
- Remember that on releasing and reacquiring locks, the state could have changed.

Example: out-of-order locks

```
//l1 protects d1, l2 protects d2. lock order: l1, l2
//decrements d2, and if the result is 0, increments d1
void increment() {
    l2->acquire();
    int t = d2;
    t--;
    if (t == 0) {
        if (trylock(l1)) d1++;
        else {
            l2->release();
            l1->acquire();   l2->acquire();
            t = d2;       t--;                    //recheck "t"
            if (t == 0) d1++;
        }
        l1->Release();
    }
    d2 = t;
    l2->Release();
}
```

Two phase locking: simple deadlock control

- Acquire all resources, if block on any, release all and retry

```
print_file:
    lock(file);
    acquire  printer;
    acquire  disk;
    … do work …
    release all
```

If any acquire fails, release all previously acquired. Could we do this this without a priori knowledge of what’s needed?

- Pro: dynamic, simple, flexible
- Con:

  Cost with number of resources?

  Length of critical section:

  Abstraction breaking: hard to know what’s needed a priori

Deadlock Detection

1: Work = Avail;
   Finish[i] = False for all i;

2: Find i such that Finish[i] = False and Request[i] <= Work
   If no such i exists, goto 4

3: Work = Work + Alloc[i]; Finish[i] = True; goto 2

4: If Finish[i] = False for some i, system is deadlocked.
   Moreover, Finish[i] = False implies that process i is deadlocked.

When to run?

Detection + correction

- Terminate threads and release resources

  Repeat until deadlock goes away

  Con: Blowing away threads leaves system in what state?

  Wild guess: probably not a sane state.

  Stylized use: acquire all locks, then modify state. Can always blow away the thread if acquire fails (basically two-phase locking with thread termination)

- More fancy: roll back actions of deadlocked threads
  - acquire locks however
  - only modify state using invertible actions
  - get stuck? System kills thread ("bad thread") and inverts actions. Repeat as necessary
  - Each thread now behaves like a "transaction" that would either complete in entirety or non at all (easy for programmer)
  - Problem: tracking actions, constructing inverses (refer databases)

Dirty secret: the most common schemes

- Prevention: Test
  - Pro: no complex machinery. Everyone understands testing
  - Con: interleavings = huge space.
- Kill app
  - Throw deadlock in the same box as infinite loops. Do what you usually do.
  - Works for some applications (emacs, gcc, …). Just rerun.
  - Con: not a valid solution for many applications (example?)

Concurrency Summary

- Concurrency errors:

  One way to view: thread checks condition(s)/examines value(s) and continues with the implicit assumption that this result still holds while another thread modifies

- Fixes?

  Rule 1: don't do concurrency (poor utilization or impossible)

  Rule 2: don't share state (may be impossible)

  Rule 3: If you violate 1 & 2, use one big lock [coarse-grain] (could lead to poor utilization, e.g., Linux on multi-core)

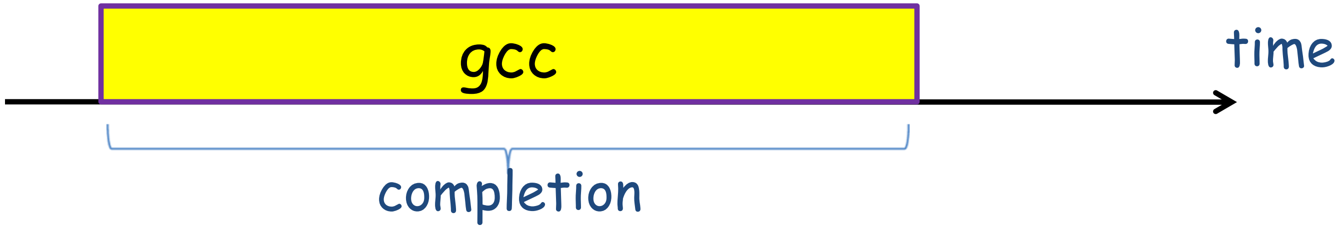
  Last resort: many locks (fine-grain: good parallelism but error prone).

Scheduling: what job to run?

- We'll have three main goals (many others possible)
- minimize response/completion time

  response time = what the user sees: elapsed time to echo keystroke to editor (acceptable delay around 50-100ms)

  Completion time: start to finish of job



- Maximize throughput: operations(=jobs) per second

  minimize overhead (context switching)

  efficient use of resources (CPU, disk, cache, …)

- Fairness: share CPU "equitably"

  Tension: unfairness might imply better throughput or better response times

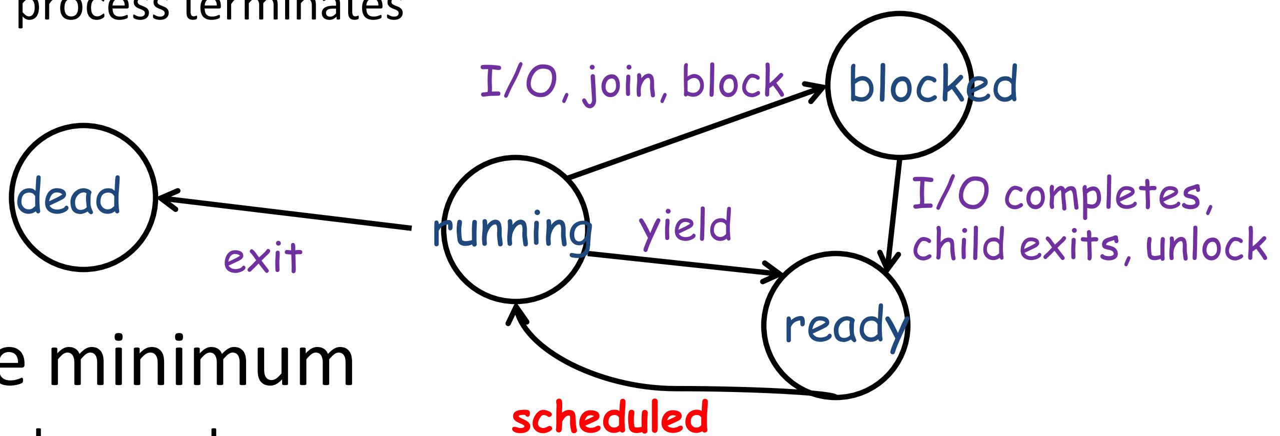
When does scheduler make decisions?

- Non preemptive minimum:

  When process voluntarily relinquishes CPU

  » process blocks on an event (e.g., I/O or synchronization)

  » process terminates



- Preemptive minimum

  All of the above, plus:

  Event completes: process moves from blocked to ready

  Timer interrupts

  Priorities: One process can be interrupted in favor of another

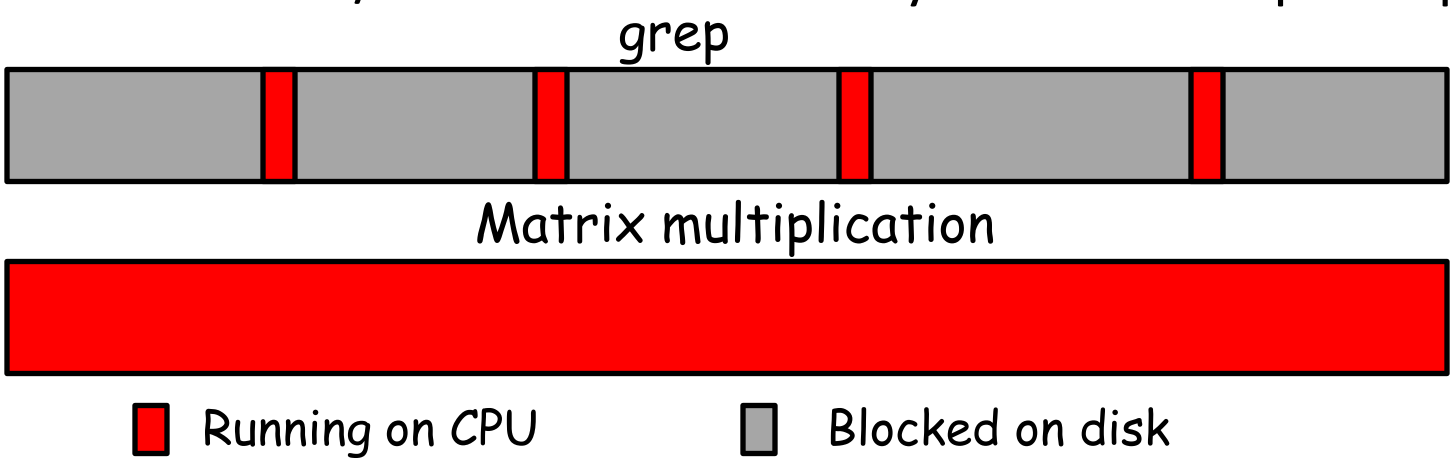
Can think of: I/O device = special CPU

- I/O device ≈ one special purpose CPU

  "special purpose" = disk drive can only run a disk job, printer a print job, …

- Implication: computer system with n I/O devices ≈ n+1 CPU multiprocessor

  Result: all I/O devices + CPU busy = n + 1 fold speedup!



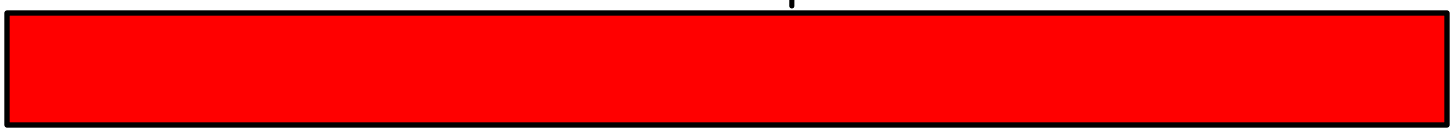
overlap them just right? ave. completion time ≈ halved

Process *model*

- Process alternates between CPU and I/O bursts

  CPU-bound job: long CPU bursts

  Matrix multiplication



  I/O-bound job: short CPU bursts

  emacs

  I/O burst = process idle, switch to another “for free”

  Problem: don’t know job’s type before running

- An underlying assumption:

  “response time” most important for interactive jobs, which will be I/O bound

Universal scheduling theme

- General multiplexing theme: what's "the best way" to run *n* processes on *k* nodes? *(k < n)*

  we're (probably) always going to do a bad job

- Problem 1: mutually exclusive objectives

  no one best way

  latency vs throughput conflicts

  speed vs fairness

- Problem 2: incomplete knowledge

  User determines what's most important. Can't mind read

  Need future knowledge to make decision and evaluate impact. Use past = future

- Problem 3: real systems = mathematically intractable

  Scheduling very ad hoc. "Try and see"

Scheduling

- Until now: Processes. From now on: resources

  Resources are things operated on by processes

  e.g., CPU time, disk blocks, memory page, network bufs

- Categorize resources into two categories:

  Non-preemptible: once given, can't be reused until process gives back. Locks, disk space for files, terminal.

  Preemptible: once given, can be taken away and returned. Register file, CPU, memory.

- A bit arbitrary, since you can frequently convert non-preemptible to preemptible:

  create a copy and use indirection

  e.g., physical memory pages: use virtual memory to allow transparent movement of page contents to/from disk.

How to allocate resources?

- Space sharing (horizontal):
  - How should the resource split up?
  - Used for resources not easily preemptible
    - e.g., disk space, terminal
  - Or when not *cheaply* preemptible
    - e.g., divide memory up rather than swap entire memory to disk on context switch.
- Time sharing (vertical):
  - Given some partitioning, who gets to use a given piece (and for how long)?
  - Happens whenever there are more requests than can be immediately granted
  - Implication: resource cannot be divided further (CPU, disk arm) or it’s easily/cheaply pre-emptible (e.g., registers)

First come first served (FCFS or FIFO)

- Simplest scheduling algorithm

  Run jobs in order that they arrive

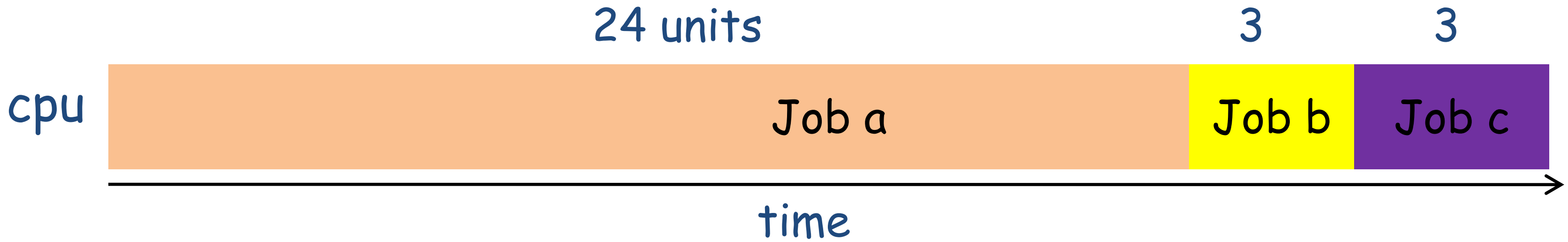
  Uni-programming: Run until done (non-preemptive)

  Multi-programming: put job at back of queue when blocks on I/O

  Advantage: very simple

  Disadvantage: wait time depends on arrival order. Unfair to later jobs (worst case: long job arrives first)

  e.g.,: three jobs (A, B, C) arrive nearly simultaneously)



what's the average wait time?

Summary

- Mutual exclusion introduces dependencies

  circular dependencies = deadlock

  can either prevent circularities or recover from them

- > 1 process = choice = scheduling
  - We'll first look at traditional systems
  - Goals: response time, throughput, fairness

  - Next time: specific scheduling algorithms