CSS503A

Lecture #7

Administrivia

- lab 1 solution
- assignment 1 solution
- assignment 1 solution - Go
- tour of /proc

- midterm
  - ground rules
  - study guide?

Lab 2/ Ass 2 : due on Tuesday (this week & next)

→ Thursday's ok

Our Story So Far

- threads vs processes: depends on the problem
- deadlock: mutual dependency
- starvation: thread/process does not get its turn

- Posix threads: low level interface
    - bypass language type system

- critical section: region of code that only one process/thread at a time can execute

- goals/requirements for critical section
    - mutual exclusion
    - progress (when critical section is open, something must be chosen)
    - bounded waiting (every thread/process gets its turn)

- race condition: incorrect/inconsistent results due to unfortunate timing

OSSF (cont.)

- mutex
- semaphore
- condition variable
- Monitor

- Decker's algorithm combines two separate approaches which are individually problematic

- modern hardware requires hardware support
  - e.g. test-and-set or swap

- Semaphore protects a resource
- mutex protects code (mutual exclusion)

Semaphore = mutex + counter + condition variable

Semaphore : P() : proben/wait/down get
V() : verhogen/signal/up release

OSSF (cont.)

- waiting for resource/condition using semaphore & condition variable:

```
acquire lock
while ! resource
    pthread_(timed)wait(cond_var, lock(, timeout))
  critical section
signal (cond-var)
release lock
```

pointers

- alt pattern:

```
acquire lock
while !resource
    wait (cond, lock)
assert ownership of resource
release lock

    non-critical
    code

acquire lock
release resource
signal cond
release lock
```

OSSF (cont.)

Monitors: language-level support

- Simulate in C++ using RAII
  - constructor locks mutex
  - destructor releases mutex

→ destructor is called automagically when variable leaves scope

Dining Philosophers Problem

- classic toy problem of concurrency
  - 5 philosophers sitting around table
  - 5 chopsticks (one left & one right of each philosopher)
  - philosopher has 3 states
    - thinking
    - hungry
    - eating

- goal: to avoid deadlock without letting one of the philosophers starve

Philosopher:

```
    get left
    get right
        eat
    release right
    release left
=> deadlock
```

Sleeping Barbers Problem

- sounds stupid, but is yet another classical illustration of concurrency

- barber sleeps until he gets a customer
- customer arrives:

```
if barber is sleeping, wake up barber
else if chair available in waiting room, take chair
else leave
```

- when barber finishes with customer,

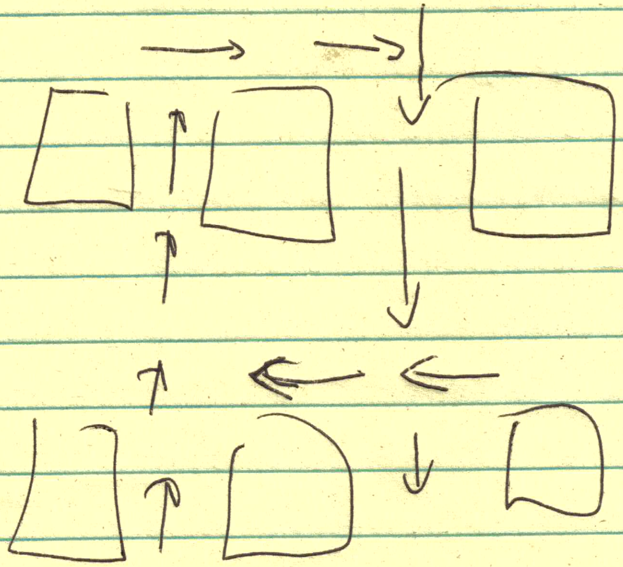
```
if customer(s) waiting, get customer
else go back to sleep
```

- with concurrency, if two customers arrive at the same time, both may try to take same chair, or simultaneously take the barber's chair

=> assignment 2

Deadlocks

- "gridlock": traffic at intersections (grid = blocks)



no-one may proceed

- prevent gridlock: don't enter intersection unless you can proceed past the intersection

Deadlocks (cont.)

slightly more complex

- multiple resources (or resource classes)
  - CPU, memory, I/O devices, ...

=> process acquires & releases exclusive access to resource

- informally, a deadlock occurs when a process P_1 holds a resource R_A and is waiting on R_B while the process P_2 is holding R_B and waiting on R_A
  - or the dependencies can be even more complex

$$\begin{tikzcd}
P_1 \arrow[r] & R_A \arrow[r] & P_2 \arrow[r] & R_B \arrow[r] & P_3 \arrow[r] & R_C \arrow[r] & \cdots & R_* \arrow[lllllll, bend left]
\end{tikzcd}$$

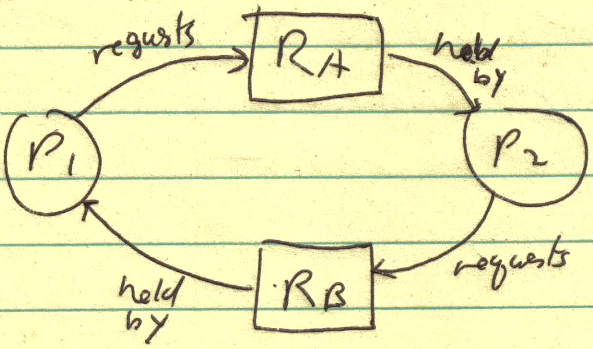
Deadlocks (cont.)

- dealing with deadlocks: 3 basic approaches
  1) prevention / avoidance
     - ensure that the system can never enter deadlock state
  2) detection & recovery
  3) ostrich (head-in-the-sand)
     - ignore the problem
     - what most OSs do anyway
     - user response: "why is this taking so long?"
       ⇒ user kill & restart

Deadlocks (cont.)

- deadlocks occur only if all of the following conditions are met
    1) mutual exclusion - exclusive access to resource
    2) hold & wait
    3) no pre-emption - process must voluntarily release held resources
    4) circular wait - cycle in 2-color resource allocation graph

individual resources (can't use graph algorithm for resource classes



Deadlocks (cont.)

- deadlock avoidance : attack one of the 4 conditions

1) disallow mutual exclusion

2) disallow hold or disallow wait
    a) require all resources to be allocated at program startup (no waiting)
    → low resource utilization
    b) allow resource requests only when process has none (no holding)
    → starvation

3) allow preemption

4) prevent circular wait
    - strict ordering of resources
    - process may only request resources in increasing order

Deadlocks (cont.)

- detection: construct resource allocation graph & look for cycle

  individual resources, not resource classes

- recovery
  1) kill process(es)
     a) abort all deadlocked processes
     b) abort one process at a time until cycle is broken
  2) pre-empt resource(s)
     - choose "victim" & roll back
       - select different victims each time (otherwise: starvation)