COS 140: Foundations of Computer Science

Process Synchronization: Semaphores

Fall 2018

Homework, etc.

- Reading: Chapter 20
- Homework: Exercises at end of Chapter 20
- Due: Friday, 11/9
- **NOTE:** Remember that Prelim II is Wednesday, 11/14!

Process Synchronization

3 / 20

Process synchronization

- Problems $\Leftarrow$ processes share resource – e.g., memory or device
- Possible: both want to use the resource simultaneously and incompatibly

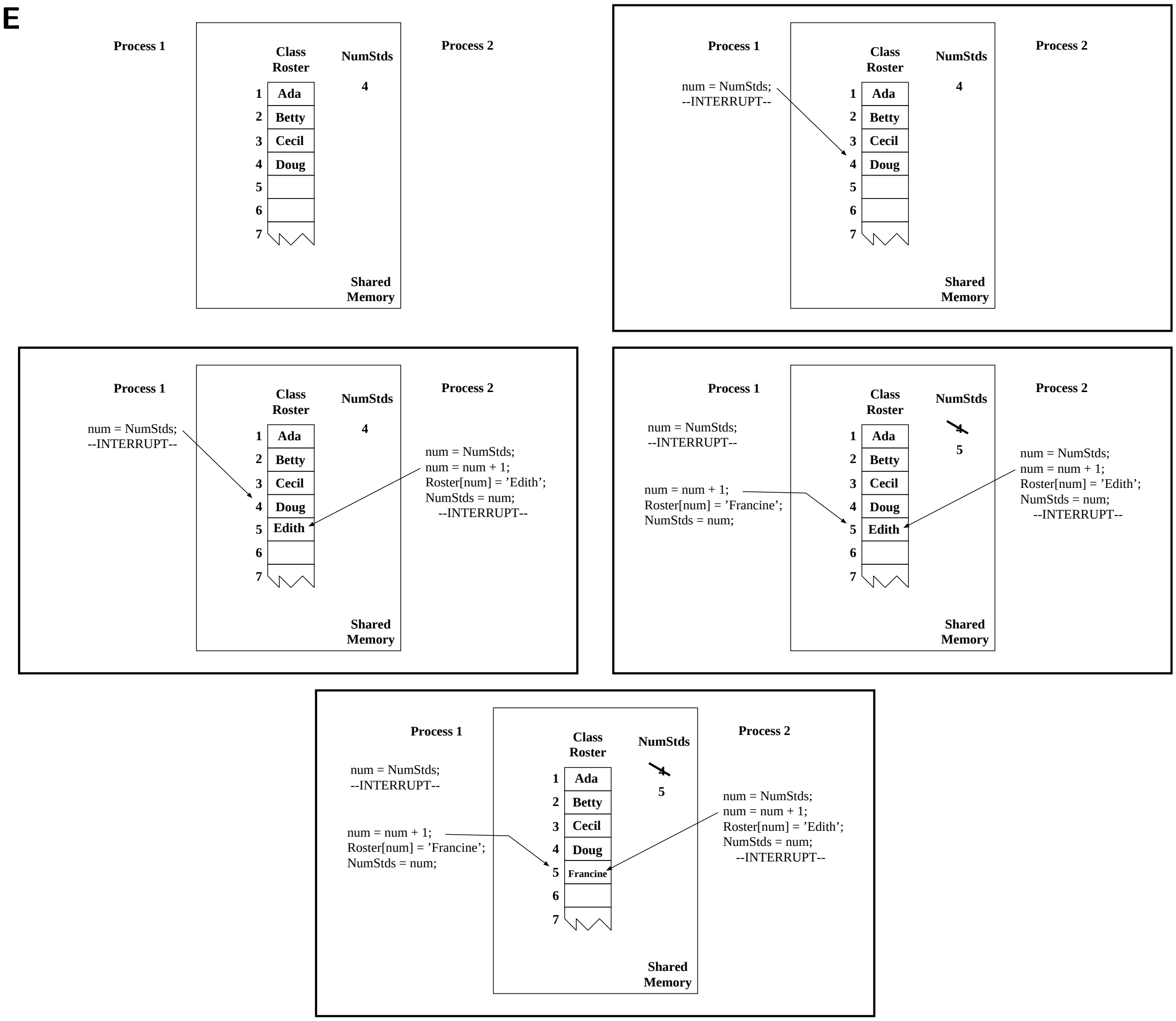
Process synchronization

- Example: Two movers, Bob and Jill, are putting furniture into house
  - Both approach doorway (shared resource) simultaneously with a chair they can't see around
  - What happens? Bob goes first; Jill goes first; both get stuck!
- Example: Two tellers, two customers, Sue and Jim, with joint account
  - Joint account = shared resource
  - Both want to put $100 in at same time, tellers update accounts
  - What can happen?
    - Sue or Jim go first → $200 in account
    - Both at same time...? Maybe only $100 in account!

Race conditions

- Examples of *race conditions*: outcome depends on timing/speed of participants
- Computer: processes are the "racers"
- Resource: any non-simultaneously-sharable thing

E



Approach: Mutual exclusion

- Identify *critical regions* where shared resource is being accessed – in teach process
- If we only allow one process into its critical region at a time → no conflict, no race condition

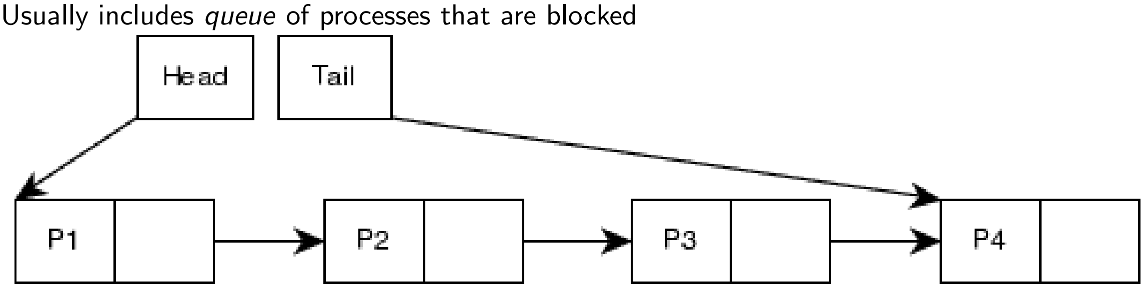
Solution requirements

- Result is predictable
- Solution does not depend on speed of processes
- Solution does not depend on number of CPUs

Semaphores

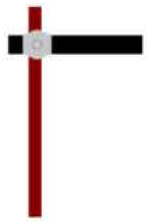
One solution: Semaphores

- *Semaphores*:
  - *synchronization primitive*
  - guarantee mutual exclusion when used correctly
- Semaphore: data structure (and associated code) that:
  - Keeps track of whether or not the resource is in use...
  - ...or alternatively, keeps track of whether anyone is in their critical region
  - *Blocks* process if it tries to enter its critical region when someone else is in theirs
- Usually includes *queue* of processes that are blocked



Using semaphores

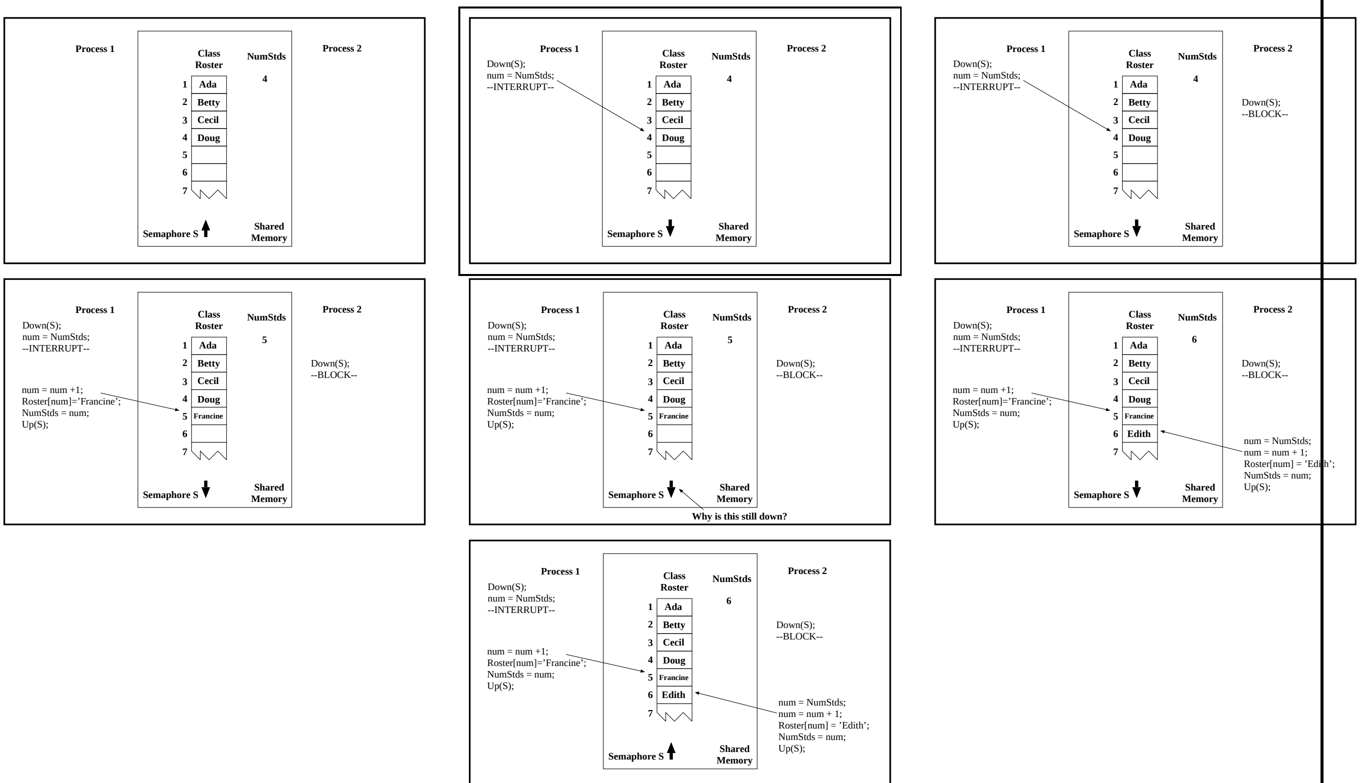
- A semaphore S has two major procedures associated with it:
  - P(S), or Down(S)
  - V(S), or Up(S)
  - (V for Dutch *verhoog* (increase) and P for *prolaag* (try to decrease) [Dijkstra])
- Think of semaphore signaling “okay” when up, “stop!” when down


Using semaphores

- Initially: semaphore is “up”
- Key: wrap calls to Down and Up around each process’ critical region
  - Suppose we have semaphore S
  - Process wants to enter critical region, calls Down(S)
  - If semaphore “up” then semaphore ⇒ “down”, process continues
  - If semaphore “down” then process blocks
  - When process leaves critical region: call Up(S)

Example



Implementation

Data structure

- Semaphores are *data structures*
- Two parts: count and queue
  - Count is an integer
  - Queue contains process IDs
- This type of semaphore = *mutual exclusion*, or *mutex*, semaphore
- Count: set to 1 for mutex semaphores

Procedures

- Can implement as separate procedures or *methods* of a semaphore class
- Down() and Up() have to be *atomic*
- Only OS can do this!
- Usually have to ask OS for semaphores, then Down() and Up() are accessed via system calls

Procedures

- Down(mutex):
  - Decrement count
  - If < 0, then block current process
  - Otherwise, continue (into critical region)
- Up(mutex):
  - Increment count
  - If there are blocked processes, allow one to continue
  - Note: semaphore is still "down" until count is positive
- Example

Counting semaphores

- Mutex semaphores: just one kind
- Semaphores really *count* number of free resources
- Mutex: only 1 unit of resource = critical region
- Other cases: may have $\geq 1 \Rightarrow$ *counting semaphores*

Example

- Producer–consumer problem
- Two processes, shared resource of finite size
- Producer puts things into the resource
- Consumer removes them
- Producer must block when full, consumer must block when empty

Producer–Consumer Problem

```
Semaphores: mutex, full, empty;
Set count of mutex=1, full=0,
    empty=size of resource;

Producer:                   Consumer:
  loop forever:               loop forever:
    Get thing to put in;        Down(full);
    Down(empty);                Down(mutex);
    Down(mutex);                Take thing out;
    Put thing in resource;      Up(mutex);
    Up(mutex);                  Up(empty);
    Up(full);                   Use thing;
  end loop.                   end loop.
```


Discussion

Problems with semaphores

- □ Easy to make mistakes:
  - Forget to do Up(S)
  - Too many Down's
  - Crossed semaphores
- □ Cheating
- □ Too low-level

Other approaches

- □ Event counters
- □ Monitors
- □ Message passing
- □ Equivalence of primitives