Semaphores

Semaphores

- Concurrency Control Mechanisms, In General
- Semaphore Concepts
- Producer / Consumer

Semaphores

Concurrency Control Mechanisms, In General

C.C. Mechanisms

- We have studied **locks** using **atomic instructions**
  - What are these two things?
- We have studied **messages / mailboxes**

- We noted that you could build a lock out of a messaging system
- **T,P,S:** Can you build a messaging system using locks?

C.C. Mechanisms

- **General rule:**
- You can build any C.C. mechanism out of any other (might be ugly), if it has the two core requirements:
  - Mutex
  - Sleep / Wakeup

Two lesser requirements:

- Arbitrary # of processes
- No need for continual participation (“a process can go on vacation”)

C.C. Mechanisms

Mutex

- The system needs a way for one process to block another from having access to shared data

Sleep / Wakeup

- System puts a process to sleep (no CPU) if it blocks for a long time

C.C. Mechanisms

- **Messages with from Locks**
- Used a spinlock (and/or disabling interrupts) to control access to data structures (mutex)
- Changed process state, called dispatcher, when needed to block (sleep / wakeup)

C.C. Mechanisms

- **Locks built from Messages**
- Read a message when you wanted a lock. Would block if no message. (mutex, sleep)
- Send a message when you release the lock (wakeup)

Semaphores

Semaphore Concepts

Semaphores

- A **semaphore** is a concurrency control mechanism built around a *counter*
  - Can never go negative
  - Increment / decrement

- Designed by Edgar Dijkstra
- One of the first C.C. mechanisms
  (older than locks, maybe?)

Semaphores

- **`P()`** (“Proberen:” Dutch for “try”) (also called **`wait`**)
  - If counter is zero, block until nonzero
  - Then decrement

- **`V()`** (“Verhogen:” Dutch for “increase”) (also called **`post`**)
  - Increment counter
  - If any `P()`s are blocked, unblock one

https://www.youtube.com/watch?v=ukM_zzrIeXs

Semaphores

- A **binary semaphore** is:
  - Initialized to 1
  - Never intended to increase beyond

- Can be used as a lock
- Optionally, can be simplified to a single bit

Semaphores

- A **counting semaphore** is:
  - Initialized to any (non-neg) value
  - Can get large
- Used to count available resources
- Blocks if everything is busy

ICA: Semaphores

Adapt our old concurrency example to use a semaphore. Use the semaphore to *protect the critical section*; do not use it to do the counting.

```
while True:
    load  x → $r1
    inc   $r1
    store $r1 → x
```

Semaphores

```
s = init_semaphore(1)
while True:
    P(s)
    load  x → $r1
    inc   $r1
    store $r1 → x
    V(s)
```

Semaphores

```
s = init_semaphore(1)
while True:
    P(s)
    load  x → $r1
    inc     $r1
    store $r1 → x
    V(s)
```

Notice that we are treating the P() and V() operations as atomic ops – even though they are subroutines.
Probably, the functions use atomic instructions or locks inside, to provide this property.

ICA: Write the Pseudocode

- Use a *counting semaphore* to simulate two types of concurrent processes: red processes need only one widget to do anything, but blue need two. (There can be many red and many blue processes.)
- The system starts with 5 widgets. Use the counting semaphore to keep track of how many are free.
- Write `start_red()`, `end_red()`, `start_blue()`, `end_blue()`

```
w = init_semaphore(5)

start_red():            start_blue():
  P(w)                    P(w)
                          P(w)

end_red():              end_blue():
  V(w)                    V(w)
                          V(w)
```

What's the bug in this code?

Semaphores

- Semaphores are susceptible to deadlock, just like any other c.c. mechanism!

What were the 4 conditions necessary for deadlock?

Semaphores

- Semaphores are susceptible to deadlock, just like any other c.c. mechanism!

- 4 conditions for deadlock:
  - Mutual exclusion
  - No pre-emption
  - Hold and wait
  - Circular wait

Semaphores

- **Mutual Exclusion**
- Yes, a counting semaphore can allow multiple users at once
- But once a resource has been “allocated” with `P()`, no one else can use it
- Once all the resources have been claimed, all `P()` operations block

```
w = init_semaphore(5)

start_red():      start_blue():
  P(w)              P(w)
                    P(w)

end_red():        end_blue():
  V(w)              V(w)
                    V(w)
```

This is the only place in our code where Hold and Wait occurs.

Circular Wait happens when many processes block in `start_blue()`

Producer / Consumer

- A classic form of process interaction is a producer / consumer
- One or more **producer** processes deliver blocks of data (messages, text, whatever) into a shared storage
  - Block when full
- One or more **consumer** processes consume the same blocks from the shared storage
  - Block when empty

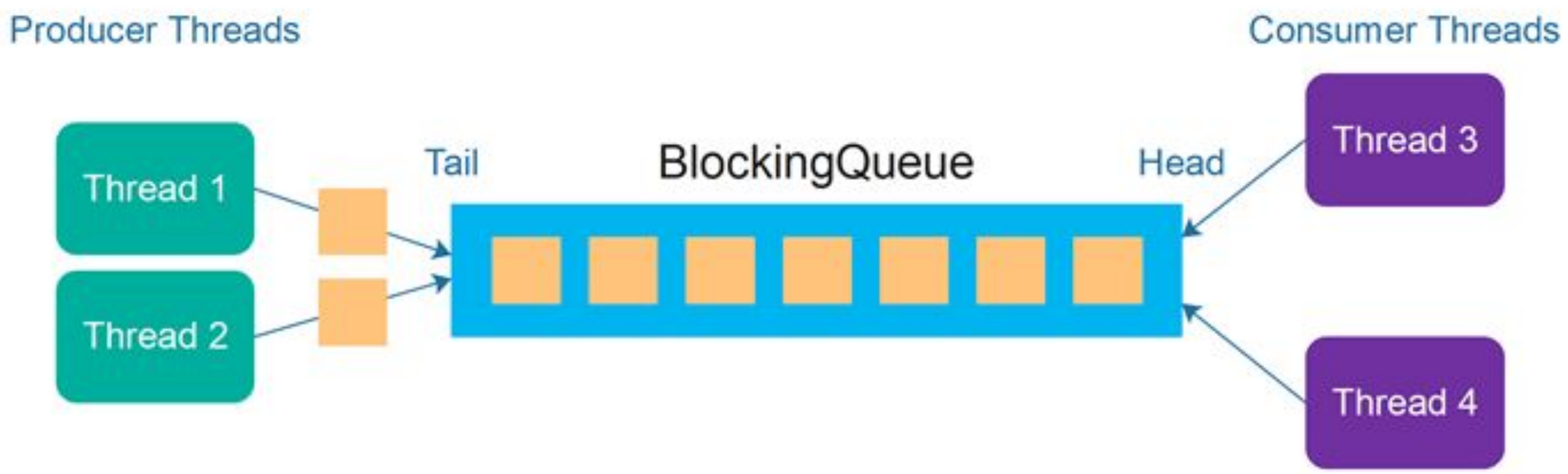
ICA: Producers and Consumers

- What is a practical use-case for the producers / consumers model?
- What problem(s) could this be used to solve with multiple processes?

ICA: Producers and Consumers

- What is a practical use-case for the producers / consumers model?
- What problem(s) could this be used to solve with multiple processes?

PIZZA EXAMPLE!



https://jenkov.com/tutorials/java-concurrency/producer-consumer.html

Code as a Group:

- Use a <u>pair</u> of counting semaphores to model a producer / consumer system.

- Write `produce()`, `consume()`. Block the processes when necessary. (Assume that the system has only 10 data buffers, and it starts with all of them empty.)

- You may hand-wave the code needed to actually allocate the buffers and copy the data blocks; we are focused on concurrency control.

```
f = init_semaphore(0)
e = init_semaphore(10)
```

```
produce():
  P(e)
  ...data...
  V(f)
```

```
consume():
  P(f)
  ...data...
  V(e)
```