# CSE 153
# Design of Operating Systems

## Winter 2023

Lecture 10: Monitors

# Today/Looking ahead

- Today:
  - Monitors (briefly)
  - Synchronization wrap up
  - Examples
- What's next?
  - Scheduling – how to decide which thread/process to run next?
    - Lab 2 ☺
  - Other concurrency issues
    - Deadlock – the deadly embrace!
- Midterm!

# Monitors

- A monitor is a programming language construct that controls access to shared data
  - Synchronization code added by compiler, enforced at runtime
  - Why is this an advantage?
- A monitor encapsulates
  - Shared data structures
  - Procedures that operate on the shared data structures
  - Synchronization between concurrent threads that invoke the procedures

# Monitor Semantics

- A monitor guarantees mutual exclusion
  - Only one thread can execute any monitor procedure at any time (the thread is "in the monitor")
  - If a second thread invokes a monitor procedure when a first thread is already executing one, it blocks
    - » So the monitor has to have a wait queue…
- Monitors also support waiting on conditions
  - Situation: we enter a monitor and find that we need to wait
    - » E.g., producer when the buffer is full
  - If we just wait, the monitor is blocked
    - » So, monitors support waiting while releasing the monitor

# Account Example

```
Monitor account {
  double balance;

  double withdraw(amount) {
    balance = balance – amount;
    return balance;
  }
}
```

**Threads block waiting to get into monitor**

```
withdraw(amount)
  balance = balance – amount;

withdraw(amount)

withdraw(amount)

  return balance (and exit)

  balance = balance – amount
  return balance;

  balance = balance – amount;
  return balance;
```

**When first thread exits, another can enter. Which one is undefined.**

- Hey, that was easy!
- But what if a thread wants to wait inside the monitor?
  - » Such as "mutex(empty)" by reader in bounded buffer?

# Monitors, Monitor Invariants and Condition Variables

- A monitor invariant is a safety property associated with the monitor, expressed over the monitored variables. It holds whenever a thread enters or exits the monitor.
- A condition variable is associated with a condition needed for a thread to make progress once it is in the monitor.

```
Monitor M {
  ... monitored variables
  Condition c;

  void enter_mon (...) {
    if (extra property not true) wait(c);        waits outside of the monitor's mutex
    do what you have to do
    if (extra property true) signal(c);          brings in one thread waiting on condition
  }
```

# Condition Variables

- Condition variables support three operations:
  - Wait – release monitor lock, wait for C/V to be signaled
    - » So condition variables have wait queues, too
  - Signal – wakeup one waiting thread
  - Broadcast – wakeup all waiting threads

- Condition variables *are not* boolean objects
  - "if (condition_variable) then" … does not make sense
  - "if (num_resources == 0) then wait(resources_available)" does
  - An example will make this more clear

# Monitor Bounded Buffer

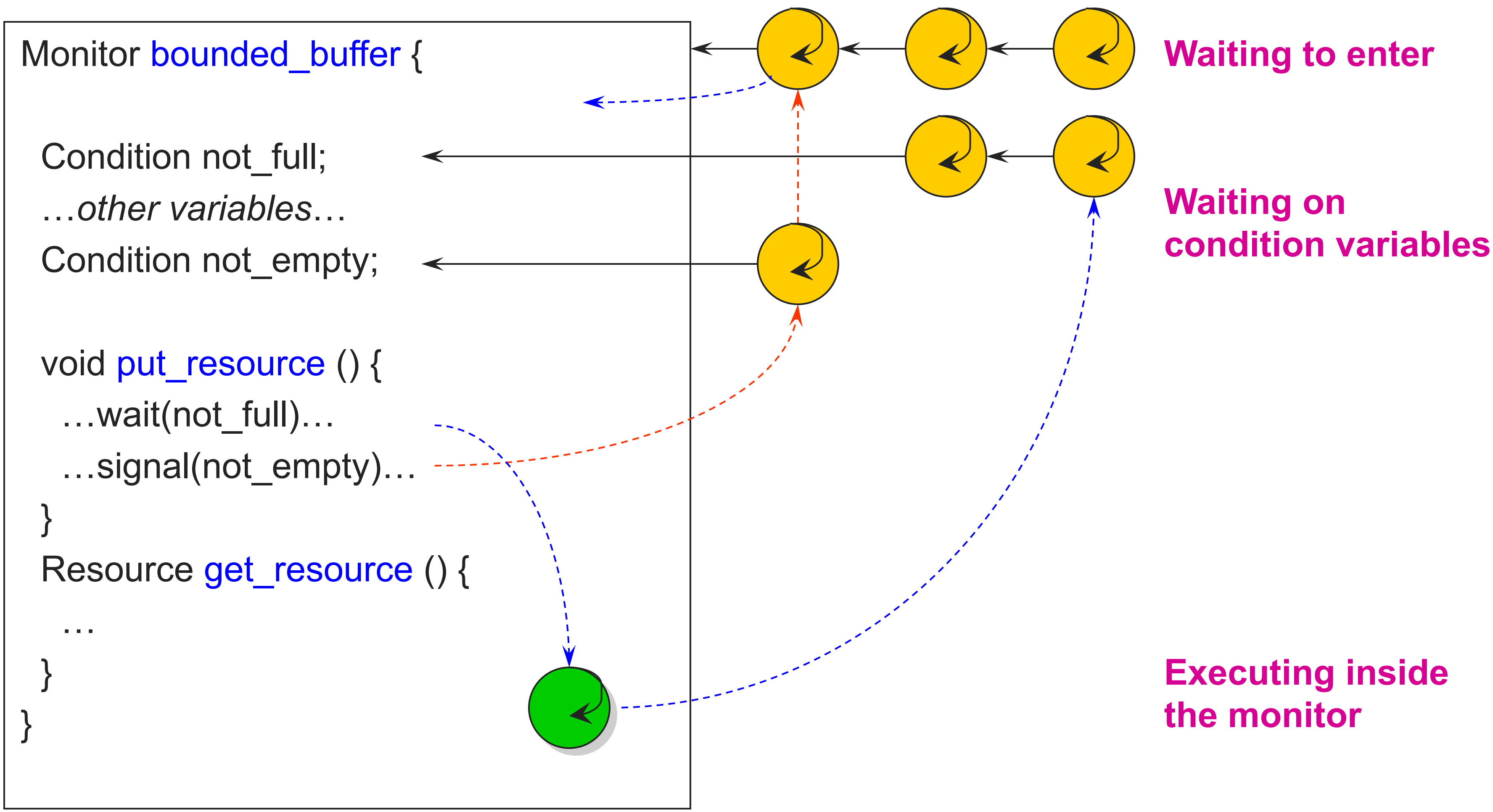
```
Monitor bounded_buffer {
  Resource buffer[N];
  // Variables for indexing buffer
  // monitor invariant involves these vars
  Condition not_full; // space in buffer
  Condition not_empty; // value in buffer

  void put_resource (Resource R) {
    if (buffer array is full)
        wait(not_full);
    Add R to buffer array;
    signal(not_empty);
  }
```

```
  Resource get_resource() {
    if (buffer array is empty)
        wait(not_empty);
    Get resource R from buffer array;
    signal(not_full);
    return R;
  }
} // end monitor
```

- What happens if no threads are waiting when signal is called?

# Monitor Queues



# Monitors and Java

- A lock and condition variable are in every Java object
  - No explicit classes for locks or condition variables
- Every object is/has a monitor
  - At most one thread can be inside an object's monitor
  - A thread enters an object's monitor by
    - » Executing a method declared "synchronized"
      - Can mix synchronized/unsynchronized methods in same class
    - » Executing the body of a "synchronized" statement
      - Supports finer-grained locking than an entire procedure
- Every object can be treated as a condition variable
  - Object::notify() has similar semantics as Condition::signal()

# Advanced synchronization (FYI)

- Concurrency patterns (see little book of semaphores)
- Other advanced primitives
  - Read-Copy-Update locks, Seqlocks, Futexes, transactional memory
- Lock free synchronization
- Synchronization pathologies and performance tuning
  - e.g., Convoying, contention

# More synchronization practice

- You take a break from studying cs153 to play frisbee with your friends.  We have one or more frisbees, and there are two or more of you.  Each student waits until they have a frisbee and their neighbor doesn’t have one and then throws the frisbee.

- What happens if the number of frisbees is equal to the number of players?

# More synchronization practice

- CS153 students are studying for the midterm over the national CS dish (Pizza). Each pizza pie has 8 slices. Each student eyes the pie, then grabs the next slice.
  - What race conditions can happen?
  - How can you resolve them?

- A student that grabs the last slice should order the next pie. Extend your implementation to do that