EDAF35: OPERATING SYSTEMS

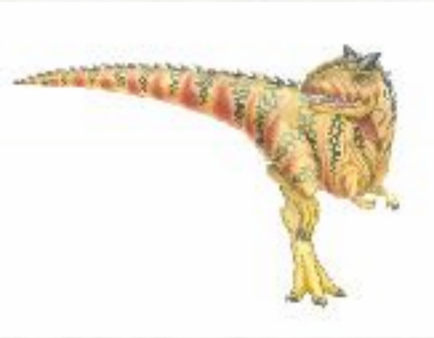
MODULE 5.B SYNCHRONIZATION

CONTENTS

SYNCHRONIZATION

- "Critical Section" problem
- Hardware support for synchronization
- Higher-level mechanisms:
  mutex, lock, semaphore, monitor, condition variable
- Deadlocks
- Alternative approaches

CHAPTER 6
SYNCHRONIZATION
(OR 5)



CHAPTER 7
DEADLOCKS

AN EXAMPLE

RACE CONDITION, CRITICAL SECTION

Producer

```
while (true) {
      /* produce an item in next produced */

      while (counter == BUFFER_SIZE) ;
            /* do nothing */
      buffer[in] = next_produced;
      in = (in + 1) % BUFFER_SIZE;
      counter++;
}
```

```
register1 = counter
register1 = register1 + 1
counter = register1
```

CIRCULAR BUFFER

machine code

Consumer

```
while (true) {
      while (counter == 0)
            ; /* do nothing */
      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      counter--;
      /* consume the item in next consumed */
}
```

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Example execution with "counter = 5" initially

S0: producer execute `register1 = counter`	{register1 = 5}
S1: producer execute `register1 = register1 + 1`	{register1 = 6}
S2: consumer execute `register2 = counter`	{register2 = 5}
S3: consumer execute `register2 = register2 - 1`	{register2 = 4}
S4: producer execute `counter = register1`	{counter = 6 }
S5: consumer execute `counter = register2`	{counter = 4}

Race Condition:
order decides the result

Critical Section:
if interleaved with another leads to a race

CRITICAL SECTION PROBLEM

SYNCHRONIZATION

Typical process (simplified)

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (true);
```

Solutions must provide

1. MUTUAL EXCLUSION
2. PROGRESS
3. BOUNDED WAIT

- Various solutions exist:
  - sw only vs. hw supported
  - kernel vs. user level
- Most common: locks

HARDWARE SUPPORT FOR SYNCHRONIZATION IS ESSENTIAL FOR EFFICIENT OPERATION

HARDWARE SUPPORT

SYNCHRONIZATION

Test and Set

```
boolean test_and_set (boolean *target)
    {
      boolean rv = *target;
      *target = TRUE;
      return rv:
}
```

Compare and Swap

```
int compare_and_swap(int *value, int expected, int new_value)
    {
        int temp = *value;
        if (*value == expected)
            *value = new_value;
        return temp;
    }
```

- atomic execution (by hardware implementation)
- semantic definitions above (not real implementation)
- building blocks for more complex constructs: **mutex locks**

```
do {
        while (test_and_set(&lock))
           ; /* do nothing */
              /* critical section */
        lock = false;
              /* remainder section */
     } while (true);
```

BUSY WAIT (SPINLOCKS)

```
do {
             while (compare_and_swap(&lock, 0, 1) != 0)
                 ; /* do nothing */
                   /* critical section */
             lock = 0;
                   /* remainder section */
    } while (true);
```

HIGHER ABSTRACTIONS: SEMAPHORES

SYNCHRONIZATION

```
Semaphore S: integer value
```

+

Two atomic operations

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
} // P(), take()
```

```
signal(S) {
        S++;
} // V(), give()
```

OBS: NO TWO PROCESSES SHOULD EXECUTE WAIT/SIGNAL AT THE SAME TIME (CRITICAL SECTIONS)

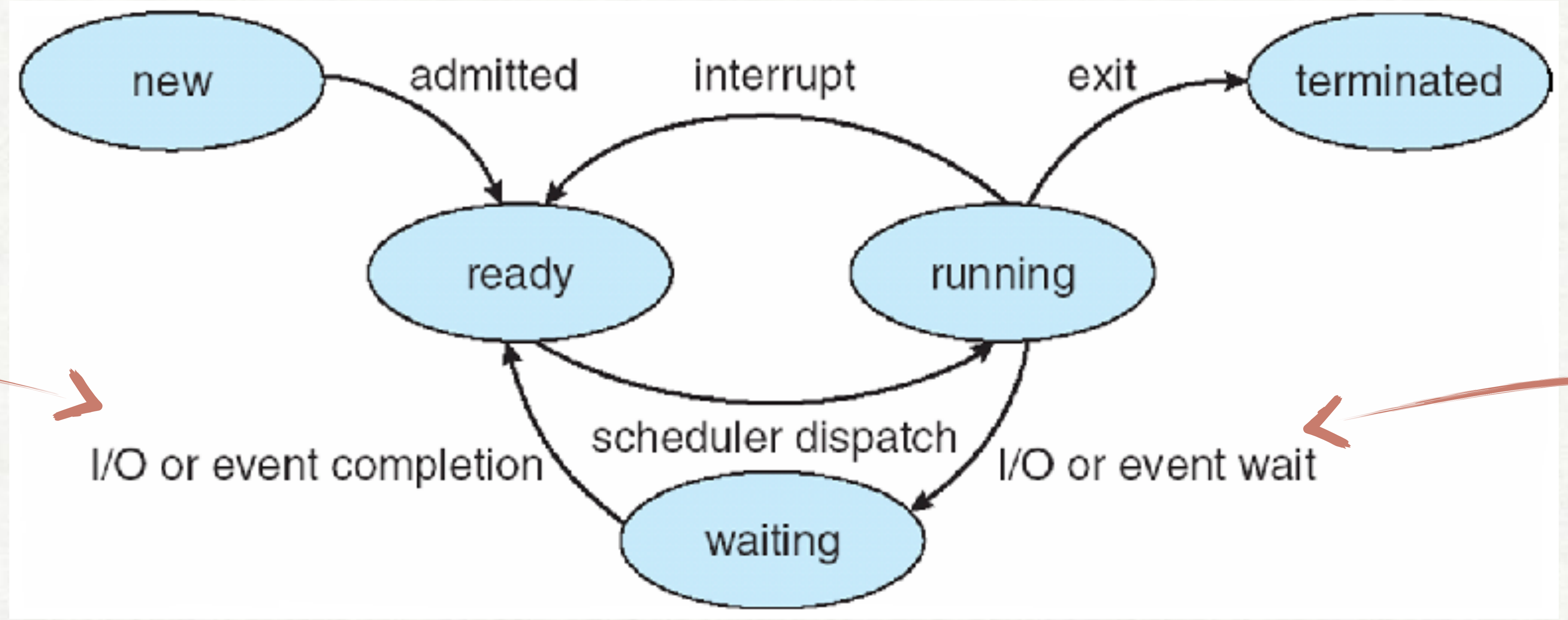
STILL **BUSY WAIT** WITH SPINLOCKS, BUT CAN WE DO WITHOUT?

- Different types:
  - counting semaphore
  - binary semaphore (vs. mutex lock)
- Different uses:
  - used as mutex locks
  - used for signaling
  - rendezvous, etc.

```
do {
    wait(S)
        critical section
    signal(S)
        remainder section
} while (true);
```

SEMAPHORES WITHOUT BUSY-WAIT

SYNCHRONIZATION

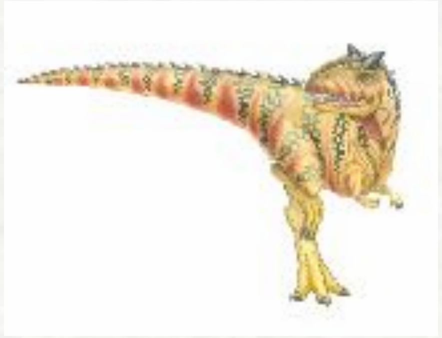


- S gets more complex: also manages a queue of blocked processes
- **wait:** if needed, P adds itself to semaphore queue and block
- **signal:** P wakes-up a blocked process and place it in OS ready-queue

CHALLENGES WITH SEMAPHORES

- **incorrect use**
- **deadlock**: wait for each other
  - ✓ prevent, avoid, detect, recover
- **starvation**: indefinite blocking
- **priority inversion**: lower-priority before higher-priority
  - ✓ priority-inheritance protocol

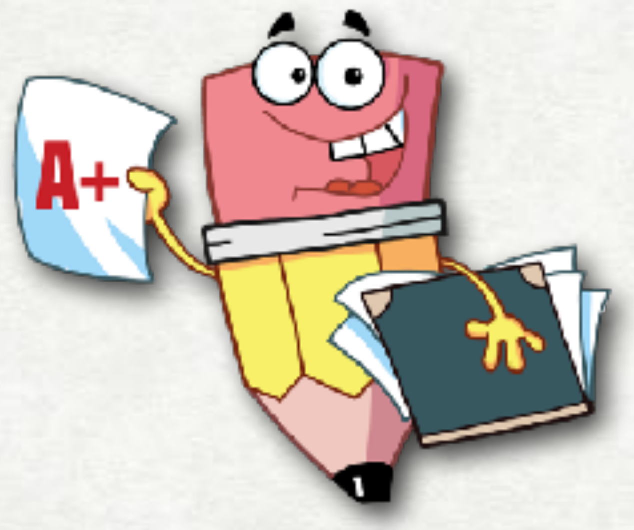
READ THE WHOLE CHAPTER 7: DEADLOCKS



CLASSIC PROBLEMS
SYNCHRONIZATION

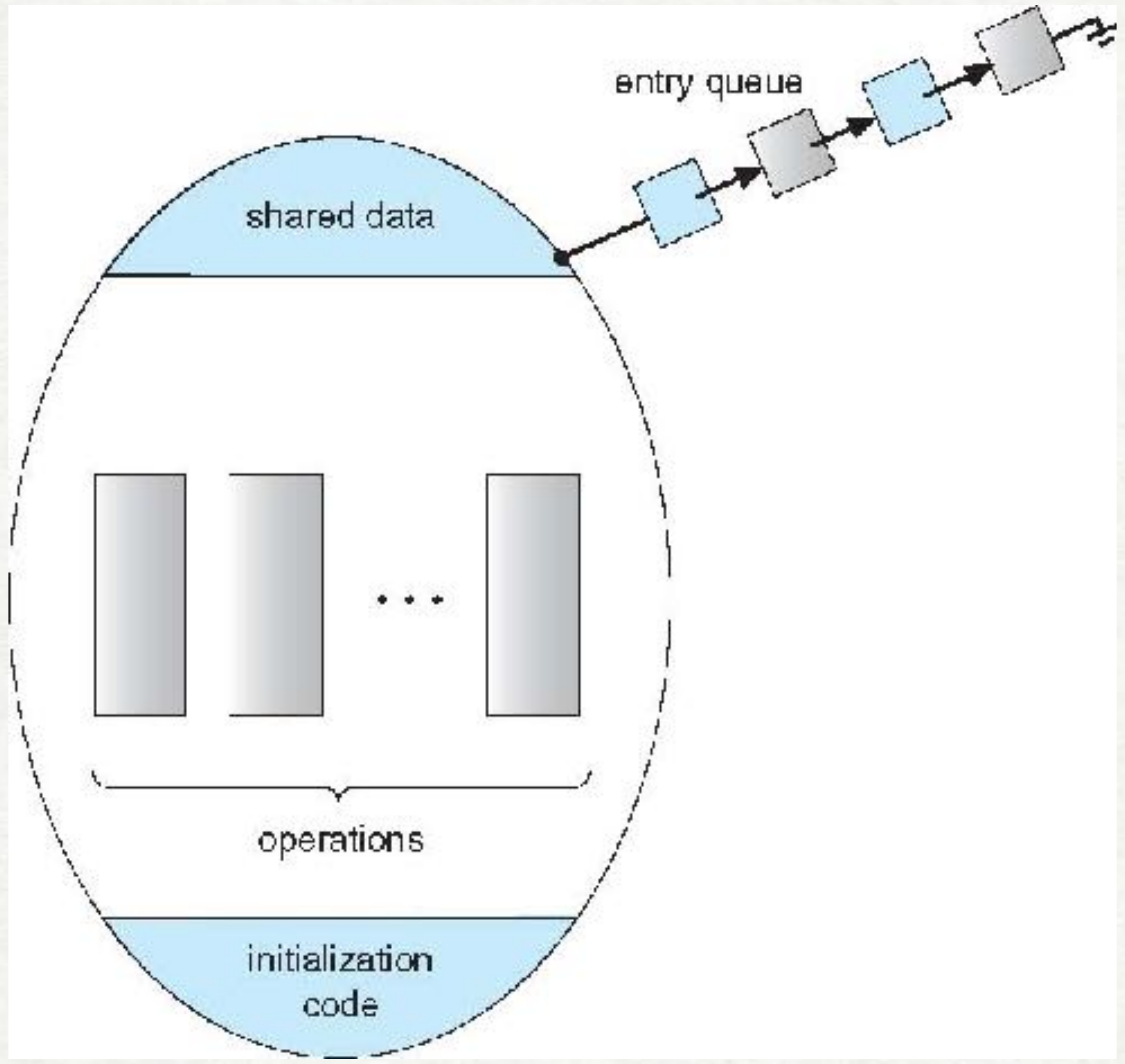
- The Bounded-Buffer Problem
- The Readers-Writers Problem
- The Dining-Philosophers Problem

READ THE BOOK
6.7 (OR 5.7)



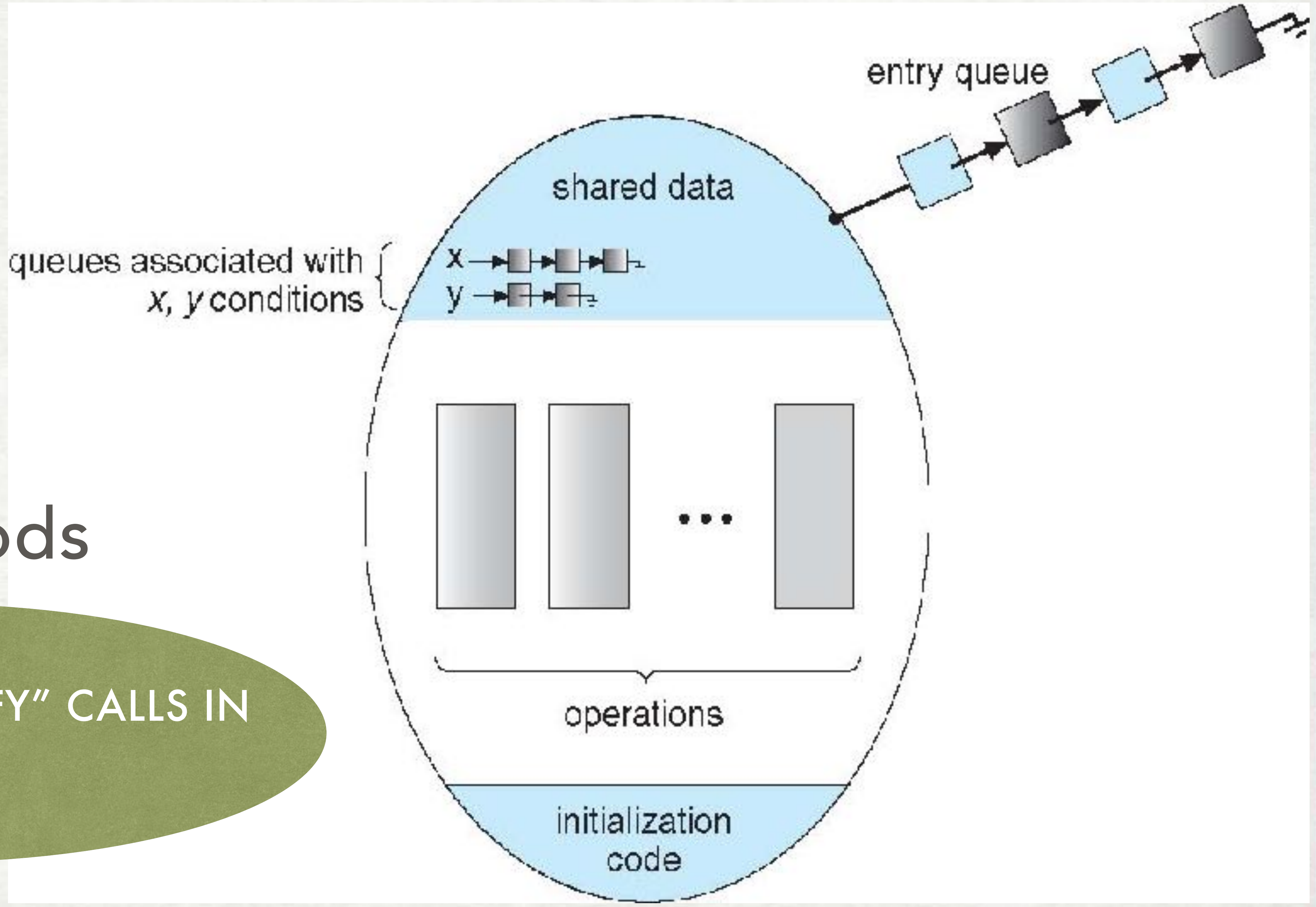
MONITORS

SYNCHRONIZATION



"SYNCHRONIZED" METHODS IN JAVA

- encapsulate shared resources/critical sections
- one process active in any of its methods ("mutex")



- with **condition variables**: more complex schemes
  - `c.wait(), c.signal()` - from monitor methods
  - similar, but not the same as the semaphore operations (how?)

"WAIT" AND "NOTIFY" CALLS IN JAVA

QUICK EXAMPLES

SYNCHRONIZATION

solaris

- adaptive mutexes
- condition variables
- readers-writers lock

Windows

- disable interrupts (uniproc) or spinlock (multiproc)
- dispatcher objects: mutex, semaphore, events, timers
- events = condition variables

Linux

- non-preemptive kernel < v2.6 (fully preemptive)
- semaphores, atomic integers, spinlocks, reader-writer locks

pthreads

- OS-independent API
- mutex lock, condition variables
- non-portable extensions: rw-locks, spinlocks

ALTERNATIVE APPROACHES

SYNCHRONIZATION

- Transactional Memory
- OpenMP
- Functional Languages
- Asynchronous Computation Models

END OF MODULE 5.B