ECS518U - Operating Systems
Week 11

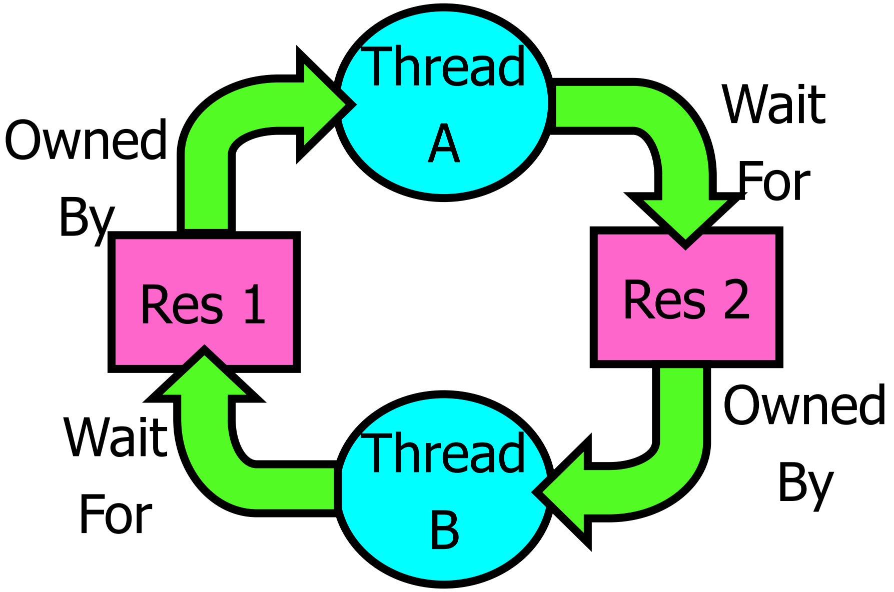
Concurrency Primitives, Resource Deadlock, Java Memory Model

Outline

- **Concurrency and the OS**
  - Deadlock conditions, dealing with deadlocks
- **Java Memory Model**
- **Primitives:** implementing locks
- Alternative concurrency abstractions
  - **Semaphores**
  - **Monitors**

Deadlock in the OS

- The situation described in Dining Philosophers can generalise in OSs where threads / processes compete for resources
- **Example:** System with 2 disk drives and two threads
  - Each thread needs 2 disk drives to function
  - Each thread gets one disk and waits for another one

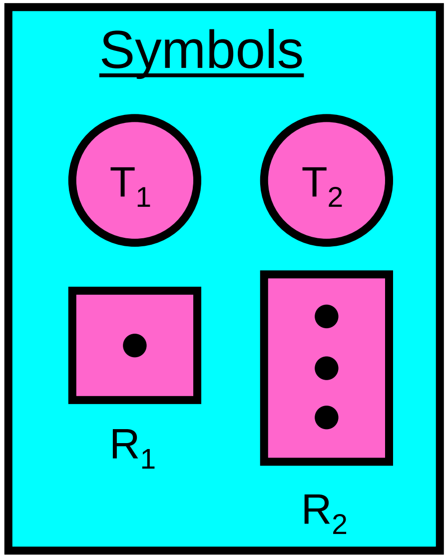


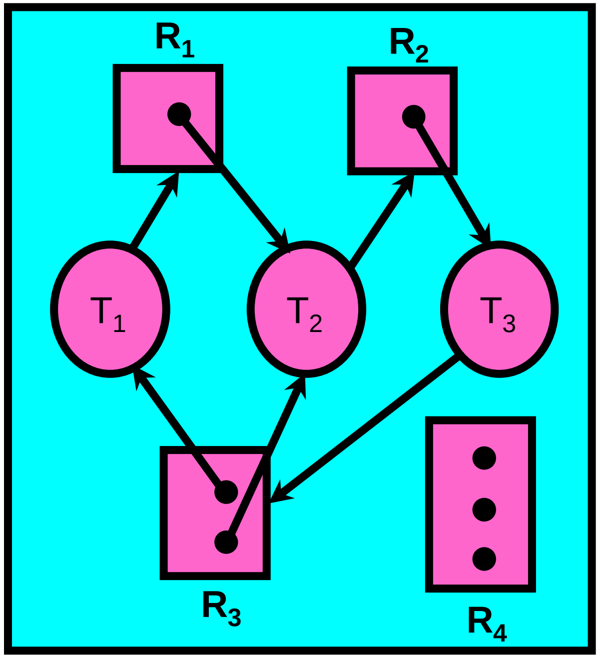
Conditions for (Resource) Deadlocks

- **These 4 conditions are necessary for a deadlock to occur (all 4 must be present)**
- **Mutual exclusion** condition
  - The resources are unshareable (only one thread/process at a time can use the resource)
- **Hold and wait** condition
  - Hold one resource while waiting for other(s)
- **No preemption** condition
  - Resource can only be given up voluntarily after the thread/process finished using it (i.e. can not be taken away)
- **Circular wait** condition
  - A circular chain, with each thread/process holding resources which are currently being requested by the next thread/process in the chain

Deadlock Modeling

- **Allocation graph**
  - A set of Threads $T_1, T_2, \ldots, T_n$
  - Resource types $R_1, R_2, \ldots, R_m$
    CPU cycles, memory space, I/O devices, …
  - Each resource type R_i can have many instances (e.g. 5 I/O devices)
  - Each thread utilises a resource as follows:
    - `Request() / Use() / Release()`
  - **Request edge** – directed edge $T_1 \rightarrow R_j$
  - **assignment edge** – directed edge $R_j \rightarrow T_i$





Allocation Graph with Deadlock

Strategies for Dealing with Deadlock

- **Detection and recovery**
  - Let deadlocks occur, detect them, take action
- **Dynamic avoidance** by careful resource allocation
  - Do not allocate resources that will lead to deadlock
  - Need to monitor all lock acquisitions
  - Selectively deny those that might lead to deadlock
- **Prevention**, by structurally negating one of the four required conditions
- (Ostrich: head in sand and pretend that deadlocks never occur)
  - Used by most OSs including UNIX

Recovery from Deadlock

- Recovery through **preemption**
  - Take one of the resources away
  - Not always possible as it depends on the nature of the resource and/or the task – it is frequently impossible
- Recovery through **rollback**
  - Bring the system back in time in some stable condition – roll back actions of deadlocked threads / processes
  - Common technique in DBs (transactions)
  - Of course, if you restart in exactly the same way, may reenter deadlock once again
- Recovery through **killing processes**
  - Choose a ‘victim’ wisely
    - e.g a dining philosopher (!!!)
  - Used in many OSs – up to the user

Deadlock Prevention

- **Attacking one of the conditions**

Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

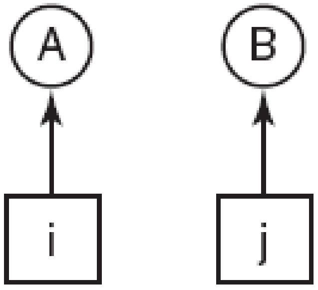
Example: Attacking the Circular Wait Condition

(a) Numerically ordered resources

(b) A resource graph

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD-ROM drive

(a)



(b)

In the olden days… (single CPU)

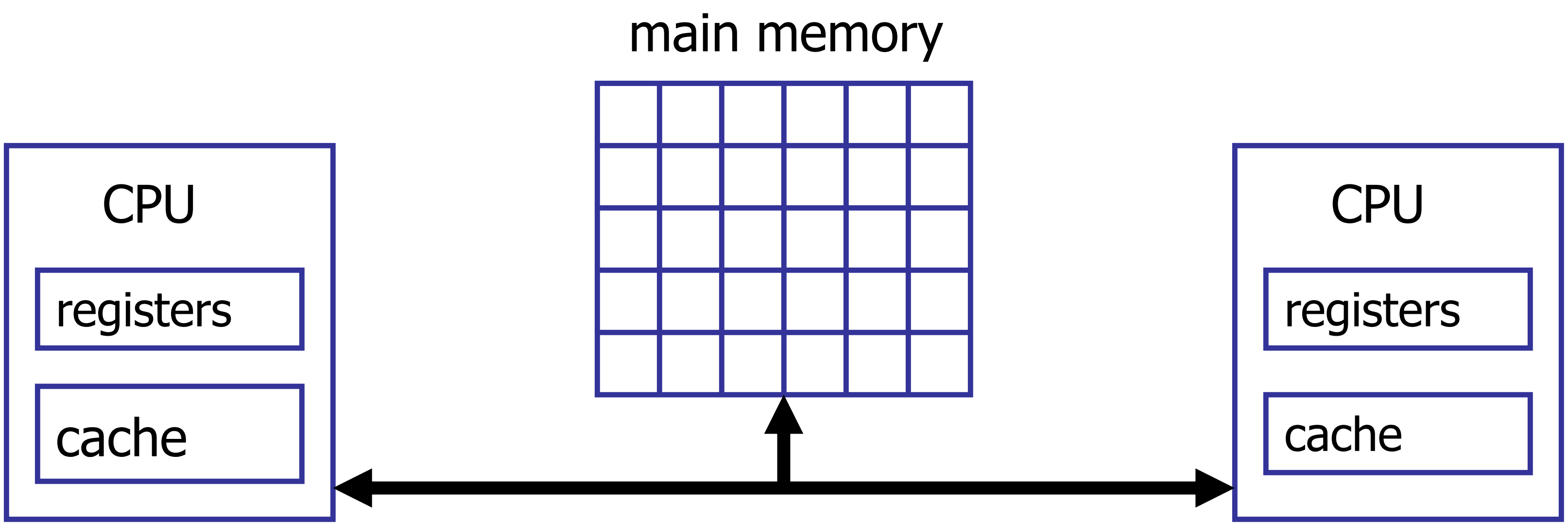
- You think: *code executed in order*
- **Reality:** compiler may re-order statements to reduce memory access delay (but you would not be able to tell the difference)
- You think: *variables updated in memory*
- **Reality:** compiler may use a register (but you would not be able to tell the difference)
- You think: *code executed sequentially*
- **Reality:** processor may have parallel pipelines (but you would not be able to tell the difference)

Java Memory Model

- What guaranteed behaviour do we have when using threads in Java?
  - If single-threaded program, no different than the olden days…
- **Interleaving – no guarantees!**
  - Methods (sequence of statements) not atomic
  - Changes by one thread interfere with changes by another
- **Visibility of memory updates – no guarantees!**
  - Compiler may eliminate / delay memory writes
  - No visible difference to *sequential* programs
  - … but concurrent programs could see differences
- If we had maintained all the 'old' guarantees, we would not have been able to effectively use multiple cores…

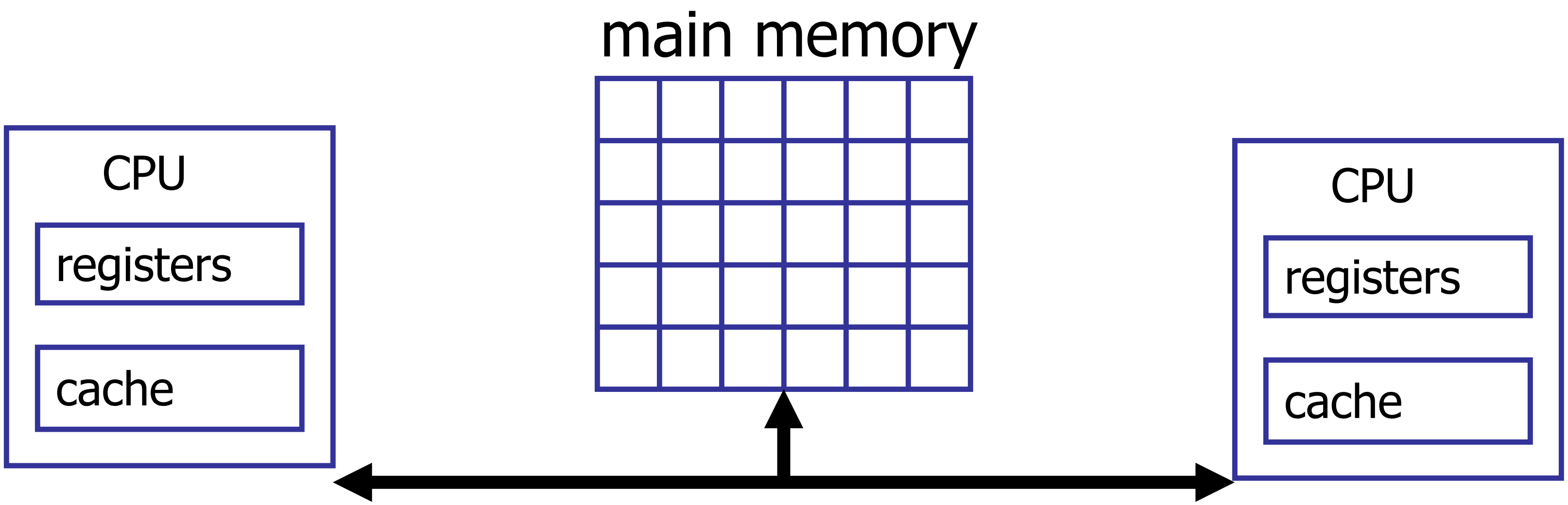
Java Memory Model

- **Shared memory multi-core system**
  - "Worst" case: threads on different processors, changes in 'local' caches/registers not visible in other CPUs
  - Using local cache and registers is faster, so it is practical



- **Atomicity**: what is indivisible
- **Visibility:** (between threads)
- **Ordering**: as written or different?

Java Memory Model ‘guarantees’



- **Atomic**
  - Synchronised methods / blocks execute atomically (no other thread comes in and executes code while a thread has the lock)
- **Visibility** (between threads)
  - Changes made inside synchronised methods are kept consistent between multiple threads
  - Otherwise, changes will be sooner or later visible, but you don’t know when
- **Ordering:** as written or different?
  - One thread: as written
  - Between threads: order of synchronised blocks is guaranteed (i.e. one thread at a time)

Primitives & Relationship to Wait/Notify

- Java has support for concurrent programming
- Other languages (e.g. C) use system calls

Programs	Shared Programs		
Higher-level API	Locks	Semaphores	Monitors
Hardware	Load/Store	Disable Ints	Test&Set Comp&Swap

- The OS depends on primitive instructions provided by the CPU (h/w level)

Key Concepts

- CPU hardware provides primitives to ensure mutual exclusion
- Different concurrency abstractions exist
- Java library has higher-level alternatives
- Modern OSs have/support kernel threads
- The OS itself is a concurrent program

Principles

- Concurrency is all about interleaving
- For mutual exclusion (i.e. create locks) we must restrict interleaving
  - With single core, we could do that by controlling the scheduler and interrupts
  - This is harder to do with multi-core systems (you could have only some control over the CPU the thread is running on)

Implementing Mutual Exclusion I

- **Interrupt Disabling**
  - A process runs until it invokes an OS service or until it is interrupted
  - Disabling interrupts guarantees mutual exclusion
- Error prone
- Limits responsiveness of scheduler
- Do we really want to let user processes control interrupts?

- OK within kernel on a uni-processor
- Does not work on a multi-core system
  - Disabling interrupts on all processors requires messages and would be very time consuming

Implementing Mutual Exclusion I

- **Naïve implementation of locks with interrupts**
  - LockAcquire() {disable Interrupts;}
  - LockRelease() {enable Interrupts;}
- Problems with this approach:
  - **Can't let user do this!** Consider following:

    ```
    LockAcquire();
    While(TRUE) {;}
    ```
  - Real-Time system—no guarantees on timing!
    - Critical Sections might be arbitrarily long
  - What happens with I/O or other important events?
    - "Plane about to stall... Help?"

Better Implementation of Locks by Disabling Interrupts

- **Key idea:** maintain a lock variable and impose mutual exclusion only during operations on that variable

`int value = FREE;` (the lock variable)

```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

```
Release() {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   enable interrupts;
}
```

New Lock Implementation: Discussion

- Why do we need to disable interrupts at all?
  - **Avoid interruption between checking and setting lock value**
  - Otherwise two threads could think that they both have lock

```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

Critical Section

- Note: unlike previous solution, **the critical section (inside Acquire()) is very short**
  - User of lock can take as long as they like in their own critical section: doesn't impact global machine behavior

Implementing Mutual Exclusion II

- **Alternative to disabling interrupts**
  - Hardware is responsible for implementing this correctly
  - Unlike disabling interrupts, can be used on both uniprocessors and multiprocessors
- **Atomic** Machine Instructions
  - Performed in a single instruction cycle
  - Combine load and store: Test & set, Compare & swap and more

```
test&set (&address) {          /* most architectures */
    result = M[address];
    M[address] = 1;
    return result;
}
```

Implementing Locks with test & set

- A simple solution:

```
int value = 0; // Free
Acquire() {
   while (test&set(value)); // while busy
}
Release() {
   value = 0;
}
```

- Simple explanation:
  - If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0, so while exits.
  - If lock is busy, test&set reads 1 and sets value=1 (no change). It returns 1, so while loop continues
  - When we set value = 0, someone else can get lock
- **Busy-Waiting:** thread consumes cycles while waiting

Higher Level Abstractions
(alternatives to the Java wait/notify abstraction)

- Interface between user program and OS
- Mutual exclusion between processes or threads

- **Semaphores**
  - Counting semaphore
  - Binary semaphore
- **Monitors**
  - In theory
  - In Java

Binary Semaphore

- Semaphore initialised to 1 or 0
- Two operations
  - **Wait** (P() – proberen in Dutch - test)
  - **Signal** (V() – verhogen in Dutch - increment)
- Wait
  - Blocks if semaphore = 0 → queue
  - If semaphore = 1, assigns 0 and proceeds
- Signal
  - Unblocks a queued process, if any
  - If no queued process, assigns 1

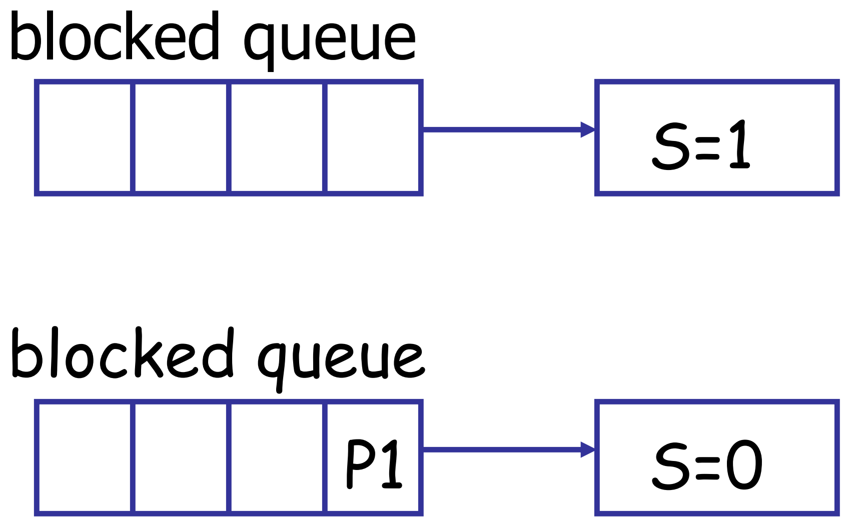
Critical Region Using Semaphore

- Critical region protected by a semaphore s

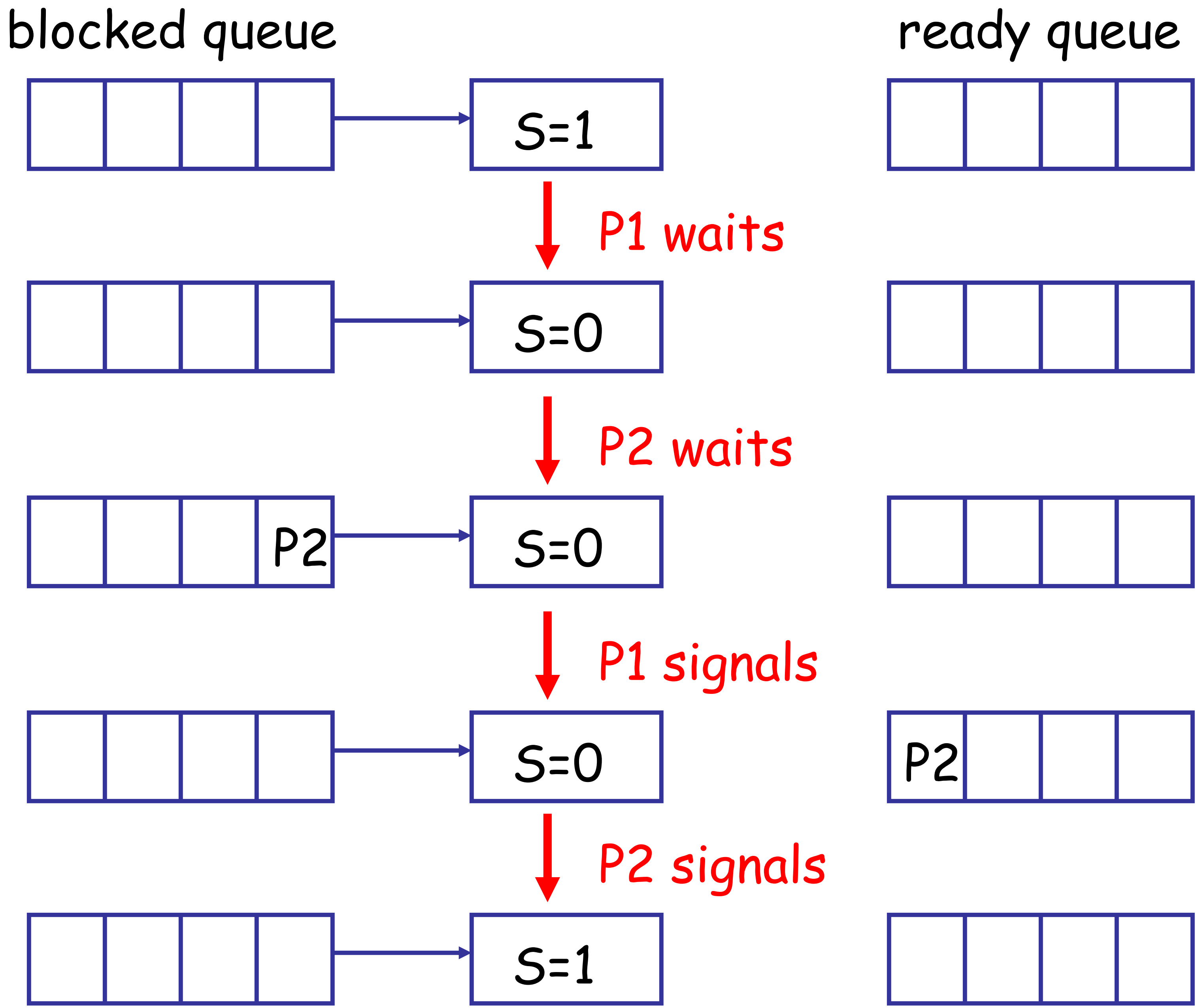
```
// enter the critical region
wait(s) ;
// in critical region
// leave the critical region
signal(s) ;
```

Semaphore Implementation

- Queue and binary variable



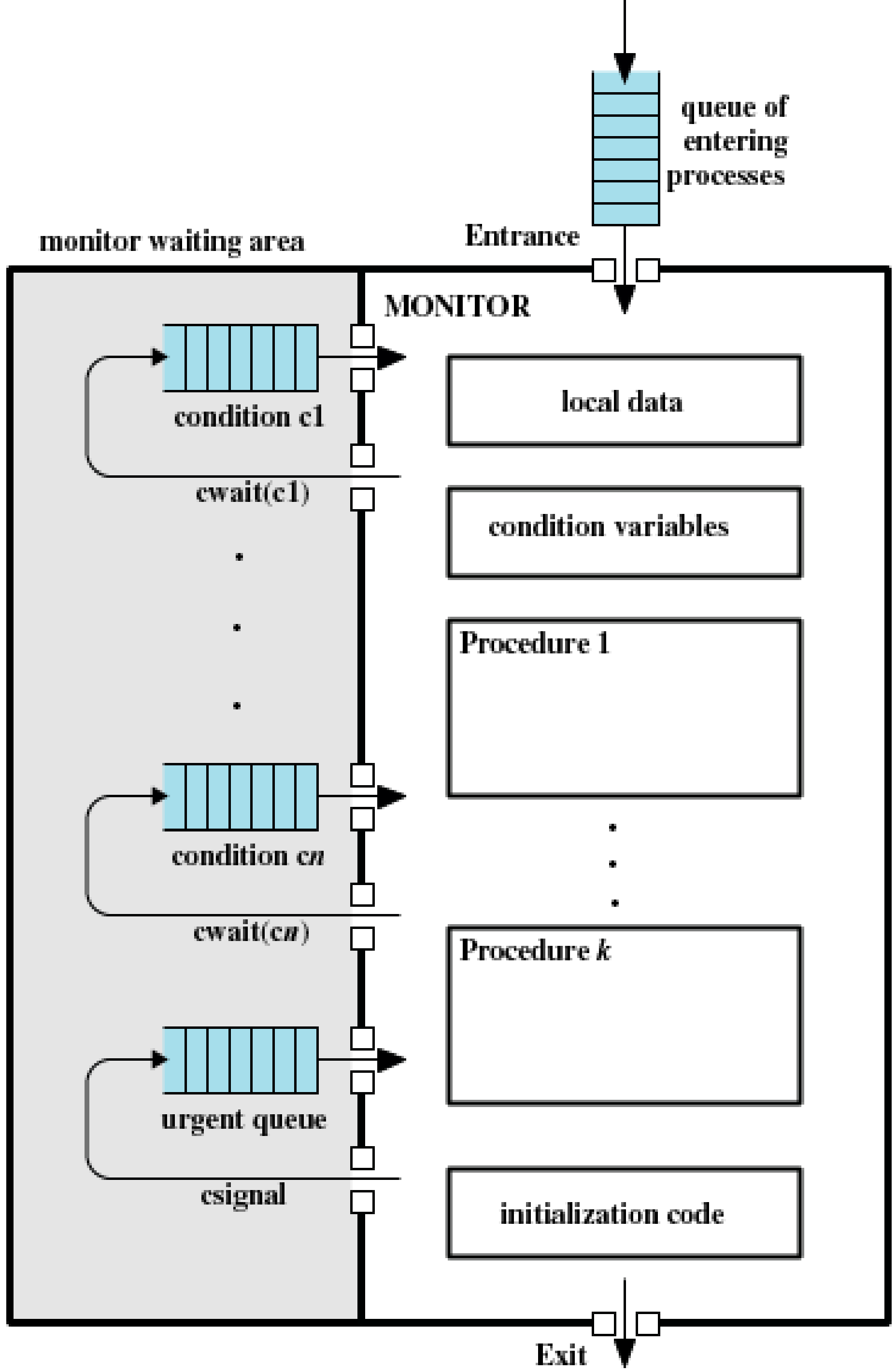
Semaphore Operation



Monitors

Monitor: a lock and zero or more condition variables for managing concurrent access to shared data

- Only one process/thread 'in' monitor at time
- Waiting area
  - Wait / Notify
  - Associated with a condition
- Java
  - Only 1 condition / queue
  - Use with care



Semaphore versus Monitor

- Semaphore in pairs: wait / signal
  - Wait but no signal → deadlock
  - Signal w/o wait → loss of mutual exclusion
- Monitors more structured
  - Implicit wait / signal pair
  - E.g. exception in Java synchronized blocks
- Monitor semantics more complex
  - When does a notified method resume?
  - In Java, ALWAYS call wait in a loop.

Summary

- Concurrency in Operating systems
  - Earliest concurrent programs, now multi-core
  - Resource **deadlocks**: conditions, detection, avoidance, ..
- Concurrency in user programs
  - **Java Memory model** & guarantees
- **Primitive** instructions for lock implementation on the h/w (CPU) level
- **Monitors** / **semaphores** as alternative higher level abstractions
- In practice use util.concurrent (separate set of slides will be available on QMPlus FYI)