Process coordination 3

CS503: Operating systems, Spring 2019

Pedro Fonseca
Department of Computer Science
Purdue University

Previous lecture(s)

- Coordination is critical in concurrent systems
  - Prevents incorrect semantics
  - Enables efficient implementation of producer-consumer
- Synchronization primitives:
  - Interrupt disabling
  - Spin locks
  - Counting semaphores:
    - Two key fields (count and blocked process list)
    - Prevent busy wait

Previous lecture(s)

- Concurrency patterns / problems:
  - Producer-consumer pattern
  - Reader-writer pattern
    - Read-write locks
- More synchronization primitives:
  - Conditional variables

Reading

- R/W locks:
  - *Ch. 5.6.1: "Readers/Writers lock"* of Operating Systems: Principles and Practice, Second Edition, by Thomas Anderson and Michael Dahlin
  - Ch 31.5: "Reader-writer locks" of Operating Systems: Three Easy Pieces, by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau
    http://pages.cs.wisc.edu/~remzi/OSTEP/threads-sema.pdf
- Conditional variables:
  - *Ch. 30: "Conditional variables"* of Operating Systems: Three Easy Pieces, by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau
    http://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf

Exam note about synchronization exercise

- Solutions should be correct, concise, readable and efficient
- No points or fewer points for:
  - Large solutions
  - Confusing or difficult-to-read solutions
  - Excessive scheduling constraints
  - Excessive use of sync. primitives

Conditional variable

- It is a synchronization object that:
  - Lets a thread efficiently wait for a change to shared state that is **protected by a lock**

Recall: Conditional variables

- When a thread calls **cond_wait**:
  - The caller is:
    - (**always**) put into the CV queue
    - releases the CV mutex
- When a thread calls **cond_signal:**
  - If there are threads waiting in that CV queue, one of them is released
  - Otherwise, the CV signal is lost
- Only call CV operations when holding the mutex

```
mutex_t   MonitorLock;
cond_t    CondVar;


// Thread 1
mutex_lock(&MonitorLock);
while (!cond)
  cond_wait(&CondVar, &MonitorLock);
mutex_unlock(&MonitorLock);


// Thread 2
mutex_lock(&MonitorLock);
if (cond)
  cond_signal(&CondVar);
mutex_unlock(&MonitorLock);
```

Recall: Conditional variables (revised with example)

- When a thread calls **cond_wait**:
  - The caller is:
    - (**always**) put into the CV queue
    - releases the CV mutex
- When a thread calls **cond_signal:**
  - If there are threads waiting in that CV queue, one of them is released
  - Otherwise, the CV signal is lost
- Only call CV operations when holding the mutex

```
mutex_t   MonitorLock;
cond_t    HasItems;


// Thread 1 (e.g., part of consumer)
mutex_lock(&MonitorLock);
while (<List empty>)
  cond_wait(&HasItems, &MonitorLock);
<remove list item>
mutex_unlock(&MonitorLock);


// Thread 2 (e.g., part of producer)
mutex_lock(&MonitorLock);
<add list item>
if (!<list empty>) // not stric. necessary here
  cond_signal(&HasItems);
mutex_unlock(&MonitorLock);
```

Exercise:
Reader-writer with conditional variables

Read-writer lock using conditional variables

```
mutex m;
cond_var unlocked;

bool writer = false;
int readers = 0;
```

```
write_lock() {
  mutex.lock();
  while (writer || (readers > 0))
    unlocked.wait(mutex);
  writer = true;
  mutex.unlock();
}

write_unlock() {
  mutex.lock();
  writer = false;
  unlocked.signal_all();
  mutex.unlock();
}
```

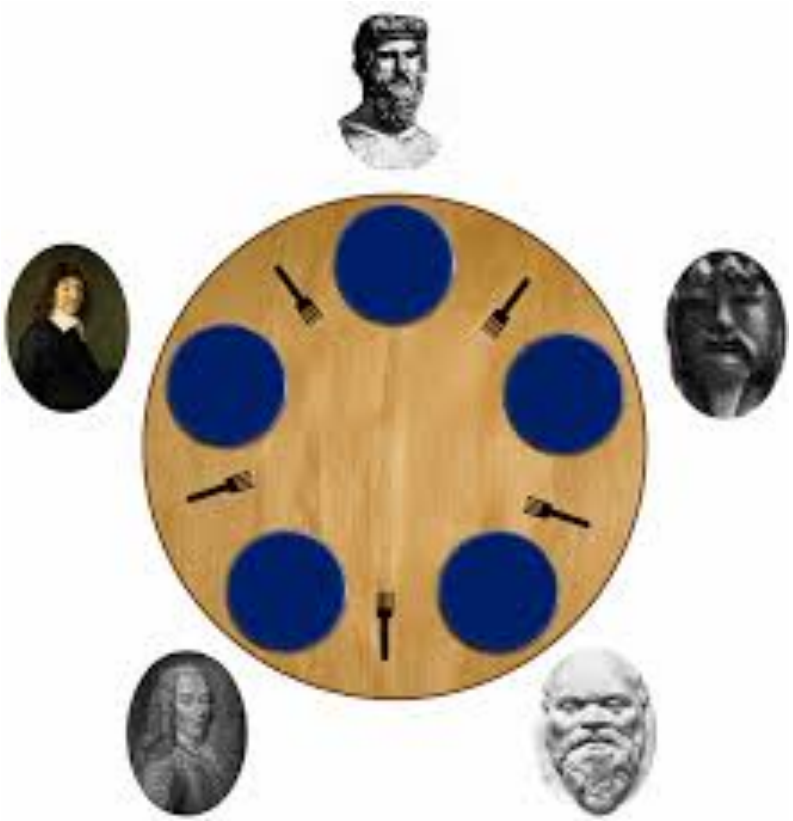
```
read_lock() {
  mutex.lock();
  while (writer)
    unlocked.wait(mutex);
  readers++;
  mutex.unlock();
}

read_unlock() {
  mutex.lock();
  readers--;
  if (readers == 0)
    unlocked.signal_all();
  mutex.unlock();
}
```

Conditional variables

- In addition to signal() and wait(), sometimes CV objects have a **broadcast()** operation:
  - It wakes up all waiting threads
- Conditional variables have no memory (just a queue of waiting threads)

Dining philosophers problem



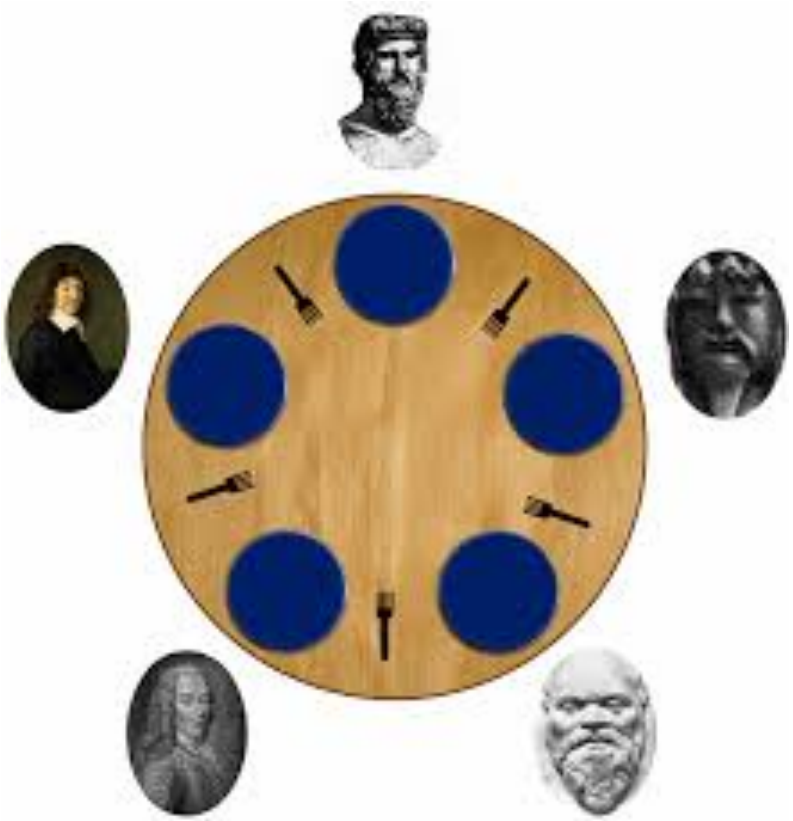
- Each philosopher alternates between **thinking** and **eating pasta**
- **Eating:** pasta requires that the philosopher picks up **two forks** (left and right) and only then can eat
- **Thinking**: After eating some pasta the philosopher puts back the forks on the table and thinks

What should be the algorithm for each philosopher so that they all remain happy (eating and thinking as fast as possible)?

Modeling this problem in C

- Each philosopher can be modeled as a thread
- Forks are shared resources:
  - Possibly with locks associated

Dining philosophers problem



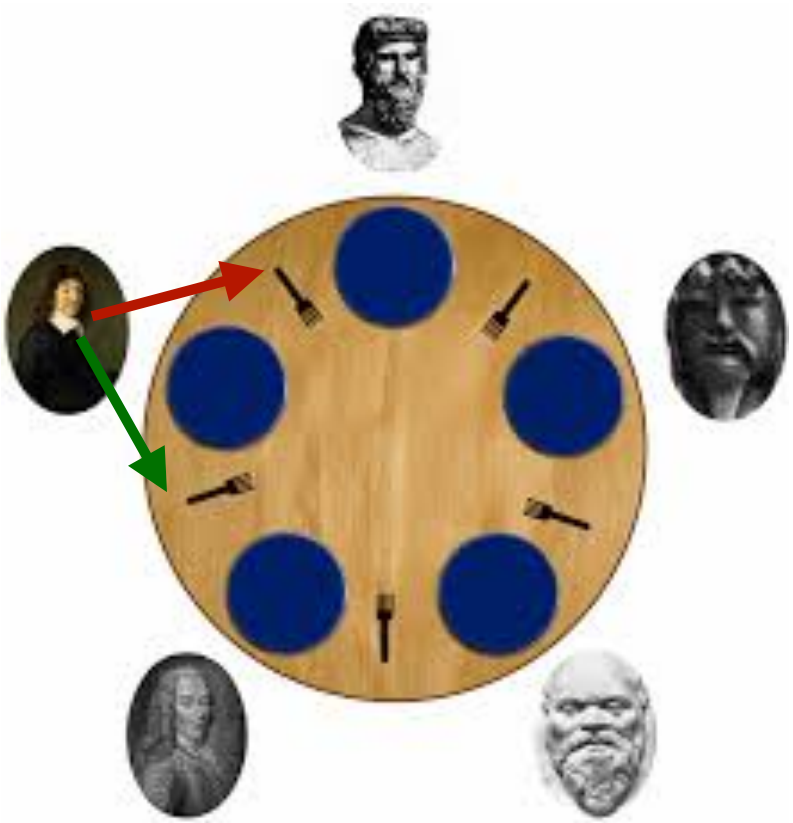
- Each philosopher alternates between **thinking** and **eating pasta**
- **Eating pasta** requires that philosophers pick up **two forks** (left and right) and only then can eat
- After eating some pasta philosophers put back the forks on the table and **think**

What should be the algorithm for each philosopher so that they all remain happy (eating and thinking as fast as possible)?

One algorithm

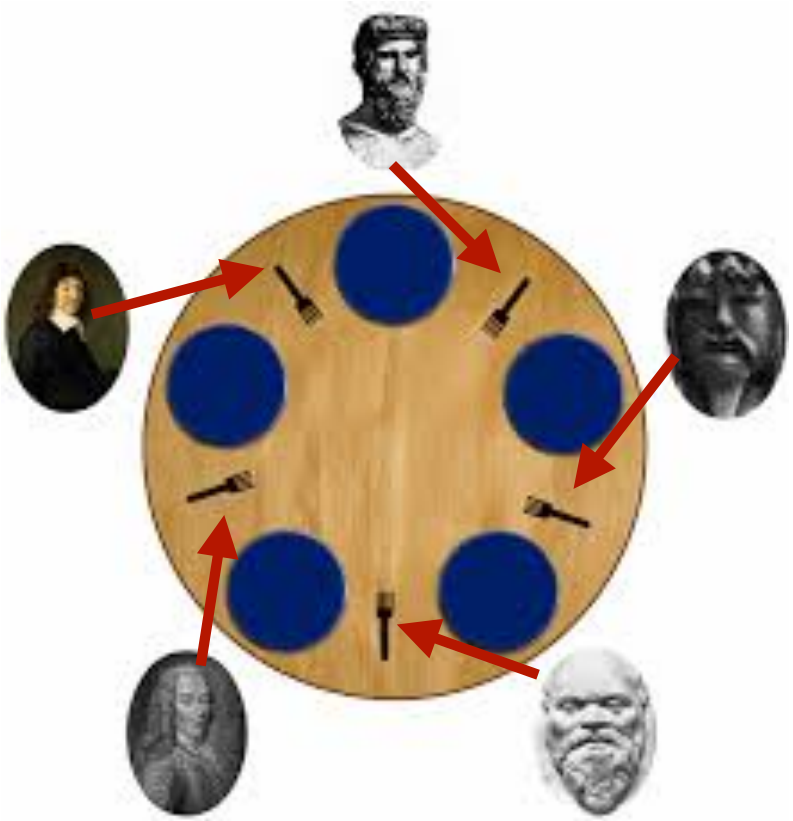
- Each philosopher could:
  - think until the left fork is available; when it is, pick it up
  - think until the right fork is available; when it is, pick it up
  - when both forks are held, eat for a fixed amount of time
  - then, put the right fork down
  - then, put the left fork down
  - repeat from the beginning

One algorithm



- Each philosopher could:
  - think until the left fork is available; when it is, pick it up
  - think until the right fork is available; when it is, pick it up
  - when both forks are held, eat for a fixed amount of time
  - then, put the right fork down
  - then, put the left fork down
  - repeat from the beginning

One algorithm

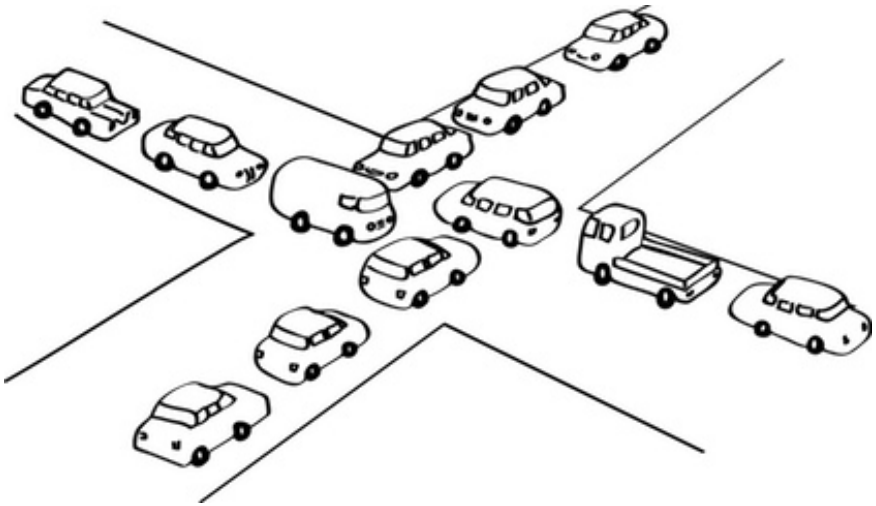


- Each philosopher could:
  - think until the left fork is available; when it is, pick it up
  - think until the right fork is available; when it is, pick it up
  - when both forks are held, eat for a fixed amount of time
  - then, put the right fork down
  - then, put the left fork down
  - repeat from the beginning

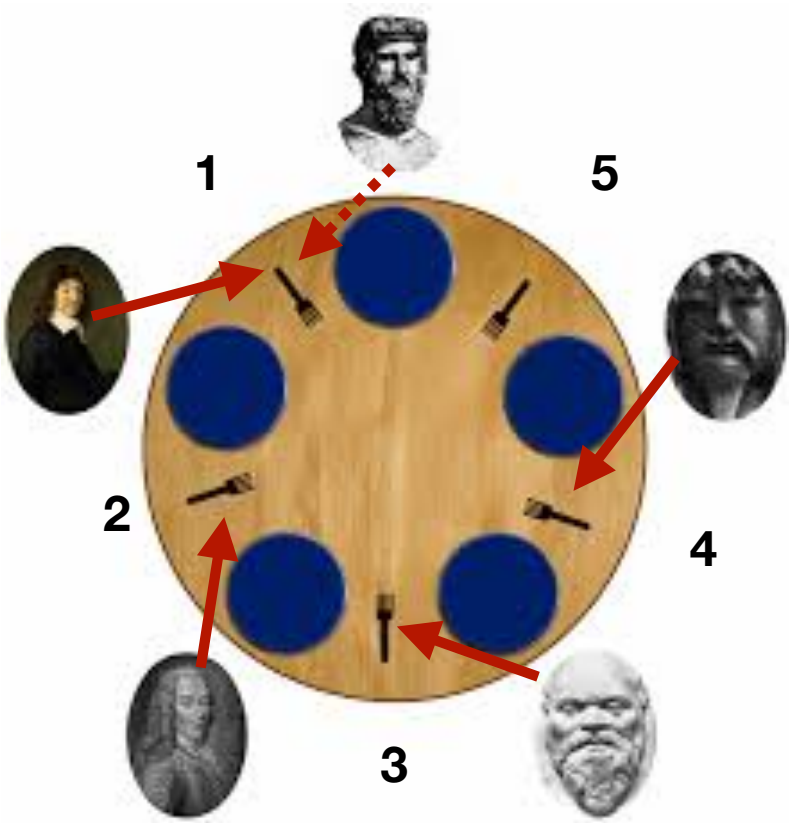
What if all philosophers reach out to the left fork at the same time?

Deadlock

- Circular dependency on shared resources held by different threads
- In the case of two threads:
  - Thread A is waiting for the resource held by Thread B
  - But Thread B is also waiting for a shared held by Thread A
- None of the deadlocked threads can make progress



Solution: Ranking shared resources



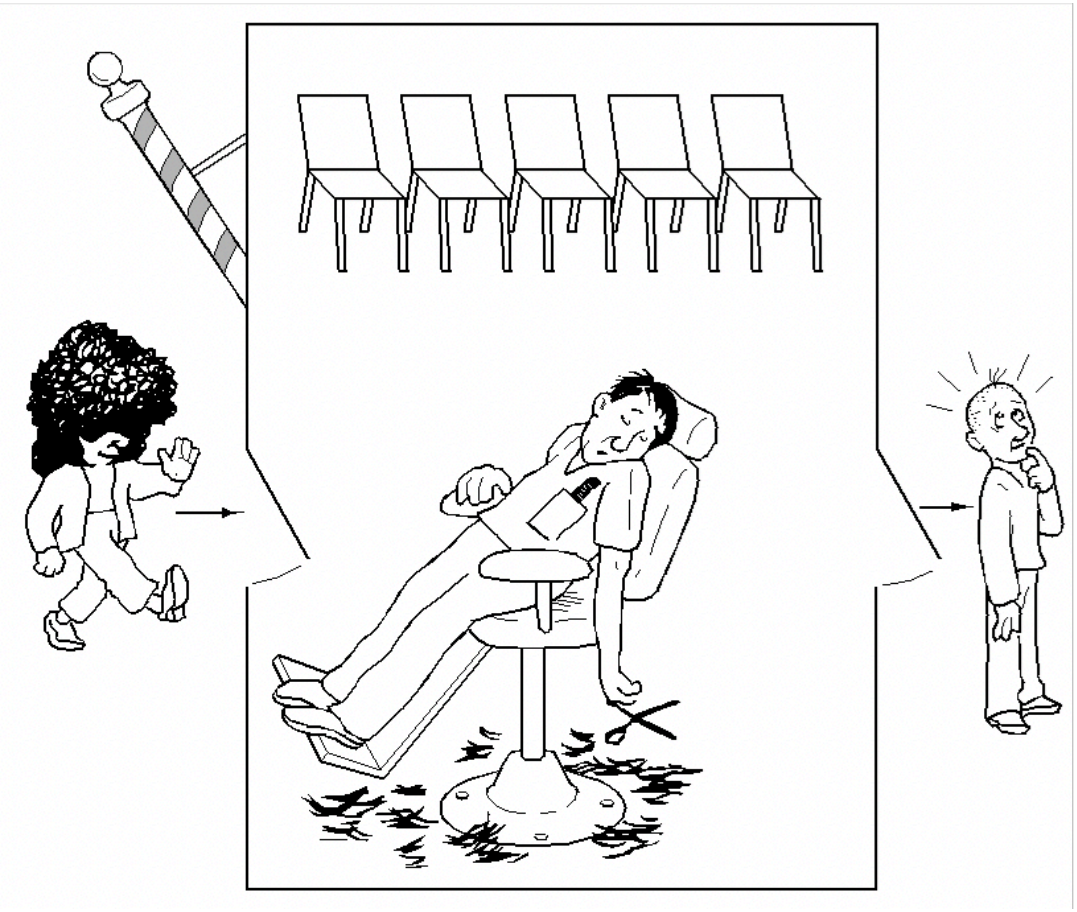
- Rank forks
- Each philosopher could:
  - think until his **highest-ranked fork** is available; when it is, pick it up
  - think until his **lowest-ranked** fork is available; when it is, pick it up
  - when both forks are held, eat for a fixed amount of time
  - [same]

Solution: Ensures there are no circular dependencies

Solutions

- Ranking shared resources
- Using an arbitrator
  - Requires centralization
  - Reduces parallelism
- Trying locks

Sleeping barber



Sleeping barber

- Barber:
  - If there are people waiting for a hair cut bring them to the barber chair, and give them a haircut
  - Else go to sleep
- Customer:
  - If the waiting chairs are all full, then leave store
  - If someone is getting a haircut, then wait for the barber to free up by sitting in a chair
  - If the barber is sleeping, then wake him up and get a haircut

Resource starvation

- **Livelock**:
  - Threads run but they are not able to make **progress**
    - i.e., not blocked but not doing anything useful (i.e., repeatedly)
  - Different than deadlock
- Deadlock and livelock are extreme cases of resource starvation

(optional) homework:
Solve the sleeping barber with semaphores

Race (condition)

- When two or more concurrent computations (e.g., thread, processes) compete for a resource

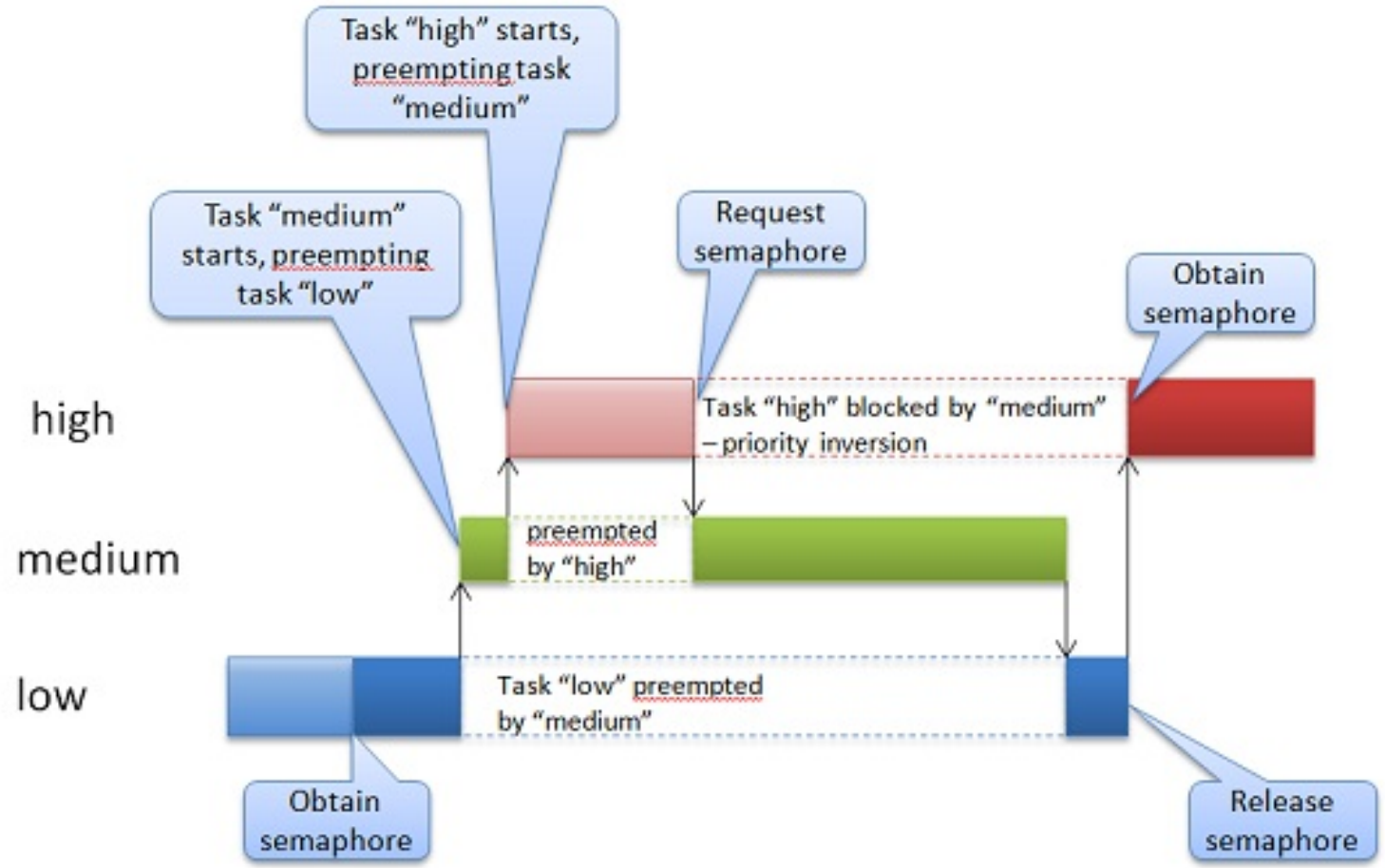
Data race

- Special type of race
- Several requirements:
  - Need to be **racing memory accesses**,
    - i.e., can happen at the same time
  - **Conflicting**
    - i.e., at least one of them is a write
  - For **data** memory accesses
    - what does this mean?
- Synchronization accesses (e.g. accesses made by lock implementations) are not data accesses
- Occurs when synchronization is not used or not used correctly

XINU semaphores queue policy

- First-come-first-serve
- Straightforward to implement
- Extremely efficient
- Works well for traditional uses of semaphores
- Potential problem: **Priority inversion**
  - Low-priority process can access a resource while a high-priority process remains blocked
  - Learn more from: Mars Pathfinder's story:
    - https://www.rapitasystems.com/blog/what-really-happened-to-the-software-on-the-mars-pathfinder-spacecraft

Priority inversion



Some solutions: priority inheritance, disabling interrupts, etc.

Xinu

- semaphore policy
- semcreate(), semdelete()
- wait(), signal(), signaln()

Summary

- Conditional variables
- Dining philosopher problem
- Sleeping barber problem
- Deadlocks, livelocks, starvation
- Priority inversion
- XINU semaphores