# Process coordination 2

CS503: Operating systems, Spring 2019

Pedro Fonseca
Department of Computer Science
Purdue University

## Admin

- Lab 1 is out
- **Due date: 02/18/2019 (Mon.), 11:59 PM**

## Previous lecture

- Coordination is critical in concurrent systems
    - Prevents incorrect semantics
    - Enables efficient implementation of producer-consumer
- Terminology: synchronization
- Mutual exclusion mechanisms to protect critical sections
- Spin locks
- Counting semaphores:
    - Two key fields (count and blocked process list)
    - Prevent busy wait
    - More general than mutual exclusion

## Terminology

- Multiprocessing: Multiple CPUs, cores, or hyperthreads
- Multiprogramming: Multiple jobs or processes
- Multithreading: Multiple threads per process

## Debugging

- Bugs associated with locks are interleaving / time dependent
  - In addition to input dependent
- Can be very subtle and extremely hard to replicate, such as deadlocks
- Lots of research in this field

## Synchronization primitives

- Spinlocks: implement mutual exclusion
- Semaphores: avoid busy wait, more general than locks
- Conditional variables: wait on a condition

## Properties

- E.g., for locks:
    - **Correctness**: Ensure mutual exclusions
    - **Fairness**: Do every process
    - **Performance**: How fast are locks acquired and released
    - Other properties?

## Locks usage

- Granularity
- Hierarchical usage of locks

## Read-writer pattern

- 1 or more readers, 1 or more writers
- Concurrency contraints:
  - Multiple readers are allowed concurrent access
    - Less constrained than mutual exclusion
  - Writers are not allowed concurrent access (WR and WW)
- Occurs often in application in kernels
  - Examples?

## Exercise: Implement a read-writer lock using semaphores

## Recall: Mutual exclusion with semaphores

- Initialize: create a mutex semaphore
- Use: bracket critical section with calls to **wait** and **signal**
- Guarantee: only one process can access the critical section at any time (other will be blocked)

```
sid = semcreate (1);

wait(sid);
// ...
// critical section (use shared resource)
// ...
signal(sid);
```

# Read-writer implementation using semaphores (updated)

```
sid writer_readers = semcreate (1);
sid mutex = semcreate(1)
int readcnt = 0;
```

```
lock_writer(){
     wait(writer_readers);
}


//PERFORM WRITE OPERATION


unlock_writer(){
    // leaves the critical section
    signal(writer_readers);

}
```

```
lock_reader(){
   wait(mutex);

   readcnt++;

   if (readcnt==1)
      wait(writer_readers);

   signal(mutex);
}


//PERFORM READ OPERATION

unlock_reader(){
   wait(mutex);
   readcnt--;
   if (readcnt == 0)
       signal(writer_readers);

   signal(mutex);
}
```

## Break

## Conditional variables

## Conditional variables

- Condition variables are variables that represent certain **conditions** and can only be used in monitors (i.e., when holding a mutex)

- Associated with each condition variable, there is a queue of threads and two operations:
  - Cond_signal()
  - Cond_wait()

- Unrelated to semaphores

## Conditional variables

- When a thread calls **cond_wait**:
  - The caller is (**always**) put into the queue of that condition variable
- When a thread calls **cond_signal:**
  - If there are threads waiting in that condition variable's queue, one of them is released
  - Otherwise, the condition signal is <u>lost</u>

## Using conditional variables

- Condition variables can only be used in a monitor (cond_signal and cond_wait)
- If there are waiting threads, one of them will be released and as a result there are **two** threads executing within the monitor (e.g., holding the lock):
  - One of them is the **caller** that triggers the release and the other is the one being **released**
  - While there are some details and different ways to prevent this from happening, for our purpose, we can assume only one of these two threads will be executing (It could be the caller or the one being released)

## cond_wait()

- A thread calls function cond_wait() to block itself until the event represented by that condition variable occurs
- Format: int cond_wait(cond_t, mutex_t);
  - The first arg is the condition variable on which the caller blocks.
  - The second arg is a mutex lock, which is usually the lock of the monitor that contains this condition variable
    - **This lock must be owned by the calling thread**

## cond_wait()

- If the cond_wait() call is successful:
  - The calling threads blocks on the indicated condition variable and the **lock owned by the calling thread is released**
  - i.e., after the calling thread blocks, the lock of the monitor is ``open'' and hence other threads can acquire it and execute within the monitor
- To wake up a thread blocked by a condition variable, use function cond_signal(); the awakened thread will **automatically be regranted the lock** it owned prior to the wait
- However, the condition that the awakened thread has been waiting may be changed before the thread starts execution so re-evaluation of the condition is required

## cond_signal()

- int  cond_signal(cond_t  *cond);
  - If there at least one thread is blocked on the condition variable, one of them is unblocked by cond_signal()
  - The awakened thread will be <u>automatically granted the lock</u> it originally owned when executed cond_wait()
- When no threads are blocked on the condition variable, cond_signal() has no effect
- Calling cond_signal() should also be protected with a lock that is used to protect the call to cond_wait()

## Typical usage pattern

```
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);
```

pseudo-code syntax

## Recommended homework

Exercise:
Reader-writer with
conditional variables

## Summary

- Properties of synchronization mechanisms
- Usage considerations
- Read-writer patterns
- Conditional variables
- **Practice exercises to learn**

## 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