Synchronization

CS 475, Spring 2018
Concurrent & Distributed Systems
Review: Threads: Memory View

Single-Threaded Process

Multi-Threaded Process

Each thread might be executing totally different code, too
Review: Race Conditions

```java
static int i = 0;
public static void increment()
{
    i = i + 1;
}
```

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>increment()</td>
<td>increment()</td>
</tr>
<tr>
<td>read i = 0</td>
<td>read i = 0</td>
</tr>
<tr>
<td>write i = 1</td>
<td>write i = 1</td>
</tr>
<tr>
<td>write i = 1</td>
<td></td>
</tr>
</tbody>
</table>
Review: Critical Section in increment()

```java
static int i = 0;
public static void increment()
{
    enterSection();
    i = i + 1;
    exitSection();
}
```

Only one thread can read/write i at once

But how to implement enterSection() and exitSection()?
Announcements

• Additional readings:
  • OS TEP Ch 28.1-28.4; Ch 31
• Reminder: HW1 is due 2pm on Weds
Critical Section Problem

Each thread/process has some *critical section* of code that:

- Changes shared variables, files etc
- When one thread/process is in a critical section, no other may be in the same critical section

Critical section problem: design a protocol to solve this

Each process/thread asks for permission to enter critical section, does its work, then exits the section
Solution to Critical-Section Problem

• Need to guarantee **mutual exclusion**
  • If one thread/process is executing its critical section, no other can execute in their critical sections

• Need to guarantee **progress**:
  • If no process is executing in its critical section, and some other would like to, then some process must be allowed to continue

• Need to guarantee **bounded waiting**:
  • If some process wants to enter its critical section, it must eventually be granted access
Peterson’s Solution

• Simple algorithm that solves critical section problem
• Requires two variables: int turn, boolean flag[]
• turn indicates which process can enter the critical section
• flag is an array indicating which process is ready to enter its critical section, flag[i] = true means P_i is ready to enter its critical section
Peterson’s Solution

Algorithm for $P_i$

```java
while(true) {
    flag[i] = true;
    turn = j;
    while(flag[j] && turn == j); //wait if j is in its critical section
    //critical section
    flag[i] = false; //signal we are done
    //do anything else that is not in critical section
}
```

“Busy waiting”

**Problem: Inefficient** - this thread keeps checking flag[], preventing other things from running on your CPU

Sidebar: tough to do correctly on modern hardware, e.g. JVM ([Double Checked Locking Manifesto](https://www.oracle.com/java/technologies/dclm.html))
while(true) {
    flag[i] = true;
    turn = j;
    while(flag[j] && turn == j); //wait if j is in its critical section
    //critical section
    flag[i] = false; //signal we are done
    //do anything else that is not in critical section
}
Locks

- Most systems have some hardware support for implementing this, based on **locks**
- This tells the OS and processor that when a thread is waiting for a lock, **not to bother running it** until it can receive the lock

---

T1 finished faster, T2 got to start sooner
Mutex Locks

• Previous solutions are complicated and generally inaccessible to application programmers
• OS designers build software tools to solve critical section problem
• Simplest is mutex lock
• Protect a critical section by first acquire() a lock then release() the lock
• Calls to acquire() and release() must be atomic
• Usually implemented via hardware atomic instructions
• Ideally: automatically sleeps calling thread if not available
• Otherwise, called a spinlock
Semaphores

• Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.

• Semaphore S – integer variable

• Can only be accessed via two indivisible (atomic) operations

• wait(): consumes a resource (once available)

• signal(): release a resource
Sempahores

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1
  - Same as a mutex lock
- Can solve various synchronization problems
Semaphores

• Define a semaphore as a record:
  
  ```c
  typedef struct {
    int value;
    struct process *L;
  } semaphore;
  ```

• Assume two simple operations provided by OS:
  • block suspends the process that invokes it.
  • wakeup(P) resumes the execution of a blocked process P.
Semaphores

• Implementation of wait and signal. Starting value is # of permits

wait(S):
    S.value--;
    if (S.value < 0) {
        enqueue(this,S.L);
        block();
    }

signal(S):
    S.value++;
    if (S.value <= 0) {
        Thread toWake = pop(s.L);
        wakeup(toWake);
    }
Deadlocks & Starvation

• Starvation: one or more threads are blocked from gaining access to a resource, and hence, can’t make progress

• Deadlock: Two or more threads are waiting for the others to do something
  • T1: Has lock 1, needs lock 2
  • T2: Has lock 2, needs lock 1
  • Hence, neither T1 nor T2 can execute
Classical Problems of Synchronization

• Bounded-Buffer Problem
• Readers and Writers Problem
• Dining-Philosophers Problem
Bounded Buffer

- Producer/consumer communicating through a buffer that holds $n$ items
- Producer can’t put too many items in at one time, consumer can’t remove if no items there
- Might have multiple producers, consumers

Example: buffer can only hold 2 items
Bounded Buffer

- What needs to be tracked?
  - Number of items in buffer
  - Which threads are waiting to put into buffer
  - Which threads are waiting to pull out of buffer
Bounded Buffer with Semaphores

- Semaphore `mutex` initialized to the value 1
- Semaphore `full` initialized to the value 0
- Semaphore `empty` initialized to the value `n`

- `mutex = 1`: Makes sure buffer is viewed atomically
- `full = 0`: Represents number of full slots, tracks threads waiting to add more (if totally full)
- `empty = n`: Represents number of empty slots, tracks threads waiting to read (if empty)
Bounded Buffer Pseudocode

**Producer**

```plaintext
do {
    /* produce an item in next_produced */
    wait(empty);
    wait(mutex);
    /* add next produced to the buffer */
    signal(mutex);
    signal(full);
} while (true);
```

**Consumer**

```plaintext
do {
    wait(full);
    wait(mutex);
    /* remove an item from buffer to next_consumed */
    signal(mutex);
    signal(empty);
    /* consume the item in next consumed */
} while (true);
```
Bounded Buffer Example

Example: buffer can only hold 2 items

- **mutex = 0**: Makes sure buffer is viewed atomically
- **full = 2**: Represents number of full slots, tracks threads waiting to add more (if totally full)
- **empty = 0**: Represents number of empty slots, tracks threads waiting to read (if empty)
Readers and Writers Problem

• Commonly called a read-write lock: data can be read by an unlimited number of threads at a time, written by at most one
• If a thread wants to write, nobody else can be reading
• High level approach:
  • Semaphore used to guard writing
  • Integer variable used to track number of readers
  • First reader acquires write semaphore, last reader to finish releases it
Readers + Writers with Semaphores

- Semaphore `rw_mutex` initialized to 1
- Semaphore `mutex` initialized to 1
- Integer `read_count` initialized to 0

- `rw_mutex = 1` Guards writing
- `read_count = 0` Tracks number of clients reading
- `mutex = 1` Guards updates to `read_count`
Readers Writers Pseudocode

**Writer**

```c
do {
    wait(rw_mutex);
    /* writing is performed */
    signal(rw_mutex);
} while (true);
```

**Readers**

```c
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
    /* reading is performed */
    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while (true);
```
Readers + Writers Example

**Reader**
- First to arrive, set `rw_mutex=0`
- Increment `read_count`
- Decrement `read_count`

**Reader**
- Increment `read_count`
- Decrement `read_count`

**Reader**
- Increment `read_count`
- Decrement `read_count`
- Decrement `read_count` (Released `rw_mutex`)

**Writer**
- Blocked: Unable to acquire `rw_mutex`
- Holds `rw_mutex`

** rw_mutex = 0**
- Guards writing

** read_count=3**
- Tracks number of clients reading

** mutex = 0**
- Guards updates to `read_count`
Readers + Writers with Semaphores

• Might lead to starvation
  • As long as there is at least one thread reading, new readers will take priority over a waiting writer
• OS typically provides read/write locks to solve this
Dining Philosophers

- Philosophers spend their lives alternating thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
- Need both to eat, then release both when done
- In the case of 5 philosophers
- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1
Dining Philosophers Solution

• Each philosopher does this:
  
  ```
  do {
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5 ] );
    // eat
    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5 ] );
    // think
  } while (TRUE);
  ```

• What is the problem with this solution?
Dining Philosophers Solution

• Resolving the deadlock in prior algorithm:
  • Allow at most 4 philosophers to be sitting simultaneously at the table.
  • Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
  • Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.
High-level synchronization mechanisms

• Semaphores are a very powerful mechanism for process synchronization, but they are a low-level mechanism
• Several high-level mechanisms that are easier to use have been proposed
• Monitors (Condition variables and locks)
• Read/Write Locks
• Java and Pthreads provide both semaphores and high-level synchronization mechanisms
• NOTE: high-level mechanisms easier to use but equivalent to semaphores in power
Locking in Java

• Most locks are *reentrant*: if you hold it, and ask for it again, you don’t have to wait (because you already have it)

• Basic primitives:
  • `synchronized{}`
  • `wait`
  • `notify`

• Plus…
  • Lock API… `lock.lock()`, `lock.unlock()`
  • The *preferred* way
Synchronized methods in Java

```java
public synchronized static void increment() {
    i = i + 1;
}
```

Result: Before entering `increment()`, thread gets a lock on the Class object of `increment()`
wait and notify()

• Two mechanisms to enable coordination between multiple threads using the same monitor (target of synchronized)

• While holding a monitor on an object, a thread can wait on that monitor, which will temporarily release it, and put that thread to sleep

• Another thread can then acquire the monitor, and can notify a waiting thread to resume and re-acquire the monitor
wait and notify() example

```java
public class BlockingQueue<T> {
    private Queue<T> queue = new LinkedList<T>();
    private int capacity;

    public BlockingQueue(int capacity) {
        this.capacity = capacity;
    }

    public synchronized void put(T element) throws InterruptedException {
        while (queue.size() == capacity) {
            wait();
        }
        queue.add(element);
        notify(); // notifyAll() for multiple producer/consumer threads
    }

    public synchronized T take() throws InterruptedException {
        while (queue.isEmpty()) {
            wait();
        }
        T item = queue.remove();
        notify(); // notifyAll() for multiple producer/consumer threads
        return item;
    }
}
```

Only one thread can be in put or take of the same queue
Synchronized methods in Java

```java
public synchronized static void increment()
{
    i = i + 1;
}
```

Result: Before entering `increment()`, thread gets a lock on the Class object of `increment()`

```java
public synchronized static void incrementOther()
{
    j = j + 1;
}
```

Result: Before entering `incrementOther()`, thread gets a lock on the Class object of `incrementOther()`

Problem?
Synchronized blocks in Java

• Can also use *any* object as that monitor

```java
static Object someObject = new Object();
public static void increment()
{
    synchronized(someObject){
        i = i + 1;
    }
}

static Object someOtherObject = new Object();
public static void incrementOther()
{
    synchronized(someOtherObject){
        j = j + 1;
    }
}
```

Now, two different threads could call `increment()` and `incrementOther()` at the same time
Assignment 1 Discussion

- [https://autolab.cs.gmu.edu/courses/CSTEST/assessments](https://autolab.cs.gmu.edu/courses/CSTEST/assessments)
Roadmap

• Weds: Java Concurrency
• Joke (didn't make it out last week):

  Knock, knock.  Race condition.  Who's there?
Checkpoint

Go to socrative.com and select “Student Login” (works well on laptop, tablet or phone)

Room Name: CS475

ID is your @gmu.edu email

Remember: this is a checkpoint for you, it is only graded for attendance