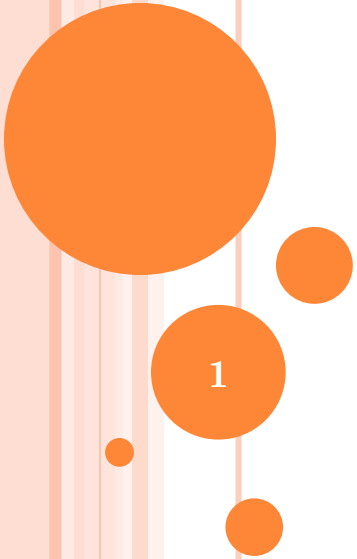

COMP 346

Tutorial 5

MONITORS

WHY DO WE NEED MONITORS?

- Semaphores are very useful for solving concurrency problems… But it's easy to make mistakes!
- If proper usage of semaphores is failed by even one process, the entire system could break down
- Solution?

  We need something better.

Monitor Concept

- Monitors are other mechanism of concurrent programming.
- One can think of **a monitor** as another **Abstract Data Type** – like *a* ***structure*** or ***class*** – with functions and private data.
- The key attribute of a monitor is that **it can be accessed only by one thread at a time**.

- It's a higher level mechanism than semaphores. A monitor is an instance of a class that can be used safely by several threads. All the methods of a monitor are executed with mutual exclusion. So at most one thread can execute a method of the monitor at the same time.

- This mutual exclusion policy makes it easier to work with monitor and to develop the method content of the monitor.

- Monitors have another feature, the possibility to make a thread waiting for a condition. During the wait time, the thread temporarily gives up its exclusive access and must reacquire it after the condition has been met.

Monitor Analogy

- *A **monitor object** can be thought of as an object where each access to it **is protected by a mutex**:*

```
mutex.wait();
myMonitor.Exm();
mutex.signal();
…
```

Monitor Example

```
monitor SharedBalance {
   private int balance;
   public SharedBalance(int amount)
   {balance = amount;}
   public credit(int amount)
   {balance += amount;}
   public debit(int amount)
   {balance  -= amount;}
}
```

SHOW ME THE MONEY

- Let's say we only allow someone to complete a debit transaction when there is enough money to take:

```
SharedBalance::debit(int amount)
{ while(balance < amount) { } // wait for the
  cash
   balance -= amount;
}
```

WHOOPS! DEADLOCK !!!

- This won't work if `a debit()` call occurs when **there is not enough money**:
  - **`debit()`** waits for someone to credit the account
  - **`credit()`** can't run because debit is already executing in the monitor!

MONITOR WITH *CONDITION VARIABLES*

- *Condition variables* are used when:
  - a thread is running in a monitor.
  - encounters a condition that is not satisfied, which can only be satisfied by another thread
- 2 operations:
- **`wait()`** - block on a variable, give up monitor
- **`signal()`** - wake up a thread blocked on this

- You wait on a condition variable(s) **when you want to get another thread to do something for you**.
- This is what happens when you **`wait()`:**
  - It temporarily blocks you,
  - Hands over ***ownership* of the monitor** you are running in **to another thread**,
  - Gives **you back the *ownership*** of the monitor later on…

SHOW ME THE MONEY ALREADY!

- Using a new member variable **condVar**:

```
SharedBalance::credit(int amount)
{
balance += amount;
 condVar.signal();
}
```

- **SharedBalance::debit(int amount) {**

```
while(balance < amount)
{ condVar.wait(); }
balance -= amount;
}
```

Monitors in Java

- Every object of a class that has a *synchronized* method has a monitor associated with it
- Any such method is guaranteed by the Java Virtual Machine execution model to execute mutually exclusively from any other synchronized methods for that object
- Access to individual objects such as arrays can also be synchronized

  - *wait()* releases a lock i.e enters holding area
  - *notify()* signals a process to be allowed to continue
  - *notifyAll()* allows all waiting processes to continue

(BARBER PROBLEM)

– one barber, one barber's chair, *n* customer chairs
– barber sleeps until a customer appears
– first customer to appear wakes barber up
– subsequent customers sit down until all chairs are occupied, otherwise they leave
– how to program the barber and customers without race conditions using Monitor?

(BARBER PROBLEM)

```
monitor SB {
condition : customers, barbers;
int waiting = 0;
entry barber {
……..
cut hair
}
entry customer {
……..
get haircut
  }
 }
}
```

(BARBER PROBLEM)

```
monitor SB {
condition : customers, barbers;
int waiting = 0;
entry barber {
if(waiting==0) wait(customers);
waiting = waiting -1;
signal(barbers);
cut hair
}
entry customer  {
if (waiting < n) {
waiting = waiting+1;
if(waiting==1) signal(customers);
wait(barbers);
get haircut
   }
 }
}
```