

## Concurrency Control

1

## Brief preview of scheduling

- Multiple threads ready to run
- Some mechanism for switching between them
  - Context switches
- Some policy for choosing the next thread to run
  - This policy may be *pre-emptive*
  - Meaning thread can't anticipate when it may be forced to yield the CPU
  - By design, it is not easy to detect it has happened (only the timing changes)

2

## Synchronization

- Threads interact in a multiprogrammed system
  - To share resources (such as shared data)
  - To coordinate their execution
- Arbitrary interleaving of thread executions can have unexpected consequences
  - We need a way to restrict the possible interleavings of executions
  - Scheduling is invisible to the application
- **Synchronization** is the mechanism that gives us this control

3

## Motivating Example

- Suppose we write functions to handle withdrawals and deposits to bank account:

```
Withdraw(acct, amt1) {  
    balance1 =  
    get_balance(acct);  
    balance1 = balance1 - amt1;  
    put_balance(acct, balance1);  
    return balance1;  
}
```

```
Deposit(acct, amt2) {  
    balance2 = get_balance(acct);  
    balance2 = balance2 + amt2;  
    put_balance(acct, balance2);  
    return balance2;  
}
```

- Now suppose you share this account with someone and the balance is \$1000
- You each go to separate ATM machines - you withdraw \$100 and your S.O. deposits \$100

4

## Example Continued

- We can represent this situation by creating separate threads for each action, which may run at the bank's central server (or at the ATM):

```
Withdraw(acct, amt) {  
    balance = get_balance(acct);  
    balance = balance - amt;  
    put_balance(acct, balance);  
    return balance;  
}
```

```
Deposit(acct, amt) {  
    balance = get_balance(acct);  
    balance = balance + amt;  
    put_balance(acct, balance);  
    return balance;  
}
```

- What's wrong with this implementation?
  - Think about potential schedules for these two threads

5

## Interleaved Schedules

- The problem is that the execution of the two processes can be interleaved:

Schedule A

```
balance1 =  
get_balance(acct);  
balance1 = balance1 - amt;  
balance2 =  
get_balance(acct);  
balance2 = balance2 + amt;  
put_balance(acct, balance1);  
put_balance(acct, balance);  
//900
```

Schedule B

```
balance = get_balance(acct);  
balance = balance - amt;  
balance = get_balance(acct);  
balance = balance + amt;  
put_balance(acct, balance);  
put_balance(acct, balance);  
//1100
```

Context switch

- what is the account balance?
- Is the bank happy with our implementation?
  - Are you?

6

## What Went Wrong

- Two concurrent threads manipulated a *shared resource* (the account) without any synchronization
  - Outcome depends on the order in which accesses take place
    - This is called a *race condition*
- We need to ensure that only one thread at a time can manipulate the shared resource
  - So that we can reason about program behavior
  - We need *synchronization*

7

## What program data is shared?

- Local variables are not shared (*private*)
  - Each thread has its own stack
  - Local vars are allocated on this private stack
  - Never pass/share/store a pointer to a local variable on another thread's stack!
- Global variables and static objects are *shared*
  - Stored in the static data segment, accessible by any thread
- Dynamic objects and other heap objs are *shared*
  - Allocated from heap with *new/delete*
  - Any properties/attributes of a class

8

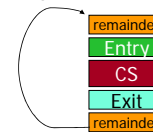
## Mutual Exclusion

- Given:
  - A set of  $n$  threads,  $T_0, T_1, \dots, T_n$
  - A set of resources shared between threads
  - A segment of code which accesses the shared resources, called the *critical section, CS*
- We want to ensure that:
  - Only one thread at a time can execute in the CS
  - All other threads are forced to wait on entry
  - When a thread leaves the CS, another can enter

9

## The Critical Section Problem

- Design a protocol that threads can use to cooperate
  - Each thread must request permission to enter its CS, in its *entry* section
  - CS may be followed by an *exit* section
  - Remaining code is the *remainder* section



- Each thread is executing at non-zero speed
  - no assumptions about relative speed

10

## Critical Section Requirements

- 1) Mutual Exclusion
    - If one thread is in the CS, then no other is
  - 2) Progress
    - If no thread is in the CS, and some threads want to enter CS, only threads not in the "remainder" section can influence the choice of which thread enters next, and choice cannot be postponed indefinitely
  - 3) Bounded waiting (no starvation)
    - If some thread T is waiting on the CS, then there is a limit on the number of times other threads can enter CS before this thread is granted access
- **Performance**
    - The overhead of entering and exiting the CS is small with respect to the work being done within it

11

## Higher-level Abstractions for CS's

- Locks
  - Very primitive, minimal semantics
- Semaphores
  - Basic, easy to understand, hard to program with
- Monitors
  - High-level, ideally has language support (Java)
- Messages
  - Simple model for communication & synchronization
  - Direct application to distributed systems

12

## Using Locks

### Function Definitions

```
Withdraw(acct, amt) {
    acquire(lock);
    balance = get_balance(acct);
    balance = balance - amt;
    put_balance(acct, balance);
    release(lock);
    return balance;
}

Deposit(account, amount) {
    acquire(lock);
    balance = get_balance(acct);
    balance = balance + amt;
    put_balance(acct, balance);
    release(lock);
    return balance;
}
```

### Possible schedule

```
acquire(lock);
balance = get_balance(acct);
balance = balance - amt;

acquire(lock);

put_balance(acct, balance);
release(lock);

balance = get_balance(acct);
balance = balance + amt;
put_balance(acct, balance);
release(lock);
```

13

## Monitors

- an *abstract data type* (data and operations on the data) with the restriction that only one process at a time can be active within the monitor
  - Local data accessed only by the monitor's procedures (not by any external procedure)
  - A process *enters* the monitor by invoking 1 of its procs
  - Other processes that attempt to enter monitor are blocked

14

## Java Synchronization

- Use synchronized Java keyword to restrict execution to 1 thread
- Method level

```
class public Example {
    synchronized public void doSomething() { // Only one thread can execute
                                                // method at any time
    }
}
```
- Object

```
class public Example {
    public void doSomething() {
        synchronized(SomeObject) { // Limit access to only
                                        // one thread per object
        }
    }
}
```
- Class

```
class public Example {
    public void doSomething() {
        synchronized(SomeObject.class) { // Limit access to only
                                            // one thread per class
        }
    }
}
```

15

## Synchronization example

- Problem: ensure consistency of a shared variable in a multithreaded program.
- Idea: use the singleton pattern to ensure a single instance.
- Problem: possible simultaneous initializations.
- Idea: use synchronization.

16

## Synchronization example

```
public class Singleton {
    private volatile static Singleton myUniqueInstance;

    private Singleton() {}

    public static Singleton getInstance() {
        synchronized (Singleton.class) {
            if (myUniqueInstance == null) {
                myUniqueInstance = new Singleton();
            }
        }
        return myUniqueInstance;
    }
}
```

17

## Synchronization efficiency

- Problem: synchronization is expensive.
- Solution: check condition first unsynchronized; then, synchronized.

18

## Synchronization efficiency - example

```
public class Singleton {
    private volatile static Singleton myUniqueInstance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (myUniqueInstance == null) {
            synchronized (Singleton.class) {
                if (myUniqueInstance == null) {
                    myUniqueInstance = new Singleton();
                }
            }
        }
        return myUniqueInstance;
    }
}
```

19

## Producer-Consumer Problem

- A group of processes/threads produce data (producers)
- A different group of processes/threads consume data (consumers)
- Data to be consumed is held in a shared buffer
  - Infinite size for now

20

## Producer-Consumer

```
Consumer
monitor {
    while(buffer.size < 1)
        wait
    remove first buffer element
}
```

```
Producer
monitor {
    add element to buffer
    signal
}
```

21

## Producer-Consumer ver2 Buffer size = 5

```
Consumer
monitor {
    while(buffer.size < 1)
        wait
    remove first buffer element
    if (buffer == 4)
        signal
}
```

```
Producer
monitor {
    while (buffer.size > 4)
        wait
    add element to buffer
    if (buffer == 1)
        signal
}
```

22

## Enforcing single access

- A process in the monitor may need to wait for something to happen
- May need to allow another process to use the monitor
- Provide a *condition* type for variables with operations
  - *wait* (suspend the invoking process)
  - *signal* (resume exactly one suspended process)

23

## More on Monitors

- If no process is suspended, a *signal* has no effect
- If process *P* executes an *x.signal* operation and  $\exists$  a suspended process *Q* associated with condition *x*, then we have a problem:
  - *P* is already "in the monitor", does not need to block
  - *Q* becomes unblocked by the signal, and wants to resume execution in the monitor
  - but both cannot be simultaneously active in the monitor!

24

## Monitor Semantics for Signal

- Hoare monitors (original)
  - `signal()` immediately switches from the caller to a waiting thread
  - The condition that the waiter was blocked on is guaranteed to hold when the waiter resumes
  - Need another queue for the signaler, if signaler was not done using the monitor
- Mesa monitors (Mesa, Java, Nachos, OS/161)
  - `Signal()` places a waiter on the ready queue, but signaler continues inside monitor
  - Condition is not necessarily true when waiter resumes
  - Must check condition again

25

## Hoare vs. Mesa Semantics

- Hoare
  - if (empty)  
wait(condition);
- Mesa
  - while(empty)  
wait(condition)
- Tradeoffs
  - Hoare monitors make it easier to reason about program
  - Mesa monitors are easier to implement, more efficient, can support additional ops like broadcast

26

## Synchronization for servlets

- Problem: multiple service method threads
  - Solution: synchronization
- Do NOT synchronize `doGet`
  - Excessive blocking
  - Does not mutex other servlets
- Synchronize on:
  - Servlet ("instance") variables
  - Session -> session scope  
`synchronized(request.getSession())`
  - Context -> application scope  
`synchronized(getServletContext())`
- Best idea: do not share resources unnecessarily
  - Use separate name spaces

27