

Concurrency

Two or more sequences of events occur “in parallel”
steps of another

□ Multiprogramming

- Single processor runs several programs at the same time
- Each program proceeds sequentially
- Actions of one program may occur between two

□ Multiprocessors

- Two or more processors
- Programs on one processor communicate with programs on another • Actions may happen simultaneously

Process: sequential program running on a processor slide 1

The Promise of Concurrency

□ Speed

- If a task takes time t on one processor, shouldn't it take time t/n on n processors?

□ Availability

- If one process is busy, another may be ready to help □

Distribution

- Processors in different locations can collaborate to solve a problem or work together

□ Humans do it so why can't computers?

- Vision, cognition appear to be highly parallel activities slide

2

Example: Rendering a Web page

- Page is a shared resource
- Multiple concurrent activities in the Web browser
 - Thread for each image load
 - Thread for text rendering

- Thread for user input (e.g., “Stop” button)



Cannot all write to page simultaneously!

- Big challenge in concurrent programming: managing access to shared resources

slide 3

Two Models for Concurrent Programming

- In any concurrent programming model, two of the most crucial issues to be addressed are **communication** and **synchronization**.

- **Communication** refers to any mechanism that allows one thread to obtain information produced by another.

programming: shared memory and message passing.

- **Shared-memory**

- In the shared memory model of concurrency, concurrent modules or processes interact by reading and writing shared objects in memory.

- **Examples** of the shared-memory model:

- A and B might be two processors (or processor cores) in the same computer, sharing the same physical memory.

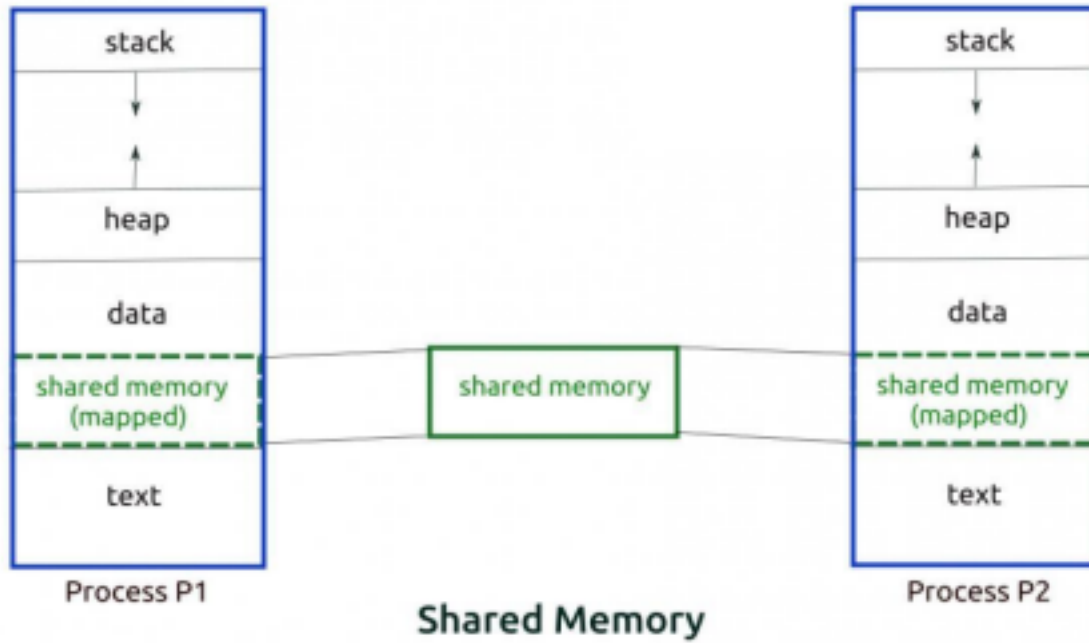
- A and B might be two programs running on the same computer, sharing a common filesystem with files they can read and write.

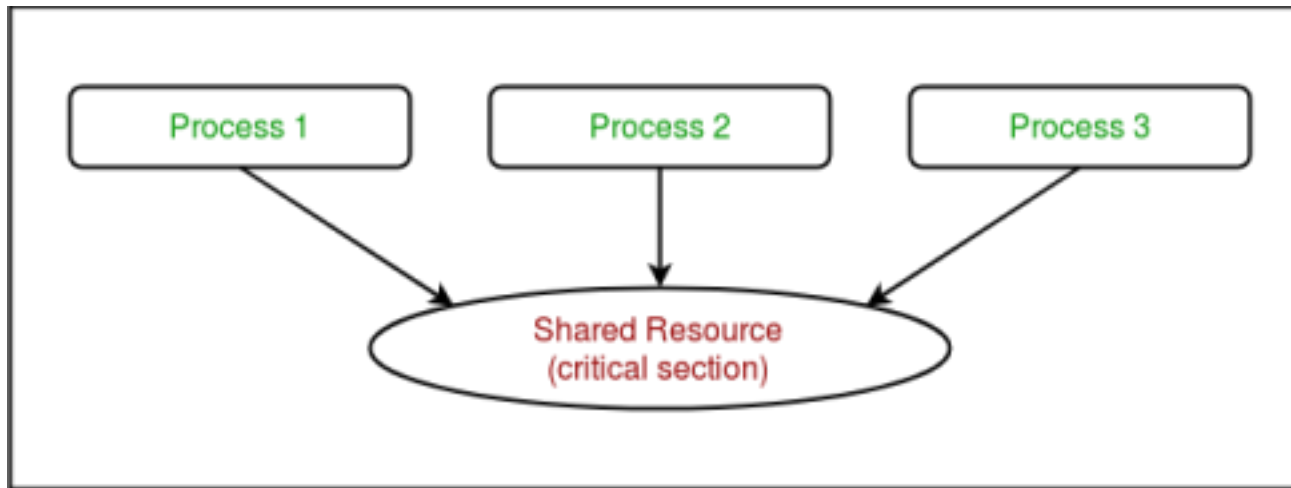
- A and B might be two threads in the same Java program sharing the same

Java [slide 4](#)

objects.

Communication: Shared Memory





slide 5

Shared Memory Example

- But sometimes it is discovered that the balance at the end of the day is not 0. If more than one `cashMachine()` call is running at the same time – then balance may not be zero at the end of the day. Why not?
- Here's one thing that can happen.
- **Interleaving**: Suppose two cash machines, A and B, are both working on a **deposit** at the same time. Here's how the `deposit()` step typically breaks down into low-level processor instructions and when A and B are running

concurrently, these low-level instructions interleave with each other.

```
A get balance (balance=0)
A add 1
A write back the result (balance=1)
      B get balance (balance=1)
      B add 1
      B write back the result (balance=2)
```

```
// each
// modif
private
    for
        }
    }
```

slide 6

Shared Memory Example

- But what if the **interleaving** looked like this:


```
A get balance (balance=0)
```

```
A add 1
```

```
A write back the result (balance=1)
```

```
B get balance (balance=0)
```

```
B add 1
```

```
B write back the result (balance=1)
```

• The balance is now 1 – **A's dollar was lost!** A and B both read the balance at the same time, computed separate final balances, and **then raced** to store back the new balance which failed to take the other's deposit into account. • This is an example of a **race condition**. A race condition means that the correctness of the program (the satisfaction of postconditions and invariants) depends on the **relative timing of events** in concurrent computations A and B. When this happens, we say "**A is in a race with B.**" • A race condition occurs when actions in two processes are **not synchronized** and program behavior depends on the order in which the actions happen

slide 7

Communication: Message Passing

- **Message-passing**
- In the message-passing model, concurrent modules **interact by sending messages to each other** through a **communication channel (queues)**.
- Modules send-off messages, and incoming messages to each module are queued up for handling. Examples include:
 - A and B might be two computers in a network, communicating by network connections.
 - A and B might be a web browser and a web server – A opens a connection to B, asks for a web page, and B sends **the web page data** back to A.
 - A and B might be two programs running on the same computer whose input and output have been connected by a pipe.

Message Passing Example

- In a [producer-consumer relationship](#), the consumer process is dependent on the producer process till the necessary data has been produced.





slide 9

Message Passing Example

- Consider same bank account example using Message Passing module. • Now not only are the cash machine modules, but the accounts are modules, too. Modules interact by sending messages to each other.
- Incoming requests are placed in a **queue** to be handled one at a time. • The

sender doesn't stop working while waiting for an answer to its request. It handles more requests from its own queue. The reply to its request eventually comes back as another message.



slide 10

Message Passing Example

- Message passing **doesn't eliminate** the possibility of **race conditions**. •

Suppose each account supports get-balance and withdraw operations, with corresponding messages.

- Two users, at cash machine A and B, are both trying to withdraw a dollar from the same account. They check the balance first to make sure they never withdraw more than the account holds, as it triggers big bank penalties
- If the account starts with a dollar in it, then what interleaving of messages will fool A and B into thinking they can both withdraw a dollar, thereby overdrawing the account?



slide 11

Concurrent Programming So Far..

- In any concurrent programming model, two of the most crucial issues to be

addressed are **communication** and **synchronization**.

- **Communication** refers to any mechanism that allows one thread to obtain information produced by another and is generally based on either shared memory or message passing.
 - In a shared-memory programming model, some or all of a program's variables are accessible to multiple threads.
 - In a message-passing programming model, threads have no common state. For a pair of threads to communicate, one of them must perform an explicit send operation to transmit data to another.

slide 12

Difference: Shared Memory and Message Passing

Shared Memory	Message Passing
---------------	-----------------

It is one of the region for data communication	Data structure used for communication.
It is used for communication between single processor and multiprocessor systems where the processes that are to be communicated present on the same machine & are sharing common address space.	It is used in distributed environments where the communicating processes are present on remote machines which are connected with the help of a network.
The shared memory code that has to be read or write the data that should be written explicitly by the application programmer.	Here no code is required because the message passing facility provides a mechanism for communication and synchronization of actions that are performed by the processes.
In shared memory make sure that the processes are not writing to the same location simultaneously.	Message passing is useful for sharing small amounts of data so that conflicts need not occur.
It follows a faster communication strategy when compared to message passing technique.	In message passing the communication is slower when compared to shared memory technique.

The Challenges of Concurrency



Concurrent programs are harder to get right •

Folklore: need at least an order of magnitude in speedup for concurrent program to be worth the effort



Some problems are inherently sequential

- Practice – many problems need coordination and communication among sub-problems



Specific issues

- **Communication** – send or receive information
- **Synchronization** – wait for another process to act
- **Atomicity** – do not stop in the middle and leave a mess

slide

Language Support for Concurrency



Threads

- Think of a thread as a system “object” containing the state of execution of a sequence of function calls
- Each thread needs a separate run-time stack (why?)
- Pass threads as arguments, return as function results



Communication abstractions

- Synchronous communication
- Asynchronous buffers that preserve message order



Concurrency control

- Locking and mutual exclusion
- Atomicity is more abstract, less commonly provided slide 15

Inter-Process Communication



Processes may need to communicate

- Process requires exclusive access to some resources
- Process need to exchange data with another process



Can communicate via:

- Shared variables
- Message passing

- Parameters

slide 16

Concurrency via Thread



slide 17



Race conditions

- A race condition occurs when actions in two processes are

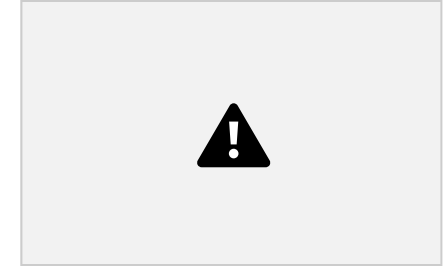
not synchronized and program behavior depends on the order in which the actions happen

Race conditions are not all bad; sometimes any of the possible program outcomes are ok (e.g. workers taking things off a task queue)



Critical section or shared resource

Real Example: Races in Action



Northeast Blackout of 2003

- Affected 50 million people in U.S. and Canada

Race condition in alarm management system caused it to stall, alarms backed up and stalled both primary and backup server

- “We had in excess of three million online operational hours in which nothing had ever exercised that bug. I'm not sure that more testing would have revealed it.”

-- GE Energy's Mike Unum

Synchronization

- SYNCHRONIZATION is the act of ensuring that events in different processes happen in a desired order
- Synchronization can be used to eliminate race conditions • In our example we need to synchronize the increment operations to enforce MUTUAL EXCLUSION on access to X
- Most synchronization can be regarded as either:
 - Mutual exclusion (making sure that only one process is executing a CRITICAL SECTION [touching a variable, for example] at a time),
or as
 - CONDITION SYNCHRONIZATION, which means making sure that a given process does not proceed until some condition holds (e.g. that

a variable contains a given value)

slide 20

Synchronization

- Concurrent programs allow multiple threads to be scheduled and executed with no user control
- We need to control interleaving to avoid racing by using synchronization techniques
 - Mutual exclusion
 - Synchronization to control order of execution
- Mutual exclusion
 - In some computers resources cannot be used by more than one thread at a time.
 - Mutual exclusive term indicates that some resource can be used by only one

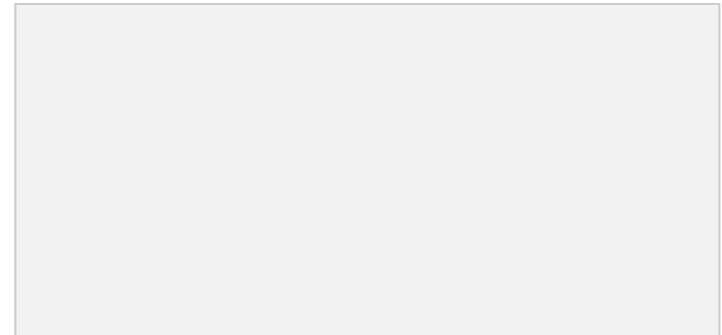
thread at a time.

– In concurrent programs shared data structures are often mutually exclusive to avoid data races and to achieve **atomicity** (Do either all or none)

- **Synchronization** refers to any mechanism that allows the **programmer to control the relative order** in which operations occur in different threads.

slide 21

Mutual Exclusion to Avoid Racing





Mutual Exclusion

- The critical section problem needs a solution to **synchronise** the different processes. The solution to the critical section problem must satisfy the following **conditions** –
 - **Mutual Exclusion**
 - Mutual exclusion implies that **only one process** can be inside the critical section at any time. If any other processes require the critical section, they must

wait until it is free.



- Progress

- Progress means that if a process is **not using** the critical section, then it should **not stop any other process** from accessing it. In other words, any process can enter a critical section if it is free.

- Bounded Waiting

- Bounded waiting means that each **process** must have a **limited waiting time**. It should not wait endlessly to access the critical section.

Mutual Exclusion Signalling

- Semaphore

- A semaphore is a signaling mechanism used in mutual exclusion – A thread that is waiting on a semaphore can be signaled by another thread.

- Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

- **Wait:** The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.
- **Signal:** The signal operation increments the value of its argument S.

```
wait(S){  
while (S<=0);
```

```
S--;}
```

```
signal(S)
```

```
{ S++;
```

```
}
```

slide 25



Semaphores



Semaphore is an integer variable and an associated process queue



Operations:

- $P(s)$ if $s > 0$ then $s--$
else enqueue process
- $V(s)$ if a process is enqueued then dequeue it
else $s++$



Binary semaphore




Counting semaphore

Non Blocking Synchronization

- Synchronization is generally implicit in message-passing models: a message must be sent before it can be received.
- In both shared-memory and message-based programs, synchronization can be implemented either by
 - blocking or
 - non-blocking i.e. spinning mechanism (also called busy-waiting) where atomicity is achieved without the need for mutual exclusion.

What Is Busy Waiting?



Busy waiting, also known as spinning, or busy looping is process synchronization technique in which a process/task waits and constantly checks for a condition to be satisfied before proceeding with its execution.

Busy-Wait Synchronization



Busy-Wait Synchronization

- Busy-waiting algorithm is called as nonblocking synchronization, • Here **repeatedly reading a shared location** until it reaches certain value is called spinning or busy wait algorithm

– Busy-wait condition synchronization is easy if we can cast a **condition** in the form of “**location X contains value Y**”: a thread that needs to wait for the condition can simply read X in a loop, waiting for Y to appear.

While (condition X not True){}

Logic that assumes X is True

- Busy-wait mutual exclusion mechanism is known as **Spin Lock**. •

Test_and_Set (TS lock) is a single invisible machine instruction which tests to see if **key** is available and **if it is**, sets to unavailable.

(**Disadvantage**:Starvation). It is modified with **wait()** and **signal()** methods •

Busy waiting is **preferable when scheduling overhead is larger** than expected wait time and where scheduling-based **blocking is improper**

- Disadvantage: wasting CPU cycles

slide 30

Monitors



Monitor encapsulates a shared resource (monitor = “synchronized object”)

- Private data
- Set of access procedures (methods)
- Locking is automatic
 - At most one process may execute a monitor procedure at a time (this process is “in” the monitor)
 - If one process is in the monitor, any other process that calls a monitor procedure will be delayed

Monitors: wait() and notify()

code.

- When a thread calls `wait()`, it gives up the corresponding lock it is holding and runs again.
 - Many other threads may be waiting on this particular lock.
 - In order to notify a waiting thread, the other thread calls the `notify()` method.
 - `wait()` and `notify()` must be called within a synchronized
- `notify()` wakes up one thread waiting on the lock object
 - `notifyAll()` can be called to wake them all up.

```
synchronized(this){  
    while(!votingComplete) wait();  
}  
countVotes();
```

```
notifyAll();
```

```
synchronized(this){  
    votingComplete = true;
```

```
}
```

slide 32

Challenges in Concurrent Programming

- While implementing concurrent programming or applying synchronizing primitives, two major issues could arise such as
 - Race condition
 - Deadlock

- LiveLock
- Starvation
- Priority Inversion

slide 33

Race Conditions



Race condition occurs when the value of a variable depends on the execution order of two or more

concurrent processes (why is this bad?)



Example

```
procedure signup(person)
begin
    number := number + 1;
    list[number] := person;
end;
signup(joe) || signup(bill)
```

slide 34

Critical Section



Two concurrent processes may access a shared

resource

 Inconsistent behavior if processes are interleaved_{(It}

simply means **performing** (data access or execution or ...) in an arbitrary order)


 Allow only one process in **critical section**

 Issues

- How to select which process is allowed to access the critical section?
- What happens to the other process?

slide 35

Dealing with Race Condition using Locks

- Python provides a synchronization tool: the <threading> module that provides **Lock class** to deal with race condition. Further, the Lock class provides different methods with the help of which we can handle race condition between multiple threads. The methods are described below –
- **acquire() method**
 - This method is used to acquire, i.e., blocking a lock. A lock can be blocking or non blocking depending upon the following true or false value –
 - With value set to True – If the acquire() method is invoked with True, which is the  default argument, then the thread execution is blocked until the lock is unlocked.
 - With value set to False – If the acquire() method is invoked with False, which is not the default argument, then the thread execution is not blocked until it is set to true, i.e., until it is locked.
- **release() method**
 - This method is used to release a lock.
 - If a lock is locked, then the release() method would unlock it. Its job is to allow exactly one thread to proceed if more than one threads are blocked and waiting for the lock to become unlocked.

Locks and Waiting

<initialize concurrency control>

Process 1:

<wait>

signup(joe); // critical section

<signal>

Process 2:

<wait> signup(bill);

// critical section

<signal>

Need atomic operations to implement wait

slide 37

Implementing Mutual Exclusion



Atomic test-and-set

- Instruction atomically reads and writes some location
- Common hardware instruction
- Combine with busy-waiting loop to implement mutex



Semaphore

- Keep queue of waiting processes
 - Avoid busy-waiting loop
- Scheduler has access to semaphore; process sleeps

- Disable interrupts during semaphore operations
 - OK since operations are short

slide 38

Lock using Synchronized Keyword

- Synchronized keyword means that a thread needs a key in order to access the synchronized code.
- To protect data, synchronize the methods that act on data i.e we lock the method that does banking transaction.
- A thread can enter a synchronized method only if the thread can get the key to the



object's lock.

- **Locks are per object** not per method. So, while the thread is holding the key, no other threads can enter any of that object's synchronized methods because the key has already been grabbed by the other thread.
- Synchronized applied to a block of code gives **more flexibility**. we can control how much code needs to be synchronized.

slide 39

Deadlock

- **Deadlock** happens when you have two threads, both of which holding the key of lock the other thread wants, are waiting for each other to release the resource they need (lock) and get stuck for infinite time, the situation is called deadlock.
- It will only happen in the case of multitasking or multi-threading,

because that's where multiple threads come into the picture.



slide 40

Deadlock

Deadlock occurs when a process is waiting for an event that will never happen



Necessary conditions for a deadlock to

exist: • Processes claim exclusive access to resources

- Processes hold some resources while waiting for others •

Resources may not be removed from waiting processes •

There exists a circular chain of processes in which each process holds a resource needed by the next process in the chain



Example: “dining philosophers”

slide 41

A Simple Deadlock Scenario



slide 42

Deadlock



LiveLock

- For example, consider a situation where two threads want to access a shared common resource via a Worker object but when they see that other Worker is also 'active', they attempt to hand over the resource to other worker and wait for it to finish. If initially we make both workers active they will suffer from livelock.



slide 44

LiveLock

- A livelock is a recursive situation where two or more threads consistently give up their resources making no progress.
- A Livelock happens when a request for an exclusive lock is continually

- denied** due to multiple overlapping shared locks that keep interfering. • The processes status continuously changes, prevents them from completing the task, and makes it even more difficult for them to complete the task. • It happens when multiple processes repeatedly perform the same interaction in reaction to changes in the other processes without performing any useful work.
- These processes are not in a waiting state and are running simultaneously, and it is different from a deadlock because all processes of deadlocks are in a state of waiting.
 - **Example:** A common instance of Livelock is that when **two persons meet face to face in a corridor**, and both move aside to let the other pass. They end up going from side to side without making any progress because they are moving in the same direction simultaneously. So, they are failed to cross each other.

slide 45

Starvation

- A **thread** is starved if it **never acquires resources** that it needs • **Starvation** or

indefinite blocking is a phenomenon associated with the Priority scheduling algorithms, in which a process ready for the CPU (resources) can wait to run indefinitely because of low priority. • Example: In a heavily loaded computer system, a steady stream of **higher priority processes can prevent a low-priority process** from ever getting the CPU.

- **Not like deadlock**, as it might get to run but **needs to wait longer** time •
- Resource starvation occurs when you have a task in your system that never gets a resource that it needs to continue with its execution.
- When there is more than one task waiting for a resource and the resource is released, the system has to choose the next task that can use it. If your system has not got a good algorithm, it can have threads that are waiting for a long time for the resource.
 - **Fairness** is the solution to this problem.

slide 46

Priority Inversion

- **Priority Inversion:**
- Priority inversion happens when a high priority task is prevented from running by a lower priority task, effectively inverting their relative priorities.
- Or it occurs when a high-priority process is in the critical section, and it is interrupted by a medium-priority process. This violation of priority rules can happen under certain circumstances and may lead to serious consequences in real-time systems;

Explicit vs. Implicit Concurrency



Explicit concurrency

- Fork or create threads / processes explicitly
- Explicit communication between processes
 - Producer computes useful value
 - Consumer requests or waits for producer



Implicit concurrency

- Rely on compiler to identify potential parallelism
- Instruction-level and loop-level parallelism can be inferred, but inferring subroutine-level parallelism has had less success

cobegin / coend



Limited concurrency primitive –

Concurrent Pascal [Per Brinch Hansen, 1970s] $x := 0;$

cobegin

begin $x := 1; x := x+1$ end; execute sequential blocks in

begin $x := 2; x := x+1$ end; parallel

coend;

$x := 1$

$x := x+1$

print(x); $x := 0$

print(x)



$x := x + 1$

$x := 2$

Atomicity at level of assignment statement slide 49

Properties of cobegin/coend



Simple way to create concurrent processes



Communication by shared variables



No mutual exclusion (“no two processes can exist in the critical section at any given point of time”)



No atomicity (when a thread modifies the state of some object (or set of objects), another thread can't see any intermediary state. Either it sees the state as it was before the operation, or it sees the state as it is after the operation)



Number of processes fixed by program
structure



Cannot abort processes

- All must complete before parent process can go on slide 50

Java Threads



Thread

- Set of instructions to be executed one at a time, in a specified order
- Special Thread class is part of the core language
 - In C/C++, threads are part of an “add-on” library



Methods of class Thread

- start : method called to spawn a new thread
 - Causes JVM to call run() method on object
- suspend : freeze execution (requires context switch)
- interrupt : freeze and throw exception to thread
- stop : forcibly cause thread to halt

<https://docs.oracle.com/javase/tutorial/essential/concurrency/simple.html> slide 51

java.lang.Thread

What does
this mean?

Creates execution environment for the thread
(sets up a separate run-time stack, etc.)

Methods of Thread Class



Runnable Interface



Thread class implements Runnable



interface Single abstract (pure virtual)

method `run()` `public interface Runnable {`

`public void run(); }`

Any implementation of Runnable must provide an implementation of the `run()` method

```
public class ConcurrentReader implements Runnable  
{ ...
```

```
public void run() { ...
```

```
... code here executes concurrently with caller ... }
```

}

Two Ways to Start a Thread



Construct a thread with a runnable object

```
ConcurrReader readerThread = new  
ConcurrReader(); Thread t = new  
Thread(readerThread);  
t.start(); // calls ConcurrReader.run() automatically
```

... OR ...



Instantiate a subclass of Thread

```
class ConcurrWriter extends Thread { ...
```

```
public void run() { ... } }  
ConcurrWriter writerThread = new ConcurrWriter();  
writerThread.start(); // calls ConcurrWriter.run()
```

slide 55

Why Two Ways?

  Java only has single inheritance

Can inherit from some class, but also implement
Runnable interface so that can run as a thread

```
class X extends Y implements Runnable { ...
```

```
    public synchronized void doSomething() { ... }
```


```
    public void run() { doSomething(); }
```

```
}  
X obj = new X();  
obj.doSomething(); // runs sequentially in current  
thread Thread t = new Thread(new X()); // new thread  
t.start(); // calls run() which calls doSomething()
```

slide 56

Interesting “Feature”

[Allen Holub, “Taming Java Threads”]



Java language specification allows access to objects
that have not been fully constructed

```
class Broken {  
    private long x;
```

```
Broken() {  
    new Thread() {  
        public void run() { x = -1; }  
    }.start();  
    x = 0;  
} }
```

Thread created within constructor can access partial object

slide 57

Interaction Between Threads



Shared variables and method calls

- Two threads may assign/read the same variable
 - Programmer is responsible for avoiding race conditions by explicit

synchronization!

- Two threads may call methods on the same object



Synchronization primitives

- All objects have an internal **lock** (inherited from Object)
- **Synchronized method** locks the object
 - While it is active, no other thread can execute inside object
- Synchronization operations (inherited from Object) –
 - Wait: pause current thread until another thread calls Notify
 - Notify: wake up waiting thread

slide 58

Synchronized Methods



Provide mutual exclusion

- If a thread calls a synchronized method, object is locked •
If another thread calls a synchronized method on the same object, this thread blocks until object is unlocked
 - Unsynchronized methods can still be called!



“synchronized” is not part of method

signature • Subclass may replace a synchronized method with unsynchronized method

Wait, Notify, NotifyAll



  `wait()` releases object lock, thread waits on internal queue

`notify()` wakes the highest-priority thread closest to the front of the object's internal queue

 `notifyAll()` wakes up all waiting threads

- Threads non-deterministically compete for access to object
- May not be fair (low-priority threads may never get access)

May only be called when object is locked (`when is that?`) slide 60

Using Synchronization

```
public synchronized void consume() {  
    while (!consumable()) {  
        wait(); } // release lock and wait for resource  
    ... // have exclusive access to resource, can consume  
}  
  
public synchronized void produce() {  
    ... // do something that makes consumable() true  
    notifyAll(); // tell all waiting threads to try consuming  
    // can also call notify() and notify one thread at a time }
```

Example: Shared Queue



Example: Producer-Consumer

Producer Producer

Buffer Consumer

Producer

Consumer

Consumer

Method call is synchronous

How do we do this in Java?

In Pictures



[from Jeffrey Smith]

Solving Producer-Consumer



Cannot be solved with locks alone



Consumer must wait until buffer is not empty •

While waiting, must sleep (use wait method)

- Need condition recheck loop



Producer must inform waiting consumers when there is something in the buffer

- Must wake up at least one consumer (use notify method)

slide 65

Implementation in Stack<T>

```
public synchronized void produce (T object)
{ stack.add(object); notify();
}
public synchronized T consume () {
```

```
        while (stack.isEmpty()) {  
            try {  
wait();  
            } catch (InterruptedException e) { } }  
        int lastElement = stack.size() - 1; T object = stack.get(lastElement);  
        stack.remove(lastElement);  
        return object; }  

```

Why is loop needed here? [slide 66](#)

Condition Rechecks



Want to wait until condition is true

```
public synchronized void lock() throws InterruptedException  
    { if ( isLocked ) wait();  
      isLocked = true; }  
public synchronized void unLock() {  
    isLocked = false;  
    notify(); }
```



Need a loop because another process may run instead

```
public synchronized void lock() throws InterruptedException  
    { while ( isLocked ) wait();
```

```
isLocked = true; }
```

slide 67

Nested Monitor Lockout Problem



Wait and notify used within synchronized code •

Purpose: make sure that no other thread has called method of same object



Wait causes the thread to give up its lock and sleep until notified

- Allow another thread to obtain lock and continue processing



Calling a blocking method within a synchronized method can lead to deadlock

Nested Monitor Lockout Example

```
class Stack {  
    LinkedList list = new LinkedList();  
    public synchronized void push(Object x) {  
        synchronized(list) {  
            list.addLast( x ); notify();  
        }  
    }  
    public synchronized Object pop() {  
        synchronized(list) {  
            if( list.size() <= 0 ) wait();  
            return list.removeLast();  
        }  
    }  
}
```

Could be blocking method of List class

} Releases lock on Stack object but not lock on list; a push from another thread will deadlock

slide 69

Preventing Nested Monitor Deadlock



No blocking calls in synchronized methods,
OR Provide some nonsynchronized method of
the blocking object



No simple solution that works for all programming situations

slide 70

Synchronized Blocks



Any Java block can be synchronized

`synchronized(obj) {`

... mutual exclusion on obj holds inside this block
... }



Synchronized method declaration is just syntactic sugar for synchronizing the method's scope

```
public synchronized void consume() { ... body ...  
} is the same as  
public void consume() {  
    synchronized(this) { ... body ... }  
}
```

slide 71

Locks Are Recursive



A thread can request to lock an object it has already

locked without causing deadlock

```
public class Foo {  
    public void synchronized f() { ... }  
    public void synchronized g() { ... f(); ... }  
}
```

```
Foo f = new Foo;  
synchronized(f) { ... synchronized(f) { ... } ... }
```

Synchronizing with Join()



Join() waits for thread to terminate

```
class Future extends Thread {  
    private int result;  
    public void run() { result = f(...); }  
    public int getResult() { return result;}  
}  
...  
Future t = new future;  
t.start() // start new thread  
...  
t.join(); x = t.getResult(); // wait and get result
```

States of a Java Thread



POSIX Threads

 Pthreads library for C



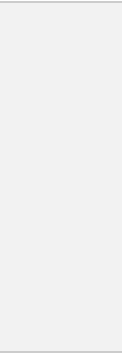
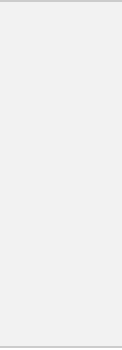
of Using POSIX Threads

Create several
child threads

Wait for children to finish

Thread Stacks





Java-Style Synchronization in C++



slide 78

Using C++ Threads



Why Not Synchronize Everything?



Performance costs

- Current Sun JVM – synchronized methods are 4 to 6 times slower than non-synchronized



Risk of deadlock from too much locking



Unnecessary blocking and unblocking of threads can reduce concurrency



Alternative: immutable objects

- Issue: often short-lived, increase garbage collection slide 80