

CPSC 457  
*Principles  
of  
Operating  
Systems:*

# Threads

...

Reading: Chapter 4 in *Operating System Concepts*,  
8/e, By Silberschatz. Galvin, and Gagne, Wiley,  
2015



# Threads

# Objectives

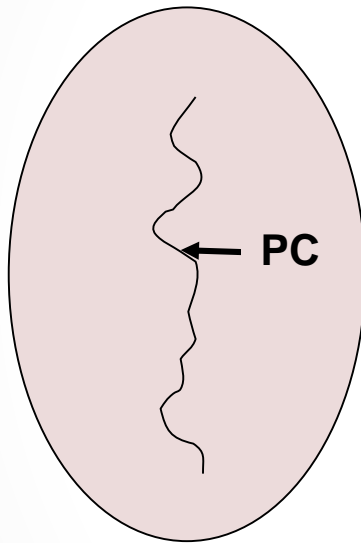
At the end of this section, you will:

1. Understand the concept of a thread
2. Compare and contrast threads and processes
3. Distinguish between user-level and kernel-level thread implementations
4. Learn how to write multithreaded programs in Java, C# and pthreads

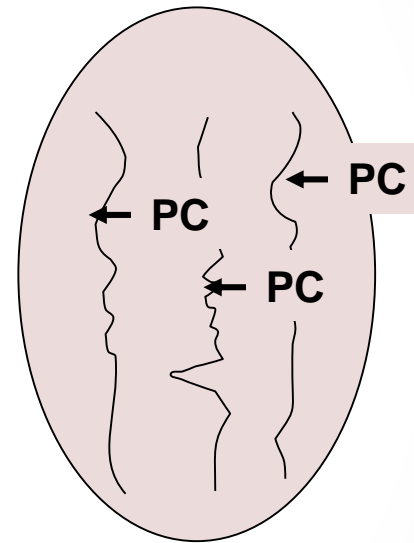
# Overhead of Processes

- Processes allow the OS to overlap I/O and computation, creating an efficient system
- **Overhead:**
  - Process creation
  - Context switching
  - Swapping
  - All require kernel intervention

# Threads enhance the process concept



Traditional Process



Multi-threaded Process

- A set of threads share the same address space

# Advantages of Threads

- Multithreaded-programs are easy to map to multiprocessors
- It is easier to engineer applications with threads



# Threads versus Processes

- Threads (same address space) are not protected from each other
  - Require care from developers
- Context switching is cheaper

# Threads versus Processes

- Traditional process executes a system call, it must block
- A thread executes a system call, peer threads may still proceed



# Thread Implementation - Packages

- Threads are provided as a package, including operations to create, destroy, and synchronize them
- A package can be implemented as:
  - User-level threads
  - Kernel threads

# User-Level Threads

- Thread library entirely executed in user mode
- Cheap to manage threads
  - Create: setup a stack
  - Destroy: free up memory
- Context switch requires few instructions
  - Just save CPU registers
  - Done based on program logic
- A blocking system call blocks all peer threads

# Kernel-Level Threads

- Kernel is aware of and schedules threads
- A blocking system call, will not block all peer threads
- Expensive to manage threads
- Expensive context switch
- Kernel Intervention

# Light-Weight Processes (LWP)

- Support for hybrid (user-level and Kernel) threads
- A process contains several LWPs
- In addition, the system provides user-level threads
- Developer: creates multi-threaded applications
- System: Maps threads to LWPs for execution

# LWP Advantages

- Cheap thread management
- A blocking system call may not suspend the whole process
- LWPs are transparent to the application
- LWPs can be easily mapped to different CPUs
- Managing LWPs is expensive (like kernel threads)



# Java Threads

- Implemented by the **Thread** class, which is part of the **java.lang** package
- Thread is system-dependent:
  - Actual implementation is provided by OS
  - Unified interface with the OS



# Java Threads

- Every thread runs within a specific method in an object
- This object is **not** an instance of the **Thread** class
- It is an instance of a class that extends **Thread** or implements the **Runnable** Interface

# Creating Threads

Threads execute methods

```
class MyThread extends Thread {  
    public void run () {  
        System.out.println("Hello World!");  
    }  
}
```

# Running Threads

- To execute a thread:
  - Instantiate an object (**MyThread**)
  - Call **start()** on the object (which calls **run()**)

```
class TestThread {  
    public static void main (String a[]) {  
        new MyThread().start();  
    }  
}
```

- **main** is executed in a separate thread

# Runnable Threads

```
class MyThread implements Runnable {  
    public void run () {  
        System.out.println("Hello World!");  
    }  
}
```

```
class TestThreads {  
    public static void main (String a[]) {  
        new Thread(new MyThread()).start();  
    }  
}
```

# Threads Example

```
class MyThread extends Thread {  
    String mySymbol;  
    public MyThread (String s) {  
        mySymbol = s; }  
    public void run () {  
        while (true)  
            System.out.print(mySymbol); } }  
  
class TestThreads {  
    public static void main (String a[]) {  
        new MyThread("A").start();  
        new MyThread("B").start(); } }
```

# Output (solo-core machine)

AAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBB  
BBBBBBBBBBBAAAAAAAAAAAAAAAAAAAAABB  
BBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAA  
AAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBAA  
AAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBB  
BBBBBBBBBAAAAAAAAAAAAAAAAAAAAABBBB  
BBBBBBBBBBBBBBBBBBBBBBB ...

# Preempting Threads

- **yield()** a class method in **Thread**
  - When invoked the scheduler suspends the thread invoking it and chooses another ready thread for dispatching

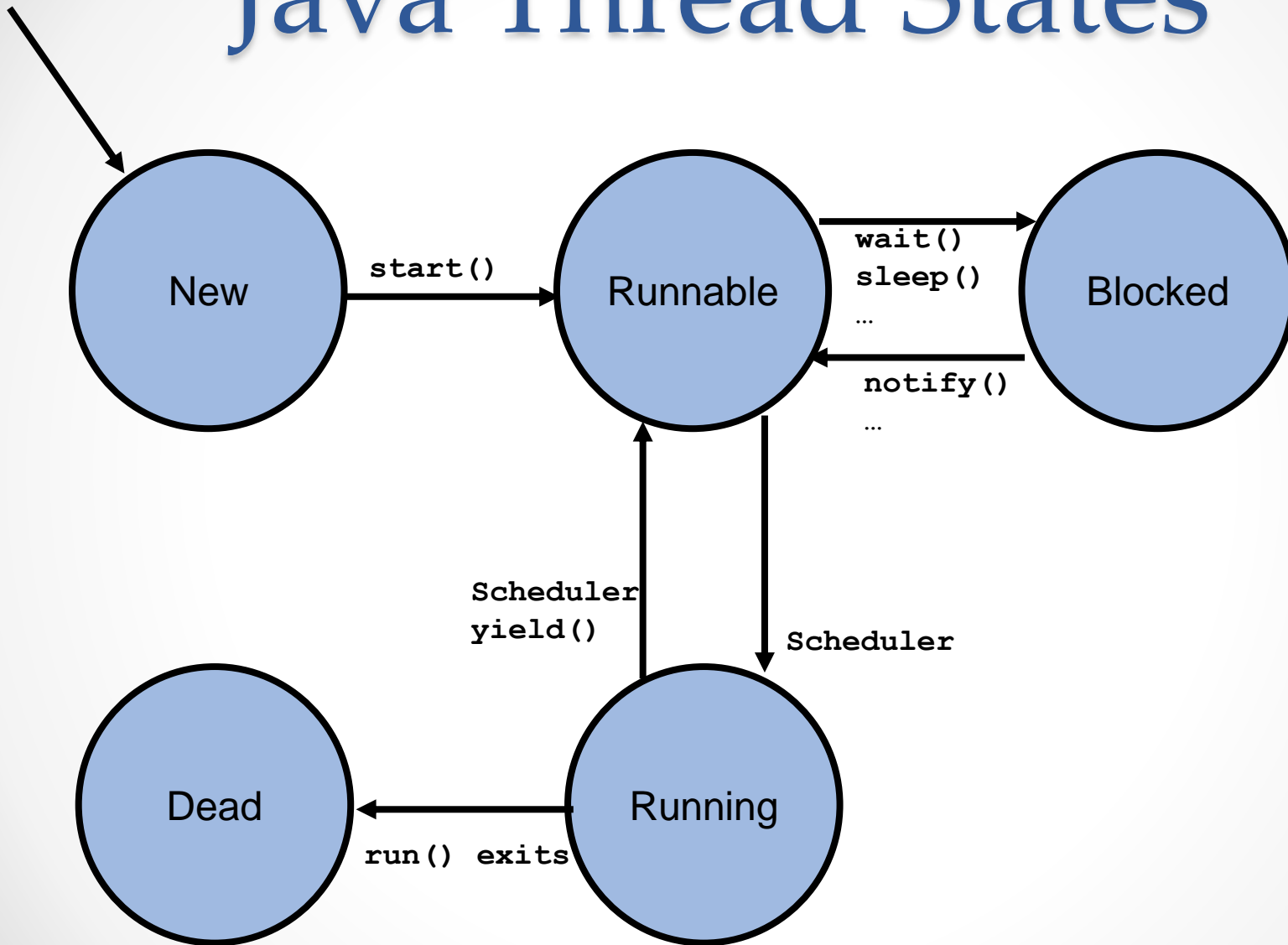
```
public void run() {  
    while (true) {  
        System.out.print(mySymbol) ;  
        Thread.currentThread().yield() ;  
    }  
}
```



# Output (solo-core machine)

[illegible]

# Java Thread States



# java.lang.Thread

- **void interrupt()**
  - Sends an interrupt to a thread
  - “Interrupted” becomes true
  - InterruptedException if sleeping or waiting
- **static boolean interrupted()**
  - Tests “Interrupted” and resets it to false
- **boolean isInterrupted()**
  - Tests “Interrupted”
- **static Thread currentThread()**
  - Returns currently executing thread

# Thread Priorities

- Priorities in Java are between **MIN\_PRIORITY** (=1) and **MAX\_PRIORITY** (=10)
- The **main** method is assigned **NORM\_PRIORITY** (=5)
- Priorities are set by **setPriority()**
- Priorities are extracted by **getPriority()**

# Grouping Threads

- `ThreadGroup grp1 = new ThreadGroup("Best Group")`
- `ThreadGroup grp2 = new ThreadGroup(grp1, "Worst Group")`
  - `grp1` is the parent of `grp2`
- `Thread t1 = new Thread(grp1, "first")`
- `int activeCount()`
  - Number of runnable/running threads in a group
- `ThreadGroup getParent()`
- `void interrupt()`
  - All threads in the group and in children groups
- `int enumerate(Thread[] list)`
  - References for all active threads in the group are stored in `Thread[]`

# New for Java 6&7

- Separation of task *submission* from *execution*
- Pools of “empty” threads are created
- “Runnable” or “Callable” tasks are assigned to these threads
- Unlike Runnable threads, Callable threads can return a value when exiting
- A thread pool is an instance of **ExecutorService**
- Steps:
  - Create tasks
  - Create a thread pool
  - Submit tasks to the pool for execution
  - Shutdown the pool



# Creating Thread Pools

- Thread pools are created by:
- *Executors.newSingleThreadExecutor()*
  - One-thread pool
- *Executors.newCachedThreadPool()*
  - As many threads are needed
  - A thread that is not used for 60s is killed and removed from the pool
- *Executors.newFixedThreadPool()*
  - Fixed number of threads
  - If more tasks than threads, tasks wait in a queue
- *Executors.newScheduledThreadPool()*
- *Executors.newSingleThreadScheduledExecutor()*
  - Scheduled: execution can be delayed



# Example – Defining Tasks

```
private final class StringTask implements
                                Callable<Integer> {
    public Integer call() {
        // blah blah
        return result; }
}
```

# Example – Thread Pool

```
ExecutorService pool =  
    Executors.newFixedThreadPool(5);  
  
    for(int i = 0; i < 10; i++){  
        pool.submit(new Task());  
    } // execute the task 10 times using 5 threads  
  
Pool.shutdown();  
// waits until all tasks are done before shutdown
```

# Getting the “result” back

```
ExecutorService threadPool =  
    Executors.newFixedThreadPool(5);  
CompletionService<Integer> pool = new  
ExecutorCompletionService<Integer>(threadPool);  
for(int i = 0; i < 10; i++){  
    pool.submit(new Task());  
}  
for(int i = 0; i < 10; i++){  
    Integer result = pool.take().get();  
    //Compute overall result  
}  
threadPool.shutdown();
```

# POSIX Threads example

...

# POSIX Threads

- Called also Pthreads
- A standard programming interface for threads in C
- IEEE POSIX 1003.1c standard
- A nice tutorial at :  
<https://computing.llnl.gov/tutorials/pthreads/>

```
#include <pthread.h>
#include <stdio.h>
void *runner(void *param); /* thread executed function */
struct shared {
    int flag; // flags the completion of the child
    int sum;
};
struct shared s; // shared between parent and child threads
```

```
int main (int argc, char *argv[])
```

D2L ref: pex1.c

```
{
```

```
    pthread_t tid;
```

```
    pthread_attr_t attr;
```

```
    pthread_attr_init(&attr); \\get default attributes
```

```
    pthread_create(&tid,&attr,runner,argv[1]);
```

```
    s.sum = 0;
```

```
    s.flag = 0;
```

```
    while (!s.flag)
```

```
        printf("Parent: sum is %d\\n", s.sum);
```

```
    pthread_join(tid,NULL);
```

```
    printf("Parent: Finally sum is %d\\n", s.sum);
```

```
    • CPSC 457
```



```
void *runner(void *param)
{
    int i, upper = atoi(param);
    for (i = 0; i <= upper; i++)
    {
        s.sum += i;
        printf("    Child: sum is %d\n", s.sum);
    }
    s.flag = 1;
    pthread_exit(0);
}
```



Threads in C#

# C# Threads

- System.Threading namespace
- Which method does a thread execute?
- ThreadStart ts = new ThreadStart(m);
- Method m
  - public void m()
- Create a new thread:
- Thread th = new Thread(ts);
- Run the thread:
  - th.Start()

# C# Thread States

- Unstarted // new
- Running // Start(), scheduled
- WaitSleepJoin // calls Wait(), Sleep(), or Join()
- SuspendRequested // Suspend() by another thread
- Suspended
- AbortRequested // Abort() by another thread
- Stopped

# C# Thread Priorities

- ThreadPriority. Highest
- ThreadPriority. AboveNormal
- ThreadPriority. Normal
- ThreadPriority. BelowNormal
- ThreadPriority. Lowest