CMPS 405: OPERATING SYSTEMS

Multithreading

This material is based on the operating system books by Silberschatz and Stallings and other Linux and system programming books.

Objectives

- ❖ Motivations
- ❖ Processes vs. Threads
- ❖ Threads' Context Switching
- ❖ Benefits of Multithreading
- ❖ Types of Multithreading Support
- ❖ Multithreading Models
- ❖ Java Threads
- ❖ POSIX Threads Primitives
- ❖ Additional Problems

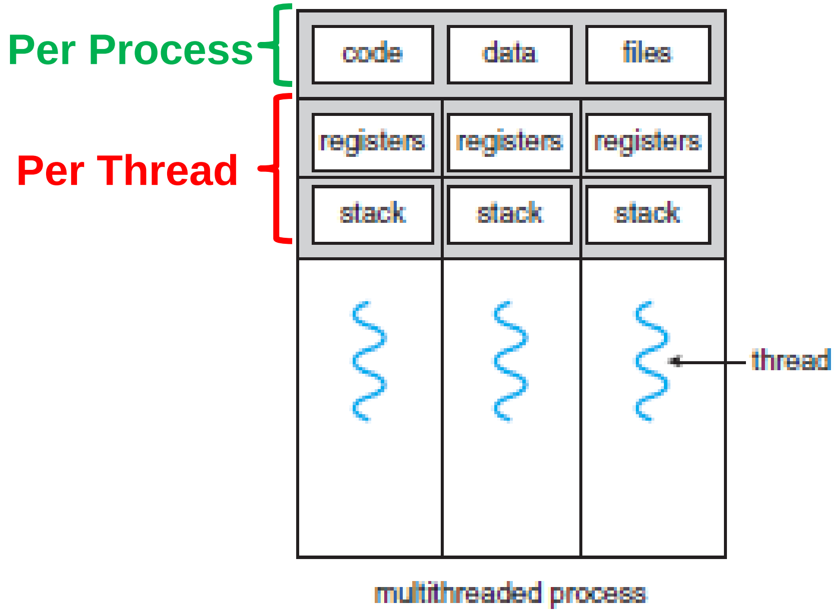
Motivation

- **In many cases, a process desire to perform more than one task at a time.**
  - This can be achieved by implementing an application as a separate process with several streams of control, called **threads**.
  - This is known as **multithreaded programming**.

- **Examples**
  - A **Web Browser** application is a process that has a number of stream of control, threads. One thread might be displaying images or text while another thread retrieves data from the network, an another downloading video or playing an audio.
  - A **Word Processor** application is a process with a number of stream of control, threads. One thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.
  - A **Web Server** listen for requests from clients. When a request is received, the server creates a new thread to service the request and resumes listening for additional requests. In this case the server code is a process creating service threads.

Definition

- A **thread** comprises a **thread ID**, a **program counter**, a **register set** (including program counter and stack pointer), and a **stack**. It shares with other threads belonging to the same process its **code section**, **data section**, and **other operating-system resources**, such as **open files** (e.g., descriptors), **signal handlers**, **current working directory**, and **user and group IDS**.



Benefits of Multithreading

1. **Responsiveness:** Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.

2. **Resource sharing:** Threads share the memory and the resources of the process to which they belong by default and within the same address space.

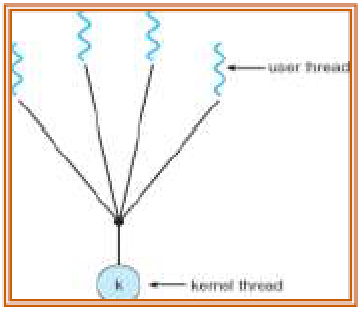
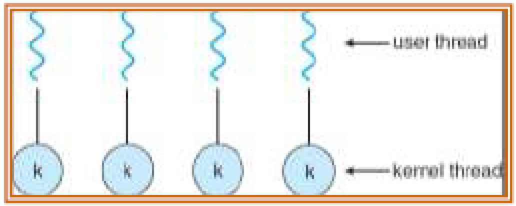
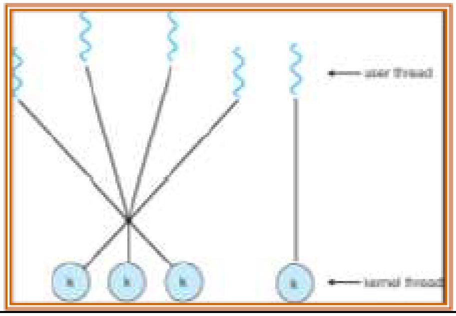
3. **Economy:** threads share memory and resources within a process therefore, they are also faster to create and context-switch. In
   - Solaris, for example, creating a process is about 30 times slower than is creating a thread, and context switching is about 5 times slower.

4. **Scalability:** Threads may run in parallel on a multiprocessor system and therefore, increases parallelism.

Processes vs. Threads

Comparison point	Process	Thread
Representation	PCB: ...	Id, PC, stack, and register set.
Type of context switch	In kernel space	In user space
Speed of context switching	Slow	Fast
Mechanism to perform context switching	by invoke system calls to kernel.	By invoking procedures in thread library
Responsiveness	Slow	Fast: same address space
Memory requirements	Heavy	Light
Existing environment	Alone	Within a process
Sharing of resources	None with other processes.	All resources within process address space
Utilizing MP Architecture	No	Yes
Concurrency	No, low	Yes, high
Speed of creation	Slow	Fast
Speed of execution	Slow	Fast
Cost – Economy	Expensive	Cheap

Multithreading Models

Comparison point	Many-to-one	One-to-one	Many-to-many/two-level
Mapping of user-level to kernel-level threads	M:1 	1:1 	M:N, N<=M 
Management	By thread library in user space	Kernel	Both
Blocking	Yes If one block, all will block, entire process.	No	No
Concurrency degree	Low	High	High
Run in parallel on multiprocessor	No	Yes	Yes
Drawback	Blocking, no Concurrency, and does not run in parallel on multiprocessor.	Overhead of creating kernel threads.	Overhead of creating kernel threads.
Control of number of threads	None	Apply restriction on the number of kernel threads. **Thread Pool:** a fixed number of reusable kernel-threads created for each process. A thread is assigned a task and once finished is made available again in the pool for future tasks to be assigned to.	
Examples	Green Threads in Solaris and GNU Portable Threads	Linux, Windows OS, and Solaris 9 and newer.	IRIX, HP-UX, True64 Unix, and Solaris older than 9.

Thread Libraries

- A thread library provides the programmer with an API for creating and managing threads. As will as runtime system to manage threads transparently (the user is not aware of the runtime system).
  - The runtime system allocates data structures to hold the thread’s ID, stack and program counter value, an scheduling and usage information.
- There are two primary ways of implementing a thread library.
  - The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.
  - The second approach is to implement a kernel-level library supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.
- Three main thread libraries are in use today: POSIX Pthreads; Win32; and Java.

Objectives

- ❖ Motivations
- ❖ Processes vs. Threads
- ❖ Threads' Context Switching
- ❖ Benefits of Multithreading
- ❖ Types of Multithreading Support
- ❖ Multithreading Models
- ❖ Java Threads
- ❖ POSIX Threads Primitives
- ❖ Additional Problems

Java Threads

- Java has built in thread support for Multithreading
  - Synchronization
  - Thread Scheduling
  - Inter-Thread Communication:

`run`	`start`	`setPriority`
`yield`	`join`	`getPriority`
`sleep`	`currentThread`	`isAlive`
`interrupt`	`currentCount`	`isInterrupted`

- Study the Java API Documentation for the class **Thread** and the Interface **Runnable**.

Java Threads

- Threading mechanisms:
  - Create a class that ***extends*** the **Thread** class, or
  - Create a class that ***implements*** the **Runnable** interface

Extending the Thread class

- Threads are implemented as objects that contain a method called ***run()***.

```
class MyThread extends Thread {
//…
    public void run(){
     // thread body of execution
   }
}
```

- Creating and starting a thread

```
Thread thr1 = new MyThread();

thr1.start();
```

OR

```
new MyThread().start();
```

Extending the Thread class

```
class MyThread extends Thread {  // the thread
        public void run() {
                System.out.println(" this thread is running ... ");
        }
} // end class MyThread
```

```
class ThreadEx1 { // a program that utilizes the thread
        public static void main(String [] args  ) {
        MyThread t = new MyThread();
        // due to extending the Thread class (above)
        // I can call start(), and this will call run().
        // start() is a method in class Thread.
            t.start();
       } // end main()
}  // end class ThreadEx1
```

Implementing the interface Runnable

```
class MyThread implements Runnable{
 // .....
  public void run(){
     // thread body of execution
  }
}
```

Creating and strating a thread

```
Runnable myObject = new MyThread();
Thread thr1 = new Thread( myObject );
thr1.start();
```

OR

```
(new Thread(new MyThread())).start();
```

Implementing the interface Runnable

```
class MyThread implements Runnable  {
        public void run() {
        System.out.println(" this thread is running ... ");
        }
} // end class MyThread
```

```
class ThreadEx2 {
        public static void main(String [] args  ) {
             Thread t = new Thread(new MyThread());
             t.start();
        } // end main()
}       // end class ThreadEx2
```

Q. How to pass arguments to a thread in Java?

A. Pass it to the constructor of the threaded class, as easy as that.

Practice

- Check the examples on BB.

- Formulate a multithread solution for a problem of your own choice and attempt it.

- Try other problems.

Objectives

- ❖ Motivations
- ❖ Processes vs. Threads
- ❖ Threads' Context Switching
- ❖ Benefits of Multithreading
- ❖ Types of Multithreading Support
- ❖ Multithreading Models
- ❖ Java Threads
- ❖ POSIX Threads Primitives
- ❖ Additional Problems

POSIX Threads

- Some times called **pthreads** because all the thread functions start with *pthread*.
  - Most of these functions return 0 on success and nonzro error code if unsuccessful.
  - These functions do not have to be restarted if interrupted by a signal.

POSIX function	Description
pthread_cancel	Terminate another thread
pthread_create	Create a thread
pthread_detach	Set thread to release resources
pthread_equal	Test two thread IDs for equality
pthread_exit	Exit a thread without exiting a process
pthread_kill	Send a signal to a thread
pthread_join	Wait for a thread
pthread_self	Find out own thread ID

int pthread_create(pthread_t * *tid*, const pthread_attr_t * *attr*, void * (start_routine*)(void *), void **arg*);**

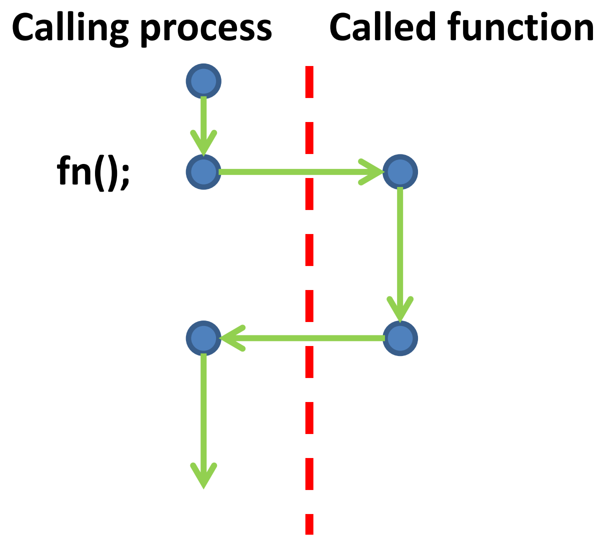
tid: holds the id of the created thread.
attr: a structure containing the settings of the thread attributes or ***NULL*** for keep default settings.
start_routine: is the function having the code to be executed by the thread.
Arg: is the only argument passed to the function ***start_routine***. If you want to pass more than one argument, then you arrange it in an array, struct, or file and then pass it.

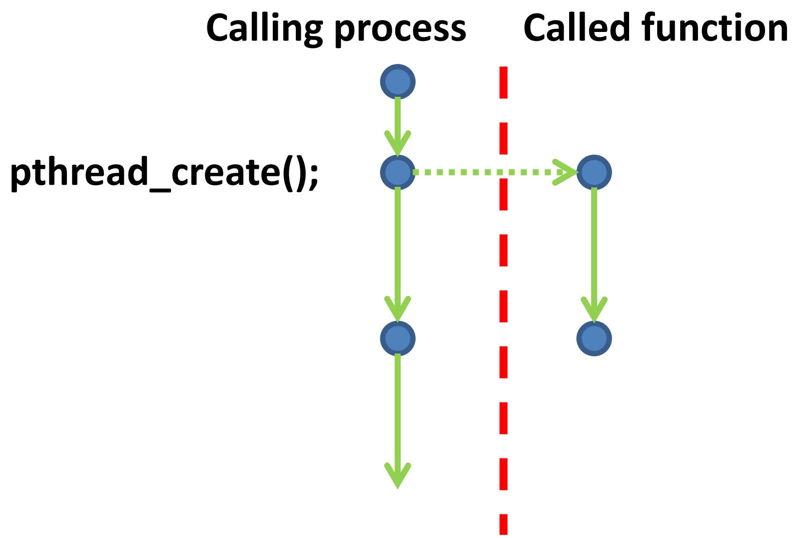
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
void *thr_fn(void *arg){
	printf("I am the thread of process %lu \n", (unsigned long) getpid());
	return((void *)0);//return(NULL);
}
void main(){
	pthread_t ntid;
	pthread_create(&ntid, NULL, thr_fn, NULL);
	sleep(2);
	printf("I am the hosting process %lu \n", (unsigned long) getpid());
	exit(0);//this will terminate the main thread an all other threads
}
```

Compile with
gcc -pthread

Thread vs. Function

- A thread is a schedulable entity executes an independent stream of instructions and never returns to the point of call. The calling program continues to execute concurrently.
- When a function is called, the calling program execution moves through the function code and returns to the point of call, generating a single stream of execution.





Mapping variables to memory

- Variables are mapped to virtual memory according to their storage classes.

  - **Global variables:** any variables declared outside of a function. One read/write instance for each global variable that can be referenced by any thread.

  - **Local automatic variables:** any variables declared inside a function without the static attribute. Each thread’s stack contains its own instances of any local automatic variables.

  - **Local static variables:** any variables declared inside a function with the static attribute. One read/write instance in virtual memory even if it was declared by each thread.

- **Shared variables:** We say that a variable *v* is shared if and only if one of its instances id referenced by more than one thread. Recall that there is only one run-time instance and this instance is referenced by both peer threads.

POSIX Thread Functions

- A thread ID is represented by the **pthread_t** data type. Implementations are allowed to use a structure to represent the **pthread_t** data type, so portable implementations can't treat them as integers. Therefore, a function must be used to compare two thread IDs.

```
#include <pthread.h>
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

Returns: nonzero if equal, 0 otherwise

- A thread can obtain its own thread ID by calling the **pthread_self** function.

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Returns: the thread ID of the calling thread

- This function can be used with **pthread_equal** when a thread needs to identify data structures that are tagged with its thread ID.
  - For example, a master thread might place work assignments on a queue and use the thread ID to control which jobs go to each worker thread.

POSIX Thread Functions

- If any thread within a process calls **exit**, **_Exit**, or **_exit**, then the **entire process terminates**.

- Similarly, when the **default action** is to terminate the process, a **signal** sent to a thread will **terminate the entire process**.

- A single thread can exit in three ways, thereby stopping its flow of control, **without terminating the entire process**.
  1. The thread can simply **return** from the start routine. The return value is the thread's exit code.
  2. The thread can be canceled by another thread in the same process by calling **pthread_cancel**.
  3. The thread can call **pthread_exit**.

POSIX Thread Functions

#include <pthread.h>
void pthread_exit(void *_rval_ptr_);

- The **rval_ptr** argument is a typeless pointer.
- This pointer is available to other threads in the process by calling the **pthread_join** function.

#include <pthread.h>
int pthread_join(pthread_t _thread_, void **_rval_ptr_);

Returns: 0 if OK, error number on failure

- The calling thread will **block** until the specified thread calls pthread_exit, returns from its start routine, or is canceled.
  - If the thread was canceled, the memory location specified by **rval_ptr** is set to **PTHREAD_CANCELED**.
- By calling **pthread_join**, we automatically place the thread with which we're joining in the **detached state** so that its resources can be recovered.
- If we're not interested in a thread's return value, set **rval_ptr** to ***NULL***.

POSIX Thread Functions

```
#include <pthread.h>
int pthread_detach(pthread_t tid);
```

Returns: 0 if OK, error number on failure

- A thread's underlying storage can be reclaimed immediately on termination if the thread has been detached.
- After a thread is detached, we can't use the **pthread_join** function to wait for its termination status.
- A thread can arrange for functions to be called when it exits, similar to the way that the **atexit** function can be used by a process to arrange that functions are to be called when the process exits. The functions are known as thread **cleanup handlers**.
- More than one cleanup handler can be established for a thread. The handlers are recorded in a stack, which means that they are executed in the reverse order from that with which they were registered.

Comparison of process and thread primitives

Process primitive	Thread primitive	Description
`fork`	`pthread_create`	create a new flow of control
`exit`	`pthread_exit`	exit from an existing flow of control
`waitpid`	`pthread_join`	get exit status from flow of control
`atexit`	`pthread_cleanup_push`	register function to be called at exit from flow of control
`getpid`	`pthread_self`	get ID for flow of control
`abort`	`pthread_cancel`	request abnormal termination of flow of control

Practice

- Check the examples on BB.

- Formulate a multithread solution for a problem of your own choice and attempt it.

- Try other problems.

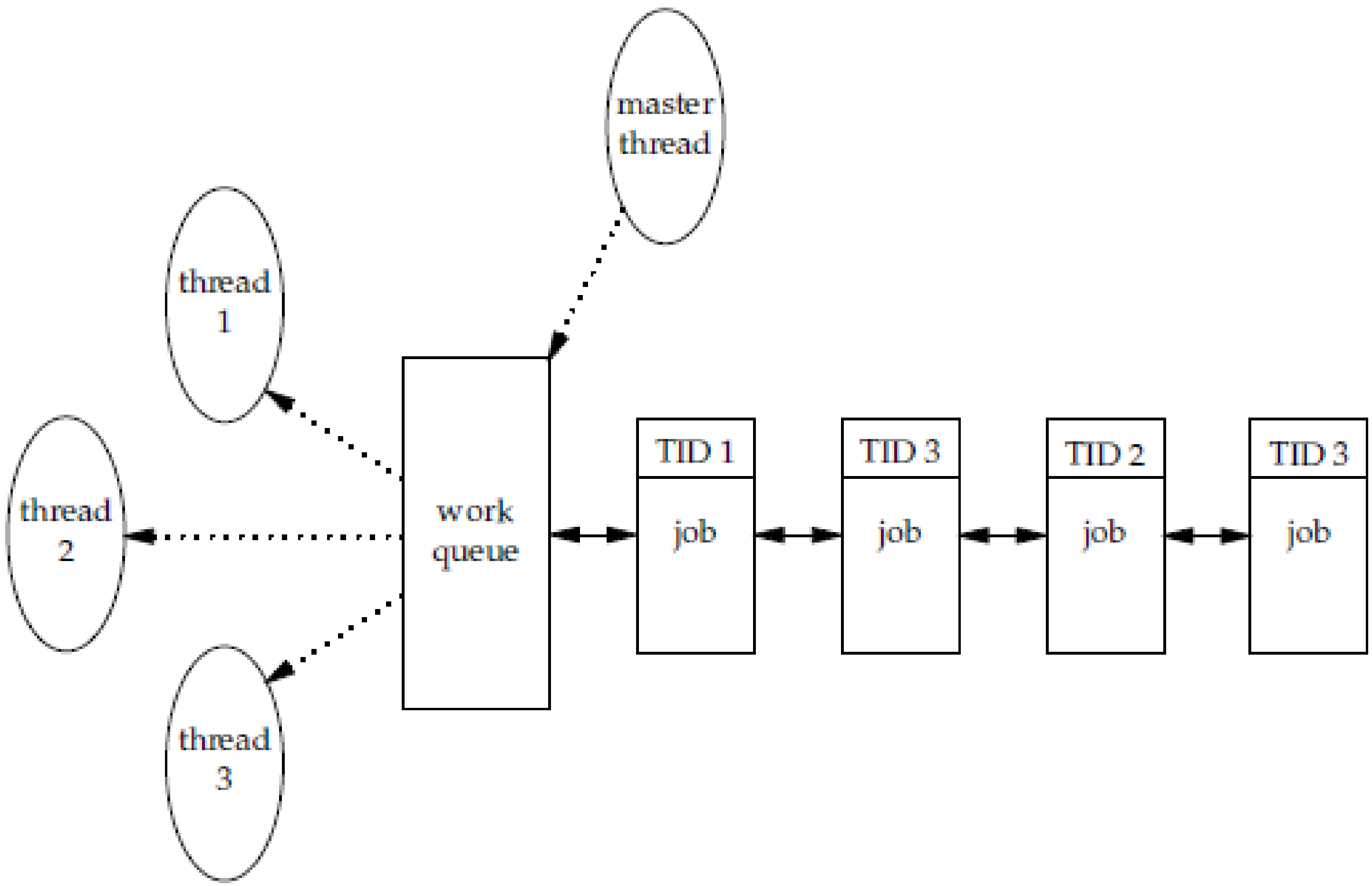
Objectives

- Motivations
- Processes vs. Threads
- Threads’ Context Switching
- Benefits of Multithreading
- Types of Multithreading Support
- Multithreading Models
- Java Threads
- POSIX Threads Primitives
- Additional Problems

Additional Problems

- ❖ The bank account problem.
- ❖ The Producer/Consumer Problem.
- ❖ The Barrier
- ❖ …

MASTER/WORKERS MODEL



Multiple Core Systems

- Nowadays, multiple computing cores are placed on a single chip and each of these cores appears as a separate processor to the operating system.

- **Challenges in programming for multicore systems**
    1. **Dividing activities:** This involves examining applications to find areas that can be divided into separate, concurrent tasks and thus can run in parallel on individual cores.
    2. **Balance:** Ensuring that a task contributes as much value to the overall process as other tasks in order to be worthy executing on a separate core.
    3. **Data splitting:** The data accessed and manipulated by the tasks must be divided to run on separate cores.
    4. **Data dependency:** The data accessed by the tasks must be examined for dependencies between two or more tasks and apply synchronization when it is dependent.
    5. **Testing and debugging:** Testing and debugging such concurrent programs is inherently difficult because there are many different execution paths.

Eight Simple Rules for Designing Multithreaded Applications

Rule 1: Identify Truly Independent Computations

Rule 2: Implement Concurrency at the Highest Level Possible

Rule 3: Plan Early for Scalability to Take Advantage of Increasing Numbers of Cores

Rule 4: Make Use of Thread-Safe Libraries Wherever Possible

Rule 5: Use the Right Threading Model

Rule 6: Never Assume a Particular Order of Execution

Rule 7: Use Thread-Local Storage Whenever Possible or Associate Locks to Specific Data

Rule 8: Dare to Change the Algorithm for a Better Chance of Concurrency