Chapter 4: Threads

1

Chapter Topics

- Multithreaded processes
- Mapping between user threads and kernel threads
- Threading Issues

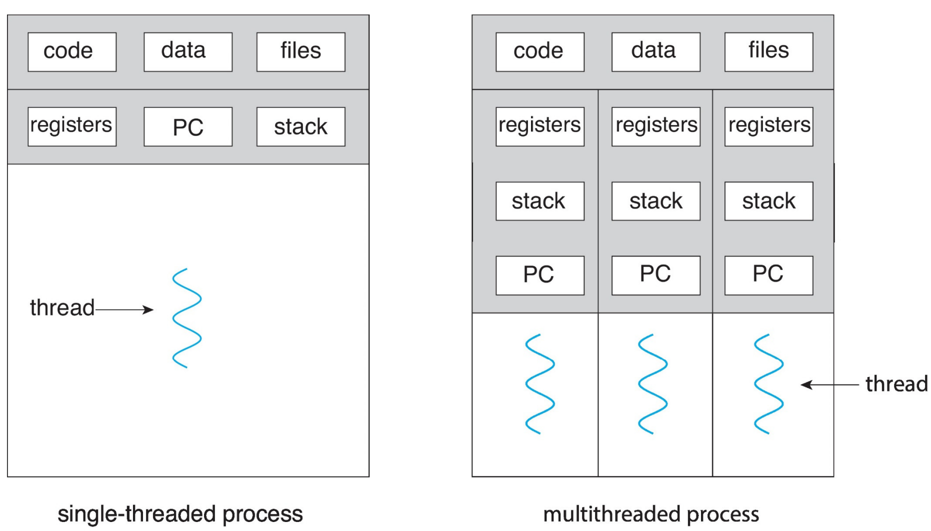
2

Thread

- A *thread* is a flow of control within a process. It is a basic unit of CPU utilization in a multithreaded computer systems.
- A traditional (or heavyweight) process has a single thread of control.
- If a process has multiple threads of control, it is called a multithreaded process. The threads belong to the same process share its code section, data section, and other OS resources, such as open files and signals.
- A thread consists of a thread ID, a program counter, a register set, and a stack.
- A multithreaded process can perform more than one task at a time.

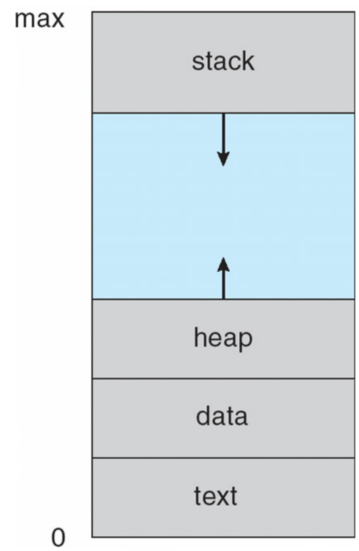
3

Single and Multithreaded Processes



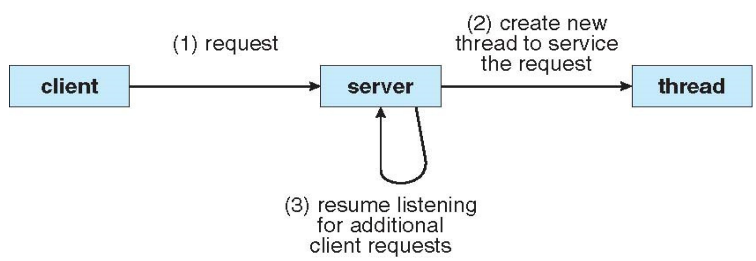
4

Logical Address Space of a Single Threaded Process



5

Multithreaded Server Architecture (aka Thread-per-request Architecture)



- The server will create a separate thread (called an I/O thread) that listens for client requests.
- When a request is accepted, the I/O thread will create a new thread (called a worker thread) to service the request and resume listening for additional requests.
- The worker thread destroys itself when it has processed the request.

6

Cost of Creating a Thread

- When a new thread is created in an existing execution environment of a process, the main tasks are:
  - Allocate a region in the address space for the stack of the thread.
  - Provide the initial values of the processor registers and the thread's execution state and priority.
- The cost of creating a new process is much bigger because a new execution environment must be created first.

7

TCP Stream Communication in Java

- The client creates a stream socket (a *Socket*) bound to any available local port then makes a connection request to a server at its server port.
- The server creates a listening socket (a *ServerSocket*) bound to a server port and waits for clients to request connections. The listening socket maintains a queue of incoming connection requests.
- When the server accepts a connection, a new stream socket (a *Socket*) is created automatically for the server to communicate with the client, meanwhile the listening socket is retained to receive the connection requests from other clients.
- Each stream socket has an *InputStream* and an *OutputStream.*
  - *InputStream* and *OutputStream* are abstract classes that define methods for reading and writing bytes, respectively.
  - A process can send data to other one by writing to its *OutputStream*, and the other process obtains the data by reading from its *InputStream.*
  - We can think of the *InputStream* and *OutputStream* as a pair of one-way pipes connected to a stream socket.

8

Java API for TCP Stream

- For TCP streams, Java API provides two classes: *ServerSocket* (which is a listening socket) and *Socket* (which is a stream socket).
- The server uses a constructor of *ServerSocket* to create a socket at a server port for listening for connection requests from clients, like:

  ServerSocket listenSocket = new ServerSocket(serverPort);
- The *accept()* method of *ServerSocket* accepts a connection request from the queue of incoming connection requests (if it is not empty); and as a result, an instance of *Socket* is created:

  Socket s_socket = listenSocket.accept();
- The client also creates a instance of *Socket* by specifying the DNS hostname and port number of the server:

  Socket c_socket = new Socket(server_hostname, serverPort);

9

Java API for TCP Stream (cont'd)

- The *Socket* class provides methods *getInputStream()* and *getOutputStream(),* which return the references to *InputStream* and *OutputStream* of the socket, respectively.
- The client can make a *DataInputStream* and a *DataOutputStream from the InputStream* and *OutputStream* of its socket, as:

```
DataInputStream in = new
        DataInputStream(c_socket.getInputStream());
DataOutputStream out = new
        DataOutputStream(c_socket.getOutputStream());
```

- *DataInputStream* and *DataOutputStream* classes implement the methods for reading and writing the binary representations (i.e., bytes) of primitive data types, respectively.
  - The methods of *DataInputStream* includes *readBoolean*, *readChar*, *readInt*, *readLong*, *readFloat*, *readDouble*, *readUTF*, etc.

10

TCP client makes connection to server, sends request and receives reply

```
import java.net.*;
import java.io.*;
public class TCPClient {
		public static void main (String args[]) {
		// args give message content and server hostname
		Socket s = null;
		   try{
				int serverPort = 7896;
				s = new Socket(args[1], serverPort);
				DataInputStream in = new DataInputStream( s.getInputStream());
				DataOutputStream out =
						new DataOutputStream( s.getOutputStream());
				out.writeUTF(args[0]);       // UTF (Unicode Transformation Format) is a string encoding
				String data = in.readUTF();
				System.out.println("Received: "+ data) ;
		   }catch (UnknownHostException e){
						System.out.println("Sock:"+e.getMessage());
		   }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
		   }catch (IOException e){System.out.println("IO:"+e.getMessage());}
		}finally {if(s!=null) try {s.close();}catch (IOException
e){System.out.println("close:"+e.getMessage());}}
		}
}
```

11

TCP server makes a connection for each client and then echoes the client’s request

```
import java.net.*;
import java.io.*;
public class TCPServer {
   public static void main (String args[]) {
		try{
				int serverPort = 7896;
				ServerSocket listenSocket = new ServerSocket(serverPort);
				while(true) {
						Socket clientSocket = listenSocket.accept();
						Connection c = new Connection(clientSocket);
						   // creates a Connection thread.
				}
		} catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
   }
}
```

// this figure continues on the next slide

12

TCP server continued

```
class Connection extends Thread {
        DataInputStream in;
        DataOutputStream out;
        Socket clientSocket;
        public Connection (Socket aClientSocket) {
          try {
                clientSocket = aClientSocket;
                in = new DataInputStream(clientSocket.getInputStream());
                out =new DataOutputStream(clientSocket.getOutputStream());
                this.start();
           } catch(IOException e)  {System.out.println("Connection:"+e.getMessage());}
        }
        public void run(){
          try {                                   // an echo server
                String data = in.readUTF();
                out.writeUTF(data);
          } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
          } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
          } finally{ try {clientSocket.close();}catch (IOException e){/*close failed*/}}
        }
}
```

13

Thread Control Block (TCB)

- **Thread Control Block (TCB)** is a data structure which contains thread-specific information needed to manage it.
- An example of information contained within a TCB is:
  - Thread Identifier
  - State of the thread (start, ready, running, waiting, or terminated)
  - Program counter
  - Stack pointer
  - Thread's other register values
  - Pointer to the PCB (Process Control Block) of the process in witch the thread was created.

14

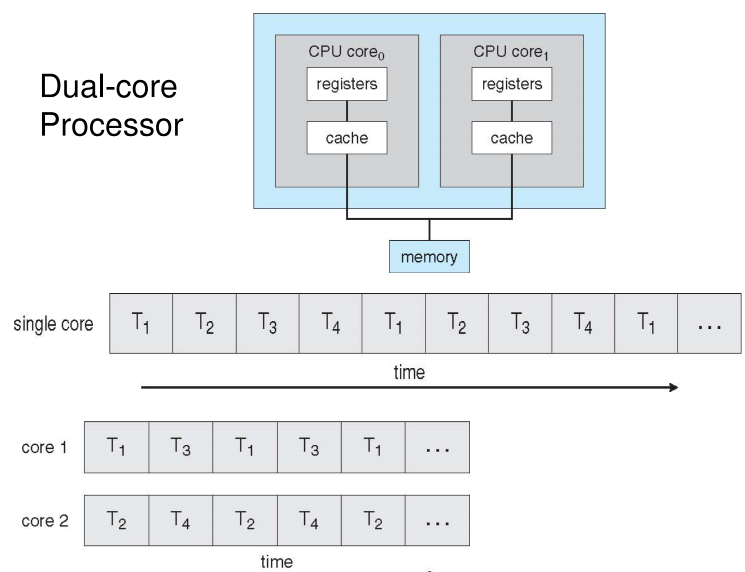
Benefits of Multithreaded Programming

- **Responsiveness:** Multithreading an interactive application allows a program to continue running even if a part of it is blocked or performing a lengthy operation.
  – For example, a multithreaded Web browser allows user interaction in one thread while an image is being loaded in another thread.
- **Resource Sharing:** Resource sharing can be done more efficiently between threads than between processes, because the threads within a process share the code, data, and open files.
- **Economy:** Both creation of a thread and context switching between threads (in the same process) take much less time than those of processes.
- **Scalability:** A single-threaded process can only run on one processor, no matter how many processors are available. But multiple threads within a process can be running in parallel on different processors.

15

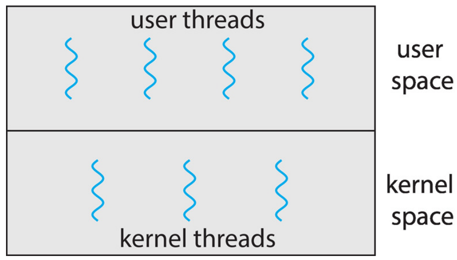
Multicore Programming

Dual-core Processor



16

User Threads and Kernel Threads



- There are two types of threads implemented: user(-level) threads and kernel(-level) threads.
- User threads are supported above the kernel and are managed by a *user-level thread library* at the user level.
- A user-level thread library is a package of routines for:
  - Creating and destroying threads.
  - Passing messages and sharing data between threads.
  - Scheduling the execution of threads.
  - Saving and restoring thread contexts.

17

User Threads and Kernel Threads (cont'd)

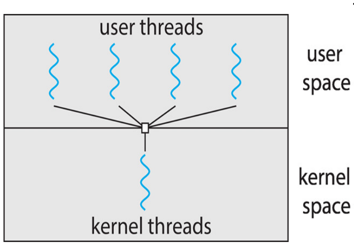
- Each user thread can't actually run on its own, and the only way for a user thread to run is if a kernel thread is actually told to execute the code contained in a user thread.
- Kernel threads are supported directly by the OS kernel. The OS kernel performs thread creation, scheduling and management in the kernel space.
- It is the kernel thread that the OS kernel schedules (specifically by the kernel scheduler) to run on the physical processors.
  - A kernel thread runs in user mode when executing user functions or library calls; it switches to kernel mode when executing system calls.
  - A kernel-only thread executes only in kernel mode.

18

Multithreading Models

1. Many-to-One mapping model
   – User processes are multithreaded, whereas kernel processes are single threaded.
   – OS kernel schedules the whole user process as an execution unit and assigns a single execution state:
2. Many-to-Many mapping model
   – OS kernel allocates each user process a smaller or equal number of kernel threads than the number of user threads in the user process.
   – OS kernel can control the total number of kernel threads contending for memory space and CPU time.
3. One-to-One mapping model
   – OS kernel allocates a kernel thread for each user thread created in a user process.

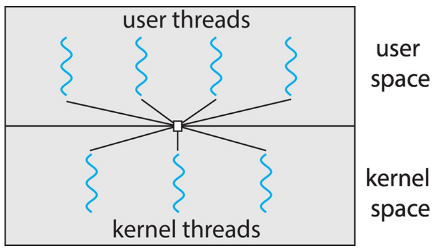
Many-to-One Model



Many-to-one model

- Many-to-one model maps (i.e., assigns) multiple user threads in a user process to a single kernel thread.
- As no kernel support for multithreaded processes is provided, this model has the following problems:
  – When a user thread makes a blocking system call, such as a call for disk I/O, it blocks the entire process it belongs to and all threads within the process.
    - This is because the kernel schedules the whole process as an execution unit and assigns a single execution state: The kernel is unaware of how the multiple threads within the process are managed by the user-level scheduler.
  – Multiple user threads within a process cannot be executed on different processors in parallel, because only one user thread can access the kernel at a time.

Many-to-Many Model



Many-to-many model

- The many-to-many model multiplexes many user threads to a smaller or equal number of kernel threads.
- All the user threads (within each user process) are scheduled by the user-level scheduler (also called process scheduler) of the process, whereas kernel threads are scheduled by the kernel scheduler.
- This technique allows an application to specify the number of kernel threads it requires.

21

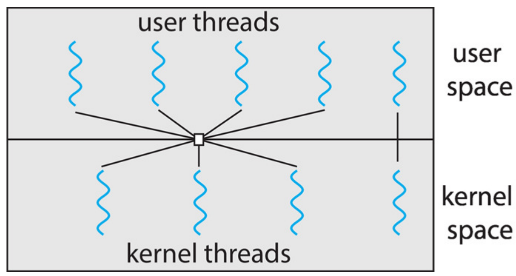
Scheduling of User Threads (in Many-to-One and Many-to-Many Models)

- Scheduling user threads within each user process has some advantages:
  – Switching between threads belonging to the same user process does not necessarily involve a system call (to the kernel). Instead, it is handled by the user-level scheduler (also called process scheduler).
  – As the scheduling of user threads (within a user process) is performed outside the kernel (by the user-level scheduler), it can be customized or changed to suit particular application environments.

22

Two-level Model

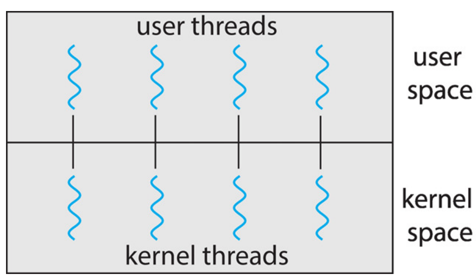
- Similar to many-to-many model, except that it allows a user-level thread to be bound to a kernel thread.



Two-level model

One-to-One Model

One-to-one model



- Creating a user-level thread requires the corresponding kernel thread that the OS can dispatch to a processor.
  - Scheduling of user-level threads (within each user process) is not required: only the scheduling of kernel threads (by the kernel scheduler) is required.
- One-to-one model provides more concurrency than many-to-many model as all the threads (within each user process) could run on different processors in parallel.
- The amount of memory consumed by kernel thread data structures can become significant as the number of threads in the system increases.

Threading Issues

- Semantics of **`fork()`** and **`exec()`** system calls.
- Thread cancellation of a target thread:
  asynchronous or *deferred.*
- Signal handling
- Thread pool model
- Thread-local storage (aka Thread-specific data)
- Scheduler Activations

25

Semantics of `fork()` and `exec()`

- When a thread in a process calls **`fork()`**, does the new process duplicate all threads or duplicate only the thread that invoked the **`fork()`** system call?
  - Some UNIX systems have chosen to have two versions of **`fork()`**: one duplicates all the threads and another that duplicates only the thread that invoked the **`fork()`** system call.
- If a thread invokes the **`exec()`** system call, the program specified in the parameters of **`exec()`** will replace the entire process, including all threads, because a new address space is created.

26

Thread Cancellation

- Thread cancellation is the task of terminating a thread before it has completed.
- A thread that is to be cancelled is often referred to as the target thread.
- Cancellation of a target thread may occur in two different scenarios:
  - **Asynchronous cancellation:** The target thread is terminated immediately.
  - **Deferred cancellation:** The target thread periodically checks (a flag) to determine whether it should terminate.
    - The target thread can perform this check at a point it can be cancelled safely. Pthreads refers to such point as cancellation point.
    - For example, the target thread can be cancelled later if it is currently updating data shared with other threads.

27

Signal Handling

- A signal is a software-generated interrupt, and it is used in UNIX systems to notify a process that a particular event has occurred:
  1. A signal is generated by the occurrence of a particular event.
  2. A generated signal is delivered to a process.
  3. Once delivered, the signal must be handled.
- Each signal has a default signal handler that is run by the kernel, which can be overridden by a user-defined signal handler that is called to handle the signal.
- Options for delivering a signal to a multithreaded process:
  - Deliver the signal to the thread that caused the signal (e.g., illegal memory access, division by zero).
  - Deliver the signal to every thread in the process when the signal is caused by an event external to the process. (e.g., ^c , a timer expires).
  - Deliver the signal to certain threads in the process (for example, the first thread that is not blocking the signal).
    - Most UNIX OSs allow a thread to specify which signals it will accept and which ones it will block (to ignore those signals).
  - Assign a specific thread to receive all signals for the process.
    - And then it can deliver the signal to the first thread that is not blocking the signal.

28

Thread Pool (aka Worker-Pool) architecture

- The idea of thread pool is creating a number of threads at the server process' startup and place them in a pool, where they sit and wait for work.
  1. When the server (actually an *I/O thread* of the server) receives a request from a client, it awakens a thread (called *worker thread*) from the pool (if one is available) and passes it the request for service.
  2. Once the thread completes its service, it returns to the pool and awaits (instead of destroying itself).
  3. If there is no available thread in the pool, the server waits until one becomes available.

- Advantages of using a thread pool (compared to Thread-per-request architecture):
  – Servicing a request with an existing thread is usually faster than creating a new thread.
  – A thread pool limits the number of worker threads for an application, which is important on a system that cannot support a large number of concurrent threads.

29

Thread-Local Storage (aka Thread-Specific Data)

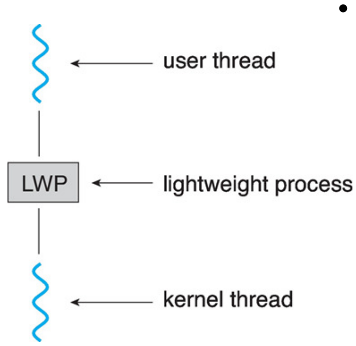
- Threads belong to a process share the data of the process. However, in some cases, each thread might need its own copy of certain data. Such data is called thread-local storage (aka thread-specific data).
- For example, in a transaction processing system, we might service each transaction in a separate thread. Since each transaction is usually assigned a unique transaction id (TID), we could use the thread-local storage to associate each thread with the transaction id.

30

Light Weight Process (LWP)

- Both many-to-many and two-level models require the communication between kernel and user-level thread library to multiplex user threads to allocated kernel threads, and to maintain appropriate number of kernel threads allocated to the application.
- Many systems implement many-to-many and two-level model by placing an intermediate data structure between the user-level thread library and kernel threads.
  - This data structure is typically known as a lightweight process (LWP), which is a kernel data structure and resides in kernel space.
  - Each LWP is attached to a kernel thread, and it is the kernel thread that the OS schedules to run on the physical processors.
    - A kernel thread runs in user mode when executing user functions or library calls; it switches to kernel mode when executing system calls.

Light Weight Process (LWP) (cont'd)

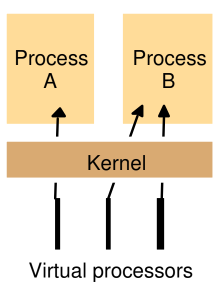


- To the user-level scheduler (aka process scheduler), the LWP appears to be a ***virtual processor*** as it has a register set (actually a data structure that can contain the values of registers) on which the process's user-level scheduler can load the context of a user thread to run.
  - The user-level scheduler has the task of assigning its READY threads to the set of virtual processors (i.e., LWPs of kernel threads) allocated to the user process.
  - The OS kernel will load the register values in the LWP to the physical registers to start executing the kernel thread.

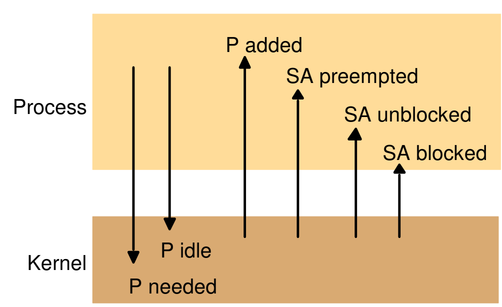
Scheduler Activations (for many-to-many and two-level thread models)

- A *scheduler activation* (*SA*) is a call from the kernel to a process, which notifies the process's **user-level scheduler** of an event.
  - Since it is a call from a lower level layer (kernel) to a user process, it can be considered as an *upcall* to a user process from the kernel.
  - Upcalls are handled by the user-level thread library with an *upcall handler*, and a kernel thread is needed to run an upcall handler.
  - Actually, a *scheduler activation* (*SA*) is a kernel thread that can notify a user-level scheduler of an event (and executing the corresponding handler).
- The idea of using scheduler activations is that what a user-level scheduler requires from the kernel is not just a set of kernel threads onto which it can map user threads. It also requires the information about the events that are relevant to its scheduling decisions.
- The *FastThreads* package adopted an event-based scheduling of user threads, using scheduler activations.

Scheduler Activations (cont'd)



A. Assignment of virtual processors to processes



B. Events between user-level scheduler & kernel
Key: P = processor; SA = scheduler activation

- In this example, the kernel can allocate *virtual processors* (i.e., LWPs each of which is attached to a kernel thread) to a user process.

Scheduler Activations (cont'd)

- A user-level scheduler notifies the kernel when either of two types of event occurs:
  - When an extra virtual processor is needed, or
  - When a virtual processor is idle and no longer needed.
- There are four types of event that the kernel notifies to the user-level scheduler:
  - ***Virtual processor allocated***: the kernel has assigned a new virtual processor to the process. The user-level scheduler can load the additional virtual processor of the notifying SA (i.e., kernel thread) with the context of a READY thread, which can thus start execution.
  - ***Thread is blocked***: a user thread is about to be blocked in the kernel, and the kernel makes an upcall (using a SA) to notify the user-level scheduler. The user-level scheduler sets the state of the corresponding thread to *BLOCKED,* and saves the context of the blocked thread (forwarded through the virtual processor of the notifying kernel thread).
    - Then the user-level scheduler can allocate a *READY* thread to the (virtual processor of) notifying kernel thread.

35

Scheduler Activations (cont'd)

- ***Thread unblocked***: a user thread that was blocked in the kernel has become unblocked and is ready to execute again. The user-level scheduler can return the corresponding thread to the *READY* list. In general, to create a notifying SA, the kernel either
  - allocates a new kernel thread to the process, or
  - preempts a kernel thread that has been allocated to a process (which requires a separate notification).
- ***Thread preempted***: the kernel has taken away the specified kernel thread from the process. The user-level scheduler saves the context of the preempted user thread (i.e., the user thread that was mapped to the lost kernel thread) and places it into the *READY* list.

- The kernel assists the user-level scheduler through its event notification and by providing the contexts of the blocked and preempted threads.

36