**Chapter 4: Thread and concurrency - Operating System Concepts-Wiley (2018)**

1.**Overview**

A thread is a basic unit of CPU utilization; it comprises a thread ID,a program counter (PC), a register set, and a stack.

* 1. **Motivation**

-Most software applications that run on modern computers and mobile devices are multithreaded.

- Applications can also be designed to leverage processing capabilities on multicore systems.Such applications can perform several CPU-intensive tasks in parallel across the multiple computing cores.

- In certain situations, a single application may be required to perform several similar tasks.

- One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request.

- Most operating system kernels are also typically multithreaded.

- Many applications can also take advantage of multiple threads, including basic sorting, trees, and graph algorithms.

1.2 **Benefits**

- Responsiveness

- Resource sharing

- Economy

- Scalability

1. **Multicore Programming**

-Earlier in the history of computer design, in response to the need for more computing performance, single-CPU systems evolved into multi-CPU systems.

* 1. **Programming challenges**

- five areas present challenges in programming for multicore systems:

+ identifying tasks

+ balance

+ data splitting

+ data dependency

+ testing and debugging

* 1. **Types of parallelism**

- Data parallelism focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core.

- Task parallelism involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation.

1. **Multithreading model**

- Our discussion so far has treated threads in a generic sense. However, support for threads may be provided either at the user level, for user threads, or by the kernel, for kernel threads.

* 1. **Many-to-one model**

- maps many user-level threads to one kernel thread.Thread management is done by the thread library in user space, so it is efficient.

- Green threads—a thread library available for Solaris systems and adopted in early versions of Java—used the many-toone model.

* 1. **One-to-one model**

- The one-to-one model maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.

* 1. **Many-to-many model**

- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine.

- Whereas the manyto-one model allows the developer to create as many user threads as she wishes, it does not result in parallelism, because the kernel can schedule only one kernel thread at a time.

- One variation on the many-to-many model still multiplexes many userlevel threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation is sometimes referred to as the two-level model.

1. **Thread libraries**

- A thread library provides the programmer with an API for creating and managing threads. 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.The second approach is to implement a kernel-level library supported directly by the operating system.

- Three main thread libraries are in use today: POSIX Pthreads, Windows, and Java.

- Pthreads, the threads extension of the POSIX standard, may be provided as either a user-level or a kernel-level library. The Windows thread library is a kernel-level library available on Windows systems. The Java thread API allows threads to be created and managed directly in Java programs.

* 1. **Pthreads.**

- Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. This is a specification for thread behavior, not an implementation.

* 1. **Windows.**

- The technique for creating threads using the Windows thread library is similar to the Pthreads technique in several ways. We illustrate the Windows thread API in the C program.

* 1. **Java.**

- Threads are the fundamental model of program execution in a Java program, and the Java language and its API provide a rich set of features for the creation and management of threads.

- All Java programs comprise at least a single thread of control—even a simple Java program consisting of only a main() method runs as a single thread in the JVM.

- Java threads are available on any system that provides a JVM including Windows, Linux, and macOS. The Java thread API is available for Android applications as well.

1. **Implicit threading**
   1. **Thread pools**

- Whereas creating a separate thread is certainly superior to creating a separate process, a multithreaded server nonetheless has potential problems.

- The first issue concerns the amount of time required to create the thread, together with the fact that the thread will be discarded once it has completed its work.

- The second issue is more troublesome. If we allow each concurrent request to be serviced in a new thread, we have not placed a bound on the number of threads concurrently active in the system.

- Unlimited threads could exhaust system resources, such as CPU time or memory. One solution to this problem is to use a thread pool.

* 1. **Fork join**

- Recall that with this method, the main parent thread creates (forks) one or more child threads and then waits for the children to terminate and join with it, at which point it can retrieve and combine their results.

- This synchronous model is often characterized as explicit thread creation, but it is also an excellent candidate for implicit threading.

* 1. **OpenMP**

- OpenMP is a set of compiler directives as well as an API for programs written in C, C++, or FORTRAN that provides support for parallel programming in sharedmemory environments.

- OpenMP identifies parallel regions as blocks of code that may run in parallel.

* 1. **Grand central dispatch.**

- Grand Central Dispatch (GCD) is a technology developed by Apple for its macOS and iOS operating systems. It is a combination of a run-time library, an API, and language extensions that allow developers to identify sections of code (tasks) to run in parallel.

- GCD schedules tasks for run-time execution by placing them on a dispatch queue. GCD identifies two types of dispatch queues: serial and concurrent.

- Tasks placed on a serial queue are removed in FIFO order. Once a task has been removed from the queue, it must complete execution before another task is removed. Each process has its own serial queue (known as its main queue).

* 1. **Intel Thread Building Blocks**

- Intel threading building blocks (TBB) is a template library that supports designing parallel applications in C++. As this is a library, it requires no special compiler or language support.

1. **Threading Issues**
   1. **The fork() and exec() System Calls**
   2. **Signal Handling**
   3. **Thread Cancellation**
   4. **Thread-Local Storage**
   5. **Scheduler Activations**

Chapter 3: Sending and Receiving Data – TCP/IP SOCKETS in JAVA

**1 Encoding Information**

**1.1 Primitive Integers**

- We can encode the values of other (larger) primitive integer types. However, the sender and receiver have to agree on several things first. One is the size (in bytes) of each integer to be transmitted.

* For types that require more than one byte, we have to answer the question of which order to send the bytes in. There are two obvious choices: start at the right end of the integer, with the least significant byte—so-called little-endian order—or at the left end, with the most significant byte— big-endian order.

**1.2 Strings and Text**

- A mapping between a set of symbols and a set of integers is called a coded character set. For example, ASCII.

- Sender and receiver have to agree on a mapping from symbols to integers in order to communicate using text messages.

**1.3 Bit-Diddling: Encoding Booleans**

- Bitmaps are a very compact way to encode boolean information, which is often used in protocols.

- The idea of a bitmap is that each of the bits of an integer type can encode one boolean value—typically with 0 representing false, and 1 representing true.

- To be able to manipulate bitmaps, you need to know how to set and clear individual bits using Java’s “bit-diddling” operations. A mask is an integer value that has one or more specific bits set to 1, and all others cleared

**2 Composing I/O Streams**

- Java’s stream classes can be composed to provide powerful capabilities.

**3 Framing and Parsing**

- Framing refers to the problem of enabling the receiver to locate the beginning and end of a message.

- Whether information is encoded as text, as multibyte binary numbers, or as some combination of the two, the application protocol must specify how the receiver of a message can determine when it has received all of the message.

- If a receiver tries to receive more bytes from the socket than were in the message, one of two things can happen. If no other message is in the channel, the receiver will block and be prevented from processing the message; if the sender is also blocked waiting for a reply, the result will be deadlock. On the other hand, if another message is in the channel, the receiver may read some or all of it as part of the first message, leading to protocol errors. Therefore framing is an important consideration when using TCP sockets.

- Two general techniques enable a receiver to unambiguously find the end of the message:

+ Delimiter-based: The end of the message is indicated by a unique marker, an explicit byte sequence that the sender transmits immediately following the data. The marker must be known not to occur in the data.

+ Explicit length: The variable-length field or message is preceded by a (fixed-size) length field that tells how many bytes it contains.

**4 Java-Specific Encodings**

- When you use sockets, generally either you are building the programs on both ends of the communication channel—in which case you also have complete control over the protocol—or you are communicating using a given protocol, which you have to implement.

Chapter 4: Beyond the Basics – TCP/IP in JAVA

**1 Multitasking**

- Iterative server: If a client connects while another is already being serviced, the server will not echo the new client’s data until it has finished with the current client, although the new client will be able to send data as soon as it connects.

- We need some way for each connection to proceed independently, without interfering with other connections.

**1.1 Java Threads**

- It allows servers to handle many clients simultaneously. Using threads, a single application can work on several tasks concurrently.

- Java provides two approaches for performing a task in a new thread: 1) defining a subclass of the thread class with a run() method that performs the task, and instantiating it; or 2) defining a class that implements the Runnable interface with a run() method that performs the task, and passing an instance of that class to the thread constructor.

- When the start() method of an instance of thread is invoked, the JVM causes the instance’s run() method to be executed in a new thread, concurrently with all others. Meanwhile, the original thread returns from its call to start() and continues its execution independently.

**1.4 Thread Pool**

- Every new thread consumes system resources: spawning a thread takes CPU cycles and each thread has its own data structures (e.g., stacks) that consume system memory. In addition, when one thread blocks, the JVM saves its state, selects another thread to run, and restores the state of the chosen thread in what is called a context switch.

- As the number of threads increases, more and more system resources are consumed by thread overhead.

- We can avoid this problem by limiting the total number of threads and reusing threads. Instead of spawning a new thread for each connection, the server creates a thread pool on start-up by spawning a fixed number of threads. When the thread finishes with the client, it returns to the pool, ready to handle another request.

- Since each thread in the pool loops forever, processing connections one by one, a thread-pool server is really like a set of iterative servers.

**2 Blocking and Timeouts**

- Socket I/O calls may block for several reasons. Data input methods read() and receive() block if data is not available. A write() on a TCP socket may block if there is not sufficient space to buffer the transmitted data. The accept() method of ServerSocket() and the socket constructor both block until a connection has been established.

- A program that has other tasks to perform while waiting for call completion may have no time to wait on a blocked method call. What about lost UDP datagrams? If we block waiting to receive a datagram and it is lost, we could block indefinitely.

**2.1 accept(), read(), write(**)

- For these methods, we can set a bound on the maximum time (in milliseconds) to block, using the setSoTimeout() method of Socket, ServerSocket, and DatagramSocket. If the specified time elapses before the method returns, an InterruptedIOException is thrown. For Socket instances, we can also use the available() method of the socket’s inputStream to check for available data before calling read()

**2.2 Connecting and Writing**

- A write() call blocks until the last byte written is copied into the TCP implementation’s local buffer; if the available buffer space is smaller than the size of the write, some data must be successfully transferred to the other end of the connection before the call to write() will return. Thus, the amount of time that a write() may block is ultimately controlled by the receiving application.

**3 Mutiple Recipients**

- So far all of our sockets have dealt with communication between exactly two entities, usually a server and a client. Such one-to-one communication is sometimes called unicast

- Networks provide a way to use bandwidth more efficiently. Instead of making the sender responsible for duplicating packets, we can give this job to the network. In our video server example, we send a single copy of the stream across the server’s connection to the network, which then duplicates the data only when appropriate. With this model of duplication, the server uses only 1Mbps across its connection to the network, irrespective of the number of clients.

**3.1 Broadcast**

- Broadcasting UDP datagrams is similar to unicasting datagrams, except that a broadcast address is used instead of a regular (unicast) IP address.

- There is no networkwide broadcast address that can be used to send a message to all hosts. Sending a single datagram would result in a very, very large number of packet duplications by the routers, and bandwidth would be consumed on each and every network. The consequences of misuse (malicious or accidental) are too great, so the designers of IP left such an Internetwide broadcast facility out on purpose

**3.2 Multicast**

- As with broadcast, one of the main differences between multicast and unicast is the form of the address. A multicast address identifies a set of receivers. The designers of IP allocated a range of the address space dedicated to multicast.

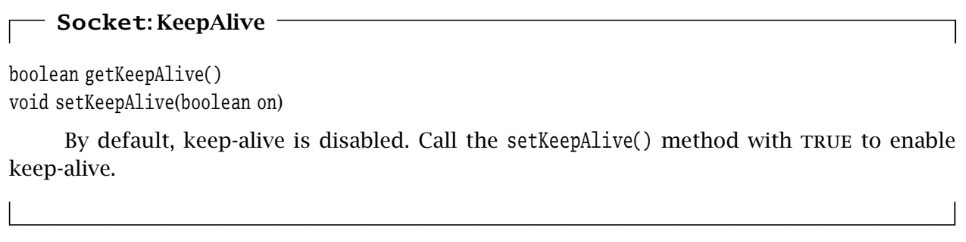
- The only significant differences between our unicast and multicast senders are that 1) we verify that the given address is multicast, and 2) we set the initial Time To Live (TTL) value for the multicast datagram. Every IP datagram contains a TTL, initialized to some default value and decremented (usually by one) by each router that forwards the packet. When the TTL reaches zero, the packet is discarded.

- UDP unicast, multicast, and broadcast are all implemented using an underlying UDP socket.

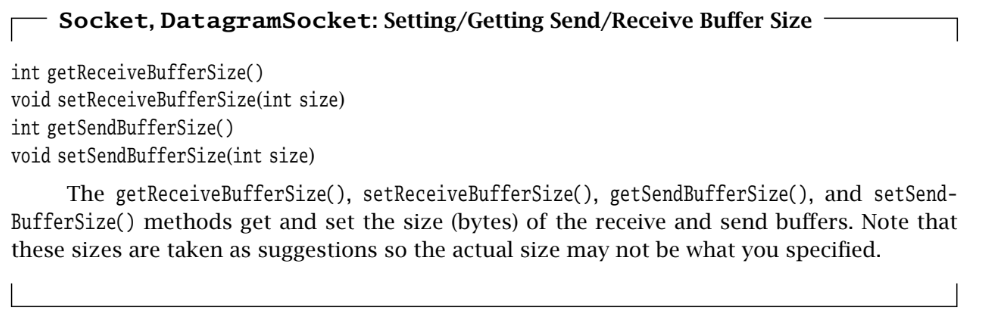
**4 Controlling Default Behaviors**

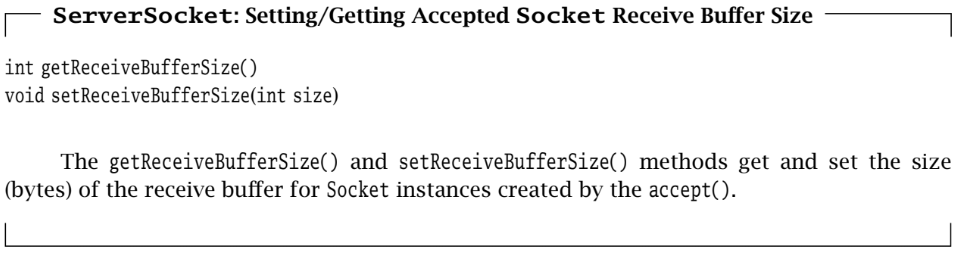
**4.1 Keep Alive**

- TCP provides a keep-alive mechanism where, after a certain time of inactivity, a probe message is sent to the other endpoint. If the endpoint is alive and well, it sends an acknowledgment. After a few retries without acknowledgment, the probe sender gives up and closes the socket, eliciting an exception on the next attempted I/O operation.



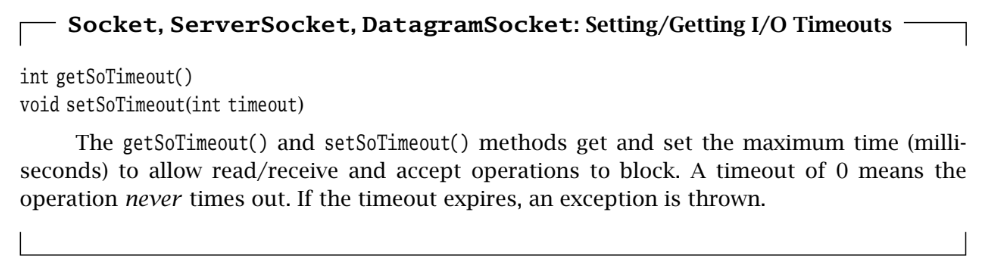
**4.2 Send and Receive Buffer Size**



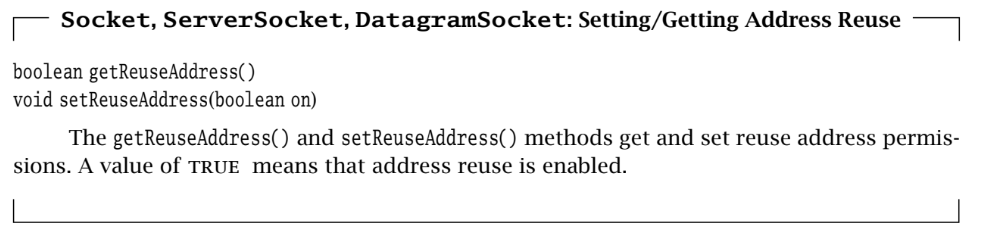


**4.3 Timeout**

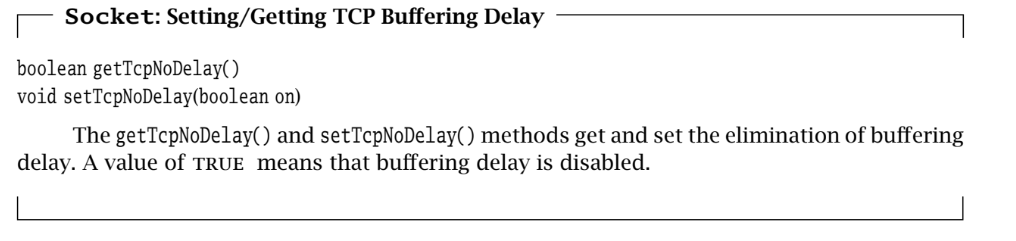
- As we’ve already seen, many I/O operations will block if they cannot complete immediately



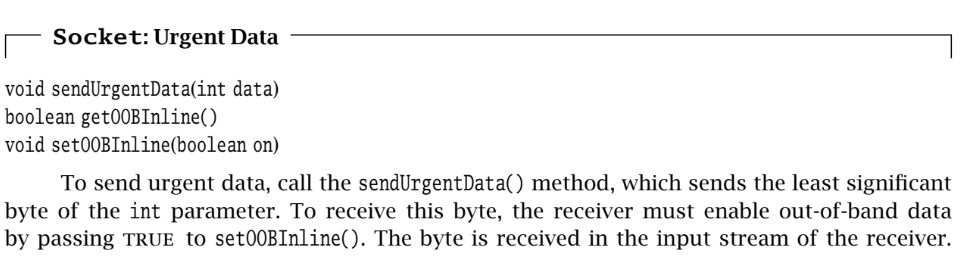
**4.4 Address Reuse**

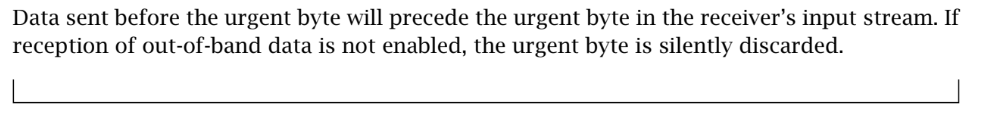


**4.5 Eliminating Buffer Delay**

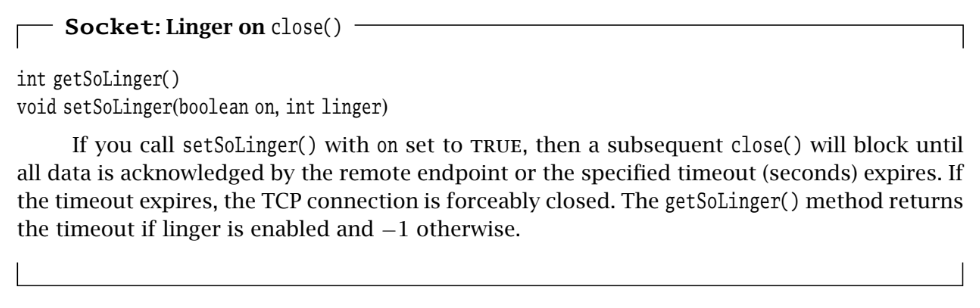


**4.6 Urgent Data**

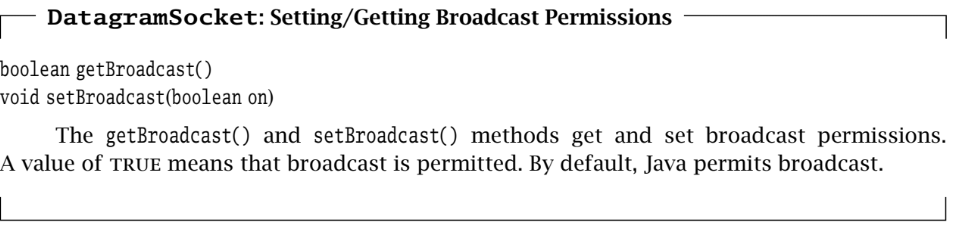




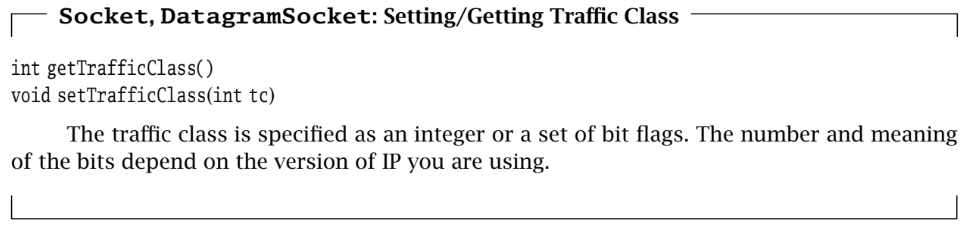
**4.7 Lingering after close**



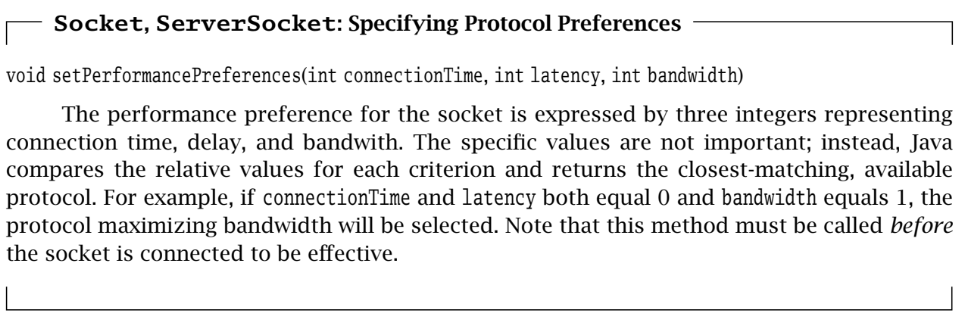
**4.8 Broadcast Permission**



**4.9 Traffic Class**



**4.10 Performance-Based Protocol Selection**



**5 Closing Connections**

- When the client is finished, it calls close(). After the server has received and echoed all of the data sent before the client’s call to close(), its read operation returns a−1, indicating that the client is finished. The server then calls close() on its socket.

- Calling close() on a Socket terminates both directions (input and output) of data flow. Once an endpoint (client or server) closes the socket, it can no longer send or receive data. This means that close() can only be used to signal the other end when the caller is completely finished communicating

**6 Applets**

- Applets can perform network communication using TCP/IP sockets, but there are severe restrictions on how and with whom they can converse. Without such restrictions, unsuspecting Web browsers might execute malicious applets that could, for example, send fake email, attempt to hack other systems while the browser user gets the blame, and so on.

- Typically, browsers only allow applets to communicate with the host that served the applet. This means that applets are usually restricted to communicating with applications executing on that host.