Basic Set of Material to Cover in Lecture on Chapter Four of 9th Edition of Silberschatz. THREADS

THREADS

OBJECTIVES

* introduce the concept of a thread
* describe APIs for Pthreads, Windows, and Java thread libraries
* discuss strategies that provide implicit threading
* issues related to multithreaded programming
* operating system support for threading in Windows and Linux

SECTION 4.1 - Overview

* Threads can share a lot of context, but each thread requires its own thread ID, program counter, register set, and stack. Threads can share code, data, open files, signals, and other OS resources.

4.1.1 - Motivation

* Many popular user programs are implemented as single multithreaded processes. Web browsers are examples of applications that utilize separate threads for such tasks as screen display, network communication, file I/O, and so forth. On multiprocessors, multithreading increases throughput.

* It is common for servers to utilize multiple threads, and to assign a separate thread to handle each client request.

* Operating Systems themselves are often multithreaded. Separate OS threads may exist for such tasks as device management, memory management, and/or interrupt handling.

4.1.2 - Benefits of Multithreading

1. **Responsiveness** - even if one thread is blocked or busy with a time-consuming task, another thread can respond to the user

2. **Resource Sharing** - threads within the same process share code and data by default - unlike separate processes.

3. **Economy** - Because the threads of a process share so much context, it is more efficient to create new threads within a process than to create new processes - there's just less to create. Context switching between threads of the same process is more efficient than context switching between processes - again because there's less context that needs to be switched.

4. **Scalability** - multiple threads within a process can exploit multiple CPUs within a multiprocessor.

Basic Set of Material to Cover in Lecture on Chapter Four of 9th Edition of Silberschatz. THREADS

SECTION 4.2 - Multicore Programming

* A system is parallel if it can perform more than one instruction simultaneously.

* A concurrent system may not be truly parallel. A concurrent system can provide the illusion of parallelism by time-sharing.

* Nowadays a single chip can have multiple CPUs (cores).

4.2.1 - Programming Challenges

* It is a challenge to design programs to exploit multicore capability:

1. Identifying tasks that can run concurrently
2. Figuring out how much CPU time to give to each task
3. Figuring out how to divide up data for the use of various threads
4. Synchronizing operations so that data dependencies are observed.
5. Verifying the correctness of parallel programs (having multiple possible interleaving execution paths).

* Many computer scientists feel that new approaches are needed in designing software, and increased emphasis on parallel processing.

4.2.2 - Types of Parallelism

* Data Parallelism

  + One example would be dividing an array into two halves and assigning separate threads to compute the sum of each half - simultaneously on different CPUs.

* Task Parallelism

  + An example of this would be one thread computing the minimum value of an array while another thread computes the average of the same array.

* Usually parallelism is a hybrid of the two basic types.

SECTION 4.3 - Multithreading Models

* There are **kernel level threads and user level threads**.

* User threads are supported above the kernel - implemented by code libraries.

* Kernel threads are supported directly by the operating system.

* Of course, at some level, kernel threads have to support user level threads.

Basic Set of Material to Cover in Lecture on Chapter Four of 9th Edition of Silberschatz. THREADS

4.3.1 - Many-to-One Model

* In this model, one kernel level thread supports a group of user-level threads.

* This model is pretty simple to implement but it does not allow parallelism, so it is not popular now.

* If one user-level thread makes a blocking system call, then the OS has to suspend the supporting kernel thread, which prevents the other user-level threads from executing.

4.3.2 - One-To-One Model

* In this model, each user-level thread has a supporting kernel thread that it does not have to share with any other user-level thread.

* This allows parallelism and each user thread can block independently.

* However user thread creation goes rather slowly because it always requires the creation of a new kernel thread.

* Also, having large number of kernel threads can be a drain on system resources.

4.3.3 - Many-To-Many Model

* In this model, a group of user threads is supported by a group of kernel threads. Generally the number of kernel threads is is less than the number of user-level threads.

* The model allows user-level threads to be created without creating more kernel level threads.

* The kernel threads can run in parallel on a multiprocessor.

* User level threads can migrate between supporting kernel threads, or be "pegged" to one specific kernel thread.

SECTION 4.4 - Thread Libraries

* Thread libraries may implement user-level threads or kernel level threads.

* The sharing of memory among threads is implemented in different ways, but often global variables are shared by all threads and local variables are not shared.

* Parent threads may run concurrently with child threads, or wait for them to exit.

* Parents running concurrently with child threads may or may not communicate and/or

share data with their children.

4.4.1 - Pthreads

* Pthreads *is a specification* for an API that can be implemented in various ways - for example, sometimes it is implemented with user threads, and sometimes with kernel threads.

* Parent threads creating child threads specify a function in which the child is to begin its execution.

* Parents and children share global variables, but not local variable.

4.4.2 - Windows Threads

4.4.3 - Java Threads

SECTION 4.5 - Implicit Threading

* Compilers can help protect program correctness when multiple threads are utilized.

4.5.1 - Thread Pools

* It takes time for a server to create a thread to handle a client request, and it's a good idea to place a limit on the number of threads operating in a server.

* Instead a server can use a thread pool - create a set of threads to use and re-use for handling clients.

* Clients have to wait if all the threads in the pool are busy with other clients.

* The Windows thread API supports thread pools.

4.5.2 - OpenMP

* The programmer can identify blocks of code as parallel regions.

* The compiled code creates a thread for each core to execute the region in parallel.

* Programmers can call for parallelizing array processing in loops.

* OpenMP can be used on Windows, Linux, MacOS X, and other systems.

4.5.3 - Grand Central Dispatch

* GCD runs under MacOS X and iOS.

* It includes extensions to C, an API, and a run-time library.

* The programmer can identify blocks of code to be executed in parallel.

* The blocks can be assigned relative scheduling priorities.

4.5.4 - Other Approaches

* Intel's Threading Building Blocks
* Several Microsoft procucts
* The java.util.concurrent package

SECTION 4.6 - Threading Issues

4.6.1 - The fork() and exec() System Calls

* If a single thread in a program calls fork(), how many of the threads in the calling process should it duplicate?

* Some version of unix have variants of fork(), so that the calling thread can duplicate all threads of the process, or just itself.

* Typically if one thread calls exec to run a program, the program will replace the entire calling process - including all its threads.

* Therefore, often a thread will duplicate only itself with fork() if it intends to call exec next, but if not it may call the version of fork() that duplicates the whole process.

4.6.2 - Signal Handling

* Signals are a form of message passing that have been used in unix-like operating systems for a very long time.

* Signals were originally designed to be sent and received by single-threaded processes.

* For multithreaded processes, questions come up about which threads belonging to a process should receive a sent signal.

  + the thread to which it applies?
  + all threads in the process?
  + certain threads?
  + assign one thread to receive all signals?

* It is clear that if a thread 'causes a problem' then usually it is the thread that should receive the signal.

* Some signals, like 'terminate', should be sent to all the threads in the process.

* If a signal should be handled only once, then it makes sense to select one thread that is not blocking it to receive it.

* Using the Pthreads API, it is possible to send a signal to a specific thread.

* Windows has a facility similar to signaling - *asynchronous procedure calls*. APCs were designed

to be sent and received by individual threads.

4.6.3 - Thread Cancellation

* A Pthreads thread can execute a function to cancel (terminate) another thread.

* The Pthreads API provides for allowing threads to defer cancellation so that they can release resources first.

4.6.4 - Thread-Local Storage

* A thread may have need of data that is not local to any function, but which is also not shared with other threads. That's the idea of thread-local storage.

4.6.5 - Scheduler Activations

* In the implementation of the relationship between user and kernel threads, the OS may use an intermediate data structure called a lightweight process (LWP).

* The LWP appears as a virtual processor to the user-thread library - something onto which user thread may be scheduled for execution.

* Each LWP is attached to a kernel thread

* Upcalls, which work somewhat like signals from the kernel to the user thread library, help implement the switching of user thread context when a user thread makes a blocking system call, and with starting blocked user threads up again.

* Upcalls execute on LWPs.

SECTION 4.7 - Operating System Examples

4.7.1 - Windows Threads

* Windows applications can be multithreaded.

* Windows uses the one-to-one model.

* The text lists various components of the Windows thread context.

4.7.2 - Linux Threads

* Linux does not use the term process or thread. The Linux term is task.

* The clone() system call can be used to copy a task, and parameters control the degree of sharing that the child task has with the parent.

* Thus the parent can create a lightweight child task that is basically the same as a thread, but it also can create a child that shares no resources, which is basically the same as forking a traditional copy of an entire process.