Cheng | Midterm 1: Chapter 4 | Study Guide

Chapter 4: Threads

Thread Defintion:

Example: Take a thread word-processor program. A single-thread word-processor program allows the process to perform only one task at a time. A user can not type characters and run a spell checker at the same time. But, modern operatins sytems have 'stepped-they're-game-up' and allow a proces to perform multiple tasks at once so multiple threads in parallel. A multi-threaded word-processor can assign one thread to manage user input while another thread runs spell checker.

Thread is a set of instructions or a task that is excuted by a process. The more threads, the more tasks a process can accomplish. Multiple threads within a process will share the address space, open files, and other resources.

There are *hella* benefits to multi-threaded programming. They are:

1. Responsiveness: Multi-threading an application that is "interactive" allows a program to continue to run even if part of it is blocked or performing a lengthy operation, the user feels an increased responsiveness.
2. Resource-sharing: Processes share their resources through things that have to be arranged by the programmer, threads share memory and resources of a process by default.
3. Economy: Instead of creating new processes, which is costly for memory and resource, it saves memory and resource to just create a new thread, the overhead might be garbage with switching between threads but it has faster context switching then with processes.
4. Scalability: Threads can run in parallel on different processing cores. A single-threaded process can run on only one processor despite how many are avaliable.

Concurrency and Parallelism:

A concurrent system supports more than one task by allowing all the tasks to make progress. In contrast, a parallel system can perform more than one task simultaneously. Thus, it is possible to have concurrency without parallelism.

Multi-threading Models: A relationship between user-level threads and kernel-level threads must exist.

Many-to-one: Maps many user-level threads to one kernel thread. Only one thread can access the kernal at a time, mulitple threads are unable to run in parallel on multicore systems.

One-to-one: Maps each user thread to a kernel thread. This allows multiple threads to run in parallel on multi-processors. And there is also concurrency because it allos another thread to run when one thread makes a blocking system call. Only drawback, there are a large number of kernel threads that can burden a system.

Many-to-many: Multiplexes many user-level threads to smaller of equal number of kernel threads. The many-to-many model suffers from neither of these shortcomings: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. The number of kernel threads does not have to equal the number of user threads like the one-to-one model. Instead, the number can be equal or less.

User-level & Kernel-level Threads:

User-level Thread: User threads are supported above the kernel and are managed without kernel support. This is a thread that the operating system does not know about. The OS only knows about the process, so it can schedule the process but not the threads within the process. There is no context switch, this is faster to switch then a kernel-level thread!

Kernel-level Threads: Kernel threads are supported and managed directly by the operating system.This is a thread that the operating system knows about. It is a lightweight process because the memory management information doesn't need to change, this is a fast context switch, but still a context switch.

Well, what does support mean? This means the management and schedung of the threads.

Thread Library: Thread management is done by the thread library in user space. It provides a programmer with an API for creating and managing threads.

There are two ways to implement a thread library:

1. Provide a library entirely in user space with no kernel support. The code and data structures exist in user space and invokes local function calls to access the API.
2. Implement a kernel-level library supported directly by the OS. The code and data structures exist in the kernel space and that means invoking a function in the API requires a system call to the kernel.

Implicit Threading: With the growth of multi-core processing, the growth of threads is increasing and so is the impact of their difficulties and challenges.

> *One way to address these difficulties and better support the design of concurrent and parallel applications is to transfer the creation and management of threading from application developers to compilers and run-time libraries. This strategy, termed implicit threading, is an increasingly popular trend.*

The solution with dealing with all these problems? Let the compilers and run-time libraries deal with thead creation. The advantage of this approach is that developers only need to identify parallel tasks, and the libraries determine the specific details of thread creation and management.

Thread Pools:

The general idea behind a thread pool is to create a number of threads at start-up and place them into a pool, where they sit and wait for work. When a server receives a request, rather than creating a thread, it instead submits the request to the thread pool and resumes waiting for additional requests. If there is an available thread in the pool, it is awakened, and the request is serviced immediately. If the pool contains no available thread, the task is queued until one becomes free. Once a thread completes its service, it returns to the pool and awaits more work.

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. Application developers insert compiler directives into

their code at parallel regions, and these directives instruct the OpenMP run- time library to execute the region in parallel. This creates as many threads as there are processing cores in the system.

Grand Central Dispatch:

Similar to OpenMP:

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. Like OpenMP, GCD manages most of the details of threading.

Similar to Thread Pool:

GCD schedules tasks for run-time execution by placing them on a dispatch queue. When it removes a task from a queue, it assigns the task to an available thread from a pool of threads that it manages.

But different in that it has two types of dispatch queues: serial and concurrent.

Serial: Tasks that are placed in the serial queue are removed in FIFO order. Once a task has be removed from the queue, it must complete execution before another task is removed. Good in ensuring the sequential execution of tasks!

Concurrent: Several tasks may be removed from the queue at a time, allowing multiple tasks to execute in parallel.

Threading Issues: The semantics of the fork() and exec() system cals change in a mulit-threaded program.

Fork(): If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded? Some UNIX systems have two versions of fork(), one that duplicates all hteads and another that dupicates only the thread that invokes the fork() system call.

Exec(): If a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process- including all threads.