4 Threads and Concurrency

Discussion in Chapter 3 assumes a process was an executing program with a signle thread of control (, or single-threaded). Most OSs now enables a process to contain multiple threads of control (, or multithreaded process).

4.1 Overview

A thread is a basic unit of CPU utilization. Threads belonging to the same process

	owns	shares
1	a thread ID	code section (`CS`)
2	a program counter (`EIP`)	data section (`DS`)
3	a register set (`EAX-EDX ESI EDI`)	other OS resources
4	a stack (`ESP EBP`)	

Figure 4.1: Resources owned or shared by threads.

Multithreaded programming has lots of benefits, including

- increased responsiveness: programs can continue even if partial functionality is blocked;
- threads belonging to the same process share resources $\Rightarrow$ decrease # of duplication;
- Economy: cost to create a thread & context switch < cost to create a process & context switch;
- Scalability: multiple threads can run on different cores in the same CPU.

4.2 Multicore Programming

Multicore is to place multiple computing cores on a single processing chip.
Concurrency is to allow multiple tasks to make progress;
Parallelism is to perform multiple tasks to run simultaneously.

Concurrency $\not\Rightarrow$ Parallelism
Parallelism $\Rightarrow$ Concurrency

1. Challenges to better utilize multicore property:
   - correctly identifing tasks that can be divided into separate ($\Leftrightarrow$ tasks that independent of one another);
   - ensuring multiple cores to perform equal work (*or: one busy, one idle*);
   - correctly identifing independent data access and/or manipulation;
   - testing and debugging.
2. Types of Parallelism
   - Data Parallelism: distributing data across multiple computing cores;
   - Task Parallelism: distributing task across multiple computing cores.

4.3 Multithreading Models

A thread *in a general sense* is supported by

- user thread (or part): supported above the kernel and managed without kernel support;
- kernel thread (or part): supported and directly managed by the OS.

Relationships exist between user threads and kernel threads, including

1. many-to-one: all user threads map to one kernel thread. (threads cannot run in parallel)
2. one-to-one: each user thread has a corresponding kernel thread. (more concurrency, but easily overhead)
3. many-to-many: user threads multiplexes # $\leq$ kernel threads;
   A variation: two-level model, enables a user-level thread to be bound to a kernel thread in many-to-many model.

$\Rightarrow$ MTM: most flexible, but difficult to implement; others: easier to implement, but are limited.

4.4 Thread Library

A thread library provides APIs for managing threads. It can be located in either user space (function call) or kernel space (system call). There are two general strategies for thread creation:

- asynchronous threading: the parent resumes its execution after creation ($\Rightarrow$ independent tasks);
- synchronous threading: the parent must wait for all its children to terminate ($\Rightarrow$ significant sharing).

`pthread`	`Windows`	`java.lang.Thread`
`pthread_create`	`CreateThread`	`new Thread(...)`
`pthread_join`	`WaitForSingleObject`	`wait` method
`pthread_exit`	`TerminateThread`	`interrupt` method
...	...	...

4.5 Implicit Threading

Unlimited threads is not a trivial undertaking, like system resources overhead. One way to address this problem is to transfer the creation and management of threading to compilers and run-time libraries, which is termed implicit threading, requiring application developers to identify **tasks** rather than threads.

4.5.1 Thread Pools

The general idea is to create a number of threads (*pool*) at start-up. They sit and wait for work at first. When a **task** comes, if some thread is waiting, it will be awakened, otherwise the task will be queued.

Benefits:

- The OS saves much time on creating threads;
- Decreased thread creation saves OS's resources;
- Separating the task to be performed from the mechanics of creating the task allows us to use different strategies for running the task. (e.g., delay or periodically)

The number of threads in the pool can be set heuristically based on factors of the application programs.

4.5.2 Fork Join

- `fork`: duplicates the current process, and returns $\begin{cases}\text{pid of the created process to the creating process}\\ \text{0 to the created process}\end{cases}$
- `exec`: replaces the specified program or commands on the current context.

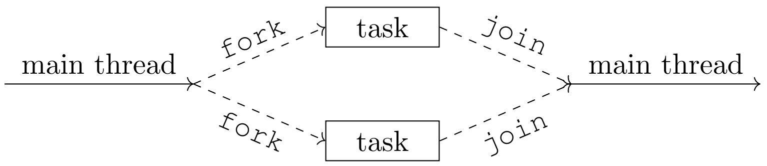

Figure 4.2: Fork-join Parallelism

4.5.3 OpenMP

OpenMP is a set of compiler directives as well as an API for programs that provides support for parallel programming in shared-memory environment. Developers insert compiler directives into their code at parallel regions, and these derivatives instruct the OpemMP run-time library to execute the region in parallel.

4.5.4 Grand Central Dispatch

GCD schedules tasks by placing them on dispatch queue. It identifiesc two types of dispatch queues: serial and concurrent. Both of them are FIFO order, but serial queue must remove the next task **after execution**, while several tasks might be removed at once in parallel queues.

4.5.5 Intel Thread Building Blocks

Intel TBB is a template library that supports designing parallel applications in C++. Developers specify tasks that can run in parallel, and the TBB task scheduler

- maps tasks onto underlying threads;
- provides load balancing;
- is cache aware (give precedences to tasks based on how likely tasks will cache data);
- provides various templates;
- ...

4.6 Threading Issues

4.6.1 `fork` and `exec`

If a thread calls `fork`, `fork` can duplicate the current thread or all threads, while both are useful. Most OS now supports both. Furthermore, `exec` would replace all duplicated threads with the new program. For example, if `exec` is called immediately after `fork`, duplication of all threads is unnecessary.

4.6.2 Signal Handling

Why authors put signals at here? I don't understand.

4.6.3 Thread Cancellation

Thread cancellation involves terminating a thread before it has completed. The thread to be canceled is refered to as the target thread. Cancellation of a thread occurred in two scenarios:

- asynchronous cancellation: the target thread is canceled immediately;
- deferred cancellation: the target thread periodically check (from flags) whether it should terminate. (useful when the target thread owns OS resources or is sharing resources with other threads.)

4.6.4 Thread-Local Storage

Thread-local storage, or TLS, is to copy static or global data to local thread storage. (e.g., process's data)

4.6.5 Scheduler Motivation

Exercises

1. - the server processing multiple requests from multiple clients;
   - the user can starts another interaction when the computer is responding to the current interaction;
   - If we need result from multiple operations on the same datasets, we can use multiple threads for each.
3. Task parallelism. The server creates a thread for each user. The server does similar operation for each kind of requests, but responds to distinctive senders.
4. User-level threads are supported above the kernel and are managed without the kernel, while kernel-level threads are supported and directly managed by the kernel.
5. (The author did not talked about it.) Save old register values, set new register values.
6. thread ID, stack, register set, program counter; pid, pcb (page tables, files, etc.), stack, register set, program counter.
7. (TBC)
8. - Code in each thread has certain priorities that must be followed (locks, critical sections, etc.);
   - Two threads communicating with each other must wait for responds to continue.
9. Better performance if different codes and/or data are independent of each other. They can be executed and/or processed separately in parallel.
10. BC
11. Yes if multiple threads can run on different processors, which is supported by modern CPUs. However, normally, no, because a multithread solution stll belongs to the same process, which means they lie on the same processors. The performance theoretically is the same.
12. Probably not. Firstly, the OS would prevent a process from accessing another process's data. If each tab is a process, if a tab crashes, it might influence some other tab(s). Secondly, if the processor is not multicore, when one tab is processing data, other tab would be blocked.
13. No. Multiple tasks executing simultaneously without making progress together does not make sense.
14. Computation
15. (1) data parallelism ($\Leftarrow$ multiple threads share the same task, but with independent input/data);
    (2) task+data parallelism ($\Leftarrow$ we can assign each column row, even each elements in parallel);
    (3) task parallelism (and probably data parallelism if two threads does not operate on the same data);
    (4) data parallelism
    (5) task parallelism