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.

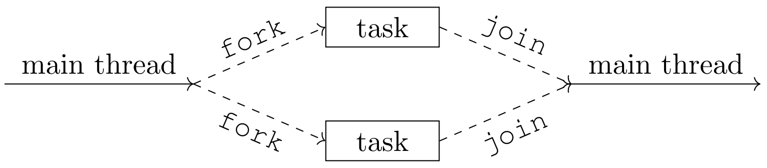

Figure 4.2: Fork-join Parallelism

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

5 CPU Scheduling

5.1 Basic Concept

In multiprogramming, as one process need to wait for some event, another process can takes over use of the CPU. This increases utilization of CPU. This can be extended to multicore architectures.

Processes alternate between

- CPU burst executing successive instructions;
- I/O burst waiting for I/O events.

An I/O-bound program has many short CPU bursts. A CPU-bound program have a few long CPU bursts. The measurement of bursts can be important when implementing a CPU-scheduling algorithm. When a CPU becomes idle, the OS would select a process from the ready queue to be executed, by CPU scheduler.

- nonpreemptive scheduling: the CPU executes till the process needs to wait for some events;
  $\Rightarrow$ probably cannot complete all tasks within a given time frame.
- preemptive scheduling: the execution on the CPU might be interrupted at any time ($\Rightarrow$ race conditions).
  $\Rightarrow$ probably results in **race conditions** (different processes sharing data or important kernel data)
  $\Rightarrow$ mechanisms, like mutex locks, to prevent race conditions

As the process to execute is confirmed, dispatcher is to give control of the CPU's core to the process selected by the CPU scheduler, contains

- switching context;
- switching to user mode;
- Jumping to the proper location in the user program to resume the program.

The time for the dispatcher from stoping one process to resuming another process is called dispatch latency.

5.2 Scheduling Criteria

Different CPU-scheduling algorithms may favor different classes of processes, and it is hard to balance all classes of processes. Lots of criteria are used to compare algorithms, including

CPU utilization rate	↑	how busy it is
Throughput	↑	# processes completed per time unit
Turnaround time	↓	time to execute some specific process
Waiting time	↓	time for some specific process to wait on the ready queue
Response time	↓	time for some specific process to start responding

However, sometimes we need to optimize minimum and maximum, or ***variance***, rather than the average.

5.3 Scheduling algorithms (of single-processor system)

1. **First-come first-serve (FCFS)**: The process that requests the CPU first is allocated the CPU first.
   - Pros: simple to implement and understand.
   - Cons: long average waiting time, which might make I/O devices idle (convoy effect).
2. **Shortest-Job-First (SJF)**: The CPU is assigned to the process with the smallest next CPU burst.
   - Pros: decreases the waiting time (of short processes more than increase the waiting time of long)
   - Cons: evaluation of the next CPU burst ($\Rightarrow$ prediction: $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$, t_n, α_n: real/predicted)
   $$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \cdots + (1-\alpha)^j \alpha t_{n-j} + \cdots + (1-\alpha)^{n+1}\tau_0 = \alpha t_n + \sum_{i=0}^{n} \tau_i (1-\alpha)^{n-i+1}.$$
   - SJF could be preemptive, which means it will preempt the current process.

3. **Round-Robin (RR)**: The CPU goes around the ready queue, allocating the CPU to each process shorter than 1 time quantum. (Each process may either release voluntarily or be preempted as time quantum is reached.)