Distributed Systems

(3rd Edition)

Chapter 03: Processes

Version: February 25, 2017

Introduction to threads

Basic idea

We build virtual processors in software, on top of physical processors:

Processor: Provides a set of instructions along with the capability of automatically executing a series of those instructions.

Thread: A minimal software processor in whose context a series of instructions can be executed. Saving a thread context implies stopping the current execution and saving all the data needed to continue the execution at a later stage.

Process: A software processor in whose context one or more threads may be executed. Executing a thread, means executing a series of instructions in the context of that thread.

Context switching

Contexts

- Processor context: The minimal collection of values stored in the registers of a processor used for the execution of a series of instructions (e.g., stack pointer, addressing registers, program counter).
- Thread context: The minimal collection of values stored in registers and memory, used for the execution of a series of instructions (i.e., processor context, state).
- Process context: The minimal collection of values stored in registers and memory, used for the execution of a thread (i.e., thread context, but now also at least MMU register values).

Context switching

Observations

1. Threads share the same address space. Thread context switching can be done entirely independent of the operating system.
2. Process switching is generally (somewhat) more expensive as it involves getting the OS in the loop, i.e., trapping to the kernel.
3. Creating and destroying threads is much cheaper than doing so for processes.

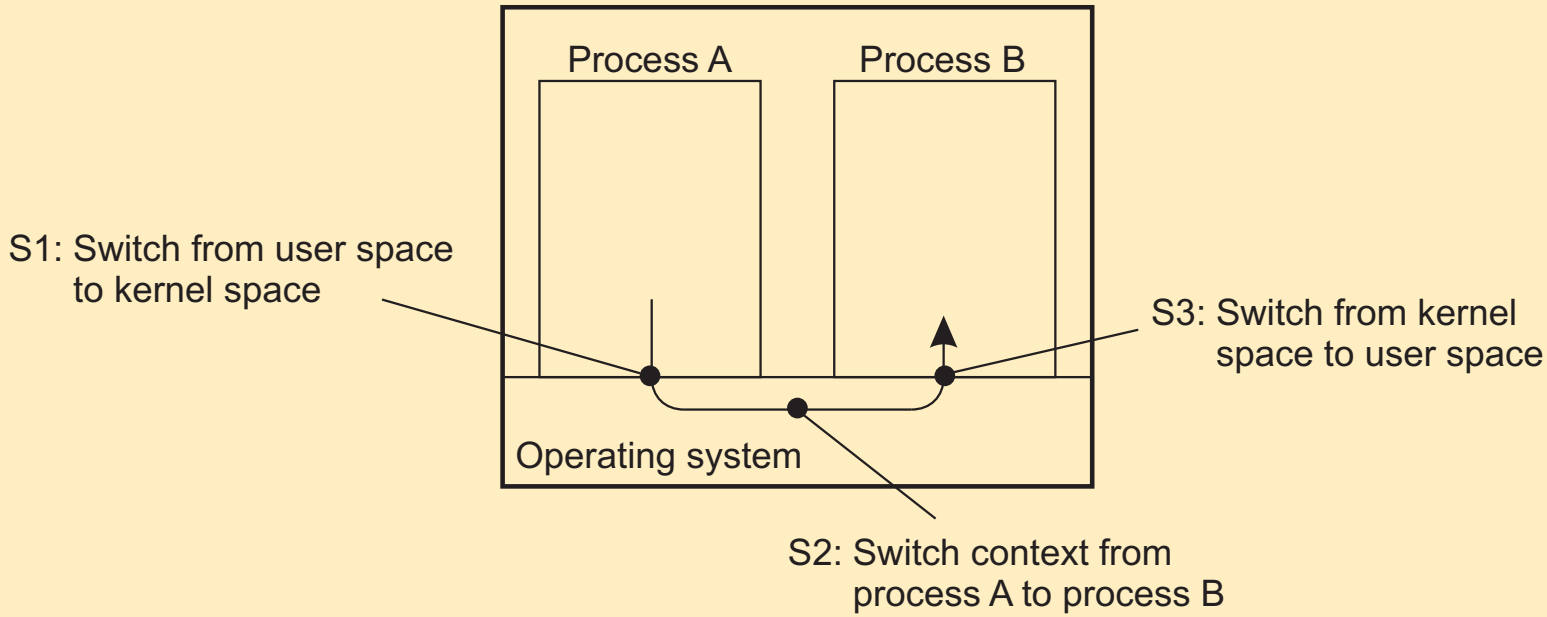
Why use threads

Some simple reasons

- Avoid needless blocking: a single-threaded process will block when doing I/O; in a multi-threaded process, the operating system can switch the CPU to another thread in that process.
- Exploit parallelism: the threads in a multi-threaded process can be scheduled to run in parallel on a multiprocessor or multicore processor.
- Avoid process switching: structure large applications not as a collection of processes, but through multiple threads.

Avoid process switching

Avoid expensive context switching



Trade-offs

- Threads use the same address space: more prone to errors
- No support from OS/HW to protect threads using each other's memory
- Thread context switching may be faster than process context switching

Threads and operating systems

Main issue

Should an OS kernel provide threads, or should they be implemented as user-level packages?

User-space solution

- All operations can be completely handled within a single process $\Rightarrow$ implementations can be extremely efficient.
- All services provided by the kernel are done on behalf of the process in which a thread resides $\Rightarrow$ if the kernel decides to block a thread, the entire process will be blocked.
- Threads are used when there are lots of external events: threads block on a per-event basis $\Rightarrow$ if the kernel can't distinguish threads, how can it support signaling events to them?

Lightweight processes

Principle operation

- User-level thread does system call ⇒ the LWP that is executing that thread, blocks. The thread remains bound to the LWP.
- The kernel can schedule another LWP having a runnable thread bound to it. Note: this thread can switch to any other runnable thread currently in user space.
- A thread calls a blocking user-level operation ⇒ do context switch to a runnable thread, (then bound to the same LWP).
- When there are no threads to schedule, an LWP may remain idle, and may even be removed (destroyed) by the kernel.

Note

This concept has been virtually abandoned – it's just either user-level or kernel-level threads.

Using threads at the client side

Multithreaded web client

Hiding network latencies:

- Web browser scans an incoming HTML page, and finds that more files need to be fetched.
- Each file is fetched by a separate thread, each doing a (blocking) HTTP request.
- As files come in, the browser displays them.

Multiple request-response calls to other machines (RPC)

- A client does several calls at the same time, each one by a different thread.
- It then waits until all results have been returned.
- Note: if calls are to different servers, we may have a linear speed-up.

Using threads at the server side

Improve performance

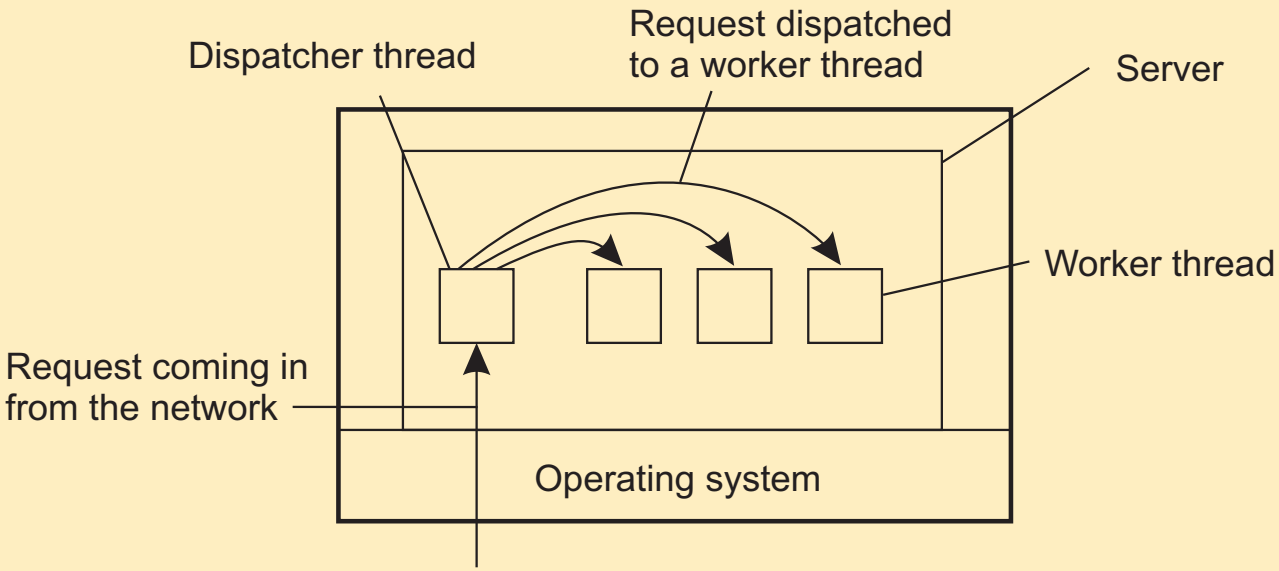
- Starting a thread is cheaper than starting a new process.
- Having a single-threaded server prohibits simple scale-up to a multiprocessor system.
- As with clients: hide network latency by reacting to next request while previous one is being replied.

Better structure

- Most servers have high I/O demands. Using simple, well-understood blocking calls simplifies the overall structure.
- Multithreaded programs tend to be smaller and easier to understand due to simplified flow of control.

Why multithreading is popular: organization

Dispatcher/worker model



Overview

Model	Characteristics
Multithreading	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls