Threads Implementation

Jo, Heeseung

Today's Topics

How to implement threads?

- User-level threads
- Kernel-level threads

Which one is the fastest?

Get factorial value of N

- iterative factorial
- recursive factorial

iterative 방식이 유리하다.

함수를 호출하고 들어갈 때마다 작업이 늘어나기 때문이다.

local function

- myswap()

다른 함수 호출이 없어 오버헤드가 덜 발생함.

library function

- strcpy()

system call

- getpid()

Kernel/User-level Threads

Who is responsible for creating/managing threads?

- The OS (kernel-level threads)
  - Thread creation and management requires system calls
- The user-level process (user-level threads)
  - A library linked into the program manages the threads

Why is user-level thread management possible?

- Threads share the same address space
  - The thread manager doesn't need to manipulate address spaces
- Threads only differ in hardware contexts (roughly)
  - PC, SP, registers
  - These can be manipulated by the user-level process itself

Kernel-level Threads (1)

OS-managed threads

- The OS manages threads and processes
- All thread operations are implemented in the kernel
- The OS schedules all of the threads in a system
  - If one thread in a process blocks (e.g., on I/O), the OS knows about it, and can run other threads from that process
  - Possible to overlap I/O and computation inside a process
- Kernel threads are cheaper than processes
  - Less state to allocate and initialize
- Windows, Solaris, Tru64 Unix, Linux, MacOS

Kernel-level Threads (2)

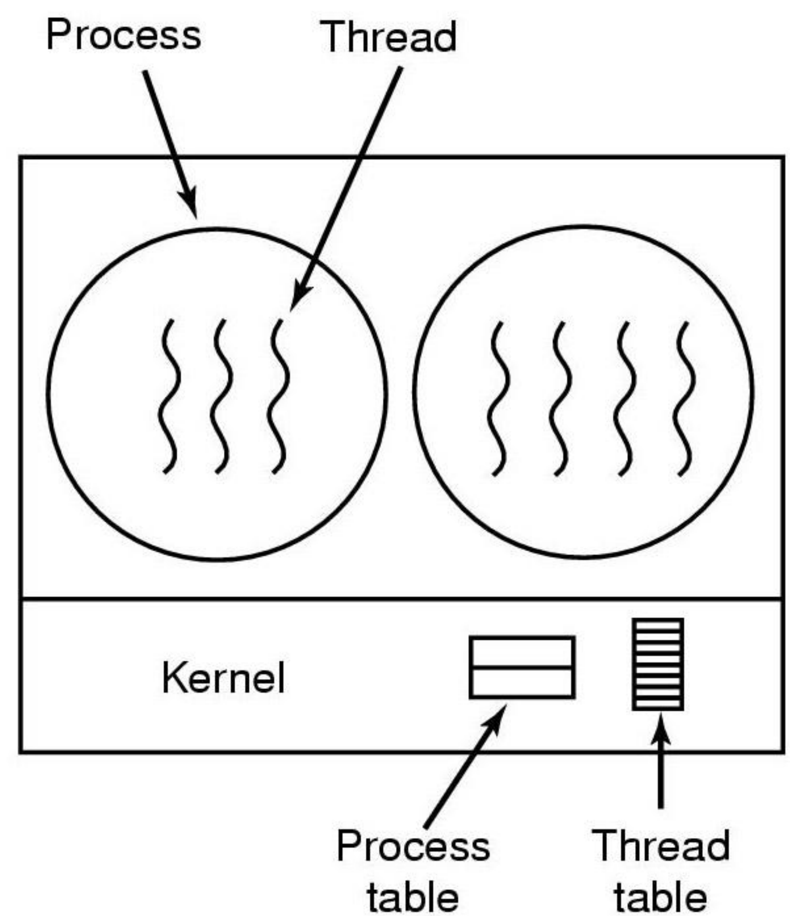
Limitations

- They can still be too expensive
  - For fine-grained concurrency, we need even cheaper threads
  - Ideally, we want thread operations as fast as a procedure call
- Thread operations are all system calls
  - The program must cross an extra protection boundary on every thread operation
  - Even when the processor is being switched between threads in the same address space
  - The OS must perform all of the usual argument checks
- Must maintain kernel state for each thread
  - Can place limit on the number of simultaneous threads
  - In Linux, 256430 (/proc/sys/kernel/threads-max)
- Kernel-level threads have to be general to support the needs of all programmers, languages, runtime systems, etc.

Implementing Kernel-level Threads

Kernel-level threads

- Kernel-level threads are similar to original process management and implementation



User-level Threads (1)

Motivation

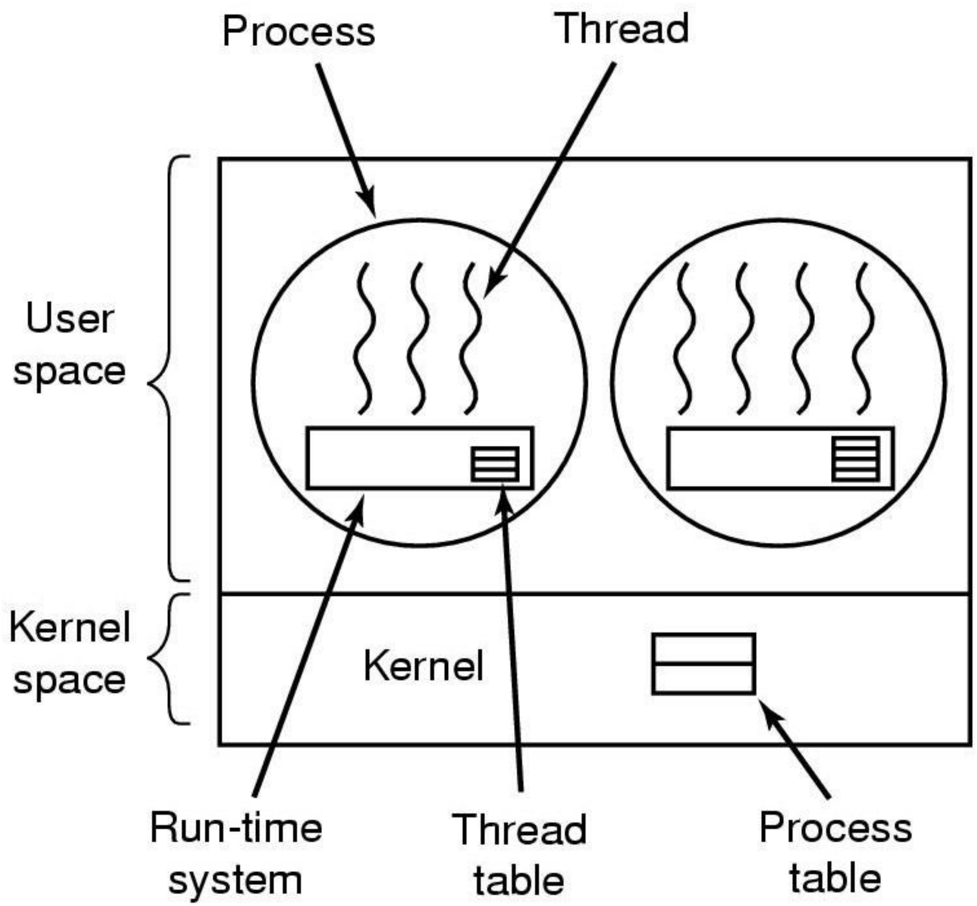
- To make threads cheap and fast, they need to be implemented at the user level
- Portable: User-level threads are managed entirely by the runtime system (user-level library)

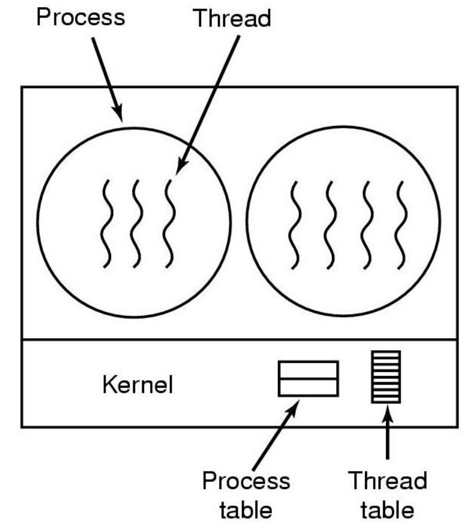
User-level threads are small and fast 커널 레벨 스레드에 비해 훨씬 더 작고 빠름.

- Each thread is represented simply by a PC, registers, a stack, and a small thread control block (TCB)
- Creating a thread, switching between threads, and synchronizing threads are done via procedure calls (No kernel involvement)
- User-level thread operations can be 10-100x faster than kernel-level threads

Implementing User-level Threads (1)

User-level threads



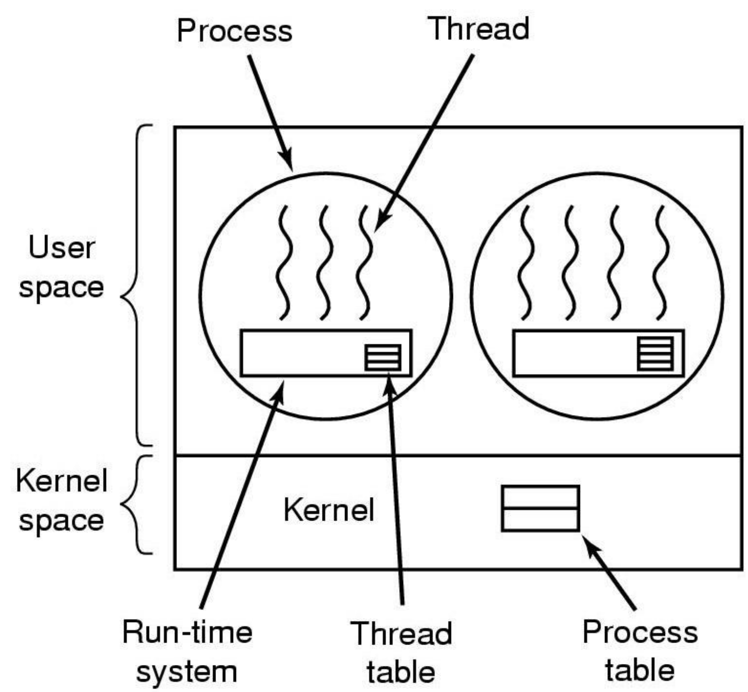


Kernel thread

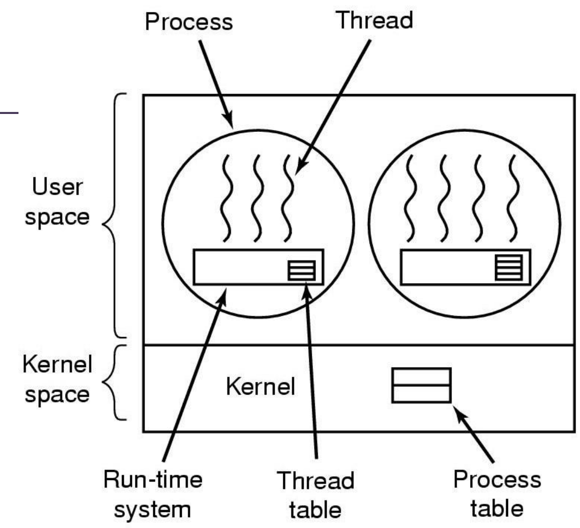
Implementing User-level Threads (2)

Thread context switch

- Very simple for user-level threads
- Save context of currently running thread
  : push all machine state onto its stack
- Restore context of the next thread
  : pop machine state from next thread's stack
- The next thread becomes
  the current thread
- Return to caller as the new thread
  : execution resumes at PC
  of next thread



User-level Threads (2)



Limitations

- User-level threads are invisible to the OS
  - They are not well integrated with the OS
- As a result, the OS can make poor decisions
  - Scheduling a process with only idle threads
  - Blocking a process whose thread initiated I/O, even though the process has other threads that are ready to run
  - Unscheduling a process with a thread holding a lock
- Solving this requires coordination between the kernel and the user-level thread manager
  - e.g., all blocking system calls should be emulated in the library via non-blocking calls to the kernel

Implementing User-level Threads (3)

Thread scheduling

- A thread scheduler determines when a thread runs
  - Just like the OS and processes
  - But implemented at user-level as a library
- Queues to keep track of what threads are doing
  - Run queue: threads currently running
  - Ready queue: threads ready to run
  - Wait queue: threads blocked for some reason (maybe blocked on I/O or a lock)

- How can we prevent a thread from hogging the CPU?

Implementing User-level Threads (4)

Non-preemptive scheduling

- Force everybody to cooperate
  - Threads willingly give up the CPU by calling yield()
- yield() calls into the scheduler, which context switches to another ready thread

```
Thread ping ()
{
  while (1) {
    printf ("ping\n");
    yield();
  }
}
```

```
Thread pong ()
{
  while (1) {
    printf ("pong\n");
    yield();
  }
}
```

- What happens if a thread never calls yield()?

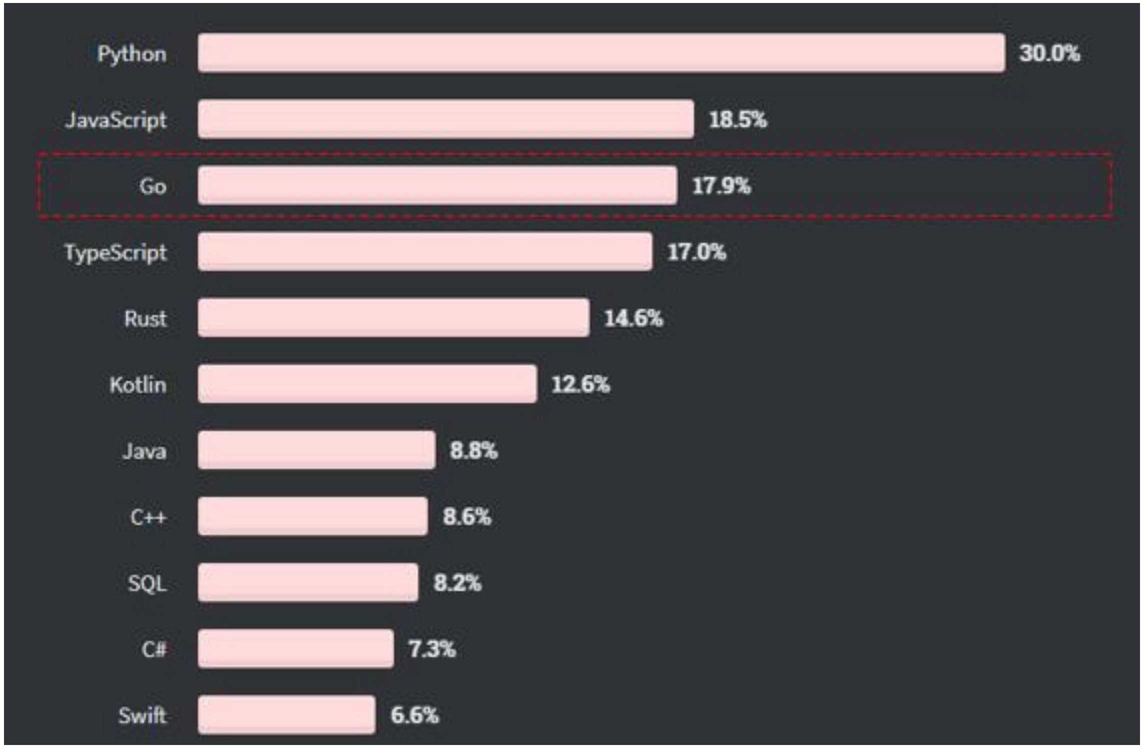
Implementing User-level Threads (5)

Preemptive scheduling

- Need to regain control of processor asynchronously
- Scheduler requests that a timer interrupt be delivered by the OS periodically
  - Usually delivered as a UNIX signal
  - Signals are just like software interrupts, but delivered to user-level by the OS instead of delivered to OS by hardware
- At each timer interrupt, scheduler gains control and context switches as appropriate

Case study : Go Lang

3rd Language to learn (2020)



Go is developed for Google infra structure

- Before Go, C++ is used
- C++ is fast, but every day code update, complex, long long build time

Case study : Go Lang

Pros

- Low memory consumption : 2KB stack
- Cheap creation and deletion of threads
  - goroutine : user-level thread (green thread)
- Cheap context switch cost

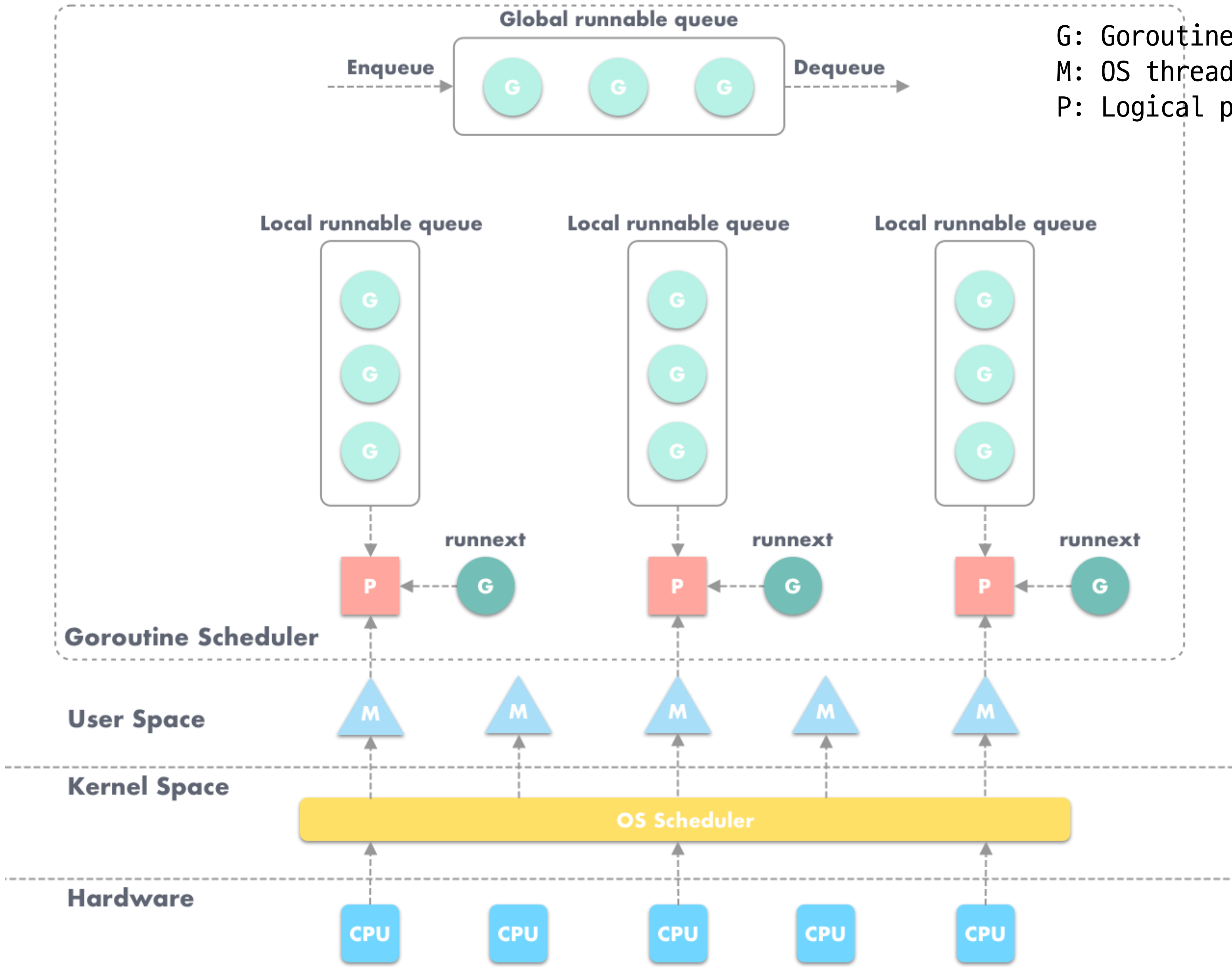
Go runtime scheduler

- Started with a Go program
- scheduling goroutine

Scheduler rule

- Kernel thread is cost, and so minimize them
- High concurrency with many many goroutine
- N goroutine on N cores

Case study : Go Lang



G: Goroutine
M: OS thread
P: Logical processor

Case study : Go Lang

Scheduler concepts

- Reuse threads
  - Unused go thread -> idle state
  - After sometime in idle, remove
- Limit threads accessing runqueue
  - Max number of threads is 10000 (default, modifiable)
  - Accessing local and global run queue needs lock
- Distributed runqueues
  - Minimize schedule lock