OS Basic: Process and Thread

Dongyoon Lee

Summary of last lectures

- Getting, building, and exploring the Linux kernel
- System call: interface between applications and kernel
- Kernel data structures
- Kernel modules
- Kernel debugging techniques

Today's agenda

- Process
- Thread

Process

- A process is a program in execution.
- A process is not the same thing as a program:
  - A program is a *passive* entity.
  - Processes are *active*.
  - Each process only runs one program at a time.
  - The same program can be run by more than one process at a time.

Multiprogramming

- A multiprogramming OS supports many concurrent processes.
    - Each process has a context, including an *address space*, and can receive *CPU cycles*.
    - The OS achieves an illusion of concurrency by switching the CPU rapidly between processes.

Process Context

- The context of a process is essentially a snapshot of the state of that process, including:
  - The CPU state, including contents of CPU registers.
  - The run state of the process (running, waiting, ready).
  - The address space of the process, which is its "view of memory." This includes:
    - *Main memory* allocated to the process.
    - *Page tables* that describe a mapping from virtual to physical addresses (more later).

(1) CPU state (x86_64)

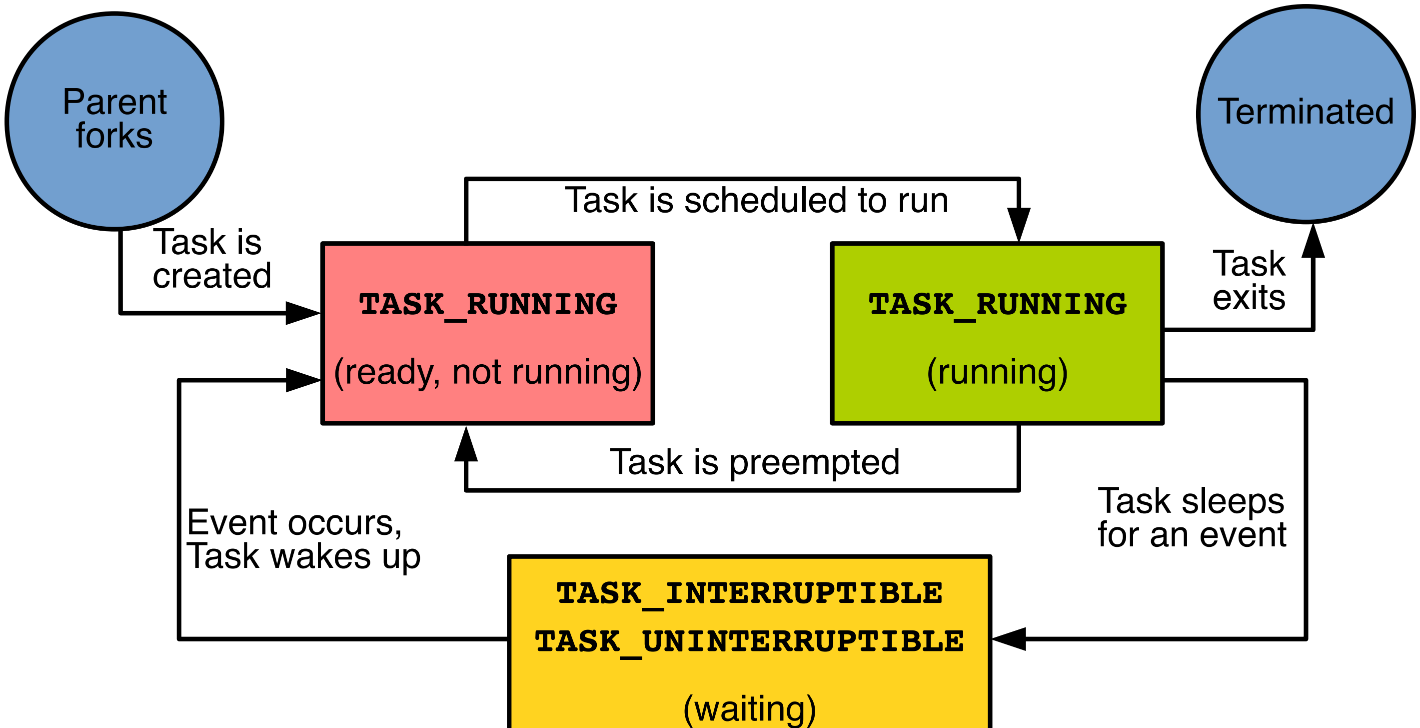
Register	Purpose	Saved across calls
%rax	temp register; return value	No
%rbx	callee-saved	Yes
%rcx	used to pass 4th argument to functions	No
%rdx	used to pass 3rd argument to functions	No
%rsi	used to pass 2nd argument to functions	No
%rdi	used to pass 1st argument to functions	No
%r8	used to pass 5th argument to functions	No
%r9	used to pass 6th argument to functions	No

(1) CPU state (x86_64)

Register	Purpose	Saved across calls
%r10-r11	temporary	No
%r12-r15	callee-saved registers	Yes
%rsp	stack pointer	Yes
%rbp	callee-saved; base pointer	Yes

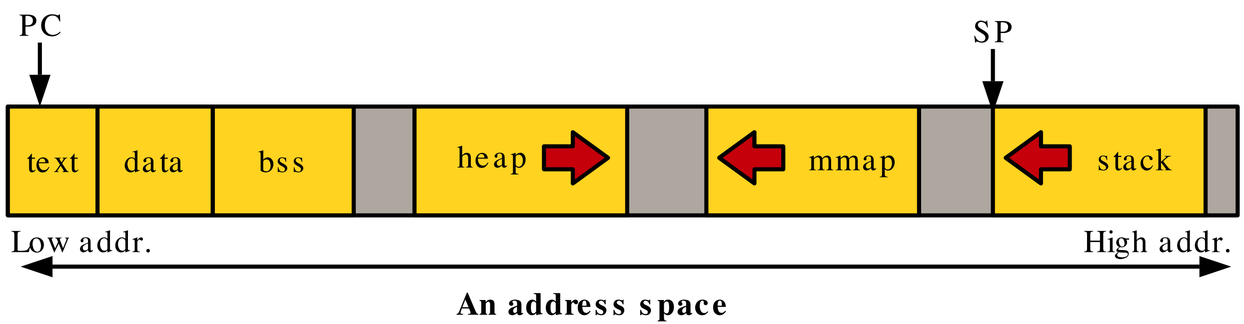
- Plus, floating-point and SIMD registers
- The program counter (PC) register points to the address of the next instruction to be executed from memory

(2) Run State of a Process



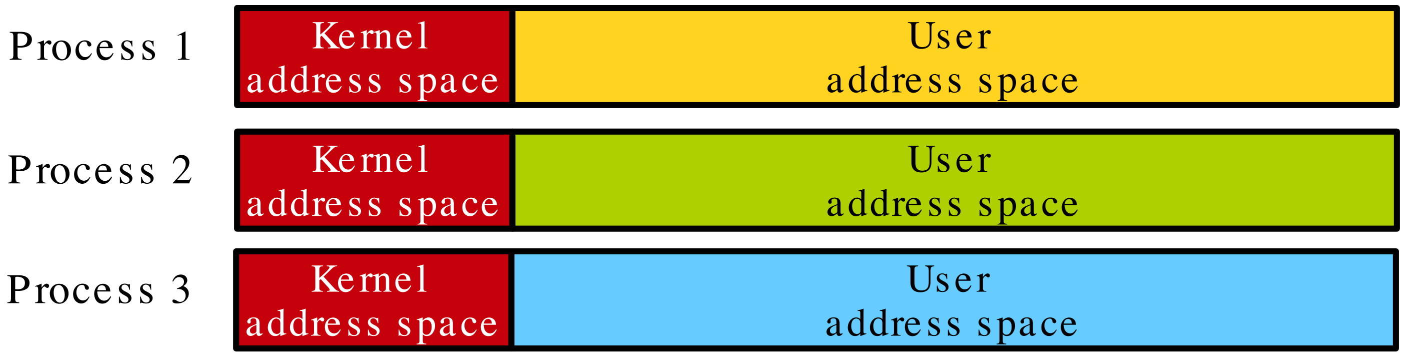
(3) Address Space

- An address space is the "view of memory" provided by the operating system for a process.



Multiple Address Spaces and OS Kernel

- A multiprogrammed OS maintains multiple address spaces simultaneously



Virtual Memory

- Virtual memory is a mechanism that permits a process to be run without having the entire contents of its address space loaded into main memory at one time.
    - The part of the address space that is loaded into main memory is called resident.
    - The remainder is called nonresident.
- Nonresident data is saved on a secondary storage area, called the backing store.

What is Virtual Memory Good For?

- A primary purpose of virtual memory is to increase the degree of multiprogramming to obtain more efficient utilization of system resources:
  - More runnable processes can be kept in main memory at one time.
  - More runnable processes means increased CPU utilization.

Other Uses of Virtual Memory

- Some other reasons for having VM are:
  - Running large applications whose address space exceeds the amount of main memory.
  - Decrease apparent startup time for large applications, by allowing applications to start with only a fraction of data resident.
  - Memory-mapped files provide an useful alternative to traditional I/O system calls.

OS and HW support for Virtual Memory

- Virtual address space is partitioned into fixed-size pages (e.g. 4KB).
- Physical memory is partitioned into page frames.
- OS manages page tables that define a mapping from (virtual) pages to (physical) page frames.
- HW (MMU) performs address translation on every memory reference.

The OS View of a Process

- Q: So, what is a process, to the OS?
- A: It's just a collection of bookkeeping data, including:
  - CPU register contents to be loaded when process runs.
  - Run state of the process (running, waiting, ready).
  - Memory allocated to the process.
  - Address space structures (e.g., code, data, stack sizes and locations).
  - Other resources in use by the process; such as data describing open files, network connections, etc.
- Linux: this data is managed/accessed by the "task" struct.

Context Switching

- The act of changing between running processes is called a *context switch*.
  - The previous process' context must be saved.
  - The next process' context must be restored.
- Once context save/restore has occurred, the OS can transfer control to the new process.

Processes vs. Threads

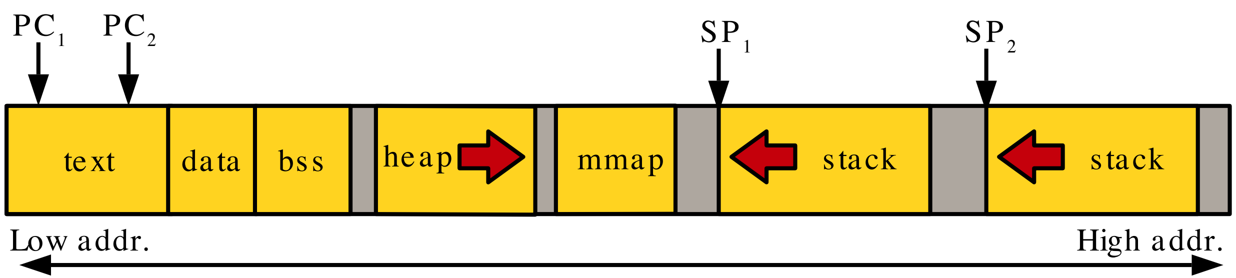
- The term process usually refers to a single “thread of control” that executes in its own private address space, separate from the address spaces of other processes.
    - Context switches between processes require changing the entire address space, and thus are fairly expensive.
    - Communication between processes requires additional special support from the operating system.

Processes vs. Threads (cont.)

- Modern operating systems support multithreaded processes:
  - Multiple cooperating "threads of control" execute within a single process.
  - Threads share most of the process' context, but have some private data (e.g. their stacks).
  - A full context switch is not required for switching between threads (so threads are *"lightweight"*).

Threads

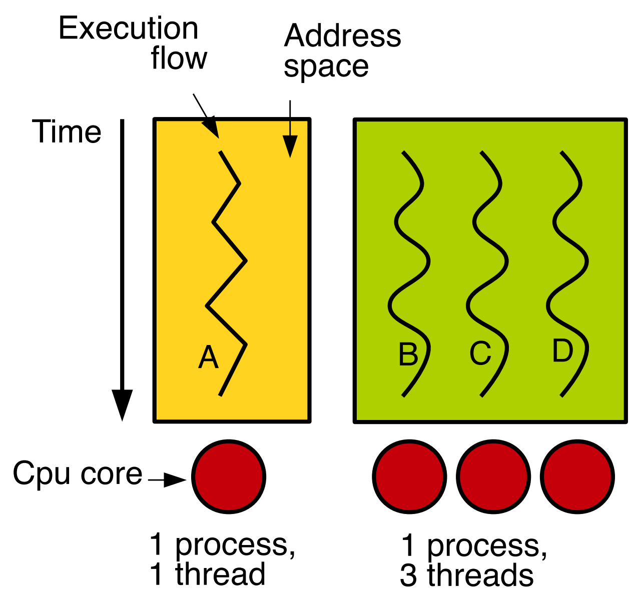
- A process (with a single address space) may have multiple threads.
- Threads share most of the process' context (e.g. heap, mmap).
- But threads have some private data (e.g. their stacks).



An address space (with two threads)

Threads (cont.)

- Threads are concurrent flows of execution belonging to the same program *sharing the same address space*



Next lecture

- Process management