CS 3214: Computer Systems
Lecture 2: Processes

Instructor: Huaicheng Li

Aug 29 2024

VIRGINIA TECH™

Administrivia

- Syllabus quiz released, deadline: 9/9 11:59pm
- Exercise 0 released, deadline: 9/6 11:59pm
- Lectures (Please bookmark them)
  - https://courses.cs.vt.edu/cs3214/fall2024/lecturesli/
  - Take a look at Dr. Back's lectures as well
- TA office hours posted
  - Google Calendar (Course website → MORE INFO → Staff)
  - Follow the queueing rules on Discord

Recap

- Systems Architecture: Applications / OS / Hardware
- Dual mode operation: Applications <-> OS (protection, isolation, performance)
  - System calls as OS APIs for applications to use
  - Dual mode operations: User/kernel mode, CPU privilege levels (Ring 3/0)
- Processes
  - Virtual resources including CPU share, address space, file descriptors, etc.
- **Time-sharing:** *N applications on 1 CPU ➔ 1/nth CPU for each application*

From Program to Process

- A process is a program during execution.
  - Program = static file (image)
  - Process = executing program = program + execution state.

- A process is the basic unit of execution in an operating system
  - Each process has a number, its process identifier (pid).

- Different processes may run different *instances* of the same program

- At a minimum, process execution requires following resources:
  - Memory to contain the program code and data
  - A set of CPU registers to support execution

From Program to Process

- We write a program in e.g. C
- A compiler turns that program into an instruction list.
- The CPU interprets the instruction list (which is more a graph of basic blocks).

```
void X (int b) {
  if (b == 1) {
...

int main() {
 int a = 2;
 X(a);
}
```

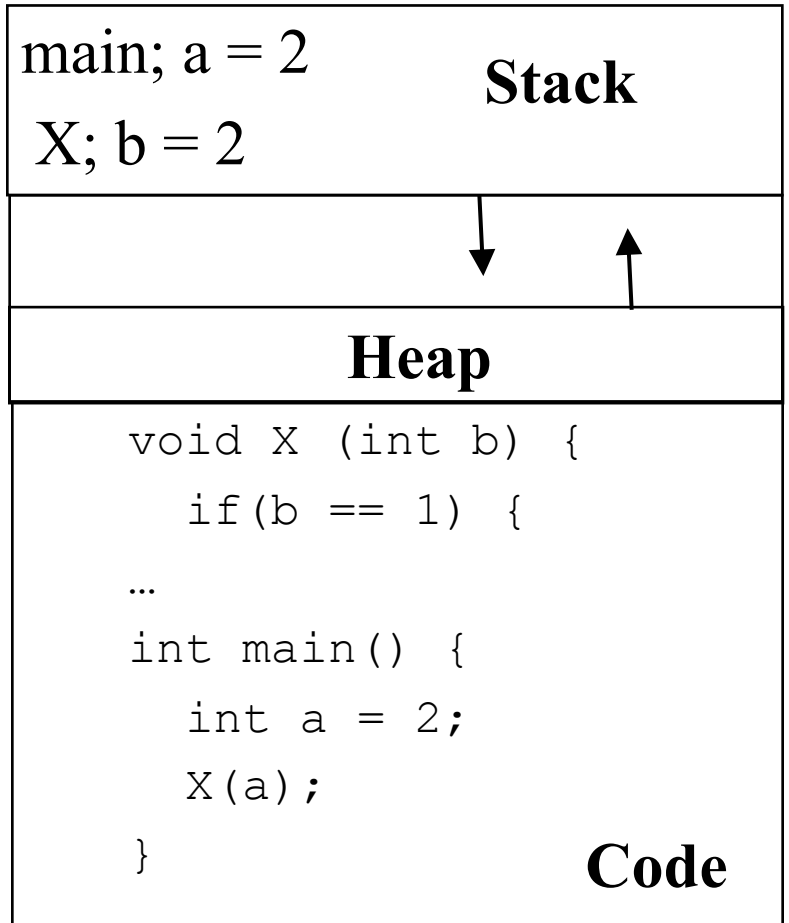
Process in Memory

What you wrote:

```
void X (int b) {
  if(b == 1) {
…
int main() {
  int a = 2;
  X(a);
}
```

Data

What is in memory:



Where do Processes Come From?

- When I type "./a.out", the binary runs, right?
  - Only true for static binaries (more later)
- In reality a loader sets up the program
  - Usually a user-level program

- To run a program, the loader:
  - reads and interprets the executable file
  - sets up the process's memory to contain the code & data from executable
  - pushes "argc" and "argv" on the stack
    - for the main() function
  - sets the CPU registers properly & calls "_start()"
- Program starts running at _start()
- When main() returns, OS calls exit() which destroys the process and returns all resources
- What bookkeeping does the OS need for processes?

Keeping Track of a Process

- A process has code
  - OS must track program counter (code location)
- A process has a stack
  - OS must track stack pointer
- OS stores state of processes' computation in a process control block (PCB)
  - E.g., each process has an identifier (process identifier, or PID)
- Data (program instructions, stack & heap) resides in memory, metadata is in PCB (which is a kernel data structure in memory)

Context Switch

- The OS periodically switches execution from one process to another
- Called a context switch, because the OS saves one execution context and loads another
- Causes?

Causes of Context Switches

- Waiting for I/O (disk, network, etc.)
  - Might as well use the CPU for something useful
- Timer interrupt (preemptive multitasking)
  - Even if a process is busy, we need to be fair to other programs
- Voluntary yielding (cooperative multitasking)
- Synchronization, IPC, etc.

Causes of Context Switching

- User → Kernel mode
  - Explicit:
    - Call system calls to enter kernel mode
    - Fault/exceptions (e.g, division by zero, attempt to execute privileged instructions
    - Synchronous
  - Implicit: (external events, e.g., hardware interrupts or preemption)
    - Preemption: higher priority kernel-level process needs to run
    - Interrupts: What is it?
    - What types of interrupts? Timer, keyboard, mouse, disk, network, etc.
    - Asynchronous
- Kernel → User mode
  - Via special privileged instruction (e.g., Intel iret)
  - A return from interrupt

I/O Example

- 1. NIC receives packet, writes packet into memory
- 2. NIC signals a hardware interrupt
- 3. CPU stops current operation, switches to the kernel mode, saves machine state on the kernel stack
- 4. CPU reads address from interrupt table indexed by interrupt number, jumps to the address of the interrupt handle (in the NIC driver)
- 5. NIC device driver processes the packet
- 6. Upon completion, CPU restores saved state from stack and returns to user mode
- Are there any other ways to perform I/O?

Timer

The timer is critical for an operating system

- It is the fallback mechanism by which the OS reclaims control over the machine
  - Timer is set to generate an interrupt after a period of time
  - Setting timer is a privileged instruction
  - When timer expires, generates a hardware interrupt
  - Handled by kernel, which controls what runs next
  - Basis for OS scheduler (process scheduling)
- Prevents infinite loops
  - OS can always regain control from erroneous or malicious
- programs that try to hog CPU
- Also used for time-based functions (e.g., sleep)

Interrupt

- Interrupts halt the execution of a process and transfer control (execution) to the operating system
  - Can the interrupt handler itself be interrupted?
  - Can we and shall we disable interrupts?
- Interrupts are used by devices to have the OS do stuff
  - What is an alternative approach to using interrupts?
  - What are the drawbacks of that approach?

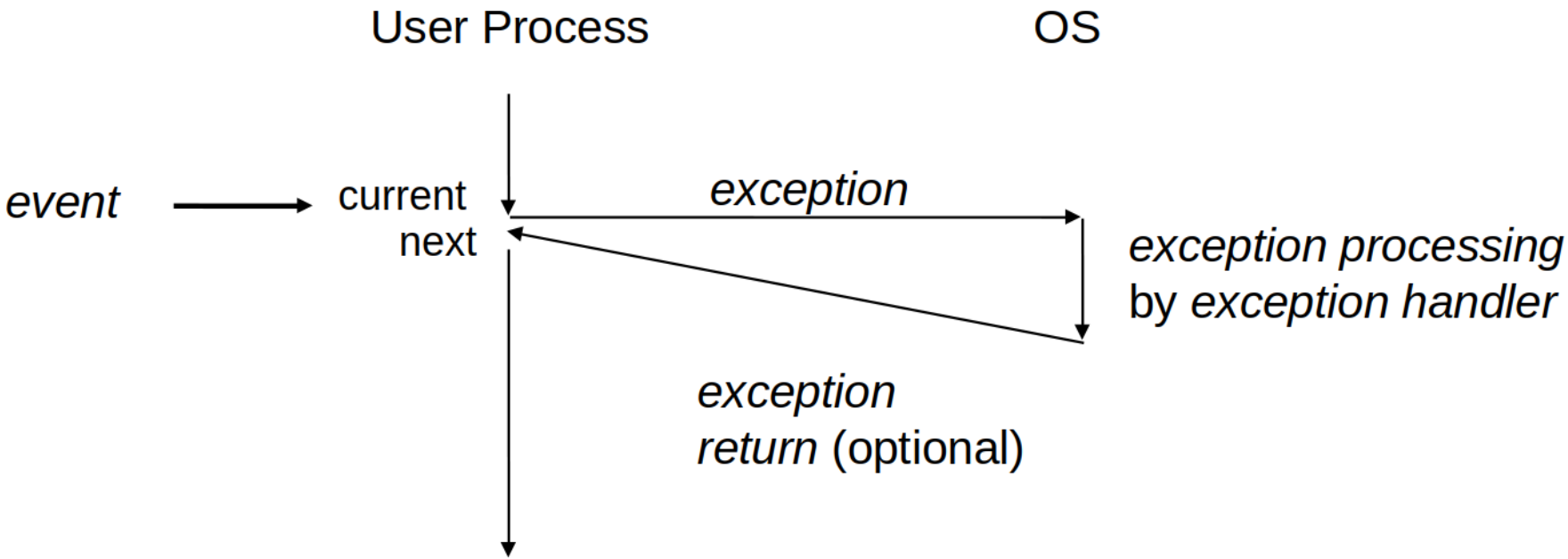
True / False

- The transition from user space to kernel space can happen without any hardware assistance/involvement
- malloc() is a system call
- Every keyboard stroke cause an interrupt

Context vs. Mode Switches

- Mode switch guarantees that kernel gains control when needed
  - To react to external events
  - To handle error situations
  - Entry into kernel is controlled
- Not all mode switches lead to context switches
  - context switches are between two processes
- Kernel decides if/when – subject to process state transitions and scheduling policies
- Mode switch does not change the identity of current process/thread

A Bottom-Up View of "Exceptions"



Process Struct in Linux

- Check Linux PCB code: ***struct task_struct { … }***
  - ***https://elixir.bootlin.com/linux/v5.19.3/source/include/linux/sched.h#L726***
  - struct mm_struct *mm;
  - struct files_struct *files;
  - struct sched_info sched_info;

Process Summary

- **Process definition:** An instance of a program that is being executed (aka, a running programming)
- Abstractions provided to a process
  - Virtual CPU: illusion of many CPUs
  - Address space – machine state
  - Files, etc.
- Time-sharing to enable multi-programming
- Context switches
  - **Context:** the state of the running program, which includes the current program text, the location within the program text (PC/IP), and all associated state: variables (global, heap, stack, CPU registers)
  - Switches – Dual Mode operations