COMP2310
Systems, Networks, & Concurrency

Convener: Shoaib Akram

shoaib.akram@anu.edu.au

Australian
National
University

Revision: System Calls & Processes

System Call – 1

- User code vs. kernel code
    - User code can only execute limited types of instruction
    - Kernel code can run any instruction type including privileged instructions

- Traps to the kernel set the mode bit on processor to kernel mode

- To request some service from the kernel, user space programs invoke the **system call**

- System calls provide the means for end user applications to access the resources in the kernel space, such as CPU, memory, storage

System Call – 2

- The system call interface serves three main purposes
    - Ensuring security
    - Abstraction
    - Portability
- The standard C library provides the system call interface as a convenience for user-space programs
- On x86-64, there are around **330** system calls

System Call – 3

- `syscall` instruction is a trap to an exceptional handler (or trap into the kernel)

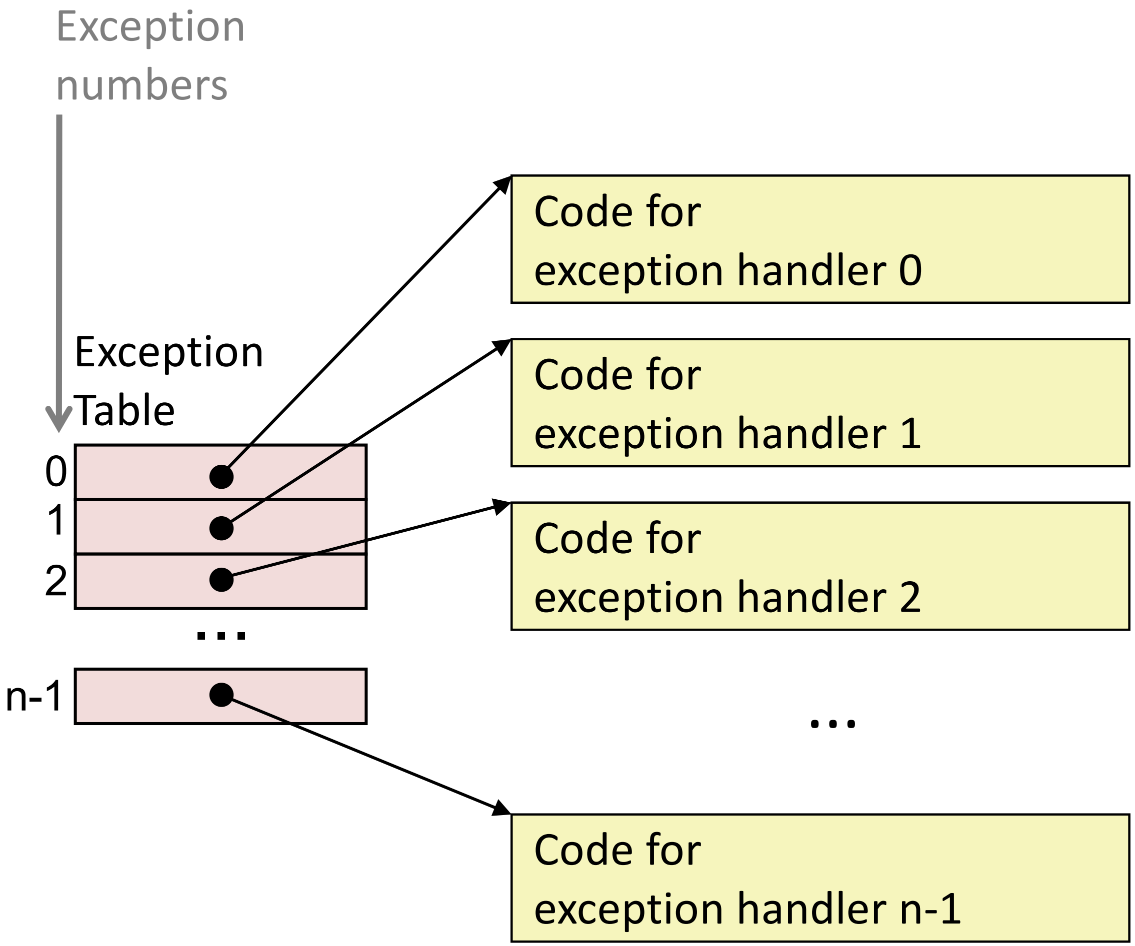
```
00000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00      mov  $0x2,%eax  # open is syscall #2
e5d7e:   0f 05               syscall         # Return value in %rax
e5d80:   48 3d 01 f0 ff ff   cmp  $0xfffffffffffff001,%rax
...
...
e5dfa:   c3                  retq
```

- Idea is to enable a **procedure-like** interface for making system calls

Recall: Sync. Exceptions

- **Caused by events that occur as a result of executing an instruction:**
  - ***Traps***
    - Intentional
    - Examples: ***system calls***, breakpoint traps, special instructions
    - Returns control to “next” instruction
  - ***Faults***
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting (“current”) instruction or aborts
  - ***Aborts***
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program

Recall: Exception Handling



- Each type of event has a unique exception number k
- k = index into exception table (a.k.a. interrupt vector)
- Handler k is called each time exception k occurs

Handlers run in kernel mode

System calls incur high overhead

In general, user to kernel mode switch is expensive

Process Context

- Each program runs in the context of some process

- Process context consists of:

  - State that the process needs to run correctly

    - GPR contents
    - PC
    - Environment variables
    - user stack and kernel stack
    - Kernel data structures maintained for the process
      - page table, file table, process table

Context Switch

- Kernel preempts a current process and *schedules* a different process

- Context switch includes:

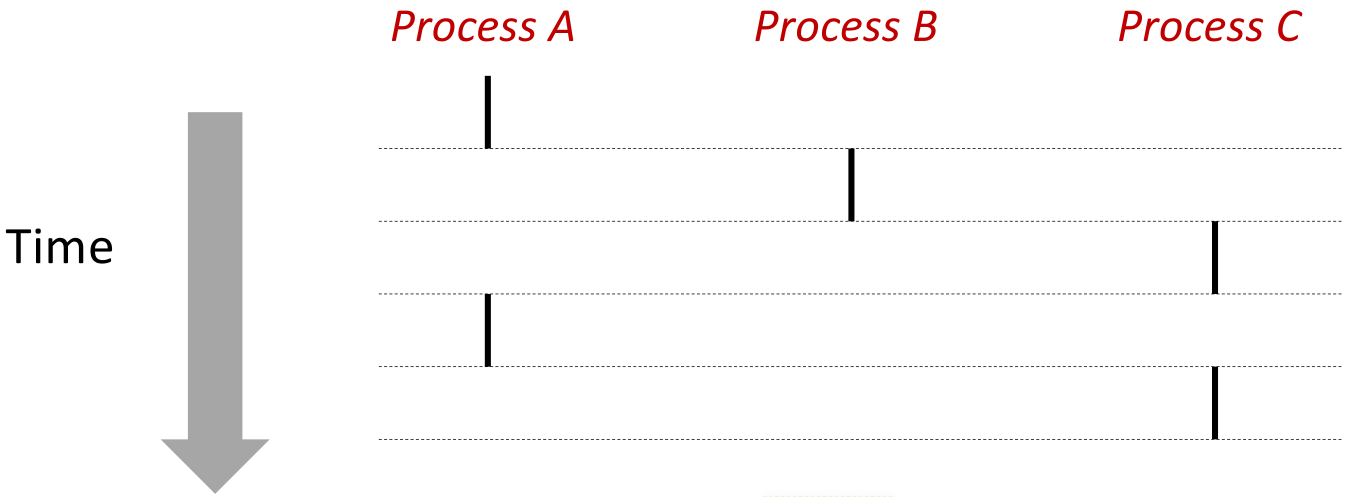
    - Saving the context of current process

    - Restoring the context of the newly scheduled process

- When does a context switch happens?
    - Timer interrupt
    - Blocking (long-running) system calls

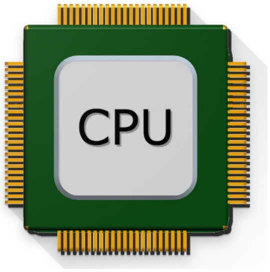
Concurrency

- Multiple logical control flows executing concurrently



Process A *Process B* *Process C*

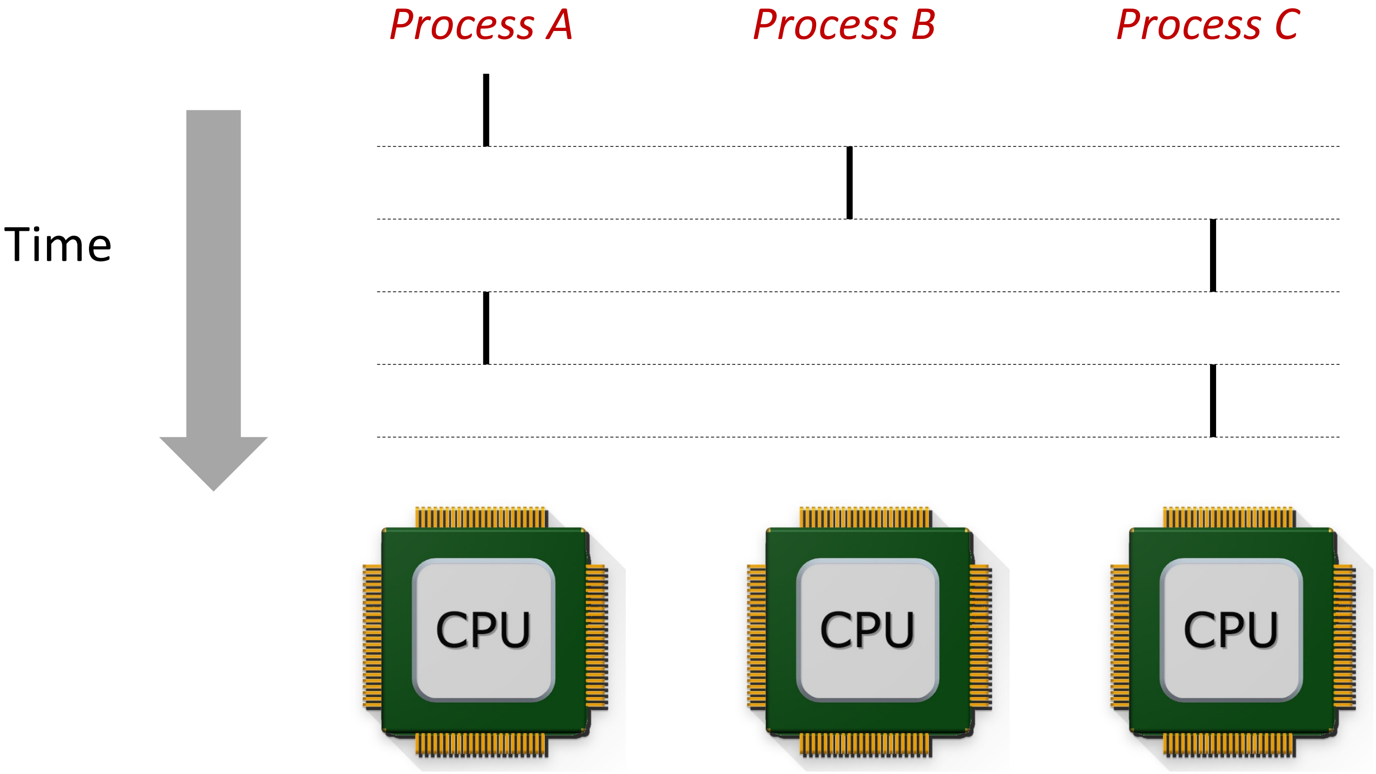
Time



3 processes, 1 CPU

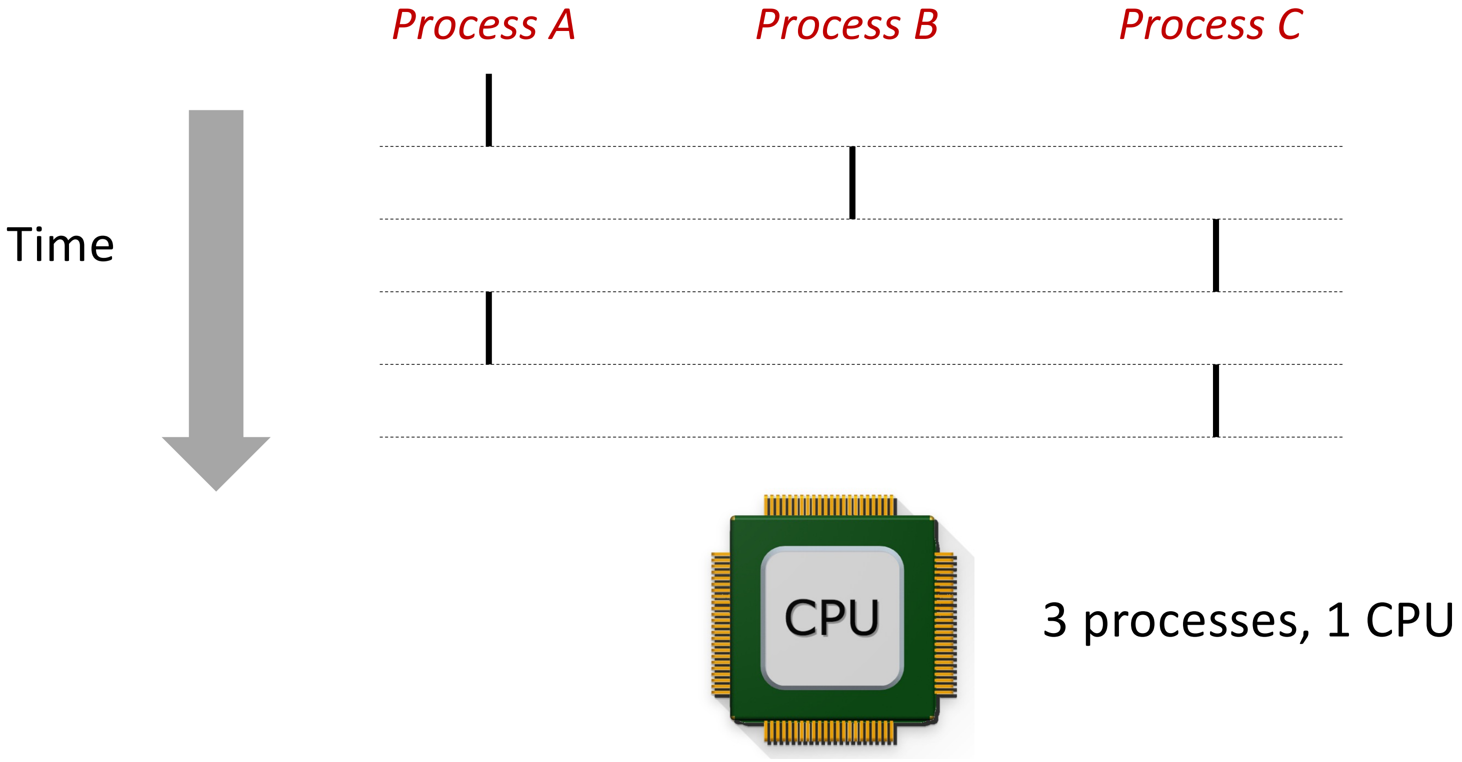
Parallelism

- Multiple logical control flows executing concurrently and simultaneously on independent CPUs



Time slicing or Multitasking

- Notion of processes taking turns to make progress on one physical CPU



Process A *Process B* *Process C*

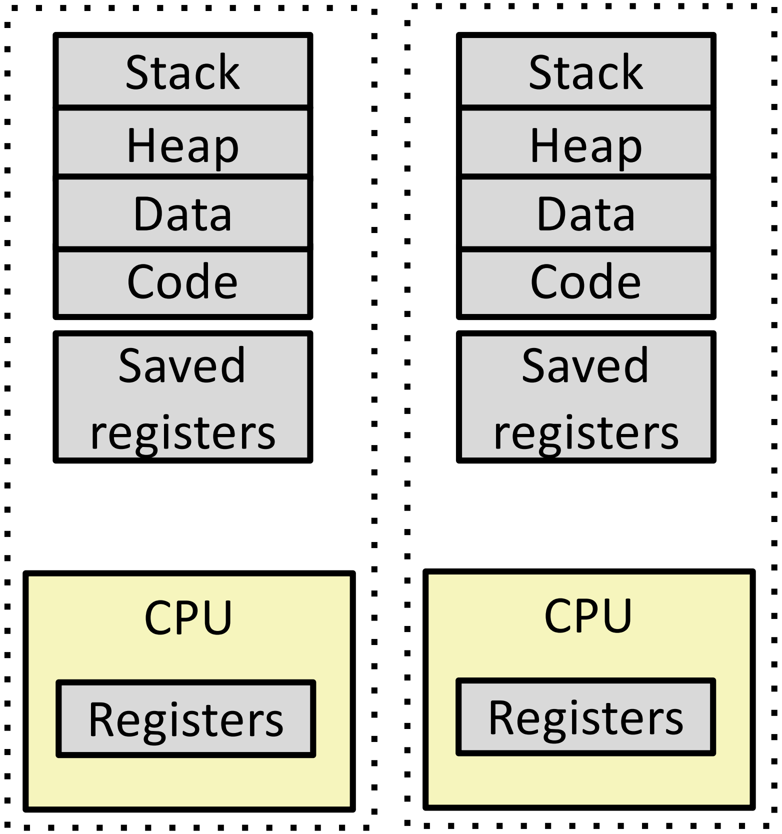
Time

3 processes, 1 CPU

What Illusions does process provides?

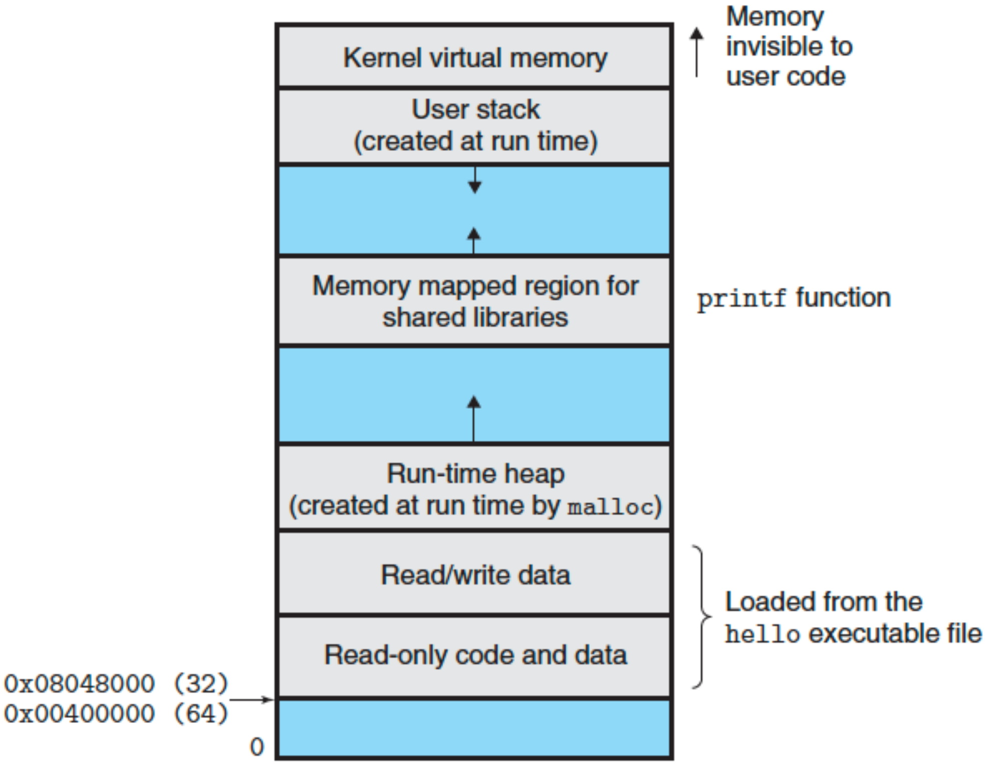
- That an application has:
  - An independent CPU for its use
  - A private 2^{64} byte or 16 Exabyte address space
    - Note that this address space is virtual
    - Suppose 100 users on a 64-bit system. How much physical memory if we guarantee 16 EB for each user?
    - Other motivation for virtual memory: protection

Parent and child processes as a result of fork()



- Child inherits a copy of parent’s context
- The layout of address space is identical
- Initially, the only difference is the value returned to child and parent
- called once, returns twice, once in parent, and once in child
- Child could be scheduled right after creation (non-determinism)

Parent and child have private address spaces



- Processes are expensive to create and maintain
- Later in the course, we will study a light-weight abstraction for concurrency called threads
- Linux prevents memory duplication b/w parent and child using a phenomenon called “copy-on-write”

Feasible orderings

- The instructions executed by parent and child are interleaved by the kernel in an order that is non-deterministic
- Not all orderings are feasible
- Infeasible orderings violate the “happen-before” notion in sequential control flow
- If instruction B appear later in program order, then it cannot execute before instruction A, that appears earlier in program order