Inf2C - Computer Systems
Lecture 15
Exceptions and Processor Management

Boris Grot

School of Informatics
University of Edinburgh

Previous lecture: Virtual memory

- Solves two problems:
  - – Capacity (physical memory is limited)
  - – Safety (physical memory must be shared by multiple programs and the OS)
- Virtual vs physical address space
  - – Each program "sees" a full 32-bit address space
  - – Actual physical memory managed by the OS
- Address translation
  - – Page table – all translations, but slow (in memory)
  - – TLB – recent entries only, but fast (cache)

Exceptions – definition

- Exceptional events that interrupt normal program flow and require attention of the CPU
- External ("interrupts")
  – Not caused by program execution
  – E.g. I/O interrupt (e.g., network packet arrived)
- Internal ("traps")
  – Caused by program execution
  – E.g. illegal instruction, arithmetic overflow, TLB miss

Intentional exceptions

- Use exception mechanism to request some OS functions

  e.g., I/O (e.g., print to screen), memory allocation

- User program uses `syscall` instruction
  - Cause register ($v0) is set with a special value to identify the syscall exception
  - OS exception handler invoked when instruction executes

- Parameters are passed to the OS through agreed upon registers (usually $a0, $a1, ..)

Syscall example

The following will print the integer in register $t0 to the screen.

```
li  $v0, 1      # service 1 is "print integer"
add $a0, $t0, $zero  # load integer into $a0
syscall
```

Exception mechanism

- Step 1: Save the address of current instruction
  - into a special register, the exception program counter (EPC)
  - Note: must return to the interrupted instruction (<u>not</u> PC+4)
- Step 2: Transfer control to the OS at a known address (i.e., exception handler PC)
- Step 3: Handle the exception
  - Deal with the cause of the exception
  - All registers must be preserved, similar to a procedure call
- Step 4: Return to user program execution
  - Handler restores user program's registers and jumps back using EPC
  - Relies on special instruction **`eret`**

Finding the exception handler

- Approach 1:
  - Jump to a predefined address (0x800000180)
  - Use the Cause register to then branch to the right handler (e.g., print int, read string, exit program)
  - Works well for syscall – cause register explicitly set
- Approach 2
  - Directly jump to a specific handler depending on the exception (**vectored interrupt**)
  - Eg:

Undefined opcode:	0xC000 0000
Overflow:	0xC000 0020
…:	0xC000 0040

Handling the exception

- Determine action required
  - By inspecting the Cause register or by virtue of being at the right handler (e.g., undefined opcode)
- If restartable:
  - Take corrective action, then use EPC to return to program
- Otherwise:
  - Terminate program and report error using EPC, cause, …
- For a critical time while the interrupt is being handled, other interrupts should not happen
  - Otherwise the EPC, Cause will be overwritten
  - This is forced by masking interrupts → by setting the exception level (EXL) bit in the status register

Protecting system resources

- The OS must guarantee safe and orderly access to critical system resources
  - Hardware (processor, networking, I/O)
  - Program memory (including page tables)
- The OS is the ultimate arbiter of what's allowed
  - TLB miss → OK (but must access page table to service)
  - Write access to a read-only page → not OK (but must access page table to check)
  - Illegal opcode → not OK (kill the program)
- Exceptions are used to hand control over to the OS
  - Need a separate mechanism to limit capabilities of user programs

Kernel vs. User Mode Protection

- Exceptions (including system calls) are handled by the OS
  - CPU has two modes of operation: **user** and **kernel** (OS)
  - Current mode identified by a bit in a special status register
  - Exception mechanism is used to force the mode to change from user to kernel for execution of OS functions
- "Privileged" instructions only executed in kernel mode
  - E.g. accessing I/O devices, handling page tables accesses
- Kernel mode can only be entered through an exception
  - User programs cannot jump to OS instruction space
- `eret` instruction sets mode back to previous mode

Advantages of Dual Mode architecture

- Guarantees that control is transferred to OS when user programs attempt to perform potentially dangerous tasks
- Allows OS to ensure that programs do not interfere with each other
  - e.g., that memory is divided appropriately
- Allows OS to ensure that programs do not have access to resources for which they do not have permission
  - e.g., files
- Ensures that user programs do not have indefinite control of the processor
  - could happen in old-school Windows or ancient DOS

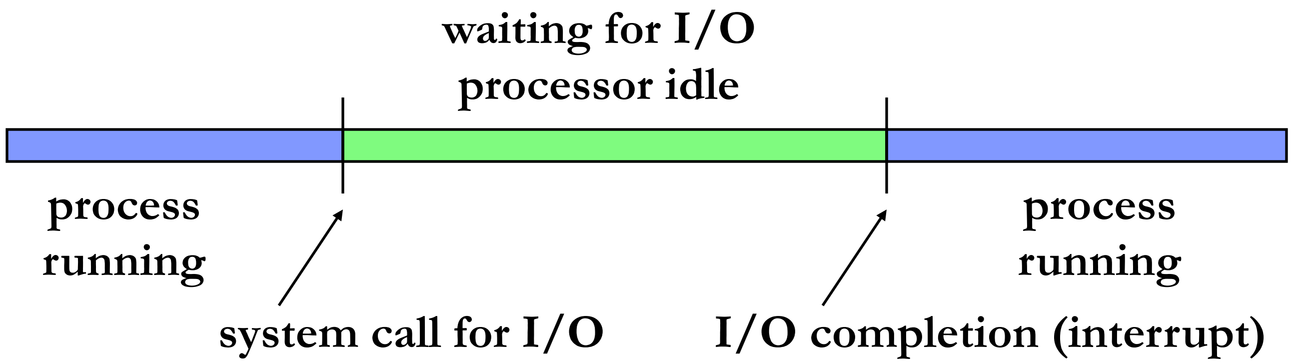
Managing the Processor

- Problem:
  - I/O takes too long → processor idle
  - User programs can crash or monopolize the CPU (either unintentionally or maliciously)
- Solution:
  - Multiplex or time-share the CPU and other resources among several user processes
  - Switch from one process to another when it performs I/O, or when it's time allocation (time slice) expires

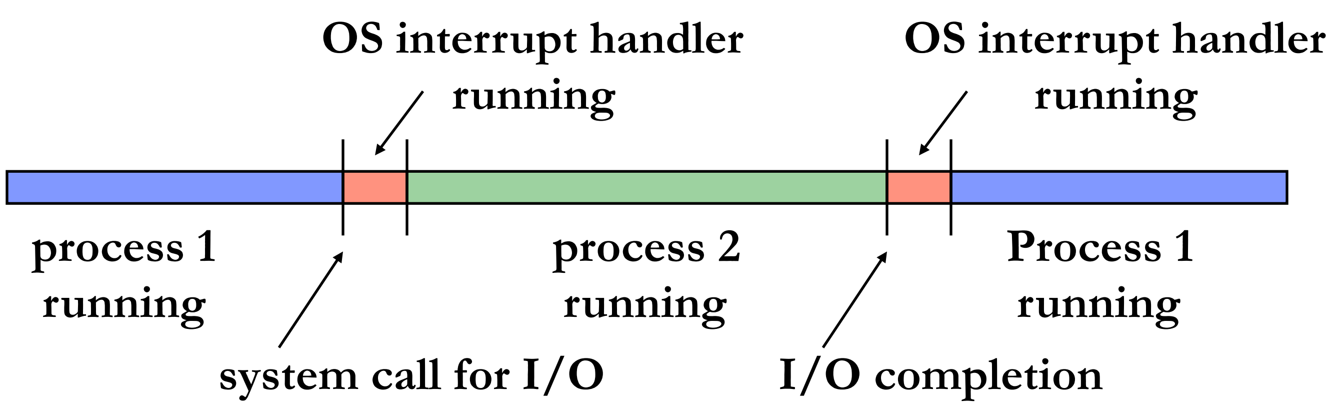
Process: "a program in execution" [Silberschatz, Galvin, Gagne]

Multi-tasking

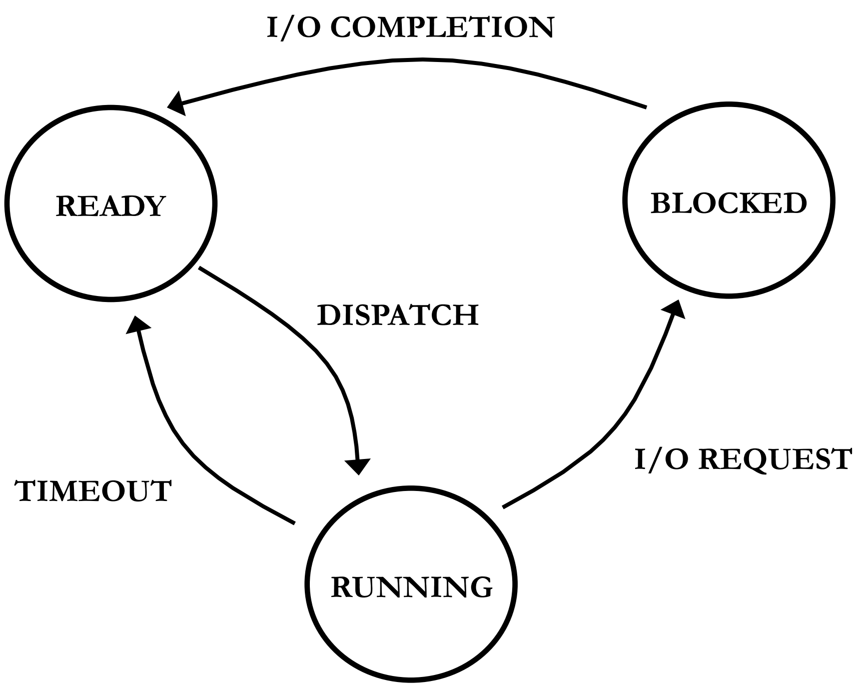
- Single-task system:



- Multi-tasking system:



Process States



States:

RUNNING: process is currently running in the CPU
READY: process is not running, but could run if brought into CPU
BLOCKED: process is not able to run because it is waiting for I/O to finish

Transitions:

I/O REQUEST: process initiates I/O
I/O COMPLETION: I/O finishes
DISPATCH: OS moves process into CPU and it starts executing
TIMEOUT: process's timeslice is over

Process States

- Step 1: process calls (or traps into) the OS, or interrupt occurs (e.g. because of timer)
- Step 2: OS's dispatcher performs context-switch:
  - Process's context is saved (registers, PC, etc) in process control block (PCB)
  - Dispatcher chooses new process to run
  - Processes' states are updated

PCB: OS data structure containing each process's information:
  - Process id (PID)
  - Process state (blocked, running, etc)
  - Process priority
  - Process permissions
  - Etc

Suspending and Resuming Processes

- Problem:
  - Might not have enough physical memory for all processses
  - Some processes have higher priority and must get more processor time (e.g., media playback)
- Solution:
  - Processes can be "swapped out" from memory to disk
  - Such processes are moved into an "inactive" state
    - 2 new process states
  - PCB of inactive processes are still kept in OS memory
  - Inactive processes are resumed by "swapping in" the data from disk back to memory

Suspending and Resuming Processes