Lecture 11: Exceptions & processor management

- Exceptions
- Operating system's main task:
  Processor management

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
- Internal ("traps") → caused by program execution
  - E.g. illegal instruction
    arithmetic overflow

Exception mechanism

- Step 1: Save the address of current instruction
  - into a special register, the exception program counter (EPC)
- Step 2: Transfer control to the OS at a known address
- Step 3: Handle the interrupt
  - 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: special instruction **`eret`**

Exception handling

- What caused the exception?
  - "Cause" register records the reason, or
  - Jump to a specific address depending on the exception (vectored interrupt)
- 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

Software Exceptions

- Use exception mechanism to request some OS functions

  e.g., I/O, dynamic memory allocation

- User program uses `syscall` instruction
  - Cause register is set with a special value to identify the syscall exception
  - OS exception handler is invoked as usual
- Parameters are passed to the OS through agreed upon registers

Kernel vs. User Mode Protection

- Why make system calls through the exception mechanism rather than through normal procedure calls?
  - CPU has dual mode of operation identified by a bit in status reg.
  - Exception mechanism is used to force the protection mode to change from user to kernel (OS) for execution of OS functions
- "Privileged" instructions only executed in kernel mode
  - E.g. accessing I/O devices, handling virtual memory
- 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 invariably transferred to OS when user programs attempt to perform potentially dangerous tasks
- Ensures that user programs do not have indefinite control of the processor (e.g., Windows 3.1 and 95 versus Windows NT & later)
- 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)

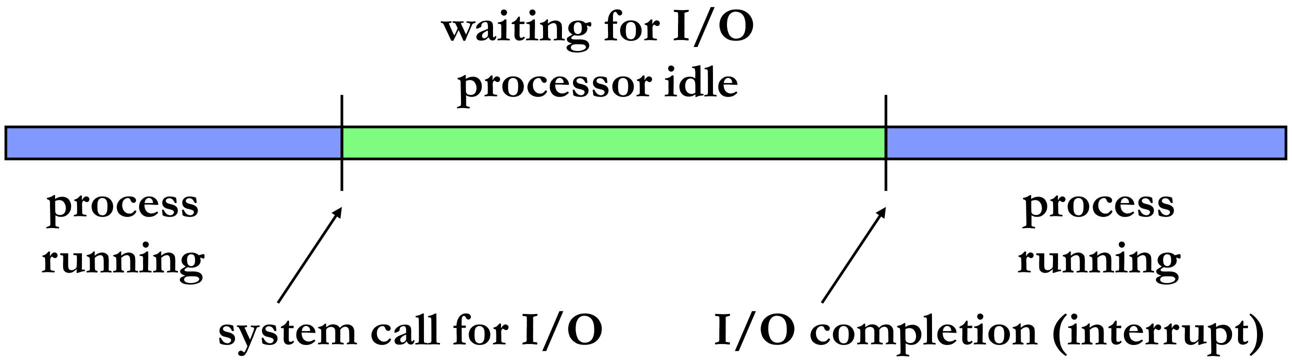
Managing the Processor

- Problem:
  - I/O takes too long → processor idle
  - User programs can crash or monopolize the CPU, 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 (timeslice) expires

Process: "a program in execution" (Silberschatz, Galvin, Gagne)

Multi-tasking

- Single-task system:


waiting for I/O
processor idle

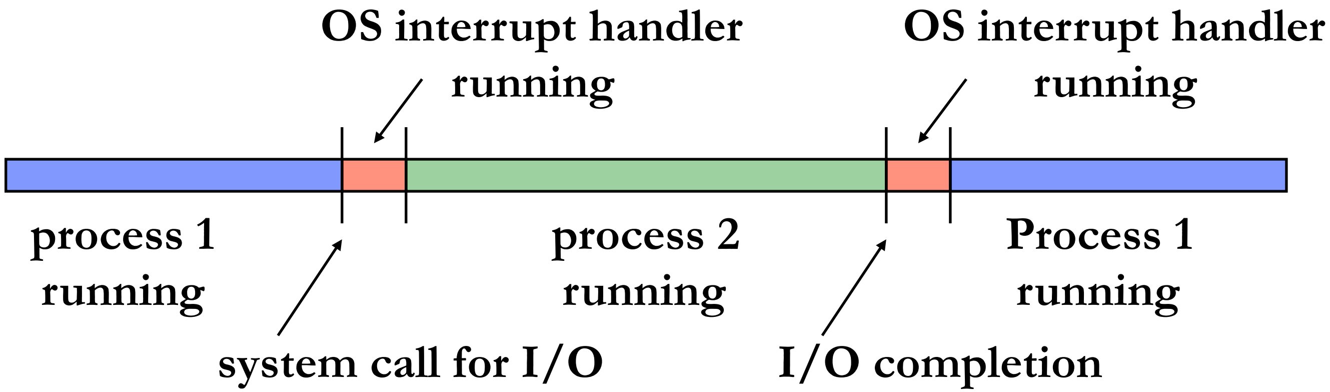
process running

process running

system call for I/O

I/O completion (interrupt)

- Multi-tasking system:


OS interrupt handler running

OS interrupt handler running

process 1 running

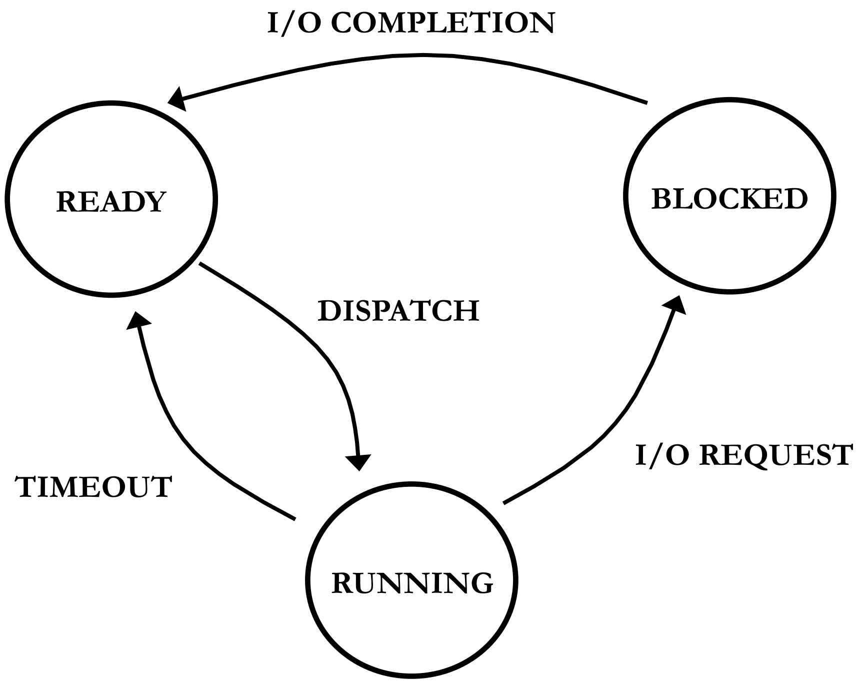
process 2 running

Process 1 running

system call for I/O

I/O completion

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 (only in pre-emptive multi-tasking systems)

Process States

- Step 1: process calls 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

Creating and Destroying Processes

- New processes can be explicitly created by the user, or implicitly by another process
- Original process → parent

  New process → child
- Processes are managed by the OS “kernel”:
  - Process dispatcher chooses which process to run next from the pool of active processes

OS Kernel

- Kernel: (small, efficient)
  - Interrupt handling
  - Process creation and destruction
  - Process state switching
  - Memory management
  - Inter-process communication and synchronization
  - I/O support

Suspending and Resuming Processes

- Problem:
  - Memory may not be enough for all active processes (more on this in other lectures)
  - Some processes have higher priority and must run at the expense of others
- Solution:
  - Processes can be "swapped out" from memory to disk (i.e., data is moved 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