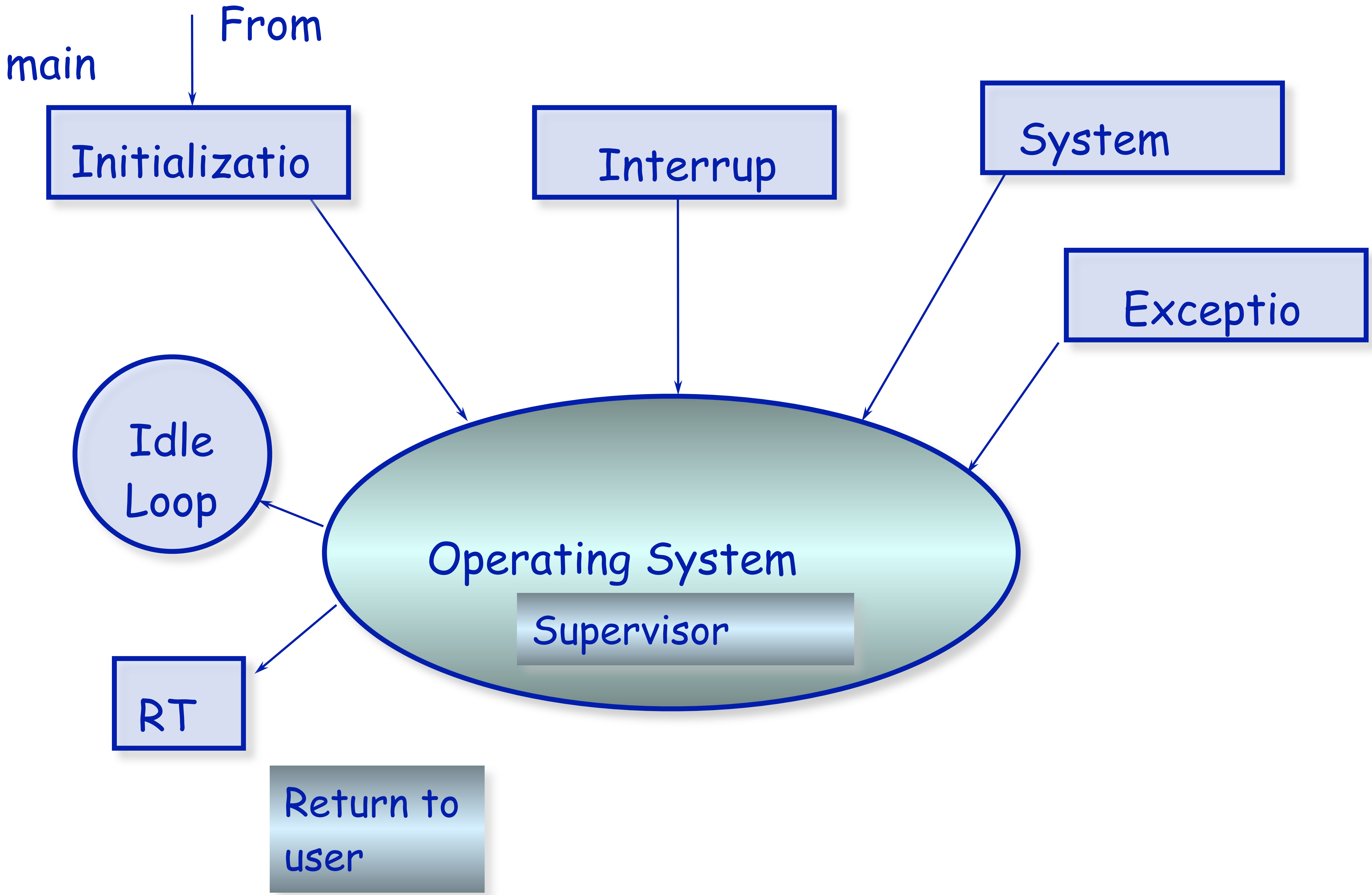
An Operating System in Action

- CPU loads boot program from ROM (e.g. BIOS in PC's)
- Boot program:
  - Examines/checks machine configuration (number of CPU's, how much memory, number & type of hardware devices, etc.)
  - Builds a configuration structure describing the hardware
  - Loads the operating system, and gives it the configuration structure
- Operating system initialization:
  - Initialize kernel data structures
  - Initialize the state of all hardware devices
  - Creates a number of processes to start operation (e.g. getty in UNIX, the Windowing system in NT, e.g.)

O.S. in Action (Cont'd)

- After basic processes have started, the OS runs user programs, if available, otherwise enters the idle loop
- In the idle loop:
  - OS executes an infinite loop (UNIX)
  - OS performs some system management & profiling
  - OS halts the processor and enter in low-power mode (notebooks)
- OS wakes up on:
  - Interrupts from hardware devices
  - Exceptions from user programs
  - System calls from user programs
- Two modes of execution
  - User mode: Restricted execution mode (applications)
  - Supervisor mode: Unrestricted access to everything (OS)

Control Flow in an OS



Input / Output

- Reading or writing data from a perepheral device is not simple
  - Each device has its own controller (hardware)
  - Each device is managed by a device driver (software to use the controller)
    - Device drivers are specific to hardware and to the operating system using the device
  - Input and output is SLOW in comparison to CPU operations.

Busy Waiting

- Reading or writing data to a device is SLOW in comparison to the time it takes to complete one CPU operation

- The CPU must send one or more instructions to the controller to make the I/O device begin to read or write.

- The CPU can then wait until the I/O operation is finishes.
  - While it waits the CPU will be in a loop
  - Each time through the loop the CPU will check a register in the controller to see if the I/O operation is complete

Alternatives to Busy waiting

- Busy waiting does not use CPU resources efficiently
- Want to use the CPU to execute other instructions while the I/O operation is being completed
- Interrupts : a mechanism to tell the CPU when the controller completes the I/O

Interrupts

- Mechanism by which other modules (memory, I/0, timers …) may interrupt the normal sequence of instructions being executed by the processor

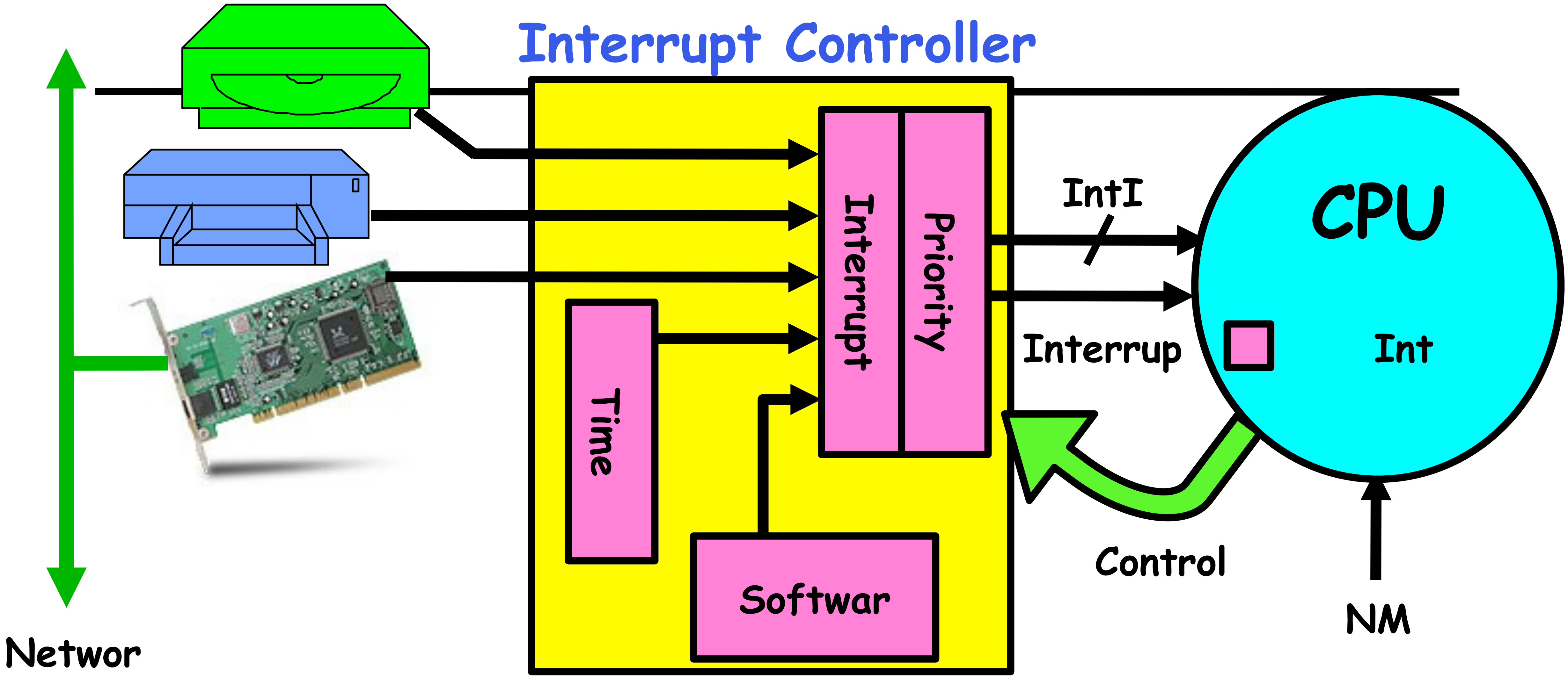
- Interrupts are a critical component of the operation of spooling and multiprogramming (more later).

- Interrupts allow the transfer of control between different programs (remember the OS is also a program)

- Interrupts are generated by hardware (asynchronous)
  - Exceptions are generated by particular instructions in software (synchronous), e.g., divide by 0, overflow, illegal instruction or address…

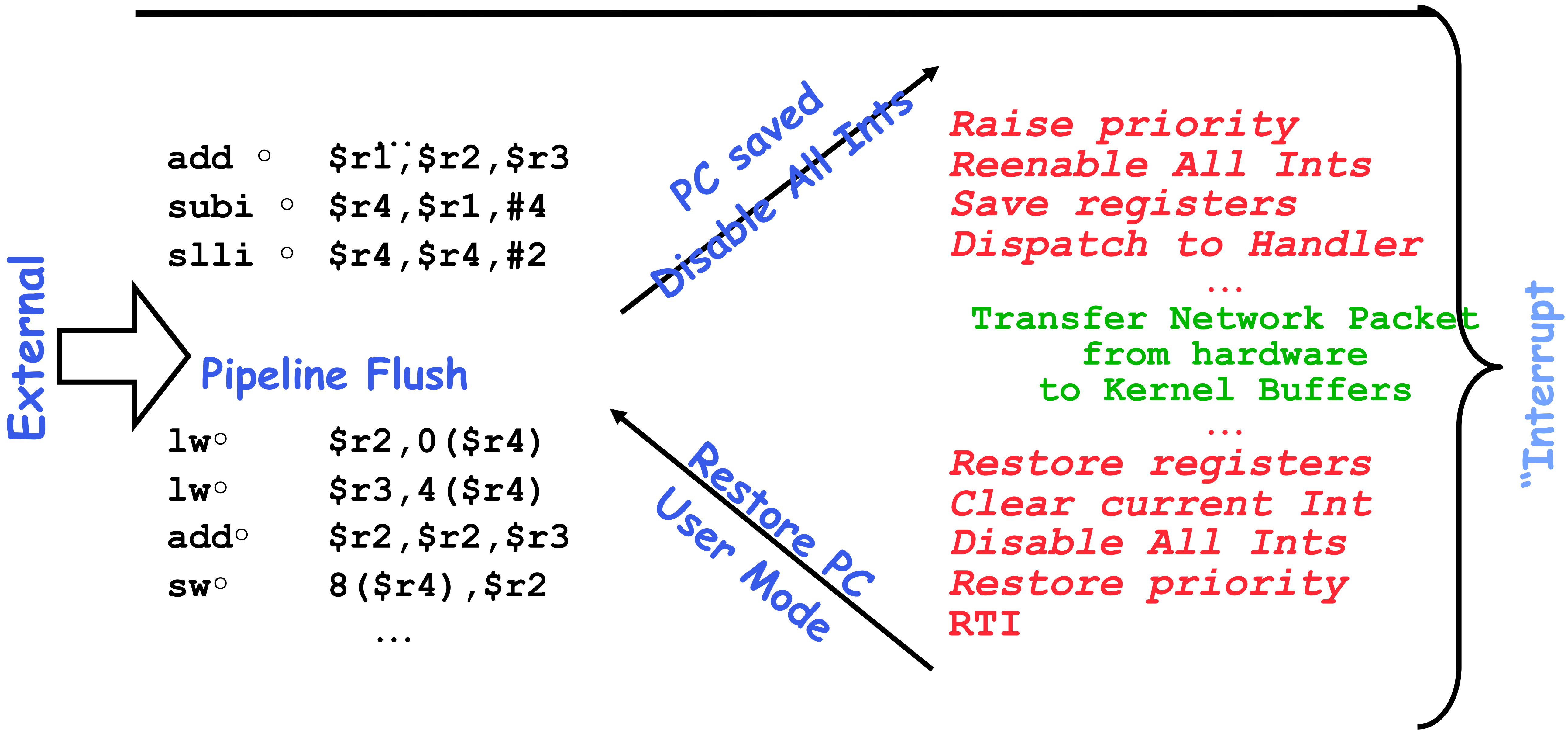
Some types of interrupts

- I/0
  - Signaling normal completion of an operation (read or write)
  - Signaling error during operation
- Timer expiry
  - Begin regularly scheduled task
  - End task that has exceeded allocated time
- Hardware failure



- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
  - Interrupt identity specified with ID line
- CPU can disable all interrupts with internal flag
- Non-maskable interrupt line (NMI) can't be disabled

Example: Network Interrupt



- Disable/Enable All Ints ⇒ Internal CPU disable bit
  - RTI reenables interrupts, returns to user mode
- Raise/lower priority: change interrupt mask
- Software interrupts can be provided entirely in software at priority switching boundaries

Interrupt processing (1)

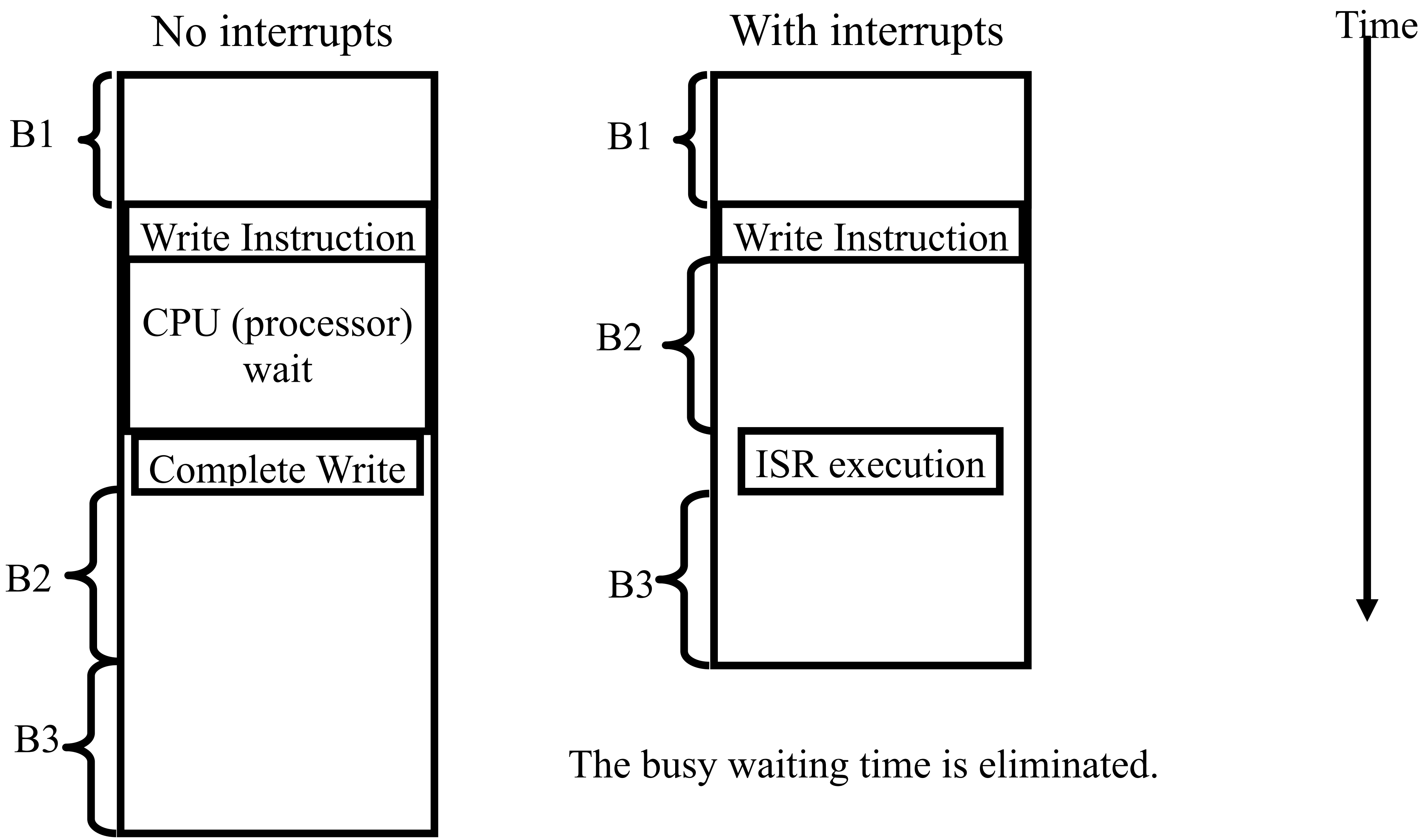
- A device issues an interrupt request
- The CPU finishes execution of the present instruction
- The CPU checks if there is a pending interrupt,
  - sends ACK to the device
  - CPU saves registers and state to the stack (including current instruction)
- The CPU loads the address of the appropriate ISR into the address register

Interrupt processing (2)

- The CPU executes the ISR
- When the ISR finishes, the saved register and state information is restored to the CPU registers
- The program counter is reset to point to the next instruction
- The original program continues execution

REMEMBER: the time when an interrupt occurs is not known in advance!! Interrupts are asynchronous

Increase in efficiency



The busy waiting time is eliminated.

On Exceptions

- Hardware calls the operating system at a pre-specified location
- Operating system identifies the cause of the exception (e.g. divide by 0)
- If user program has exception handling specified, then OS adjust the user program state so that it calls its handler
- Execute an RTI instruction to return to the user program
- If user program did not have a specified handler, then OS kills it and runs some other user program, as available

Key Fact: Effects of exceptions are visible to user programs and cause abnormal execution flow

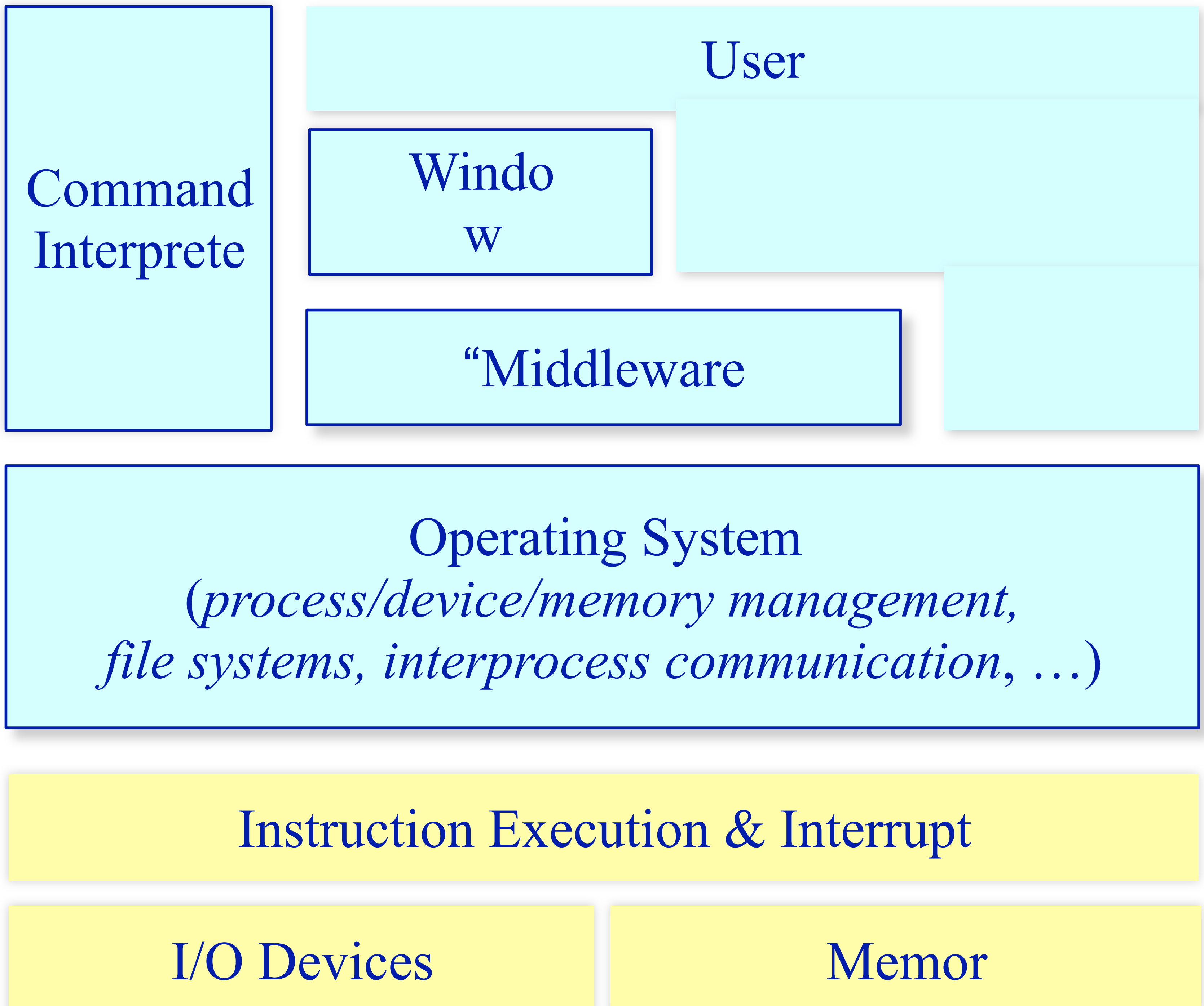
On System Calls

- User program executes a trap instruction (system call)
- Hardware calls the operating system at a pre-specified location
- Operating system identifies the required service and parameters (e.g. open(filename, O_RDONLY))
- Operating system executes the required service
- Operating system sets a register to contain the result of call
- Execute an RTI instruction to return to the user program
- User program receives the result and continues

Key Fact: To the user program, it appears as a function call executed under program control

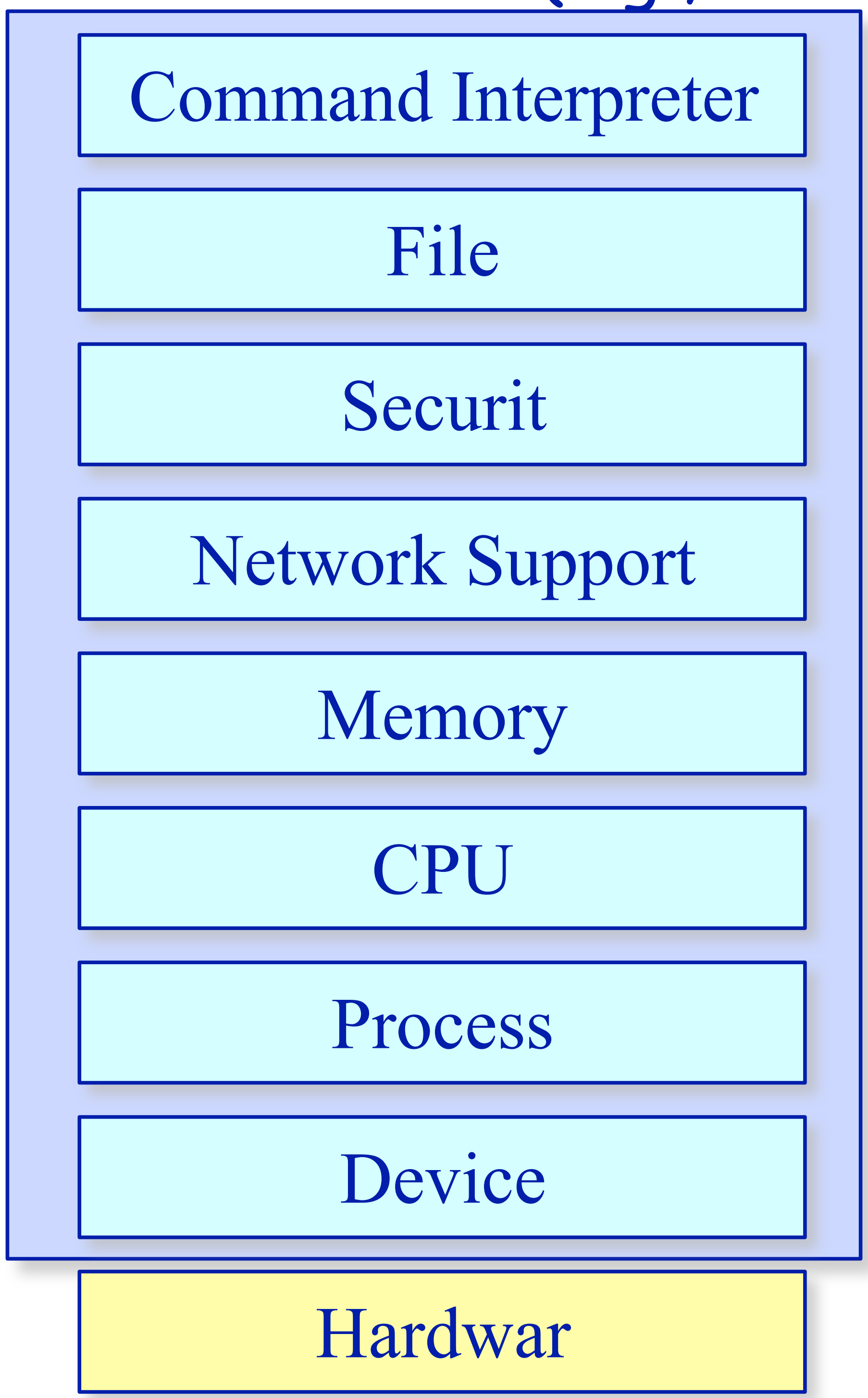
Operating System Today

High-level software architecture

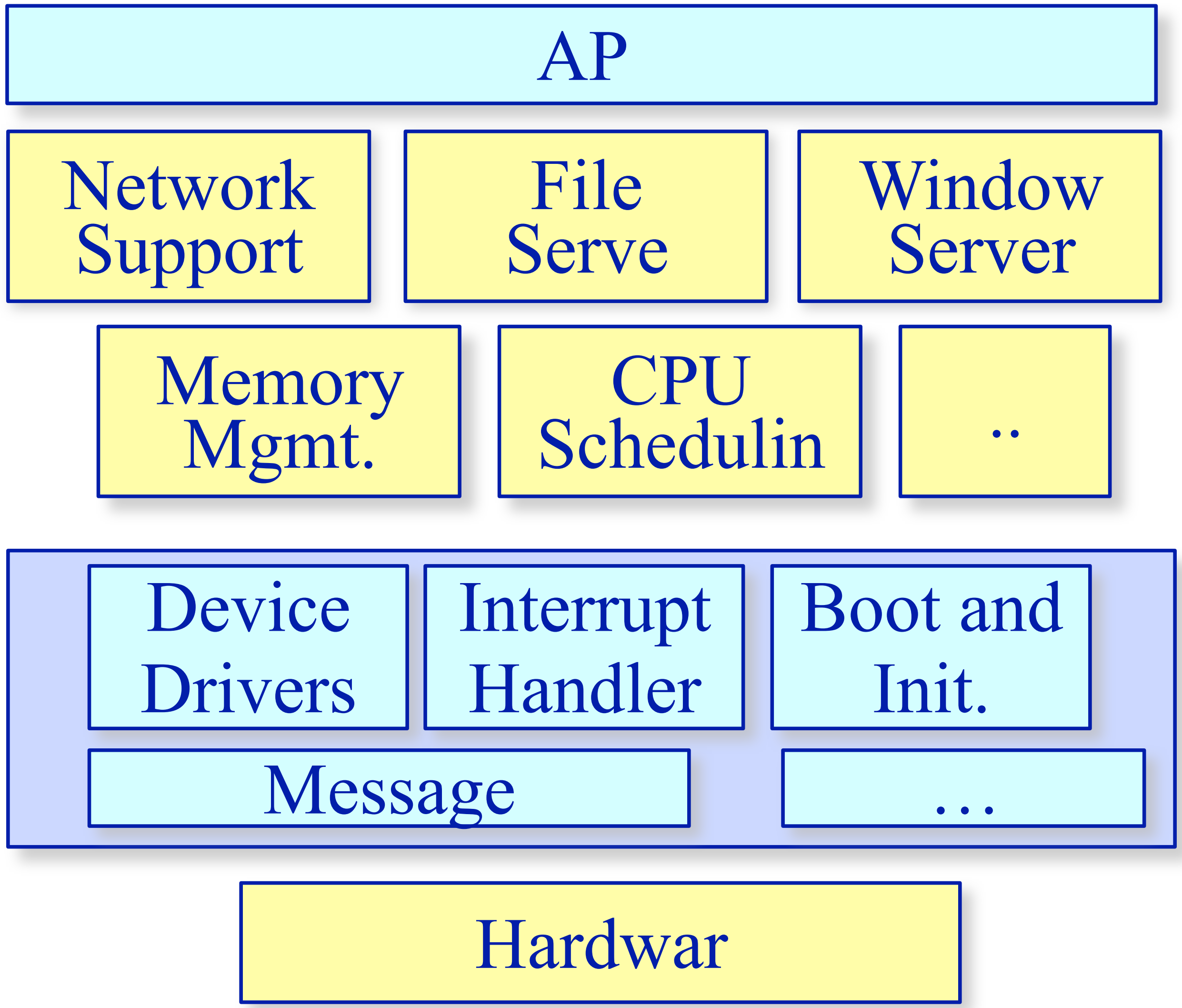


Operating System Structures

- Monolithic OS (e.g., Unix)



- Micro-kernel OS (e.g., Mach, Exokernel, …)



Summary

- An OS is just a program:
  - It has a main() function, which gets called only once (during boot)
  - Like any program, it consumes resources (such as memory), can do silly things (like generating an exception), etc.

- But it is a very strange program:
  - It is "entered" from different locations in response to external events
  - It does not have a single thread of control, it can be invoked simultaneously by two different events (e.g. system call & an interrupt)
  - It is not supposed to terminate
  - It can execute any instruction in the machine