CS 333
Introduction to Operating Systems

Class 1 - Introduction to OS-related Hardware and Software

Jonathan Walpole
Computer Science
Portland State University

About the instructor

- **Instructor - Jonathan Walpole**
  - Professor at PSU since 2004, OGI 1989 – 2004
  - Research Interests: *Operating System Design, Parallel and Distributed Computing Systems*
  - http://www.cs.pdx.edu/~walpole

About CS 333

- **Goals of the class**
  - understand the basic concepts of operating systems
    - *designing & building* operating systems, not *using* them!
  - gain some practical experience so its not just words!

- **Expectations**
  - reading assignments should be read *before* class
  - active participation in class discussions
  - no cheating!

Grading

- **Exams**
  - Mid-term - 25%
  - Final - 25%

- **Coursework**
  - Project – 50%

Text books

"Operating Systems Concepts: Essentials" by Silberschatz, Galvin, Gagne

"The BLITZ System" by Harry Porter

The project

- **You will read, understand and write real operating system code!**
- **We will be using the BLITZ system, written by Harry Porter**
- **About BLITZ**
  - CPU emulator, assembler, high-level language, operating system, and debugging environment
  - Simple enough to understand *in detail* how everything works!
  - Realistic enough to understand *in detail* how everything works!
  - Runs on the departmental Sun machines (cs.pdx.edu), plus Macs and x86/Linux

Administrative

- **Class web site**
  - www.cs.pdx.edu/~walpole/class/cs333/spring2012/home.html
  - Find my website from the faculty listing on the department website. Follow *teaching* link to current classes
- **Class mailing list**
  - https://mailhost.cecs.pdx.edu/cgi-bin/mailman/listinfo/cs333
- **Project 0**
  - read the class web site
  - join the class mailing list
- **Project 1**
  - due next week!
  - see class web site for project assignments

Class 1 - Introduction to OS-related Hardware and Software

Overview

- What is an Operating System?
- A review of OS-related hardware

What is an operating system?

- **Operating system --“a *program* that controls the execution of application programs and implements an interface between the user of a computer and the computer hardware”**
  - Narrow view of a computer and OS
    - **Traditional computer with applications running on it (e.g. PCs, Workstations, Servers)**
  - Broad view of a computer and OS
    - **Anything that needs to manage resources (e.g. router OS, embedded system, cell phone OS ...)**

Two key OS functions

- **Abstract Machine**
  - Hides complex details of the underlying hardware
  - Provides common API to applications and services
  - Simplifies application writing

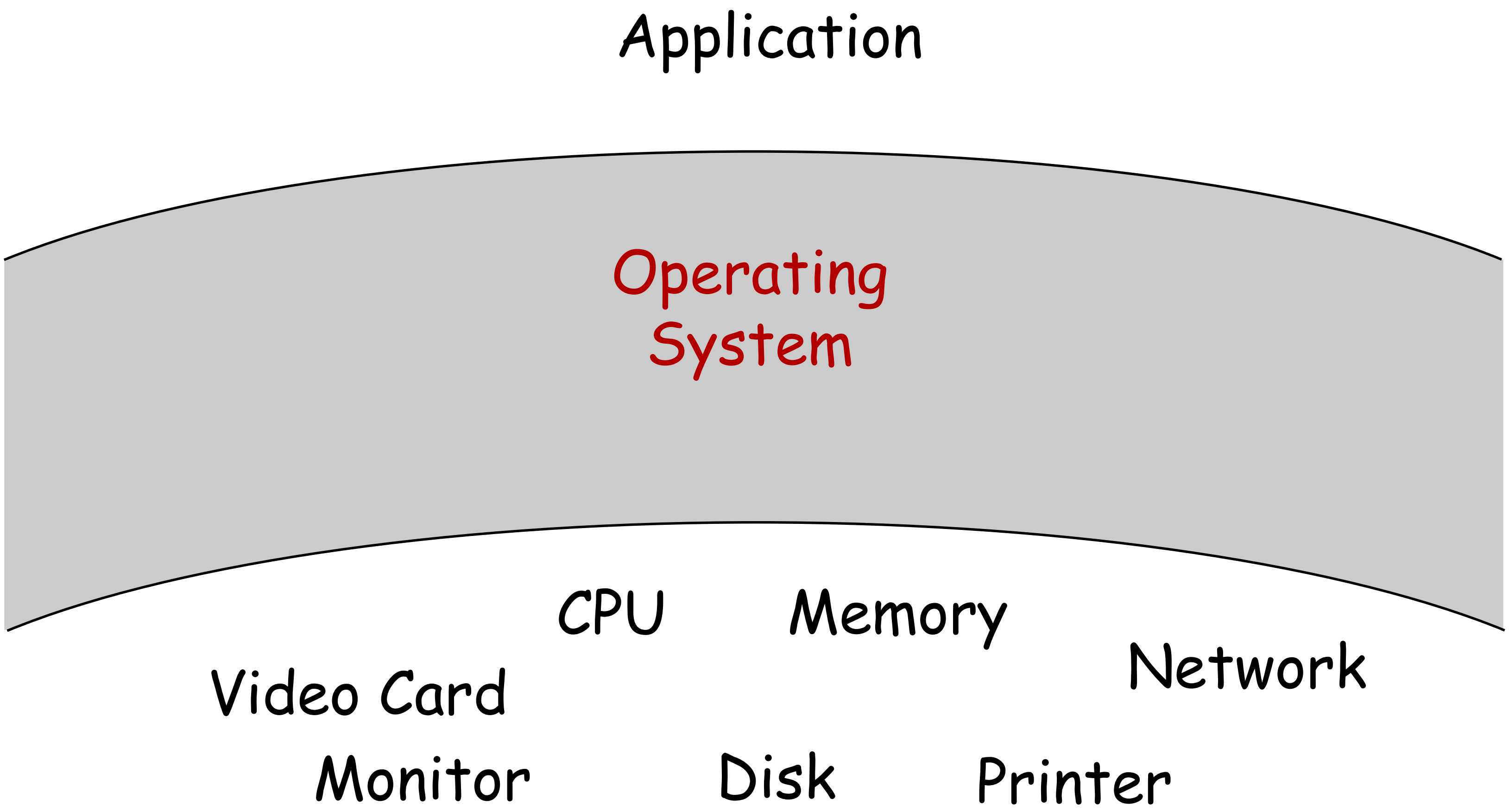
- **Resource Manager**
  - Controls accesses to *shared* resources
    - **CPU, memory, disks, network, ...**
  - Allows for global policies to be implemented

Why is abstraction important?

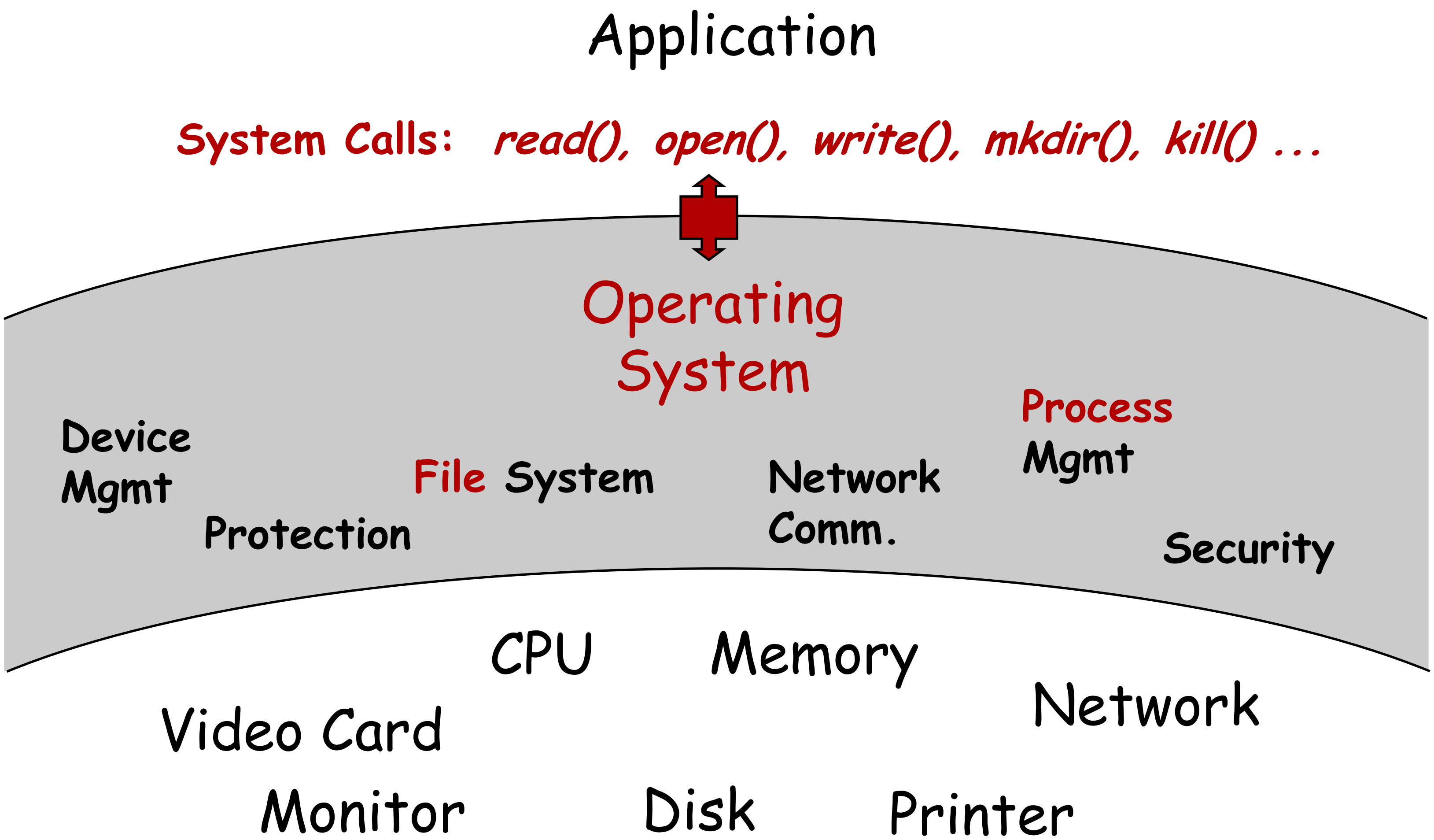
- **Without OSs and abstract interfaces, application writers must program all device access directly**
  - load device command codes into device registers
  - handle initialization, recalibration, sensing, timing etc for physical devices
  - understand physical characteristics and layout
  - control motors
  - interpret return codes … etc

- **Applications suffer severe code bloat!**
  - very complicated maintenance and upgrading
  - no portability
  - writing this code once, and sharing it, is how OS began!

Providing abstraction via system calls



Providing abstraction via system calls



OS as a resource manager

- **Allocating resources to applications across space and time**
  - time sharing a resource (scheduling)
  - space sharing a resource (allocation)
- **Making efficient use of limited resources**
  - improving utilization
  - minimizing overhead
  - improving throughput/good put
- **Protecting applications from each other**
  - enforcement of boundaries

Problems an OS must solve

- Time sharing the CPU among applications
- Space sharing the memory among applications
- Space sharing the disk among users
- Time sharing access to the disk
- Time sharing access to the network

More problems an OS must solve

- **Protection**
  - of applications from each other
  - of user data from other users
  - of hardware/devices
  - of the OS itself!
- **The OS is just a program! It needs help from the hardware to accomplish these tasks!**
  - When an application is running, the OS is not running!
  - When the OS is not running, it can't do anything!

Overview

- What is an Operating System?
- A review of OS-related hardware

Instruction sets

- **A CPU's instruction set defines what it can do**
  - different for different CPU architectures
  - all have load and store instructions for moving items between memory and registers
    - *Load a word located at an address in memory into a register*
    - *Store the contents of a register to a word located at an address in memory*
  - many instructions for comparing and combining values in registers and putting result into a register
- **Look at the Blitz instruction set which is similar to a SUN SPARC instruction set**

Basic anatomy on a CPU

- Program Counter (PC)

Basic anatomy on a CPU

- **Program Counter (PC)**
  - Holds the memory address of the next instruction

Basic anatomy on a CPU

- **Program Counter (PC)**
  - Holds the memory address of the next instruction
- **Instruction Register (IR)**

Basic anatomy on a CPU

- **Program Counter (PC)**
  - Holds the memory address of the next instruction
- **Instruction Register (IR)**
  - Holds the instruction currently being executed

Basic anatomy on a CPU

- **Program Counter (PC)**
  - Holds the memory address of the next instruction
- **Instruction Register (IR)**
  - holds the instruction currently being executed
- **General Registers (Reg. 1..n)**

Basic anatomy on a CPU

- **Program Counter (PC)**
  - Holds the memory address of the next instruction
- **Instruction Register (IR)**
  - holds the instruction currently being executed
- **General Registers (Reg. 1..n)**
  - hold variables and temporary results

Basic anatomy on a CPU

- **Program Counter (PC)**
  - Holds the memory address of the next instruction
- **Instruction Register (IR)**
  - holds the instruction currently being executed
- **General Registers (Reg. 1..n)**
  - hold variables and temporary results
- **Arithmetic and Logic Unit (ALU)**

Basic anatomy on a CPU

- **Program Counter (PC)**
  - Holds the memory address of the next instruction
- **Instruction Register (IR)**
  - holds the instruction currently being executed
- **General Registers (Reg. 1..n)**
  - hold variables and temporary results
- **Arithmetic and Logic Unit (ALU)**
  - performs arithmetic functions and logic operations

Basic anatomy on a CPU

- **Stack Pointer (SP)**

Basic anatomy on a CPU

- **Stack Pointer (SP)**
  - holds memory address of a stack with a frame for each active procedure's parameters & local variables

Basic anatomy on a CPU

- **Stack Pointer (SP)**
  - holds memory address of a stack with a frame for each active procedure's parameters & local variables
- **Processor Status Word (PSW)**

Basic anatomy on a CPU

- **Stack Pointer (SP)**
  - holds memory address of a stack with a frame for each active procedure's parameters & local variables
- **Processor Status Word (PSW)**
  - contains various control bits including the mode bit which determines whether privileged instructions can be executed at this time

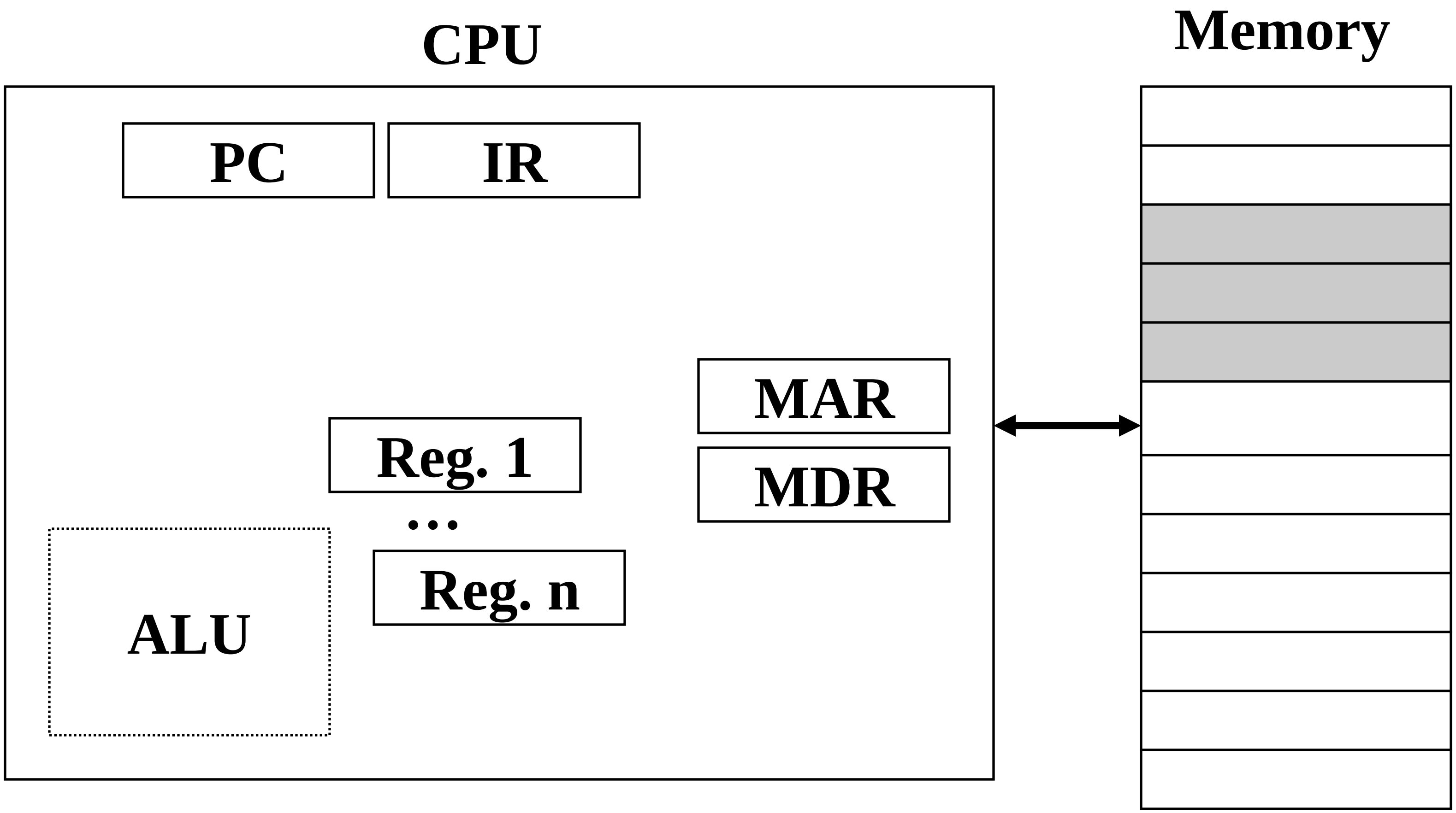
Basic anatomy on a CPU

- **Stack Pointer (SP)**
  - holds memory address of a stack with a frame for each active procedure's parameters & local variables
- **Processor Status Word (PSW)**
  - contains various control bits including the mode bit which determines whether privileged instructions can be executed
- **Memory Address Register (MAR)**
  - contains address of memory to be loaded from/stored to
- **Memory Data Register (MDR)**
  - contains memory data loaded or to be stored

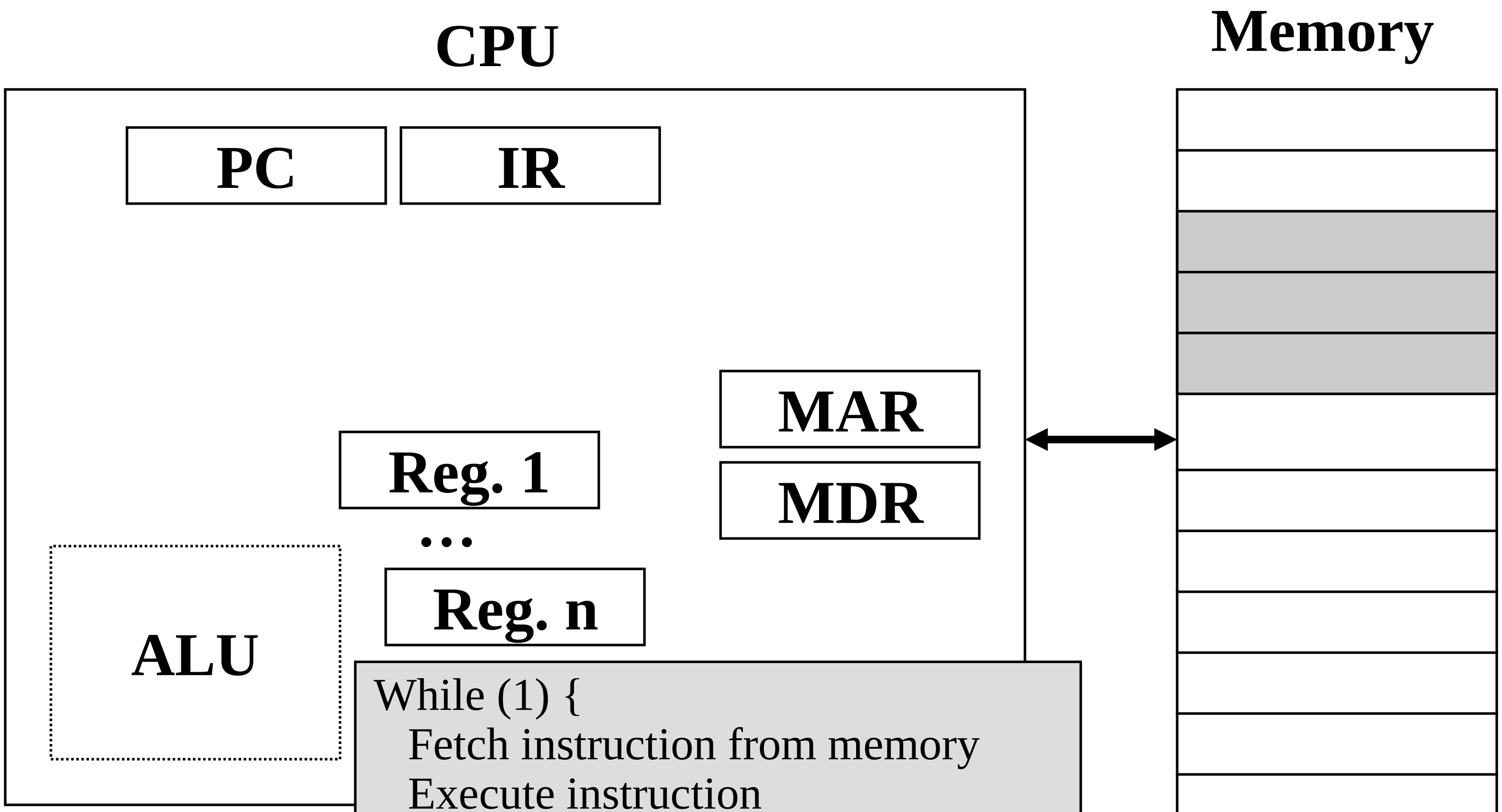
Program execution

- **The Fetch/Decode/Execute cycle**
  - fetch next instruction pointed to by PC
  - decode it to find its type and operands
  - execute it
  - repeat
- **At a fundamental level, fetch/decode/execute is all a CPU does, regardless of which program it is executing**

Fetch/decode/execute cycle

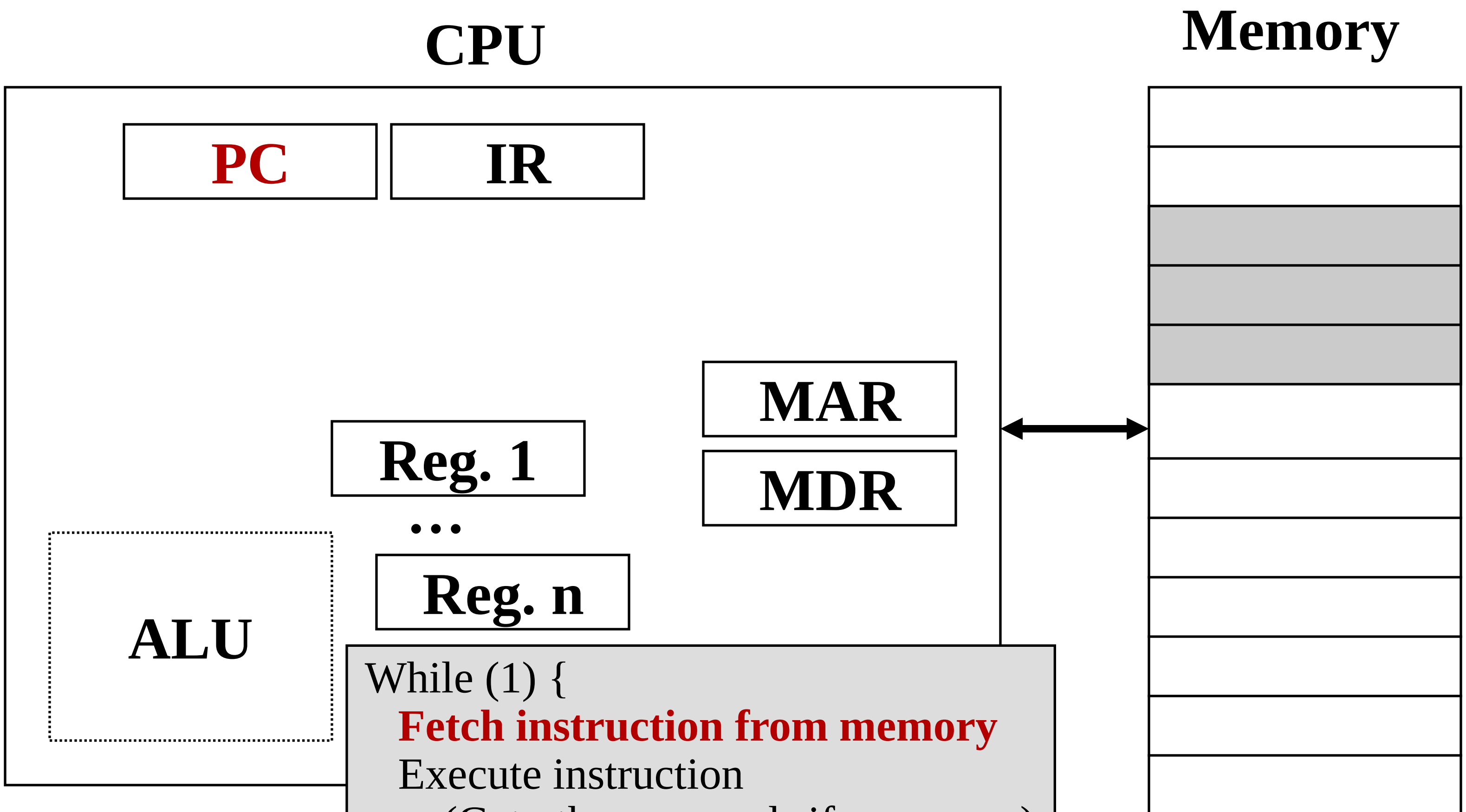


Fetch/decode/execute cycle



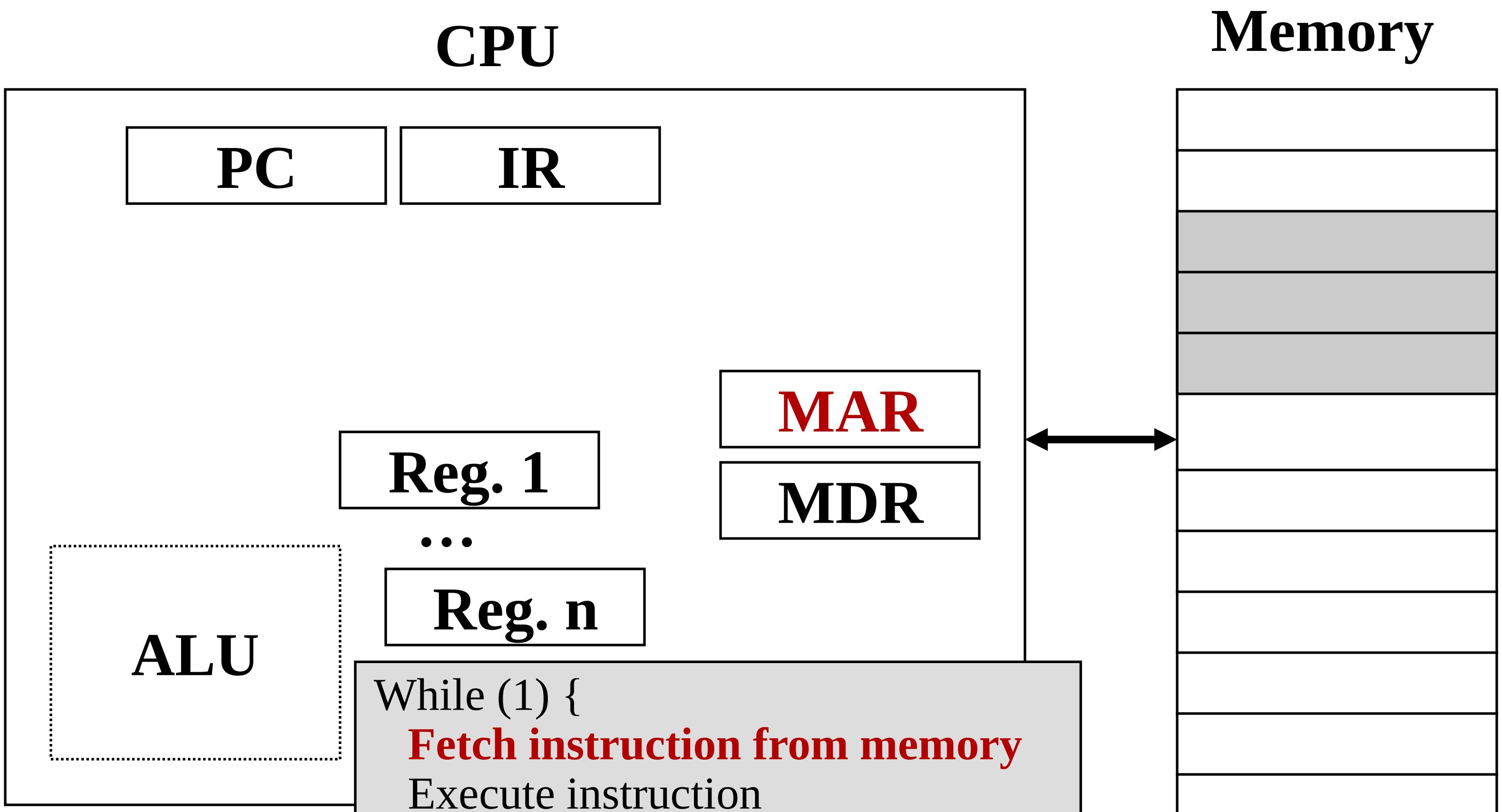

```
While (1) {
  Fetch instruction from memory
  Execute instruction
     (Get other operands if necessary)
  Store result
}
```

Fetch/decode/execute cycle



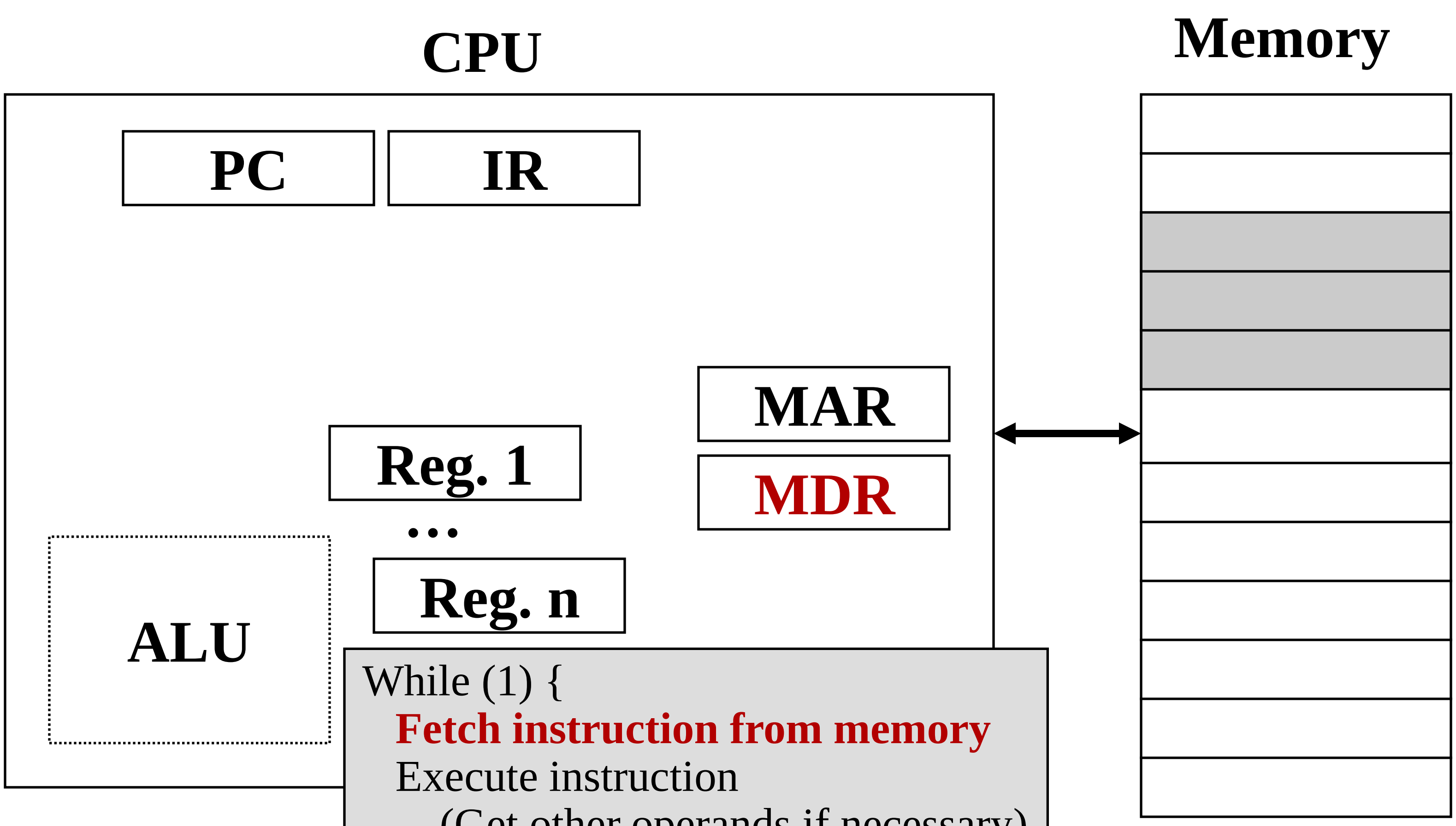
While (1) {
  Fetch instruction from memory
  Execute instruction
    (Get other operands if necessary)
  Store result
}

Fetch/decode/execute cycle



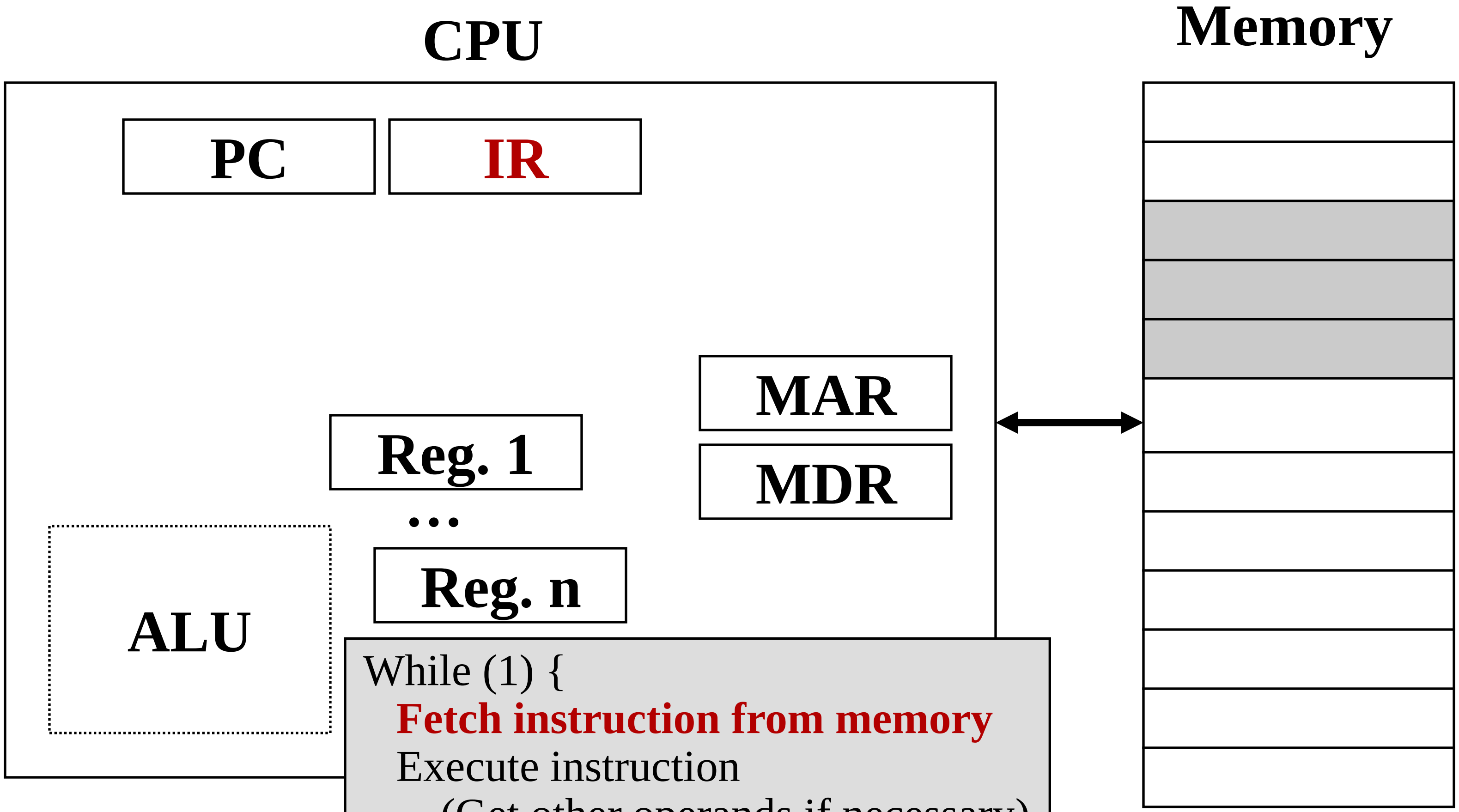

```
While (1) {
   Fetch instruction from memory
   Execute instruction
      (Get other operands if necessary)
   Store result
}
```

Fetch/decode/execute cycle



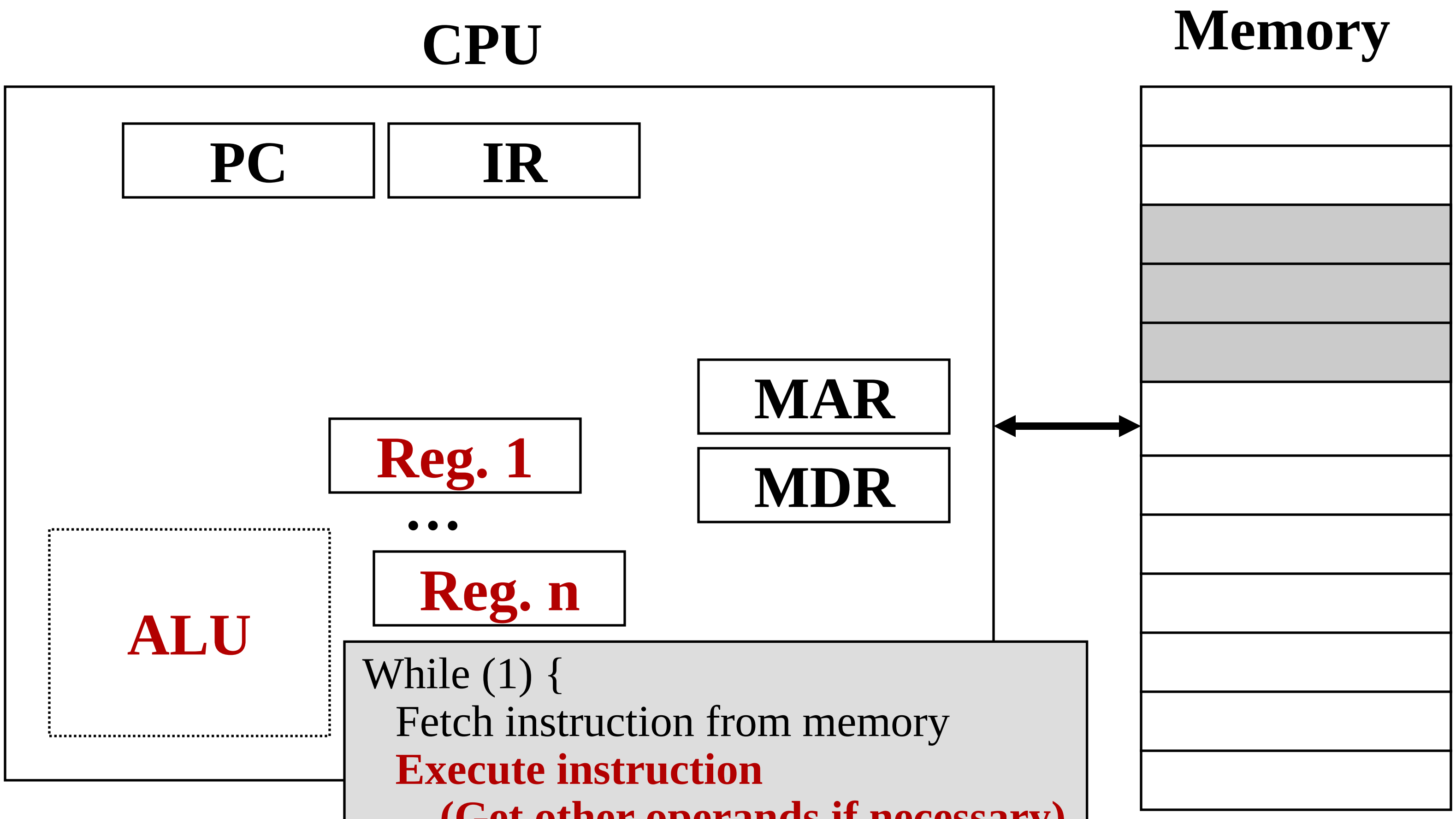

```
While (1) {
   Fetch instruction from memory
   Execute instruction
      (Get other operands if necessary)
   Store result
}
```

Fetch/decode/execute cycle



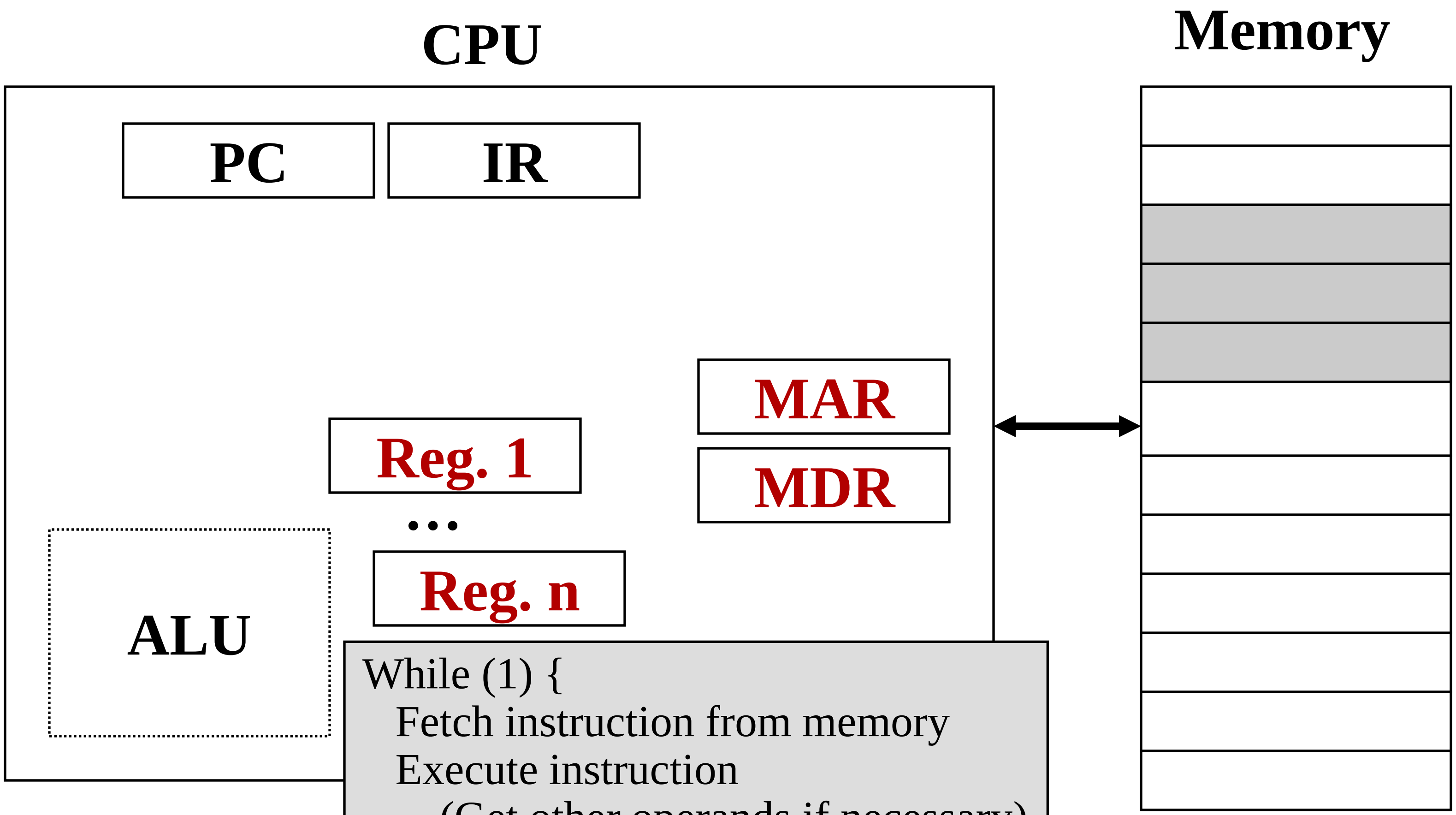
While (1) {
  Fetch instruction from memory
  Execute instruction
    (Get other operands if necessary)
  Store result
}

Fetch/decode/execute cycle



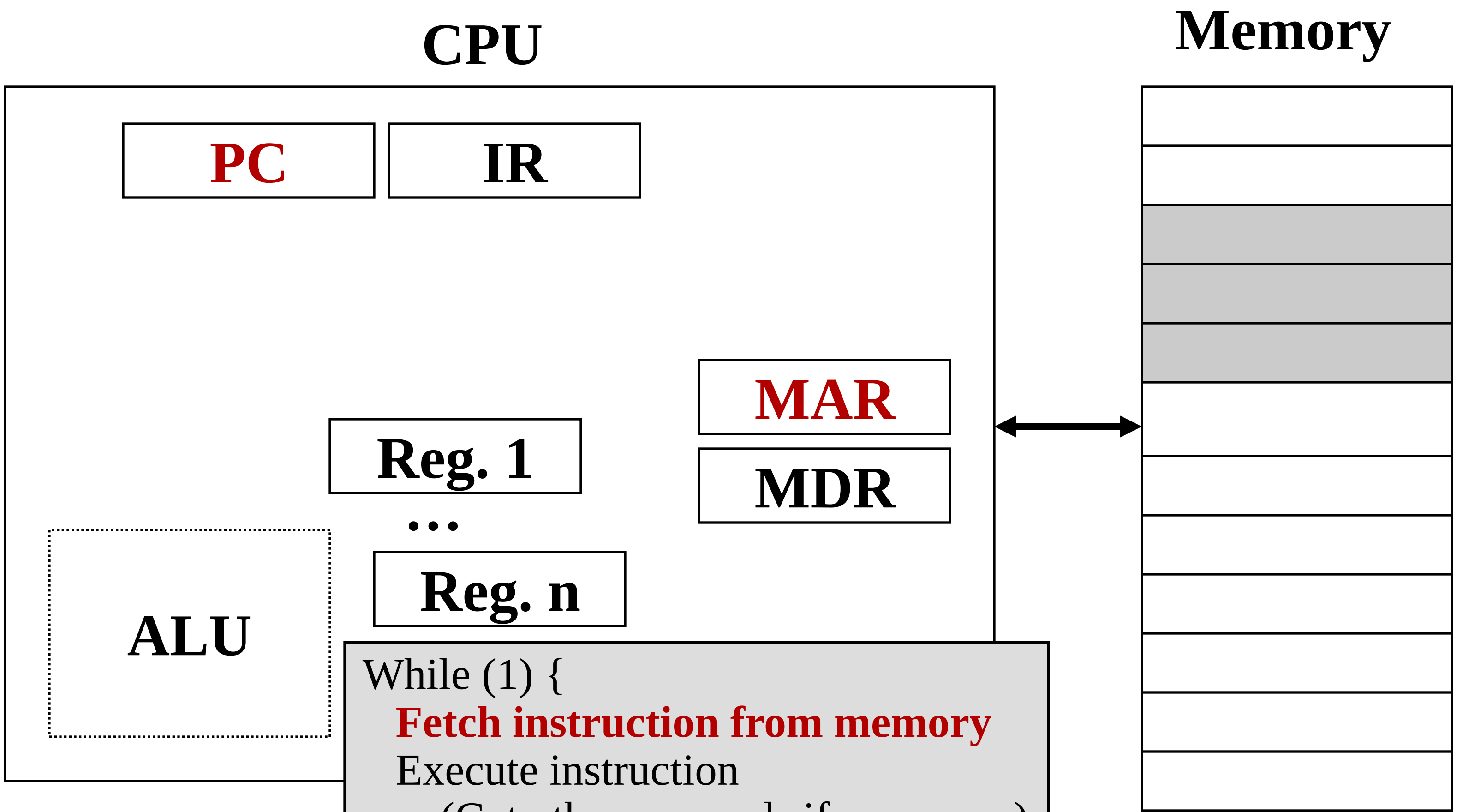

```
While (1) {
  Fetch instruction from memory
  Execute instruction
    (Get other operands if necessary)
  Store result
}
```

Fetch/decode/execute cycle



While (1) {
Fetch instruction from memory
Execute instruction
(Get other operands if necessary)
Store result
}

Fetch/decode/execute cycle




```
While (1) {
   Fetch instruction from memory
   Execute instruction
      (Get other operands if necessary)
   Store result
}
```

The OS is just a program!

- **The OS is a sequence of instructions that the CPU will fetch/decode/execute**
  - How can the OS cause application programs to run?
  - How can the OS switch the CPU to run a different application and later resume the first one?
  - How can the OS maintain control?
  - In what ways can application code try to seize control indefinitely (ie. cheat)?
  - And how can the OS prevent such cheating?
  - How can applications programs cause the OS to run?

How can the OS invoke an application?

How can the OS invoke an application?

- **Somehow, the OS must load the address of the application's starting instruction into the PC**
  - The computer boots and begins running the OS
    - **OS code must be loaded into memory somehow**
    - **fetch/decode/execute OS instructions**
    - **OS requests user input to identify application "file"**
    - **OS loads application file (executable) into memory**
    - **OS loads the memory address of the application's starting instruction into the PC**
    - **CPU fetches/decodes/executes the application's instructions**

How can OS guarantee to regain control?

- **What if a running application doesn't make a system call and hence hogs the CPU?**
  - OS needs interrupts from a timer device!
  - OS must register a future timer interrupt before it hands control of the CPU over to an application
  - When the timer interrupt goes off the interrupt hardware jumps control back into the OS at a pre-specified location called an interrupt handler
  - The interrupt handler is just a program (part of the OS)
  - The address of the interrupt handler's first instruction is placed in the PC by the interrupt h/w

What if the application tries to cheat?

- **What stops the running application from disabling the future timer interrupt so that the OS can not take control back from it?**
  - Disabling interrupts must be a privileged instruction which is not executable by applications
  - The CPU knows whether or not to execute privileged instructions based on the value of the mode bit in the PSW!
- **Privileged instructions can only be executed when the mode bit is set**
  - disabling interrupts
  - setting the mode bit!
  - Attempted execution in non-privileged mode generally causes an interrupt (trap) to occur

What other ways are there to cheat?

- **What stops the running application from modifying the OS?**
  - Specifically, what stops it from modifying the timer interrupt handler to jump control back to the application?

What other ways are there to cheat?

- **What stops the running application from modifying the OS?**
  - Memory protection!
  - Memory protection instructions must be privileged
  - They can only be executed with the mode bit set …
- **Why must the OS clear the mode bit before it hands control to an application?**

How can applications invoke the OS?

- **Why not just set PC to an OS instruction address and transfer control that way?**

How can applications invoke the OS?

- **Special instruction causes a trap / interrupt**
- **Trap instruction changes PC to point to a predetermined OS entry point instruction and simultaneously sets the mode bit**
  - application calls a library procedure that includes the appropriate trap instruction
  - fetch/decode/execute cycle begins at a pre-specified OS entry point called a system call
  - CPU is now running in privileged mode
- **Traps, like interrupts, are hardware events, but they are caused by the executing program rather than a device external to the CPU**

How can the OS switch to a new application?

- **To suspend execution of an application simply capture its memory state and processor state**
  - preserve all the memory values of this application
  - copy values of all CPU registers into a data structure which is saved in memory
  - restarting the application from the same point just requires reloading the register values

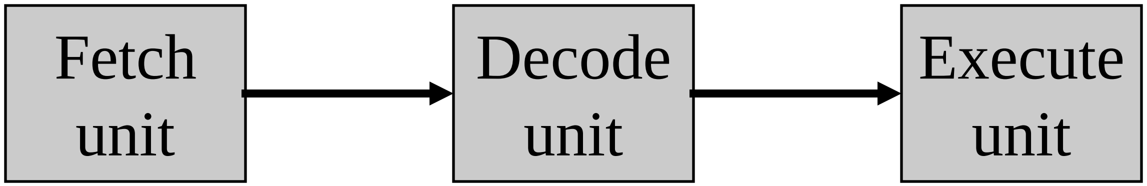
Recap

- Why do we need a timer device?
- Why do we need an interrupt mechanism?
- Why are system calls different to procedure calls?
- How are system calls different to interrupts?
- Why is memory protection necessary?

Why its not quite that simple ...

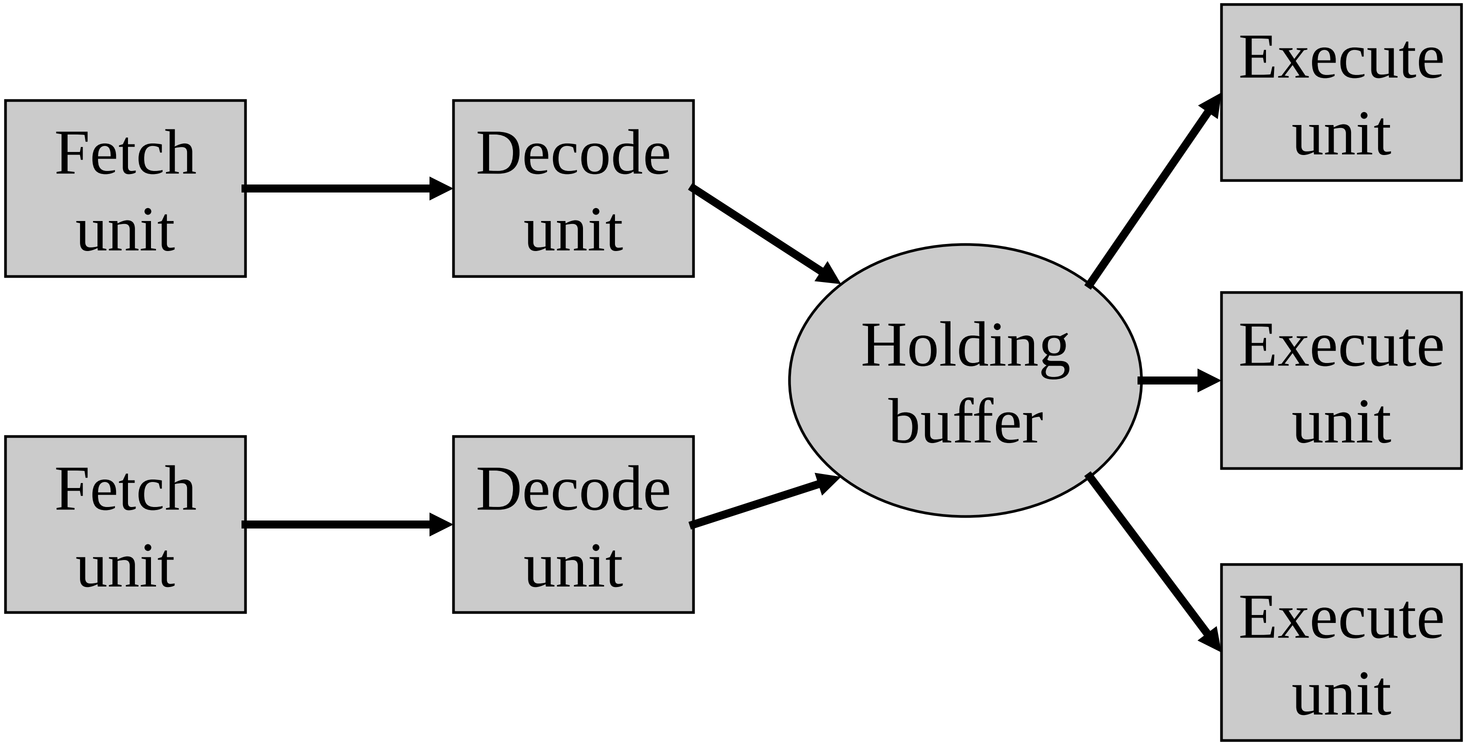
- Pipelined CPUs
- Superscalar CPUs
- Multi-level memory hierarchies
- Virtual memory
- Complexity of devices and buses
- Heterogeneity of hardware

Pipelined CPUs



Execution of current instruction performed in parallel with decode of next instruction and fetch of the one after that

Superscalar CPUs



What does this mean for the OS?

- **Pipelined CPUs**
  - more complexity in capturing state of a running application
  - more expensive to suspend and resume applications
- **Superscalar CPUs**
  - even more complexity in capturing state of a running application
  - even more expensive to suspend and resume applications

- **More details, but fundamentally the same task**
- **The BLITZ CPU is not pipelined or superscalar**

The memory hierarchy

- 2GHz processor → 0.5 ns
- Data/inst. cache → 0.5ns – 10 ns, 64 kB- 1MB
  (this is where the CPU looks first!)
- Main memory → 60 ns, 512 MB – 1GB
- Magnetic disk → 10 ms, 160 Gbytes

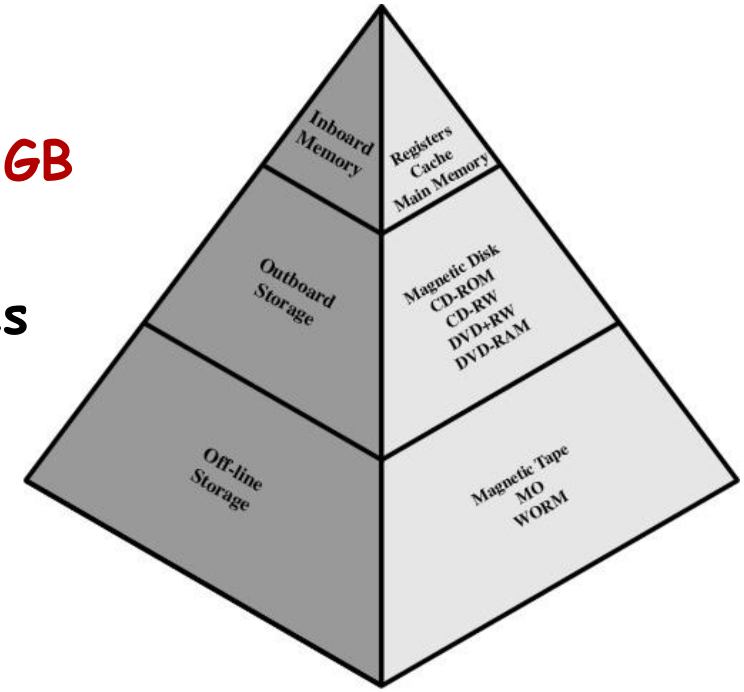


Figure 1.14 The Memory Hierarchy

Who manages the memory hierarchy?

- **Movement of data from main memory to cache is under hardware control**
  - cache lines loaded on demand automatically
  - replacement policy fixed by hardware
- **Movement of data from cache to main memory can be affected by OS**
  - instructions for "flushing" the cache
  - can be used to maintain consistency of main memory
- **Movement of data among lower levels of the memory hierarchy is under direct control of the OS**
  - virtual memory page faults
  - file system calls

OS implications of a memory hierarchy?

- How do you keep the contents of memory consistent across layers of the hierarchy?
- How do you allocate space at layers of the memory hierarchy "fairly" across different applications?
- How do you hide the latency of the slower subsystems?
    - These include main memory as well as disk!
- How do you protect one application's area of memory from other applications?
- How do you relocate an application in memory?

Quiz

How does the OS solve these problems:

- Time sharing the CPU among applications?
- Space sharing the memory among applications?
- Protection of applications from each other?
- Protection of hardware/devices?
- Protection of the OS itself?

What to do before next class

- Reading for today's class
- Reading for Wednesday's class
- Assignment 0 – read class web page and join class email list
- Start project 1 – Introduction to BLITZ