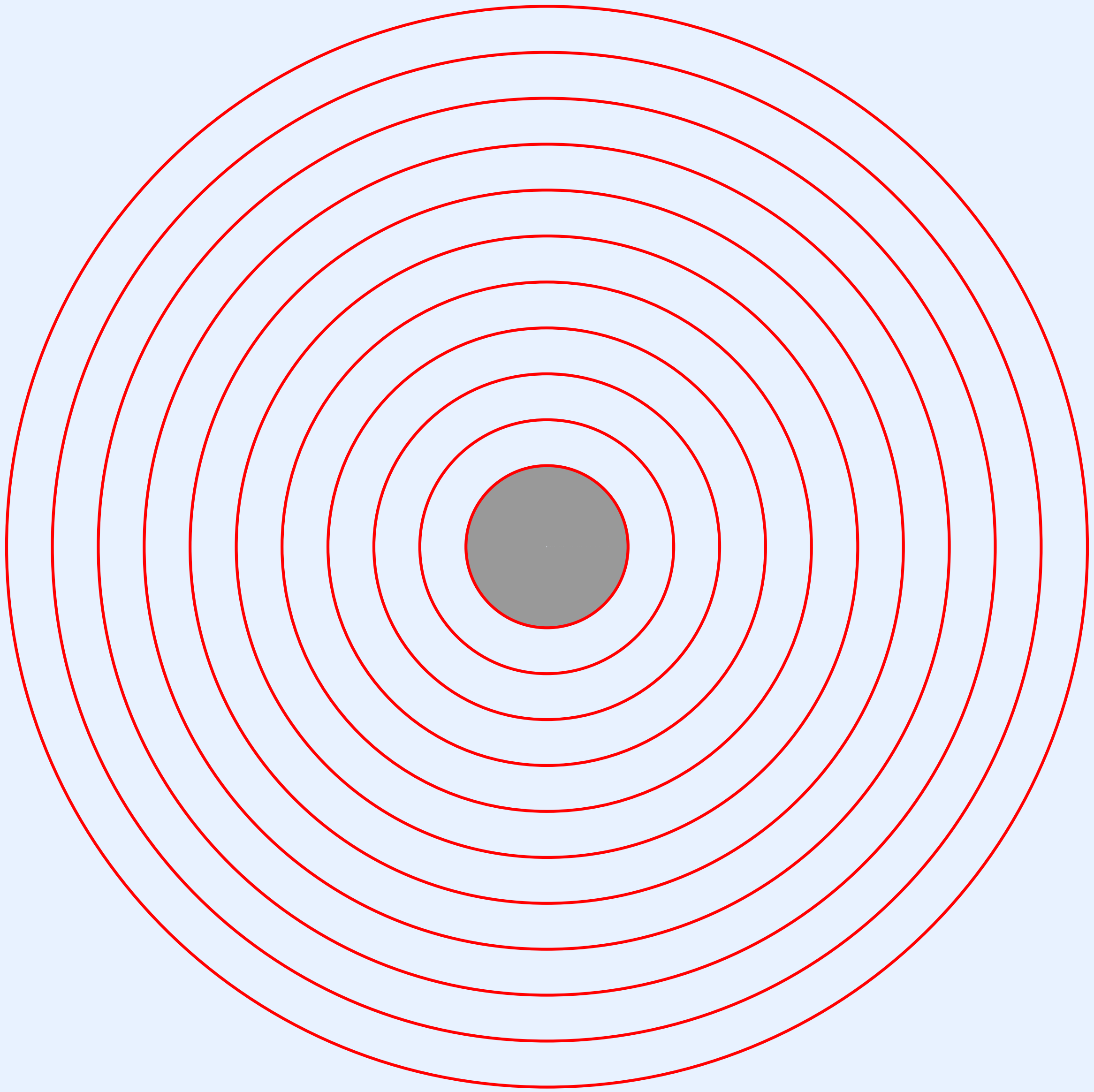
Module III

An Overview Of the Hardware And Runtime Environment

Location Of Hardware In The Hierarchy



Hardware Features An OS Uses Directly

- *Instruction Set Architecture (ISA)* — the instructions the processor offers
- The *general-purpose registers*
  - Used for computation
  - Saved and restored during function invocation
- The main memory system
  - Consists of an array of bytes
  - Holds code as well as data
  - Imposes endianness for integers
  - May provide address mapping for virtual memory

General-Purpose Register Example #1 (32-bit Intel x86)

Name	Use
EAX	Accumulator
EBX	Base
ECX	Count
EDX	Data
ESI	Source Index
EDI	Destination Index
EBP	Base Pointer
ESP	Stack Pointer

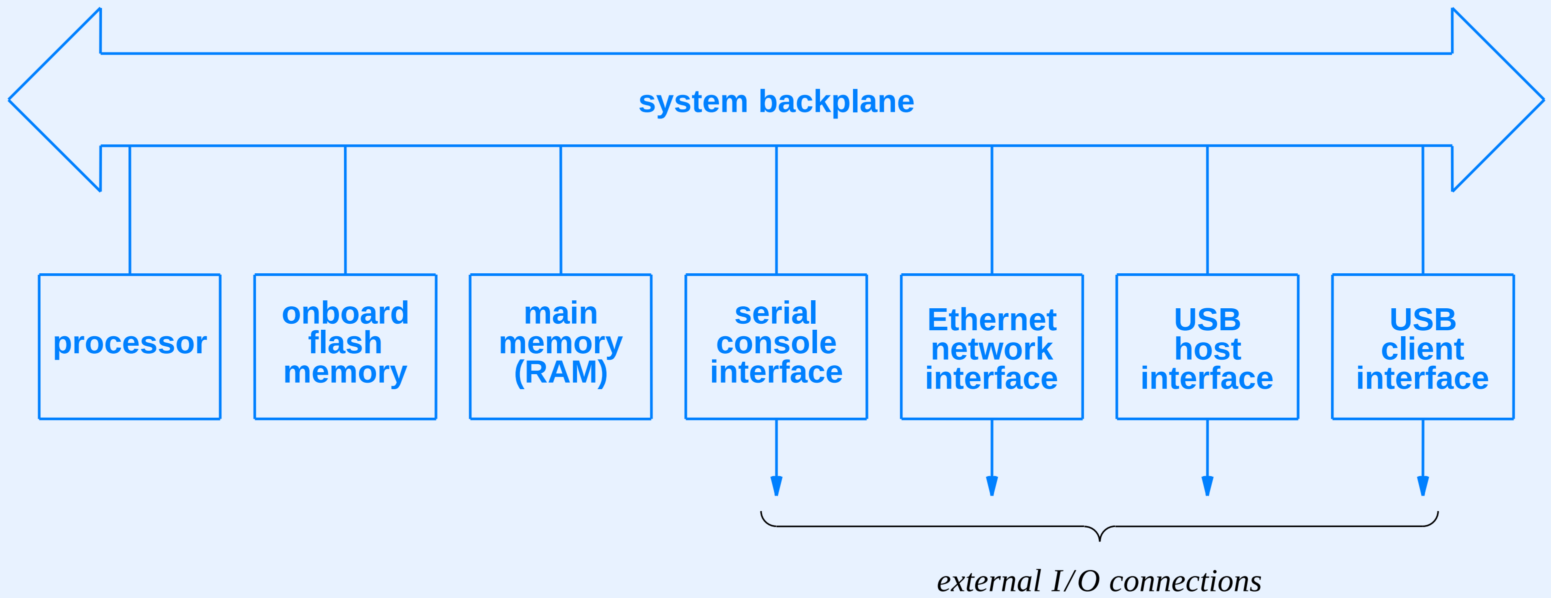
General-Purpose Register Example #1 (32-bit ARM)

Name	Alias	Use
R0 – R3	a1 – a4	Argument registers
R4 – R11	v1 – v8	Variables and temporaries
R9	sb	Static base register
R12	ip	Intra procedure call scratch register
R13	sp	Stack pointer
R14	lr	Link register used for return address
R15	pc	Program counter

Logical And Physical Organizations Of A Platform

- Logically. a computer consists of a
  - Processor
  - Memory
  - Storage
  - I/O devices
- Physically, a computer can consist of
  - A Single self-contained circuit board
  - Many interconnected circuit boards
  - A single chip that contains a processor, memory, and I/O interfaces (called a *System on Chip* (*SoC*)

Illustration Of A Bus Interconnecting Components



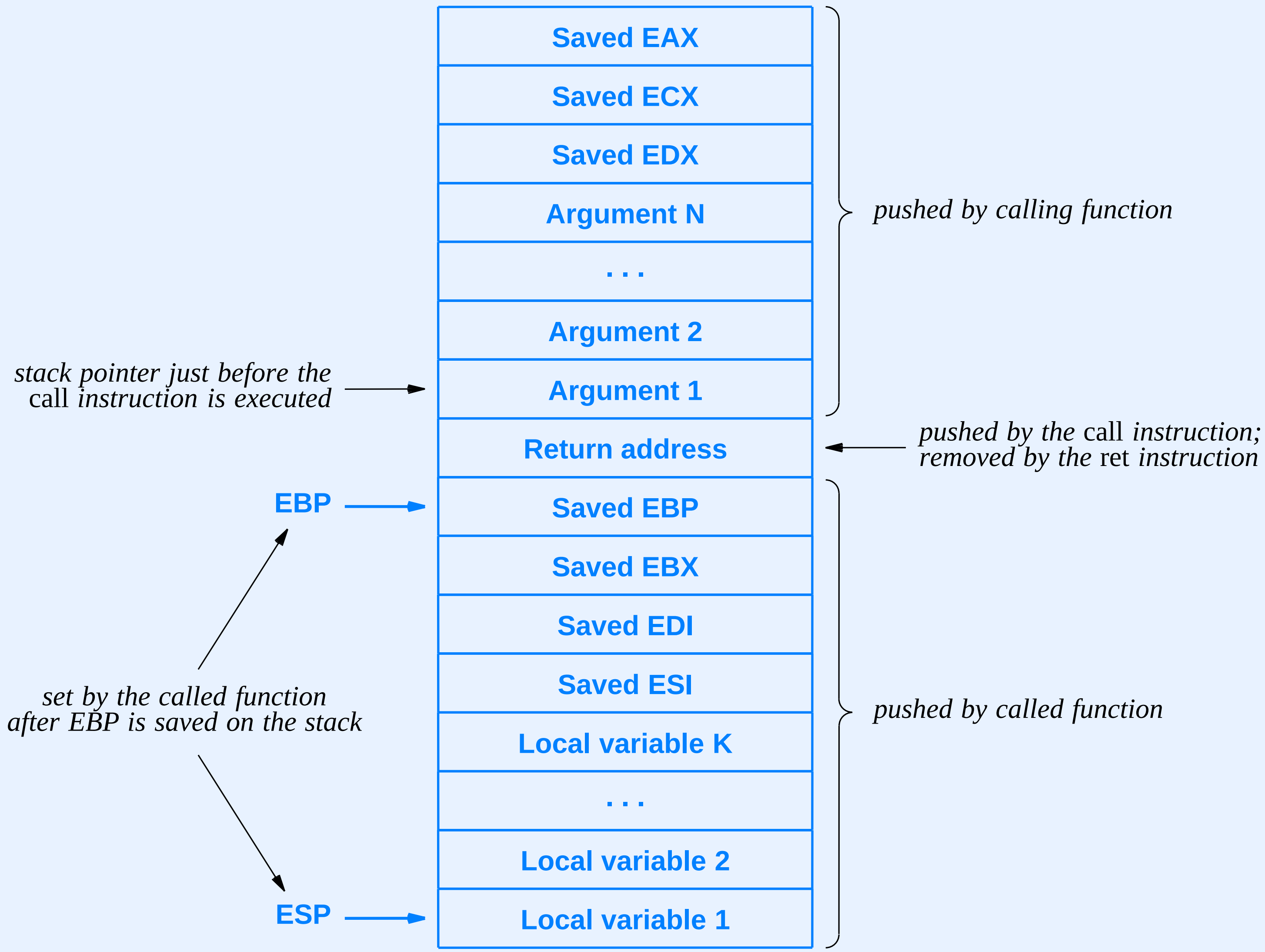
Buses And Fetch-Store

- A bus only permits two operations
  - *Fetch* (processor supplies an address; the hardware returns the value at that address)
  - *Store* (processor supplies a value and an address; the hardware stores the value at the specified address)
- Bus operations
  - Make perfect sense for values in memory
  - Are also used to communicate with I/O devices

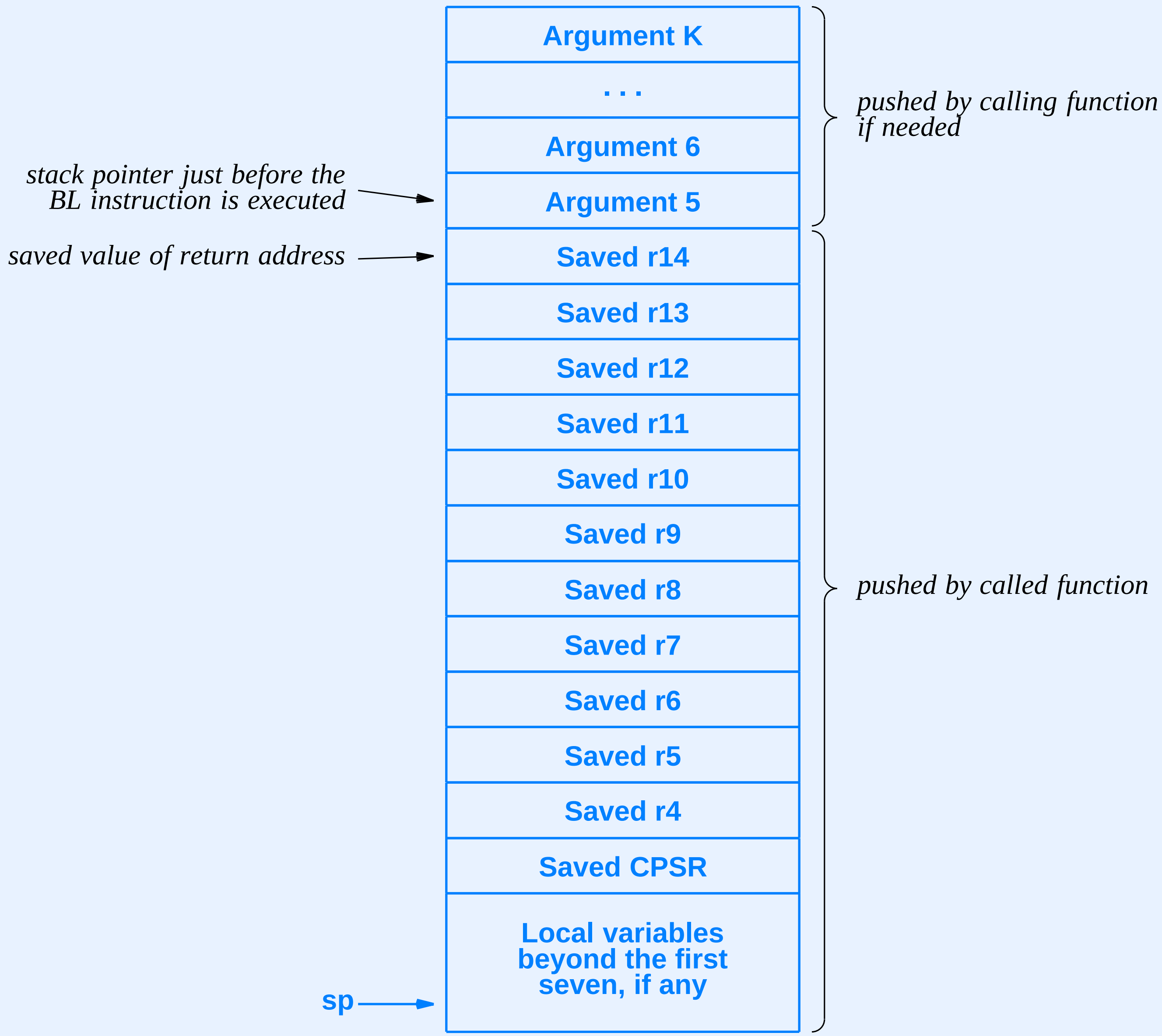
Review From Compilers: Calling Conventions

- How the compiler/hardware pushes values on the run-time stack during a function call (and pops them when the function returns)
- Terminology: some sources use the term *activation record* to refer to the values on the stack for a given function call
- Calling conventions differ among architectures (and possibly compilers)

Example Calling Conventions (x86)



Example Calling Conventions (ARM)



Interrupt

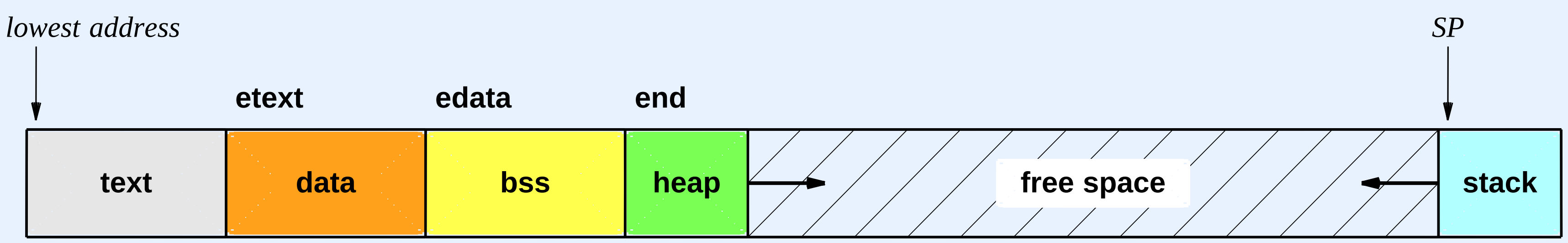
- A hardware mechanism used by an I/O device to tell the processor that an input or output operation has completed
- Causes the processor to stop what it is doing and jump to interrupt code for the device
- Steps taken when an interrupt occurs
  – The hardware or the operating system saves the state of the running computation
  – The processor runs the interrupt code for the device (which must have been placed in memory before the interrupt occurred)
  – When the interrupt code finishes, the OS or hardware must restore the saved state and resume executing at the point where the interrupt occurred
- The running program remains completely unaware that an interrupt occurred while it was running (unless it can measure that the computation took a little longer than expected)

Vectored Interrupts

- We will see that
  - Each device on the bus is assigned a unique *Interrupt Request Number (IRQ)*, 1, 2, 3, ...
  - When it interrupts, a device sends its IRQ over the bus to the processor
  - The hardware uses the IRQ as an index into an array of pointers to functions that handle interrupts for each of the devices
- Note: some processors adds one or more additional interrupt numbers for *exceptions* (e.g., a *divide-by-zero* exception)

Review Of Storage Layout

- When it compiles a C program, the compiler generates four memory *segments*
  - Text segment (compiled code)
  - Data segment (initialized global data values)
  - Bss segment (uninitialized global data values)
  - Stack segment (to hold the run-time stack of activation records)



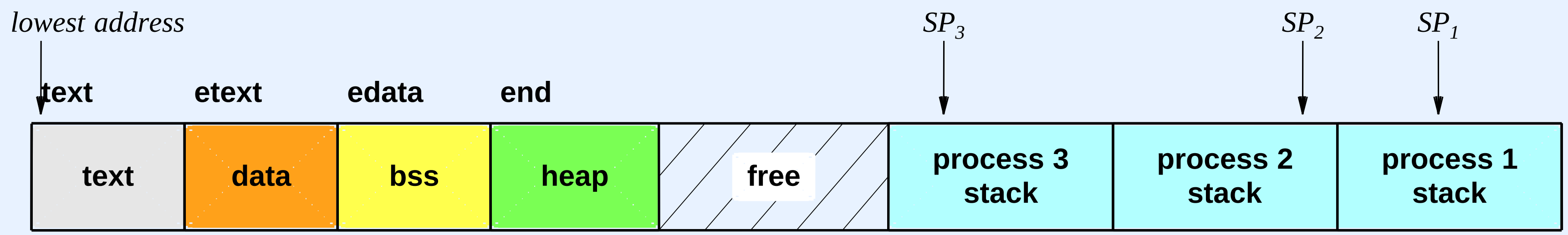
Locations Of Segments

- A compiler includes global variable names that specify segment addresses
  - Symbol *text* occupies the first byte of the text segment
  - Symbol *etext* occupies the first byte beyond the text segment
  - Symbol *edata* occupies the first byte beyond the data segment
  - Symbol *end* occupies the first byte beyond the bss segment
- A programmer can access the names by declaring them *extern*

```
extern  char   text, etext, edata, end;
```

- Only the addresses are significant; the values are irrelevant
- Note: some assembly languages prepend an underscore to, external names (e.g., _*end*)

Storage Layout When Xinu Runs



- Notes:
  - Each process has its own stack for local variables, arguments, and function calls
  - The stack for a process is allocated when a process is created and released when the process exits
  - The text, data, bss, and heap are shared among all processes

Single Core Vs. Multicore Systems

- In almost every class, students are eager to learn about multicore systems
- Our approach: we will start by considering a single-core operating system
- Why?
  - You will see that single-core systems are complex and difficult to understand
  - A multicore operating system is *much* more complex
  - One must understand the principles and operation of a single-core system before diving into the complexities of a multicore system
- Don’t worry — everything you learn about a single-core system will be important in understanding multicore systems

Two Observations

The interface that an operating system provides to application programs operates at a much higher level of abstraction than the underlying hardware.

Because an operating system hides hardware details, it is possible to define a single set of high-level abstractions that can be implemented on multiple hardware architectures.

Questions?