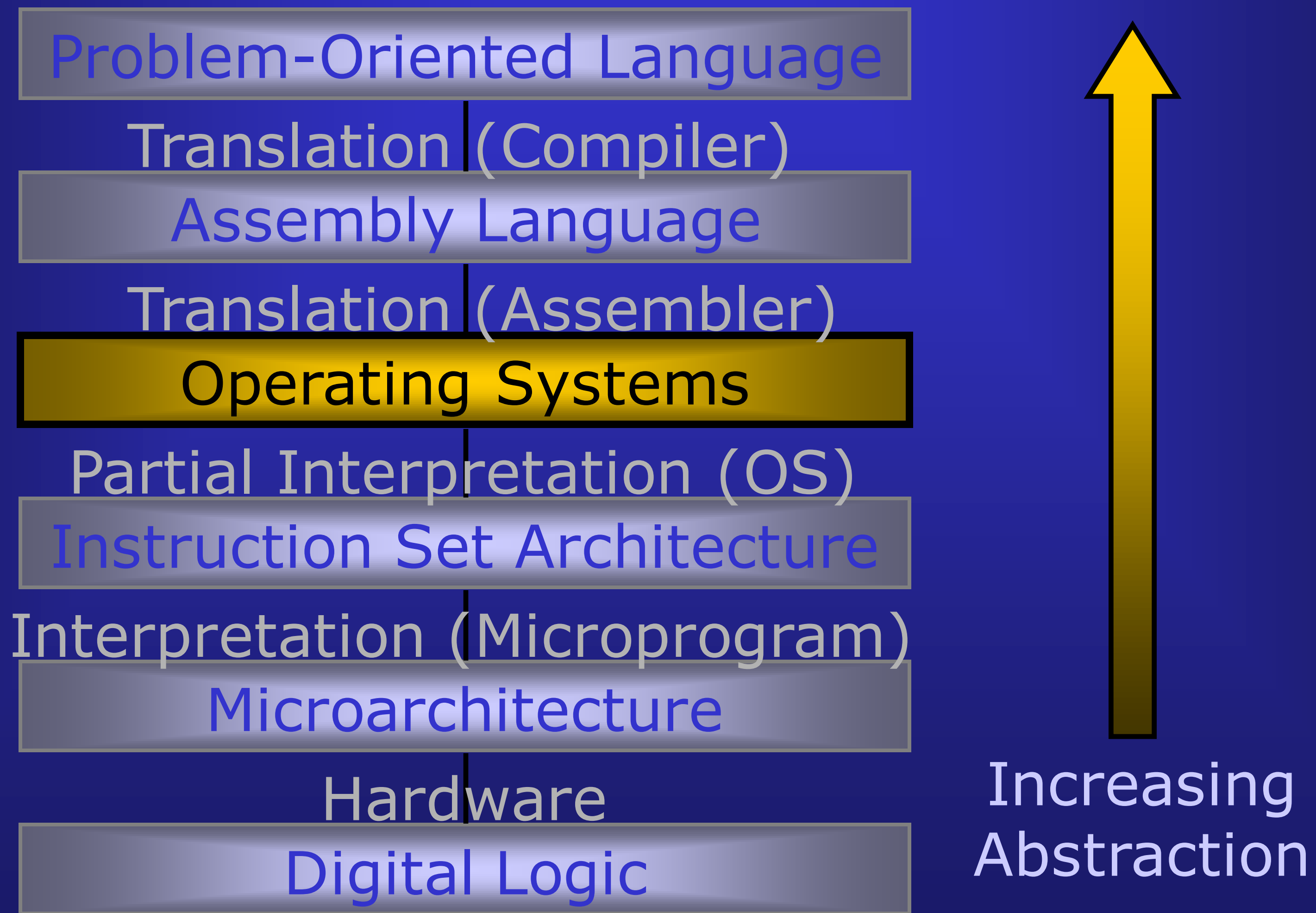
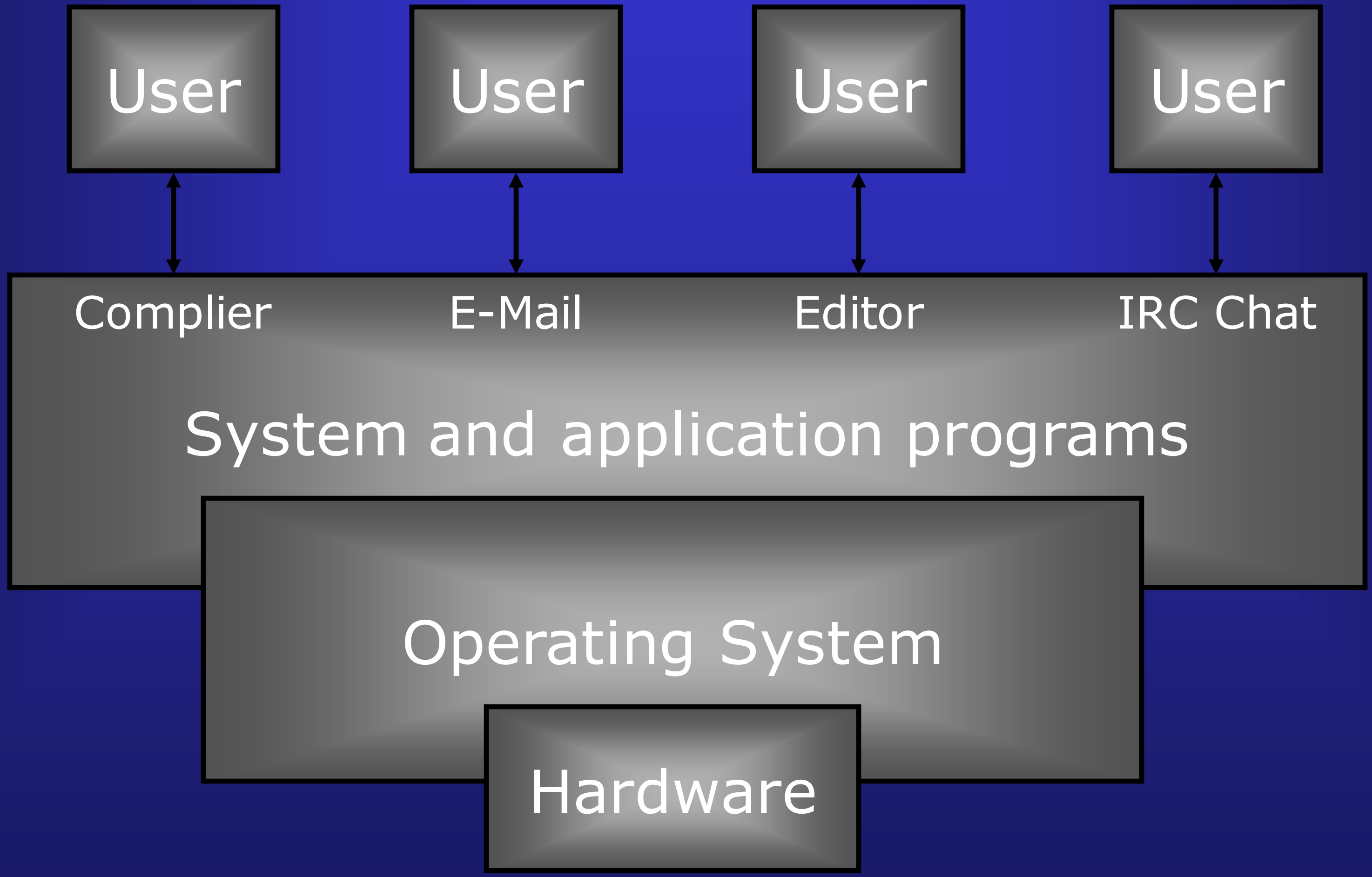
Multilevel Machines



The Operating System Layer



Operating System Tasks

- Process Management
- Memory Management
- File-System Management
- Communications
- Error Detection
- Accounting
- Protection

Protection: The Need for Hardware Support

- Need to ensure that no program can interfere with any other
  - Could overwrite other programs
  - Need to protect the OS in particular
- Graceful recovery from errors detected by the hardware
  - E.g. illegal instructions, divide-by-zero…
- Hardware invokes the OS
  - *Trap*: a "software interrupt"
  - CPU transfers control to an address stored in a trap register

Dual-Mode Operation

- At least two separate modes, implemented using a *mode bit*
  - User mode (mode bit = 0)
  - Supervisor mode (mode bit = 1)
- OS runs in supervisor mode
- User mode is intended to restrict access to the machine
- Also need *privileged instructions*
  - Executed only in supervisor mode
  - Cause a trap if executed in user mode
  - E.g access to the mode bit

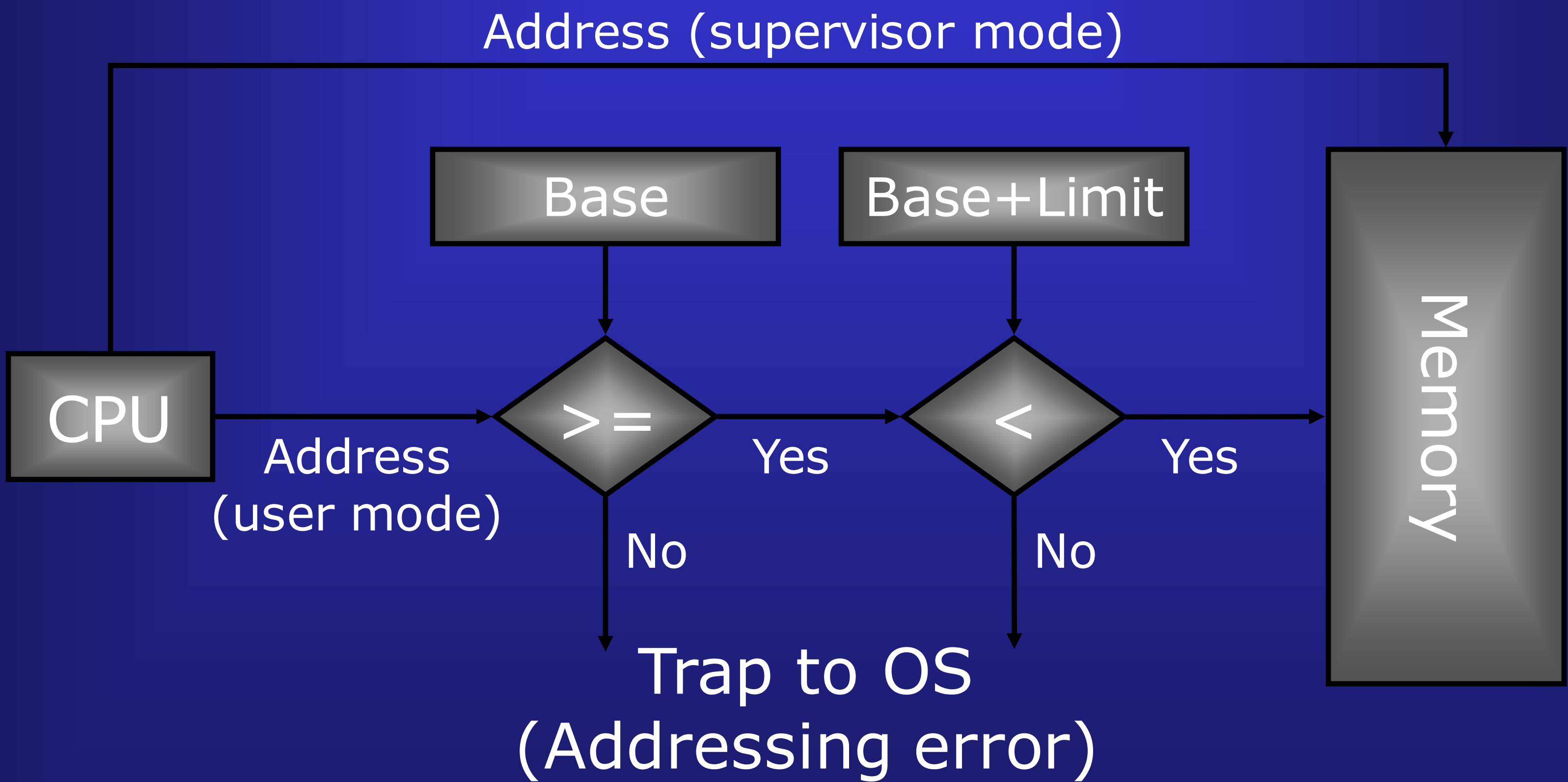
Privileged Instructions: I/O Protection

- User program could issue illegal I/O instructions
  - Write to a non-existence device
  - Read more data than a disk holds
- Solution: make *all* I/O instructions privileged
- User programs can then only perform I/O by requesting it through the OS
- OS retains control over user programs

Memory Protection

- Need to limit access to main memory
  - Prevent modification of OS and other programs
- Give each program a range of legal addresses
  - But need to compare *every* address issued by the user's program
- *Base* and *Limit* registers
  - Implemented in hardware
  - Privileged instructions used to set them
- Supervisor mode has unlimited access

Base and Limit Registers



CPU Protection

- Need to prevent programs from hogging the CPU
  - Infinite loops
  - Waiting for non-existent resources
- Use a *timer* to control program execution
- When timer expires, switch to OS
  - *Context switch*
- This is the idea of *time-slicing*
  - Each program runs for a few msec

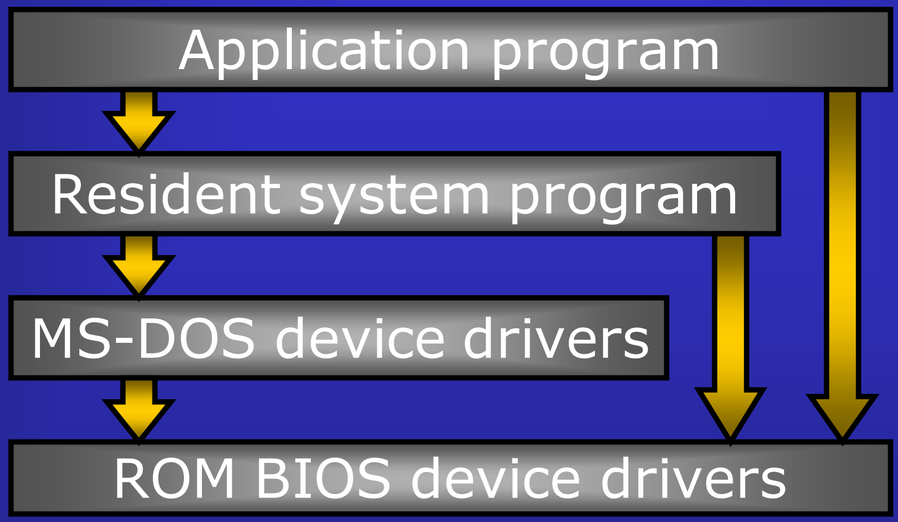
System Calls (the *API*)

- Provide the interface between a process and the OS
  - Process manipulation
  - File and I/O manipulation
  - Information maintenance
  - Communications
- Ask the OS to perform some task on the processes behalf
- Some languages (C, C++, perl,…) allow system calls to be made directly
  - Java hides all system calls

System Programs

- Provide an environment for program development and execution
- Interfaces to system calls
  - Status information (date, time, space…)
  - File modification/manipulation (file creation, deletion, renaming, various editors,…)
  - Programming support (compilers, linkers, debuggers, assemblers, interpreters…)
  - Program loading and execution (command interpreters, graphical interfaces…)
  - Communications (email, ftp, telnet…)

Simple Structure: MS-DOS



- No clear separation of functionality
- No dual-mode, so applications can access hardware directly
  - Fast, but potentially unsafe

PORTABLE OS

1. Problems:

   a) Architecture and OS have strong ties due to which computer manufactures face difficulties in marketing new Architectures as OS development take substantial time

   b) Develop new set of skills for each OS and M/c

2. Hence portability is required and standardization is required from both user and architecture point of view
3. Two components

   a) Policy : governs the use of resource, RR policy
   b) Mechanisms : implements the policies, RR implementation

4. Portable OS is structured so as to segregate the policy from the mechanism. This makes the policy implementation of a m/c on which the OS is to be used

5. Mechanism : Kernel of OS to be rewritten for the new m/c

6. UNIX supports portability, layered OS

7. Now work is going on for micro-kernel level to introduce more portability.

UNIX

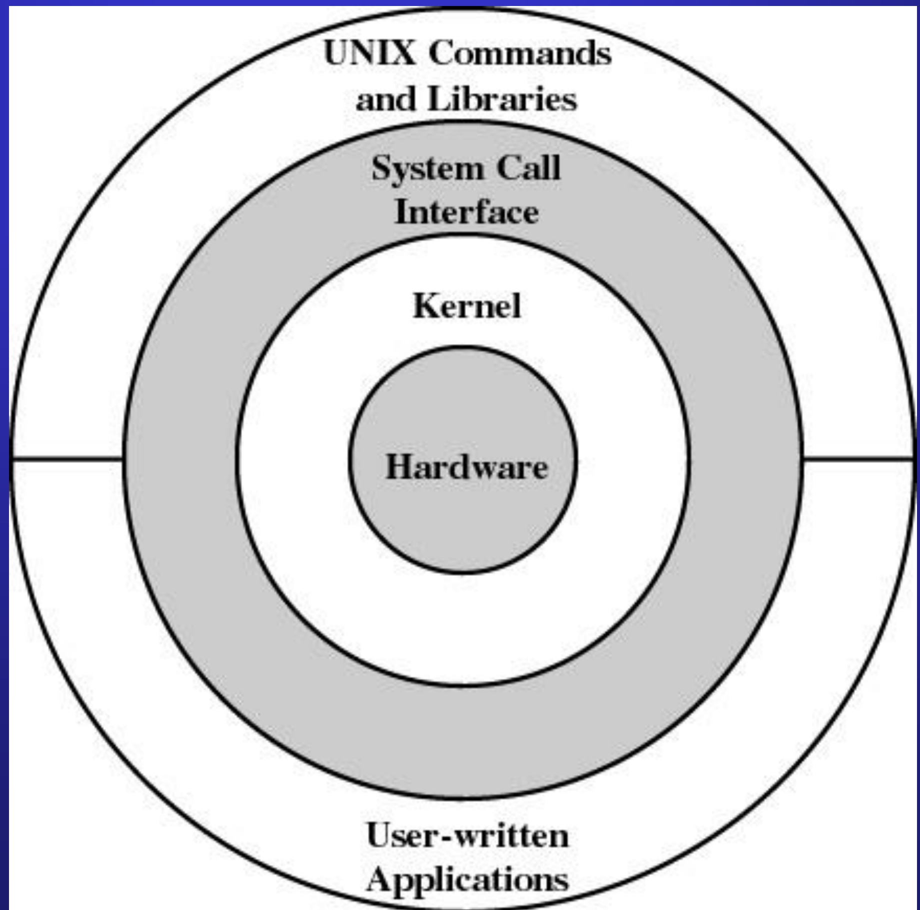


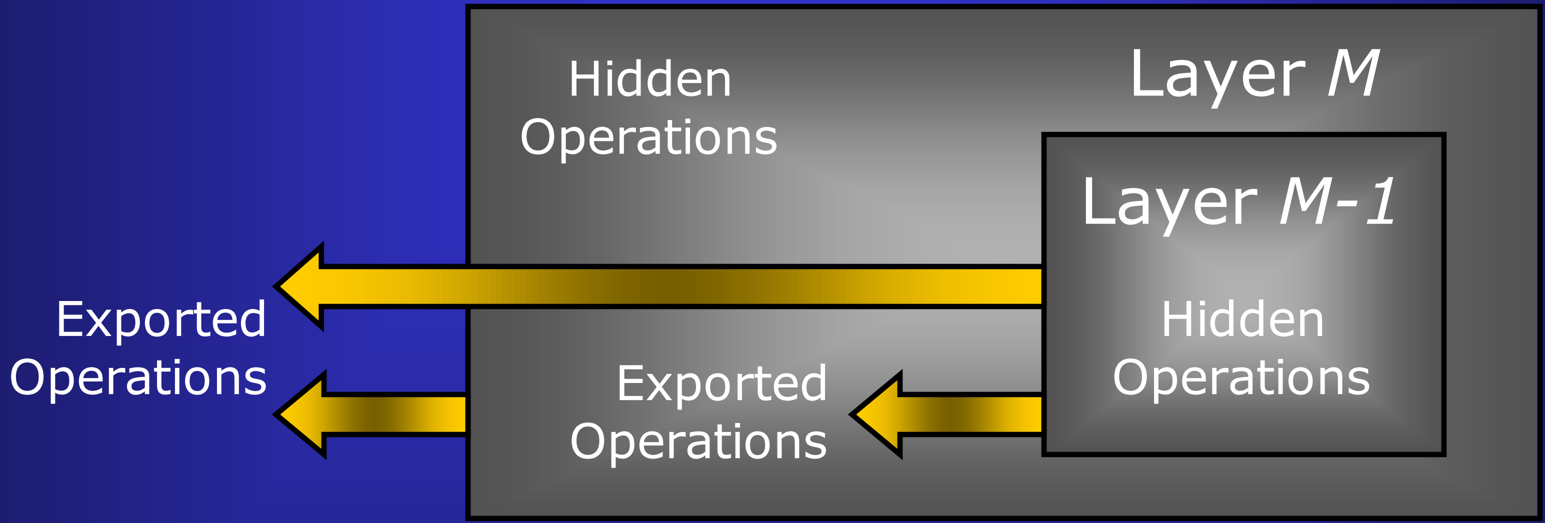
Figure 2.15 General UNIX Architecture

Kernel + System Programs: UNIX

Users		
Shells and commands Compilers and interpreters System libraries		
System-call interface to kernel		
Terminal handling Character I/O Terminal drivers	File system Block I/O Disk/tape drivers	CPU scheduling Page replacement Demand paging Virtual Memory
Kernel interface to hardware		
Terminal controllers Terminals	Device controllers Disks and tapes	Memory controllers Physical memory

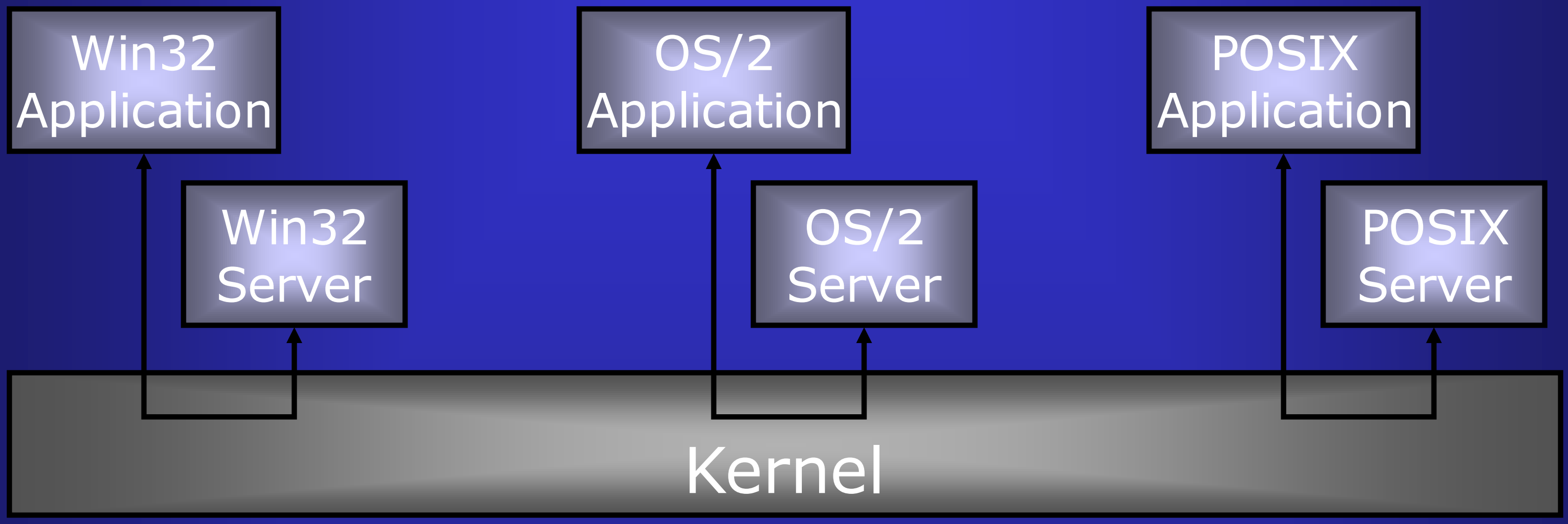
- Easy to port to different hardware
- But the kernel is very complex

Layered Structure: OS/2



- Bottom Layer = hardware
- Top layer = user interface
- Modularity: very easy to debug
  - Difficult to assign appropriate functions to each layer
  - Layers add overhead, so can be inefficient

Microkernels: NT



- All non-essential parts of kernel are user-level programs
  - Reliability and security
- Message-passing communication
  - Client/Server architecture
- Easy to extend OS
  - Don't need to modify kernel

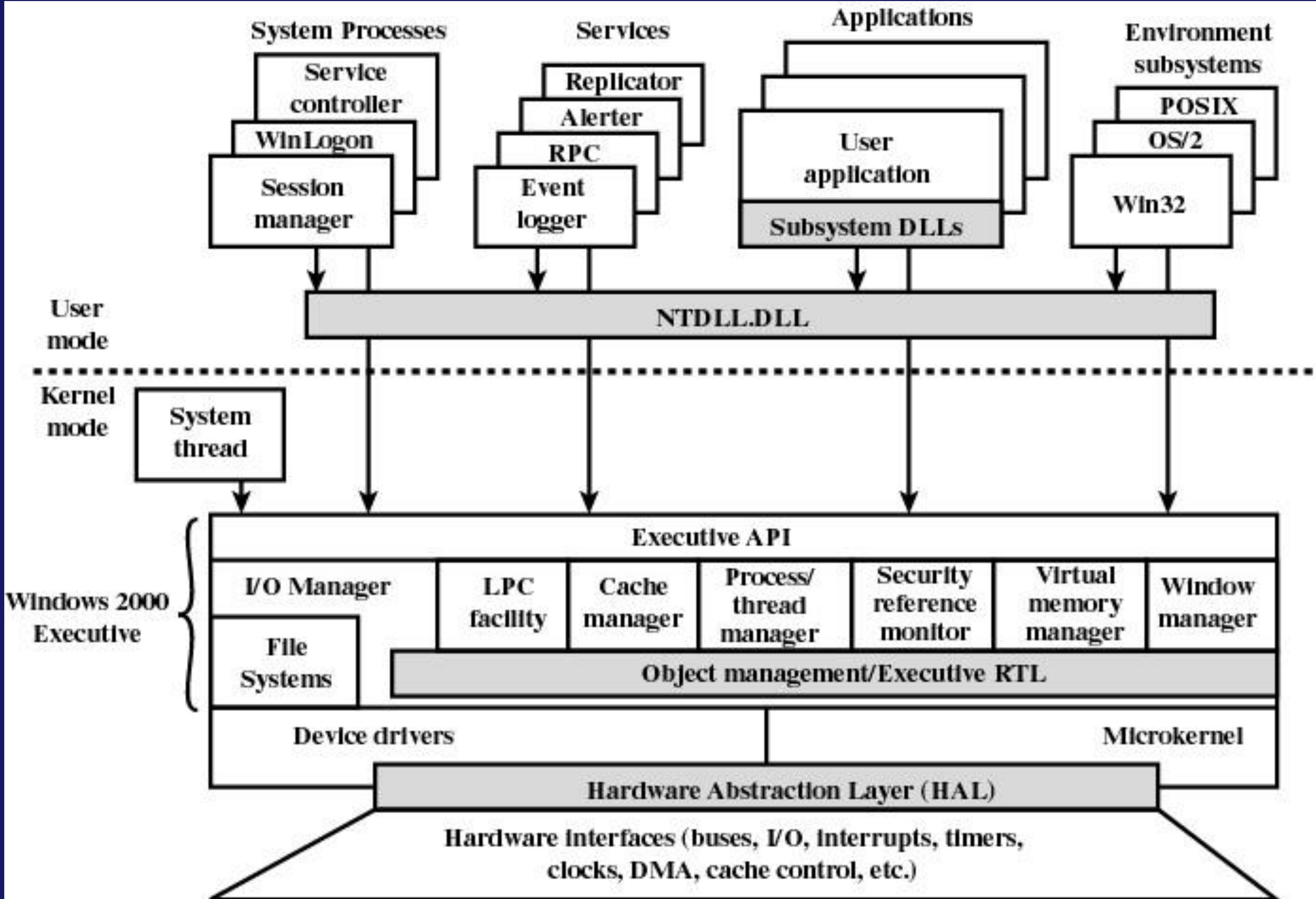
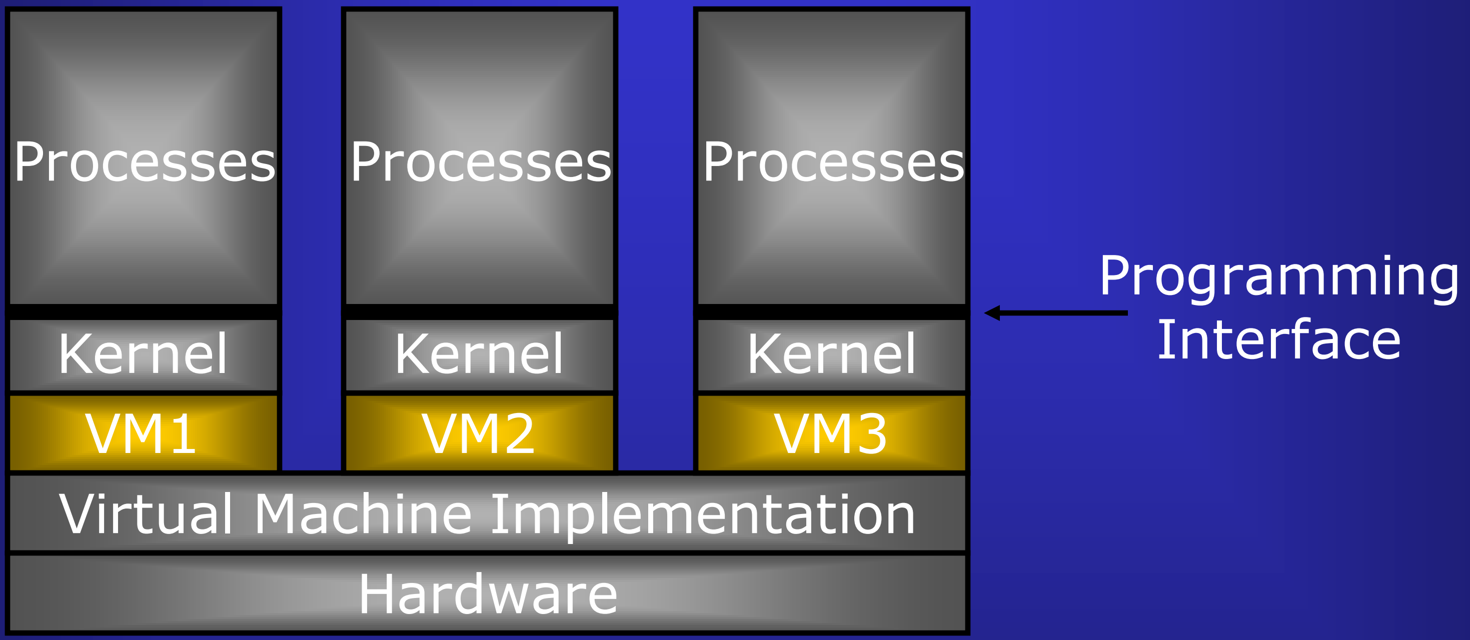


Figure 2.13 Windows 2000 Architecture

Virtual Machines: IBM



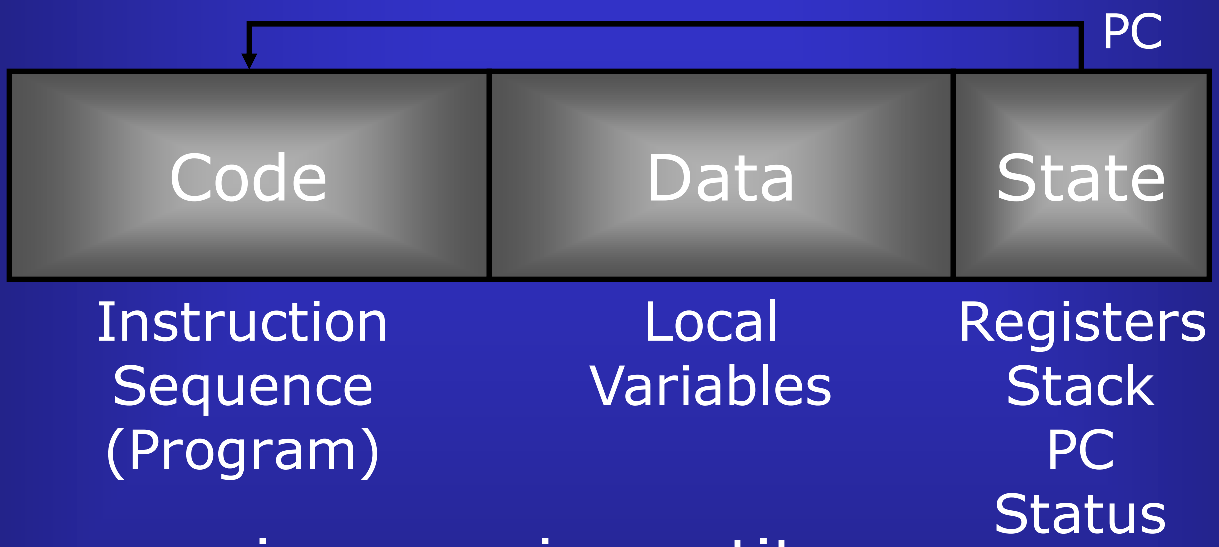
- Provide processes with a virtual copy of underlying hardware
  - Instruction interpretation
- Users have their own virtual machine
  - Difficult to provide *exact* duplicate of hardware

Processes and Threads

We will see:

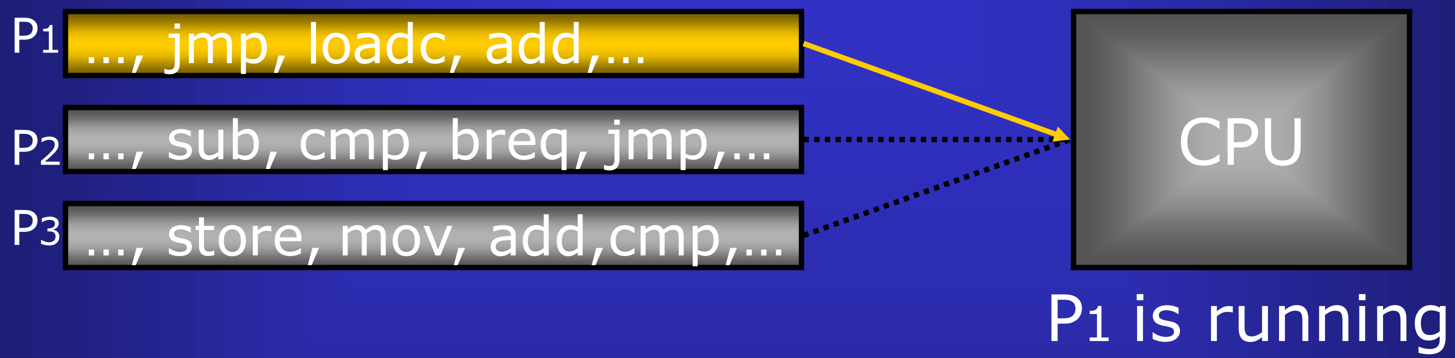
- What a process is, and what it does
- The idea of concurrent execution
- What a thread is
- How processes are implemented in the operating system
- How the operating system decides which job to run

The Process Concept



- A *program* is a passive entity
  - Just a sequence of instructions
- A *process* is an active entity
  - The program in execution
- A *processor* is an agent which executes processes

Concurrency

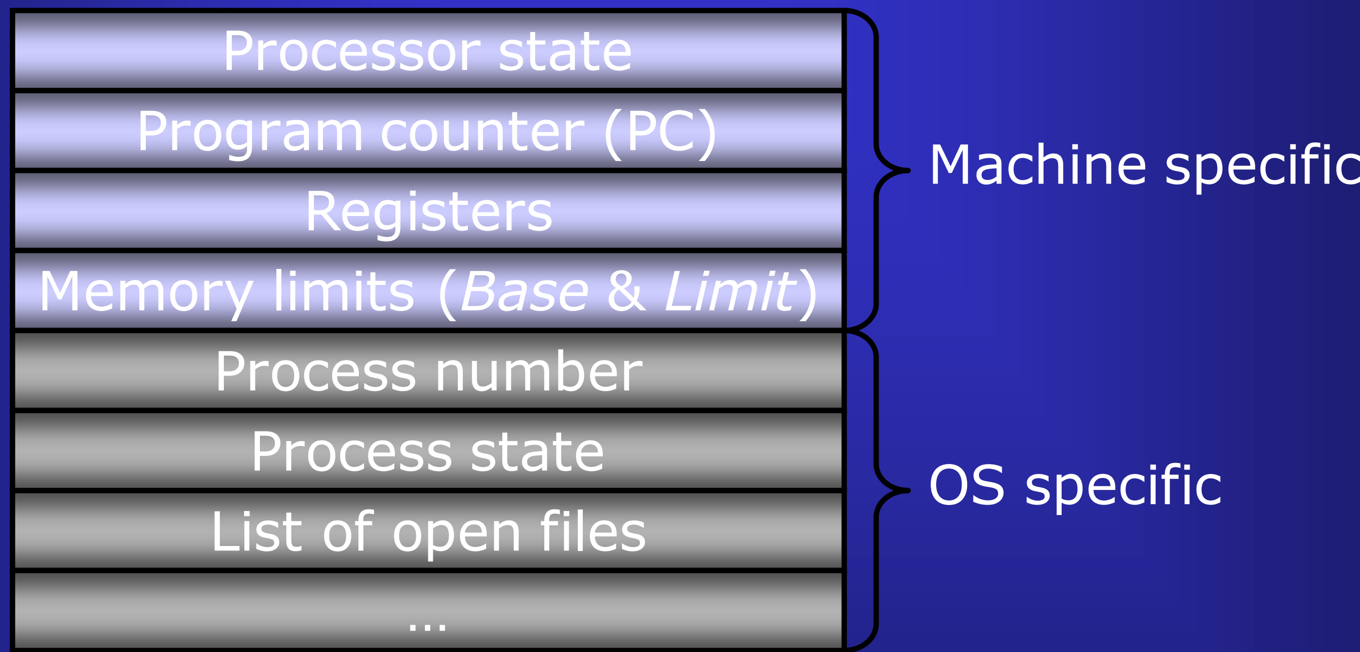


- *Concurrent processes*
  - Activation of more than one process
- Apparent concurrency can be achieved by switching a processor between several processes
  - *Instruction interleaving*

Processes can Share Code

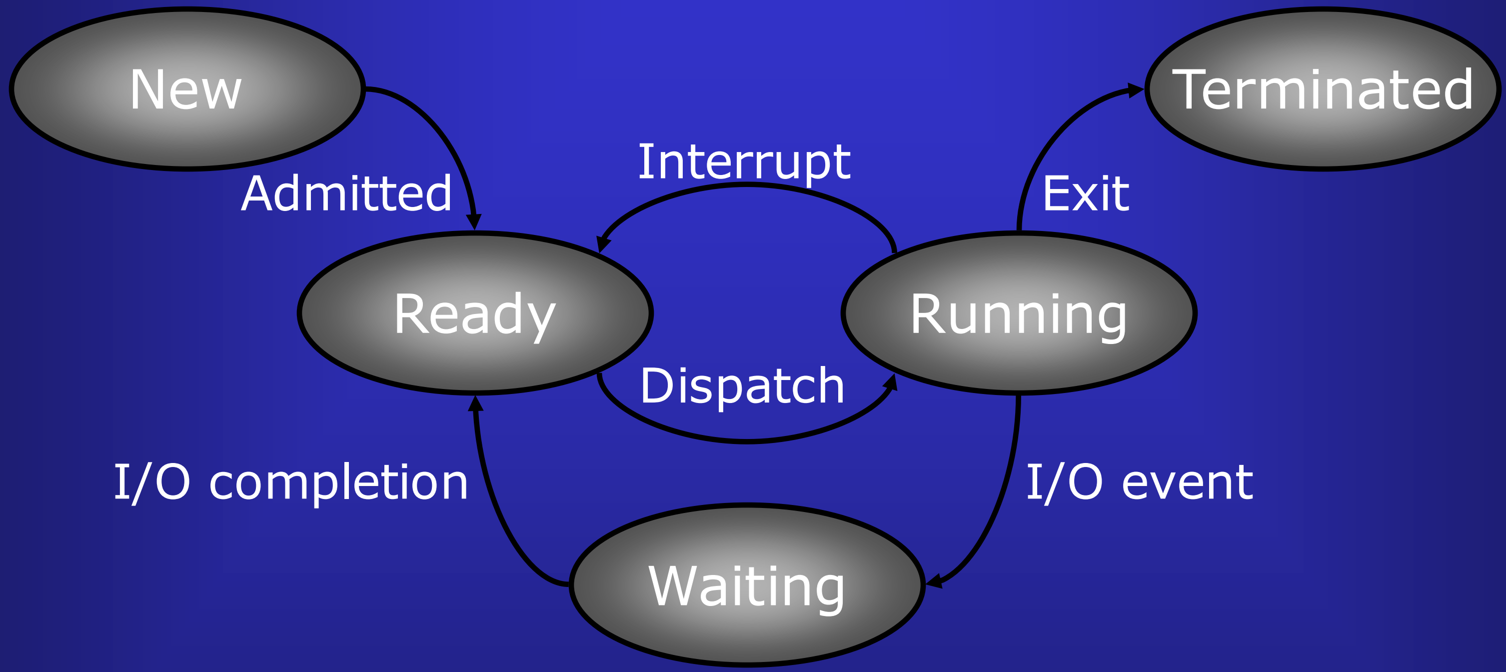
- Two concurrent instances of a text-editor
- Each uses the same program, but has different data and state information

Process Control Block



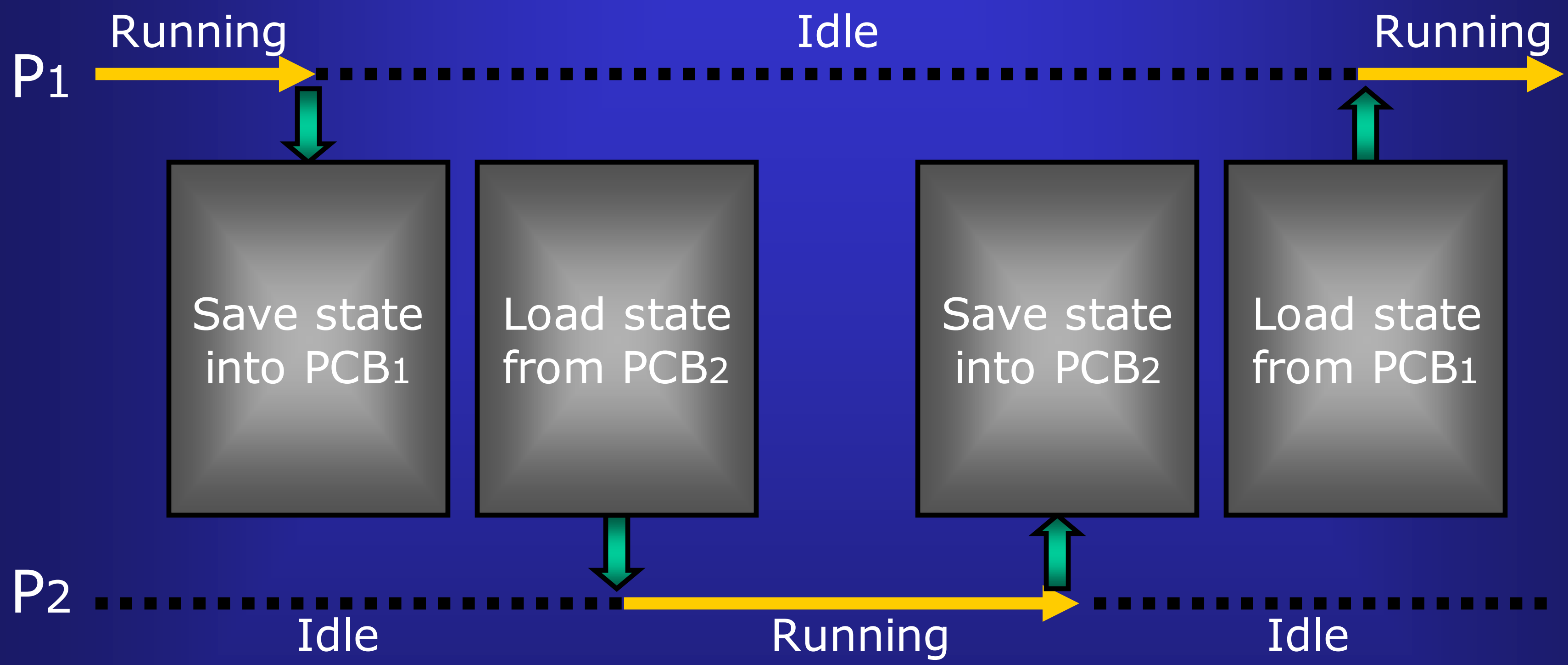
- Representation of a process in the OS
- Contains all the data about the process
  - One PCB per process

Process State



- As a process executes, it changes *state*
  - Only one *running* process per CPU
  - Many *ready* or *waiting*

Context Switching

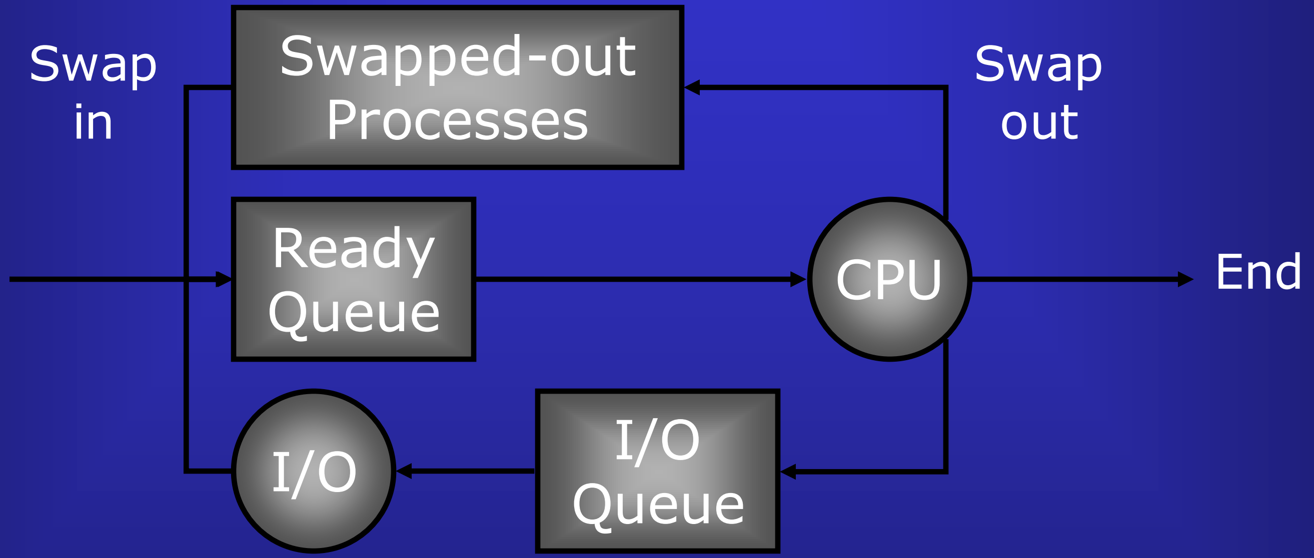


- Save state of current process
- Load state of next process
- The CPU does no useful work while switching

Threads

- Context switches take time
  - As OS becomes more complex, the amount of data associated with a process increases
- A *thread* is a lightweight process
  - One process can have many threads
  - Separate machine state
  - Share OS resources (code, files,…)
- Kernel threads (NT, Digital UNIX)
  - Run as separate entities in OS
- User threads (thread libraries)
  - Run under a single process

Swapping



- Swapped-out processes are kept on disk
- Aim to swap out *non-interactive* processes
- Improves the performance of interactive processes

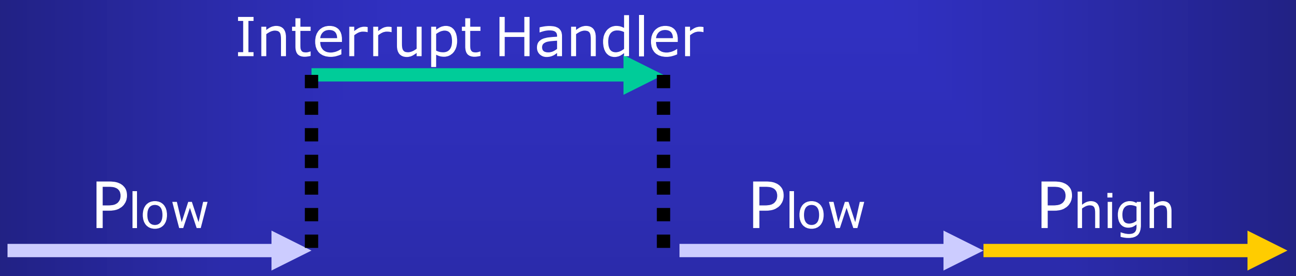
Process Scheduling

- Aim to have a process running at all times
  - Time-sharing
  - Maximise use of the CPU
- The *ready queue*
  - List of processes which are ready to run
- Processes remain in the ready queue until they are *dispatched*
- The *scheduler* decides which process is to run next
  - How it does this affects the whole system

Priority Pre-emption

- Each process is assigned a priority
- Processes run until they suspend themselves or are interrupted
- If an interrupt makes a higher-priority process ready, then run it instead

Run-To-Completion



- Process runs until it suspends itself
- After an interrupt, the same process continues to run

Interrupt Latency

- Time between an interrupt occurring and a process being run in response to that interrupt
- Interrupt Latency= Interrupt lock-out time +
  Time to service interrupt +
  Time to schedule process
- Advantages of priority pre-emption
  - Interrupt latency is constant
  - Can have low-priority processes which do not interfere with response times

Interrupt Latency with Natural Break

Ready queue = P_1 P_2

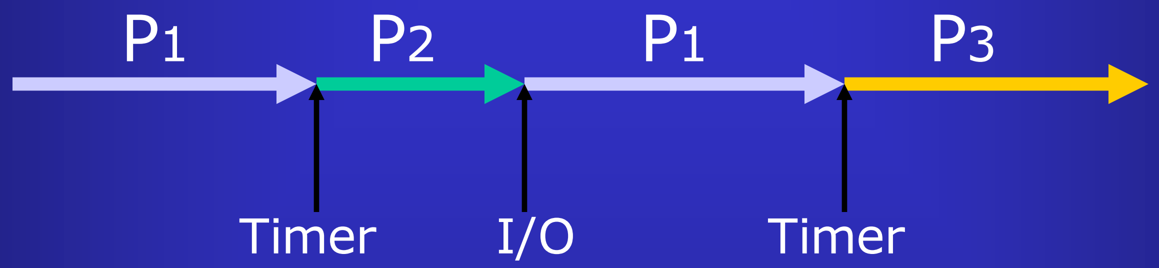
Interrupt puts P_3 onto ready queue

Ready queue = P_1 P_2 P_3

P_1 runs for 2 msec, P_2 runs for 3 msec
Interrupt latency with natural break = 5 msec

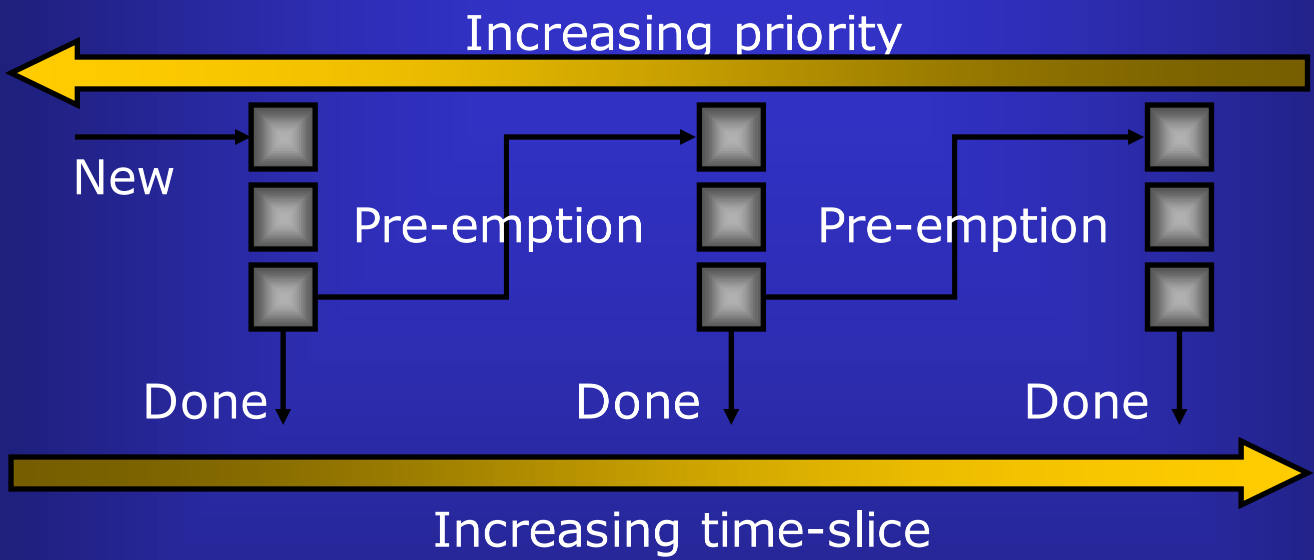
- Advantages
  - Efficient implementation of mutual exclusion
- Disadvantages
  - User processes must be well-behaved
  - Response time depends on user programs

Time-Sliced Scheduling



- Process runs until its *time-slice* expires or until it suspends itself
- Ensures that CPU is shared fairly between users
- Interrupt latency is the size of the time-slice

Multi-Level Queues



- Dynamic priority assignment
- Favours interactive jobs
  - Get a good response time
- Used in UNIX

Summary

- The fundamental unit of computation is a *process*
- Managing processes one of the most important jobs of an operating system
- The operating system uses a scheduling strategy to decide which process to run when

Types of Process Interaction

- Sharing
  - Two processes sharing an object
  - Require *mutually exclusive* access to prevent interference
- Synchronisation
  - One process triggering another
- Communication
  - One process passing a message to another
- These are all closely related