Operating Systems

David Vernon

Course Overview

- Key objective is to introduce the concept of operating systems (OS)
  - The need for operating systems
  - Understand design choices, design tradeoffs, and implications
  - Develop a knowledge of how real operating systems work

Course Content

- Historical overview (OS in context)
- Structure of OS
  - basic components
  - interfaces to hardware (HW)
- Process Management
  - processes, threads, concurrency, multi-tasking, and scheduling

Course Content

- Memory Management
  - virtual memory
  - memory protection and sharing
  - integrated and separated I/O

Course Content

- Device management
  - I/O devices and interfaces
  - polling and interrupts
  - errors
  - system calls
  - memory management

Course Content

- File management
  - filing systems
  - interface and implementation
  - file and directory structure
  - access control
  - physical space management
  - backup and archiving

Course Content

- Networking issues
  - design
  - topology
  - protocols
  - performance

Laboratory Experiment & Assignments

- Process manipulation facilities in Unix
- Process coordination and resource sharing
- Device drivers
- Assignment: implementation of resource allocation

Course Textbook

- “Modern Operating Systems”
  Andrew S. Tanenbaum
  Prentice Hall 1992
  ISBN 0-135-881870

Supplementary Reading

- “Operating Systems Concepts”
  A. Silberschatz and P.B. Galvin
  Addison Wesley 1997
  ISBN 0-201-591138

Assessment of Performance

- First examination (1 hour) 15%
- Second examination (1 hour) 15%
- Assignment/Laboratory work 20%
- Final examination (2 hours) 50%

Historical Overview

- Two goals for this section:
  - Why we need operating systems and what they are
  - How operating systems have evolved

The Need for Operating Systems

- Why do we need operating systems?
  - To shield programmers and users from the ‘bare’ machine

The Need for Operating Systems

Components of a Computer System

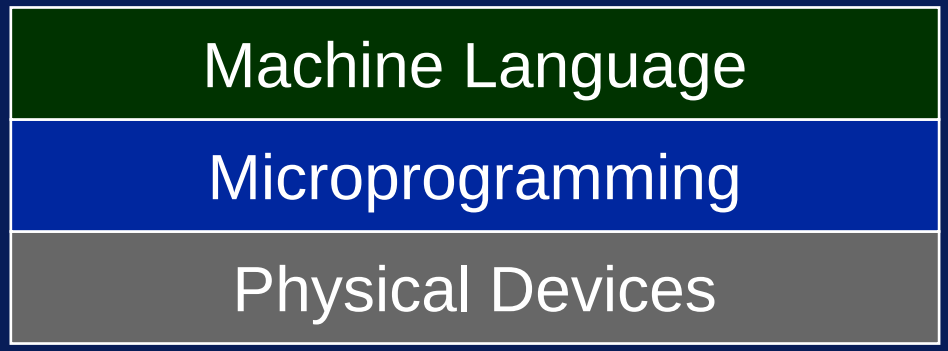
Physical Devices

The Need for Operating Systems

Components of a Computer System

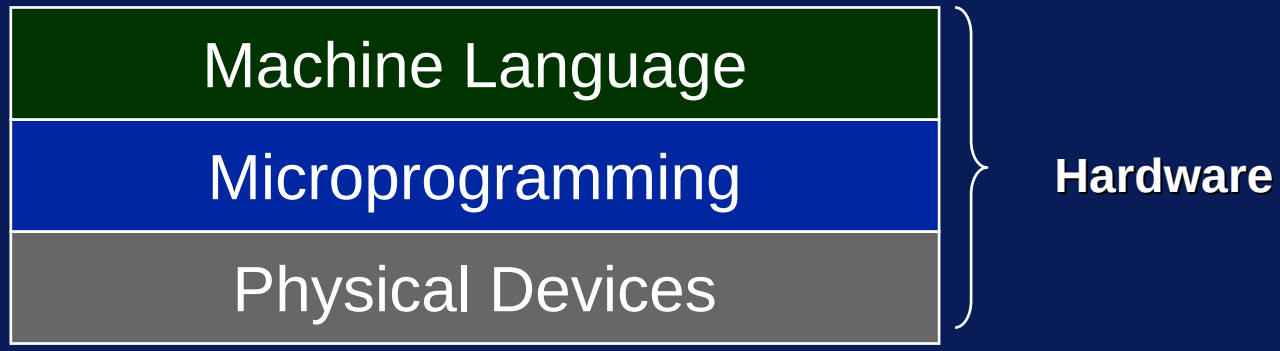
The Need for Operating Systems

Components of a Computer System



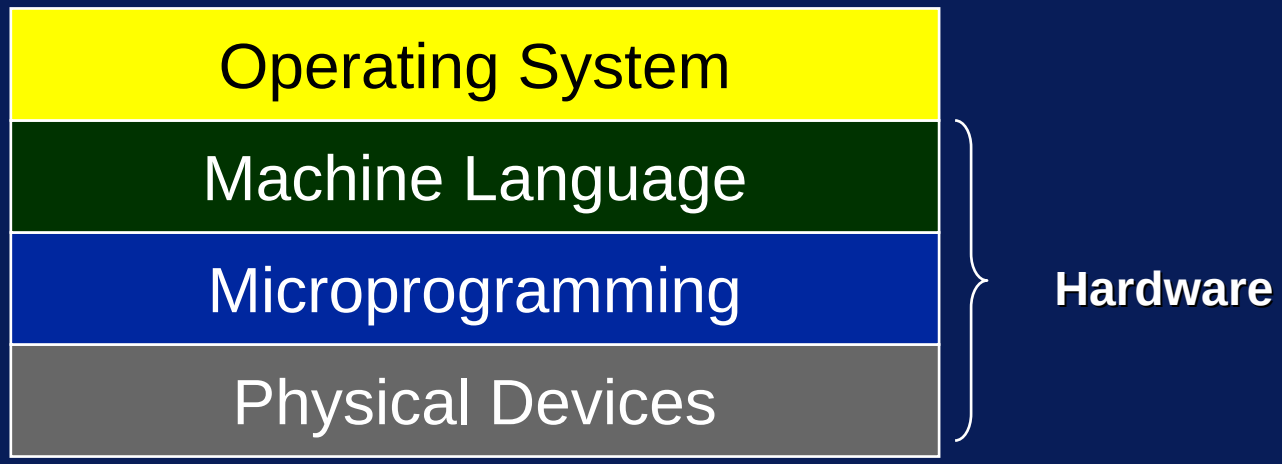
The Need for Operating Systems

Components of a Computer System



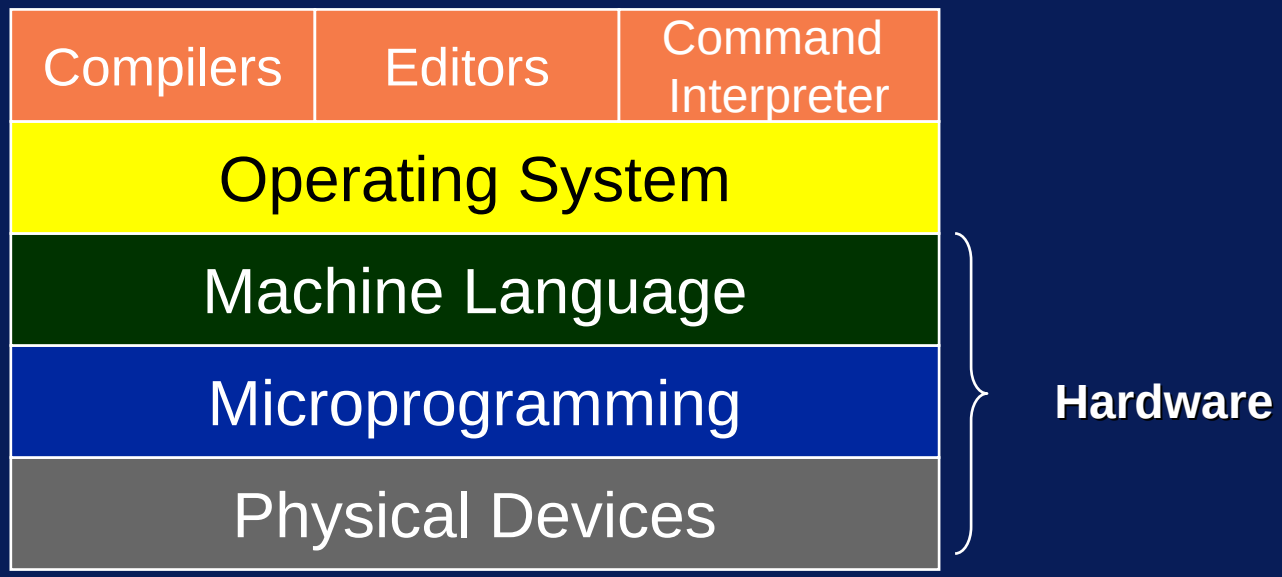
The Need for Operating Systems

Components of a Computer System



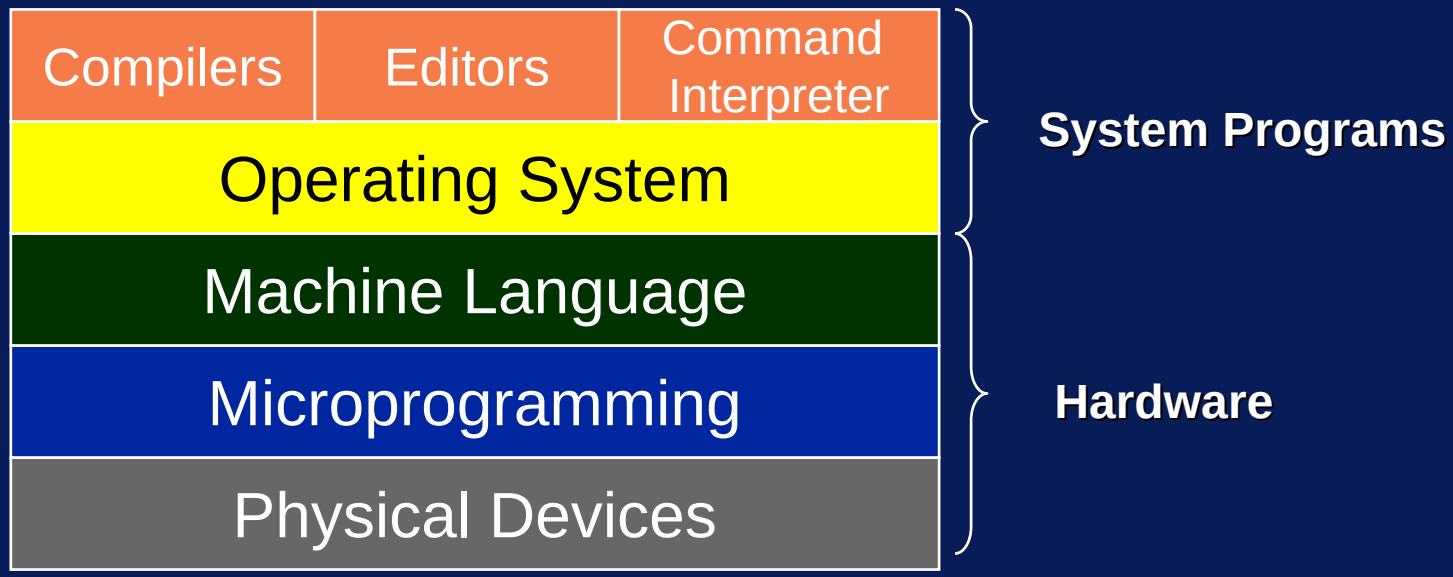
The Need for Operating Systems

Components of a Computer System



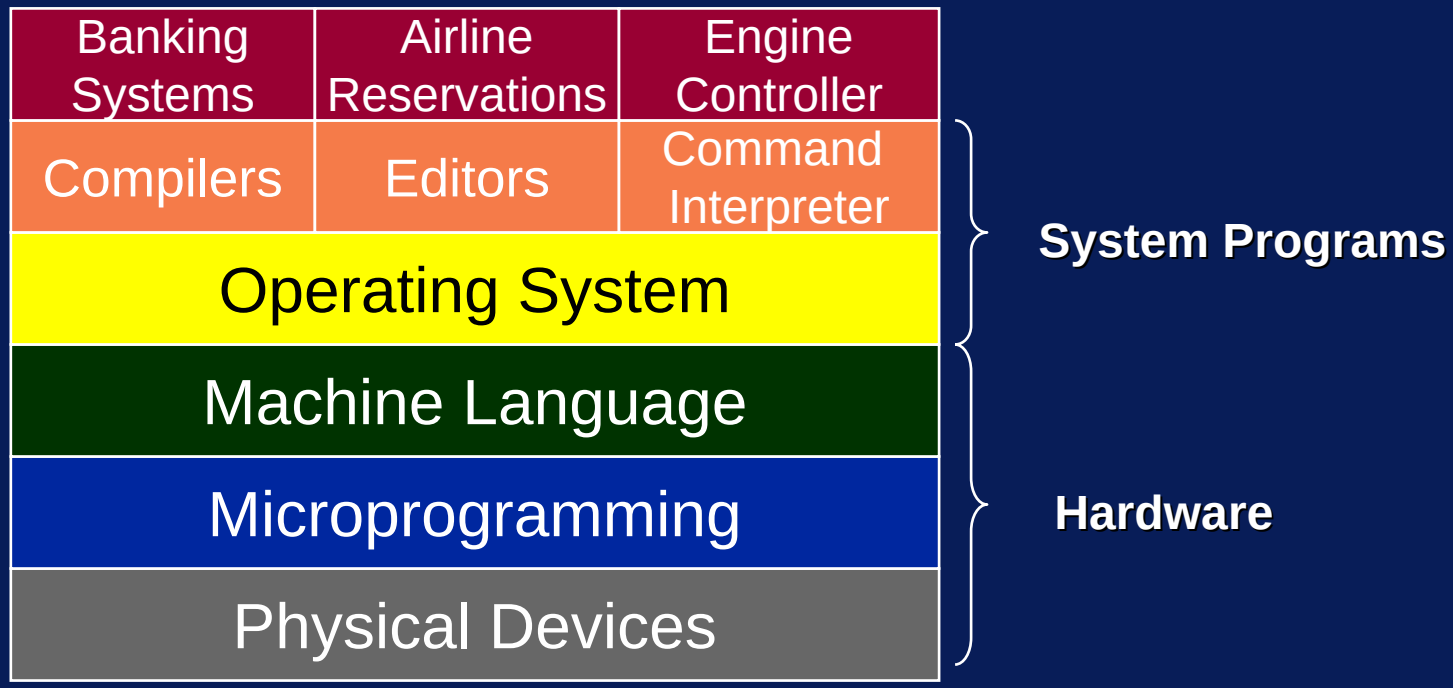
The Need for Operating Systems

Components of a Computer System



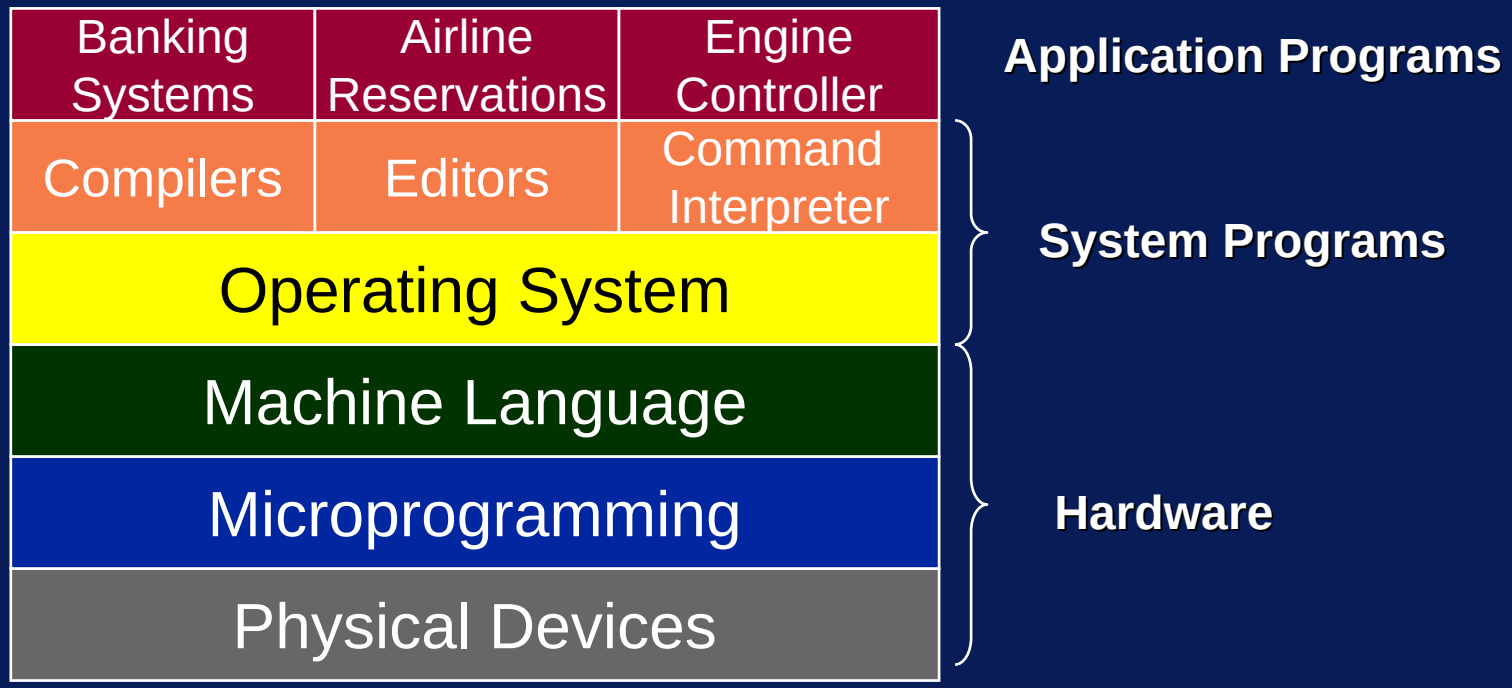
The Need for Operating Systems

Components of a Computer System



The Need for Operating Systems

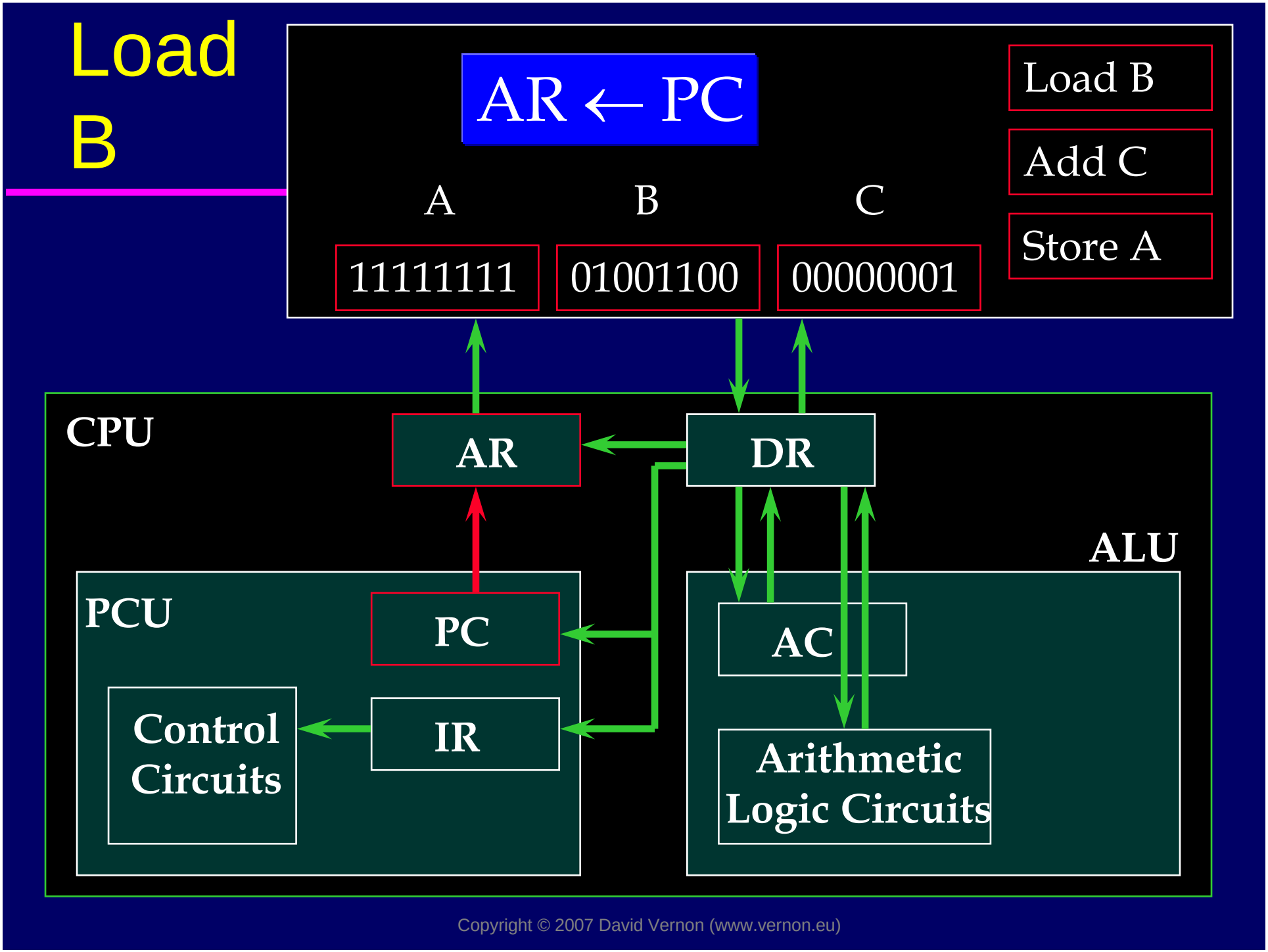
Components of a Computer System



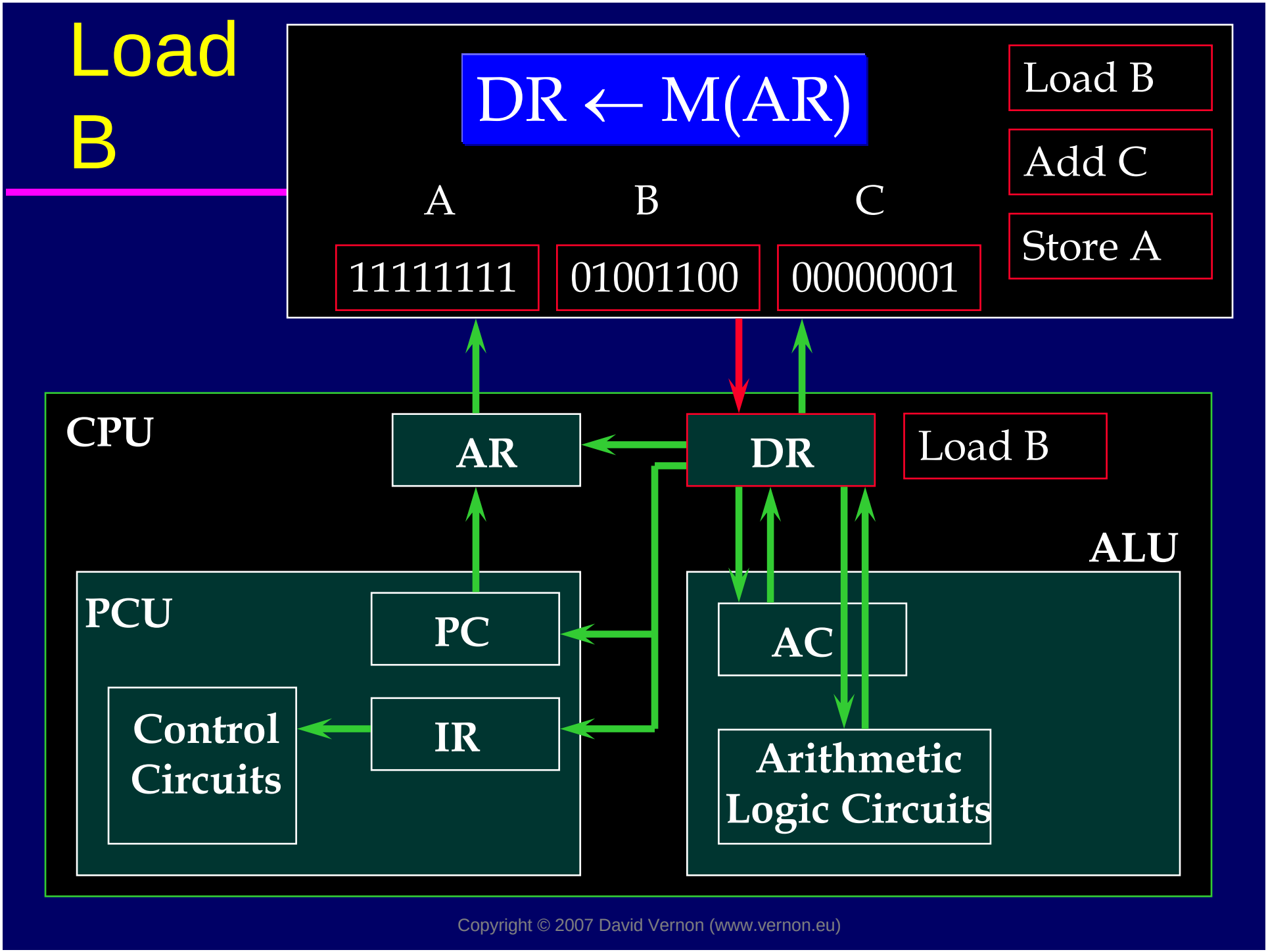
The Need for Operating Systems

- Example of how a ‘bare’ machine would transfer data from memory to the processor

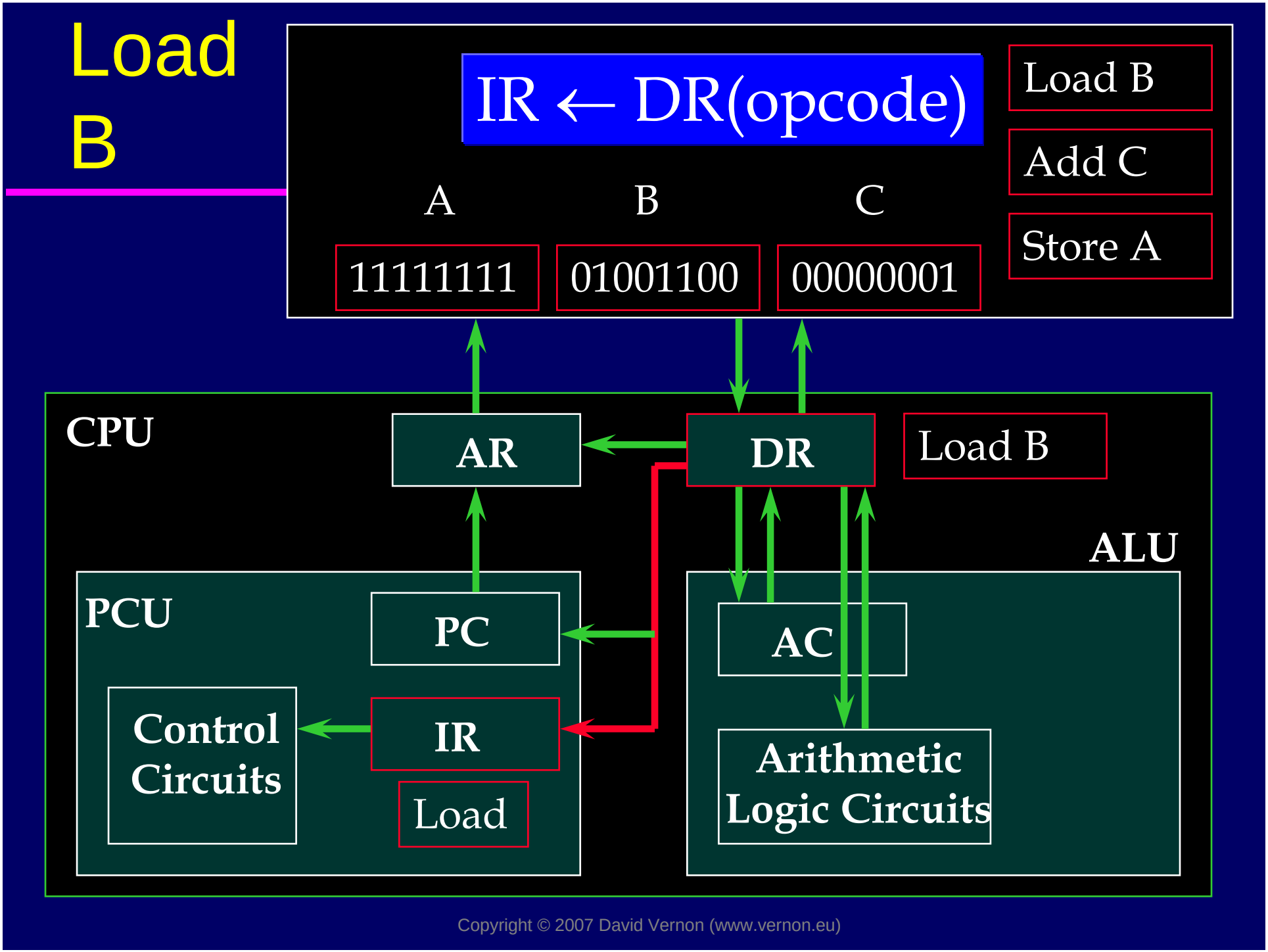
Load B



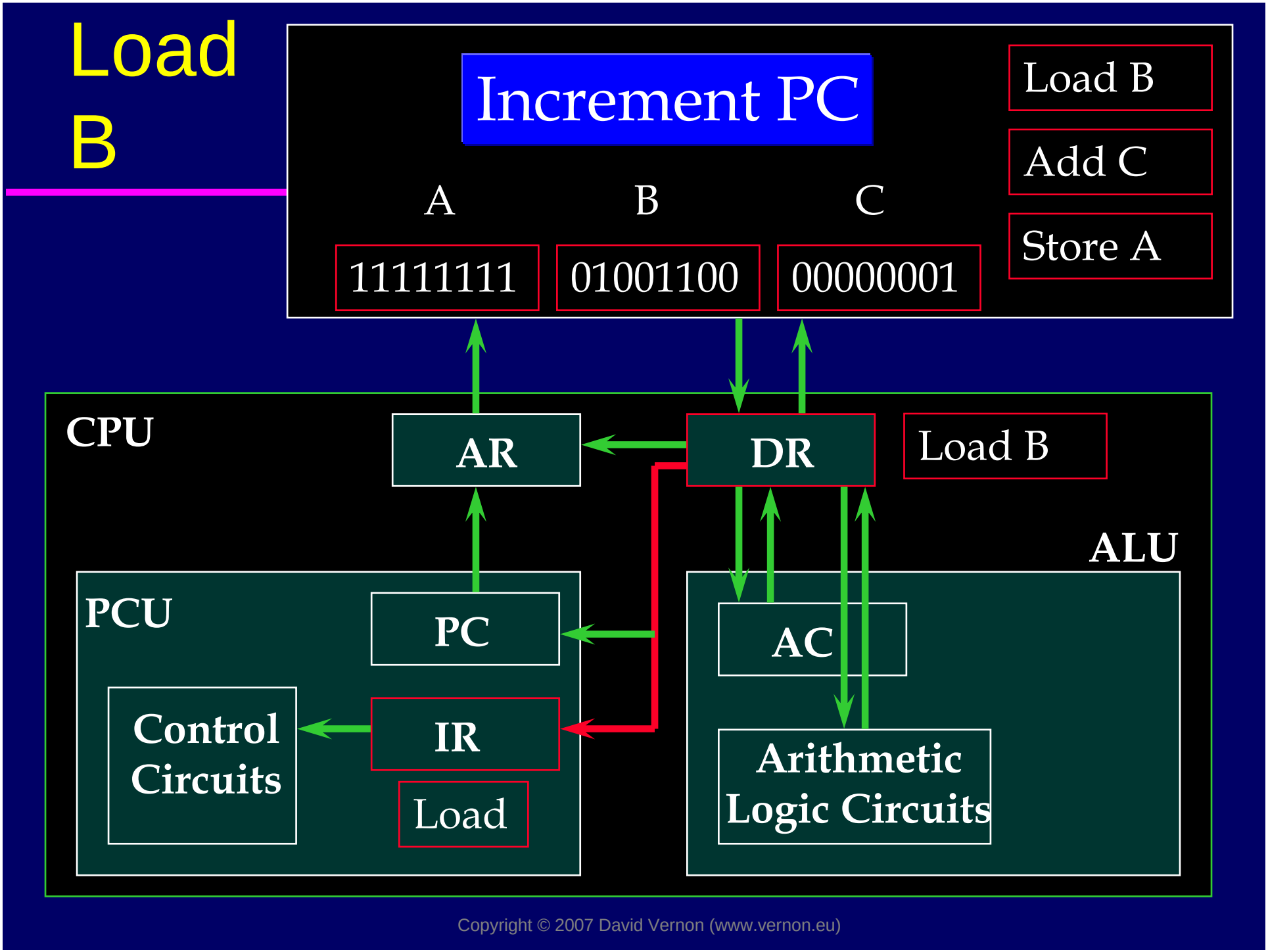
Load B



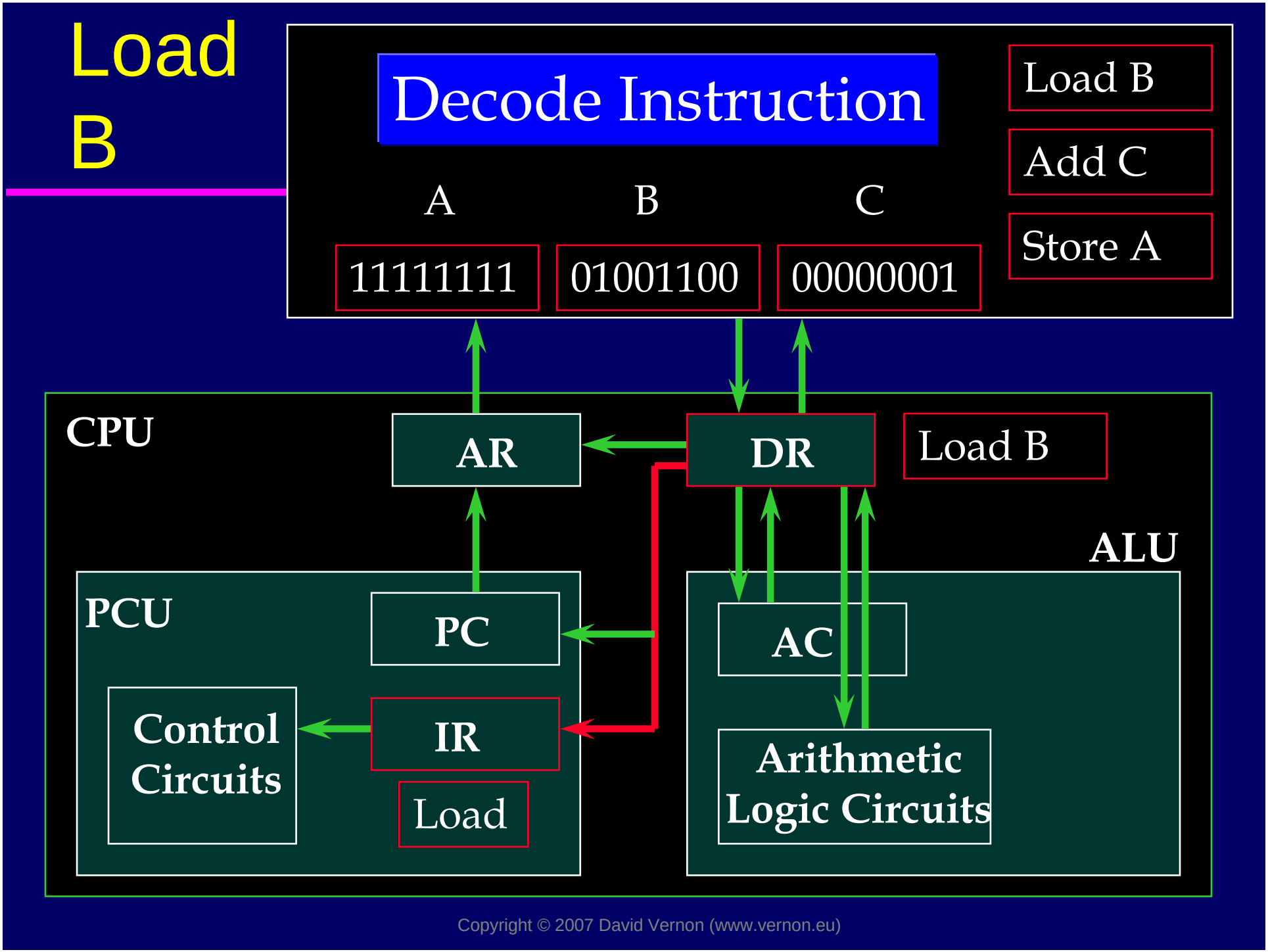
Load B



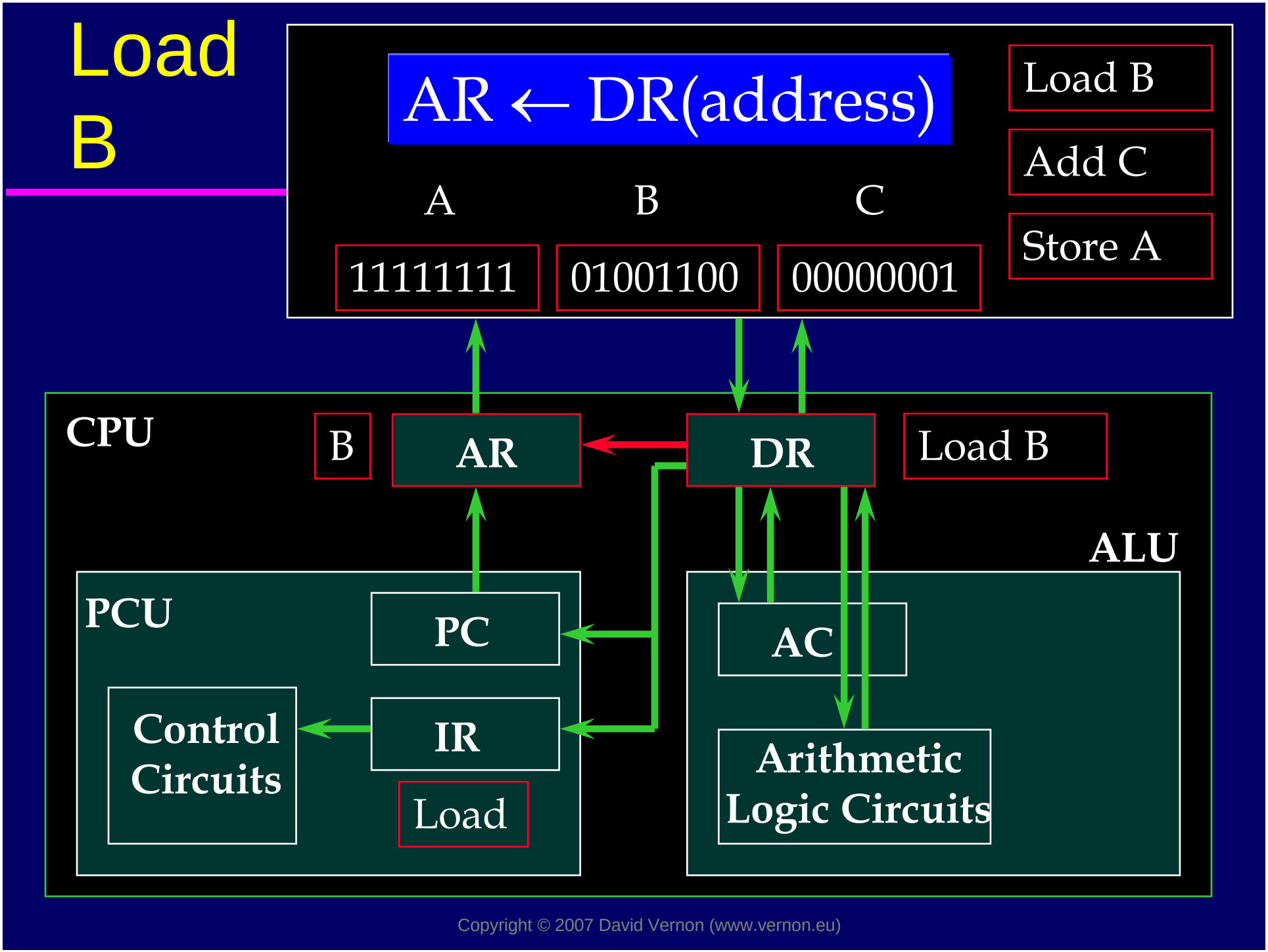
Load B



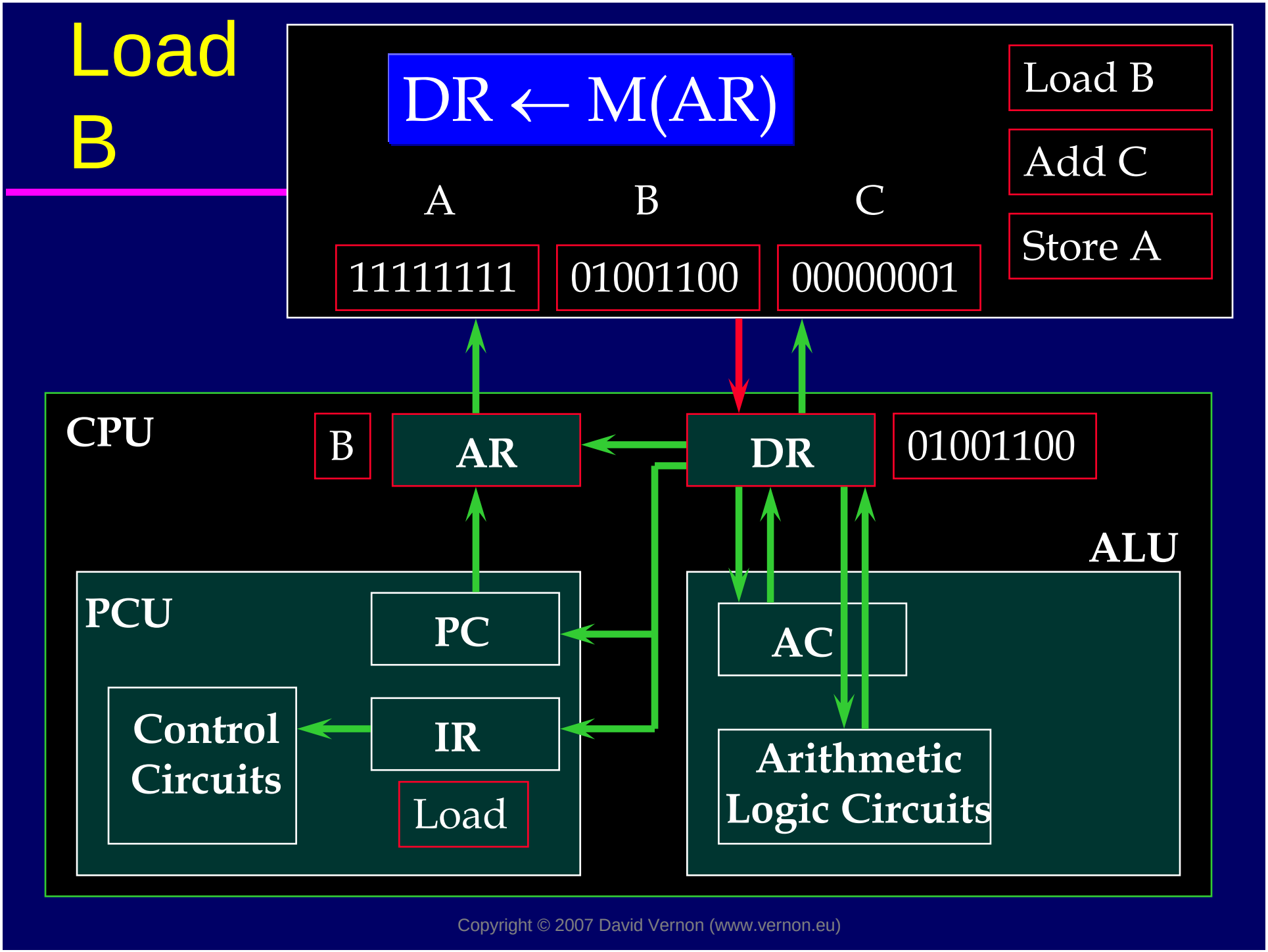
Load B



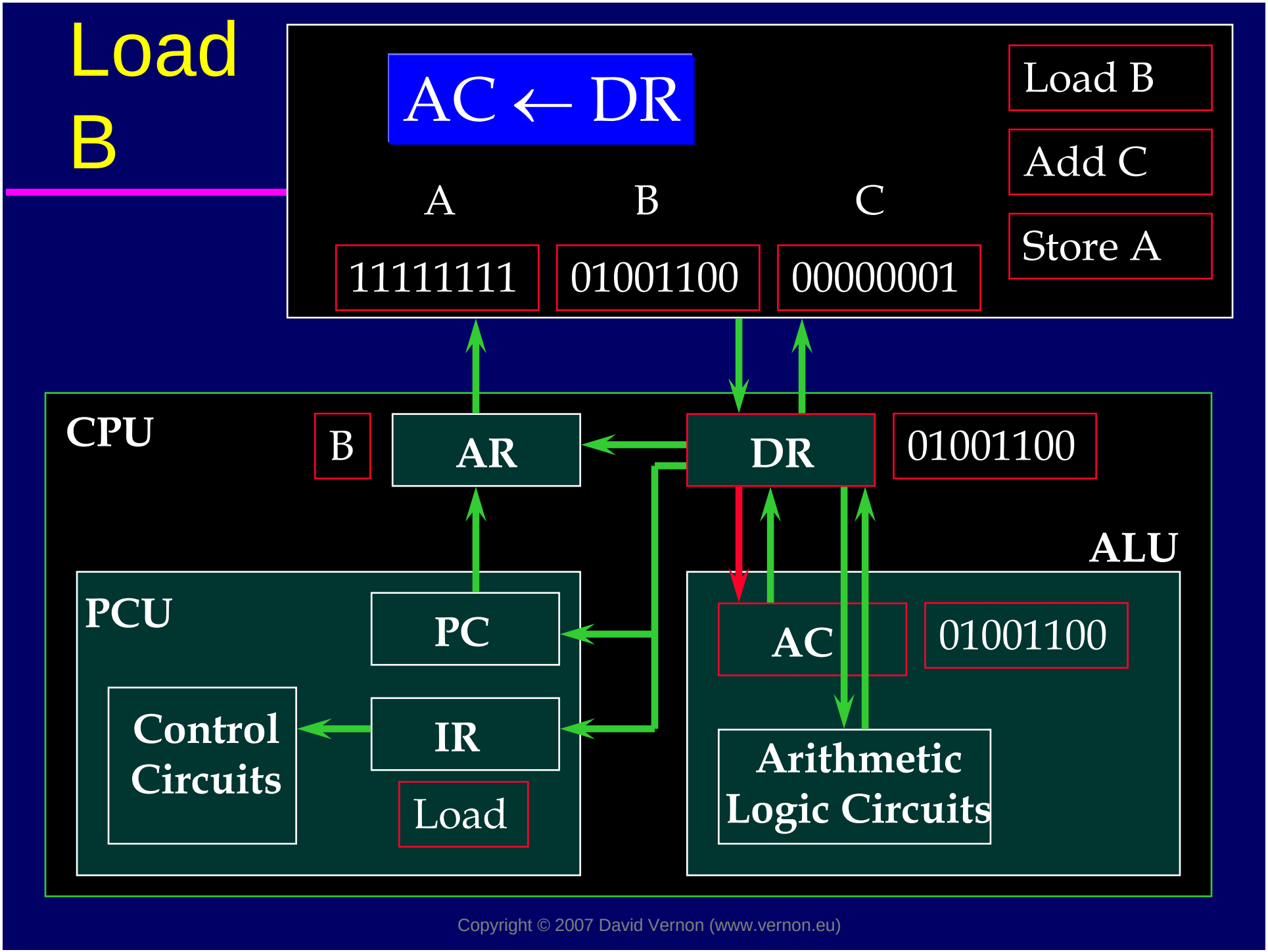
Load B



Load B



Load B



The Need for Operating Systems

- Physical devices
  - integrated circuits (processor, registers, memory, disks, displays, keyboards, modems, …)
- Microprogramming
  - ‘register transfer’ instructions
- Machine Language
  - ADD, MOVE, JUMP, … instructions
    - » ADD etc are assembly language instructions
    - » Their binary equivalent is machine language

The Need for Operating Systems

- Machine (assembly) Language
  - between 50 and 300 instructions
  - I/O is effected by moving values to ‘device registers’
    - » e.g. read a disk by loading address, byte count, read/write flag, memory address in to the disk registers
    - » but many many parameters are needed
    - » timing is an important issue

The Need for Operating Systems

- The operating system hides this complexity from the programmer or user
  - by providing macroscopic (and more abstract) instruction
    - » System Calls

The Need for Operating Systems

- Compilers, Editors, Command Interpreters (the shell)
  - are system programs
  - are NOT part of the operating system
- Why?

The Need for Operating Systems

- Operating System runs in ‘kernel’ mode
  - also called supervisor mode
  - it has full access to all the hardware devices
- Other system software and user applications run in ‘user’ mode
  - unable to access hardware devices directly

What is an Operating System

- There are two views of Operating Systems
  - OS as an extended machine
  - OS as a resource manager

What is an Operating System

- OS as an extended machine
  - provide a simpler interface to the hardware
  - provide a more abstract interface
  - the abstraction makes the software more machine independent
  - the user sees
    - » an extended machine
    - » a virtual machine

What is an Operating System

- OS as a resource manager
  - manage and coordinate:
    - » processor
    - » memory
    - » timers
    - » disks
    - » terminals
  - provide orderly and controlled allocation of processor, memory, I/O devices to programs competing for them

What is an Operating System

- OS as a resource manager
  - keep track of who is using what resource
  - grant resource requests
  - account for usage
  - mediate conflicting requests for the same resources from different programs and users

History of Operating Systems

- First generation computers (1945-55)
  - vacuum tubes and plug-boards
  - all programming was done in machine language (not assembly language)
  - No programming languages
  - No operating systems

History of Operating Systems

- Second generation computers (1955-65)
  - Transistors and batch systems
  - program in FORTRAN or assembly
  - punch program on cards
  - hand to operator
  - operator runs a batch of jobs
  - returns results to user

History of Operating Systems

- Second generation computers (1955-65)
  - Sometimes the batch of cards (jobs) was transferred to tape (using a cheaper computer)
  - main computer read the tape and produced results (on tape)
  - another computer produced the final result for the users
  - each job had to use control cards

History of Operating Systems

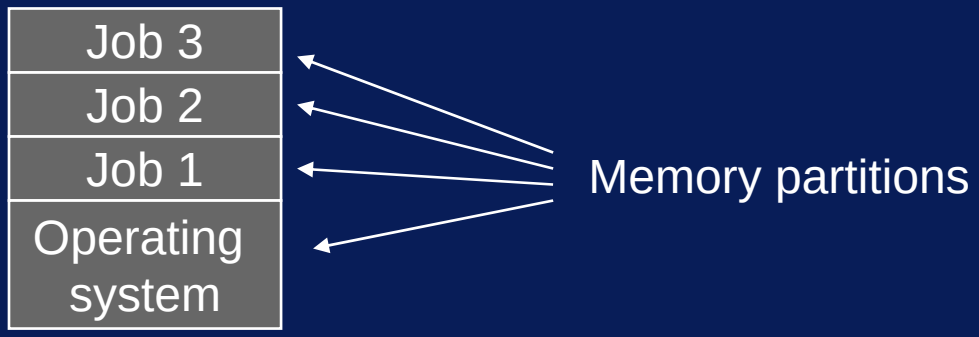
- Second generation computers (1955-65)
  - very primitive OS was used to interpret the control cards

History of Operating Systems

- Third generation computers (1965-80)
  - Family of computers
    - » different facilities (I/O, computing power, memory)
    - » unified single operating system (again, providing abstraction away from the hardware)
    - » IBM 360
      - millions of lines of assembly language code
      - written by thousands of programmers
      - and incorporating thousands of bugs!

History of Operating Systems

- Third generation computers (1965-80)
  - Capable of running many jobs ‘at once’
  - multiprogramming
  - minimise the idle time cause by slow I/O component of a typical job



History of Operating Systems

- Third generation computers (1965-80)
  - Could read all jobs and store on disk
  - and read from disk whenever a running job finished
  - again maximising the usage of the computer
  - Spooling (Simultaneous Peripheral Operation On Line)

History of Operating Systems

- Third generation computers (1965-80)
  - Could allow many users to access the computer simultaneously
  - timesharing

History of Operating Systems

- Fourth generation computers (1980-90)
  - LSI (Large Scale Integration) integrated circuits
  - Personal Computers
  - Workstations
  - Networking
  - Operating systems:
    - » MS-DOS (PC)
    - » Unix (workstation)

History of Operating Systems

- Fourth generation computers (1980-90)
  - Network operating systems
    - » explicit access to multiple computers
    - » explicit access to files on other computers
  - Distributed operating systems
    - » appears to the user as a single machine operating system
    - » even though it is running on many machines
    - » the problems of coordination amongst the differents computers is hidden

Operating System Concepts

- System calls
  - ‘extended instructions’ provided by the OS
  - to hide the bare machine and provide an interface between user programs and OS

Operating System Concepts

- Processes
  - a program in execution, comprising
    - » executable program
    - » program's data stack
    - » program counter
    - » stack pointer
    - » contents of all registers
    - » any other information needed to run the program

Operating System Concepts

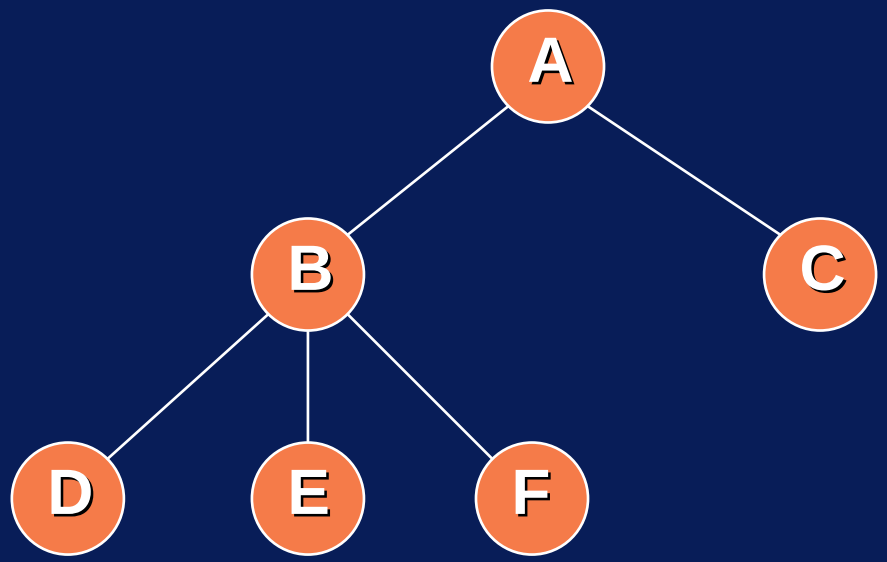
- Processes
  - All this is needed so that a process can be
    - » suspended
    - » resume
  - All the information (apart from the contents of its address space) is stored in a process table

Operating System Concepts

- Processes
  - a suspended process comprises
    - » core image
    - » process table entry
  - Key process management system calls:
    - » creation of processes
    - » termination of processes
    - » allocate/deallocate memory

Operating System Concepts

- Processes
  - Processes can create other processes
    - » child processes



A Process Tree

Operating System Concepts

- Processes
  - Signal
    - » a way of allowing one process to communicate with another process
    - » In particular, if the other process is not expecting the communication
    - » A signal causes the process to suspend its operation (saving it's registers) and start running a special signal handling procedure
    - » Once the signal handler is complete, the process resumes

Operating System Concepts

- Processes
  - Signal
    - » Just like a software interrupt
    - » Hardware traps, e.g. illegal instruction, invalid address, are often converted to signals to the offending process

Operating System Concepts

- Processes
  - UID
    - » User Identification
    - » used in multiprogramming (multiuser) OS to allow a process to be associated with a given user
    - » 16 or 32 bit integer
    - » often used to implement privacy (exclusion/inclusion of users in a group)
    - » there are also GIDs (Group Identication)

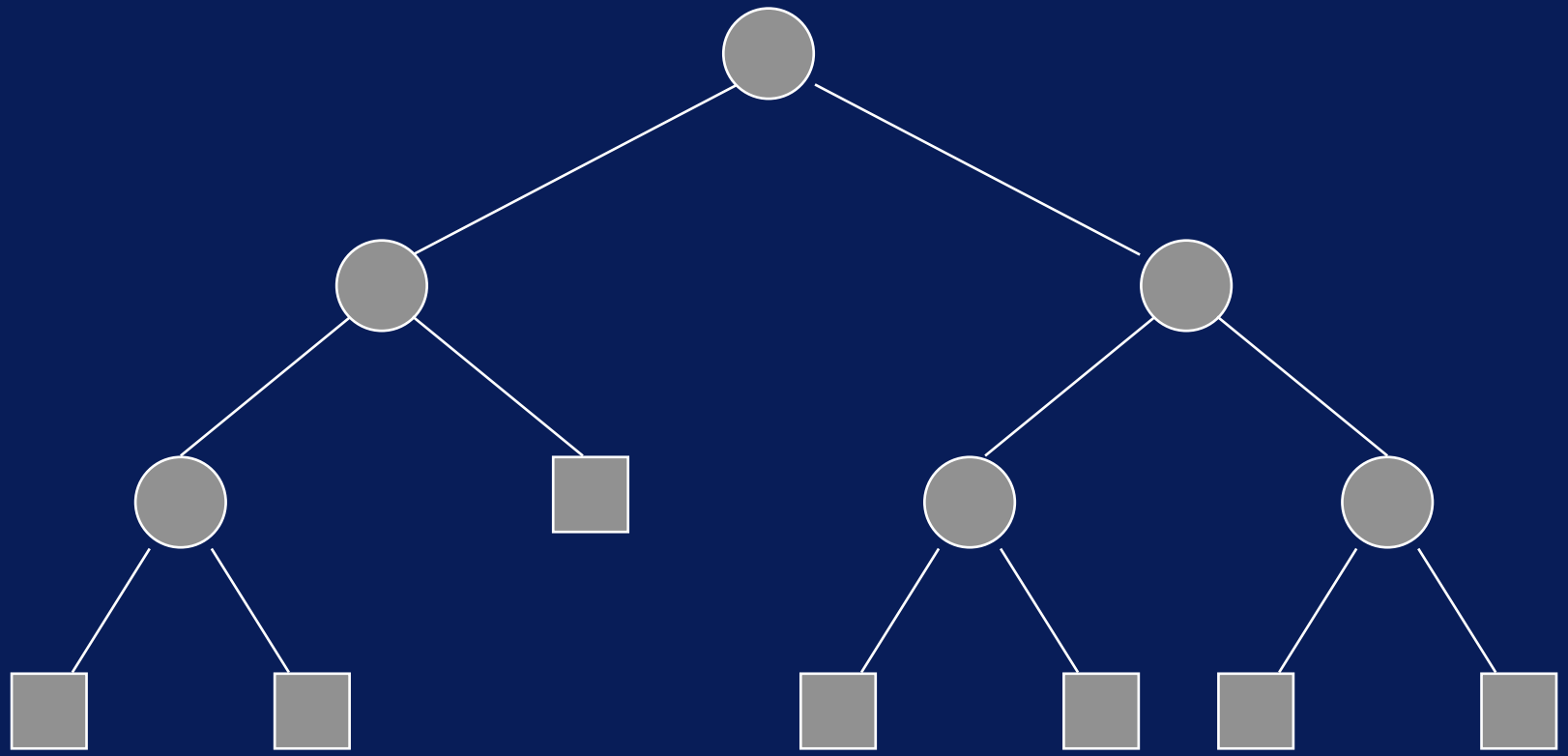
Operating System Concepts

- Files
  - OS provides a device-independent abstract model of files
  - System calls provide this mechanism
  - Also allow logical organization of files in a hierarchy of directories

Operating System Concepts

- Files
  - System calls are needed to create and remove directories and files
  - The hierarchy of directories and files is called the file system
  - The route from the root of the hierarchy (tree) to the required file or directory is called the path name (/usr/home/dvernon/*
  - The top directory is called the root directory

Operating System Concepts



Operating System Concepts

- Files
  - Every process is running in some directory (or has unqualified access to some directory)
  - This the working directory and does not have to be specified by a path name

Operating System Concepts

- Files
  - Privacy
    - » restrict access to file and directories
    - » Unix uses a 9-bit binary protection code:
    - » 3 bits identify the read-write-execute permissions for
      - the user
      - a particular group of which he use is a member
      - system administrator
    - » These arethe RWX bits

Operating System Concepts

- Files
  - An example of the protection code:

    rwxr-x--x

  - means the owner can read, write, or execute the file; anyone the users group can read it and executed it; anyone else can only execute it.

Operating System Concepts

- Files
  - On opening a file before reading or writing, The system returns a integer called a file handler or handle
    - » which is then referenced in further system calls (until the file is closed).
- Devices are dealt with in the same was as files
  - block file (randomly addressable)
  - character file (streams of characters)

Operating System Concepts

- Files
  - Standard input file (normally the keyboard)
  - Standard output file (normally the screen)
  - Standard error file (normally the screen)
  - Pipes
    - » a pseudo-file used to connect two processes

Operating System Concepts

- System Calls
  - the interface between the operating system and the user
  - extended instructions

Operating System Concepts

- System Calls
  - For each system call there is a corresponding library procedure which can be called by the user program
    - » put parameters in a specified place
    - » issue a TRAP instruction
  - The procedure hides the detail of the TRAP and looks like any other procedure

Operating System Concepts

- The Shell
  - OS carries out system calls
  - The shell is the user-interface program for the OS
    - » but it's not part of the OS

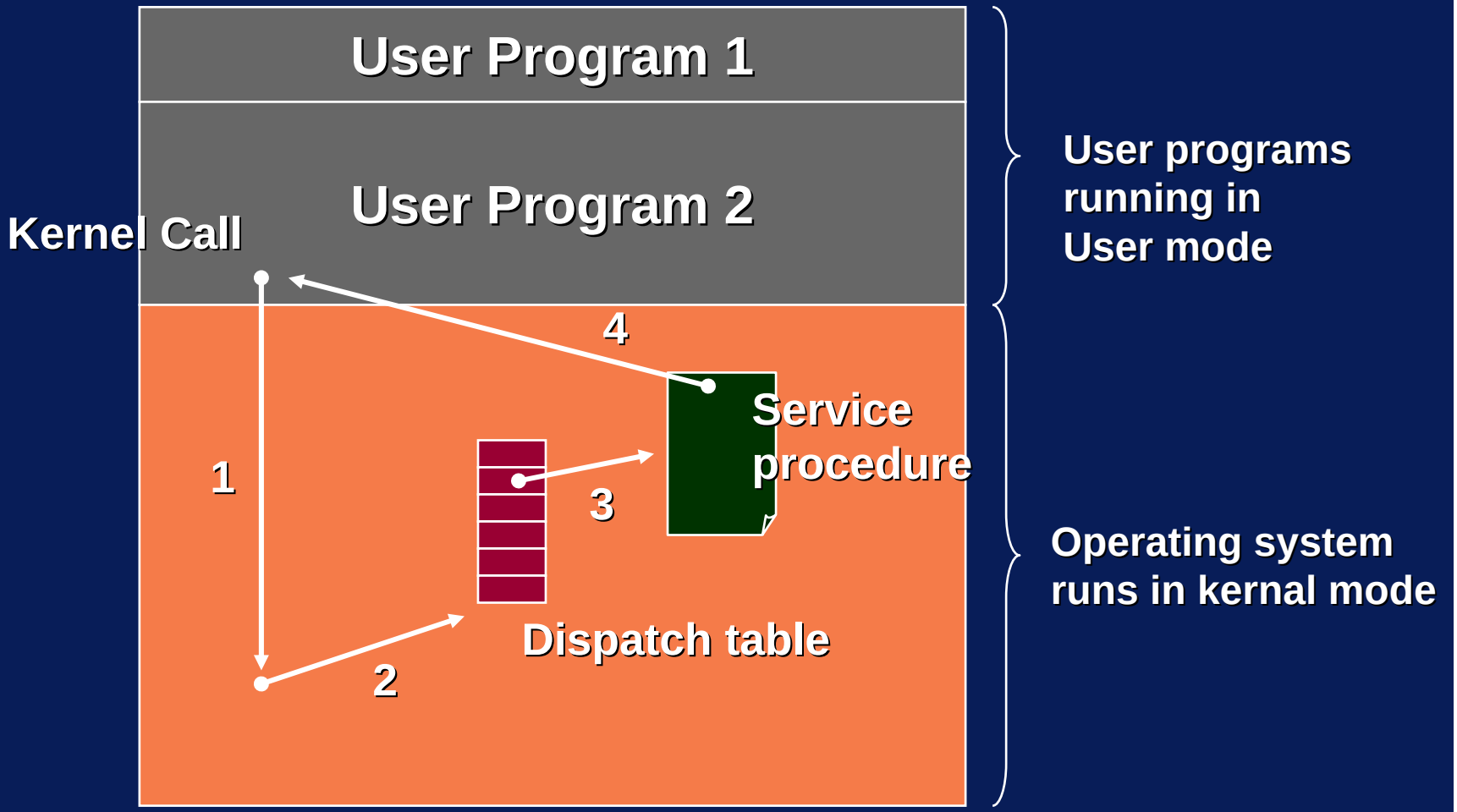
Operating System Structure

- We will look briefly a four ways of structuring an operating system
  - Monolithic system
  - Layered system
  - Virtual machines
  - Client-server models

Operating System Structure

- Monolithic system
  - NO structure!
  - OS is a collection of procedures
  - All procedures are visible to each other
  - No information hiding
  - Services (system calls) are requested by
    - » putting parameters in registers or on the stack
    - » executing a kernel call or supervisor call

Operating System Structure

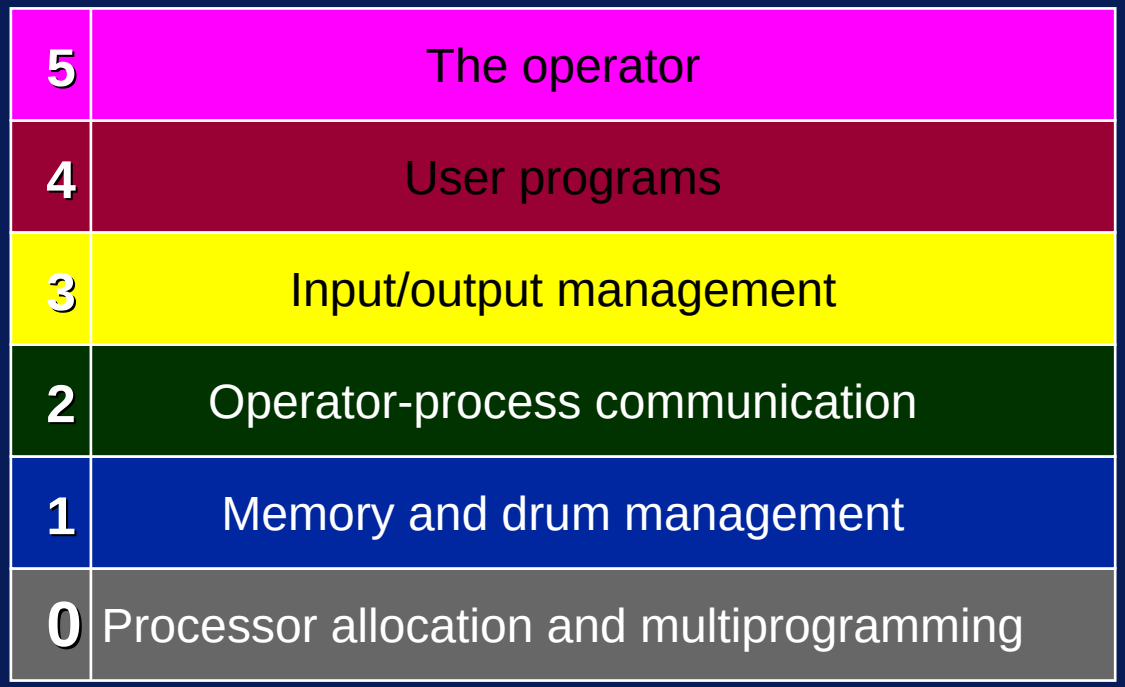


Operating System Structure

- Monolithic system
  - Main program that invokes the requested service procedure
  - Set of service procedures to carry out the system call
  - Set of utility procedures available to the service procedures

Operating System Structure

- Layered Systems



Operating System Structure

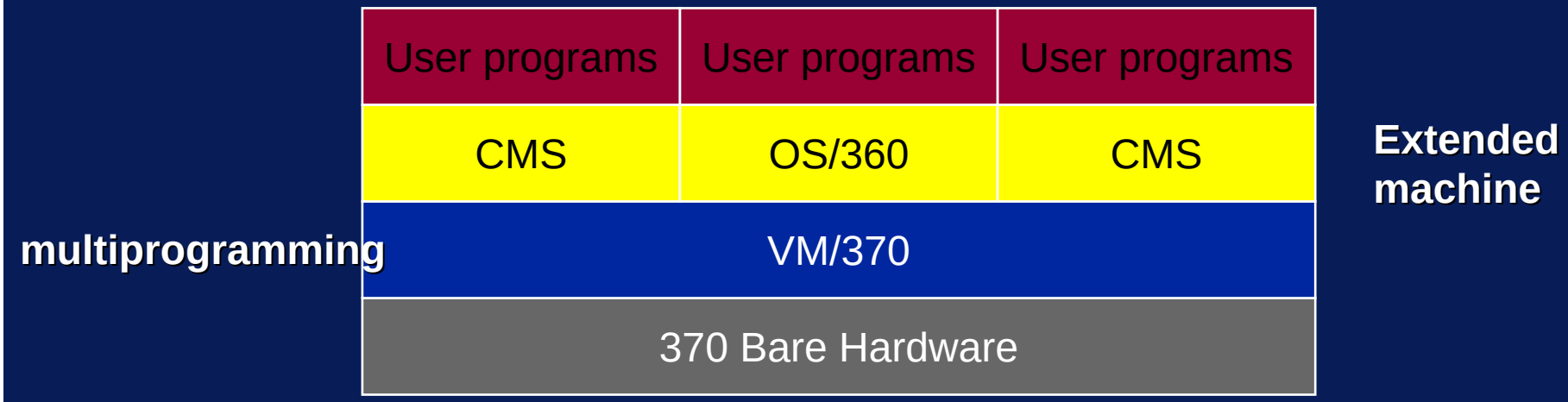
- Virtual Machines
  - Different operating systems can run on the same machine
  - But can they do so simultaneously?
  - Yes!
  - Consider IBM VM/370
    - » VM - Virtual Machine Monitor

Operating System Structure

- Virtual Machines
  - It provides a hardware emulation layer between the hardware and the many operating systems concurrently running

Operating System Structure

- Virtual Machine

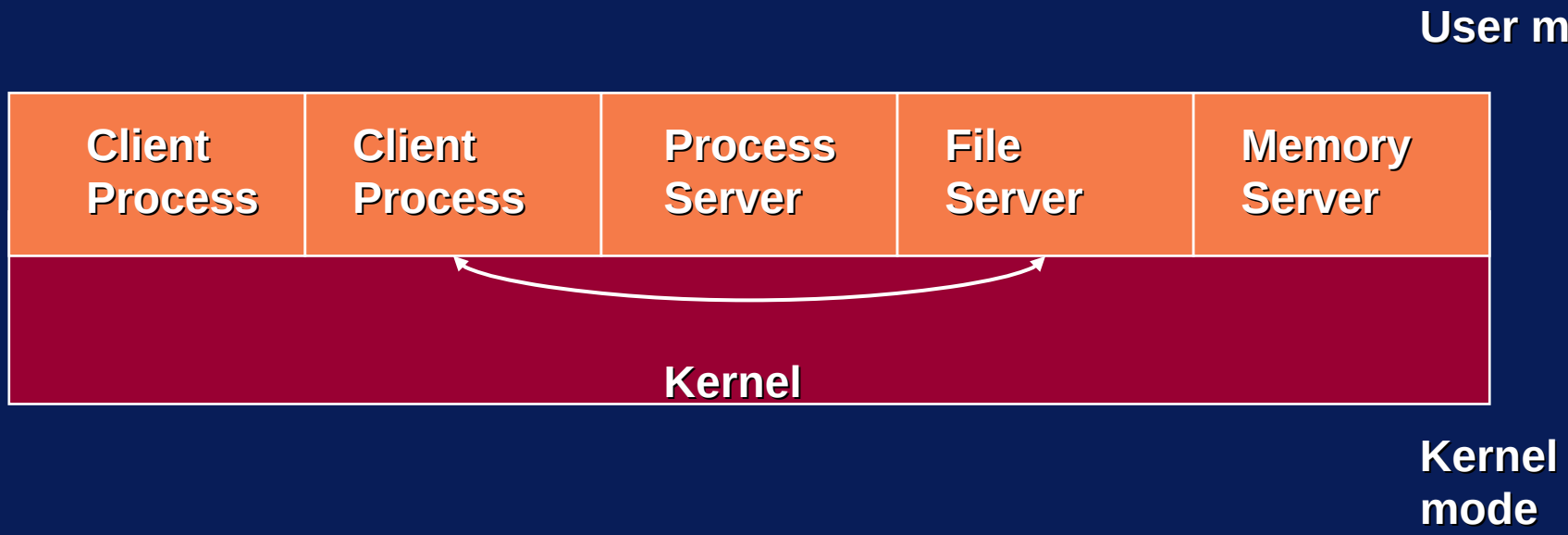


Operating System Structure

- Client-Server Model
  - User a minimalist OS called a kernel
  - Implement most of the OS functions in user processes - Client Processes
  - Clients send a request for some service to a Server process
  - The Kernel handles the communication of request and response and device I/O

Operating System Structure

- Client-Server Model

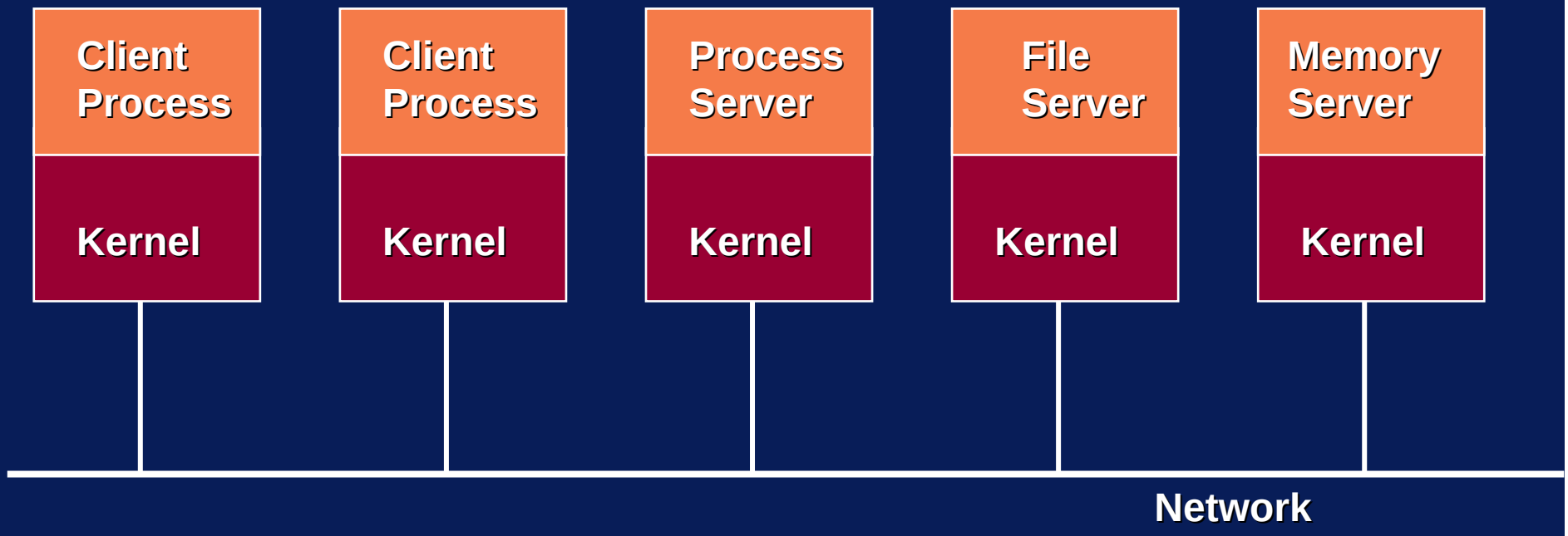


Operating System Structure

- Client-Server Model
  - Servers run in user mode
  - don’t have direct access to the hardware
  - if an error occurs, the server will crash but typically not the entire machine
  - Also facilitates distributed systems

Operating System Structure

- Client-Server Model



PROCESSES

Processes

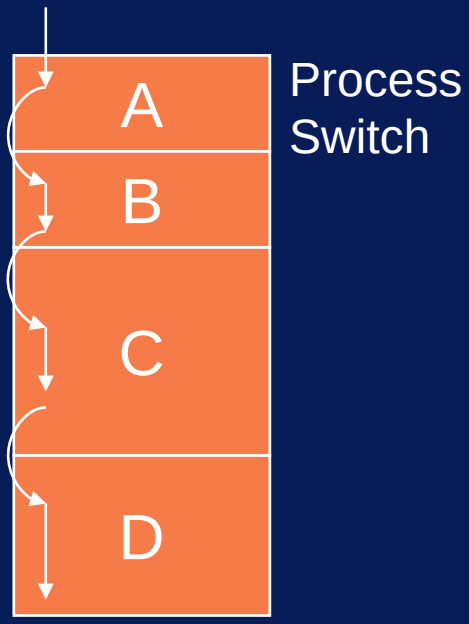
- All modern computer systems can multiplex their time between several tasks
  - » running code
  - » reading/writing disks
  - » printing
  - » network communication
  - – In a multiprogramming environment, they also multiplex between different users program
    - » This provides a form of pseudo-parallelism

Processes

- The Process Model
  - All software is organized in (sequential) processes
    - » executable code, stack, PC, registers, variables, …
  - Each process has its own virtual CPU
  - At any one instant, only one process is actually running (in a single CPU computer)
  - The CPU switches between processes, according to some scheduling algorithm

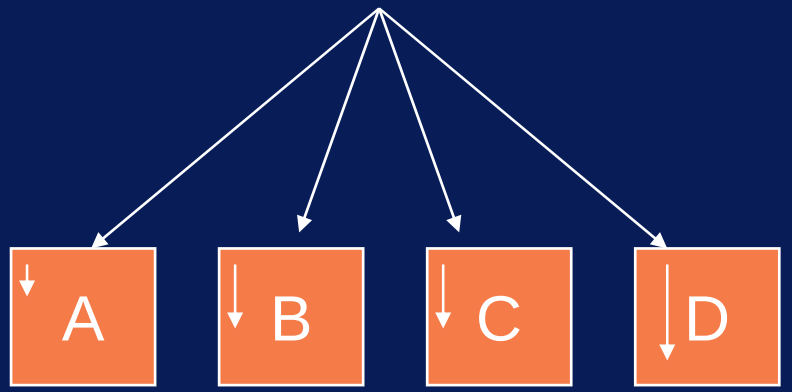
Processes

Single program counter



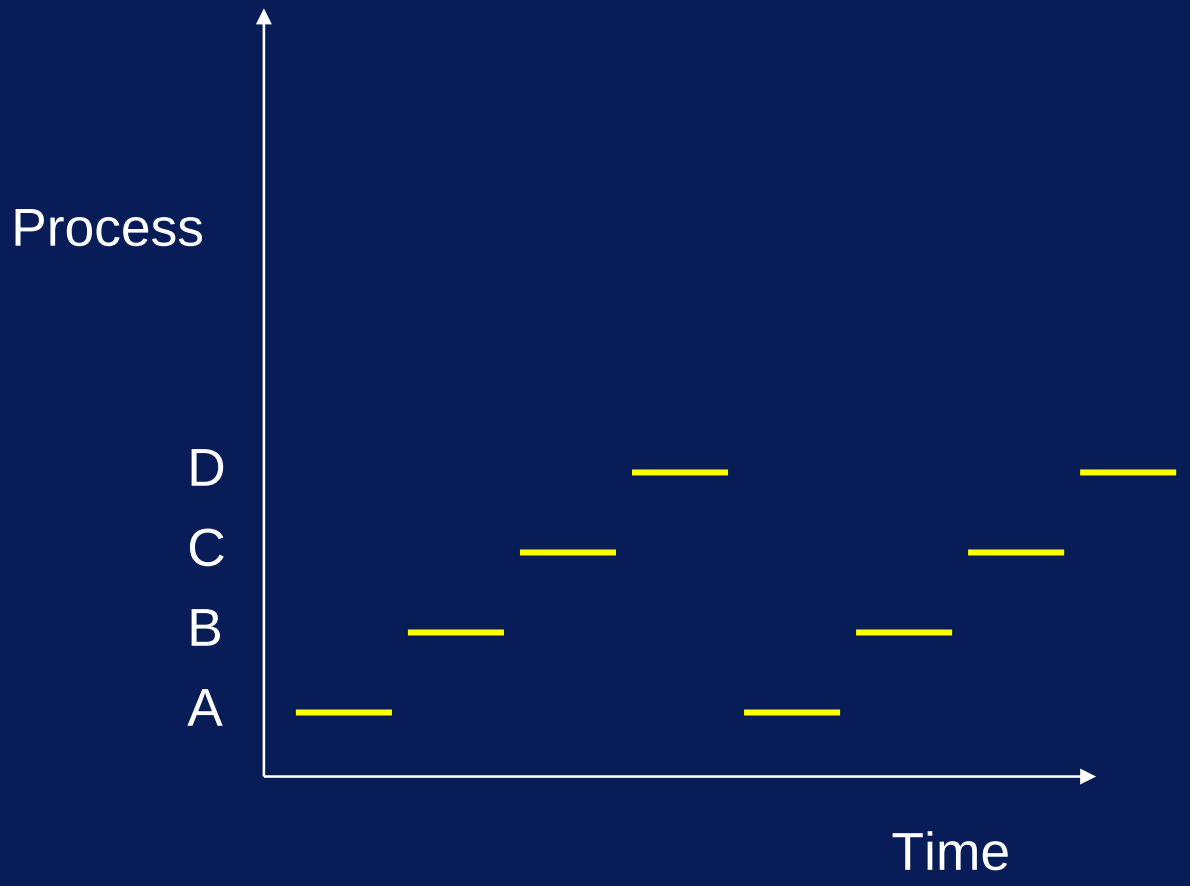
Physical view of Multiprogramming

Four progam counters



Logical view of Multiprogramming

Processes



Process activation vs. time

Processes

- Because the OS schedules process activity according to current needs, software-based timing assumptions are inherently unreliable

Processes

- A process is an activity:
  - software program
  - input
  - output
  - state (PC, registers, memory)

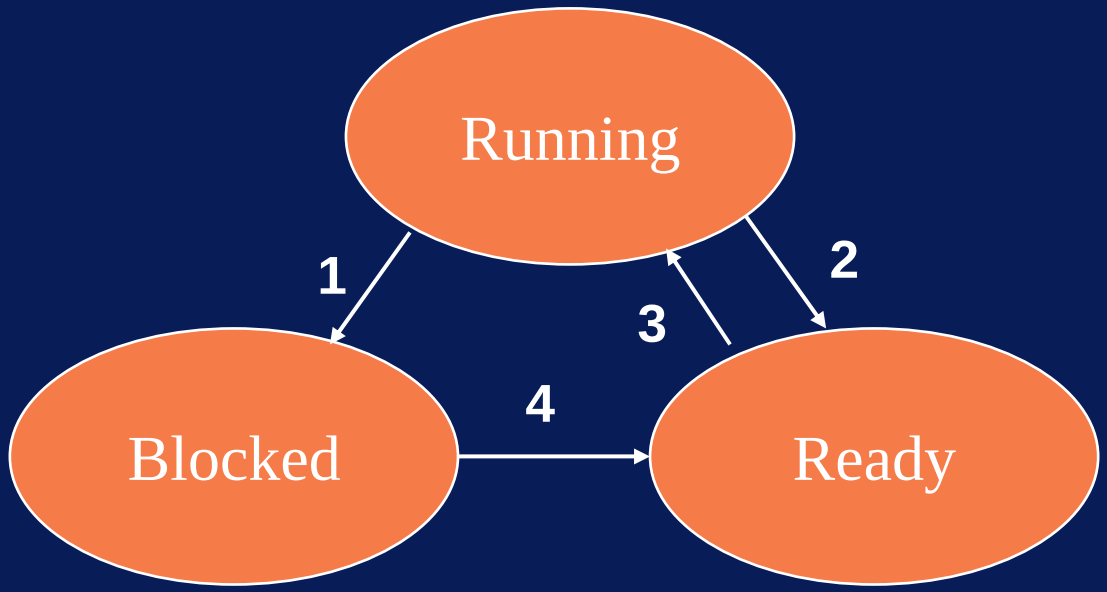
Processes

- Process Hierarchies
  - process creation (FORK in UNIX)
    - » parent forks a child process; both remain instantiated and running
- Process States
  - processes can communicate with one another
  - if a process is expecting input, and that input is not available, the process must block

Processes

- A process may be in one of three different states:
  - Running (using CPU resources)
  - Ready (stopped/suspended by the OS but able to run)
  - Blocked (unable to run until some external event happens)

Processes



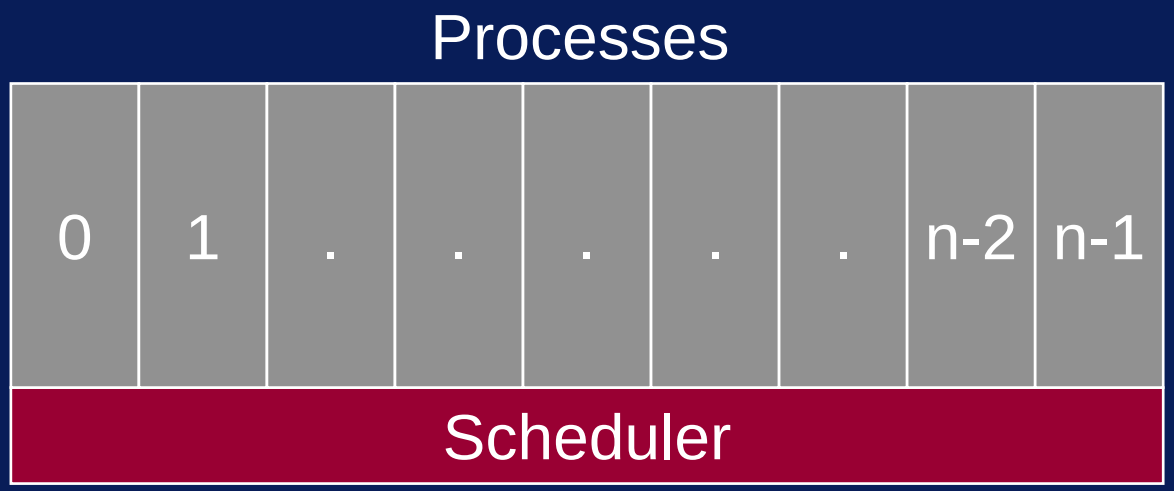
1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Processes

- Transition 1
  - sometimes the process must issue a BLOCK system call
  - sometimes the process is automatically blocked if no input is available (e.g STDIO)
- Transition 2 & 3
  - caused by the process scheduler
- Transition 4
  - occurs when external event triggers availability of resource that caused block

Processes

- Device I/O … the different views
  - bare machine: interrupts
  - OS machine: processes and process blocking/unblocking



Processes

- Implementation of Processes
  - OS maintains a process table
    - » one entry per process
    - » each entry comprises (typically)
      - PC, stack pointer, memory allocation, status of open files, accounting and scheduling information, information relating to process management, memory management, and file management

Processes

- Interrupts and device I/O on a process-oriented machine
  - » Interrupt from a device (e.g. disk) occurs
  - » Hardware stacks PC and status register
  - » Hardware load new PC from interrupt vector
  - » Assembly language procedure saves registers
  - » Assembly language procedure set up new stack
  - » C procedure marks service process as READY
  - » Scheduler decides which process to run next
  - » C procedure returns to the assembly code
  - » Assembly language procedure starts up current process

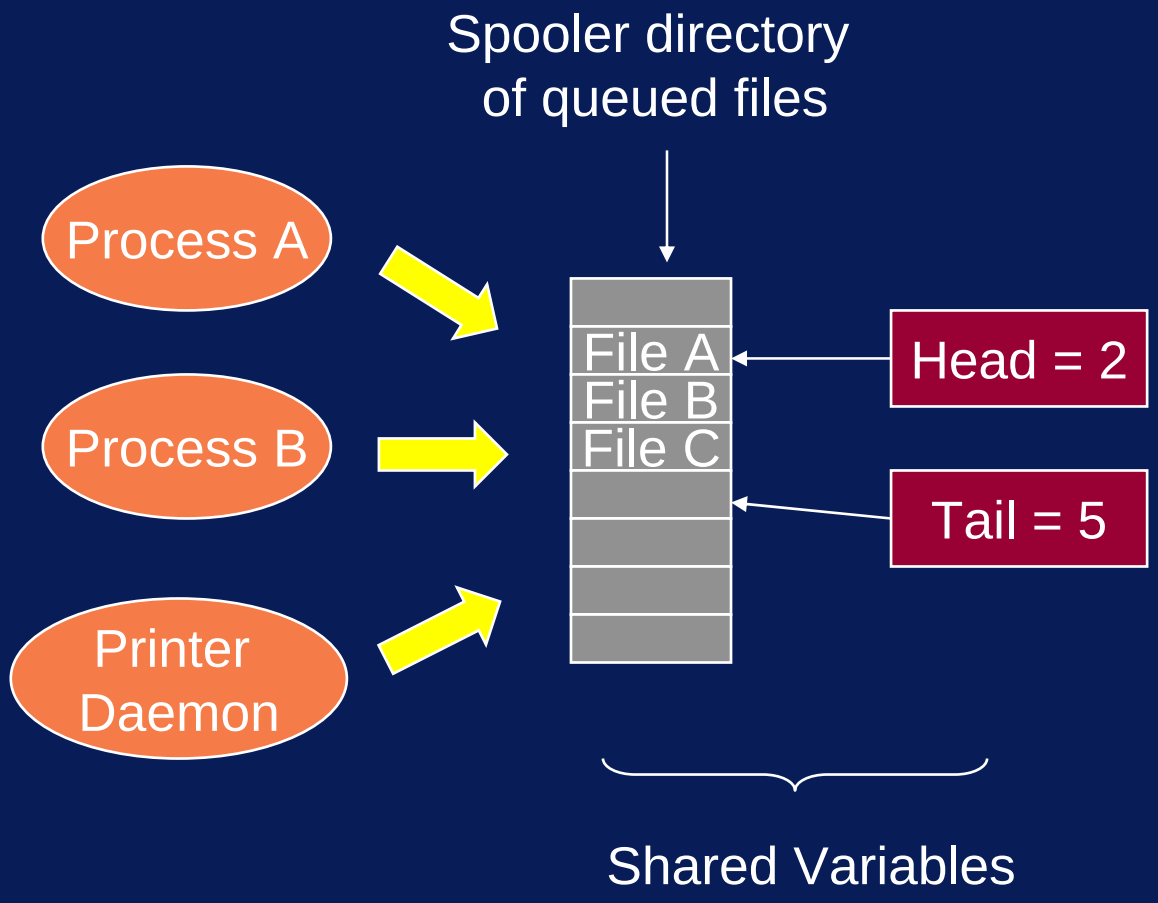
Processes

- InterProcess Communication IPC
  - Processes often communicate by sharing a common storage location
    - » shared memory
    - » shared file
  - Doing this effectively causes some significant problems
  - In this section, we will look at these problems and several solutions

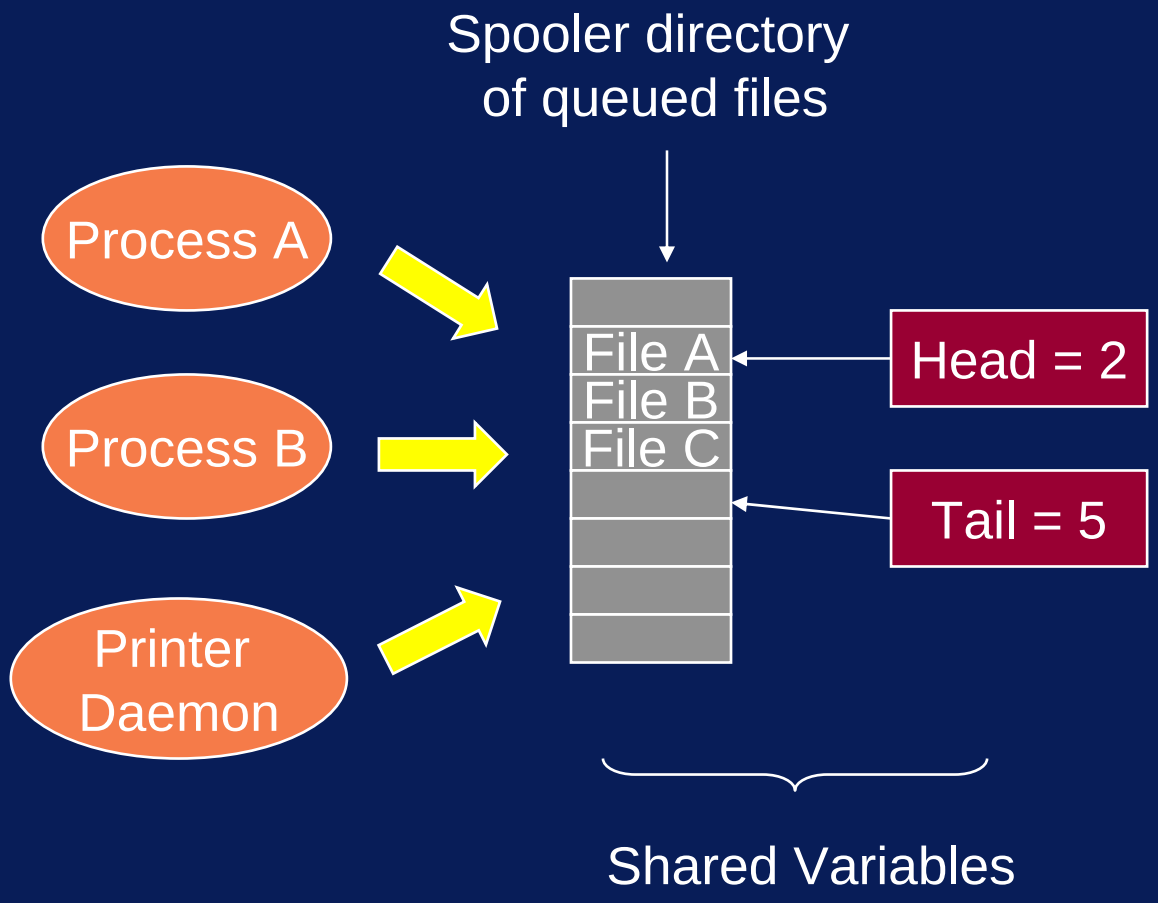
Processes

- InterProcess Communication IPC
  - The problems of sharing resources
  - Consider a typical IPC task - a print spooler

Processes

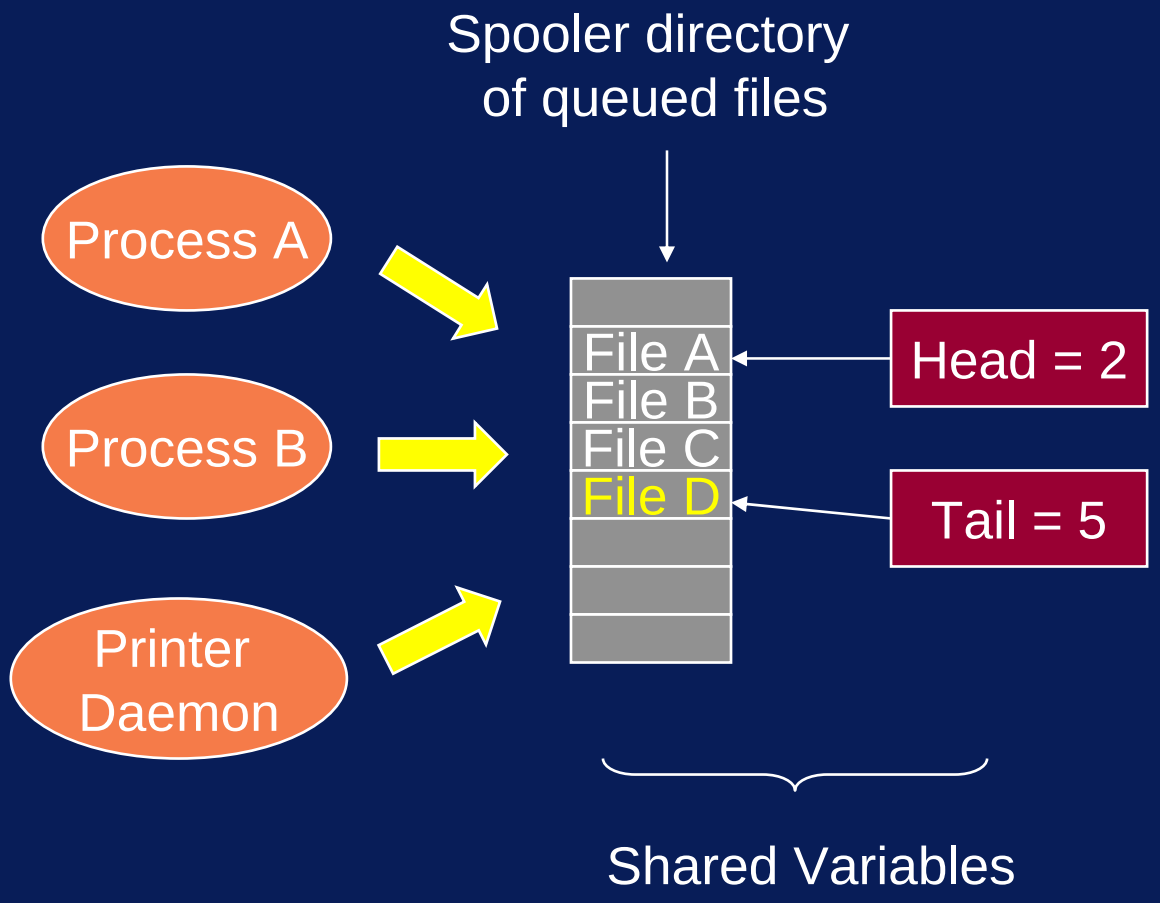


Processes



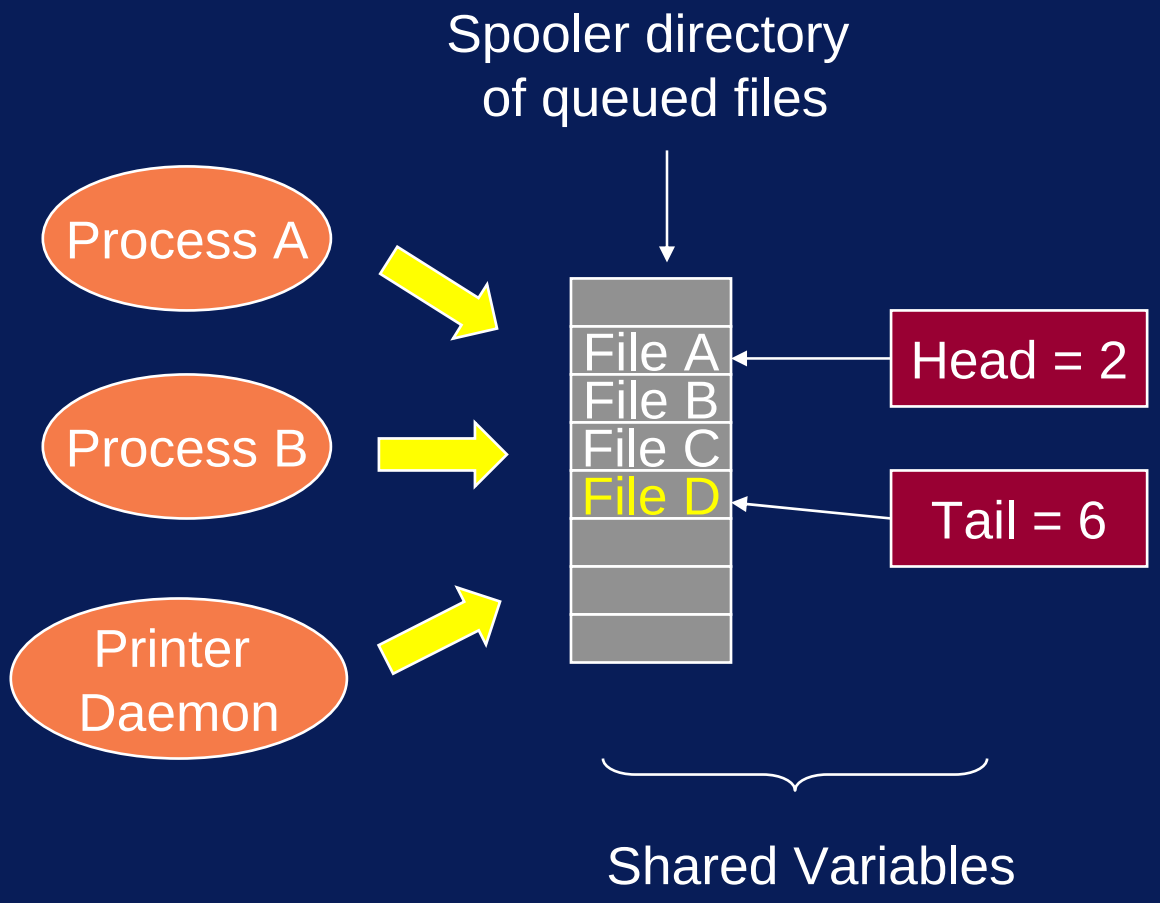
Process A queues a file for printing:
- reads Tail (5)

Processes



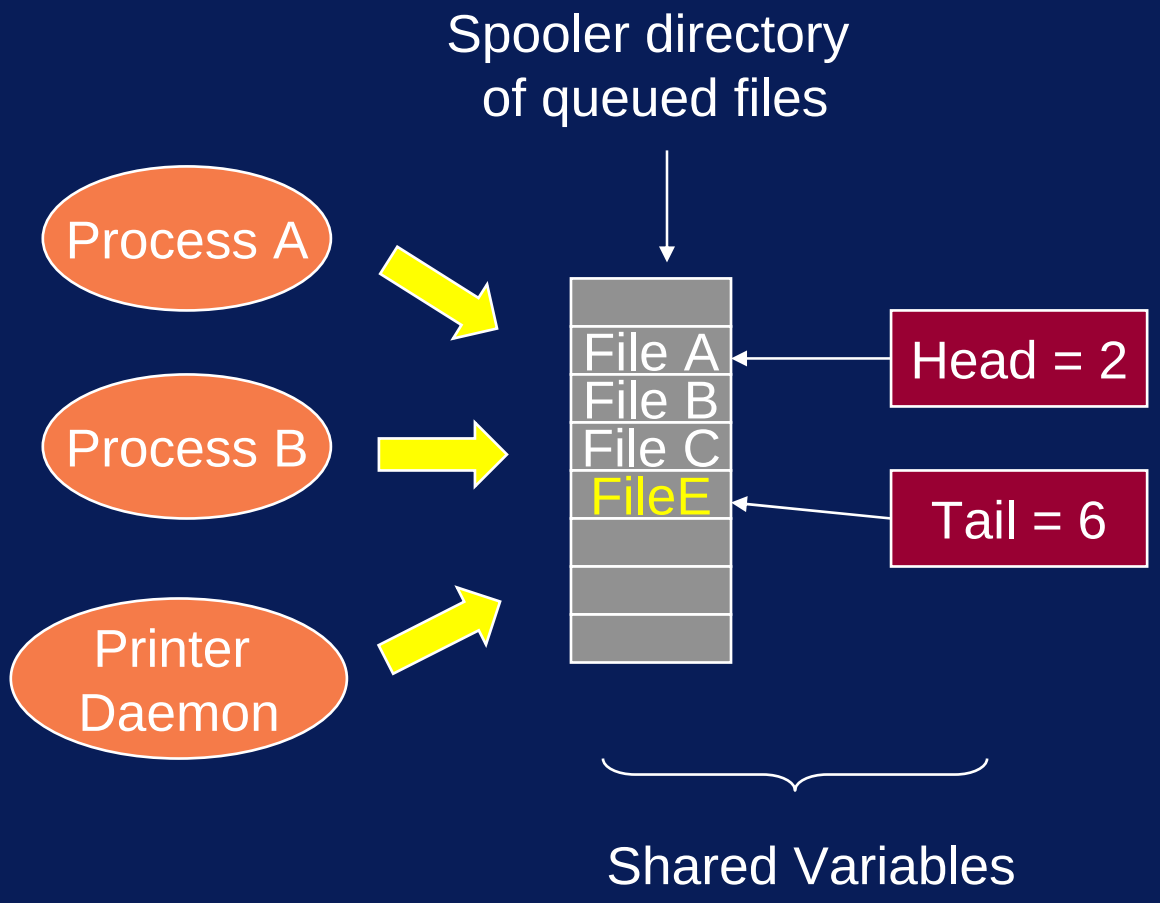
Process A queues a file for printing:
- reads Tail (5)
but then Process A suspended by scheduler
Process B queues a file for printing:
- reads Tail (5)
- writes filename to directory
- updates tail writes 6

Processes



Process A queues a file for printing:
- reads Tail (5)
but then Process A suspended by scheduler
Process B queues a file for printing:
- reads Tail (5)
- writes filename to directory
- updates tail writes 6

Processes



Process A resumes
- writes filename to directory (slot 5)
- effectively over-writing B's entry
- updates tail (writes 6)

Processes

- InterProcess Communication IPC
  - This unintentional overwriting of B's queued request by A's queued request is called a race condition
  - Note that the Printer Daemon dequeues printer requests (and prints the file)

Processes

- InterProcess Communication IPC
  - Race conditions have to be avoided
  - This is done by ensuring mutual exclusion of access to the shared resources by the relevant processes
  - This is only necessary for the sections of a process concerned with the shared resources; these are called critical sections

Processes

- InterProcess Communication IPC
  - Race conditions can be avoided if 4 conditions are satisfied:
    - » No two processes may be simultaneously inside their critical sections
    - » No assumptions may be made about speeds or number of CPUs
    - » No process running outside its critical section may block other processes
    - » No process should have to wait forever to enter its critical section

Processes

- InterProcess Communication IPC
  - In the following we will study 5 approaches for achieving mutual exclusion:
    - » Mutual exclusion with Busy Waiting
    - » Sleep and Wakeup
    - » Semaphores
    - » Monitors
    - » Message passing
  - In fact, all are theoretically equivalent

Processes

- InterProcess Communication IPC
  - Mutual Exclusion with Busy Waiting
    - » Technique 1: Disabling interrupts
      - Process scheduling only happens on clock or other interrupts
      - So process might disable interrupts upon entering a critical region
      - And re-enable them on exiting
      - But it's not a good idea to allow user processes the power to disable interrupts (what if they fail to re-enable them …)
      - Also, won't work in a multi-CPU system

Processes

- InterProcess Communication IPC
  - Mutual Exclusion with Busy Waiting
    - » Technique 2: Lock Variables
    - » Use a single shared ‘lock’ variable
    - » Process A tests lock
    - » if zero, enter critical region and set to 1
    - » if non-zero, wait until it is zero & keep testing
    - » But we will have the same problem as before:
      - Process A might be suspended before it gets a chance to set the lock to 1 and process B then won’t know that Process A is about to enter its critical region

Processes

- InterProcess Communication IPC
  - Mutual Exclusion with Busy Waiting
    - » Technique 3: Strict Alternation
    - » Continuously test a variable to see if your turn has come!

```
/* Process A */

while (TRUE) {
   while (turn != 0);/* wait */
   critical_section();
   turn = 1;
   noncritical_section();
}
```

```
/* process B */

while (TRUE) {
   while (turn != 1);/* wait */
   critical_section();
   turn = 0;
   noncritical_section();
}
```

Processes

- InterProcess Communication IPC
  - Mutual Exclusion with Busy Waiting
    - » Technique 3: Strict Alternation
    - » Continuous testing of a variable is called busy waiting
      - should be avoided as it wastes CPU time
    - » Another problem: strict alternation (my turn, your turn, my turn, …) means that if Process A can't re-enter its critical region until Process B has had a turn
      - consider what would happen if B didn't need to take a turn for a long time
    - » This is not a real contender as a solution

Processes

- InterProcess Communication IPC
  - Mutual Exclusion with Busy Waiting
    - » Technique 4: Peterson's Solution (1981)
    - » processes call routine **`enter_region`** before entering critical region
      - pass their process number as an argument
      - if resource is available, **`enter_region`** returns having set a flag to say that process n is now active in its critical region
      - otherwise it starts busy waiting until the other process is no longer active
    - » processes call routine **`leave_region`** after exiting critical region

Processes

```
#include “prototypes.h”
#define FALSE 0
#define TRUE  1
#define N     2                     /* number of processes */

int turn;                           /* global variable: who’s turn is it */
int interested[N];                  /* all values initally 0 (FALSE)     */

void enter_region(int process) /* process: who is entering (0 or 1) */
{
   int other;                       /* id number of the other process    */

   other = 1 - process;             /* set it to the other (of 2) processes *
   interested[process] = TRUE; /* record interest in entering region   *
   turn = process;                  /* set flag                             *
   while (turn == process && interested[other] == TRUE) ; /* wait */
}

void leave_region(int process) /* process: who is leaving (0 or 1) */
{
   interested[process] = FALSE;/* indicate departure                */
}                              /* from critical region              */
```

Processes

```
/* process 0 */

main () {
   while (TRUE) {
      do_some_work();
      enter_region(0);
      critical_section();
      leave_region(0);
      do_more_work();
   }
}
```

```
/* process 1 */

main () {
   while (TRUE) {
      do_something();
      enter_region(0);
      critical_section();
      leave_region(0);
      do_something_else();
   }
}
```

Processes

```
/* process 0 */

main () {
   while (TRUE) {
      do_some_work();
      enter_region(0);
      critical_section();
      leave_region(0);
      do_more_work();
   }
}
```

```
/* process 1 */

main () {
   while (TRUE) {
      do_something();
      enter_region(0);
      critical_section();
      leave_region(0);
      do_something_else();
   }
}
```

Processes

```
/* process 0 */

main () {
   while (TRUE) {
      do_some_work();
      enter_region(0);
      critical_section();
      leave_region(0);
      do_more_work();
   }
}
```

```
/* process 1 */

main () {
   while (TRUE) {
      do_something();
      enter_region(0);
      critical_section();
      leave_region(0);
      do_something_else();
   }
}
```

```
while (turn == process && interested[other] == TRUE) ; /* wait */
```

Processes

```
/* process 0 */

main () {
   while (TRUE) {
      do_some_work();
      enter_region(0);
      critical_section();
      leave_region(0);
      do_more_work();
   }
}
```

```
/* process 1 */

main () {
   while (TRUE) {
      do_something();
      enter_region(0);
      critical_section();
      leave_region(0);
      do_something_else();
   }
}
```

process: 0
other: 1

```
while (turn == process && interested[other] == TRUE) ; /* wait */
```

Processes

```
/* process 0 */

main () {
   while (TRUE) {
      do_some_work();
      enter_region(0);
      critical_section();
      leave_region(0);
      do_more_work();
   }
}
```

```
/* process 1 */

main () {
   while (TRUE) {
      do_something();
      enter_region(1);
      critical_section();
      leave_region(1);
      do_something_else();
   }
}
```

process: 0
other: 1

process: 1
other: 0

```
while (turn == process && interested[other] == TRUE) ; /* wait */
```

Processes

```
/* process 0 */

main () {
   while (TRUE) {
      do_some_work();
      enter_region(0);
      critical_section();
      leave_region(0);
      do_more_work();
   }
}
```

```
/* process 1 */

main () {
   while (TRUE) {
      do_something();
      enter_region(1);
      critical_section();
      leave_region(1);
      do_something_else();
   }
}
```


```
process: 1
other:   0
```

```
while (turn == process && interested[other] == TRUE) ; /* wait */
```

Processes

```
/* process 0 */

main () {
   while (TRUE) {
      do_some_work();
      enter_region(0);
      critical_section();
      leave_region(0);
      do_more_work();
   }
}
```

```
/* process 1 */

main () {
   while (TRUE) {
      do_something();
      enter_region(1);
      critical_section();
      leave_region(1);
      do_something_else();
   }
}
```


```
process: 1
other:   0
```

Processes

```
/* process 0 */

main () {
   while (TRUE) {
      do_some_work();
      enter_region(0);
      critical_section();
      leave_region(0);
      do_more_work();
   }
}
```

```
/* process 1 */

main () {
   while (TRUE) {
      do_something();
      enter_region(1);
      critical_section();
      leave_region(1);
      do_something_else();
   }
}
```

Processes

```
/* process 0 */

main () {
   while (TRUE) {
      do_some_work();
      enter_region(0);
      critical_section();
      leave_region(0);
      do_more_work();
   }
}
```

```
/* process 1 */

main () {
   while (TRUE) {
      do_something();
      enter_region(1);
      critical_section();
      leave_region(1);
      do_something_else();
   }
}
```

Processes

```
/* process 0 */

main () {
   while (TRUE) {
      do_some_work();
      enter_region(0);
      critical_section();
      leave_region(0);
      do_more_work();
   }
}
```

```
/* process 1 */

main () {
   while (TRUE) {
      do_something();
      enter_region(1);
      critical_section();
      leave_region(1);
      do_something_else();
   }
}
```

Processes

- InterProcess Communication IPC
  - Mutual Exclusion with Busy Waiting
    - » Technique 5: TSL
    - » hardware-based solution
    - » uses a special instruction: Test and Set Lock TSL
      - reads contents of memory word into a register
      - stores a non-zero value at that memory location
    - » since interrupts only occur at the end of an instruction cycle, TSL is ‘atomic’ and can’t be interrupted
    - » TLS locks the memory bus: other CPUs can’t access that location until the operation is complete

Processes

- InterProcess Communication IPC
  - Mutual Exclusion with Busy Waiting
    - » Technique 5: TSL

```
enter_region:
   tsl register, flag      ! Copy flag to register and set flag to 1
   cmp register, #0        ! Was flag zero?
   jnz enter_region        ! If it was non-zero, lock was set, so loop
   ret                     ! Now zero, so return to caller and allow entry
                           ! to critical region

leave_region:
   mov flag, #0            ! Store a 0 in flag
   ret                     ! Return to caller
```

Processes

- InterProcess Communication IPC
  - Mutual Exclusion with Busy Waiting
    - » Both Peterson's Solution and the TSL solution are correct
    - » But both require busy waiting:
      - if a process wants to enter its critical region,
      - it checks a flag to see if entry is allowed
      - if it is, proceed
      - otherwise, loop continuously until it is
    - » Not only is busy waiting wasteful of CPU time, it also causes other problems: priority inversion

Processes

- InterProcess Communication IPC
  - Mutual Exclusion with Busy Waiting
    - » Priority inversion
      - Low priority process A is in critical region
      - High priority process B is busy waiting
      - Since B is always read to run (in wait loop)
        - process A never gets scheduled,
        - and never leaves its critical region
        - And, hence, B never enters its critical region
      - A type of ‘livelock’ occurs

Processes

- InterProcess Communication IPC
  - Sleep and Wakeup
    - » IPC primitives that block instead of busy waiting
    - » SLEEP
      - system call that causes the caller to block (i.e be suspended until another process wakes it up)
    - » WAKEUP
      - system call that wakes or unblocks a process
      - one parameter: the process id. of process to be unblocked

Processes

- InterProcess Communication IPC
  - Sleep and Wakeup
    - » The Producer-Consumer Problem
    - » (also known as the bounded buffer problem)
    - » Two processes share a fixed-size buffer
      - producer process put information in
      - consumer process takes information out
    - » Problem for producer when buffer is full
    - » Solution:
      - producer goes to sleep
      - and is awakened by consumer when consumer has removed some items

Processes

- InterProcess Communication IPC
  - Sleep and Wakeup
    - » Problem for consumer when buffer is empty
    - » Solution:
      - consumer goes to sleep
      - and is awakened by producer when producer has added some items
    - » Use a shared variable **`count`** to identify the number of items in the buffer (max = N)
    - » Both producer and consumer need to check **`count`** to see what action to take
    - » And we can get the familiar race conditions

Processes

```
#include “prototypes.h”
#define N     100                 /* number of slots in the buffer      */

int count = 0;                    /* number of items in the buffer      */

void producer(void)
{
   int item;

   while (TRUE) {                 /* repeat forever                     */
      produce_item(&item);        /* generate next item                 */
      if (count == N) sleep();    /* if buffer is full, go to sleep     */
      enter_item(item);           /* put item in buffer                 */
      count = count + 1;          /* increment count of items in buffer */
      if (count == 1) wakeup(consumer) /* was buffer empty              */
   }
}
```

Processes

```
void consumer(void)
{
   int item;

   while (TRUE) {              /* repeat forever                          */
      if (count == 0) sleep(); /* if buffer is empth, go to sleep         */
      remove_item(&item);      /* take item out of buffer                 */
      count = count - 1;       /* deccrement count of items in buffer     */
      if (count == N-1) wakeup(producer); /* was buffer full              */
      consume_item(&item);
   }
}
```

Processes

- InterProcess Communication IPC
  - Sleep and Wakeup
    - » race condition occurs when:
      - buffer is empty
      - consumer reads count (contents is 0)
      - scheduler suspends consumer
      - scheduler runs producer
      - producer enter item in buffer
      - producer increments count (contents is now 1)
      - producer sends WAKEUP consumer
      - But consumer is NOT asleep, so WAKEUP is lost
      - scheduler suspends producer
      - scheduler resumes consumer

Processes

- InterProcess Communication IPC
  - Sleep and Wakeup
    - » race condition occurs when:
      - consumers tests value (previously read) count
      - value is 0 so consumer sleeps()
      - scheduler resumes producer
      - eventually, producer will fill the buffer and sleep()
      - now both producer and consumer sleep … deadlock!
    - » Key problem: waking up a process that is not asleep … wasting a wakeup call
    - » The solution is to use Semaphores

Processes

- InterProcess Communication IPC
  - Semaphores (Dijkstra 1965)
    - » a special type of variable
    - » used to count the number of wakeups (for future use)
    - » can have a value 0 … no wakeups saved
    - » can have a positive integer value … the number of wakeups pending
    - » two operations: DOWN and UP
      - (P and V in the original paper)

Processes

- InterProcess Communication IPC
  - Semaphores (Dijkstra 1965)
    - » DOWN
      - generalization of SLEEP

checks if semaphore value
is greater than zero

```
if semaphore > 0
   then
      decrement semaphore
   else
      process is blocked (sleep)
      decrement semaphore
```

Processes

- InterProcess Communication IPC
  - Semaphores (Dijkstra 1965)
    - » DOWN
      - generalization of SLEEP

checks if semaphore value
is greater than zero

if semaphore > 0
  then
    decrement semaphore
  else
    process is blocked (sleep)
    decrement semaphore

ATOMIC OPERATION

Processes

- InterProcess Communication IPC
  - Semaphores (Dijkstra 1965)
    - » UP
      - generalization of WAKEUP

        increments semaphore value

        if there is one or more processes
        sleeping on the semaphore
        then
        choose a process at random
        unblock it
        (allow it to complete its DOWN operation)

Processes

- InterProcess Communication IPC
  - Semaphores (Dijkstra 1965)
    - » UP
      - generalization of WAKEUP

ATOMIC OPERATION

increments semaphore value
if there is one or more processes
sleeping on the semaphore
then
choose a process at random
unblock it
(allow it to complete its DOWN operation)

Processes

- InterProcess Communication IPC
  - Semaphores
    - » UP and DOWN *must* be implemented in an atomic/indivisible manner
    - » Usually implemented as system calls, with interrupts disabled for the duration
    - » On multi-CPU systems, each semaphore can be protected using TSL instruction
      - This is valid as the UP and DOWN semaphore operations only require a few microseconds

Processes

- InterProcess Communication IPC
  - Semaphores
    - » solution to producer-consumer problem
      - use 3 semaphores
        - `full` number of slots that are full (0 initially)
        - `empty` number of slots that are empty (N initially)
        - `mutex` ensure mutual exclusion (for access to buffer by producer and consumer)
      - `mutex` is initialized to 1 and is a **binary semaphore**
        - it is used to guarantee only one process can enter its critical region at a given time
        - `down(mutex)` before entering critical region
        - `up(mutex)` after entering critical region

Processes

```
#include “prototypes.h”
#define N     100              /* number of slots in the buffer         */
typedef int semaphore          /* semaphores are a special kind of int */
semaphore mutex = 1;           /* control access to critical region    */
semaphore empty = N;           /* counts empty buffer slots            */
semaphore full  = 0;           /* counts full buffer slots             */


void producer(void)
{
   int item;

   while (TRUE) {                /* repeat forever                    */
      produce_item(&item);       /* generate next item                */
      down(&empty);              /* decrement empty count             */
      down(&mutex);              /* enter critical region             */
      enter_item(item);          /* put item in buffer                */
      up(&mutex);                /* leave critical region             */
      up(&full);                 /* increment count of full slots     */
   }
}
```

Processes

```
void consumer(void)
{
   int item;

   while (TRUE) {                /* repeat forever                    */
      down(&full);               /* decrement full count              */
      down(&mutex);              /* enter critical region             */
      remove_item(&item);        /* take item out of buffer           */
      up(&mutex);                /* leave critical region             */
      up(&empty);                /* increment count on empty slots    */
      consume_item(&item);
}
```

Processes

- InterProcess Communication IPC
  - Semaphores
    - » In the solution to the producer-consumer problem we used semaphores for two distinct purposes
    - » mutual exclusion (`mutex`)
    - » synchronization (`empty` and `full`)
      - guarantee certain event sequences do (or do not) occur
        - – producer stops when buffer is full
        - – consumer stops when buffer is empty

Processes

```
/* In the following _incorrect_ version of the producer we reverse   */
/* the order of down call on the semaphores.                         */
/* This will cause DEADLOCK if the buffer is full … Why?             */


void producer(void)
{
   int item;

   while (TRUE) {                  /* repeat forever                 */
      produce_item(&item);         /* generate next item             */
      down(&mutex);                /* enter critical region          */
      down(&empty);                /* decrement empty count          */
      enter_item(item);            /* put item in buffer             */
      up(&mutex);                  /* leave critical region          */
      up(&full);                   /* increment count of full slots  */
}
```

Processes

- InterProcess Communication IPC
  - Monitors
    - » A monitor is a collection of procedures, variables, and data-structures that are all grouped together in a special kind of module or package
    - » Processes may call the monitor procedures but the data-structures are private
    - » Only one process can be active in a monitor at any one instant
    - » Monitors are a programming language construct (so the compiler can treat them differently, typically by including mutual exclusion code)

Processes

- InterProcess Communication IPC
  - Monitors
    - » It is up to the compiler to implement the mutual exclusion … typically with a binary semaphore
    - » As it happens, most languages do NOT have monitors built in (exception: Concurrent Euclid) so they are not used much

Processes

- InterProcess Communication IPC
  - Message passing
    - » Semaphores were designed for use on
      - single CPU systems or
      - multi CPU systems with shared memory (in which case, we also use TSL)
    - » In a distributed system with multiple CPUs and private memory, we need something more
    - » message passing

Processes

- InterProcess Communication IPC
  - Message passing
    - » SEND
      - system call
      - `send (destination, &message);`
    - » RECEIVE
      - system call
      - `receive(source, &message);`
      - If no message is available, the receiver could block

Processes

- InterProcess Communication IPC
  - Message passing issues
    - » lost messages
      - receiver should send an acknowledgement message
      - if sender has not received an ack (in a certain time span) it re-sends
      - if the ack is lost, a second (unrequired) message is sent
      - solved by attaching a sequence number to each message (which will be the same for the resent message)

Processes

- InterProcess Communication IPC
  - Message passing issues
    - » Process naming
      - process@machine
      - process@machine.domain
    - » Authentication
      - Is the sender who he says he is?
      - Encryption can help

Processes

- InterProcess Communication IPC
  - Message passing solution to Producer-Consumer problem
    - » assume all messages are the same size
    - » assume message sent but not yet received are buffered automatically by the OS
    - » assume N messages

Processes

- InterProcess Communication IPC
  - Message passing solution to Producer-Consumer problem
    - » Consumer starts by sending N empty messages to the producer
    - » When a producer has a item to give to the consumer, it takes an empty message and send back a full one
    - » When a consumer receives a full message, it processes it and sends back an empty

Processes

- InterProcess Communication IPC
  - Message passing solution to Producer-Consumer problem
    - » If the producer works faster than the consumer, the producer will be blocked (waiting for an empty message)
    - » If the consumer works faster then the producer, then all messages will be empties waiting for producer to accept them; the consumer will be blocked waiting for a full message

Processes

- InterProcess Communication IPC
  - Message passing
    - » mailboxes
      - a place to buffer a certain number of messages
      - normally specified when the mailbox is created
      - SEND to and RECEIVE from mailboxes (rather than processes)
      - when a process tries to send to a mailbox that is full, it is suspended until a message is removed from that mailbox

Processes

- InterProcess Communication IPC
  - Message passing
    - » The UNIX IPC mechanism between user processes is the pipe
      - a pipe is a effectively a mailbox
      - but pipes don’t preserve message boundaries
      - process A sends 10 100byte messages to a pipe
      - process B can receive 10 100byte messages
      - or 1 1000byte message
      - or 20 50byte messages

Processes

```
/*** producer-consumer solution with message passing ***/

#include “prototypes.h”
#define N     100              /* number of slots in the buffer          */
#define MSIZE 4                /* message size                           */
typedef int message[MSIZE]

void producer(void)
{
   int item;
   message m;                    /* message buffer                       */

   while (TRUE) {
      produce_item(&item);       /* generate next item                   */
      receive(consumer, &m);     /* wait for an empty to arrive          */
      build_message(&m, item); /* construct a message to send            */
      send(consumer, &m);        /* send item to consumer                */
   }
}
```

Processes

```
void consumer(void)
{
   int item, i;
   message m;

   for (i=0; i<N; i++)
      send(producer, &m);      /* send N empties                    */

   while (TRUE) {              /* repeat forever                    */
      receive(producer, &m);   /* get message containing item       */
      extract_item(&m, &item); /* take item out of message          */
      send(producer, &m);      /* send back empty reply             */
      consume_item(&item);     /* do something with item            */
}
```

Processes

- InterProcess Communication IPC
  - Classical IPC Problem: The Readers and Writers Problem
    - » Access to a database by many processes
      - readers
      - writers
    - » Many can read at the same time
    - » But if a process is writing, no-one else should have access (even readers)

Processes

- InterProcess Communication IPC
  - Classical IPC Problem: The Readers and Writers Problem
    - » Solution using semaphores
    - » first reader does a DOWN on the semaphore db
    - » subsequent readers simply increment a counter rc
    - » as readers leave, they decrement rc
    - » last reader to leave does an UP on db
    - » if a writer is blocked, he can now get in
    - » this approach assumes readers have priority over writers

Processes

```
/*** solution to readers & writers with semaphores ***/

#include “prototypes.h”

typedef int semaphore
semaphore mutex = 1;          /* controls access to rc                  */
semaphore db = 1;             /* controls access to the database        */
int rc = 0;                   /* # of processes reading or wanting to */

void reader(void)
{
   while (TRUE) {             /* repeat forever                         */
      down(&mutex);           /* get exclusive access to rc             */
      rc = rc + 1;            /* one reader more now                    */
      if (rc == 1) down(&db); /* if this is the first reader            */
      up(&mutex);             /* release exclusive access to rc         */
      read_database();        /* access the database                    */
      down(&mutex);           /* get exclusive access to rc             */
      rc = rc - 1;            /* one reader fewer now                   */
      if (rc == 0) up(&db)    /* if this is the last reader …           */
      up(&mutex);             /* release exclusive access to rc         */
      use_data_read();        /* non critical section                   */
```

Processes

```
void writer(void)
{
   while (TRUE) {                  /* repeat forever                        */
      think_up_data();             /* non-critical section                  */
      down(&db);                   /* get exclusive access to database      */
      write_database();            /* update the data                       */
      up(&db);                     /* release exclusive access              */
}
```

Processes

- Process Scheduling
  - Scheduler: that part of the OS concerned with deciding which processes should run
  - Scheduling Algorithm: algorithm used by the scheduler

Processes

- Process Scheduling
  - Criteria for a good scheduling algorithm:
    - » Fairness: make sure each process gets its fair share of the CPU
    - » Efficiency: keep the CPU busy 100% of the time
    - » Response Time: minimize response time for interactive users
    - » Turnaround: minimize the time batch users must wait for output
    - » Throughput: maximise the number of jobs processed per hour
  - These goals are contradictory; why?

Processes

- Process Scheduling
  - Nearly all computers have a real-time clock which generates a (clock) interrupt at a given frequency (e.g. 50Hz or system-definable)
  - On each interrupt, the scheduler runs and decides which processes should run next
  - Preemptive scheduling: temporarily suspend processes to allow others to run
  - Non-preemptive scheduling: allow processes to run to completion

Processes

- Process Scheduling
  - Round Robin Scheduling
    - » each process is assigned a quantum of time for which it is allowed to run (in one go)
    - » at the end of a quantum, if the process is still running, then it is suspended (preempted)
    - » another process is run
    - » if the process blocks in a given quantum, the scheduler re-schedules immediately
    - » Keep a list of runnable processes and schedule each in turn

Processes

- Process Scheduling
  - Round Robin Scheduling
    - » how do we choose the quantum?
      - Too short and the context switch time (sometimes called process switch time) is large with respect to time the process runs (and efficiency decreases)
      - Too long and the response to short interactive requests is poor
      - A quantum on 100ms is a reasonable compromise
    - » Implicit assumption: all processes are equally important

Processes

- Process Scheduling
  - Priority Scheduling
    - » each process is assigned a priority
    - » the runnable process with the highest priority is run
    - » decrease the priority of a process at each clock interrupt (to stop high priority processes monopolizing the system)
    - » Priorities can be assigned dynamically or statically
      - for example: assign a high priority to I/O bound processes (why?) with a priority equal to 1/f, where f is the fraction of the last quantum that the process used.

Processes

- Process Scheduling
  - Priority Scheduling
    - » Some OS have priority classes
    - » and do round robin scheduling within a class
    - » Note: if priorities are not reassigned from time to time, some low priority processes will starve

Processes

- Process Scheduling
  - ‘Shortest Job First’ Scheduling
    - » Designed for batch systems
    - » Scheduler runs the shortest job first
    - » It minimizes the overall response time
    - » How to estimate the length of a job?
      - Estimate based on previous runs
      - E.g. take weighted average of runs at time t_i and t_{i-1}
        $t_{i+1} = w_{1\ i-1}\ t_{i-1} + w_{1\ i}\ t_i$
      - *This is called aging*

Processes

- Process Scheduling
  - Guaranteed Scheduling
    - » Decide what proportion of CPU time each process should get (e.g. equal share)
    - » Schedule processes to ensure this happens (e.g. by scheduling those processes who have current accumulated CPU time below their assigned level)

Processes

- Process Scheduling
  - Two Level Scheduling
    - » What if not all runnable processes are in memory (some might have been swapped out to disk)?
    - » The context switch time for a swapped process is orders of magnitude longer than that of a process in memory
    - » Solution: use two schedulers:
      - high-level scheduler decides which processes should be memory-resident and which should be swapped to disk (i.e. it schedules memory)
      - low-level scheduler runs/suspends memory resident processes (i.e. it schedules CPU time)

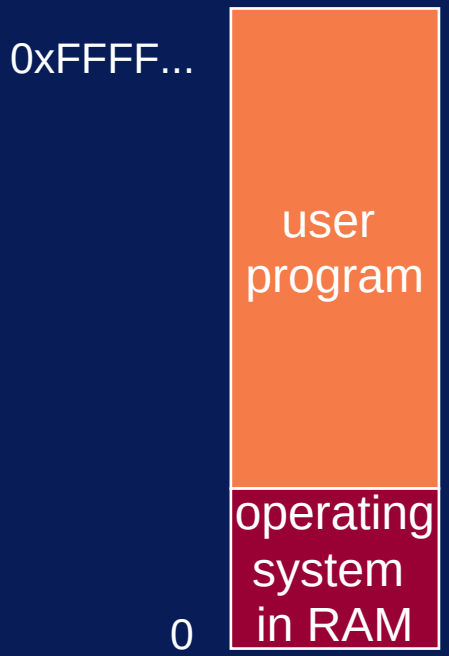
Memory Management

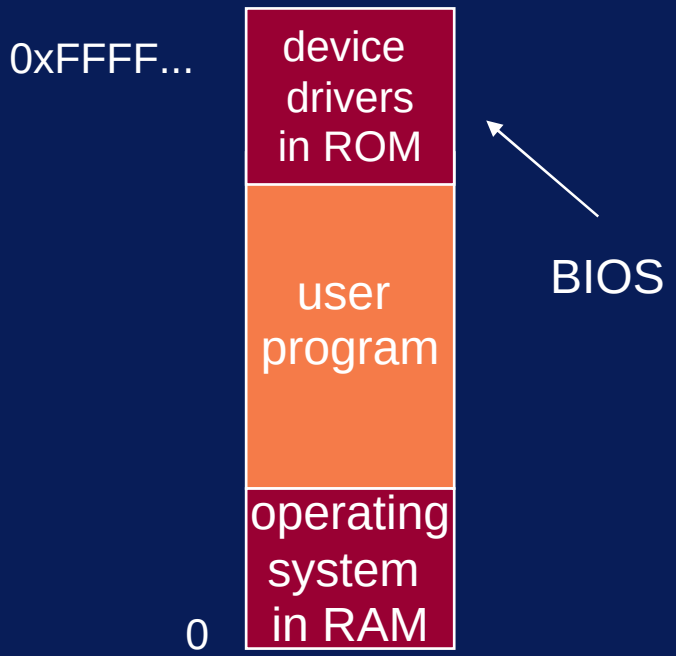
Memory Management

- The part of the operating system that manages memory is called the memory manager
- Two classes of memory management systems:
  - processes stay resident in memory until execution completes (i.e. without swapping)
  - processes are swapped in and out of memory during execution

Memory Management

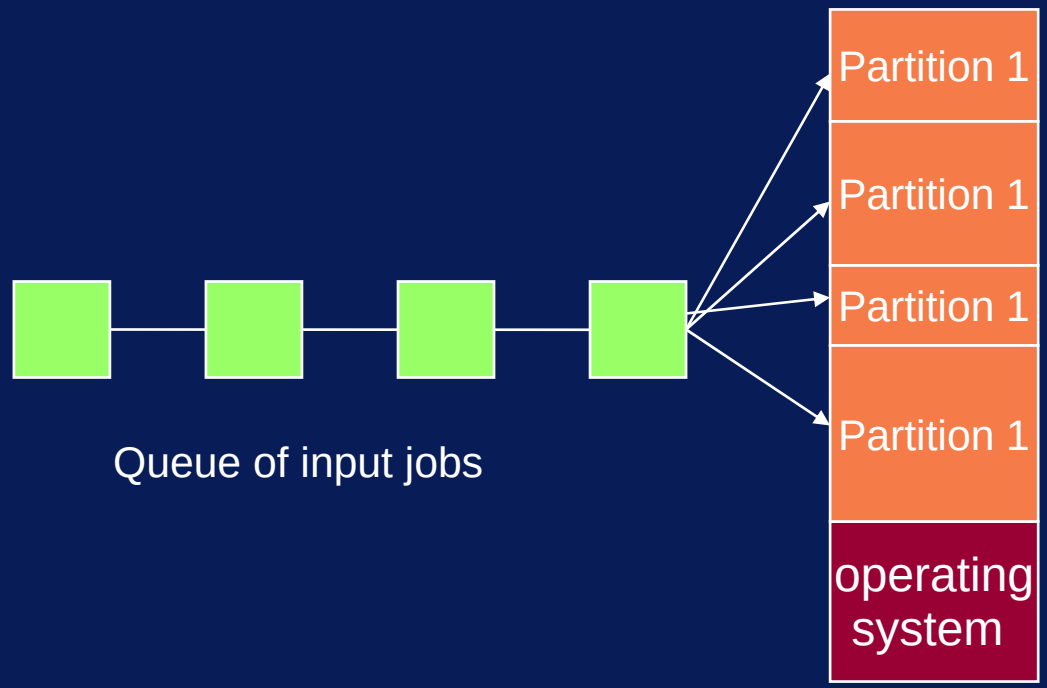
- Memory management without swapping
  - simplest approach: one process only in memory (and running)





Memory Management

- Memory management without swapping
  - Multiprogramming with fixed partitions



Memory Management

- Memory management without swapping
  - Multiprogramming causes two problems
    - » Relocation
      How to make references to memory addresses relative to the partition at which the process is loaded
    - » Protection
      How to ensure that all memory references are restricted to addresses in the partition assigned

Memory Management

- Memory management without swapping
  - One solution to the relocation and protection problems is to use base and limit registers
    - » when a process is scheduled, the base register is loaded with the address of the start of its partition and the limit register is loaded with the length of the partition
    - » Every memory address generated automatically has the base register contents added before the address is asserted

Memory Management

- Memory management without swapping
  - One solution to the relocation and protection problems is to use base and limit registers
    - » Addresses are also checked against the limit register to ensure that they don't address a location outside the current partition
    - » The Intel x86 uses base registers (the segment registers) but not limit registers

Memory Management

- Memory management with swapping
  - What if there is not enough memory to hold all processes?
  - We move processes back and forth between memory and disk
  - This is called swapping
  - Typically, when we do this we allow the partitions to change size: variable partitions
  - This means we need to keep track of memory usage (memory management)

Memory Management

- Swapping
  - memory management using linked lists
    - » keep a linked list of allocated and free memory segments
    - » each entry in the list:
      - allocated process segment
      - free hole segment between processes
      - start address of the segment
      - length of the segment
    - » When a process terminates, it becomes a free (hole) segment and we can try to combine adjacent memory segments to create fewer and bigger segments

Memory Management

- Swapping
  - memory management using linked lists
    - » To do this, the list is kept in sorted order by start address
    - » Memory is allocated for new processes from the list
    - » There are several algorithms for allocation from the list
      - first fit - first segment in the list big enough to hold the process
      - best fit - best fitting segment

Memory Management

- Swapping
  - memory management using linked lists
    - » These allocation algorithms can be made faster if we keep separate lists for processes and another for holes
    - » Unfortunately, they slow down deallocation since a freed segment has to be removed from the process list and added to the hole list

Memory Management

- Swapping
  - memory management using the Buddy System
    - » Keep several lists of free block of memory
    - » Each list corresponds to a block of a certain size (1, 2, 4, 8, 16, 32, …. 128, 256, 512, 1024, 2048kb ...); all block sizes are powers of 2
    - » Largest block size is the full size of physical memory (e.g. 16Mbyte)
    - » Initially, all of memory is free and the 16Mb list has one single entry; all other lists are empty

Memory Management

- Swapping
  - memory management using the Buddy System
    - » if a 3Mb process is swapped into memory:
      - a 4Mb block will be requested
      - none is available
      - so the 16Mb block is split in two (and both are added to the 8Mb list) - these are called buddies
      - one of the 8Mb blocks is split in two (and both are added to the 4Mb list)
      - one of the 4Mb blocks is allocated

Memory Management

- Swapping
  - memory management using the Buddy System
    - » if a 2Mb process is now swapped into memory:
      - a 2Mb block will be requested
      - none is available
      - so the remaining 4Mb block is split in two (and both are added to the 2Mb list)
      - one of the 2Mb blocks is allocated

Memory Management

- Swapping
  - memory management using the Buddy System
    - » if a process is now swapped out of memory:
      - its block is added back to the appropiate list
      - if two blocks in that list are adjacent (i.e. they form one contiguous segment of memory) they are merged, removed from that list, and added to the list of the next bigger size blocks.
      - this list is then checked to see if it is possible to merge,
      - and so on.

Memory Management

- Swapping
  - memory management using the Buddy System
    - » Buddy systems are fast
    - » but memory-inefficient (processes aren't always as big as the block)

Memory Management

- Virtual Memory
  - What happens if the process is too big to fit in memory?
  - The OS keeps those parts of the program currently in use in main memory and keeps the rest on disk
  - Pieces of the program and data are swapped between disk and memory as needed

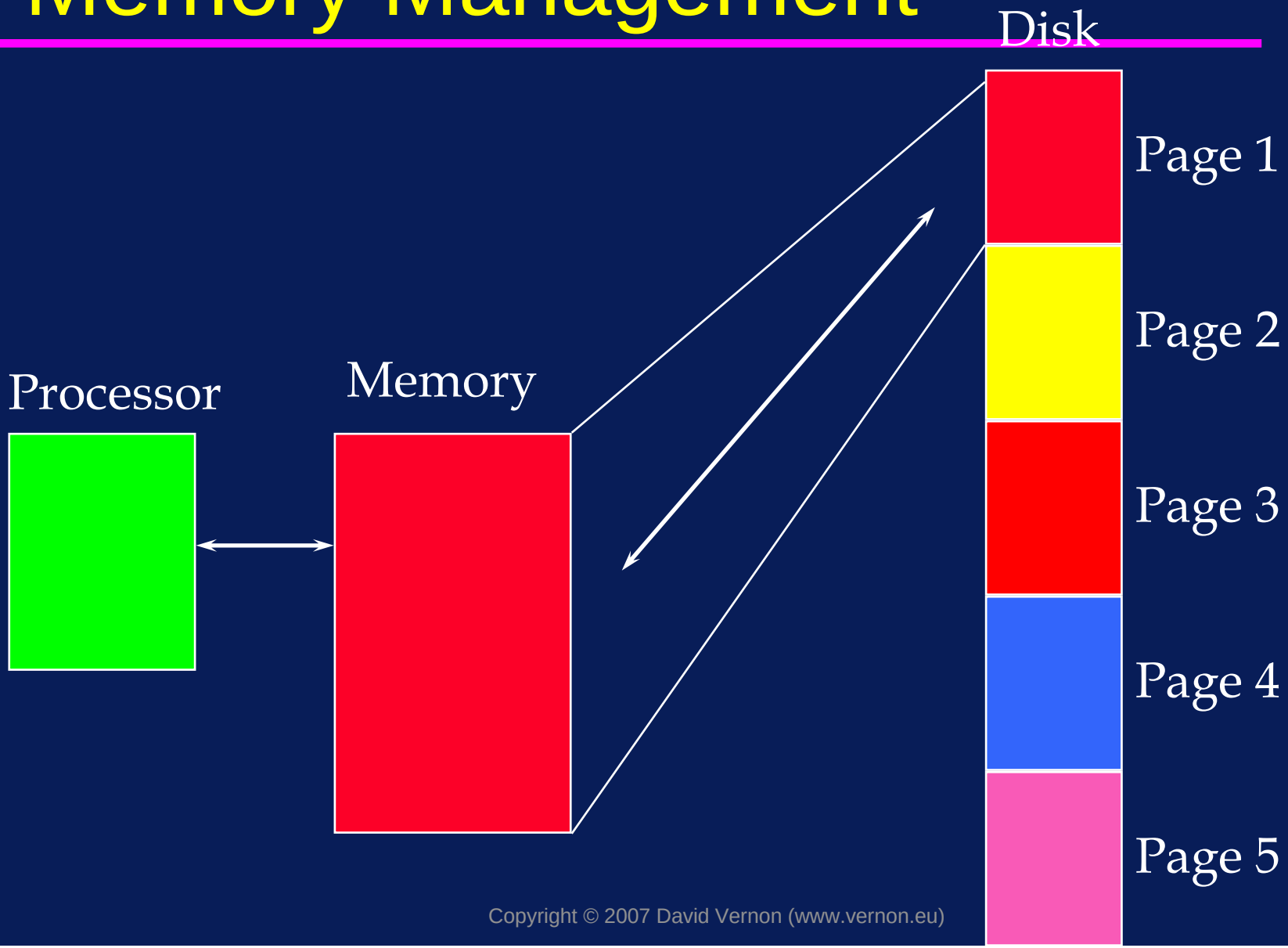
Memory Management

- Virtual Memory
  - Creates the illusion of a computer with more memory than it actually has
  - How? By moving some data to disk.
  - Blocks of ‘virtual’memory are them swapped in and out of disk whenever they are required:

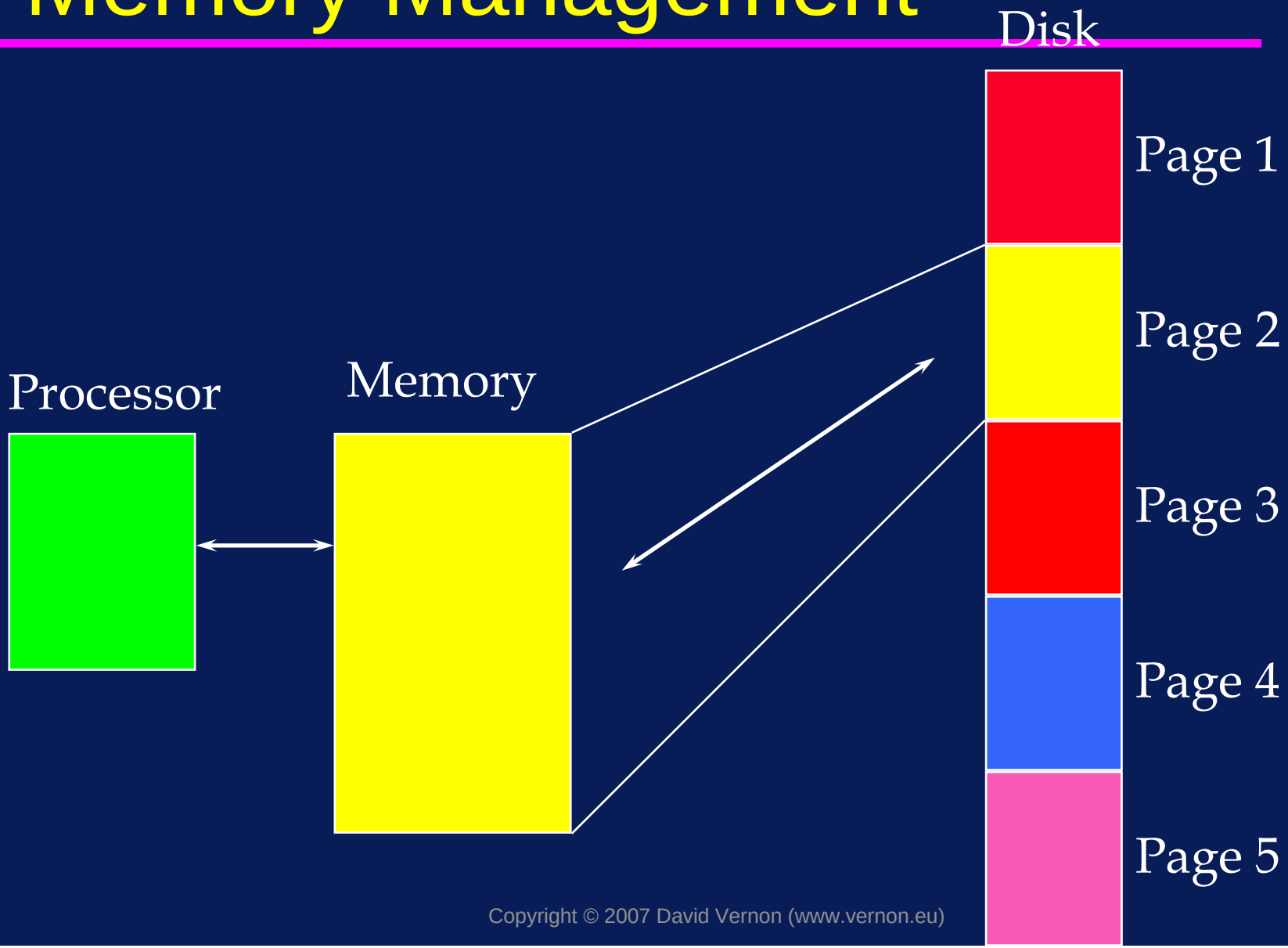
Memory Management

- When a program (or its data) needs to be larger than the size of available memory ...
- The program is divided into a number of pieces
- To run such a program:
  - Bring first piece into memory
  - Execute it until the next piece is needed
  - Bring it in ... and so on

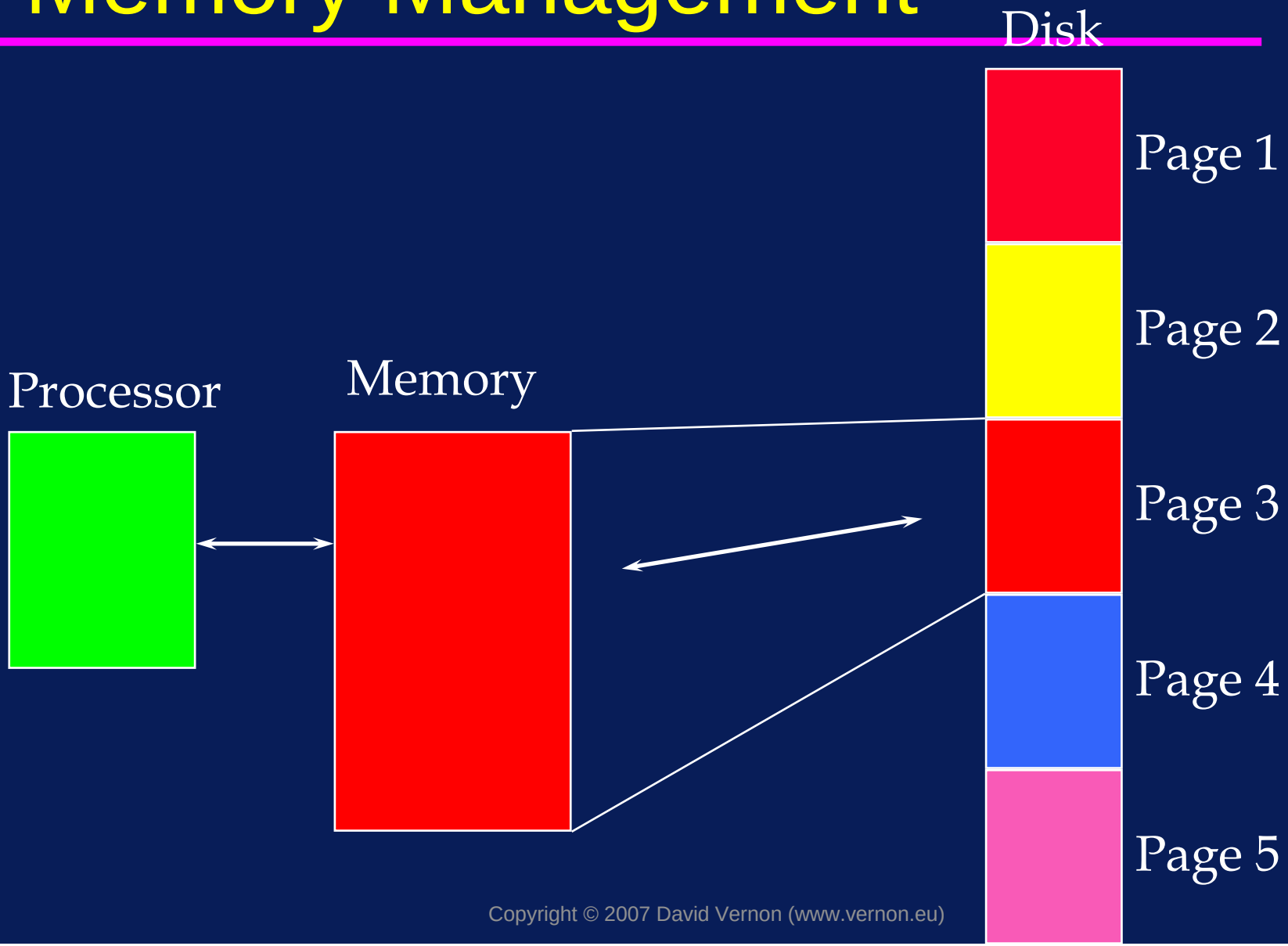
Memory Management



Memory Management



Memory Management



Memory Management

- The pieces which a program are broken into are called SEGMENTS and PAGES
- A segment is a variable-length piece of memory
- A page is a fixed-length piece of memory

Memory Management

- Virtual Memory
  - Paging
    - » most virtual memory systems use a technique called paging
    - » Virtual Address Space: the (logical) range of memory locations which a program can address
    - » the size of the address space is a function of the number of bits in the address
    - » On computers without virtual memory, the virtual address routed directly to the address but and that physical address is asserted

Memory Management

- Virtual Memory
  - Paging
    - » On computers with virtual memory, the virtual address is routed to a Memory Management Unit (MMU) which maps the virtual address onto a physical address (and then asserts that address on the bus)
    - » The virtual address space is divided into units called pages
    - » The corresponding units in physical memory are called page frames
    - » Pages and page frames are always the same size

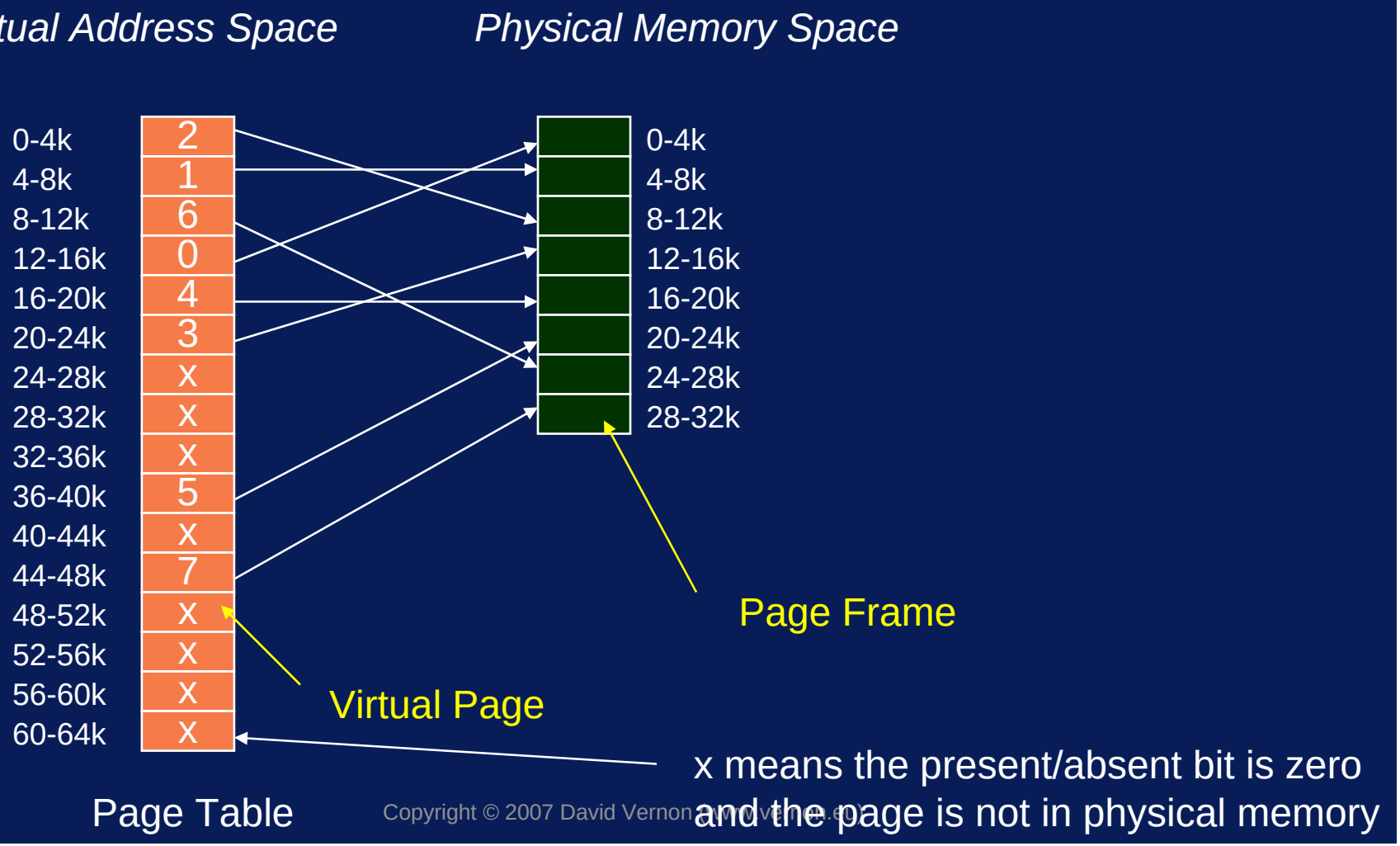
Memory Management

- Virtual Memory
  - Paging
    - » Pages sizes commonly range from 512 bytes to 8k
    - » Transfers between memory and disk are always in units of a page

Memory Management

- Virtual Memory
  - Paging; Scenario 1
    - » virtual address maps to the physical address
    - » MMU puts mapped address onto the address bus
    - » Virtual page *p* is mapped to physical page frame *q*

Memory Management



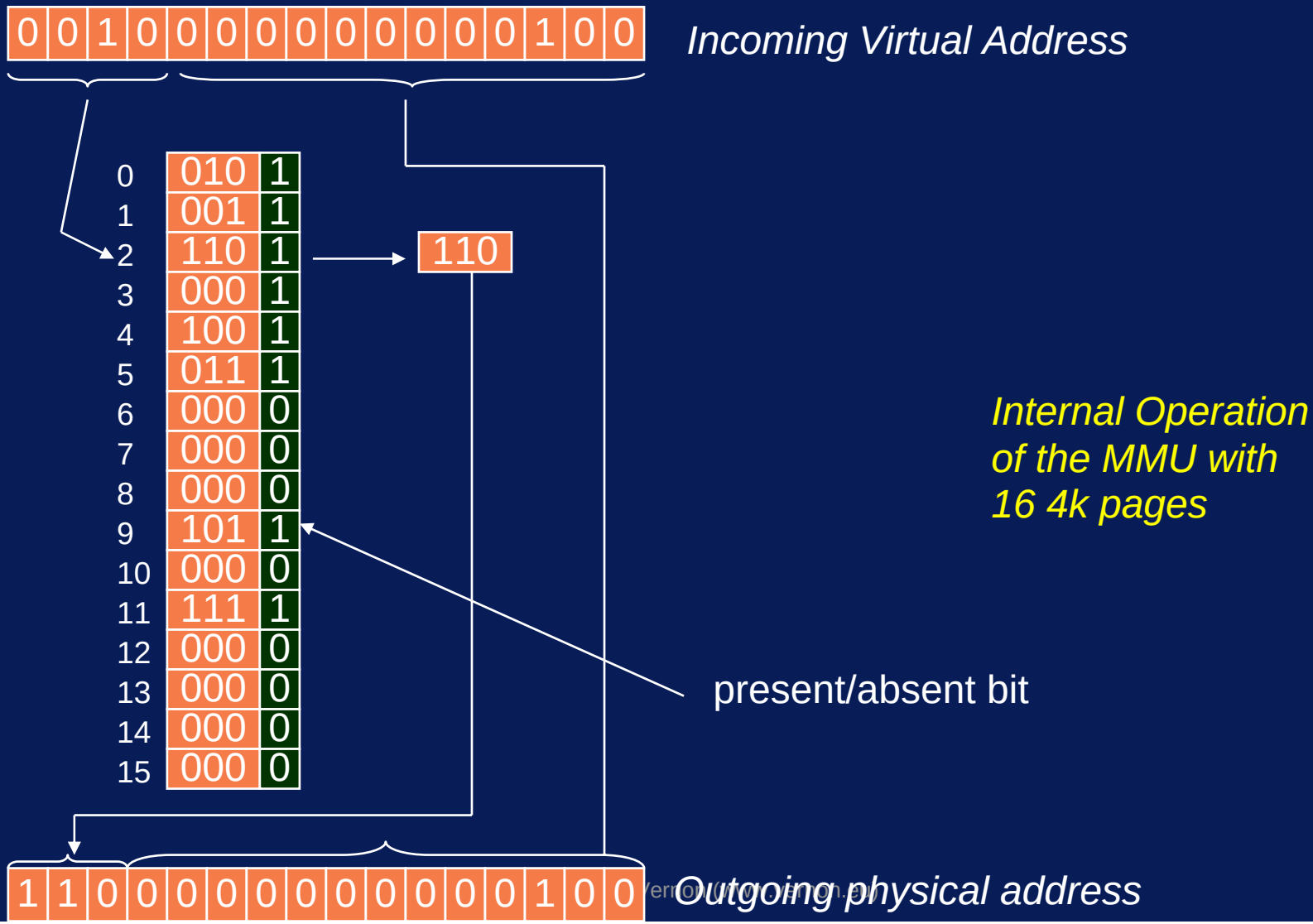
Memory Management

- Virtual Memory
  - Paging; Scenario 2
    - » virtual address is not mapped to the physical memory (hence the page must be on disk)
    - » The required page is swapped into a page frame of a little-used page in physical memory as follows:

Memory Management

- Virtual Memory
  - Paging; Scenario 2
    - MMU check the present/absent bit in the page table entry
    - Page is not pesent so CPU traps to the OS - this trap is called a page fault
    - OS finds a little-used page frame and writes its contents back to disk
    - It then fetches the page just referenced into the page frame just made available
    - It updates the map
    - Finally, it restarts the trapped instruction

Memory Management



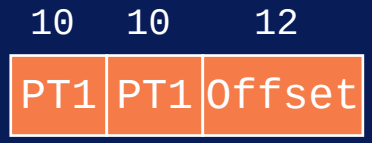
Internal Operation of the MMU with 16 4k pages

Memory Management

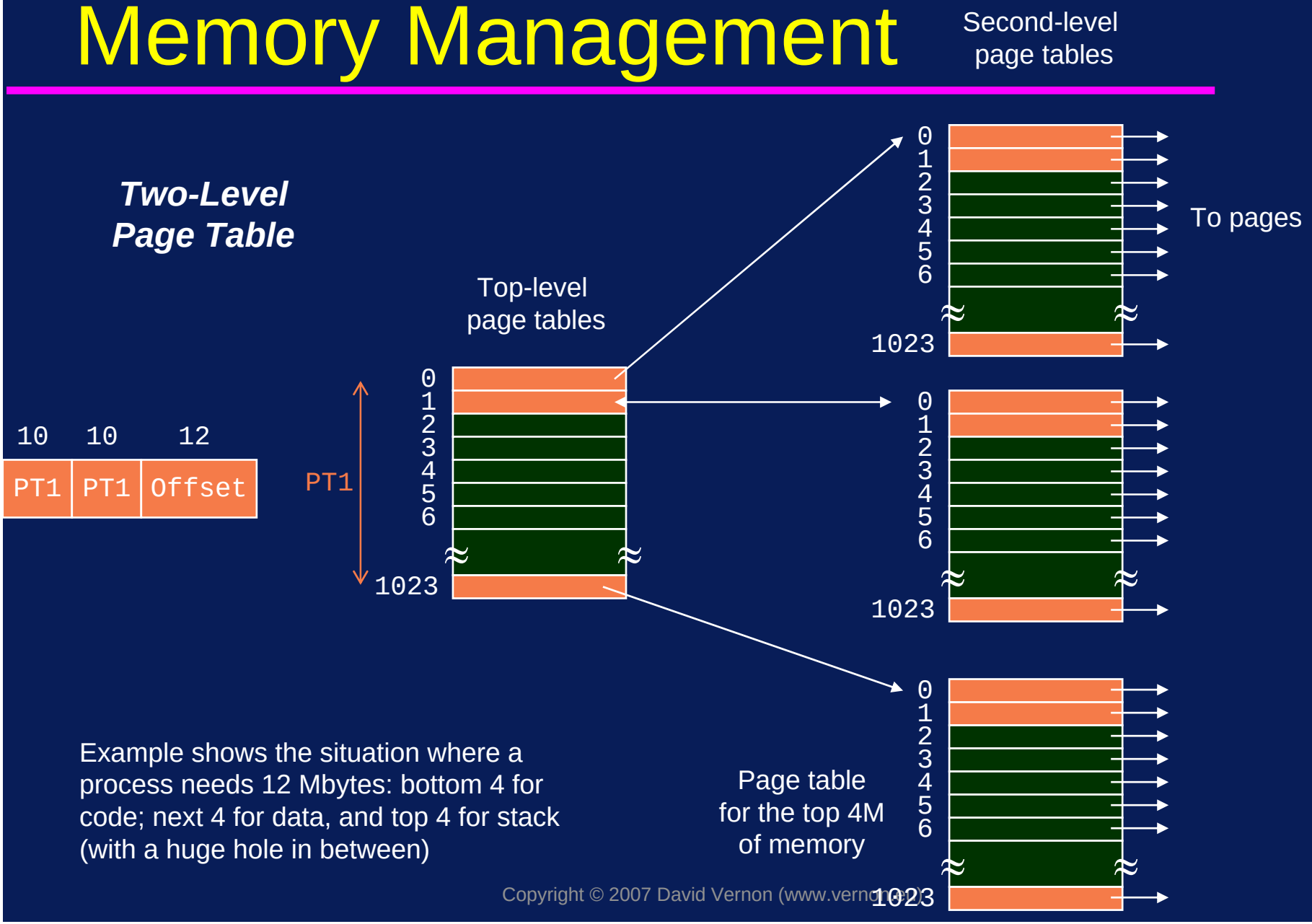
- Virtual Memory
  - Issues with page tables
    - » the page table can be extremely large (consider the case of a 32-bit virtual address space: how many 4k page table entries are required? What about a 64-bit space?)
    - » The mapping must be fast (Why? A virtual-to-physical mapping must be made for every memory reference)

Memory Management

- Virtual Memory
  - Multilevel Paging
    - » This is a solution to the problem of having all of a huge page table in memory all of the time
    - » Instead of a single page table we have many tables which are organized hierarchically
    - » Consider a 32-bit virtual address
      - 10-bit PT1 field
      - 10-bit PT2 field
      - 12-bit Offset field
      - Pages are 4k (why?)
      - There are 2^{20} of them (why?)



Memory Management



Memory Management

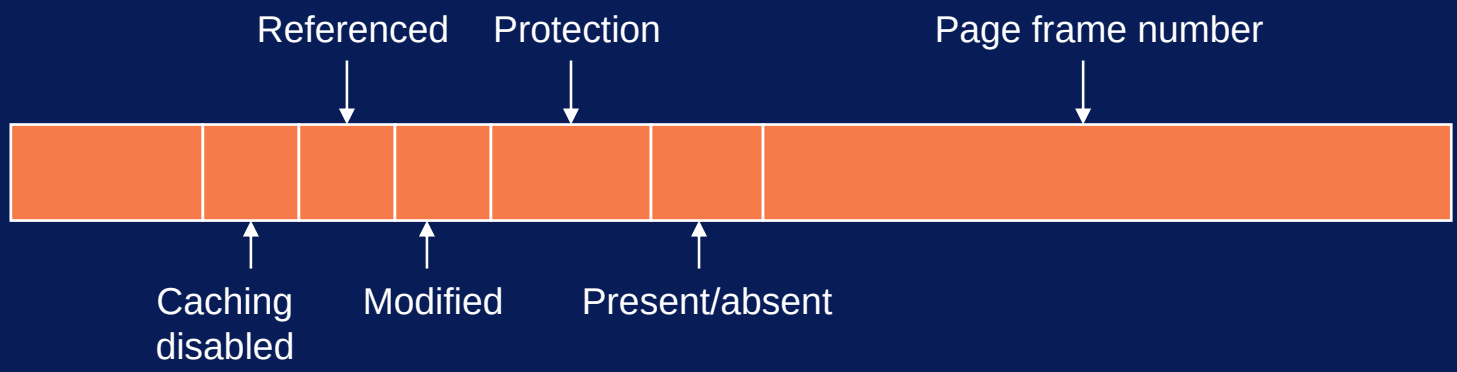
- Virtual Memory
  - Multilevel Paging
    - » PT1 indexes top-level page table (1024 entries)
    - » Only entries 0, 1, and 1023 point to the second-level page tables
    - » which in turn point to the physical memory for code, data, and stack, respectively
    - » Even though there are over a million pages (2^{20}) and a thousand page tables, only 4 are needed.
    - » We can also have 3-level and 4-level paging (but rarely worth doing it for more than 3-level)

Memory Management

- Virtual Memory
  - Multilevel Paging
    - » Example: 32-bit virtual addresss 0x00403004
    - » PT1 is:
    - » PT2 is:
    - » Offset is:
    - » Which entry contains the page frame number of the virtual address?
    - » What happens if the present/absent bit is zero in this page table entry?
    - » What happens if it is set?
    - » Where does the offset come in?

Memory Management

- Virtual Memory
  - Detailed structure of a page table entry
    - » exact structure is highly-machine dependent
    - » but most machine use the same information
    - » size: 32-bits is common



Memory Management

- Virtual Memory
  - Detailed structure of a page table entry
    - » Page frame number
    - » Present/absent bit:
      - 1=> valid & can be used
      - 0 => page not in memory; access will cause a page fault
    - » Protection bit(s)
      - 1 bit => 0 for read/write; 1 for read only
      - 3 bits => read, write, execute permissions
    - » Modified bits
      - 1 => page written to; useful if need to swap a page to disk (need to write if modified; no need if not)

Memory Management

- Virtual Memory
  - Detailed structure of a page table entry
    - » Referenced bit
      - 1 => page accessed; useful for page replacement algorithms (see later).
    - » Caching bit
      - 1 => disable caching
      - important if device registers are memory-mapped
      - disabling forces access to the physical register, not just the cache

Memory Management

- Virtual Memory
  - Detailed structure of a page table entry
    - » Note that the disk address used to hold the page when it is not in memory is not held in the page table entry
    - » Why?
    - » Page table only facilitates translation of virtual addresses to physical addresses
    - » If there is a page fault, then the OS uses software tables to identify the disk address, resolve the page fault, and swap the pages between page frames

Memory Management

- Virtual Memory
  - The PDP-11 uses 1-level paging
  - The VAX uses 2-level paging
  - The SPARC uses 3-level paging
  - The 68030 uses 4-level paging
  - See Tanenbaum for details

Memory Management

- Virtual Memory
  - In most paging schemes, the page tables are kept in memory (rather than in registers)
  - This means that a single logical memory access may require several additional memory accesses (to the page tables and to the final physical page frame)
  - This applies both to instruction fetches and data movements
  - The impact on the execution performance can be very significant (e.g. 25% of full capacity)

Memory Management

- Virtual Memory
  - The solution is to use a device called associative memory (sometimes called a lookaside buffer)
  - Based on the observation that most programs tend to make a large number of references to a small number of pages (and not a small number of references to a large number of pages)

Memory Management

- Virtual Memory
  - The associative memory comprises up to 32 entries which are identical to the page table entries
  - When translating a virtual memory address to a physical address, the MMU first checks to see if its virtual page number is present in any of the associative memory entries (and it check all of them in parallel)
  - If it is, the page frame is taken from the associative memory

Memory Management

- Virtual Memory
  - If it isn't:
    - » the page table is accessed in the normal way and the page frame is retrieved
    - » one of the associative memory entries is deleted
    - » and the page table entry of the page just accessed in inserted instead
    - » Thus, if the same page is accessed again, the associative memory will now 'hit'

Memory Management

- Virtual Memory
  - Page Replacement Algorithms
    - » If a page fault occurs, the OS has to choose a page to remove from memory (i.e. from the from a page frame) to make room for the the page to be brought in.
    - » If the page to be removed has been modified, it has to be written to disk (otherwise it doesn't)

Memory Management

- Virtual Memory
  - Page Replacement Algorithms
    - » The Not-Recently-Used Page Replacement Algorithm
      - Periodically (e.g. every clock interrupt), reset the Referenced bit of the page table entry
      - On a page fault, check the Referenced and Modified bits of the page table entry
        - Class 0: not referenced, not modified
        - Class 1: not referenced, modified
        - Class 2: referenced, not modified
        - Class 3: referenced, modified
      - Remove a page at random from the lowest numbered non-empty class

Memory Management

- Virtual Memory
  - Page Replacement Algorithms
    - » The First-In, First-Out Page Replacement Algorithm
      - maintain a FIFO list (a queue) of all pages in memory
      - queue new pages
      - dequeue pages to be removed
      - Not often used

Memory Management

- Virtual Memory
  - Page Replacement Algorithms
    - » The Second-Chance Page Replacement Algorithm
      - similar to the FIFO algorithm but check the Referenced bit
      - REPEAT
        - dequeue the page
        - if the Referenced bit is set, clear the Referenced bit, and enqueue it
        - if the Reference bit is not set, remove the page
      - UNTIL a page with a clear Referenced bit is dequeued
      - Guaranteed to terminate. Why?

Memory Management

- Virtual Memory
  - Page Replacement Algorithms
    - » The Clock Page Replacement Algorithm
      - similar to the Second Chance algorithm except use a circular queue and, instead of dequeueing and enqueueing, move the queue head pointer

Memory Management

- Virtual Memory
  - Page Replacement Algorithms
    - » The Least Recently Used (LRU) Page Replacement Algorithm
      - Remove the page that has been unused for the longest time
      - Expensive (in cost) to implement in hardware
      - Expensive (in time) to simulate in software

Memory Management

- Segmentation
  - The virtual memory systems discussed so far are 1-Dimensional and linear with a single virtual address space
  - Such a system requires the programmer to manage the allocation of memory effectively, especially if dealing with large arrays (tables)
  - Segmentation provides a very general alternative

Memory Management

- Segmentation
  - Provide the machine with many independent address spaces: SEGMENTS
  - Each segment consists of a linear sequence of addresses (0 to some maximum)
  - Different segments can, and usually do, have different lengths
  - Segment lengths can change during program execution and they can do so independently of one another

Memory Management

- Segmentation
  - To specify an address in a segmented (or 2-Dimensional) memory, the program must supply a two-part address:
    - » a segment number, and
    - » an address within the segment
  - A segment is a logical entity of which the programmer is aware and uses as such
  - A segment usually contains data of one given type (program code, array, stack, variables)
  - It does not usually mix types

Memory Management

- Segmentation
  - It also eases problems with linking separately compiled functions
    - » Let each function occupy a different segment, with address 0 as its starting address
    - » A function call requires the two-part address (n,0) where n is the segment number
    - » If the function in segment n is modified and recompiled, changing its size, we don't need to alter any of the calling routines

Memory Management

- Segmentation
  - » On the other hand, if we were using a normal linear memory model, the modified function would have caused a displacement of other functions (remember, it changed its size and now occupies a difference amount of memory) and, hence, they will all have different start addresses and their calling functions will have to be re-linked.

Memory Management

- Segmentation
  - It also facilitates sharing functions in a shared library
    - » Commonly-used functions (e.g. the graphical user interface functions of a window system) can be put in a segment and shared by processes
    - » Without having to have the functions included in the address space of every process which uses them
    - » This can be done on a paged system, but it is much more difficult

Memory Management

- Segmentation
  - It also facilitates protection
    - » Different segments (and hence different types of data) can have different types of protection associated with them
      - a function segment could be execute-only
      - an array might be read & write only, not execute
    - » attempts to execute the array segment can then be trapped
    - » All this works because the programmer is aware of what is in each segment
    - » This isn't the case in a paged system

Memory Management

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was the technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection