2026 CS 332 Subject 2

NJIT

New Jersey's Science & Technology University

THE EDGE IN KNOWLEDGE

Alex. Gerbessiotis

CS Department

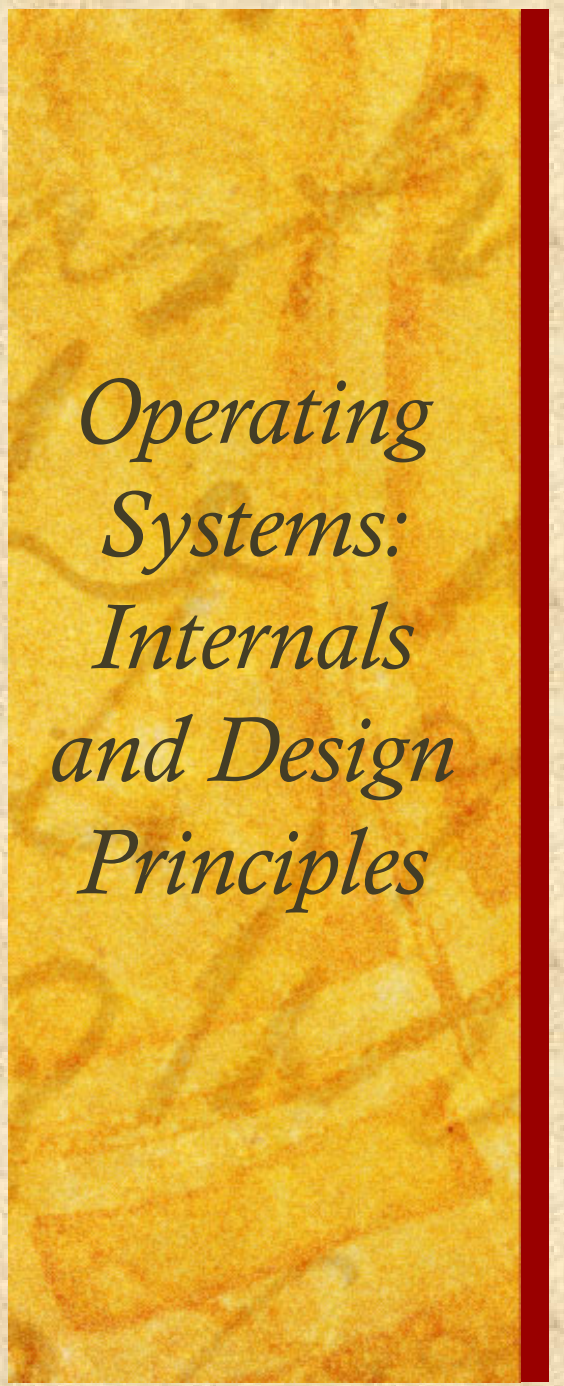
The yellowish/brownish notes are copyright the textbook author.

The bluish notes are copyright the instructor and are add-ons and provide more explanation or additional details

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Module 1

© 2019-2027. Alex. Gerbessiotis. All rights reserved.



Chapter 2 Operating System Overview

Ninth Edition

By William Stallings

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Operating System

- A program that controls the execution of application programs
- An interface between applications and hardware

Main objectives of an OS:

- Convenience
- Efficiency
- Ability to evolve

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

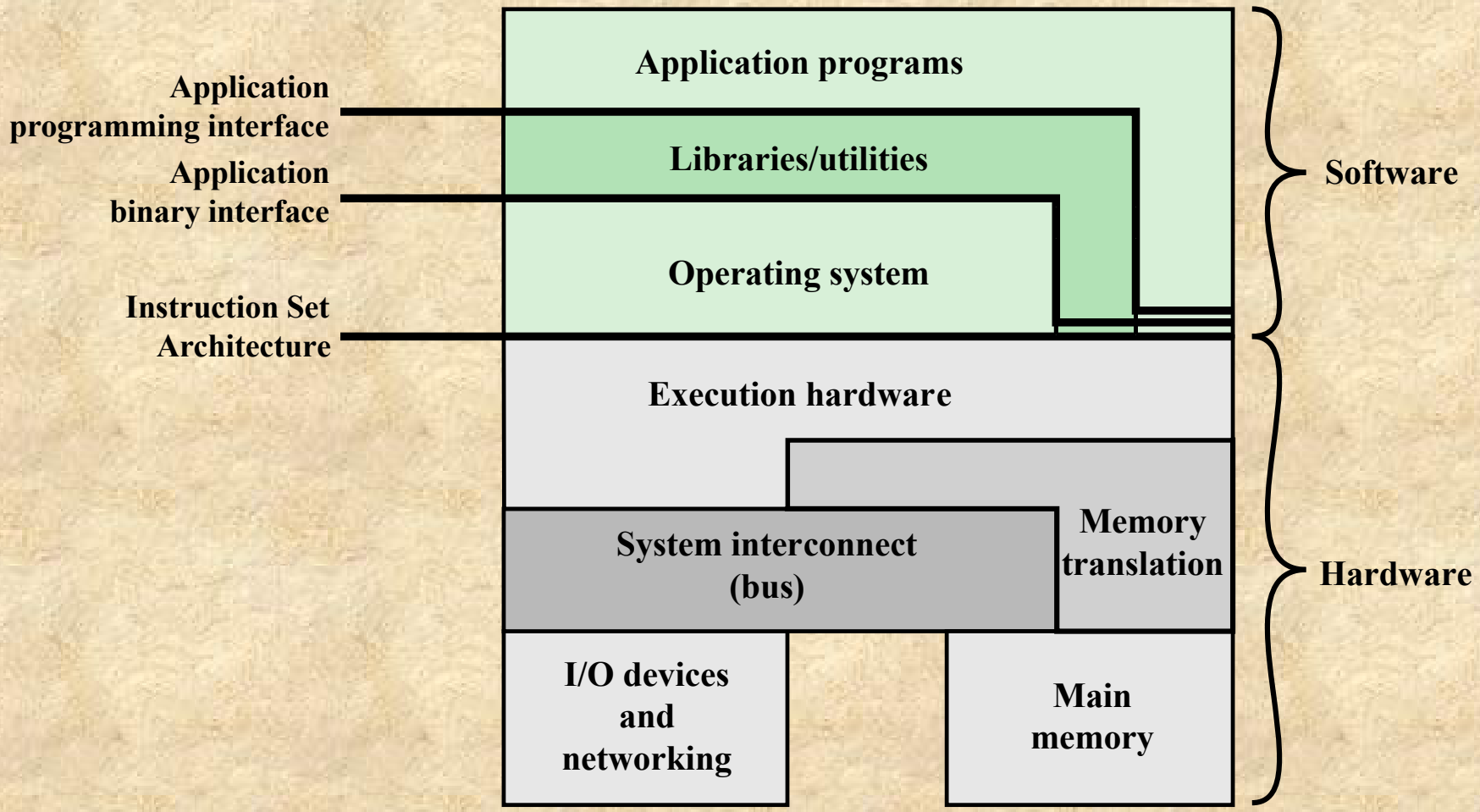


Figure 2.1 Computer Hardware and Software Structure

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Operating System Services

- Program development
- Program execution
- Access I/O devices
- Controlled access to files
- System access
- Error detection and response
- Accounting

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

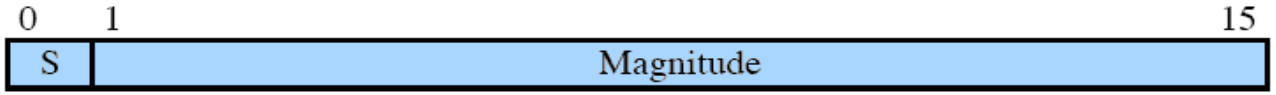
Key Interfaces

- Instruction set architecture (ISA)
- Application binary interface (ABI)
- Application programming interface (API)

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

"ISA" for textbook

(a) Instruction format



(b) Integer format

Program counter (PC) = Address of instruction
Instruction register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

(c) Internal CPU registers

0001 = Load AC from memory
0010 = Store AC to memory
0101 = Add to AC from memory

(d) Partial list of opcodes

Figure 1.3 Characteristics of a Hypothetical Machine

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

ABI

ABI: How to call functions , which registers to use

ABI: Parameter passing through registers

ABI: Parameter passing through stack

```
int f(int x, int y, int z) {
  int b,c;
        ...
}
```

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

API

For example FILE I/O in Unix/Linux

- lseek
- creat
- open
- close
- read
- write

Run programs

- fork
- exec

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

The Operating System as Resource Manager

- The OS is responsible for controlling the use of a computer's resources, such as I/O, main and secondary memory, and processor execution time

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Evolution of Operating Systems

- A major OS will evolve over time for a number of reasons:

Hardware upgrades

New types of hardware

New services

Fixes

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Evolution of …OS…names

Alex. Gerbessiotis
COMPUTER SCIENCE

Monitor Program: (Resident Program): A main memory resident program that is a primitive form of an operating system that always is in main memory. Originally it was the interrupt handling routine (interrupt monitor) The term was used in the early era of computing

OS: The development of the original monitor program into what is known as an operating system

Kernel: The part of OS most frequently-used (including its data structures) that stays in (main) memory all the time

An early OS (Monitor) was by default a kernel as it was always memory resident

Example 1: The Linux OS consists of the Linux Kernel plus other components that are part of the Linux Operating System.

Example 2: The Android OS consists of the Linux Kernel plus other services that are unique into what is known as Android.

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Kernel

Monolithic Kernel: A single kernel containing all kernel functionality

Microkernel: Many smaller kernels providing the services of a monolithic kernel; each microkernel providing through system calls specific OS services in kernel space (def on next slide).

Example 1. Mach OS was/had a microkernel. Apple OS was based on the Mach OS

Example 2. Windows NT was also based on Mach OS but provided Windows functionality

Example 3. Windows 10 is not a microkernel...

Example 4. Linux's kernel is a monolithic kernel. (Linux has some modularity with its kernel module though.)

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

User and System Functions

Privileged instructions or code:

(a) system function (calls) : eg printf, cout, println
(b) kernel code
(c) interrupt activity
(d) I/O device drive code (sensitive code)
etc
AND their DATA (includes registers)
utilize special memory area that is
privileged (protected) space and known as
monitor space
system space
kernel space
Such code is executed in
privileged mode
also known as
monitor mode, system mode, or
kernel mode
i.e.privileged execution involved and CPU is in privileged mode (Intel uses the term ring level to describe it with privileged mode being ring level 0 or lower)

Kernel stack is used for function calls and function variables

Unprivileged instructions or code

(a) User function (calls)

or in general any instruction or code that is NOT privileged,

utilize regular memory area that is
unprivileged (unprotected) space and known as
user space

Such code is executed in
unprivileged mode
also known as
user mode

CPU is in user mode during that execution (Intel ring level 3)

User stack is used for user function calls and user function variables

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Context switches

Subject 1 type of context switch(es)

User Program A to Interrupt Handling Code

and

Interrupt Handling Code to User Program A

Subject 2 type of context switch

includes Subject 1 type of context switches i.e.

and it is more general

User Program A (user mode) to OS (system mode)
OS (system mode) to User Program A or B (user mode)

In general context switches involve mode switches

user mode to system mode
system mode to user mode

Example: ABC is CPU execution of three user programs (task or jobs) A, B and C.
This actually means
OS->A->OS->B->OS->C->OS a minimal of six context switches.
If A is doing I/O for example more context switches are there because of I/O (interrupts)

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

User mode to Kernel mode


```
myfunction()                  //User program
 FILE *fp;                    //user mode
 fp = fopen("myfile","rw");   //user mode ; user stack
....
}
FILE *fopen(...) {            // System Function Call
   KernelModeSwitch           // switch into kernel mode
    Retrieve user  data       // register or user stack copied
   return value in a register  / to kernel stack
   say  AC ( Intel"s AX)      // still in kernel mode
   UserModeSwitch             //  switch into user mode
  return                      //  user mode; ready to use
}                             //  user stack
```

When the fopen("myfile","rw") line is executed in the user program the parameters "myfile" and "rw" are pushed into the user stack along with address of fp (return value) and the address of the next instruction in user program

Next the user program CALLS the address of fopen and alerts OS for a privileged instruction invocation.

Kernel/OS takes over until execution is complete, and then do we enter user mode (by the Kernel/OS that releases the privileged execution).

Types of context switches so far

Subject 1: User Program A to Interrupt(i.e. OS), and Interrupt (i.e. OS) to User Program A

Subject 2: See above or
User Program A to OS, and OS (other than Interrupt) to User Program A
or
User Program A to OS, and OS to User Program B (later in Subject 2)

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Job, Program, Task

Job, Program, Task:	**To be used interchangeably to mean same thing. But**
Code:	Part of a program or the source code that generates an executable program or component of.
Program:	**A (user) computer program in executable (machine code) form** (eg payroll program). Short for executable program. Resides on disk
Task:	**A (computer) program along with input files required to perform a certain task.** (eg payroll program plus data files to generate payroll for a given period)
Job:	Implies the use of a Job Control Language (JCL), a modern day script language, to describe for example how source code of program is to be compiled or how resulting machine code (executable) is to be used with data files to perform a task or both (aka script)
Tasks and Job	We might say a Job consists of multiple tasks!

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Program vs 'Program in execution'

Program: Resides on disk. OS does not know about it.

Program in execution: Resides in main memory (all or some parts of it) and OS(kernel) knows about it.

It is described by text+data + register file (of current execution) aka program state plus all the info available to the kernel re the program in execution.

Will be defined as '**process**' later on.

Long term scheduler: (Job scheduler) Picks a program from SM and move it to MM to execute it. Available in Batched systems, MultiProgramming and Timesharing systems.

Short term scheduler: (CPU dispatcher) Picks a program in execution (process) and run it on the CPU next. Used in multiprogramming and time sharing and multitasking. Increasing complexity from former to latter.

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

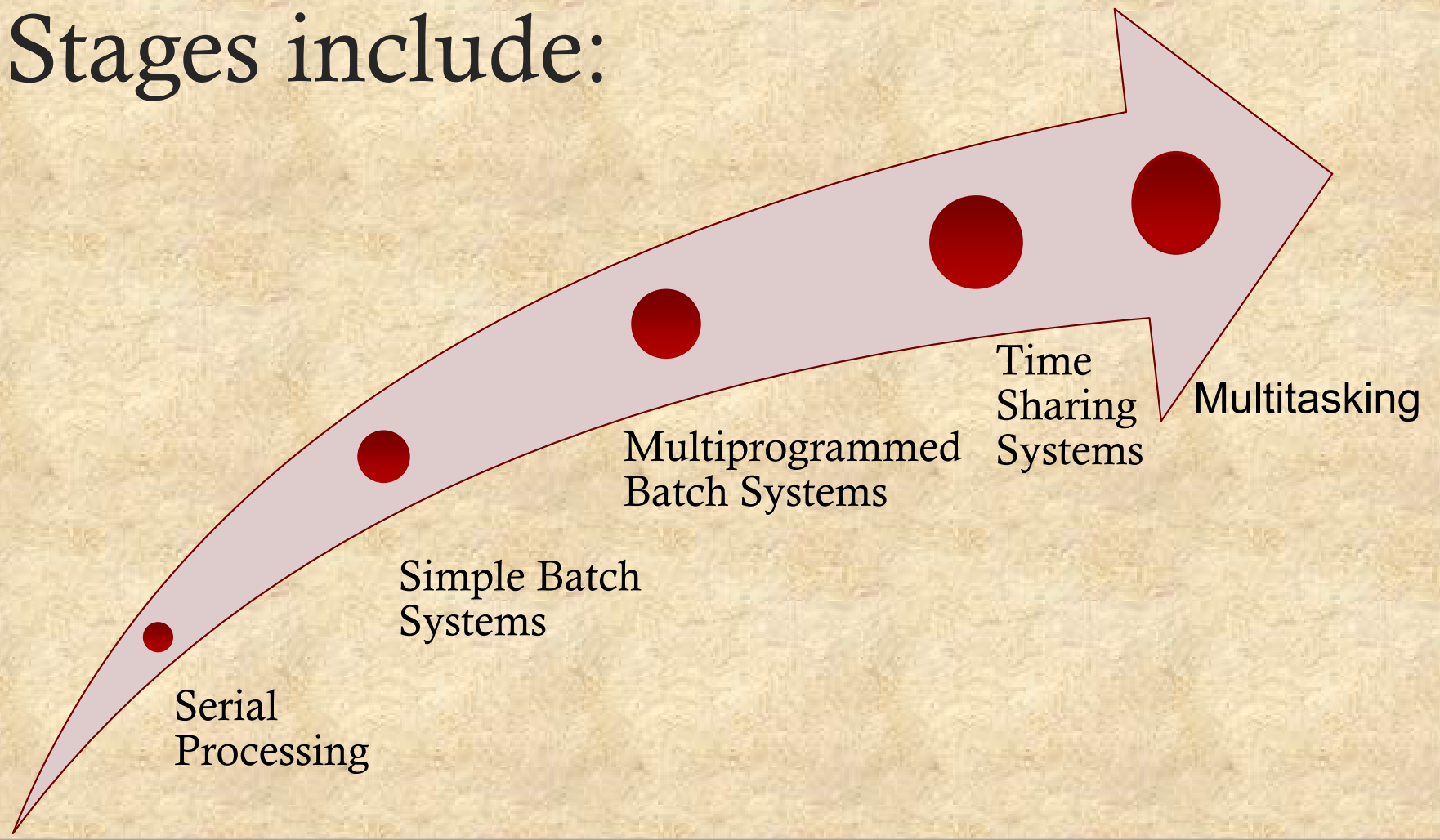
Terms to be used

Arrival time:	Time a program/job/etc arrives in a computer system/CPU and becomes a process!
Start time:	Time CPU starts executing an instruction of process
Finish(Completion) time:	Time it completes all of its execution (CPU and I/O)
Turnaround time(Tr):	Amount of time a job is in the system Tr = CompletionTime - ArrivalTime
Note:	ArrivalTime is not necessarily t=os. Watch out!
Response time:	Time by CPU to respond to program/job/etc since arrival ResponseTime = Start time – Arrival time
Service time (CPU time):	Amount of time a job is to use the CPU
I/O time:	Amount of time a job is waiting for I/O
Execution time:	Amount of time a job has used the CPU so far. Execution time <= Service time
Mean Turnaround Time: (MTT)	Average Tr for all jobs/tasks/etc involved.
CPU utilization:	Percent of time CPU is busy executing a program/job/task
Throughput:	Number of job completed in a unit of time
CPU-bound job:	Job is using primarily the CPU
I/O-bound job:	Job is doing primarily I/O

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Evolution of Operating Systems

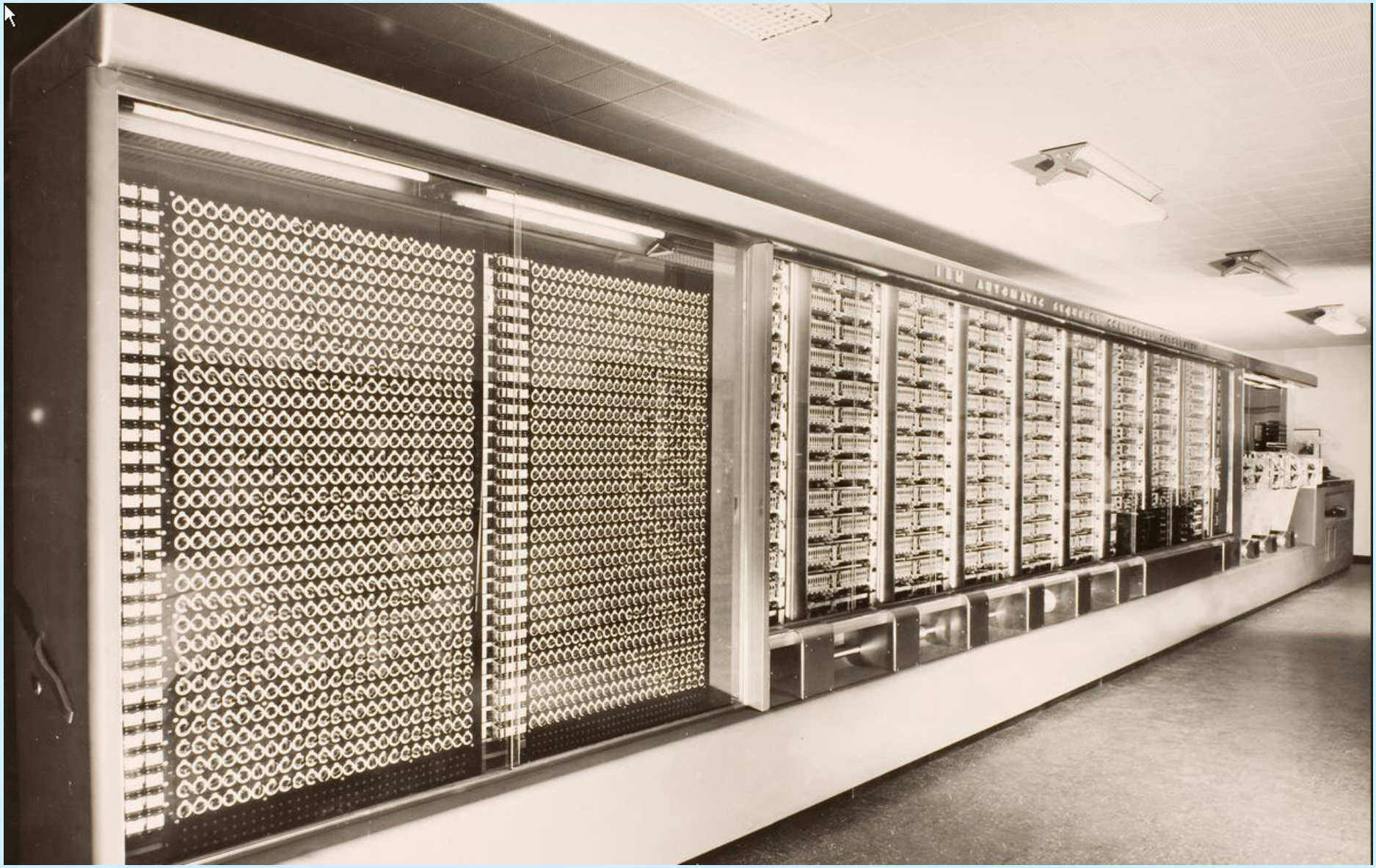
- Stages include:



© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

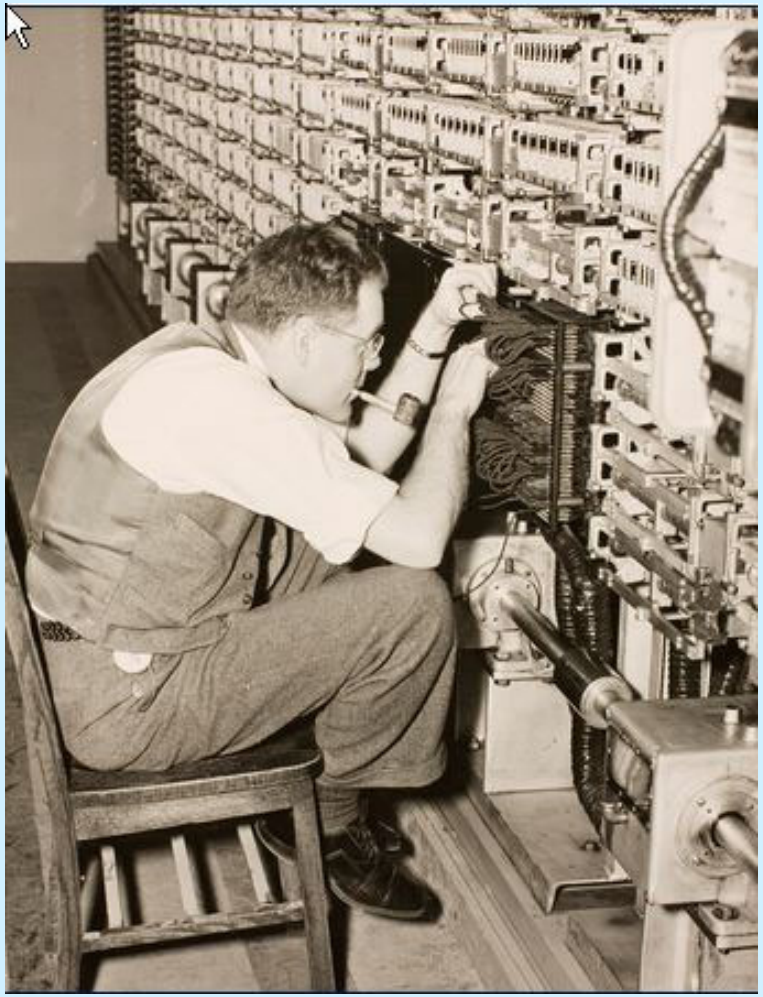
Harvard Mark I (1944)

sites.harvard.edu/~chsi/markone/crew.html



© 2019-2027. Alex. Gerbessiotis. All rights reserved.

sites.harvard.edu/~chsi/markone/crew.html



Harvard Mark I

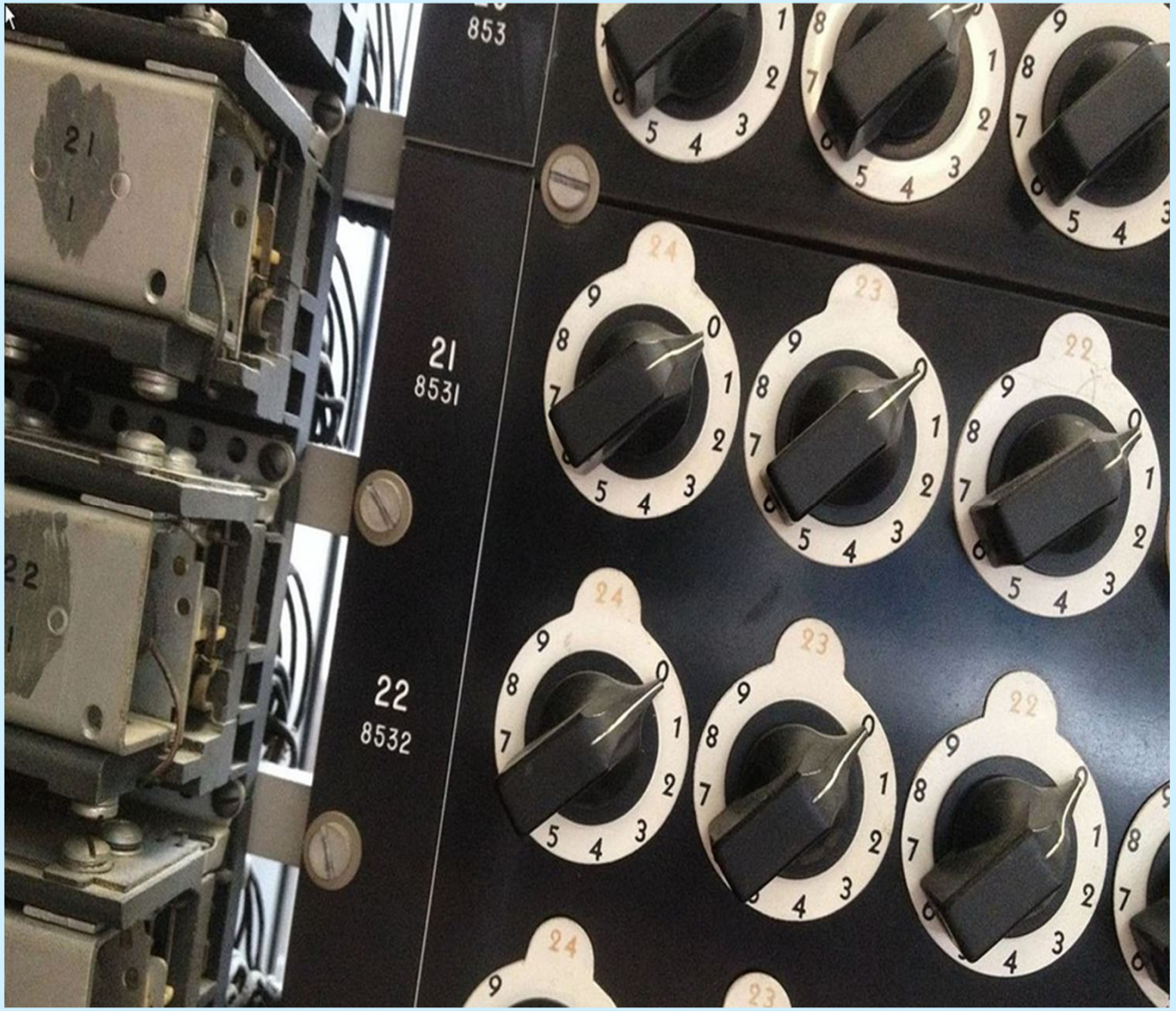
Grace Murray Hopper (seated, second from right) and Howard Aiken (seated, centre), along with other members of the Bureau of Ordnance Computation Project, in front of the Harvard Mark I computer at Harvard University, 1944.

U.S. Department of Defense

Debugging ?

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Havard Mark I : I/ of I/O



© 2019-2027. Alex. Gerbessiotis. All rights reserved.

1. Serial Processing

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Serial Processing

Earliest Computers:

- No operating system
  - Programmers interacted directly with the computer hardware
- Computers ran from a console with display lights, toggle switches, some form of input device, and a printer
- Users have access to the computer in "series"

Problems:

- Scheduling:
  - Most installations used a hardcopy sign-up sheet to reserve computer time
    - Time allocations could run short or long, resulting in wasted computer time
- Setup time
  - A considerable amount of time was spent on setting up the program to run

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

1.Serial Processing

Serial processing or Serial computing (or UniProgramming):

A user or job has exclusive access to a computer system and all of its resources.

SerialProcessing-1 (by hand 1940s-1950s). Have: Nothing (bare metal)

Programmer is Sysadmin. Goes into room (reservation 9am to 10am) and spends time to load data (eg 30 + 30 rows of Harvard Mark I). Code is in the form of punched paper. When done with 60 numbers, programmer executes program by pressing a button.

Output through a telegraphic printer (long narrow paper i.e. one long line of output) slow and blocks CPU (programmed I/O also known as blocked I/O) or inspection of memory!

Most of the time wasted in computer preparation.

(In Harvard Mark I only data in main memory ; program was in punched tape.)
Computer idle time: ~1hour because of preparation

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

SerialProcessing-1

SerialProcessing-1: User A books system 9-10am. **Have: nothing but computer**
Program A is 10s CPU, 10s I/O;

CPU utilization: 10s/1hr = 10s/3600s ~ 0%
Throughput: 1job/hour
No need to deal with turnaround time

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Serial Processing-2

SerialProcessing-2 (by punch cards to CPU directly). **Have: Punch cards**

Programmer prepares program in punched cards and also the program!
(Von-Neumann architecture now or maybe.)
Submits cards to a SysAdmin who feeds cards into a punch card reader that initializes computer accordingly. Execution follows.

Computer idle time: Several minutes to read punched cards (I/O) and set computer.

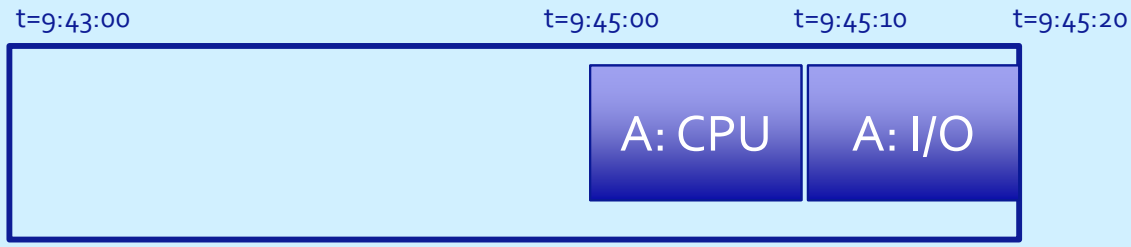
© 2019-2027. Alex. Gerbessiotis. All rights reserved.

SerialProcessing-2

SerialProcessing-2: User A arrives at 9:43:00 sharp and gives immediately punched cards and paper to Assistant sysadmin. Computer on standby. Assistant takes cards to computer room and feeds them into computer. Processing starts at 9:45:00. (Appointment time was 9:43:00)
Program A is 10s CPU, 10s I/O.

CPU utilization: 10s/2min+20s = 10s/140 ~ 8%

Throughput: ~20job/hour



© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Serial Processing-3

SerialProcessing-3. Have: Punch cards, SM (HDD or Tape)

Programmer prepares data and program

(a) in punched cards (binary code aka executable aka exec)

or

(b) in punched card have source code in in a higher-level language (eg assembly, then Fortran, then Cobol)

or

(c) submits a tape with the exec as in case (a) or code as in case (b).

Eventually exec gets transferred into a magnetic tape drive (or HDD). [It is possible that a (separate) computer will compile+assemble source code into exec then into Tape or HDD.] (Main) computer might be busy with other programs.
Sysadmin, when (Main) computer gets idle, transfers program from (HDD, Tape) into MM.

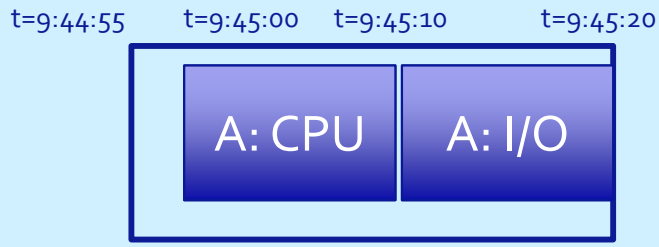
Exec program is brought into main memory in ~10s (tape drive) or 0.2s (1960s HDD).
Computer idle time: few to no more than one second because of data transfer

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

SerialProcessing-3

SerialProcessing-3: Arrival time of user is irrelevant. Loading of the exec from SM(tape drive or HDD) into the CPU is relevant. In 1950s to 1960s it could take from one sec (HDD) to a couple of seconds(tape drive) depending on exec size

CPU utilization: 10s/25 = 40%
Throughput: ~144job/hour



© 2019-2027. Alex. Gerbessiotis. All rights reserved.

SerialProcessing review

SerialProcessing-5 : Modern day view, primitive OS style.

Two programs A and B are to use a computer

A is 10s CPU and 10s I/O afterwards

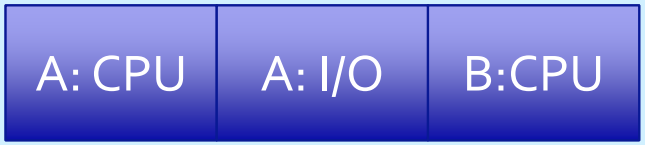
B is 10s CPU only.

Arrival times t=0s for A and B (arrival time usage fluid)

Schedule : A B A in system 0-20s; B in system 20-30s

Tr(A)=20s Tr(B)=30s ;Tr(<A,B>) = 30s

MTT : (20+30)/2 =25s



Schedule : B A B in system 0-10s; A in system 10-30s

Tr(A)=30s Tr(B)=10s Tr(<B,A>)=30s

MTT : (30+10)/2 = 20s

CPU utilization (either schedule) : 20s/30s = 66%

Throughput (either schedule) : 2jobs/30s or 4 jobs/minute or 240jobs/hour

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Serial Computing limitations

1. CPU utilization in previous example stale at 66%
2. A runs then B or B runs then A are only possibilities.
3. No matter what CPU (processor) utilization stays at 66%

Next evolution: B uses CPU while A is doing I/O (multiprogramming)

Multiprogramming would be facilitated by

Batch systems (processing)

Requirements: A primitive OS (monitor) in addition to interrupt handling

Next: Batch systems

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

2. Simple Batch Systems

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Simple Batch Systems

- Early computers were very expensive
  - Important to maximize processor utilization
- Monitor
  - User no longer has direct access to processor
  - Job is submitted to computer operator who batches them together and places them on an input device
  - Program branches back to the monitor when finished

Simple Batch System support (by an OS, by the CPU architecture, by Interrupts) will make the gap small or negligible...

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Batch Systems

Batch Systems Narrative

Computers are expensive but CPU (Processor utilization) is low in early computers

Several programs (of the same or multiple users) arrive at the same time (in punch cards, or tapes)

Computer has the ability to store programs in SecondaryM (SM), supports interrupts and interrupt handling and in later years has enough main memory to store in MM several programs.

A job control language is available (primitive scripting language) and Computer's 'Monitor Program' (primitive OS) can do elementary sequencing (which program to run from a batch of programs)

Objective: **Improve Turnaround time and Throughput**

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Batch Systems

Batch Systems Narrative

JOB CONTROL LANGUAGE (JCL)

Direct Exec Input form

```
user      tom
time      10h/month
CPU       20s
program A
start  text 5000
Line 1
…
Line 32: Last instruction
end text
start data at 6000
2
3
0
remaining 57 value 0
end data
```

Source code Input form

```
user tom
time 10h/month
CPU  60s
program B
start assembly PC=7000
LOAD AC,  8000
ADD  AC,  8001
STORE AC, 8002
end assembly
start data address-value
8000  2
8001  3
8002  0
remaining 57 value 0
end data
```

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Job Control Language (JCL)

Special type of programming language used to provide instructions to the monitor

What compiler to use

What data to use

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Batch Systems

Batch Systems Support

OS: Rudimentary OS available in the form of a Monitor (the computer)

Separation of Space:

- Monitor Space for Monitor Program (e.g. address space 0-4999)
- User space for User program (e.g address space 5000 and higher for one or more)

Separation of Execution

- Monitor Space code execution in monitor mode (privileged)
- User Space code execution in user mode (unprivileged)

Long Term Scheduler (Job Scheduler)

- Does Job Sequencing: Which program to bring into MM from SM

Loader (plays also role of CPU Dispatcher)

- Load program from SM into MM on direction by Long Term Scheduler

Sequencing functionality

- Long term Scheduler + Loader functionality

Job Control Language interpreter

- Separate punch cards' text , data , etc from JCL commands

Other (later years)

- Assembler to convert assembly into executable (e.g. program B)
- Compiler to convert source code (Fortran, Cobol) into executable

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Monitor Point of View

- Monitor controls the sequence of events
- *Resident Monitor* is software always in memory
- Monitor reads in job and gives control
- Job returns control to monitor

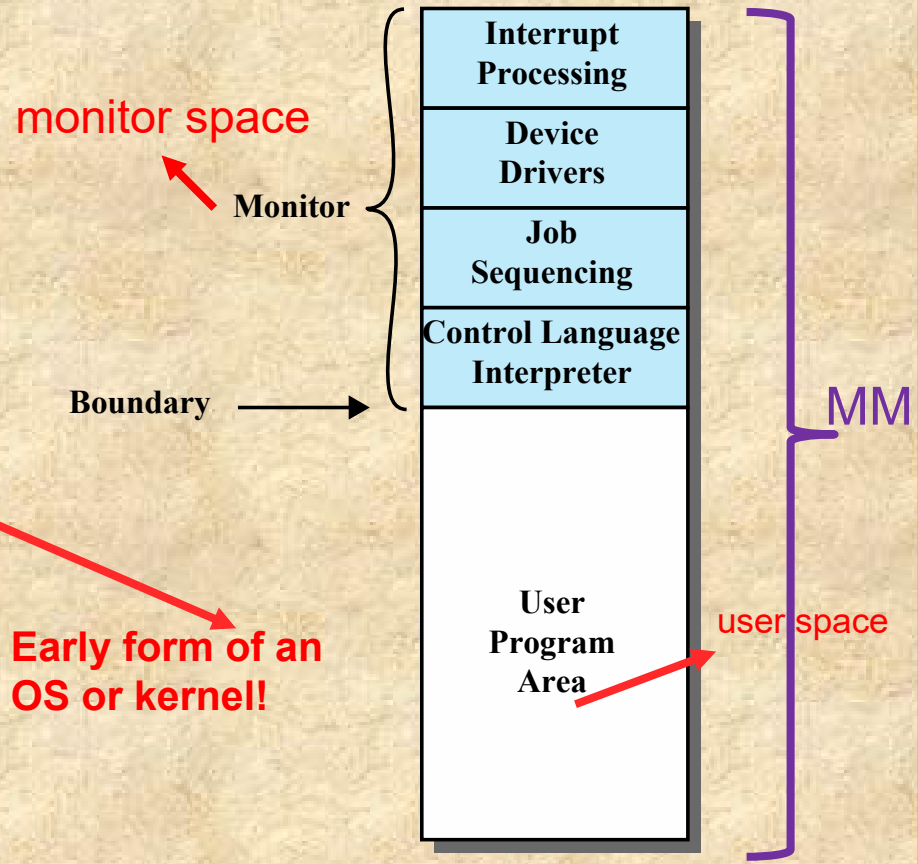


Figure 2.3 Memory Layout for a Resident Monitor

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Batch System Support

Batch Systems Support

Separation of Space: **MEMORY PROTECTION**

NOTE: Still serial execution.

Separation of Execution: **PRIVILEGED INSTRUCTIONS + MEMORY PROTECTION**

Job Scheduler, Loader **TIMER + INTERRUPTS + ABOVE**

Memory Protection — **User space vs Monitor Space**

Timer — **Monitor monitors user activity and terminates violating programs. OR user program is given a certain amount of time to use CPU**

Privileged instructions — **Privileged-mode: Monitor mode (aka kernel mode, system mode)**

Interrupts — **context switch concept**
Interrupt-based I/O increases CPU utilization

Context switch — **switch between user mode and monitor mode and other way around. OVERHEAD**

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Desirable Hardware Features

Memory protection

- While the user program is executing, it must not alter the memory area containing the monitor

Timer

- Prevents a job from monopolizing the system

Privileged instructions

- Can only be executed by the monitor

Interrupts

context switch

- Gives OS more flexibility in controlling user programs

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Modes of Operation

User Mode

- User program executes in user mode
- Certain areas of memory are protected from user access
- Certain instructions may not be executed

Kernel Mode

- Monitor executes in kernel mode
- Privileged instructions may be executed
- Protected areas of memory may be accessed

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Simple Batch System Overhead

- Processor time alternates between execution of user programs and execution of the monitor
- Sacrifices:
  - Some main memory is now given over to the monitor
  - Some processor time is consumed by the monitor
- Despite overhead, the simple batch system improves utilization of the computer

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

BatchSystems-1

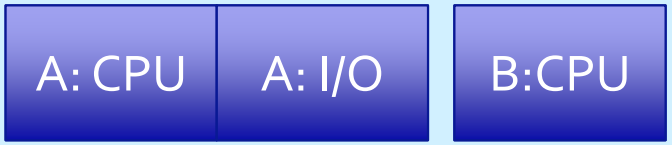
BatchSystems-1 : Arrival assumes they are in S.M. after punch card processing has been completed.

Arrival time assumed t=0 for both A,B. (Context switch time is artificially 0s)

Gap is Monitor activity: Assume it is 0!

Schedule : A B A in system 0-20s; B in system 0-30s
Tr(A)=20s* Tr(B)=30s* ;Tr(<A,B>) = 30s
MTT : (20+30)/2 =25s

Schedule : B A B in system 0-10s; A in system 0-30s
Tr(A)=30s Tr(B)=10s Tr(<B,A>)=30s
MTT : (30+10)/2 = 20s

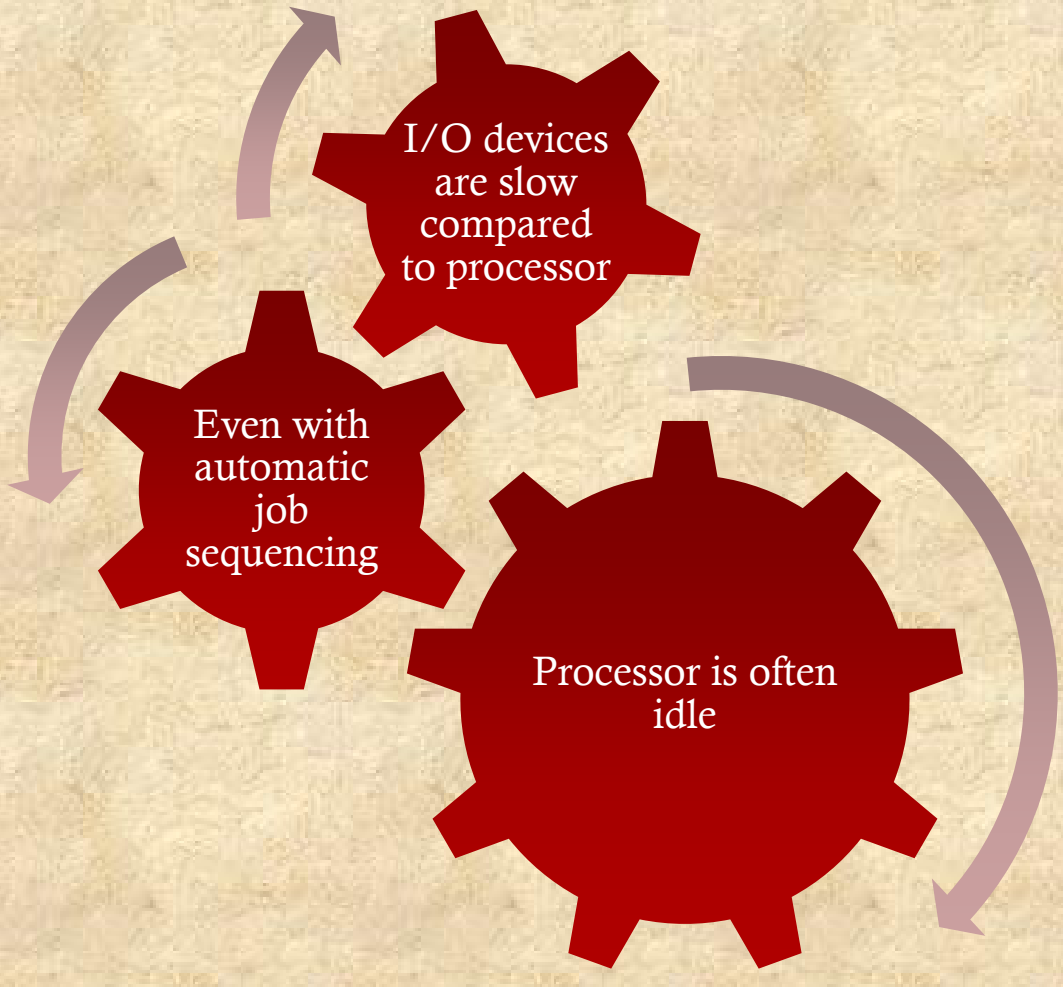


CPU utilization (either schedule) : 20s/30s = 66%

Throughput (either schedule) : 2jobs/30s or 4 jobs/minute or 240jobs/hour

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Multiprogrammed Batch Systems



© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

3. MultiProgramming

WATCH OUT!

Chapter 1: Multicomputer, Multiprocessor, Multicore

Chapter 2: Multiprogramming, Multitasking, Multiprocessing

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

MultiProgramming intro

Multiprogramming objective: Maximize Processor(CPU) utilization

OS handles more than one program at a time;
One program (say A) does I/O; another program (say B) uses CPU;
Users think that two programs are using CPU at the same time

Term Multiprogramming used in early era of computing (e.g. batch systems)

Do not forget: CPU can only execute one instruction at a time that can belong to one program at a time!

Requirement: Multiprogramming first became available in batch systems.
All programs (e.g. A, B) must be ready to use CPU (i.e. Job has been processed/sequenced) and thus executables are available
Executables in SM though in later years preferable to be in MM

I/O requirements: Not user-generated but multiprogramming generated overhead
(moving A and B back and forth between SM and MM as needed)

Context switches: Plenty and considered overhead
(we do not count context switches because of program I/O)

Job Scheduler : Which (of A,B, etc) to bring in

Medium Term Scheduler: Alert Swapper for swapping
Swapper : Send 'process' A to SM, bring 'process' B from SM; move-out/move-in is swapping!

Rudimentary CPU dispatcher: Dispatch A to make room for B.

term process is not refined yet : so we use 'process' instead

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

MultiP Possibility-1

Multiprogramming

Possibility 1. Programs A,B have been turned into 'processes' but medium term scheduler moved them with the assistance of a swapper to SM. Assume A,B have to keep things simple arrival time os. A moves first from SM to MM and starts executing, after some CPU time it issues I/O. OS stores A back to SM and brings into MM B that completes its execution. Then A after I/O completes, it is brought back into CPU and completes execution and done(this last step is not clearly shown in example on next page as width of As activity is thin and thus we assume that is not there and not accounted).

Job Scheduler (i.e. Long term scheduler) Brought program A and program B into MM (became processes)

Medium term scheduler Sends process A,B back to SM; then brings A back into MM for execution

Observation-1.I/O operation(s) to bring program into MM but also back to SM

Swapper in action : MM to SM or SM to MM

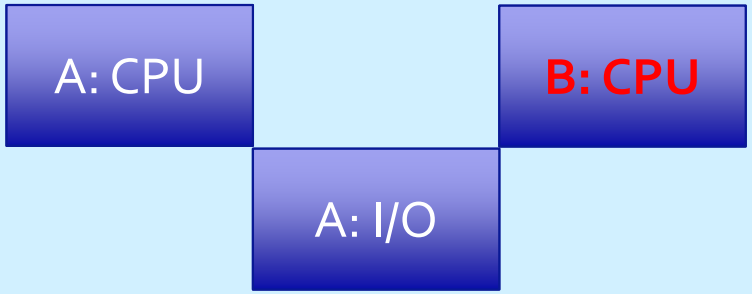
Medium term scheduler: interacts with swapper

Major Context switches : too many to track

Next page: OS overhead not shown assume os

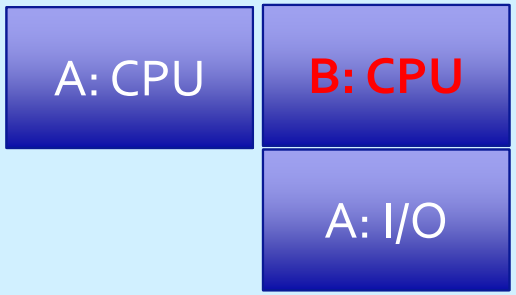
© 2019-2027. Alex. Gerbessiotis. All rights reserved.

MultiProgramming-1



Serial Processing + Batch Processing

Tr(A)=20s, Tr(B)=30s, Tr(A+B)=30s
CPU(A+B)= 20/30=66% (CPU utilization)
MTT = (20+30)/2= 25s
Throughput:2job/30s = 4job/mn=240job/h
R(A)=0s, R(B)=20s Response time



(Batched) Multiprogramming

Tr(A)=20s, Tr(B)=20s, Tr(A+B)=20s
CPU(A+B)=20/20=100%
MTT=(20+20)/2=20s
Throughput: 2job/20s = 360job/h
R(A)=0s, R(B)=10s Response time

Arrival time for both A, B is assumed t=0s

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

MultiP limitations

Multiprogramming

Context switches : Too many, too costly (involve SM)

Better MultiProgramming: Have both A and B in main memory in advance
Context switching then takes microseconds, not milliseconds

CPU (processor) utilization : Max Focus on maximizing it

Introduction of VDU(Video display units aka terminals, monitor, screen), keyboard shifts focus to users

Users (humans) are impatient : **Response time vs CPU utilization**

Previous example R(A) = 0s, R(B)=10s or more . For humans anything more than 3-5s response time is too much.

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Timesharing to minimize response time

Timesharing can be interactive or not.
In modern OS timesharing can be Interactive (Linux ssh) or NOT (other Linux).

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

4. Timesharing

Timesharing initially was interactive timesharing

Interactive: Sit in front of monitor + keyboard and type in say (e.g. secure shell)

Timesharing nowadays can be interactive OR non-interactive

Time sharing: Share the time of the CPU!

Linux is an interactive and non-interactive timesharing (and multitasking) OS. (Wait for definition of multitasking in section 5.)

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Timesharing intro

Alex. Gerbessiotis
COMPUTER SCIENCE

Timesharing objective: Minimize response time (cpu utilization is not an issue)

Timesharing: time sliced use of CPU('s time)

The time of the CPU is slived into time slices of some duration T. There are N tables each one with a VDU+keyboard. Users get seated on each table and interact with the computer (run programs). A multiplexer (connector) allows each table/user to access the CPU for a period T at a time. If user program can't finish execution in period T gets swapped out to SM to make room for next table's user whose swapped out program (if any) is swapped in to MM from SM.
Tables are run round robin. If N= 5, then 0->1->2->3->4->0->1-> etc

Requirement: Interactive system setup
CPU belongs to Table and associated User for a fraction of CPU's time : timeslice
Executable code for program is preferable; SM resident preferable.

I/O requirements: Timesharing generates overhead because of timeslices.
Lots of I/O if programs in SM and not in MM.

Context switches: Lots of context switches!

Job scheduler: As capable as before

CPU dispatcher: More capable to deal with lots of context switching because of timesharing.

Optimal Configuration : All programs in MM or context switching is expensive

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Timesharing Example 1

Alex. Gerbessiotis
COMPUTER SCIENCE

Timesharing

Say N=5 tables; context switch is d and primarily expensive SM I/O (typical 200ms for 1960s)
Response time ? CPU utilization? I/O not timeshared!

```
OS -> 0-> OS ->1-> OS ->2 -> OS ->3 -> OS -> 4 ->OS->0->OS->1->OS->2 and so on
      T d + d  T d + d  T  d + d  T   d +  d T d + d
```

CPU Utilization: **At most 5T/ (10d+5T) over a full round robin round (assumes T is pure CPU execution)**
Max Response Time: **At most 10d+4T (user of Table 0 just missed time slice)**
CPU Utilization: Say user at Table 0 is asleep, at Table 1 eats, Table 2 drinks, Table 3 talks to Table 4 user. What is the CPU utilization then?

In a system d=200ms and N=5. What should T be to maintain (for full CPU utilization of a time slice)

CPU Utilization at least 90% ?	Anwer: T>= 3.6s too much f
What's max response time them ?	Answer: 10d+4T=10*0.2s + 4*3.6 = 16.4s
What is T for a max response time of 5s?	Answer: 10d+4T <=5s ➔ T=0.75s
What is CPU utilization then?	Answer: 5T/(10d+5T)= 3.75/(3.75+2) =65%

In the 1950s student and faculty got upset if they had to wait more than 2s to get the attention of the CPU. Thus (N-1)T < 5s
The first system could support 10 interactive users maximum. Thus N=10. Thus a timeslice of T=0.5s since (N-1)T=9*0.5s<5s
(The timeslice includes in this case context switch time c).

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Time-Sharing Systems

- Can be used to handle multiple interactive jobs
- Processor time is shared among multiple users
- Multiple users simultaneously access the system through terminals, with the OS interleaving the execution of each user program in a short burst or quantum of computation

Time-sharing can also be NON-INTERACTIVE
Time-sharing: Minimize response time!

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

	Batch Multiprogramming	Time Sharing
Principal objective	Maximize processor use	Minimize response time
Source of directives to operating system	Job control language commands provided with the job	Commands entered at the terminal

Table 2.3 Batch Multiprogramming versus Time Sharing

not always

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Compatible Time-Sharing System (CTSS)

- One of the first time-sharing operating systems
- Developed at MIT by a group known as Project MAC
- The system was first developed for the IBM 709 in 1961
- Ran on a computer with 32,000 36-bit words of main memory, with the resident monitor consuming 5000 of that
- Utilized a technique known as *time slicing*
  - System clock generated interrupts at a rate of approximately one every 0.2 seconds
  - At each clock interrupt the OS regained control and could assign the processor to another user
  - Thus, at regular time intervals the current user would be preempted and another user loaded in
  - To preserve the old user program status for later resumption, the old user programs and data were written out to disk before the new user programs and data were read in
  - Old user program code and data were restored in main memory when that program was next given a turn

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

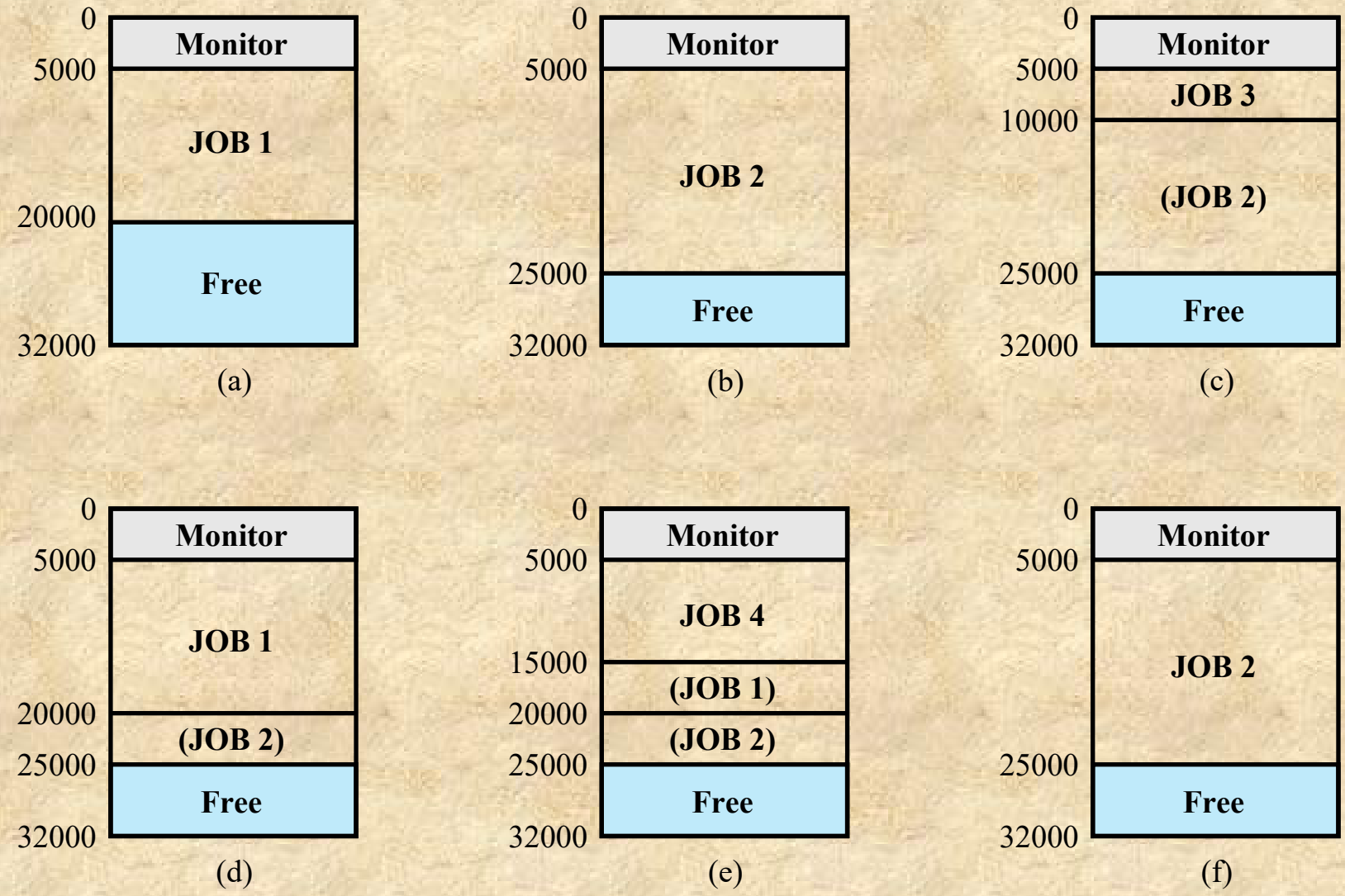


Figure 2.7 CTSS Operation

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Timesharing-1

Timesharing 1:

N=2 and T=5s. Two users A,B all arrived at t=0s and book tables 0,1 respectively.
All CPU-bound (use CPU primarily and/or exclusively)

Assumption 1: **Context switch time c=0s.**
Service time s(A)=5s, s(B)=3s

```
  0 1 2 3 4 5     6 7 8 9 0
-> A A A A A  -> B B B      ->
   table 0       table 1
```

AB

R for Response time.

R(A)=0s, R(B)=5s

 # context switches in [0-,10+]s : 4

CPU utilization: **8s/ 10s = 80%**

Turnaround time :

Tr(A)=5s, Tr(B)=8s or 10s ?

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Timesharing-2

Timesharing 2:

N=2 and T=5s. Two users A,B all arrived at t=0s and book tables 0,1 respectively.
All CPU-bound (use CPU primarily and/or exclusively)

Assumption 1: **Context switch time c=0s.**
Service time s(A)=5s, s(B)=3s but in the pattern 2s CPU, 1s I/O, 1s CPU

```
   0 1 2 3 4 5     6 7 8 9 0
 -> A A A A A  -> B B   B    ->
I/O:                  B
    table 0       table 1
```

AB

R for Response time.

R(A)=0s, R(B)=5s

 # context switches in [0-,10+]s : 4

CPU utilization: **8s/ 10s = 80%**

Turnaround time :

Tr(A)=5s, Tr(B)=9s or 10s ?

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Timesharing-3

Timesharing 3:

N=2 and T=5s. Two users A,B all arrived at t=0s and book tables 0,1 respectively.
A is CPU-bound, B is not

Assumption 1: **Context switch time c=0s.**
Service time s(A)=5s, s(B)=3s but in the pattern 2s CPU, 3s I/O, 1s CPU

```
   0 1 2 3 4 5     6 7 8 9 0    1 2 3 4 5     6 7 8 9 0
 -> A A A A A  -> B B       ->           -> B
I/O:                  B B B
    table 0       table 1       table 0         table 1
```

AB

R for Response time.

R(A)=0s, R(B)=5s

 # context switches in [0-,10+]s : 4

CPU utilization: **8s/ 20s = 40%**

Turnaround time :

Tr(A)=5s, Tr(B)=16s or 20s ?

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Example-1

Example-1. Four CPU-bound tasks have service times **A: 8u , B:4u, C:4u and D:5u** and they all arrive at t=0 in this order (lexicographic) .
Answer the following questions re (1) serial processing, (2) (batched) multiprogramming, (3) timesharing with ts=1u.
(a) What is the response time R(A), R(B), R(C), R(D) of the CPU for A, B, C, D?
(b) What is the turnaround time for A, D i.e. Tr(A), Tr(D)?
(c) What is the CPU utilization (aka processor utilization) for the execution of the job consisting of tasks A,B,C,D?
(d) Which one has most context switches among (1), (2) and (3)?

1. Serial processing (we timeline the execution 1 letter is 1u and shows who is using the CPU)

123456789012345678901
AAAAAAAABBBBCCCCDDDDD **This is the CPU timeline;all tasks CPU-bound**

(a) R(A)=0u, R(B)=8u, R(C)=12u, R(D)=16u
(b) Tr(A)=8u, Tr(D)=21u
(c) 21/21= 100% (d) #context switches 0, or in modern realizations 1+2+2+2+1=8

2. Multiprogramming. As above. Assume all in MM is given.

(d) #context switches 1+2+2+2+1=8

3. Timesharing with T=1u

123456789012345678901234567890
ABCDABCDABCDABCDA--DA---A---A

(a) R(A)=0u, R(B)=1u, R(C)=2u, R(D)=3u
(b) Tr(A)=29u, Tr(D)=20u
(c) Not 100% . No objective to maximize CPU utilization just minimize response time.
(d) # context switches ~ 42 or so. (A--D we have from A to OS, then OS to D)

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

5. Multitasking

Linux is an interactive and multitasking operating system.

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Multitasking intro

Multitasking objective: Maximize CPU utilization + minimize response time

Multitasking = Multiprogramming in a timesharing system

OS handles several programs at a time; all programs in MM (possibly except for initial loading)
Each program in an ordered fashion gets access of the CPU for a max fraction of time (quantum)
System can be interactive or not
If program is not to use full time of quantum; CPU preempts it (moved out of CPU)

Requirement:	Program 'must be ready' to use CPU i.e. they are processes and in MM
I/O requirements:	Lots of I/O if programs (processes) not in MM (processes)
Context switches:	Overhead. Lots of context switches!
Quantum:	New name for timeslice; enhanced CPU dispatcher
Job scheduler:	As before
CPU dispatcher:	More capable than timesharing (fill the gaps)

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Example-1

Example-1. Four CPU-bound tasks have service times A: 8u , B:4u, C:4u and D:5u and they all arrive at t=0 in this order (lexicographic) and are in MM already.
Answer the following questions re (4) multitasking with quantum q=1u.

(a) What is the response time R(A), R(B), R(C), R(D) of the CPU for A, B, C, D?
(b) What is the turnaround time for A, D i.e. Tr(A), Tr(D)?
(c) What is the CPU utilization (aka processor utilization) for the execution of the job consisting of tasks A,B,C,D?
(d) Which one has most context switches among (1), (2) and (3)?

4. Multitasking q=1u

```
12345678901234567890123456789
ABCDABCDABCDABCDA--DA---A---A this was timesharing (reference)
ABCDABCDABCDABCDADAAA   that's multitasking
```

(a) R(A)=0u, R(B)=1u, R(C)=2u, R(D)=3u no change
(b) Tr(A)=21u improved Tr(D)=18u improved
(c) 100% .
(d) # context switches 42 or may be 38 ?

How many context switches in the following ? `XY---YX`
`OS to X to OS to Y to OS to Y to OS to X to OS` : 8 context switches.

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Example 1: remaining cases

1. Serial Example
   (4 Users CPU-bound work:
   A:8u B:4u C:4u D:5u)

```
12345678901234567890 R:  Response time; Tr: turnaround time
AAAAAAAABBBBCCCCDDDDD        R(D)=16u, Tr(A)=8u, Tr(D)=21u,
                             CPU utilization: 100%
```

2. MultiProgramming.

```
No need; all 4 jobs are CPU-bound no IO activity!
```

3. Timesharing example: timeslice=1u
4 Jobs: A:8u B:4u C:4u D:5u

```
12345678901234567890123456789012
ABCDABCDABCDABCDA--DA---A---A---  R(D)=3u,Tr(A)=29u,Tr(D)=20u
                                 CPU utilization  < 100%
Timeslices 4,8,12,16,20,24,28,32 belong to 'program' D even
if D is done...Likewise ...,18,22,26,30 belong to B,
...,19,23,27,31 to C.
(in practice D is run from terminal 4, A from terminal 1,
etc)
```

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

MultiP vs MultiT

Multiprogramming

max CPU utilization

Programs might be in HDD
Complex Job Scheduler by Monitor
Elementary or no CPU dispatcher
Some context switching
Timeslice
Cooperative 'multitasking'
(non-preemptive)

Multitasking

CPU sharing

CPU sharing to minimize response time
+ max CPU utilization

Programs MUST be in main memory
Elementary Job Scheduler
CPU Dispatcher by/in 'kernel'
Frequent context switching
Quantum (form of timeslice)
Preemptive 'multitasking'

Timeslice belongs to User/Program
If program does not plan to use timeslice
It gets wasted unless program cooperatively releases it to OS (cooperative multitasking)

Quantum belongs to/is controlled by OS

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Multitasking concluded

Cooperative multitasking: The CPU dispatcher can not stop a program from using the CPU but the program itself voluntarily does so

Preemptive multitasking: The CPU dispatcher initiates all CPU utilization.

Example 1 part(3): A timeslice is allocated to Program B,C whether it wants to use it or not. If B,C signals it does not need the CPU this is cooperative multitasking. Otherwise the timeslice is wasted, as it was observed earlier for timesharing.

Example 1 part (4): A quantum is allocated to Program B. When the OS detects that B does not need the CPU, B gets preempted and another program starts using CPU eg D.

Early versions of Windows or MacOS X used cooperative multitasking

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

6. Multiprocessing

Multitasking where a task is split into more than one programs (processes); if more than one processors are available the programs(processes) can be executed concurrently ('in parallel') to improve performance

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

7. Real time systems

Response time is maintained within given bounds

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Module 2

OS contributions

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Major Achievements

- Operating Systems are among the most complex pieces of software ever developed
- Major advances in development include:
  - Processes
  - Memory management
  - Information protection and security
  - Scheduling and resource management
  - System structure

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Process

- Fundamental to the structure of operating systems

A *process* can be defined as:

A program in execution

An instance of a running program

The entity that can be assigned to, and executed on, a processor

A unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Program (on a disk)

A program (on a disk) is associated with the following

- **text (also known as instructions or code),**
- **initialized data**
- **uninitialized data (also known as bss)**

and has provisions for

- **heap data** for dynamically allocated data
- **user stack data** for user function call invocation [user-mode]
- **shared (memory) data** to be shared
- **kernel stack data** for user's system call invocation [kernel-mode]

but these latter areas are populated when 'program is in execution' in main memory

Note: if we use generically the term stack we still imply a user stack and a kernel stack

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Process: program in execution

Data are kept separate from instructions (code i.e. text from now on ,program) in memory
Process state (context) is the values of the registers during the execution of the program

Program

Passive
OS does not know about it
On disk
Prev page: Text
Prev page: User+Kernel stack
Prev page: Other data

Process

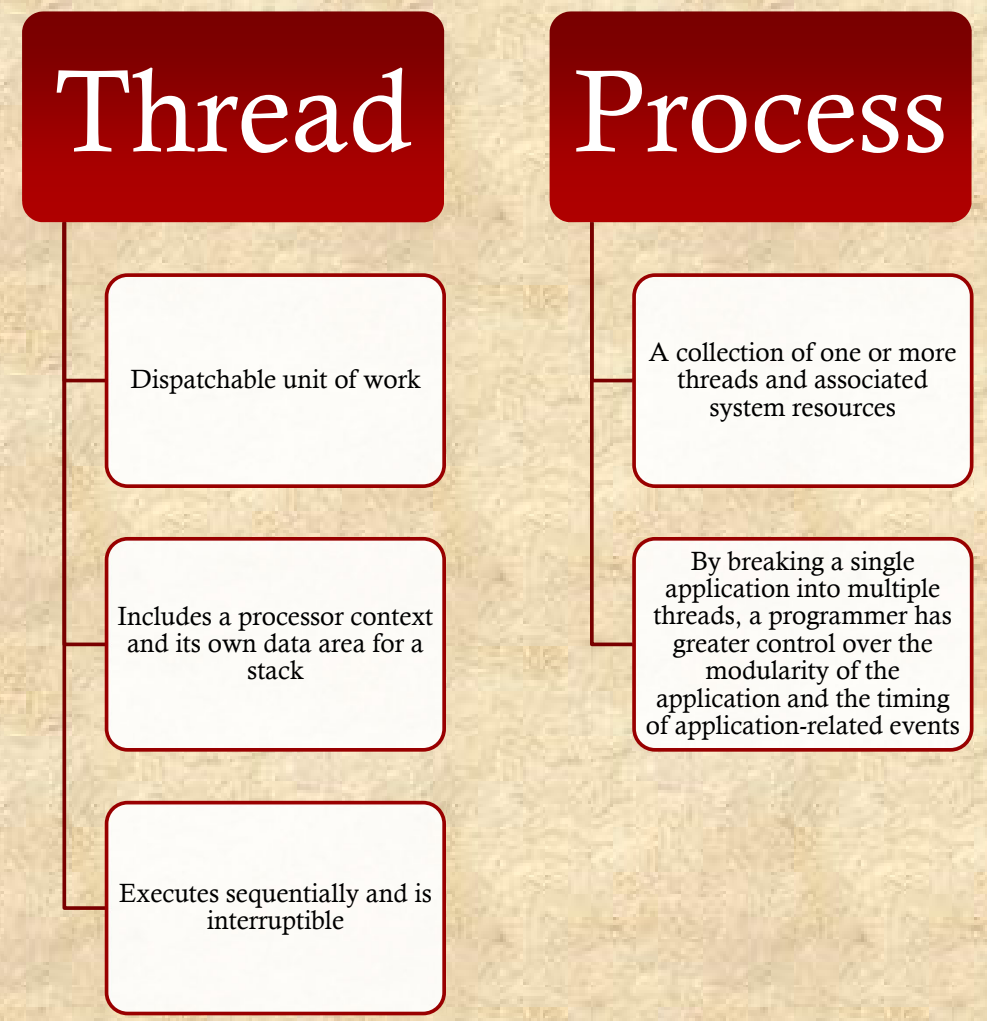
Active
OS knows about it
In main memory (all or part)
Prev page: Text
Prev page: User+Kernel stack
Prev page: Other data
Process state (context)
    Processor state (Chap 1)
    OS related info (PID,priority,etc)

Files for I/O

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Multithreading

- Technique in which a process, executing an application, is divided into threads that can run concurrently



© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Thread: a lightweight process

A thread is spawn (created) by a process and shares with it text and data(init glob vars, uninit glob vars, heap data, files) but has its own register file, stack.

Thread

Active
OS knows about it or not!
In main memory (all or part)
Text: of Process
User+Kernel stack
Other data: **of Process**

Processor Registers/file/state
OS related info: of Process

Files for I/O: of Process

Process

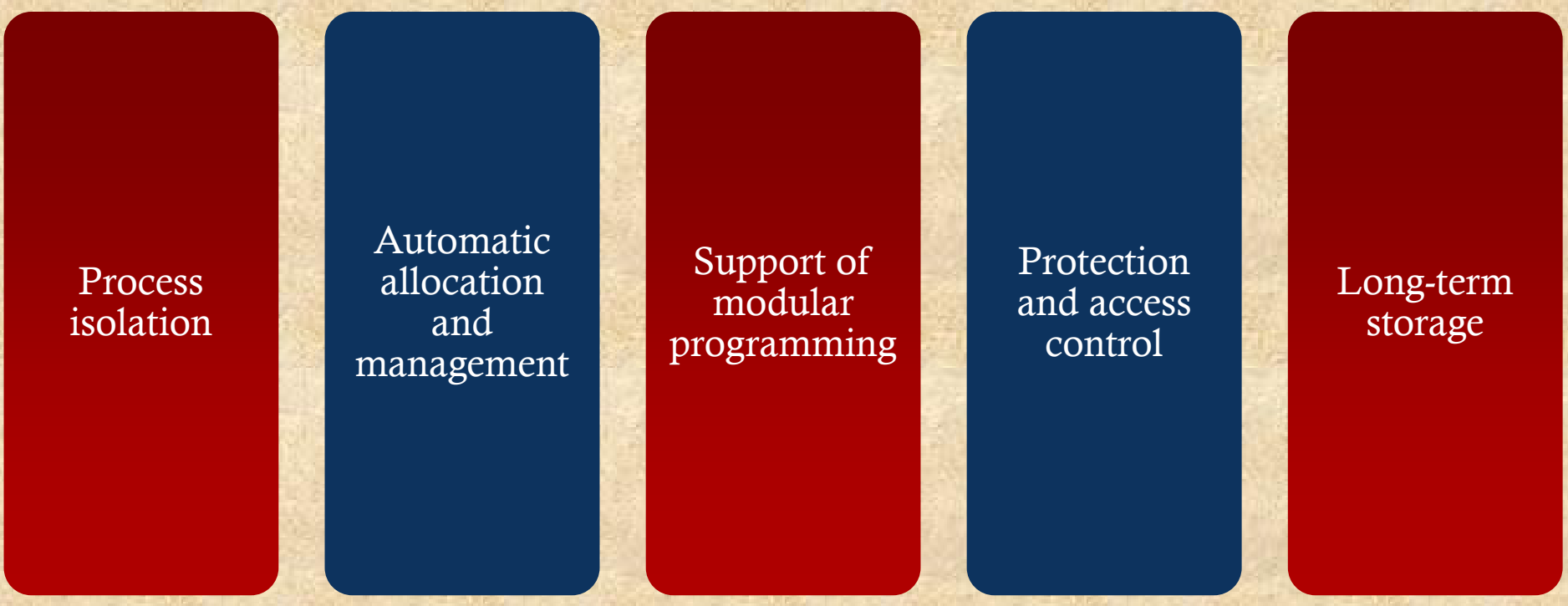
Active
OS knows about it
In main memory (all or part)
Text
User+Kernel stack
Other data
Process state (context)
Processor Registers/state/file (Chap 1)
OS related info (PID,priority,etc)

Files for I/O

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Memory Management

- The OS has five principal storage management responsibilities:



© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Virtual Memory

Prior to Virtual Memory

A program's size is determined by size of MM or size of available MM

With Virtual Memory

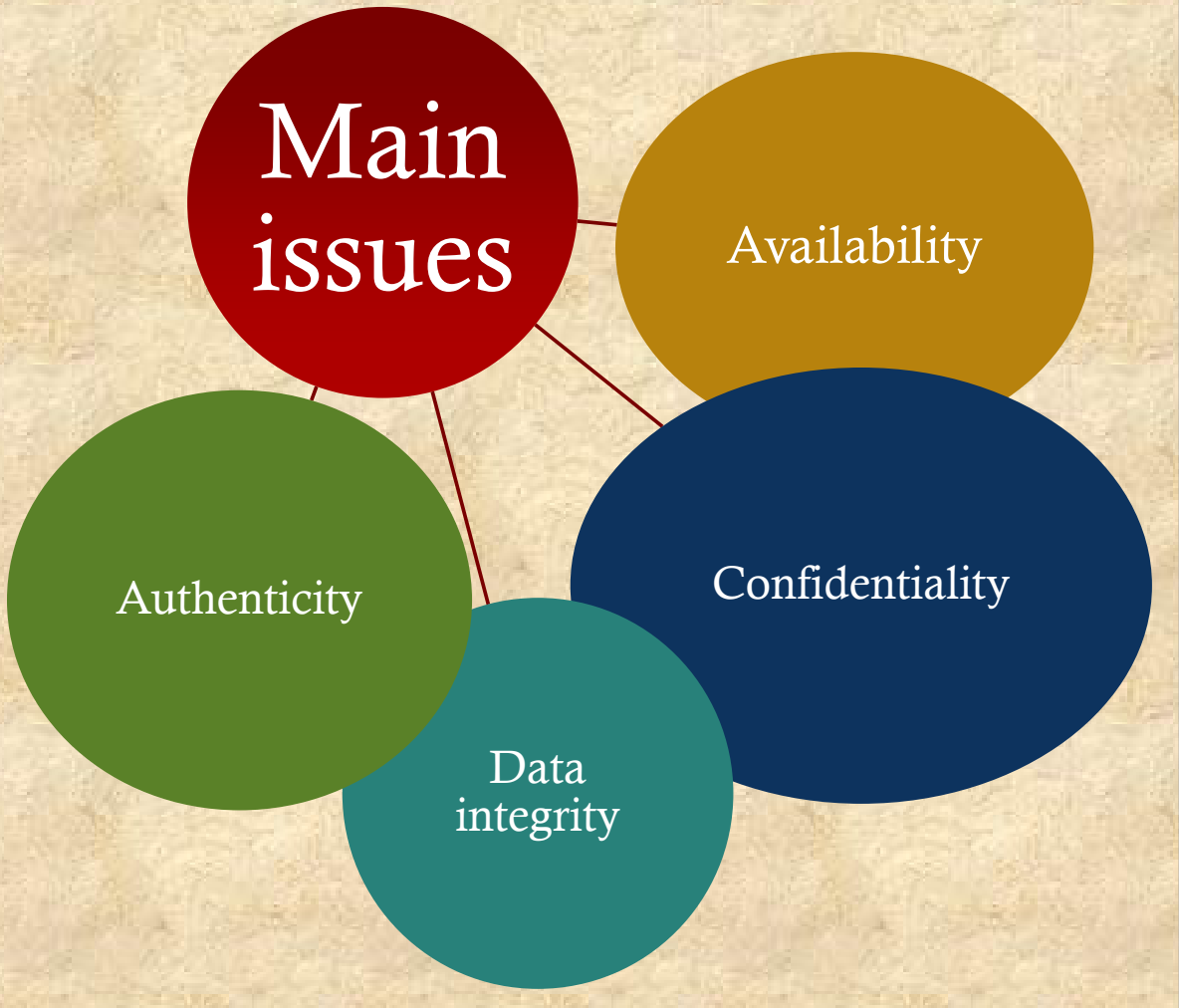
A program's size is determined by size of (available) SM and is independent of size of MM

Not all of a program is stored in MM at any time. Paging support is needed for Virtual memory

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Information Protection and Security

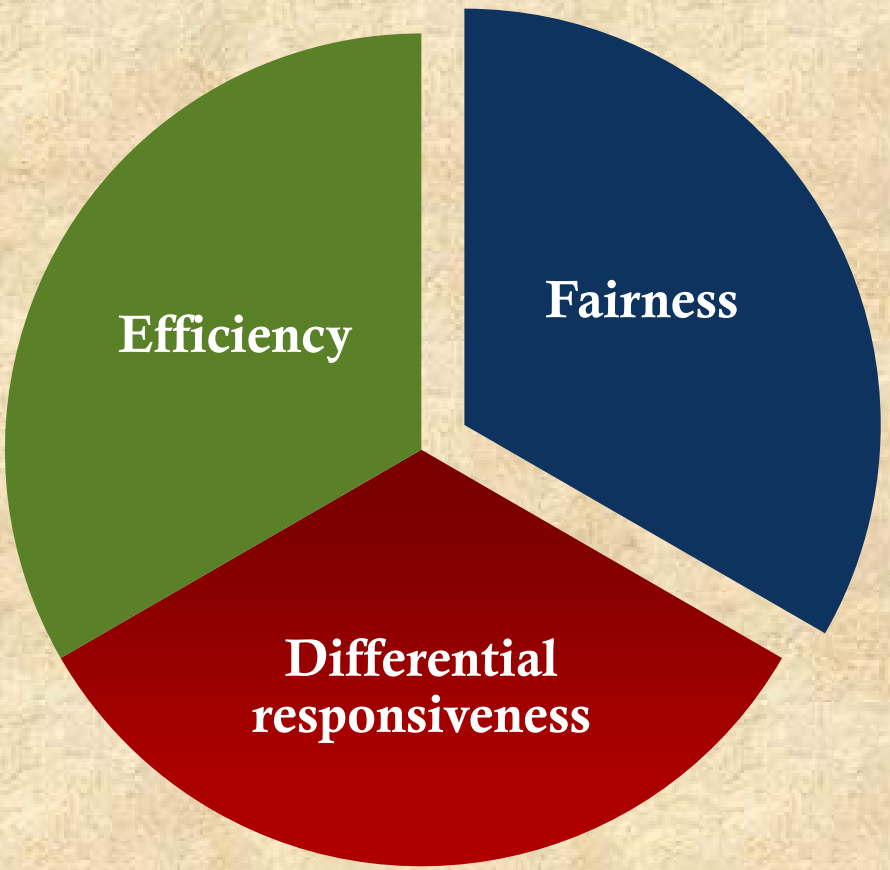
- The nature of the threat that concerns an organization will vary greatly depending on the circumstances
- The problem involves controlling access to computer systems and the information stored in them



© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Scheduling and Resource Management

- Key responsibility of an OS is managing resources
- Resource allocation policies must consider:



© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Fault Tolerance

- Refers to the ability of a system or component to continue normal operation despite the presence of hardware or software faults
- Typically involves some degree of redundancy
- Intended to increase the reliability of a system
  - Typically comes with a cost in financial terms or performance
- The extent adoption of fault tolerance measures must be determined by how critical the resource is

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Class	Availability	Annual Downtime
Continuous	1.0	0
Fault Tolerant	0.99999	5 minutes
Fault Resilient	0.9999	53 minutes
High Availability	0.999	8.3 hours
Normal Availability	0.99 - 0.995	44-87 hours

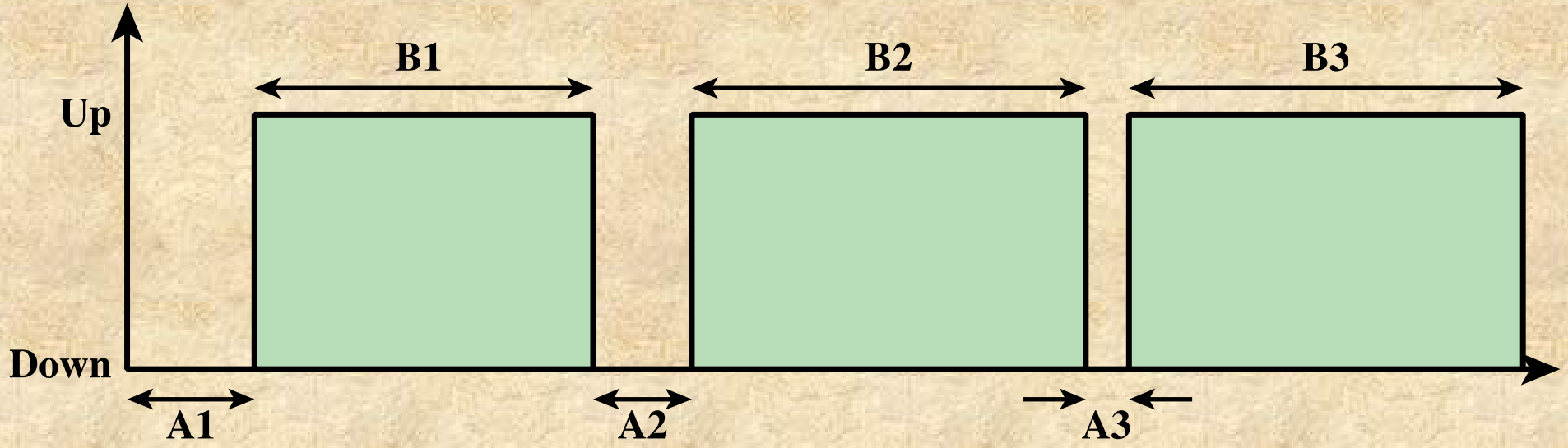
Table 2.4 Availability Classes

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Fundamental Concepts

- The basic measures are:
  - Reliability
    - *R(t)*
    - Defined as the probability of its correct operation up to time *t* given that the system was operating correctly at time *t=o*
  - Mean time to failure (MTTF)
    - Mean time to repair (MTTR) is the average time it takes to repair or replace a faulty element
  - Availability
    - Defined as the fraction of time the system is available to service users' requests

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.



$$MTTF = \frac{B1 + B2 + B3}{3} \qquad MTTR = \frac{A1 + A2 + A3}{3}$$

Figure 2.13 System Operational States

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Definitions

MTTF:
(Mean Time To Failure)

Average time it takes for component to fail from start of operation, or start after a previous replacement of a failed component

MTTR:
(Mean Time To Repair)

Average time it takes to repair or replace a failed component from the point it failed to the point it was replaced or repaired

MTBF:
(Mean Time Between Failures)

MTBF = MTTF + MTTR

Annualized Failure Rate:

1-exp(-8760/MTTF) ~ 8760/MTTF , MTTF in h

https://www.toshiba-storage.com/trends-technology/mttf-what-hard-drive-reliability-really-means/

Availability:

Fraction of time the system is available to service users' requests

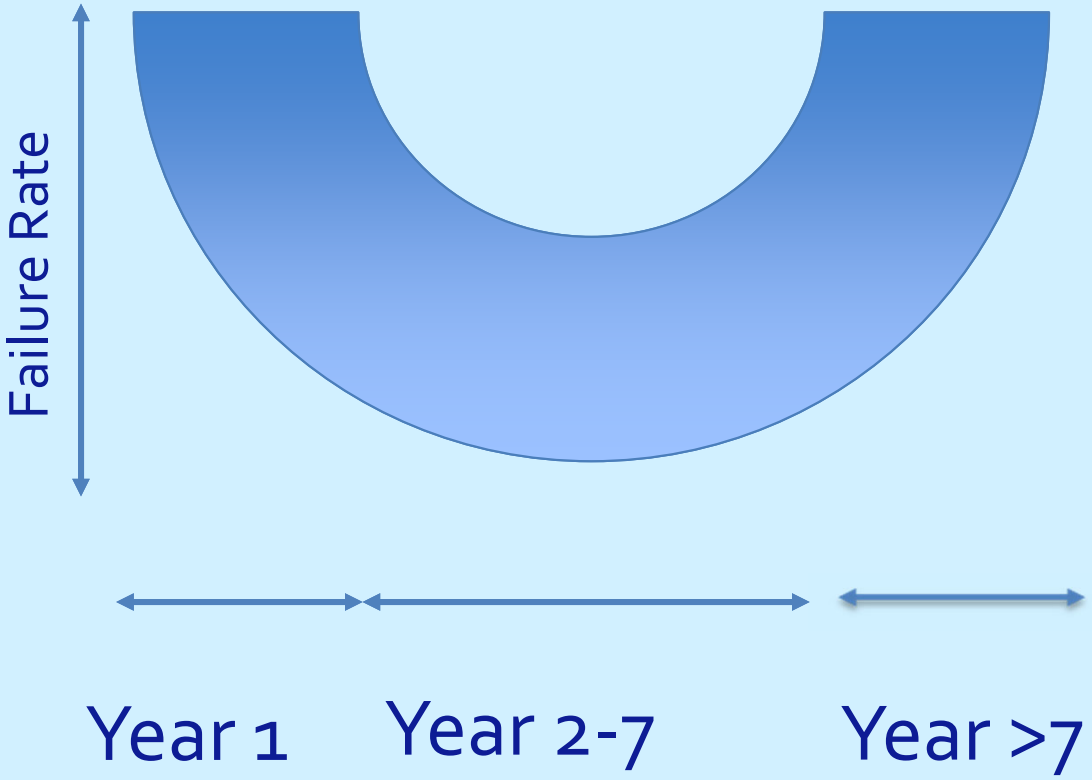
Reliability, R(t):

Probability of correct operation at t given it worked correctly starting at t=0

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Case Study : HDD

Bathtub effect



Google (2007)

0-3mo	2.5%
6mo	2%
12mo	2%
24mo	8%
36mo	10%
48mo	6%
60mo	7%

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Note: SSD

SSD : Solid State Drives

Some Rated for 50TiB write volume;
Currently most rated for 100TiB write volume;
25%-33% will not be able to surpass 600TiB writes
Almost all will fail to do 1000TiB writes!

(Data for 256GiB SSD reported.)

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Review 2

2.1 A manufacturer tests 1000 HDD. After 1000 hours of operation the first HDD failed. What is the MTTF for this HDD batch?

Answer: MTTF = (period-of-time * num-devices) / num-of-fails

= (1000hrs * 1000 HDD) / 1 fail ~ 1,000,000hr

2.2 What is the AFR ?

Answer: AFR ~ 8760/MTTF ~ 0.876%

2.3. One hour after the experiment and the publication of the MTTF by public relations, another 9 drives failed. What is the MTTF?

Answer:

For [0, 1001hr] we have MTTF = 1001 * 1000 /10 ~ 100,000hr !

The average of the two MTTF is 550,000hr approximately.

Spinning MTTF numbers ...

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Traditional UNIX Systems

- Developed at Bell Labs and became operational on a PDP-7 in 1970
- The first notable milestone was porting the UNIX system from the PDP-7 to the PDP-11
  - First showed that UNIX would be an OS for all computers
- Next milestone was rewriting UNIX in the programming language C
  - Demonstrated the advantages of using a high-level language for system code
- Was described in a technical journal for the first time in 1974
- First widely available version outside Bell Labs was Version 6 in 1976
- Version 7, released in 1978, is the ancestor of most modern UNIX systems
- Most important of the non-AT&T systems was UNIX BSD (Berkeley Software Distribution), running first on PDP and then on VAX computers

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

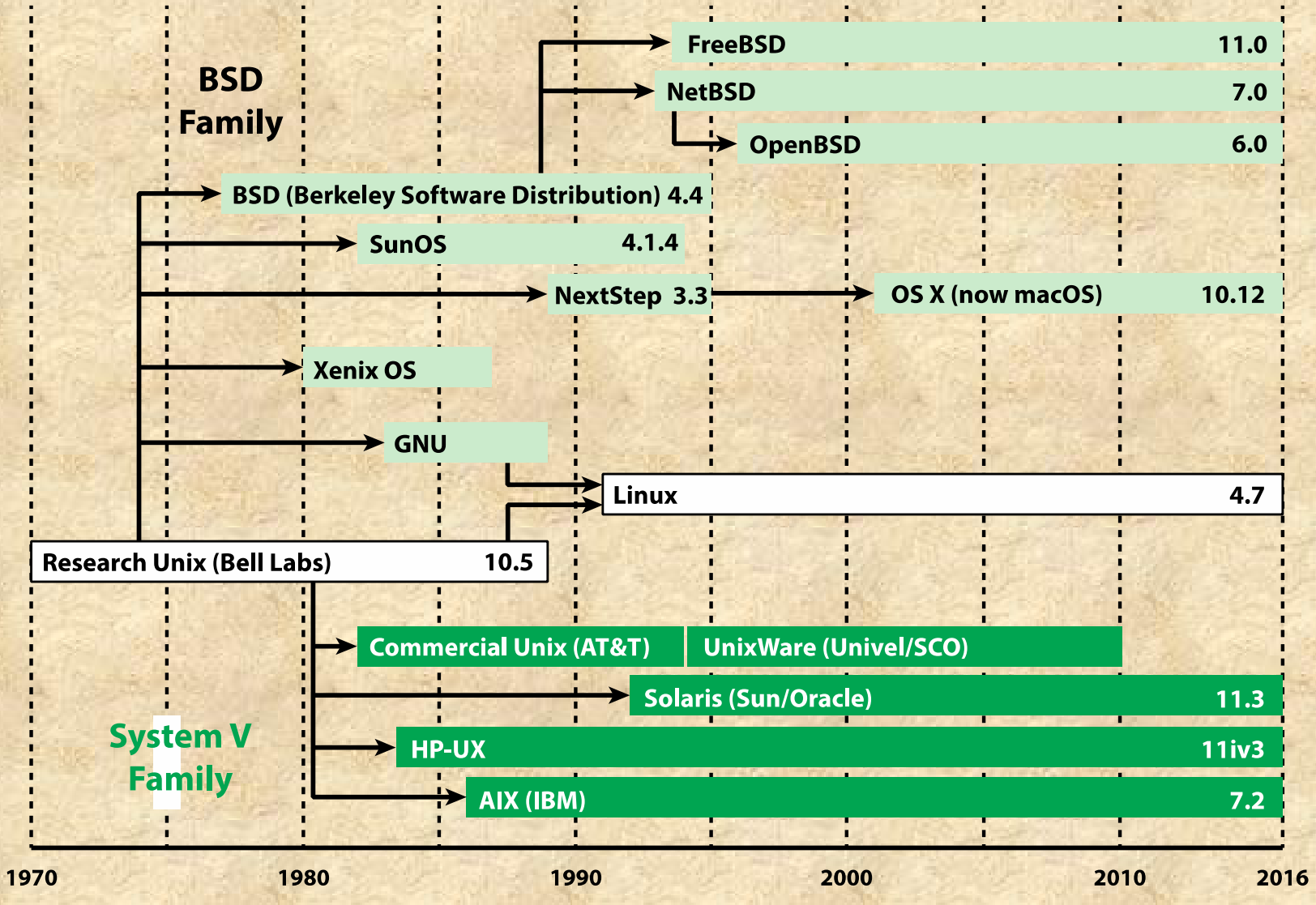


Figure 2.17 Unix Family Tree

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

LINUX Overview

- Started out as a UNIX variant for the IBM PC
- Linus Torvalds, a Finnish student of computer science, wrote the initial version
- Linux was first posted on the Internet in 1991
- Today it is a full-featured UNIX system that runs on virtually all platforms
- Is free and the source code is available
- Key to the success of Linux has been the availability of free software packages under the auspices of the Free Software Foundation (FSF)
- Highly modular and easily configured

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Modular Structure

- Linux development is global and done by a loosely associated group of independent developers
- **Although Linux does not use a microkernel approach, it achieves many of the potential advantages of the approach by means of its particular modular architecture**
- Linux is structured as a collection of modules, a number of which can be automatically loaded and unloaded on demand

Loadable Modules

- Relatively independent blocks
- A module is an object file whose code can be linked to and unlinked from the kernel at runtime
- A module is executed in kernel mode on behalf of the current process
- Have two important characteristics:
  - Dynamic linking
  - Stackable modules

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

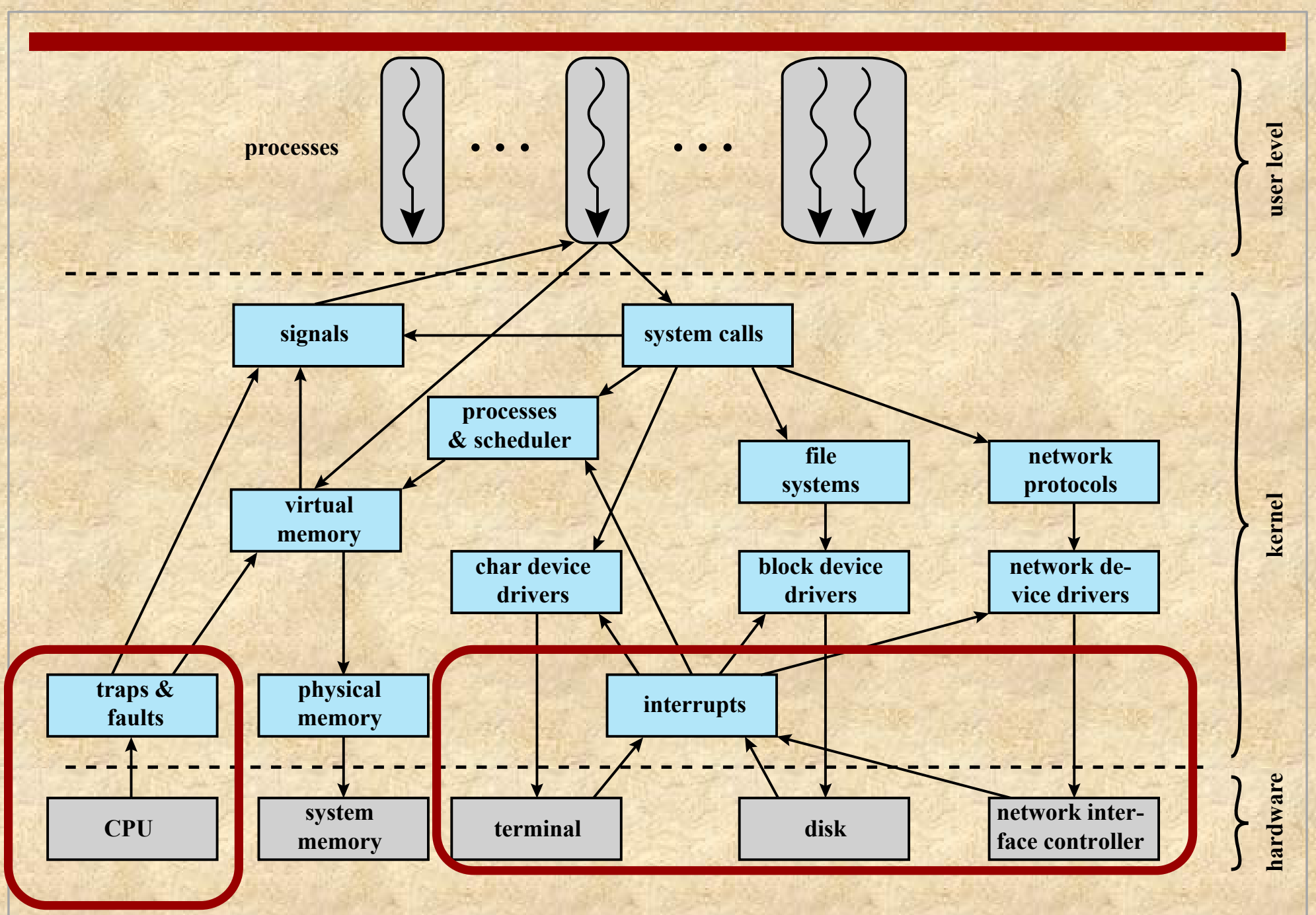


Figure 2.19 Linux Kernel Components

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

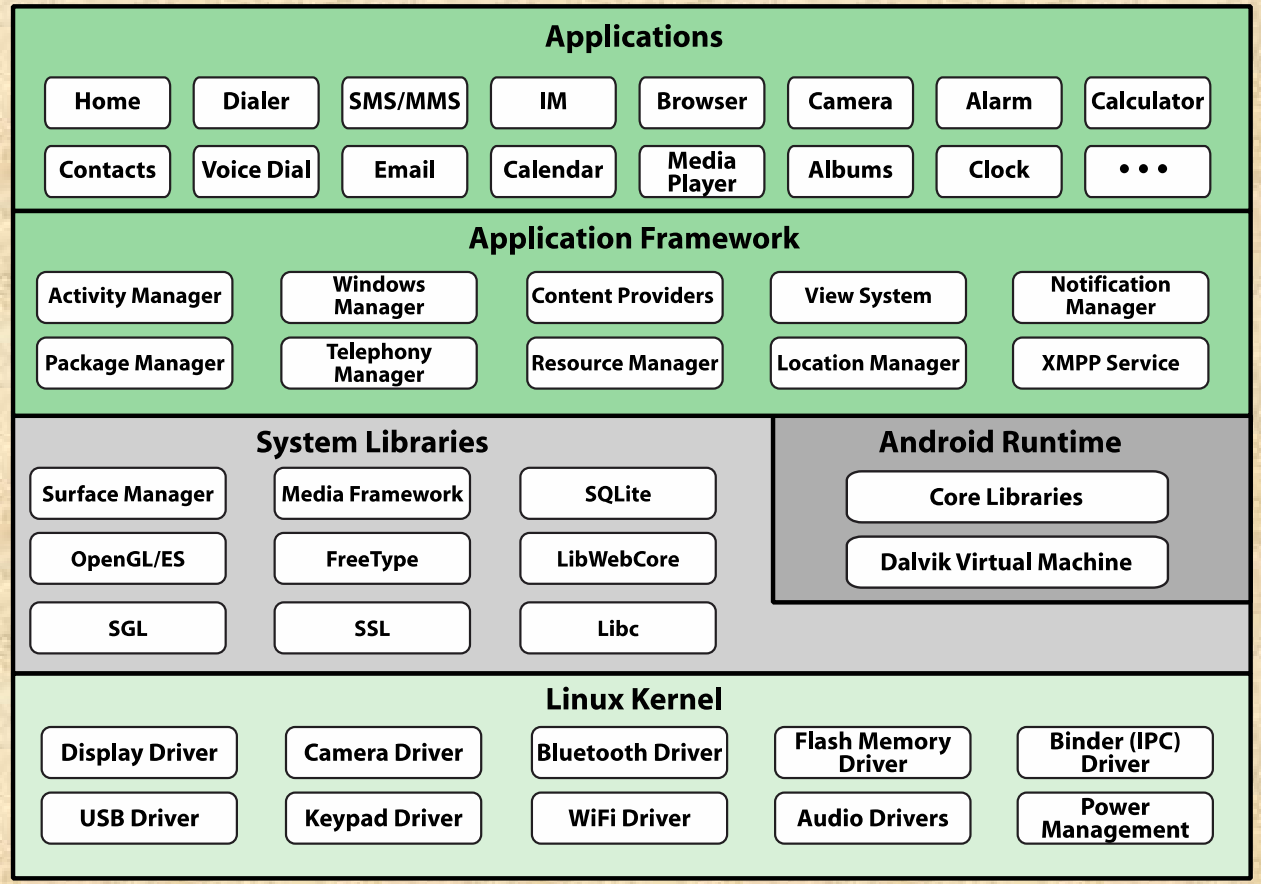


Figure 2.20 Android Software Architecture

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Android Runtime

- **Most Android software is mapped into a bytecode format which is then transformed into native instructions on the device itself**
- Earlier releases of Android used a scheme known as Dalvik, however Dalvik has a number of limitations in terms of scaling up to larger memories and multicore architectures
- More recent releases of Android rely on a scheme known as Android runtime (ART)
- ART is fully compatible with Dalvik's existing bytecode format so application developers do not need to change their coding to be executable under ART
- Each Android application runs in its own process, with its own instance of the Dalvik VM

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Advantages and Disadvantages of Using ART

Advantages

- Reduces startup time of applications as native code is directly executed
- Improves battery life because processor usage for JIT is avoided
- Lesser RAM footprint is required for the application to run as there is no storage required for JIT cache
- There are a number of Garbage Collection optimizations and debug enhancements that went into ART

Disadvantages

- Because the conversion from bytecode to native code is done at install time, application installation takes more time
- On the first fresh boot or first boot after factory reset, all applications installed on a device are compiled to native code using dex2opt, therefore the first boot can take significantly longer to reach Home Screen compared to Dalvik
- The native code thus generated is stored on internal storage that requires a significant amount of additional internal storage space

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

Definitions

Compiler
(Old times)

Input: Source code (on disk)
Output: Assembly code (on disk)

Compiler
(Nowadays)

Input: Source code (on disk)
Output: Executable code or Machine code (on disk)

```
gcc x.c -c
gcc x.c -o x
javac x.java
```

(all intermediate output is/was also on disk)

Loader

Moves program from disk to memory i.e
program becomes a process.
and starts executing from a predefined address

```
./x
```

Interpreter

Interprets and runs Source code directly

```
python x.py
```

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Compiler vs Interpreter

Compiled execution: compile +run two steps

1/compile: Source-Code to Machine-Code

gcc myc.c -o myc or gcc –v myc.c –o myc

2/run Load-into-CPU+Execute MachineCode

./myc equivalent to double clicking on an icon in Windows

Interpreted execution: interpret+run in one step

1/

2/interpret+ run Interpret-and-run Source-Code directly

python mypy.py

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Java compiled execution

Compiled execution: compile +run two steps

1/compile: **javac my.java**

1a. **Source-Code to Java Byte-Code** **[Time: 100s]**

converts a .java file into .jar or .class file (java byte code not machine code)

At runtime

2/run **java my.class**

2a. Invoke Java Virtual Machine (JVM) to load
Java-Byte-Code and **[Time: 20s]**
2b. 'interepret/run' it by JIT (Just In Time) interpreter, and **[Time:20s]**
2c. converting it into machine code for execution, and **[Time:20s]**
2d. executing it… **[Time: 20s]**

Note 1: Java Byte-Code is the same for all platforms (i.e. ISA) whether Intel, PowerPC, ARM etc. Thus **Java Byte-Code is portable**

Note 2: JVM interprets the byte code and eventually converts into machine code; it is the converted product, the machice code that is being executed.

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Java in Android (Dalvik)

Compiled execution: Android Dalvik (before KitKat 4.4)

1/compile: **javac my.java**

1a. **Source-Code to Java Byte-Code** **[Time: 100s]**

converts a .java file into .jar or .class file

1b. Java Byte-Code into Dalvik Byte-Code [Time:60s]

Use dx tool to convert .jar or .class files into .dex

jar cvf my.jar my.class
dx -dex --output=my.dex my.jar **dx or d8**
...

At runtime

2/run **dalvikvm my.dex**

2a. Invoke Dalvik Virtual Machine (JVM) to load Dalvik-Byte-Code and **[Time:15s]**
2b. 'interepret/run' it by JIT (Just In Time) interpreter, and **[Time:5s]**
2c. converting it into machine code for execution, and **[Time:5s]**
2d. executing it... **[Time:25s]**

.dex file is being interpreted and converted into machine code for execution

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

Java in Android (Art)

Compiled execution: Android Art (KitKat 4.4 or later)

1/compile: **javac my.java**

1a. Source-Code to Java Byte-Code **[Time: 100s]**

converts a .java file into .jar or .class file

1b. Java Byte-Code into Dalvik Byte-Code **[Time:60s]**

Use dx tool to convert .jar or .class files into .dex

```
jar cvf my.jar my.class
dx -dex --output=my.dex my.jar
...
```

At install time (once)

1.5.a,b,c. Convert Dalvik Byte-Code into machine code **[Time:25s]**

(AOT: Ahead Of Time compilation not JIT)

At runtime

2/run **adb push my.dex**

2d. By executing machine code… **[Time:25s]**

machine code for execution just like a C or C++ executable…

machine code now can't run on another hardware phone needs reinstall

© 2019-2027. Alex. Gerbessiotis. All rights reserved.

New Jersey's Science &
Technology University

THE EDGE IN KNOWLEDGE

Alex. Gerbessiotis

CS Department