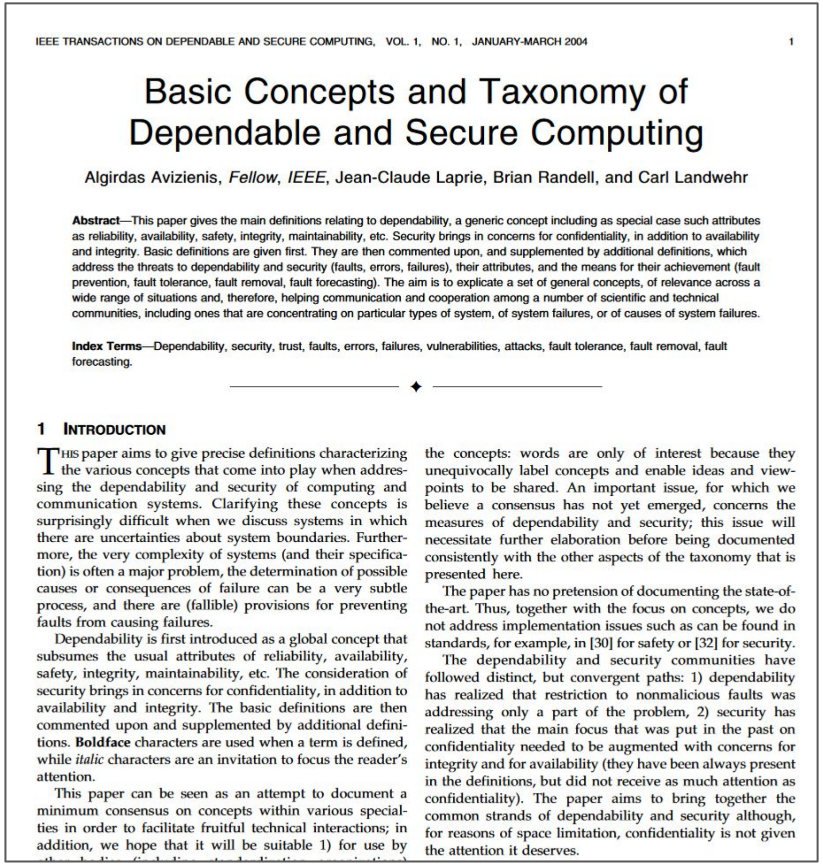
Operating Systems
CS-C3140, Lecture 3

Alexandru Paler

Fault Tolerance

- Refers to the ability of a system or component to continue normal operation despite the presence of hardware or software faults
  - involves some degree of redundancy
  - Intended to increase the reliability of a system
  - 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



IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, VOL. 1, NO. 1, JANUARY-MARCH 2004 1

Basic Concepts and Taxonomy of Dependable and Secure Computing

Algirdas Avizienis, *Fellow*, *IEEE*, Jean-Claude Laprie, Brian Randell, and Carl Landwehr

Abstract—This paper gives the main definitions relating to dependability, a generic concept including as special case such attributes as reliability, availability, safety, integrity, maintainability, etc. Security brings in concerns for confidentiality, in addition to availability and integrity. Basic definitions are given first. They are then commented upon, and supplemented by additional definitions, which address the threats to dependability and security (faults, errors, failures), their attributes, and the means for their achievement (fault prevention, fault tolerance, fault removal, fault forecasting). The aim is to explicate a set of general concepts, of relevance across a wide range of situations and, therefore, helping communication and cooperation among a number of scientific and technical communities, including ones that are concentrating on particular types of system, of system failures, or of causes of system failures.

Index Terms—Dependability, security, trust, faults, errors, failures, vulnerabilities, attacks, fault tolerance, fault removal, fault forecasting.

✦

1 Introduction

THIS paper aims to give precise definitions characterizing the various concepts that come into play when addressing the dependability and security of computing and communication systems. Clarifying these concepts is surprisingly difficult when we discuss systems in which there are uncertainties about system boundaries. Furthermore, the very complexity of systems (and their specification) is often a major problem, the determination of possible causes or consequences of failure can be a very subtle process, and there are (fallible) provisions for preventing faults from causing failures.

Dependability is first introduced as a global concept that subsumes the usual attributes of reliability, availability, safety, integrity, maintainability, etc. The consideration of security brings in concerns for confidentiality, in addition to availability and integrity. The basic definitions are then commented upon and supplemented by additional definitions. **Boldface** characters are used when a term is defined, while *italic* characters are an invitation to focus the reader's attention.

This paper can be seen as an attempt to document a minimum consensus on concepts within various specialties in order to facilitate fruitful technical interactions; in addition, we hope that it will be suitable 1) for use by

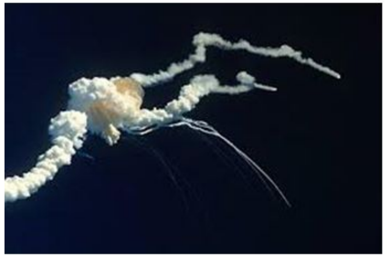
the concepts: words are only of interest because they unequivocally label concepts and enable ideas and viewpoints to be shared. An important issue, for which we believe a consensus has not yet emerged, concerns the measures of dependability and security; this issue will necessitate further elaboration before being documented consistently with the other aspects of the taxonomy that is presented here.

The paper has no pretension of documenting the state-of-the-art. Thus, together with the focus on concepts, we do not address implementation issues such as can be found in standards, for example, in [30] for safety or [32] for security.

The dependability and security communities have followed distinct, but convergent paths: 1) dependability has realized that restriction to nonmalicious faults was addressing only a part of the problem, 2) security has realized that the main focus that was put in the past on confidentiality needed to be augmented with concerns for integrity and for availability (they have been always present in the definitions, but did not receive as much attention as confidentiality). The paper aims to bring together the common strands of dependability and security although, for reasons of space limitation, confidentiality is not given the attention it deserves.

From Failure to Disaster

- Computers are components in larger technical or societal systems
- Failure detection and manual back-up system can prevent disaster
  - Used routinely in safety-critical systems
  - Manual control/override in jetliners
  - Ground-based control for spacecraft
  - Manual bypass in nuclear reactors
- Catastrophic
  - Serious consequences
- Major
  - Incorrect operation
  - Possibly recoverable
- Minor
  - Inconvenience
- Not noticed



Space Shuttle Challenger Disaster

- Second shuttle after Columbia, first mission on April 4, 1983
- Exploded 73 seconds after launch on its 10th mission (January 28, 1986)
- Seven crew members killed
- Temperatures dipped below freezing on launch day
- Engineers determined that the cold temperatures caused a loss of flexibility in the O-rings that decreased their ability to seal the field joints, which allowed hot gas and soot to flow past the primary O-rings

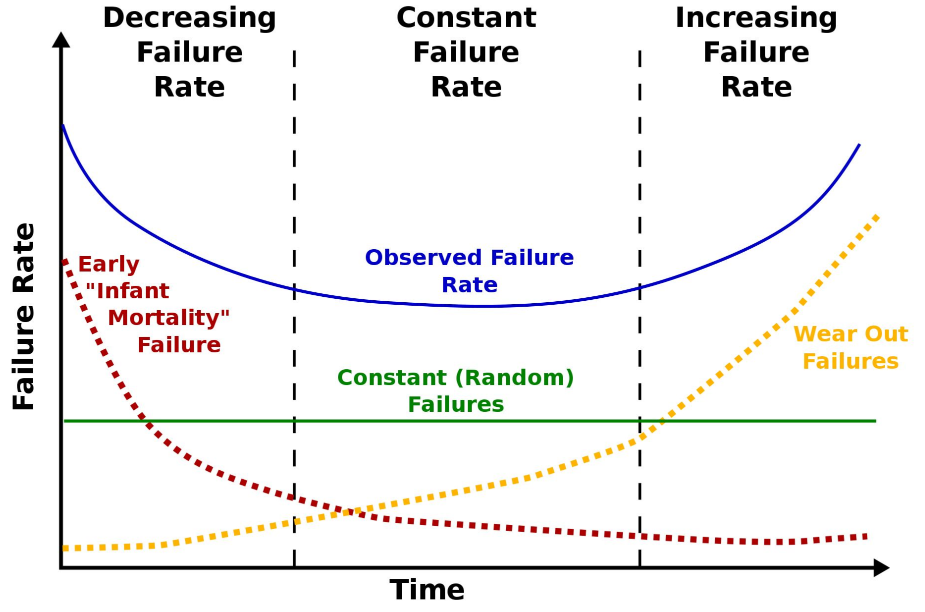
Using slides from https://wiki.aalto.fi/download/attachments/16221306/EmbeddedSystemsFaultToleranceBasics.pdf

Fault-Tolerance Steps

- Fault Detection
  - Determining that a fault has occurred
- Diagnosis
  - Determining what caused the fault, or exactly which subsystem or component is faulty
- Containment
  - Prevents the propagation of faults from their origin at one point in a system to a point where it can have an effect on the service to the user
- Masking
  - Insuring that only correct values get passed to the system boundary in spite of a failed component.
- Compensation
  - If a fault occurs and is confined to a subsystem, it may be necessary for the system to provide a response to compensate for output of the faulty subsystem.
- Repair
  - The process in which faults are removed from a system. In well-designed fault tolerant systems, faults are contained before they propagate to the extent that the delivery of system service is affected.
  - This leaves a portion of the system unusable because of residual faults.
  - If subsequent faults occur, the system may be unable to cope because of this loss of resources, unless these resources are reclaimed through a recovery process which insures that no faults remain in system resources or in the system state

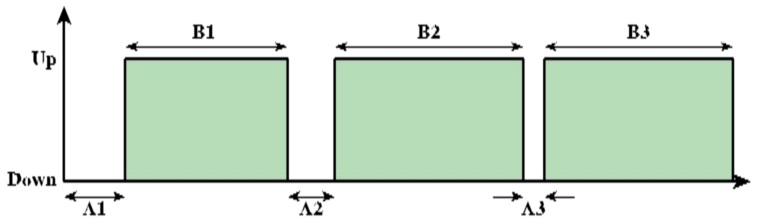
Fundamental Concepts

- Reliability R(t)
  - continuity of correct service
  - 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
  - readiness for correct service
  - The fraction of time the system is available to service users' requests
- Safety: absence of catastrophic consequences on the user(s) and the environment
- Integrity: absence of improper system alterations.
- Maintainability: ability to undergo modifications and repairs



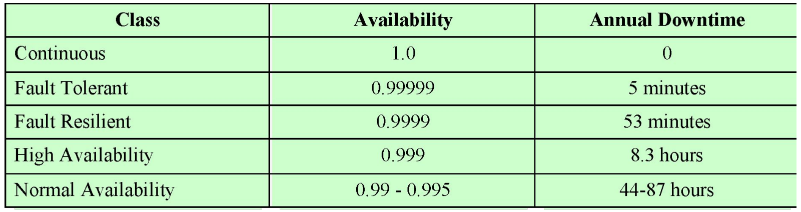
https://en.wikipedia.org/wiki/Bathtub_curve

Example and Classes



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

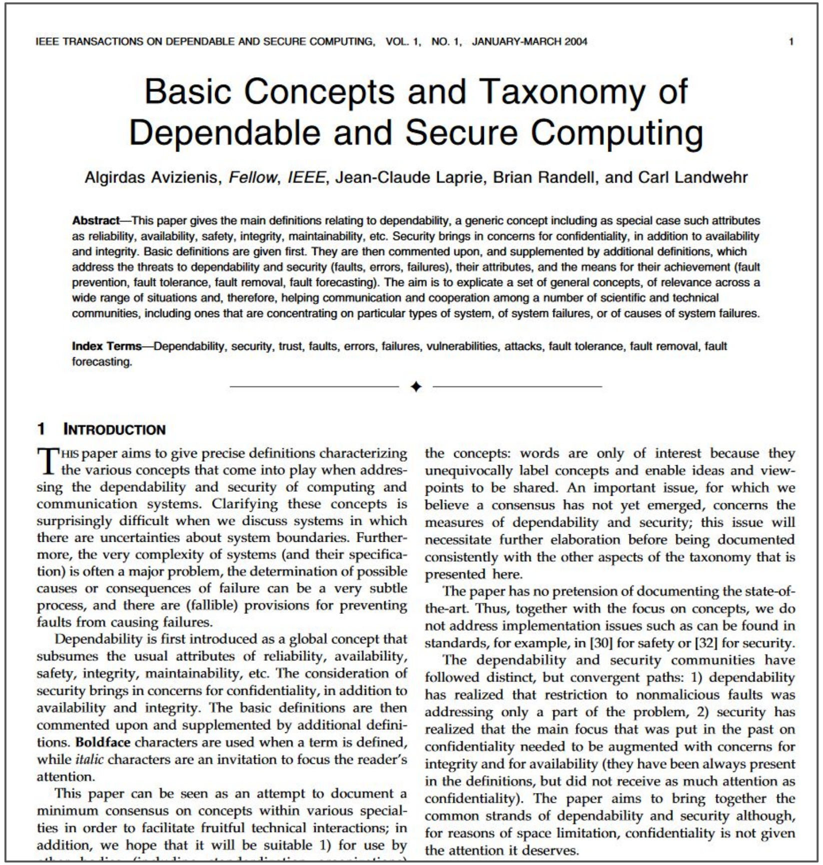
Figure 2.13 System Operational States



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

Fault Categories

- Permanent
  - A fault that, after it occurs, is always present
  - The fault persists until the faulty component is replaced or repaired
- Temporary
  - A fault that is not present all the time for all operating conditions
  - Can be classified as
    - Transient – a fault that occurs only once
    - Intermittent – a fault that occurs at multiple, unpredictable times



IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, VOL. 1, NO. 1, JANUARY-MARCH 2004 1

Basic Concepts and Taxonomy of Dependable and Secure Computing

Algirdas Avizienis, *Fellow*, *IEEE*, Jean-Claude Laprie, Brian Randell, and Carl Landwehr

Abstract—This paper gives the main definitions relating to dependability, a generic concept including as special case such attributes as reliability, availability, safety, integrity, maintainability, etc. Security brings in concerns for confidentiality, in addition to availability and integrity. Basic definitions are given first. They are then commented upon, and supplemented by additional definitions, which address the threats to dependability and security (faults, errors, failures), their attributes, and the means for their achievement (fault prevention, fault tolerance, fault removal, fault forecasting). The aim is to explicate a set of general concepts, of relevance across a wide range of situations and, therefore, helping communication and cooperation among a number of scientific and technical communities, including ones that are concentrating on particular types of system, of system failures, or of causes of system failures.

Index Terms—Dependability, security, trust, faults, errors, failures, vulnerabilities, attacks, fault tolerance, fault removal, fault forecasting.

1 INTRODUCTION

THIS paper aims to give precise definitions characterizing the various concepts that come into play when addressing the dependability and security of computing and communication systems. Clarifying these concepts is surprisingly difficult when we discuss systems in which there are uncertainties about system boundaries. Furthermore, the very complexity of systems (and their specification) is often a major problem, the determination of possible causes or consequences of failure can be a very subtle process, and there are (fallible) provisions for preventing faults from causing failures.

Dependability is first introduced as a global concept that subsumes the usual attributes of reliability, availability, safety, integrity, maintainability, etc. The consideration of security brings in concerns for confidentiality, in addition to availability and integrity. The basic definitions are then commented upon and supplemented by additional definitions. **Boldface** characters are used when a term is defined, while *italic* characters are an invitation to focus the reader's attention.

This paper can be seen as an attempt to document a minimum consensus on concepts within various specialties in order to facilitate fruitful technical interactions; in addition, we hope that it will be suitable 1) for use by

the concepts: words are only of interest because they unequivocally label concepts and enable ideas and viewpoints to be shared. An important issue, for which we believe a consensus has not yet emerged, concerns the measures of dependability and security; this issue will necessitate further elaboration before being documented consistently with the other aspects of the taxonomy that is presented here.

The paper has no pretension of documenting the state-of-the-art. Thus, together with the focus on concepts, we do not address implementation issues such as can be found in standards, for example, in [30] for safety or [32] for security.

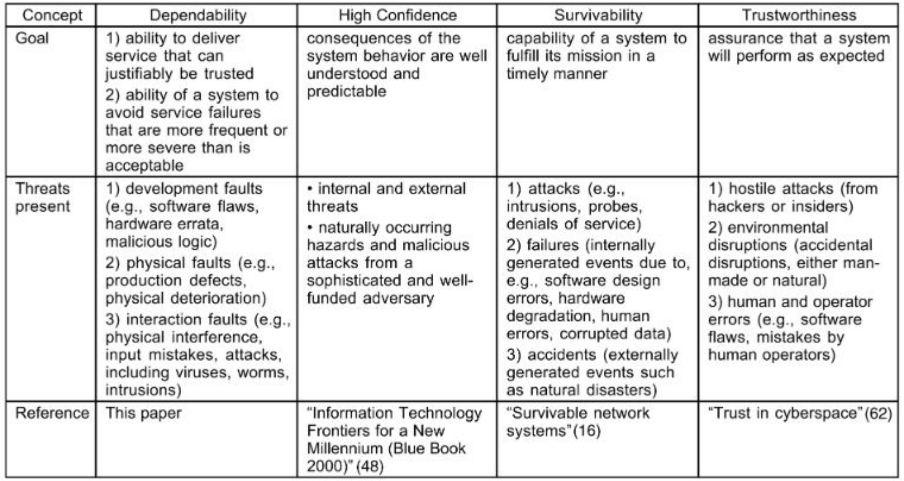
The dependability and security communities have followed distinct, but convergent paths: 1) dependability has realized that restriction to nonmalicious faults was addressing only a part of the problem, 2) security has realized that the main focus that was put in the past on confidentiality needed to be augmented with concerns for integrity and for availability (they have been always present in the definitions, but did not receive as much attention as confidentiality). The paper aims to bring together the common strands of dependability and security although, for reasons of space limitation, confidentiality is not given the attention it deserves.

Causes of Errors

- Nondeterminate program operation
  - When **programs share memory**, and their execution is interleaved by the processor, they may interfere with each other by overwriting common memory areas in unpredictable ways
  - **The order in which programs are scheduled** may affect the outcome of any particular program
- Improper synchronization
  - It is often the case that **a routine must be suspended awaiting an event elsewhere in the system**
  - Improper design of the signaling mechanism can result in loss or duplication
- Failed mutual exclusion
  - More than one user or program attempts to **make use of a shared resource at the same time**
  - There must be some sort of mutual exclusion mechanism that permits only one routine at a time to perform an update against the file
- **Deadlocks**
  - It is possible for two or more programs to be hung up waiting for each other

Dependability and RAS (Reliability, Availability and Serviceability)

- A system should
  - be capable of detecting hardware errors, and, when possible correcting them in runtime
  - provide mechanisms to detect hardware degradation, in order to warn the system administrator to take the action of replacing a component before it causes data loss or system downtime
- **Dependability**
  - original definition: is the ability to deliver service that can justifiably be trusted
  - alternate definition: the ability to avoid service failures that are more frequent and more severe than is acceptable
- The dependence of system A on system B, thus, represents the extent to which system A's dependability is (or would be) affected by that of System B.
- The concept of dependence leads to that of trust, which can very conveniently be defined as accepted dependence.



Concept	Dependability	High Confidence	Survivability	Trustworthiness
Goal	1) ability to deliver service that can justifiably be trusted 2) ability of a system to avoid service failures that are more frequent or more severe than is acceptable	consequences of the system behavior are well understood and predictable	capability of a system to fulfill its mission in a timely manner	assurance that a system will perform as expected
Threats present	1) development faults (e.g., software flaws, hardware errata, malicious logic) 2) physical faults (e.g., production defects, physical deterioration) 3) interaction faults (e.g., physical interference, input mistakes, attacks, including viruses, worms, intrusions)	• internal and external threats • naturally occurring hazards and malicious attacks from a sophisticated and well-funded adversary	1) attacks (e.g., intrusions, probes, denials of service) 2) failures (internally generated events due to, e.g., software design errors, hardware degradation, human errors, corrupted data) 3) accidents (externally generated events such as natural disasters)	1) hostile attacks (from hackers or insiders) 2) environmental disruptions (accidental disruptions, either man-made or natural) 3) human and operator errors (e.g., software flaws, mistakes by human operators)
Reference	This paper	"Information Technology Frontiers for a New Millennium (Blue Book 2000)" (48)	"Survivable network systems" (16)	"Trust in cyberspace" (62)

Fig. 15. Dependability, high confidence, survivability, and trustworthiness.

https://www.kernel.org/doc/html/v4.12/admin-guide/ras.html

Methods of Redundancy - No single point of failure

- Design the system with multiple instances of critical units in such a manner that the failure of some of these units does not directly fail the entire system.
- A number of techniques can be incorporated into OS software to support fault tolerance:
  - Process isolation
  - Concurrency controls
  - Virtual machines
  - Checkpoints and rollbacks

Spatial (physical) redundancy

Involves the use of multiple components that either perform the same function simultaneously or are configured so that one component is available as a backup in case of the failure of another component

Temporal redundancy

Involves repeating a function or operation when an error is detected

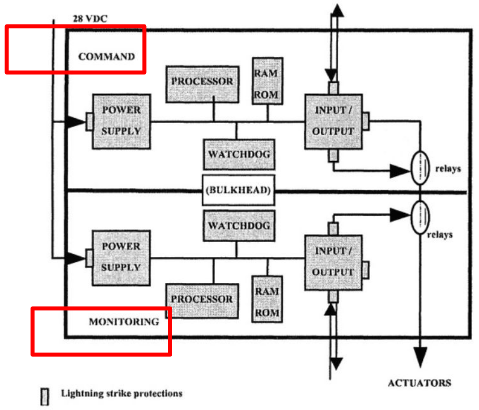
Is effective with temporary faults but not useful for permanent faults

Information redundancy

Provides fault tolerance by replicating or coding data in such a way that bit errors can be both detected and corrected

Example: Fly-by-wire and redundancy

- The first electrical flight control system (a.k.a. Fly-by-Wire) for a civil aircraft was designed by Aerospatiale and installed on Concorde.
- Airbus A320 entered operation in 1988
  - Other models include A340 and A380
  - Primary (P) and secondary (S) computers (different designs and suppliers)
  - Multiple redundant software modules



https://link.springer.com/content/pdf/10.1007/978-1-4020-8157-6_18.pdf

Two types of computers

- PRIM's (primary computers)
- SEC's (secondary computers)

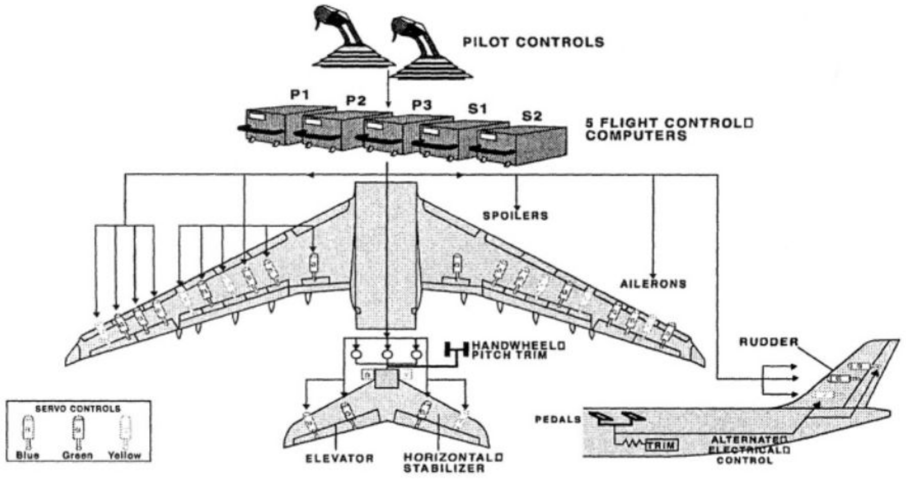


Figure 2: A340-600 system architecture

Symmetric Multiprocessor OS Considerations

- A multiprocessor OS must provide all the functionality of a multiprogramming system plus additional features to accommodate multiple processors

- **Key design issues:**

Simultaneous concurrent processes or threads

Kernel routines need to be reentrant to allow several processors to execute the same kernel code simultaneously

Scheduling

Any processor may perform scheduling, which complicates the task of enforcing a scheduling policy

Synchronization

With multiple active processes having potential access to shared address spaces or shared I/O resources, care must be taken to provide effective synchronization

Memory management

The reuse of physical pages is the biggest problem of concern

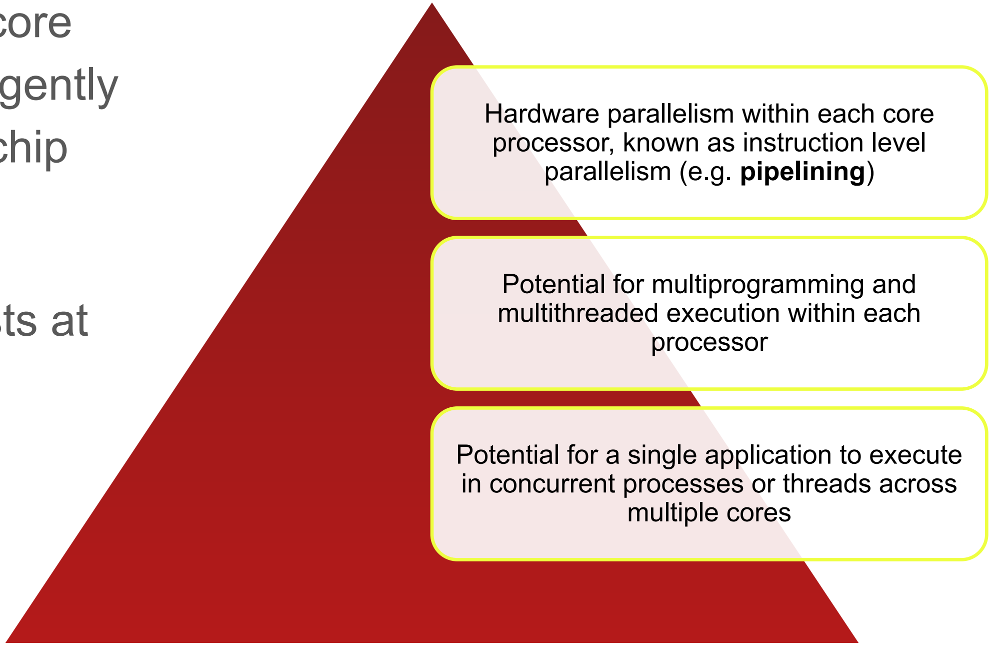
Reliability and fault tolerance

The OS should provide graceful degradation in the face of processor failure

Parallelism and Multicore OS Considerations

- The design challenge for a many-core multicore system is to efficiently harness the multicore processing power and intelligently manage the substantial on-chip resources efficiently

- Potential for parallelism exists at three levels:


Hardware parallelism within each core processor, known as instruction level parallelism (e.g. **pipelining**)
Potential for multiprogramming and multithreaded execution within each processor
Potential for a single application to execute in concurrent processes or threads across multiple cores

Instruction Level Parallelism (ILP): Pipelining

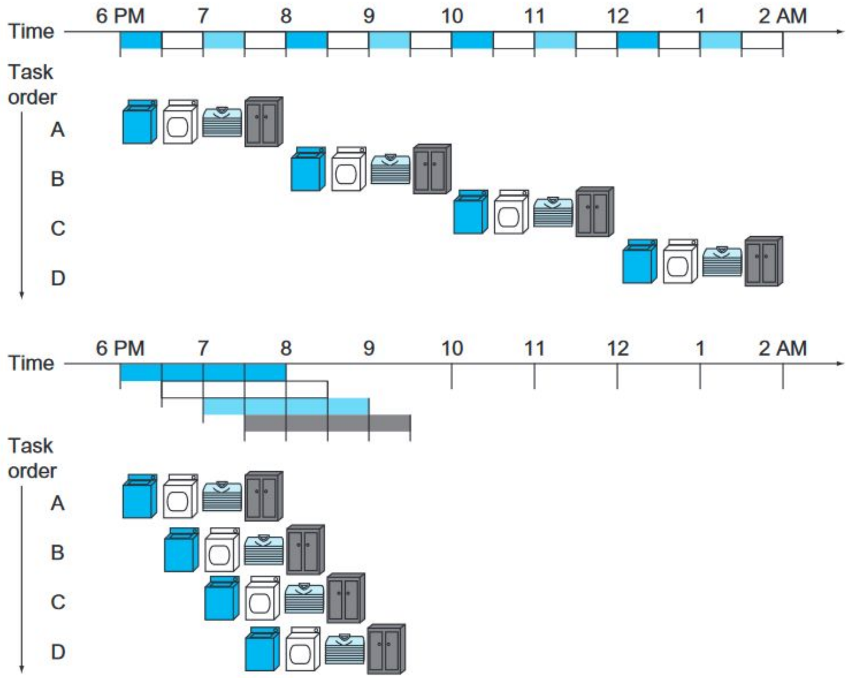


FIGURE 4.25 The laundry analogy for pipelining. Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, "folder," and "storer" each take 30 minutes for their task. Sequential laundry takes 8 hours for 4 loads of wash, while pipelined laundry takes just 3.5 hours. We show the pipeline stage of different loads over time by showing copies of the four resources on this two-dimensional time line, but we really have just one of each resource.

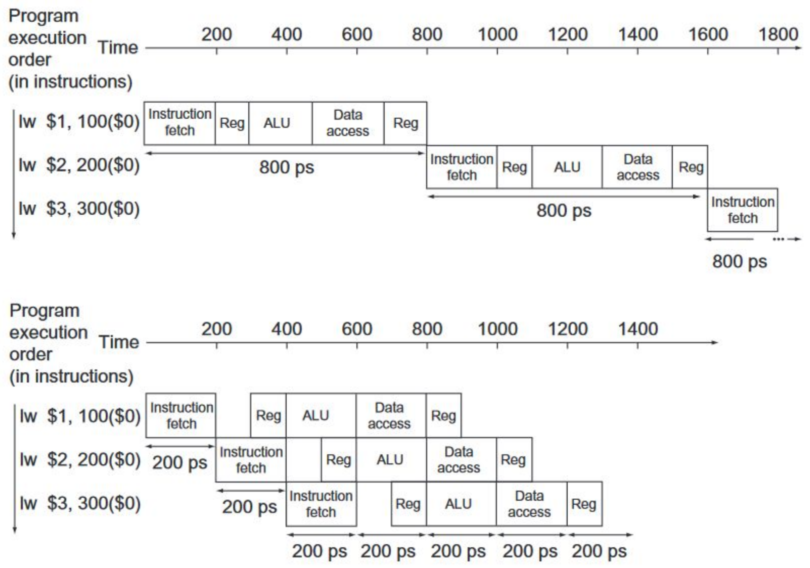


FIGURE 4.27 Single-cycle, nonpipelined execution in top versus pipelined execution in bottom. Both use the same hardware components, whose time is listed in Figure 4.26. In this case, we see a fourfold speed-up on average time between instructions, from 800 ps down to 200 ps. Compare this figure to Figure 4.25. For the laundry, we assumed all stages were equal. If the dryer were slowest, then the dryer stage would set the stage time. The pipeline stage times of a computer are also limited by the slowest resource, either the ALU operation or the memory access. We assume the write to the register file occurs in the first half of the clock cycle and the read from the register file occurs in the second half. We use this assumption throughout this chapter.

Virtualization and Clouds

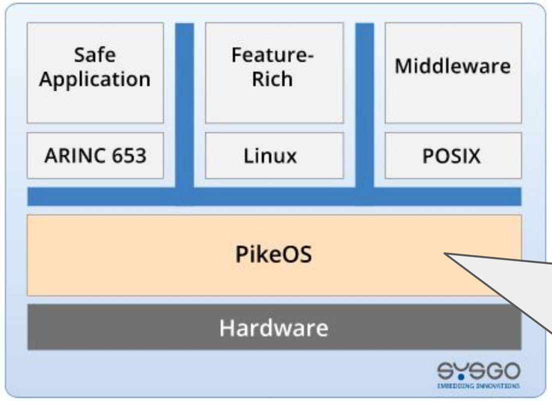
- Traditional cloud computing is about
  - IaaS (Infrastructure as a Service)
    - Virtual machines, storage, communication and balancing, etc.
  - PaaS (Platform as a Service)
    - Runtimes, servers, etc. and the related development tools
  - SaaS (Software as a Service)
    - Basically, on-demand software
- Also, cluster computing uses layers
  - Technically many such setup are based on virtualization
- Virtualization: using virtual (abstract) resources instead of the concrete resources (on which they are based)

- Allows one or more cores to be dedicated to a particular process and then leave the processor alone to devote its efforts to that process
- Multicore OS could then act as a hypervisor that makes a high-level decision to allocate cores to applications but does little in the way of resource allocation beyond that

Virtualization - Hypervisor

The virtual machine monitor (VMM), or **hypervisor**

- runs on top of (or is incorporated into) the host OS
- supports VMs, which are emulated hardware devices
- each VM runs a separate OS
- handles each operating system's communications with the processor, the storage medium, and the network
- hands off the processor control to a virtual OS on a VM



PikeOS is a combination of hypervisor and real-time operating system and is ARINC 653 standard compliant.

ARINC 653 standard defines virtualization in terms of static resources (memory, I/O) as well as processor time. It also defines an API suitable for Avionics concepts. The Airbus A380 and the Boeing A787 are examples of the IMA concept success.

https://www.sysgo.com/virtualization

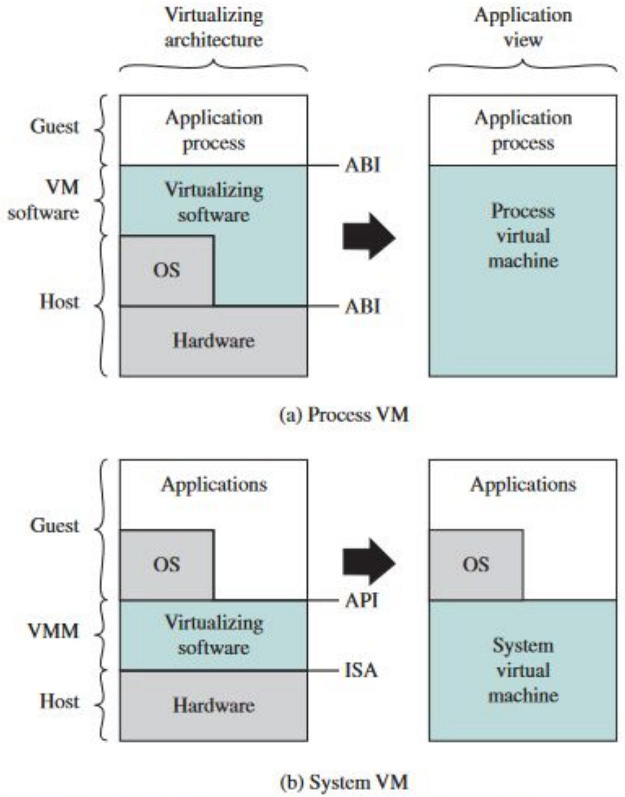
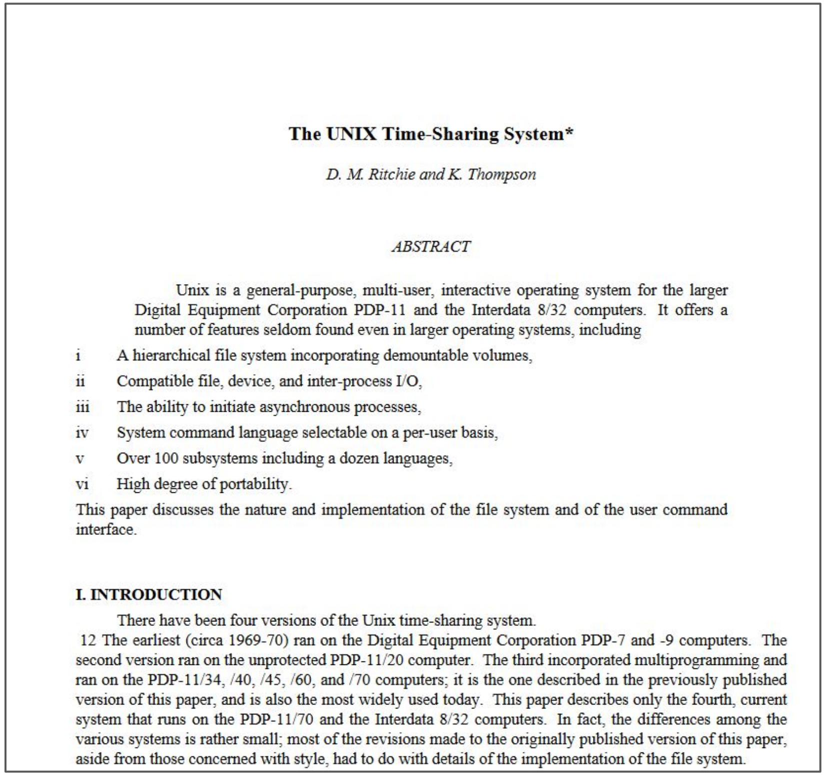


Figure 2.14 Process and System Virtual Machines

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



The UNIX Time-Sharing System*

D. M. Ritchie and K. Thompson

ABSTRACT

Unix is a general-purpose, multi-user, interactive operating system for the larger Digital Equipment Corporation PDP-11 and the Interdata 8/32 computers. It offers a number of features seldom found even in larger operating systems, including

i A hierarchical file system incorporating demountable volumes,
ii Compatible file, device, and inter-process I/O,
iii The ability to initiate asynchronous processes,
iv System command language selectable on a per-user basis,
v Over 100 subsystems including a dozen languages,
vi High degree of portability.

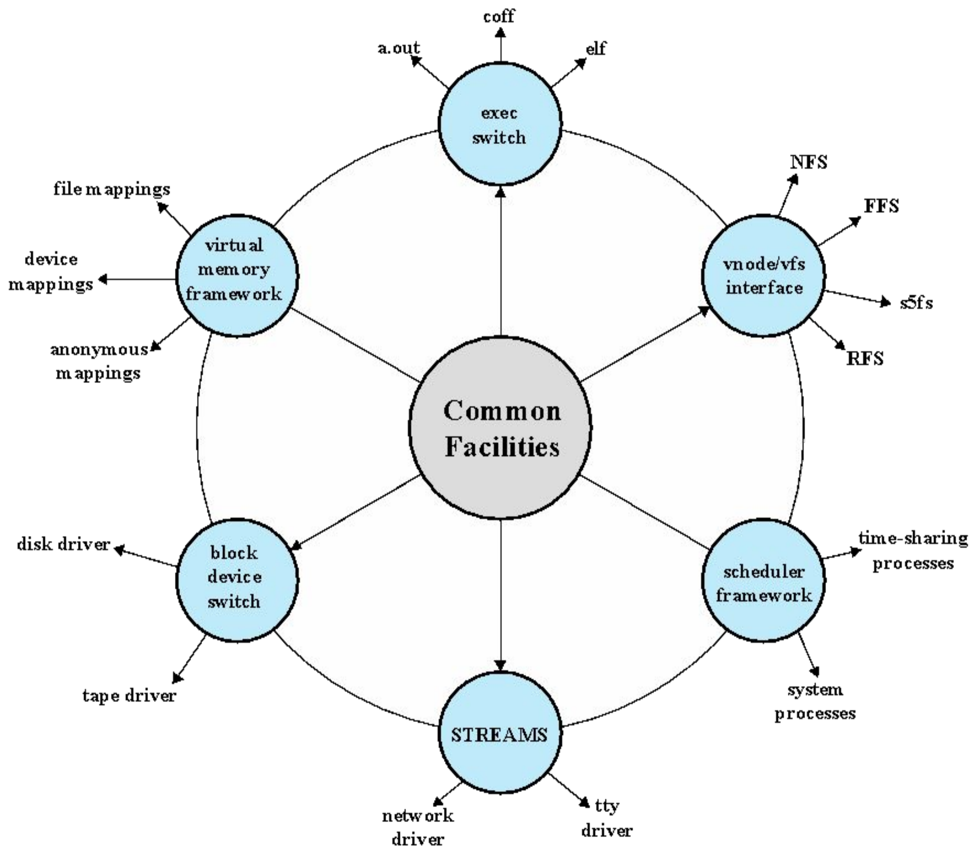
This paper discusses the nature and implementation of the file system and of the user command interface.

I. INTRODUCTION

There have been four versions of the Unix time-sharing system.
12 The earliest (circa 1969-70) ran on the Digital Equipment Corporation PDP-7 and -9 computers. The second version ran on the unprotected PDP-11/20 computer. The third incorporated multiprogramming and ran on the PDP-11/34, /40, /45, /60, and /70 computers; it is the one described in the previously published version of this paper, and is also the most widely used today. This paper describes only the fourth, current system that runs on the PDP-11/70 and the Interdata 8/32 computers. In fact, the differences among the various systems is rather small; most of the revisions made to the originally published version of this paper, aside from those concerned with style, had to do with details of the implementation of the file system.

https://www.bell-labs.com/usr/dmr/www/cacm.pdf

Modern Unix Kernel



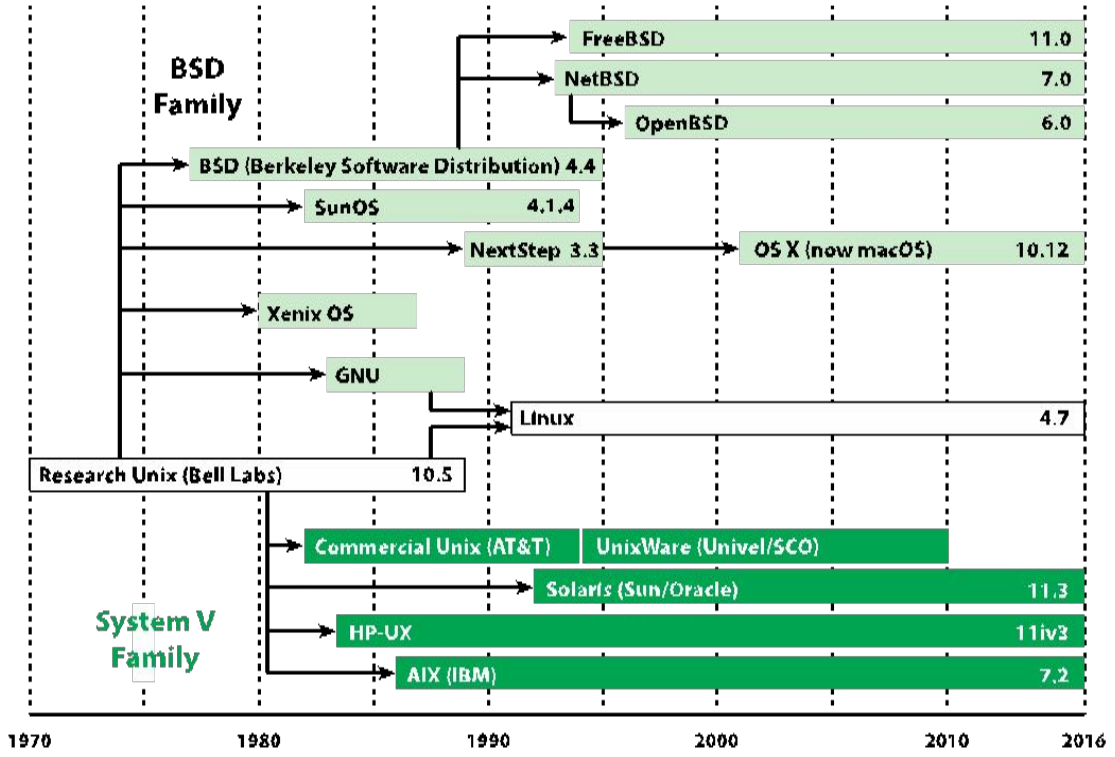
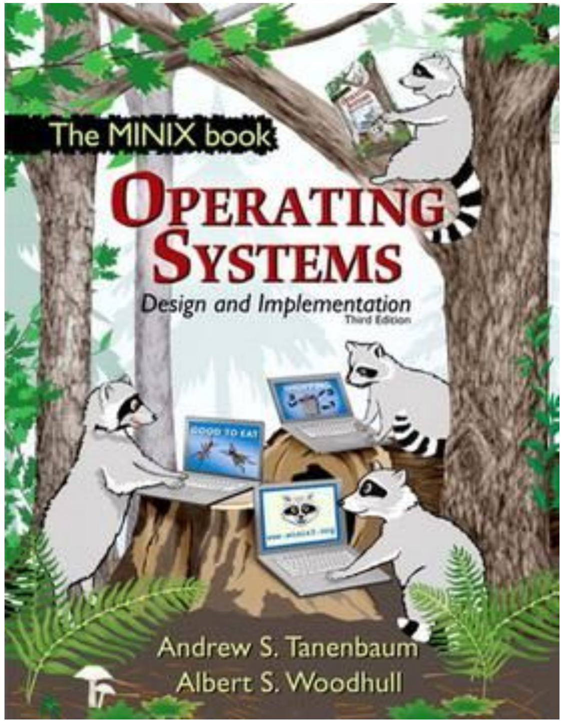


Figure 2.17 Unix Family Tree

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



Modular Structure – Loadable Modules

- 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
- 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

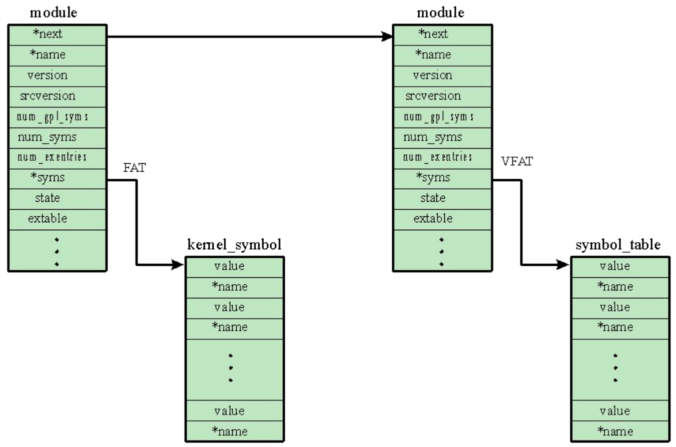


Figure 2.18 Example List of Linux Kernel Modules

Linux kernel components

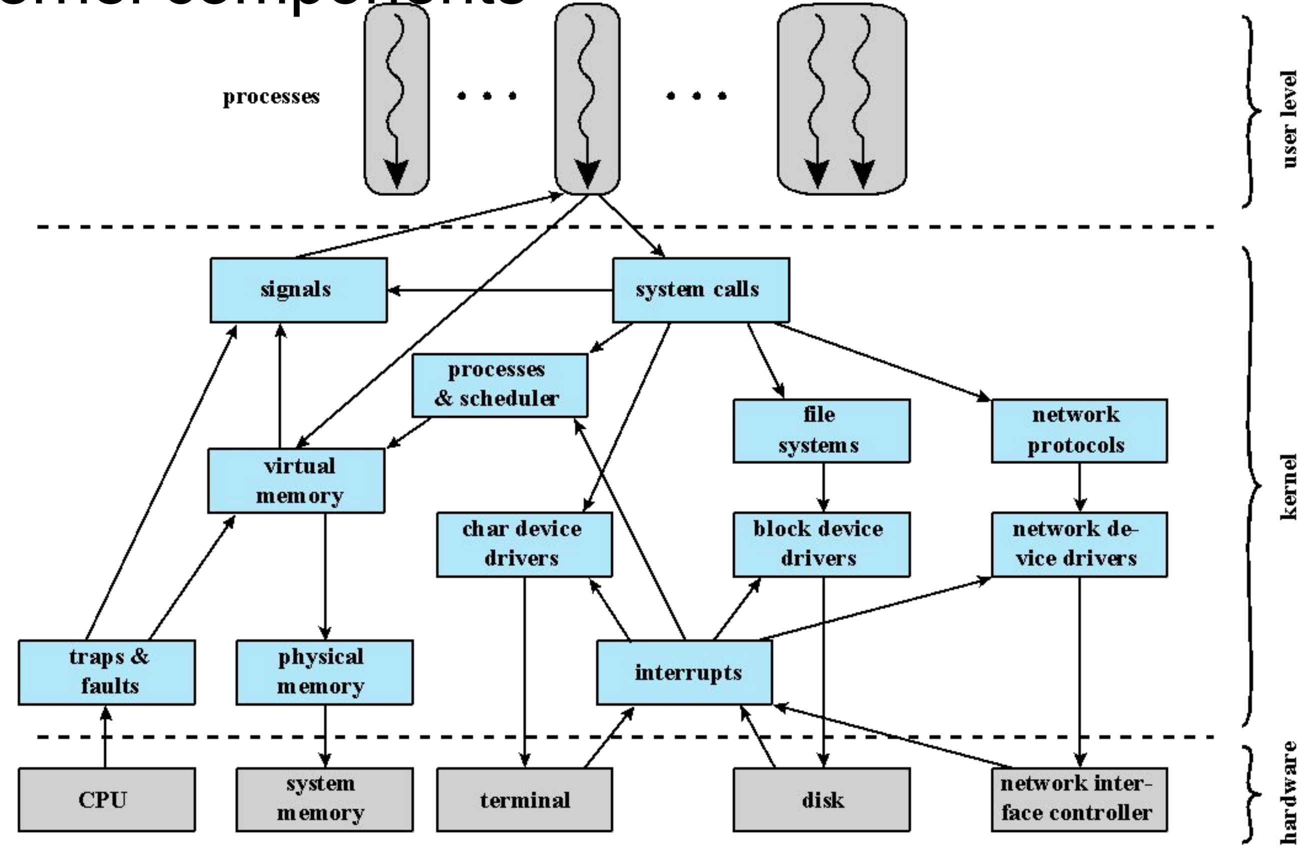


Figure 2.19 Linux Kernel Components

Linux Signals and System Calls

SIGHUP	Terminal hangup	SIGCONT	Continue
SIGQUIT	Keyboard quit	SIGTSTP	Keyboard stop
SIGTRAP	Trace trap	SIGTTOU	Terminal write
SIGBUS	Bus error	SIGXCPU	CPU limit exceeded
SIGKILL	Kill signal	SIGVTALRM	Virtual alarm clock
SIGSEGV	Segmentation violation	SIGWINCH	Window size unchanged
SIGPIPT	Broken pipe	SIGPWR	Power failure
SIGTERM	Termination	SIGRTMIN	First real-time signal
SIGCHLD	Child status unchanged	SIGRTMAX	Last real-time signal

Filesystem related	
close	Close a file descriptor.
link	Make a new name for a file.
open	Open and possibly create a file or device.
read	Read from file descriptor.
write	Write to file descriptor
Process related	
execve	Execute program.
exit	Terminate the calling process.
getpid	Get process identification.
setuid	Set user identity of the current process.
ptrace	Provides a means by which a parent process my observe and control the execution of another process, and examine and change its core image and registers.
Scheduling related	
sched_getparam	Sets the scheduling parameters associated with the scheduling policy for the process identified by `pid`.
sched_get_priority_max	Returns the maximum priority value that can be used with the scheduling algorithm identified by `policy`.
sched_setscheduler	Sets both the scheduling policy (e.g., FIFO) and the associated parameters for the process `pid`.
sched_rr_get_interval	Writes into the timespec structure pointed to by the parameter `tp` the round robin time quantum for the process `pid`.
sched_yield	A process can relinquish the processor voluntarily without blocking via this system call. The process will then be moved to the end of the queue for its static priority and a new process gets to run.

Interprocess Communication (IPC) related	
msgrcv	A message buffer structure is allocated to receive a message. The system call then reads a message from the message queue specified by msqid into the newly created message buffer.
semctl	Performs the control operation specified by cmd on the semaphore set semid.
semop	Performs operations on selected members of the semaphore set semid.
shmat	Attaches the shared memory segment identified by shmid to the data segment of the calling process.
shmctl	Allows the user to receive information on a shared memory segment, set the owner, group, and permissions of a shared memory segment, or destroy a segment.
Socket (networking) related	
bind	Assigns the local IP address and port for a socket. Returns 0 for success and –1 for error.
connect	Establishes a connection between the given socket and the remote socket associated with sockaddr.
gethostname	Returns local host name.
send	Send the bytes contained in buffer pointed to by *msg over the given socket.
setsockopt	Sets the options on a socket
Miscellaneous	
fsync	Copies all in-core parts of a file to disk, and waits until the device reports that all parts are on stable storage.
time	Returns the time in seconds since January 1, 1970.
vhangup	Simulates a hangup on the current terminal. This call arranges for other users to have a "clean" tty at login time.

Summary

- Hardware types
- Operating system objectives and functions
    - User/computer interface
    - Resource manager
        - Evolution of operating systems
    - Serial processing
    - Simple/multiprogrammed/time-sharing batch systems
        - Major achievements
        - Fault tolerance
    - Fundamental concepts
    - Faults
    - OS mechanisms
- Traditional Unix systems
    - History/description
- Linux
    - History
    - Modular structure
    - Kernel components

Process Description and Control

Process Elements

- Resources are made available to multiple applications
  - The processor is switched among multiple applications so all will appear to be progressing
  - The processor and I/O devices can be used efficiently
- Two essential elements of a process are:
  - Code: when the processor begins to execute the program code, we refer to this executing entity as a process
  - Data: which may be shared with other processes that are executing the same program

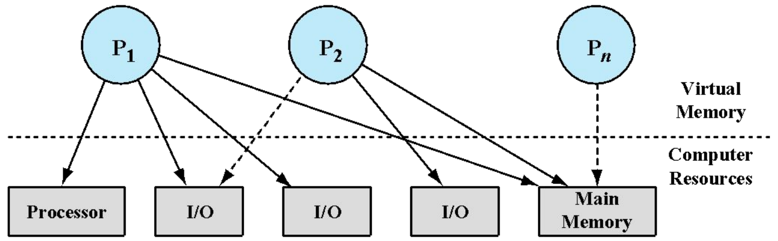


Figure 3.10 Processes and Resources (resource allocation at one snapshot in time)

Program code

A set of data associated with that code

Tables within the OS

- Memory tables:
  - keep track of both main (real) and secondary (virtual) memory
  - processes use some sort of virtual memory and/or simple swapping mechanism
- The memory tables must include the following information:
  - The allocation of main memory to processes
  - The allocation of secondary memory to processes
- I/O tables
  - used by the OS to manage the I/O devices and channels of the computer system.
  - At any given time, an I/O device may be available or assigned to a particular process.
  - If an I/O operation is in progress, the OS needs to know the status of the I/O operation and the location in main memory being used as the source or destination of the I/O transfer.
- File tables
  - provide information about the existence of files, their location on secondary memory, their current status, and other attributes
  - Much, if not all, of this information may be maintained by a file management system
- The OS must maintain process tables to manage processes

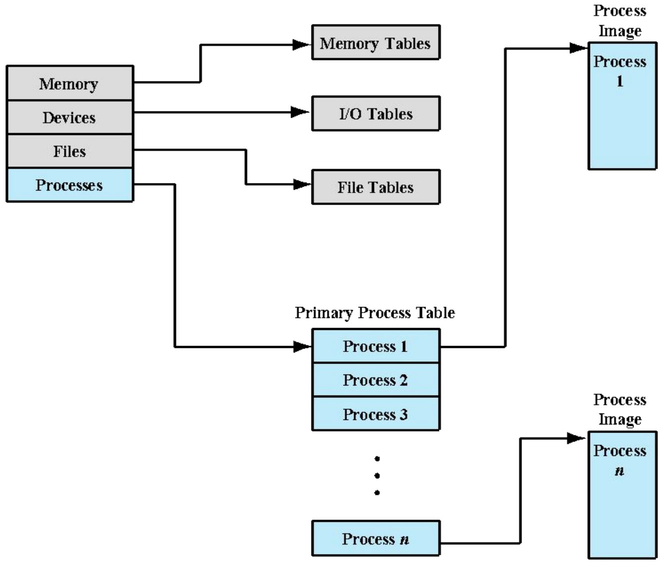
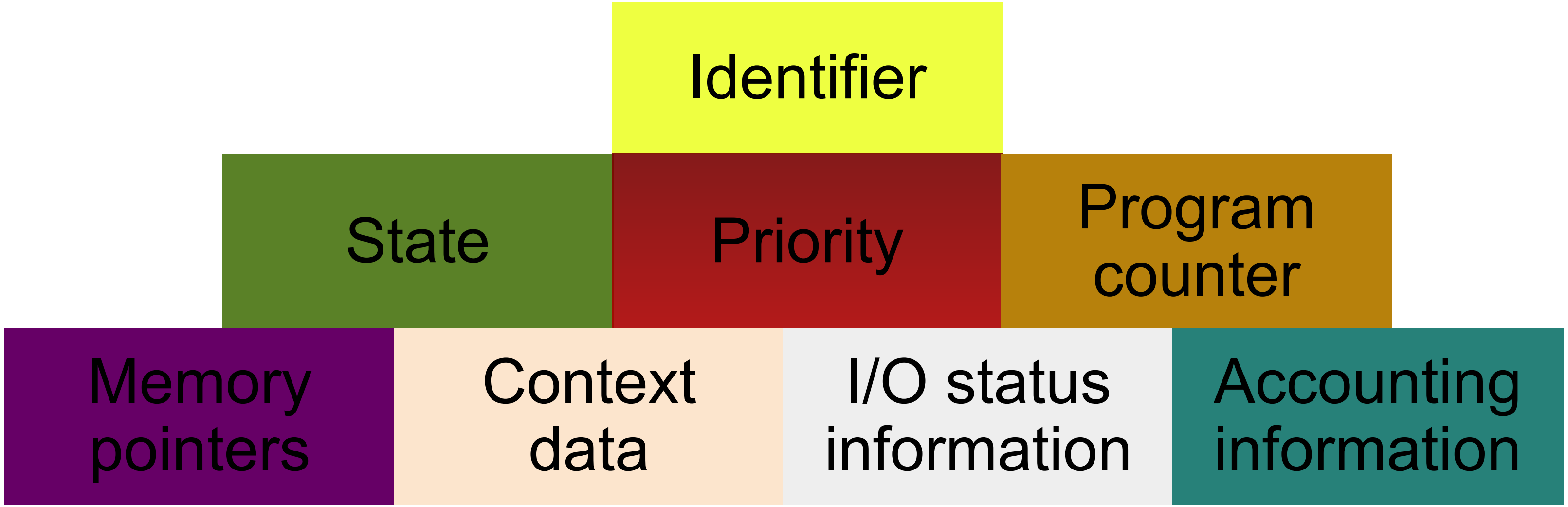


Figure 3.11 General Structure of Operating System Control Tables

Process Elements

- While the program is executing, a process can be uniquely characterized by a number of elements, including:



Process Control Block

- Contains the process elements
- It is possible to interrupt a running process and later resume execution as if the interruption had not occurred
- Created and managed by the operating system
- Key tool that allows support for multiple processes

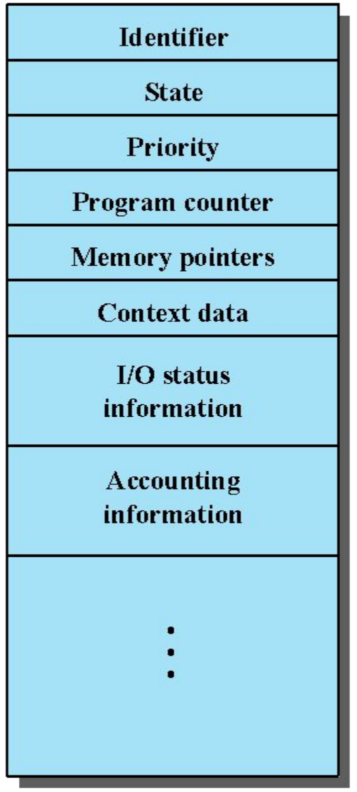


Figure 3.1 Simplified Process Control Block

Process States

Process States

Trace

The behavior of an individual process by listing the sequence of instructions that execute for that process

The behavior of the processor can be characterized by showing how the traces of the various processes are interleaved

Dispatcher

Small program that switches the processor from one process to another

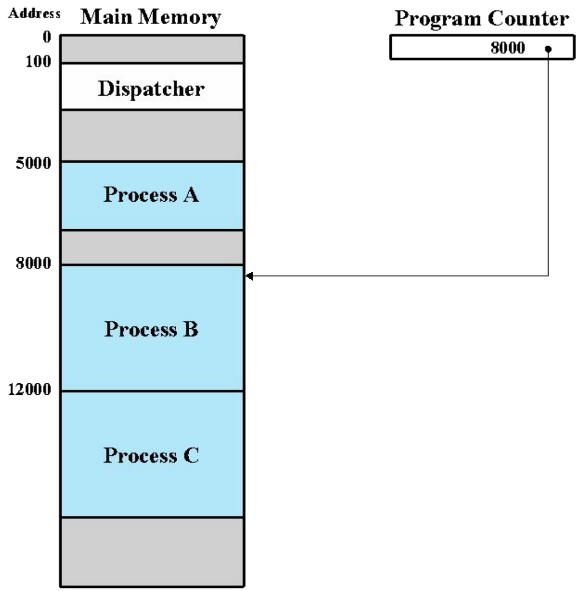


Figure 3.2 Snapshot of Example Execution (Figure 3.4) at Instruction Cycle 13

Process Execution

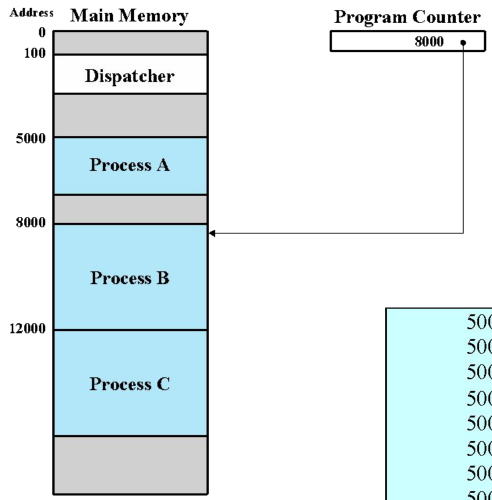


Figure 3.2 Snapshot of Example Execution at Instruction Cycle 13

(a) Trace of Process A	(b) Trace of Process B	(c) Trace of Process C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

Figure 3.3 Traces of Processes of Figure 3.2

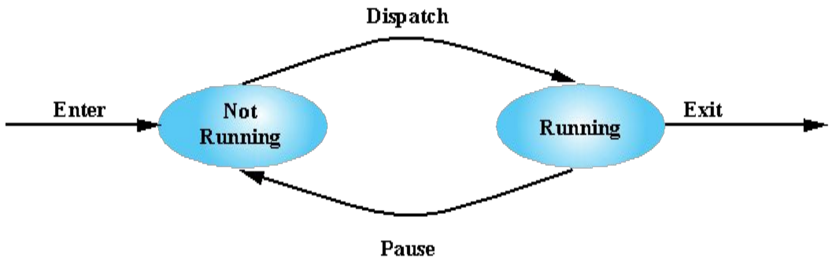
```
1    5000                        27   12004
2    5001                        28   12005
3    5002                        -------------------- Timeout
4    5003                        29   100
5    5004                        30   101
6    5005                        31   102
-------------------- Timeout     32   103
7    100                         33   104
8    101                         34   105
9    102                         35   5006
10   103                         36   5007
11   104                         37   5008
12   105                         38   5009
13   8000                        39   5010
14   8001                        40   5011
15   8002                        -------------------- Timeout
16   8003                        41   100
----------------I/O Request      42   101
17   100                         43   102
18   101                         44   103
19   102                         45   104
20   103                         46   105
21   104                         47   12006
22   105                         48   12007
23   12000                       49   12008
24   12001                       50   12009
25   12002                       51   12010
26   12003                       52   12011
                                 -------------------- Timeout
```

100 = Starting address of dispatcher program

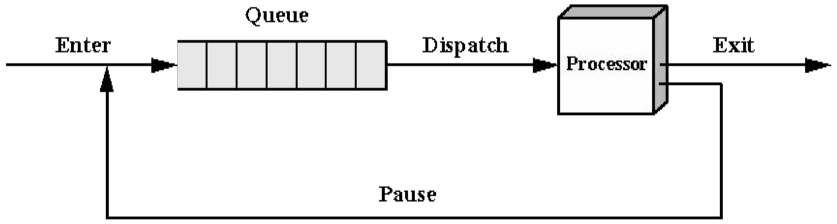
Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

Figure 3.4 Combined Trace of Processes of Figure 3.2

Two-State Process Model



(a) State transition diagram

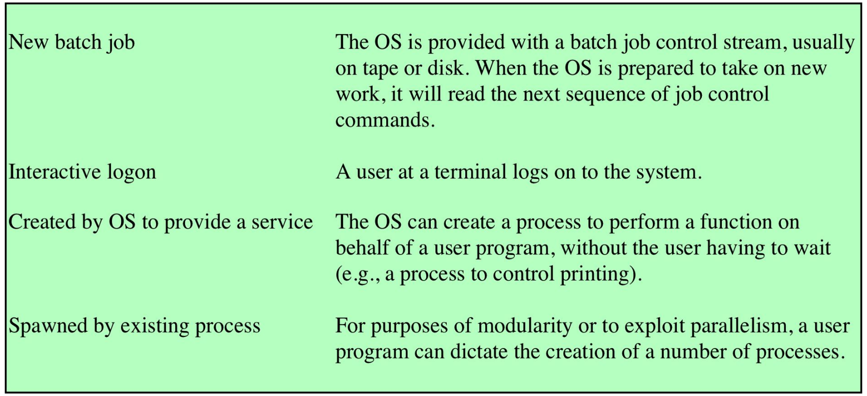


(b) Queuing diagram

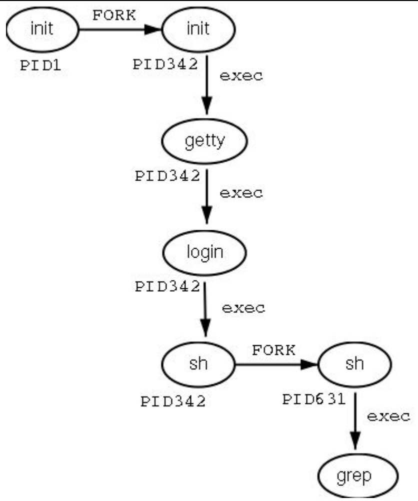
Figure 3.5 Two-State Process Model

Process Creation

- Process spawning
  - When the OS creates a process at the explicit request of another process
  - Parent process: is the original, creating, process
  - Child process: is the new process
- Linux
  - An existing process makes an exact copy of itself
  - The child process has the same environment as its parent, only the process ID number is different. This procedure is called **forking**.
  - After the forking process, the **address space** of the child process is overwritten with the new process data. This is done through an **exec** call to the system.
  - Many programs, for instance, **daemonize** their child processes, so they can keep on running when the parent stops or is being stopped.



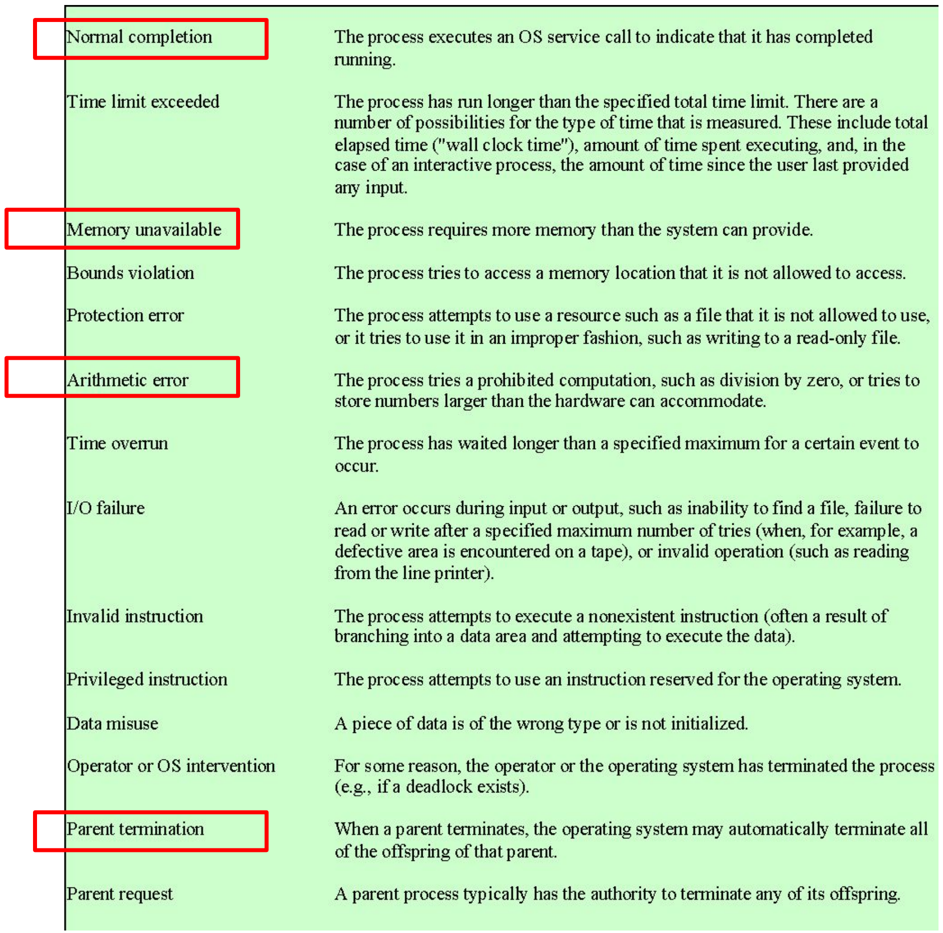
New batch job	The OS is provided with a batch job control stream, usually on tape or disk. When the OS is prepared to take on new work, it will read the next sequence of job control commands.
Interactive logon	A user at a terminal logs on to the system.
Created by OS to provide a service	The OS can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.



https://www.linuxtopia.org/online_books/introduction_to_linux/linux_Process_creation.html

Process Termination

- There must be a means for a process to indicate its completion
- A batch job should include a HALT instruction or an explicit OS service call for termination
- For an interactive application, the action of the user will indicate when the process is completed (e.g. log off, quitting an application)
- Linux:
  - Program returns its exit status to the parent
  - This exit status is a number providing the results of the program's execution
  - The system of returning information upon executing a job has its origin in the C programming language in which UNIX has been written.



Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (e.g., if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

https://www.linuxtopia.org/online_books/introduction_to_linux/linux_Ending_processes.html

Five-State Process Model

- **New**: A process that has just been created but has not yet been admitted to the pool of executable processes by the OS. Typically, a new process has not yet been loaded into main memory, although its process control block has been created.
- **Ready**: A process that is prepared to execute when given the opportunity.
- **Running**: The process that is currently being executed.
- **Blocked/Waiting**: A process that cannot execute until some event occurs, such as the completion of an I/O operation.
- **Exit**: A process that has been released from the pool of executable processes by the OS, either because it halted or because it aborted for some reason.

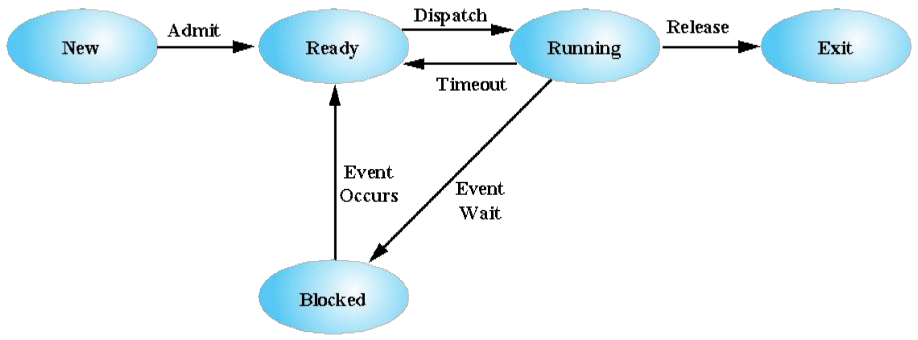


Figure 3.6 Five-State Process Model

Example: Trace of Process Execution

```
1    5000                      27   12004
2    5001                      28   12005
3    5002                      -------------------- Timeout
4    5003                      29   100
5    5004                      30   101
6    5005                      31   102
-------------------- Timeout   32   103
7    100                       33   104
8    101                       34   105
9    102                       35   5006
10   103                       36   5007
11   104                       37   5008
12   105                       38   5009
13   8000                      39   5010
14   8001                      40   5011
15   8002                      -------------------- Timeout
16   8003                      41   100
----------------I/O Request    42   101
17   100                       43   102
18   101                       44   103
19   102                       45   104
20   103                       46   105
21   104                       47   12006
22   105                       48   12007
23   12000                     49   12008
24   12001                     50   12009
25   12002                     51   12010
26   12003                     52   12011
                               -------------------- Timeout
```

100 = Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

Figure 3.4 Combined Trace of Processes of Figure 3.2

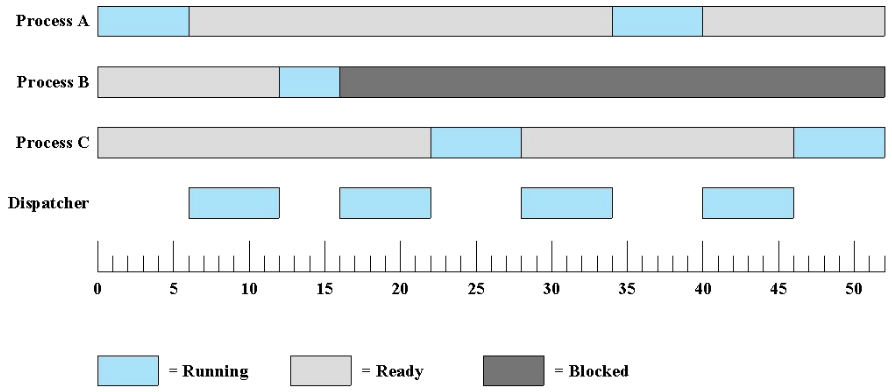


Figure 3.7 Process States for Trace of Figure 3.4

Faster/Efficient Scheduling of Processes

- Example, Linux Scheduler
  - separate run queue for each processor
  - each processor only selects processes from its own queue to run
  - it's possible for one processor to be idle while others have jobs waiting in their run queues
  - Periodically, the queues are rebalanced:
    - if one processor's run queue is too long
    - some processes are moved from it to another processor's queue
- 140 separate queues, one for each priority level
  - Number can be changed at a given site
  - Two sets, active and expired
  - Priorities 0-99 for real-time processes
  - Priorities 100-139 for normal processes

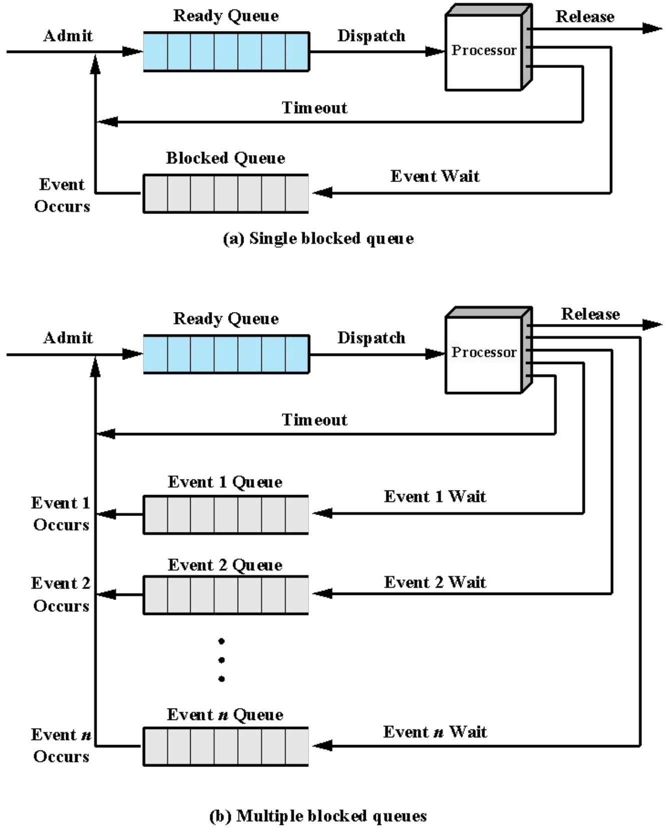
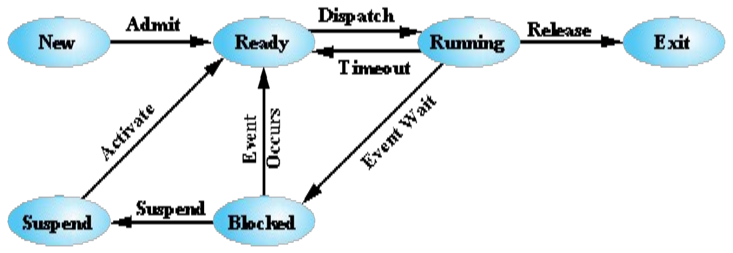


Figure 3.8 Queuing Model for Figure 3.6

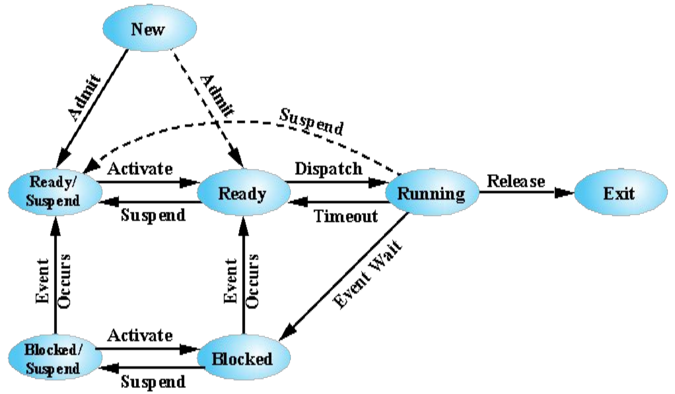
https://www.cs.columbia.edu/~smb/classes/s06-4118/l13.pdf

Suspended Processes - Swapping

- Swapping
  - is an I/O operation and therefore there is the potential for making the problem worse, not better
  - disk I/O is generally the fastest I/O on a system swapping will usually enhance performance.
- Moving part of all of a process from main memory to disk
  - When none of the processes in main memory is in the Ready state, the OS swaps one of the blocked processes out on to disk into a suspend queue
  - This is a queue of existing processes that have been temporarily kicked out of main memory, or suspended
  - The OS then brings in another process from the suspend queue or it honors a new-process request
  - Execution then continues with the newly arrived process



(a) With One Suspend State



(b) With Two Suspend States

Figure 3.9 Process State Transition Diagram with Suspend States

Characteristics of a Suspended Process

- The process may or may not be waiting on an event
- The process is not immediately available for execution
- The process was placed in a suspended state by an agent: either itself, a parent process, or the OS, for the purpose of preventing its execution
- The process may not be removed from this state until the agent explicitly orders the removal

Swapping	The OS needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS reason	The OS may suspend a background or utility process or a process that is suspected of causing a problem.
Interactive user request	A user may wish to suspend execution of a program for purposes of debugging or in connection with the use of a resource.
Timing	A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time interval.
Parent process request	A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendants.

Process Control

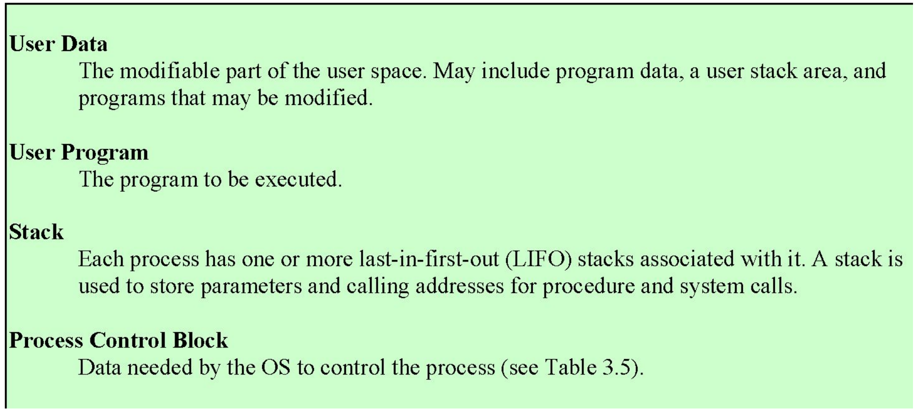
Process Control Structures

- A process must include a program or set of programs to be executed
  - A process will consist of at least sufficient memory to hold the programs and data of that process
  - The execution of a program typically involves a stack that is used to keep track of procedure calls and parameter passing between procedures
- Each process has associated with it a number of attributes that are used by the OS for process control
  - The collection of program, data, stack, and attributes is referred to as the process image
  - Process image location will depend on the memory management scheme being used

To manage and control a process the OS must know:

- Where the process is located
- The attributes of the process that are necessary for its management

Typical Elements of a Process Image

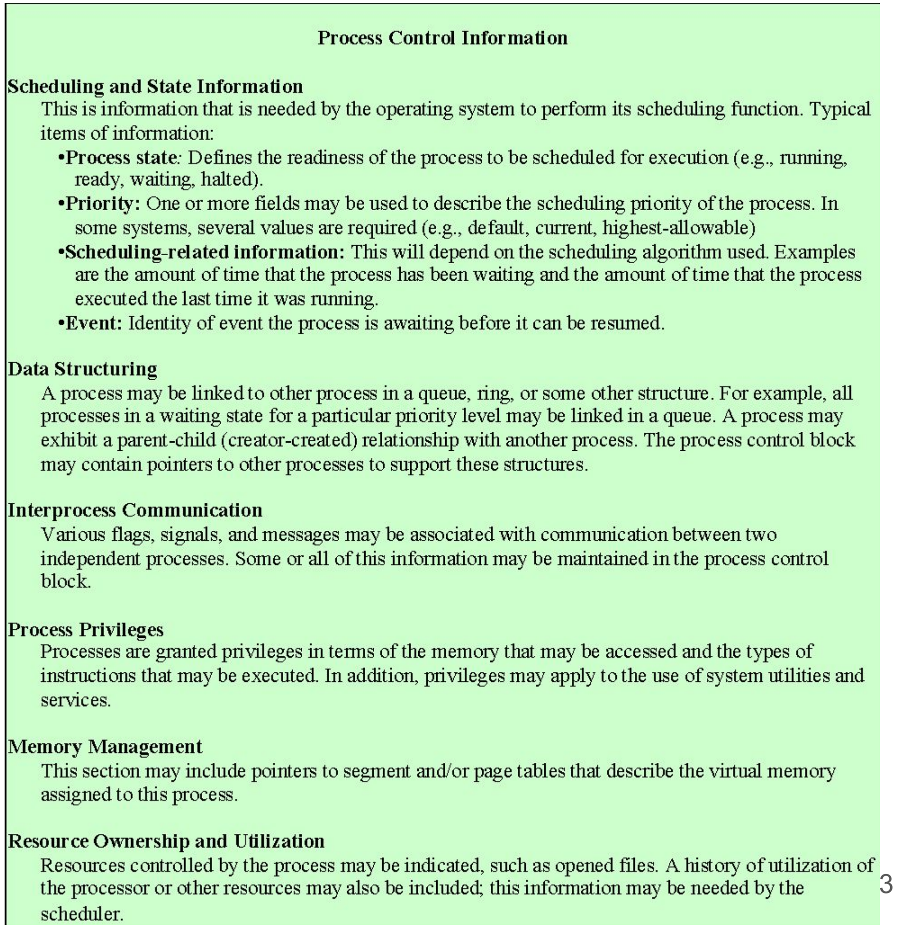


User Data
The modifiable part of the user space. May include program data, a user stack area, and programs that may be modified.

User Program
The program to be executed.

Stack
Each process has one or more last-in-first-out (LIFO) stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls.

Process Control Block
Data needed by the OS to control the process (see Table 3.5).



Process Control Information

Scheduling and State Information
This is information that is needed by the operating system to perform its scheduling function. Typical items of information:

- •**Process state**: Defines the readiness of the process to be scheduled for execution (e.g., running, ready, waiting, halted).
- •**Priority:** One or more fields may be used to describe the scheduling priority of the process. In some systems, several values are required (e.g., default, current, highest-allowable)
- •**Scheduling-related information:** This will depend on the scheduling algorithm used. Examples are the amount of time that the process has been waiting and the amount of time that the process executed the last time it was running.
- •**Event:** Identity of event the process is awaiting before it can be resumed.

Data Structuring
A process may be linked to other process in a queue, ring, or some other structure. For example, all processes in a waiting state for a particular priority level may be linked in a queue. A process may exhibit a parent-child (creator-created) relationship with another process. The process control block may contain pointers to other processes to support these structures.

Interprocess Communication
Various flags, signals, and messages may be associated with communication between two independent processes. Some or all of this information may be maintained in the process control block.

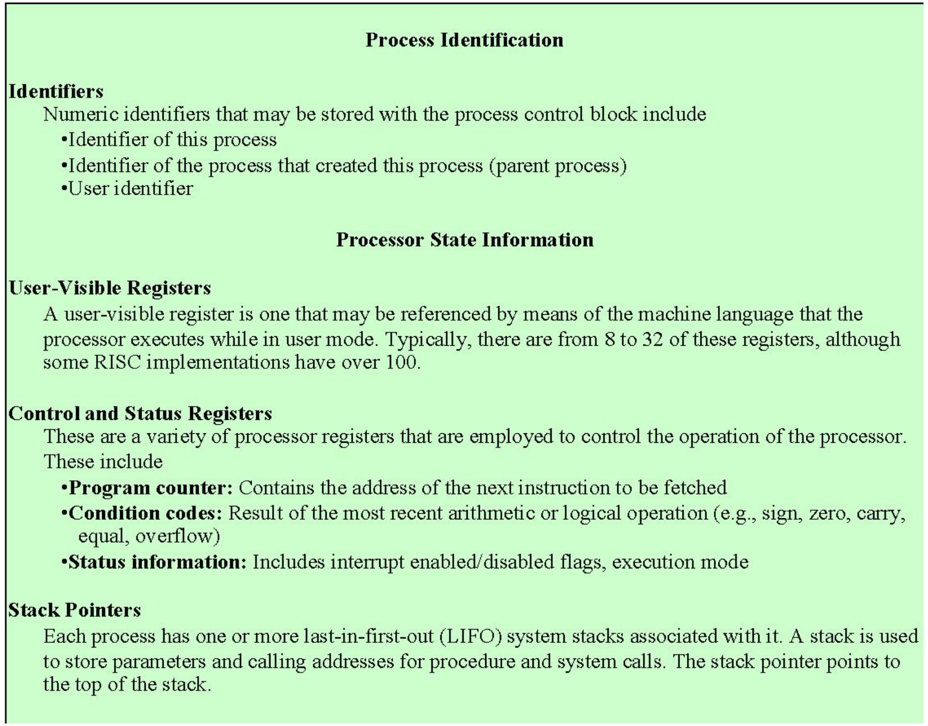
Process Privileges
Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed. In addition, privileges may apply to the use of system utilities and services.

Memory Management
This section may include pointers to segment and/or page tables that describe the virtual memory assigned to this process.

Resource Ownership and Utilization
Resources controlled by the process may be indicated, such as opened files. A history of utilization of the processor or other resources may also be included; this information may be needed by the scheduler.

Process Identification

- Each process is assigned a unique numeric identifier
  - Otherwise there must be a mapping that allows the OS to locate the appropriate tables based on the process identifier
- Tables controlled by the OS may use process identifiers to cross-reference process tables
- Memory tables may be organized to provide a map of main memory with an indication of which process is assigned to each region
  - Similar references will appear in I/O and file tables
  - When processes communicate with one another, the process identifier informs the OS of the destination of a particular communication
- When processes are allowed to create other processes, identifiers indicate the parent and descendents of each process



Process Identification

Identifiers
Numeric identifiers that may be stored with the process control block include
•Identifier of this process
•Identifier of the process that created this process (parent process)
•User identifier

Processor State Information

User-Visible Registers
A user-visible register is one that may be referenced by means of the machine language that the processor executes while in user mode. Typically, there are from 8 to 32 of these registers, although some RISC implementations have over 100.

Control and Status Registers
These are a variety of processor registers that are employed to control the operation of the processor. These include
•**Program counter:** Contains the address of the next instruction to be fetched
•**Condition codes:** Result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal, overflow)
•**Status information:** Includes interrupt enabled/disabled flags, execution mode

Stack Pointers
Each process has one or more last-in-first-out (LIFO) system stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls. The stack pointer points to the top of the stack.