Queensland University of Technology

CAB403

Systems Programming

Timothy Chappell

Dinal Atapattu

March 22, 2024

Contents

Chapter 1

Introduction to Operating Systems

1.1 Operating System Structures

1.1.1 Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users
- Operating System services provides functions that are helpful to the user
  - User Interface - Almost all opeating systems have a user interface (UI)
    - Graphical (GUI)
    - Command Line (CLI)
    - Batch
  - Program Execution - The system must be able to load a program into memory and run that program, end execution, either normally or abnormally (indicating error)
  - I/O operations - A running program may require I/O, which involves either a file or I/O device

Chapter 2

Operating System Structures

An operating system provides the environment within which programs are executed. Internally, operating systems vary greatly in their makeup as they are organised to varying specifications.
Operating systems are viewed from several points. Such as the services provided (users), the interface provided (programmers), and the components and their interconnections (operating system designers).

2.1 Operating System Services

2.2 User and Operating System Interface

2.2.1 Command Interpreters

2.2.2 Touch Screen Interface

2.3 System Calls

2.3.1 Example

2.3.2 API

2.3.3 Types of syscalls

Chapter 3

Processes

Batch systems define a job as a program and its data, and a time-shared system defines a task or user program as the unit of work.
Processes are programs in execution. The status of the current activity of a process is represented by the value of the program counter and the processor's registers. Programs are passive entities that are stored entirely in disk (executable fil), programs become processses when the executable is loaded into memory. The memory layout of a process is divided into multiple sections, including

- **Text Section** - The program code
- **Data Section** - Global variables
- **Heap** - Dynamically allocated memory during runtime
- **Stack** - Temporary data such as function parameters, return addresses, and local variables

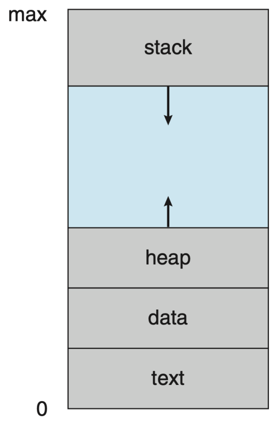


Figure 3.1: Process Layout in Memory

The text and data sections are fixed size, while the heap and stack can change dynamically during runtime. Each time a function is called, an activation record containing function parameters, local variables, and return address is pushed to the stack; when control is returned, the activation record is popped from the stack. While the heap and stack grow *toward* each other, they are not allowed to collide (*overlap*).

3.1 Process State

As a process executes, it changes state. The state of a process is defined partly by the current activity of the process. A process may be in one of the following states

- **New** - The process is being created
- **Running** - Instructions are being executed
- **Waiting** - The process is waiting for some event to occur (I/O completion, signal)
- **Ready** - The process is waiting to be assigned to a processor
- **Terminated** - The process has finished execution

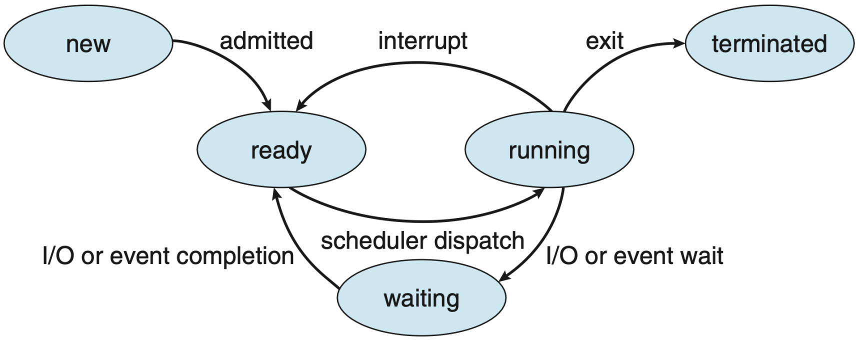


Figure 3.2: Process State Diagram

Only one process can be *running* on any processor at any instant. Many processes may be *ready* and *waiting*.

3.2 Process Control Block

Each process is represented in the operting system by a process control block (PCB) - also called a process control block, also called a task control block.
It contains many pieces of information associated with a specific process, including;

- **Process State** - New, Ready, Running, Waiting, Halted, Terminated, etc.
- **Program Counter** - Address of the next instruction to be executed for the process
- **CPU Registers** - Contents of all process-centric registers, these reisters vary from machine to machine and architecture to architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition code information. This state information must be saved along with the program counter when an interrupt occurs in order to allow the process to continue correctly after it is rescheduled to run
- **CPU Scheduling Information** - Process priority, pointers to scheduling queues, any other scheduling parameters
- **Memory Management Information** - Base and limit registers, page tables or segment tables, depending on the memory system used by the operating system. Depends on the memory system used by the operating system
- **Accounting Information** - Amount of CPU and real time used, time limits, account numbers, job or process numbers, etc.
- **I/O Status Information** - I/O Devices allocated to the process, list of open files, etc.

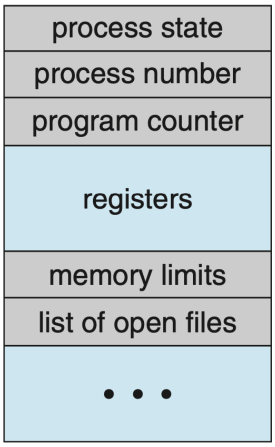


Figure 3.3: Process Control Block (PCB)

3.3 Threads

In the process model, a process is a program that performs a single thread of exeuction. A thread is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, and a thread ID. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.
Modern operating systems have extended the process concept to allow a process on multiple threads of execution, allowing to perform more than one task at a time which is beneficial for multicore systems, where multiple threads can run in parallel. A multithreaded word process for example could, manage user input on one thread while another runs the spell checker.
The PCB is expanded in these systems to include information for each thread, such as the thread ID, register set, and stack pointer, but the remaining information is shared among the threads within the same process.

3.4 Process Scheduling

The objective of multiprogramming is to have a process running at all times in order to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
These objectives are met using a process scheduler, which selects an available process for program execution on a core. Each CPU core can run one process at a time.

Chapter 4

Threads

A thread is a fundamental unit of CPU utilization, which forms the basis of multithreaded computer systems. Many modern applications are multithreaded, with threads running within an application. An application can divide tasks to seperate threads such as

- Updating the display
- Fetching data
- Spell checking
- Answering network requests

Thread creation, as opposed to process creation is lightweight, can simplify code, and increase efficiency. For this reason, kernels are often multithreaded.

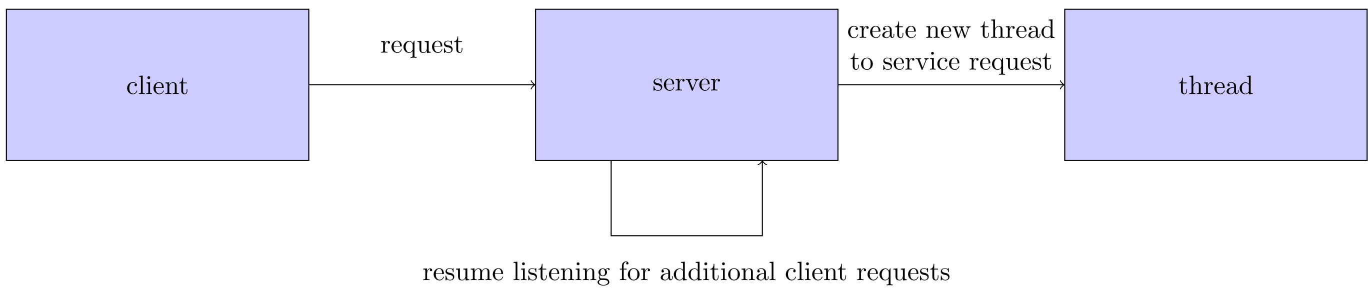


Figure 4.1: Multithreaded Server Architecture

Multithreading has the following benefits

- **Responsiveness**

  May allow continued execution if part of a process is blocked, important with user interfaces.
- **Resource Sharing**

  Threads share resources of process, easier to manage than shared memory or message parsing.
- **Economy**

  Cheaper than process creation, thread switching has lower overhead than context switching.
- **Scalability**

  Processes can take advantage of a multiprocessor architecture

4.1 Multicore Programming

In the early history of computer design, in order to combat the need for increased computing performance, single-CPU systems evoled into mult-CPU systems. Later, this evolved into including multiple compute cores on a single processing chip, where each core appears as a seperate CPU to the operating systems. These systems are defined as **multicore**.

4.1.1 Programming Challenges

The trend towards multicore systems continually places pressure on system designers and programmers to make better use of multiple compute cores. Designers of operating systems must write scheduling algorithms that use multiple processing cores to allow parallel execution

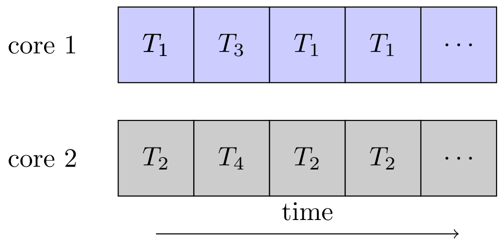


Figure 4.2: Parallel Execution on a Multicore System

In general, five areas present challenges in programming for multicore systems

- Identifying tasks

  This involves examining applications to find areas that can be divided into seperate, concurrent tasks. Ideally, tasks are independent of one another and thus can run in parallel on individual cores

- Balance

  While Identifying tasks that run in parallel, programmers msut also ensure that tasks perform equal work of equal value. In some instances, a certain task may not contribute as much value to the overall process as other tasks. Using seperate execution cores for that task might not be worth the cost

- Data Splitting

  Data accessed and manipulated by tasks must be divded to run on seperate cores, similar to how applications are divided to seperate tasks

- Data Dependency

  The data accessed by he tasks must be examined for dependencies between the two or more tasks. When data is dependent between cores, programmers must ensure that the execution of the tasks is synchronized to accomodate the dependency.

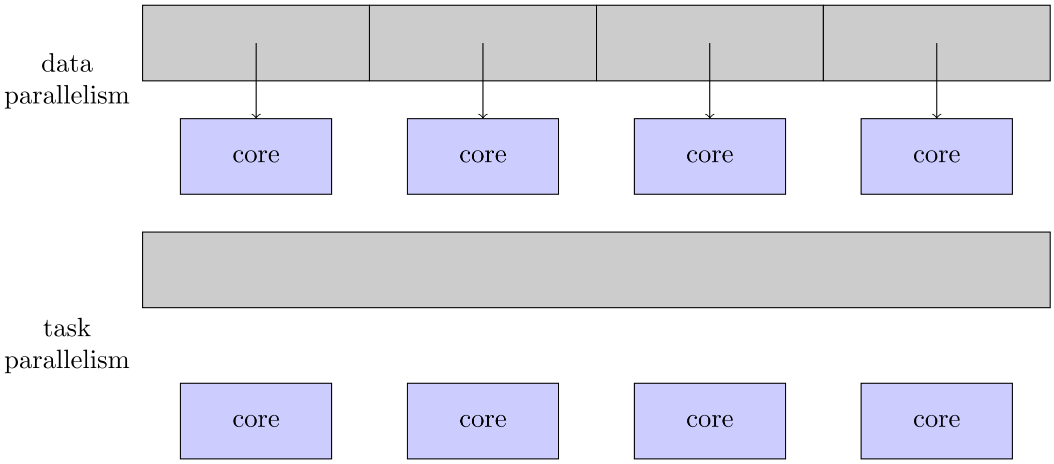


Figure 4.3: Data and Task Parallism

- Testing and Debugging

  When a program is running in parallel n multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single threaded applications

4.1.2 Parallelism

Types of parallelism

- Data Parallelism

  Focuses on distributing subsets of the same data across multiple compute cores, performing the same operation on each core.

  Example: summing the contents of an array size N, on a dual core system, thread A sums the elements [0] ... $[N/2-1]$, thread B sums the elements $[N/2]$ ... $[N-1]$. These threads will run in parallel on seperate cores.

- Task Parallelism

  Distributes tasks across multiple cores. Each thread peforms a unique operation. Different threads may operate on the same or different data.

  Example: Dual core system, applying two different arithmetic and/or other operation on the same block of data on seperate threads. These threads will will run parallel on seperate cores.

Data and Task parallelism may be done together, in a hybrid solution.

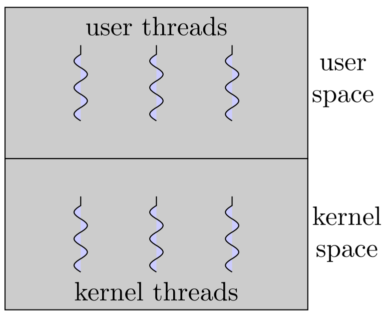


Figure 4.4: Use and Kernel Data Spaces

4.2 Multithreading Models

4.2.1 Many-To-One Model

Many user-level threads are mapped to a single kernel thread.
One thread block causes all to block.
Multiple threads cannot run in parallel on a multicore system because the kernel can only handle a single thread.
Rarely used.
Examples include

- Solaris Green threads
- GNU Portable threads

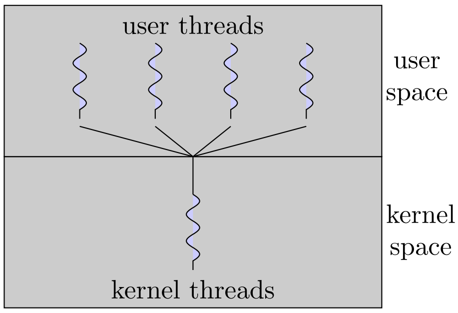


Figure 4.5: Many-to-one model

4.2.2 One-To-One Model

Each user-level thread maps to a kernel thread.
Creating a user level thread creates a kernel thread.
More concurrency than many-to-one.
Number of threads per process sometimes restricted due to overhead.
Examples include.

- Windows
- Linux
- Solaris (9 and later)

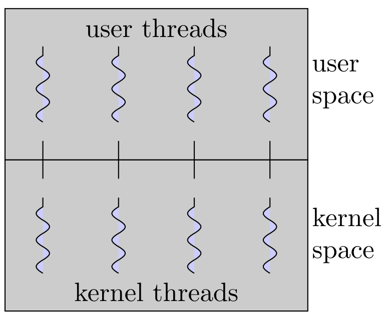


Figure 4.6: One-to-one model

4.2.3 Many-to-many model

Allows many user level threads to be mapped to many kernel threads.
Allows the operating system to create a sufficient number of kernel threads.
Examples include

- Solaris (pre version 9)
- Windows (*ThreadFiber* package)

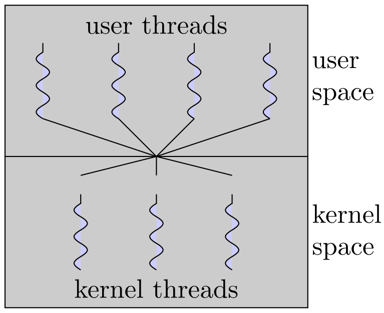


Figure 4.7: Many-to-many model

4.2.4 Two-level model

Similar to Many-to-many, except allows a user thread to be **bound** to a kernel thread Examples include

- IRIX
- HP-UX
- Tru64 UNIX
- Solaris (8 and earlier)

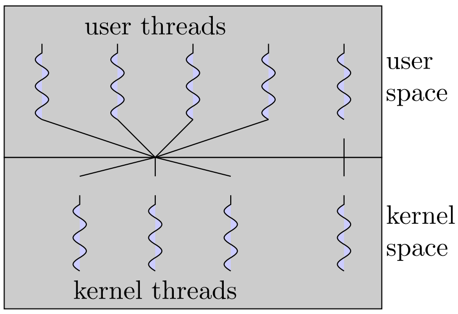


Figure 4.8: Two-level model

4.3 Thread Libraries

4.3.1 Pthreads

Refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronisation. This is a specification not an implementation

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

long double sum;             /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char **argv)
{
    pthread_t thread_id;        /* thread identifier */
    pthread_attr_t thread_attr; /* thread attributes */

    if (argc != 2)
    {
        fprintf(stderr, "usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0)
    {
        fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
        return -1;
    }
    /* get default attributes */
    pthread_attr_init(&thread_attr);
    /* create the thread */
    pthread_create(&thread_id, &thread_attr, runner, argv[1]);
    /* wait for the thread to exit */
    pthread_join(thread_id, NULL);
    printf("sum = %Lf\n", sum);
}

/* thread executes this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;
    for (i = 1; i <= upper; i++)
    {
        sum += i;
    }
    pthread_exit(0);
}
```

Figure 4.9: Multithreading using the pthreads API

Although Windows does not natively support pthreads, some third-party implementations are available

```c
#define NUM_THREADS 10
/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
{
    pthread_join(workers[i], NULL);
}
```

Figure 4.10: Joining 10 threads using the pthreads API

4.3.2 Windows Threads

Similar to the technique used in pthreads. Uses a different library

```c
#include <windows.h>
#include <stdio.h>

DWORD Sum; /* data is shared by the threads */
/* The thread will execute this function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 1; i <= Upper; i++)
    {
        Sum += i;
    }
    return 0;
}

int main(int argc, char **argv)
{
    DWORD ThreadID;
    HANDLE ThreadHandle;
    int Param;

    Param = atoi(argv[1]);
    /* create the thread */
    ThreadHandle = CreateThread(
        NULL, /* default security attributes */
        0, /* default stack size */
        Summation, /* thread function */
        &Param, /* parameter to thread function */
        0, /* default creation flags */
        &ThreadID /* returns thread identifier */
    );

    /* close the thread handle */
    CloseHandle(ThreadHandle);

    printf("sum = %d\n", Sum);
}
```

Figure 4.11: Multithreading using the Windows API

4.4 Implicit Threading

The growing popularity of multicore processing means that applications now require hundreds, or thousands of threads. When designing such programs, correctness grows more and more difficult. Creating and management of threads done by compilers and run-time libraries are favoured in this case.

4.4.1 Thread Pools

Creates a number of threads in a pool where they await work Advantages

- Usually faster to service a request with an existing thread than create a new thread
- Alows the number of threads in an application(s) to be bound to the size of the pool

- Seperating tasks to be performed from mechanics of creating task allows different stratgeies for running task

  Tasks could be scheduled to be run periodically

```
DWORD WINAPI PoolFunction(PVOID Param)
{
    /*
     * this function runs as a seperate thread
     */
}
```

Figure 4.12: Thread pooling in the Windows API

4.4.2 OpenMP

Set of compiler directives and an API for C, C++, FORTRAN.
Provides support for parallel programming and shared-memory environments.
Identifies parallel regions - blocks of code that can run in parallel.

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv)
{
    int thread_id;

    #pragma omp parallel
    {
        printf("Hello from process %d\n", omp_get_thread_num());
    }
    return 0;
}
```

Figure 4.13: OpenMP

4.4.3 Grand Central Dispatch

- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++, API and run-time library
- Allows identification of parallel sections
- Manages most details of threading
- Blocks is in "^{}" - `^{ printf("I am a block"); }`
- Blocks are placed in dispatch queue and then assigned to avaiable threads in thread pool when removed
- Two types of dispatch queues
  - **Serial** - blocks are removed FIFO, queue is per process, called **Main Queue**

    Programmers create additional serial queues within program

- **Concurrent** - removed in FIFO, but many be removed at a time

  Three system wide queues with priorities low, default, high

```
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_async(queue, ^{ printf("I am a block."); });
```

4.5 Threading Issues

- Semantics of **fork()** and **exec()** system calls.
- Signal handling (synchronous and asynchronous)
- Thread cancellation of target thread (asynchronous or deferred)
- Thread local storage
- Scheduler activations

4.5.1 Semantics of fork() and exec()

- Does fork() duplicate only the calling thread or all threads?

  Some UNIXes have two versions of fork
- exec() usually works as normal (replaces running process including all threads)

4.5.2 Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A signal handler is used to process signals
  - Signal is generated by an event
  - Signal is delivered to a process
  - Signal is handled by one of two signal handlers

    default

    user-defined
- Every signal has a default handler that kernel runs when handling a signal
  - User-defined signal handler can override default
  - For single-threaded, signal delivered to process
- Where is the signal delivered in multi-threaded?
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in process

4.5.3 Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is target thread
- Two general approaches
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

```
pthread_t thread_id;
/* create the thread */
pthread_create(&thread_id, 0, worker, NULL);
...
/* cancel the thread */
pthread_cancel(thread_id);
```

Figure 4.14: pthread code to create and cancel a thread

- Invoking thread cancellation requests cancellation, but actual cancellation depends on the thread state

Mode	State	Type
Off	Disabled	-
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

Figure 4.15: Thread States

- If a thread has cancellation disabled, cancellation remains pending till enabled
- Default type is deferred

  Cancellation only occurs when thread reaches cancellation point
  - i.e., `pthread_testcancel();`
  - Then cleanup handler is invoked
- On Linux, thread cancellation is handled through signals

4.5.4 Thread-Local Storage

- Thread-local-storage (TLS) allows each thread to have its own copy of data
- Useful when you don't have control over thread creation (i.e., using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to static data
  - TLS is unique to each thread

4.5.5 Scheduler Activations

- Both Many to Many and Two-Level require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads (lightweight process (LWP))
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP is attached to a kernel thread
  - How many LWPs to create?
- Scheduler activations provide upcalls - a communication mechanism from the kernel to the upcall handler in the thread library
- This communication allows an application to maintain the correct number of kernel threads

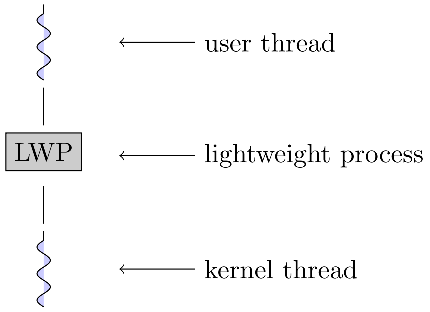


Figure 4.16: Lightweight Process (LWP)

Chapter 5

Synchronisation

Processes can either execute concurrently or in parallel. These processes may be interrupted at any time, partially completing execution. Concurrent access to any shared memory may result in data inconsistency if not properly controlled.
Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Example:

> Suppose we wanted to provide a solution to the consumer-product problem that fills **all** the buffers. We can do so by having an integer `counter` that keeps track of the number of full buffers. Initially, `counter` is set to 0. It is incremented by the producer after producing a new buffer, and decremented by the consumer after it consumes a buffer.

```
while (true) {
    /* produce an item in next produced */
    while (counter == BUFFER_SIZE);
        /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

```
while (true) {
    while (counter == 0);
        /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /* consume the item in next_consumed */
}
```

Figure 5.1: Producer and Consumer

5.1 Race Conditions

While the producer and consumer routines shown are correct seperately, they may not function correctly when executed concurrently. For example: Suppose that the value of the variable `count` is currently 5 and the producer and consumer processes concurrently execute the statements "`count++`" and "`count--`". This will lead to the variable being 4, 5 or 6. With the only correct value being `count == 5`, which is executed if the producer and consumer execute seperately. We can show the value maybe incorrect by using the following implementation

$$\begin{aligned} register_1 &= count \\ register_1 &= register_1 + 1 \\ count &= register_1 \end{aligned}$$

where $register_1$ is one of the local CPU registers.
Similarly, the statement `count - 1` is implemented as follows

$$\begin{aligned} register_2 &= count \\ register_2 &= register_2 - 1; \\ count &= register_2 \end{aligned}$$

where again, $register_2$ is a local CPU register.
Even though $register_1$ and $register_2$ are local CPU registers, the value of this register will be saved and stored by the interrupt handler
The concurrent execution `count++` and `count -- 1` is equivalent to the sequential execution where lower evel statements previously are intef rleaved in an arbitrary order, with the order of the high-level statement is preserved.

$$T_0 \quad producer \quad \text{execute} \quad register_1 = count \quad \{register_1 = 5\}$$

5.2 The Critical-Section Problem

In a system conssiting of n processes $P_0, P_1, ..., P_{n-1}$. Each process has a segment of code, called a critical section in which the process is accessing and updating data that is shared with at least one process. The important feature of the system is that when one process is executing in its critical section, no other process is allowed to execute in its critical section at the same time.
The *The critical section problem* is to design a protocol that processes can use to synchronize their activity to enter its critical section.
It consists of the following sections

- Entry section Implements requests for permission to enter its critical section. Decides the entry of a particular process
- Critical Section Part which only one process can enter and modify the shared variable. This part of the process ensures that no other processes can access the shared resource
- Exit Section Allows for other processes waiting in the entry section to enter the critical section. It checks that a process if completed the critical section can move to the remainder section
- Remainder Section The other parts of the code, other than the entry, critical, and exit sections

5.2.1 Solution to the Critical Section Problem

A solution must satisfy the following 3 requirements

- Mutual Exclusion

  If process P_i is executing in its critical section, then no other process can be executing in their critical section

- Progress

  If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

- Bounded Waiting

  There exists a bound, or limit, on the number of time that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

We assume that each process is executing at a nonzero speed. However, we can make no assumption about the relative speed of the processes.
Many kernel-mode processes may be active in the operating system at the same time. As a result the kernel code (code implementing the operating system) may be subject to several possible race conditions.

Chapter 6

Safety Critical Systems

Safety is the freedom from conditions that cause death, injury, illness, damage to or loss of equipment or property, or environmental harm
Software is inherently not safe or unsafe, however can contribute to unsafe conditions in a safety critical system. Such software is **Safety Critical**
IEEE definition: "Software whose use in a system can result in unacceptable risk. Safety-critical software incliudes softwre whose operation or failure to operate can lead to a hazardous state. Software intended to reover from hazardous states, and software intended to mitigate the severity of an incident"

6.1 MISRA C

- Motor Industry Software Reliability Association

6.2 NASA Power of 10

- Avoid complex flow constructs (goto, recursion, jumps)
- All loops must have fixed bounds (prevents runaway code)
- Avoid heap memory allocation (no malloc, define everything in main)
- Restrict functions to a single page (max 50 lines)
- Use a minimum of two runtime assertions per function
- Restrict the scope of data to the smallest possible
- Check the return value of all non-void functions, or cast to void to indicate the return is useless
- Use the preprocessor sparingly. (DO NOT USE stdio.h, local.h, abort/exit/system from stdlib.h, time handling from time.h)
- Limit pointer use to a single dereference and DO NOT USE FUNCTION POINTERS
- Compile with all warnings active (Wall, Wextra, etc.) all warnings should then be addressed before release

Chapter 7

Distributed Systems

A distributed system is a collection of processors that do not share memory or a clock. Instead, each node has its own local memory. The nodes communicate over various networks, such as high-speed busses. Applications of distributed systems vary widely, from providing transparent file access inside an organization, to large-scale cloud storage services, to business analysis of trends on large datasets, to parallel processing of scientific data. With the most ubiqiutous form of a distributed system being the Internet.

7.1 Basic Concepts

A distributed system is a collection of loosely coupled nodes interconnected by a communication network. From the point of view of a specific node in a distributed system, the rest of the nodes and their respective resources are remote, whereas its own resources are local.
Processors are variously called **nodes, computers, machines, hosts**

- **Site** is the location of the processor
- Generally a **server** has a resource a **client** node at a different site wants to use

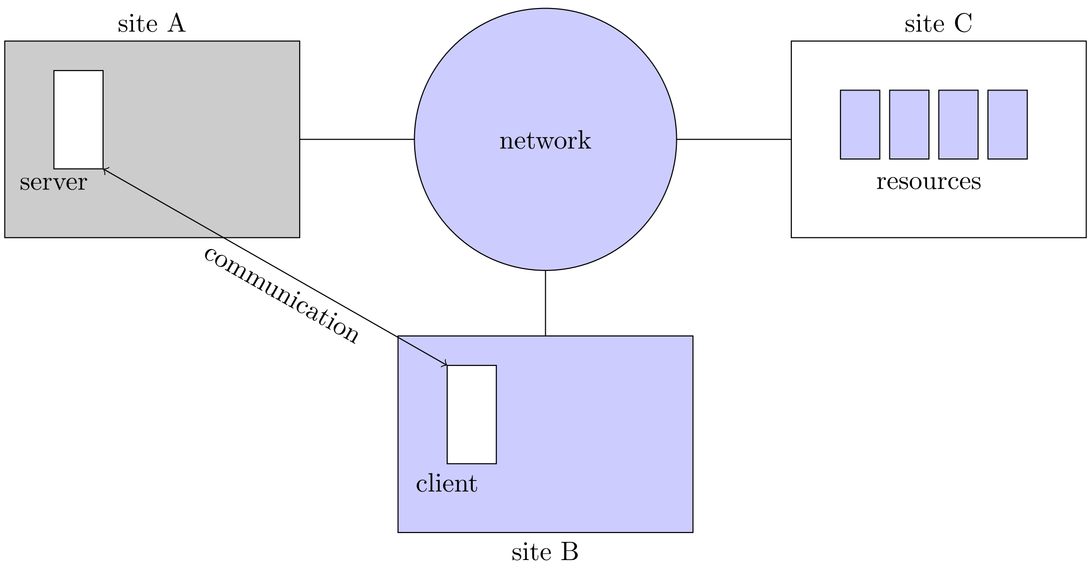


Figure 7.1: A client-server distributed system

7.2 Advantages of Distributed Systems

- **Resource Sharing**
  - Sharing and printing files at remote sites
  - Processing information in a distributed database
  - Using remote specialized hardware devices
- **Computation speedup**
  - **Load sharing**
  - **Job migration**
- **Reliability**

  Detect and recover from site failure, function transfer, reintegrate failed site.
- **Communication**

  Message passing
  All higher-level functions of a standalone system can be expanded to encompass a distributed system

Computers can be downsized, more flexibility, better user interfaces and easier maintenance by moving from a large system to a cluster of smaller systems performing distributed computing

7.3 Network-Operating Systems

7.4 Distributed Operating Systems

Chapter 8

CPU Scheduling

CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive.

8.1 Basic Concepts

A single core system can only run a single process at a time. Others must wait till the CPU's core is free and can be rescheduled.
Multiprogramming is the idea of having a process running at all times to maximise CPU utilization.
A process is executed until it must wait, typically for the completion of an I/O request. A simple computer system just idles during this period, waiting compute time, no work is accomplished. With multiprogramming, we try to use this time productively. By keeping several processes in memory, when one process has to wait, the operating takes the CPU away from the process and gives it to another process. On a multicore system this is extended to all processing cores on the system.
Such scheduling is fundamental to an operating system's functionality. Almost all computer resources are scheduled before use. The CPU, being one of the primary resources of a computer needs special attention during its scheduling

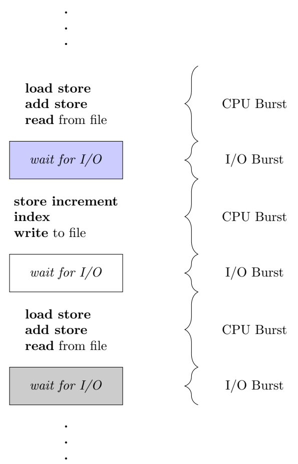


Figure 8.1: Alternating Sequence of CPU and I/O Bursts

8.2 CPU Scheduler

Whenever the CPU idles, the operating system must select a process in the ready queue to be executed. This is selected by the CPU Scheduler. Which selects a process from the processes in memory that are ready to execute, and allocates the CPU to it.

The short-term scheduler seelcts from processes in the ready queue and allocates the CPU to one of them. CPU scheduling happens when

- Switching from running to waiting
- Switching from running to ready
- Switching from waiting to ready
- Terminating

8.2.1 Preemptive and Non-Preemptive Scheduling

CPU-scheduling decisions may take place under the following four circumstances

1. Process switching from running state to the waiting state (I/O request, wait invocation for Terminating a child process)
2. Process switching from running to ready state (interrupt)
3. Process switching from waiting to ready (I/O completion, release of a

8.2.2 API

8.2.3 Types of syscalls

semaphore)

4. Process terminating

For circumstances 1 and 4, there is no choice in scheduling, a new process (if exists) must be selected for execution. For circumstances 2 and 3, a choice exists.
Circumstance 1, and 4 are called non preemptive or cooperative. Otherwise it is preemptive.
Non-preemptive scheduling keeps the CPU once the process is allocated till termination or switching to the wait state.

8.2.3.1 Preemptive Scheduling

Virtually all modern operating systems use preemptive scheduling.
However, Preemptive scheduling can result in race conditions when data is shared between processes (first process is updating data while the second process tries to read it, which is in an inconsistent state).
Kernel design must be modified to accomodate a preemptive scheduler as race conditions in updating or modifying important kernel data (IO queues, process table, etc.) while they are being read can cause the system to crash.
Therefore, a preemptive kernel requires mechanisms such as mutex locks to prevent race conditions when accessing shared kernel data structues.
Most modern operating systems are full prememptive when running in kernel mode.

8.2.3.2 Non Preemptive Scheduling

Non preemptive scheduling is simpler than preemptive scheduling as there is no need for any special hardware or kernel mode privileges.
However, non preemptive scheduling is not suitable for modern operating systems as it does not allow the kernel to respond to interrupts while running in kernel mode.
Non preemptive scheduling is used in embedded systems where the kernel is the only process running on the system.
Non preemptive scheduling is also used in real time systems where the kernel is the only process running

on the system.

8.3 Dispatcher

Is the module that gives control of the CPU's core to the process selected by the scheduler. Involves

- Context switching
- Switching to user model
- Jumping to the proper location in the user program to resume that program

Speed is crucial with the dispatcher as it is invoked during every context switch.
Dispatch latency is the time taken for the dispatcher to stop one process and starty another running.

8.4 Scheduling

Optimisation objectives

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

8.4.1 First-Come, First-Served (FCFS) Scheduling

The process that requests the CPU first is allocated the CPU first.
Managed using a FIFO queue.
Given the following process table

Process	Burst Time	Arrival Time
P1	24	0
P2	3	1
P3	3	2

(a) Process Table

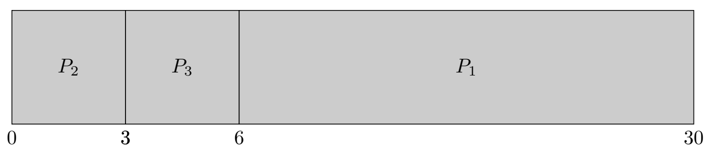
(b) FIFO Gantt Chart

Waiting time for P_1 is 0, P_2 is 24, P_3 is 27. Average waiting time is 17.

Figure 8.2: Process Table and FIFO Gantt Chart with waiting times

Process	Burst Time	Arrival Time
P1	24	3
P2	3	1
P3	3	2

(a) Process Table



(b) FIFO Gantt Chart

Waiting time for P_1 is 6, P_2 is 0, P_3 is 3. Average waiting time is 3.

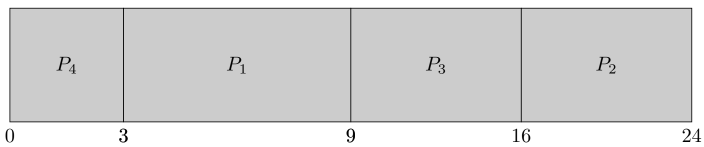
Figure 8.3: Process Table and FIFO Gantt Chart with waiting times

8.4.2 Shortest-Job-First (SJF) Scheduling

By associating each process with the length of its next CPU burst, we can schedule the process with the shortest time.
SJF is optimal, giving the minimum average waiting time for a given set of processes.
However, knowing the length of the next CPU burst is difficult to predict the length of the next CPU burst.

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

(a) Process Table



(b) SJF Gantt Chart

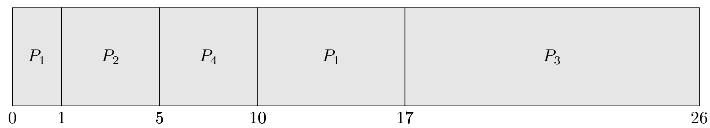
Average waiting time is 7

Figure 8.4: Process Table and SJF Gantt Chart with waiting times

SJF can be either preemptive or non preemptive.
Preemptive SJF is also called Shortest Remaining Time First

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

(a) Process Table with Arrival Time



(b) SRTF Gantt Chart

Average waiting time is 6.5

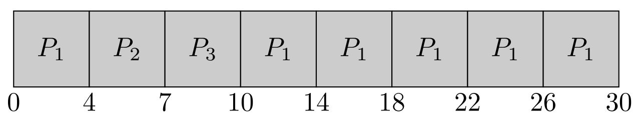
Figure 8.5: Process Table and SRTF Gantt Chart with waiting times

8.4.3 Round Robin Scheduling

Each process is given a small unit of CPU time (time quantum or time slice q) that is 10-100 millisecond.
After that time, the process is preempted and added to the end of the ready queue.
If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units at once. No process waits more than (n-1)q time units.
Timer interrupts every quantum to schedule the next process.

Process	Burst Time
P1	24
P2	3
P3	3

(a) Process Table



(b) Round Robin Gantt Chart

Figure 8.6: Round Robin Gantt Chart with time quantum 4

Typically, round robin has a higher average turnaround than SJF but has better reponse.

Chapter 9

Deadlocks

9.1 System Model

A system consists of finite resources distributed among a number of competing threads. These resources must be partitioned into several types or classes, each consisting of identical instances.
Under normal operation, a thread may utilize a resource in the following sequence

1. **Request** - The thread requests the resource. If the request cannot be granted immediately (for example, if a mutex lock is held by another thread) the requesting thread must wait till it can acquire the resource.
2. **Use** - The thread can operate on the resource (for example, if the resource is a mutex lock, it can access its critical section)
3. **Release** - The thread releases the resource (for example, if the resource is a mutex lock, it releases the lock, allowing another thread to acquire it)

9.2 Deadlock Characterization

Deadlocks can arise if four conditions hold simultaneously

- **Mutual exclusion** - At least one resource must be held in a nonsharable mode; that is, only one thread ata time can use the resource. If another thread requests that resource, the requesting thread must be delayed until the resource is released.
- **Hold and Wait** - A thread must be holding at least one resource and waiting to acquire additional resources that are currently being hld by other threads
- **No Preemption** - Resources cannot be preempted; resources can only be released voluntarily by the thread holding it, after that the thread has completed its task.
- **Circular Wait** - A set of waiting threads must exist such that T_0 is waiting for a resource held by T_1, T_1 is waiting for a resource held by T_2, ... T_{n-1} is waiting for a resource held by T_n, and T_n is waiting for a resource held by T_0.

Note that circular wait implies hold and wait

9.3 Resource-Allocation Graph

A set of vertices V and a set of edges E.

- V is partitioned into two types

  P - Set consisting of all the threads in the system

  R - Set consisting of all resource types in the system

- **Request edge**

  Directed edge $T_i \rightarrow R_j$

- **Assignment edge**

  Directed edge $R_j \rightarrow T_i$

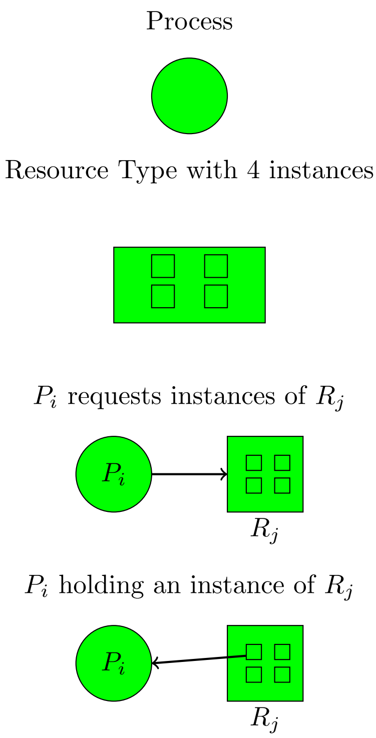
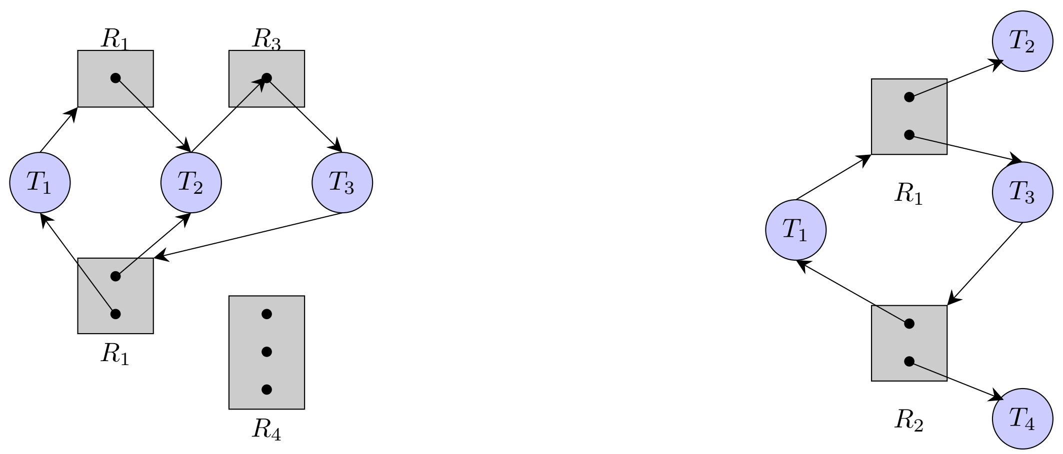


Figure 9.1: Resource Allocation Graph



(a) Resource Allocation Graph with Deadlock (b) Resource Allocation Graph without Deadlock

Figure 9.2: Resource Allocation Graphs

9.3.1 Basic Facts

- If a graph has no cycles there is no deadlock

- If a graph has a cycle

  If only one instance per resource type, then there's a deadlock

  If several instances per resource type, then deadlock may or may not exist

9.4 Methods for Handling Deadlocks

Ensure the system will never enter a deadlock state, or make sure that once a deadlock occurs, it will be recovered from the state. OR ignore the problem and pretend that deadlocks never occur in the system (UNIX, Linux, Windows).

9.5 Deadlock Prevention

Restrain the way requests can be made

- **Mutual Exclusion** - Not required for sharable resources, must hold for nonsharable resources
- **Hold and Wait** - Must guarantee that whenever a thread requests a resource, it does not hold any other resources
  - Require that each thread request all its resources at one time
  - Require that once a thread holds a resource, it cannot request additional resources
  - Require that once a thread requests a resource, it cannot request additional resources till the requested resource has been allocated to it
- **No preemption**
  - If a process that is holding some resources requests another resource that cannot immediately allocated to it, then all resources held must be released.
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain old resources, as as the ones requested
- **Circular Wait**
  - Impose a total ordering of all resource types, and require that each thread requests resources in an increasing order of enumeration

9.6 Deadlock Avoidance

Requires that the system has some additional **a priori** information available.

- Simplest and most useful model requires that each process declare the **maximum number** of resources of each type it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes/threads.

9.6.1 Basic Facts

- If a system is in safe state, no deadlocks
- If a system is in unsafe state, possibility of deadlock
- Avoidance, ensure that a system will never enter an unsafe state

9.7 Safe State

When a process requests and available resource, the system must decide if immediate allocation leaves the system in a safe state.
The system is in a **safe state** if there exists a sequence of **ALL** the processes in the system such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$. i.e.,

- If P_i requests a resource, it may have to wait till all the P_j with $j < i$ have finished
- When P_j is finished, P_i can obtain its needed resources, execute, return allocated resources, and terminate
- When P_i terminates, P_{i+1} can obtain its needed resources, and so on

9.8 Deadlock Avoidance Algorithms

9.8.1 Resource Allocation Graph Scheme

- Claim edge $P_i \rightarrow R_j$ indicated that processes P_i may request resource R_j.
- Claim edge converts to request edge when a process/thread requests a resource.
- Request edge converted to an assignment edge when the resource is allocated to the process/thread.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.

(a) Safe State (b) Unsafe State

Figure 9.3: Resource Allocation Graph States

The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph.

9.9 Banker's Algorithm

Used for multiple instances of resources, where each process must a priori claim the maximum number of resources. When a process requests a resource it may have to wait. When a process gets all of its resources, it must return them in a finite amount of time.

9.9.1 Data Structures

With n processes and m resource types, the following data structures are used

- **Available** - Vector of length m. If *available[j]* = k, there are k instances of resource type R_j available.
- **Max** - $n \times m$ matrix. If *Max[i,j]* = k, then process P_i may request at most k instances of resource type R_j.
- **Allocation** - $n \times m$ matrix. If *Allocation[i,j]* = k, then process P_i is currently allocated k instances of R_j.
- **Need** - $n \times m$ matrix. If *Need[i,j]* = k, then process P_i may need k more instances of R_j to complete its task.

Need[i,j] = *Max[i,j]* - *Allocation[i,j]*

9.9.2 Safety Algorithm

1. Letting **Work** and **Finish** be vectors of length m and n respectively.
2. Initialising **Work** = **Available** and **Finish**[**i**] = false for $i = 0, 1, ..., n-1$
3. Find an **i** such that **Finish**[**i**] = **false** and **Need**≤**Work**

   If no such **i** exists, go to step 4
4. **Work** = **Work** + **Allocation**[**i**], **Finish**[**i**] = **true**

   Go to step 2
5. If **Finish**[**i**] = **true** for all **i**, the system is in a safe state.

9.9.3 Resource-Request Algorithm for Process P_i

Request[**i**] = request vector for process P_i. If **Request**[**i,j**] = **k**, then process P_i wants k instances of resource type R_j.

1. if **Request**[**i**] ≤ **Need**[**i**], go to step 2. Otherwise, raise error condition since process has exceeded its maxmimum claim.
2. if **Request**[**i**] ≤ **Available**, go to step 3. Otherwise, P_i must wait as resources are unavailable.
3. Pretend to allocate the requested resources to P_i by modifying the state as follows

   Available = **Available** - **Request**
   Allocation[**i**] = **Allocation**[**i**] - **Request**[**i**]
   Need[**i**] = **Need**[**i**] - **Request**[**i**]

   - If safe state, then resources will be allocated to P_i
   - If unsafe state, P_i must wait, and the old resource-allocation state will be stored

9.9.4 Example

Given 4 processes $[P_0, P_1, P_2, P_3, P_4]$ and 3 resource types [A (10 instances), B (5 instances), C (7 instances)].
At time t_0, the following snapshot of the system has been taken

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

The content of matrix **Need** is calculated as follows

	Allocation	Max	Need
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3
P_1	2 0 0	3 2 2	1 2 2
P_2	3 0 2	9 0 2	6 0 0
P_3	2 1 1	2 2 2	0 1 1
P_4	0 0 2	4 3 3	4 3 1

This system is in a safe state since the sequence $< P_1, P_3, P_4, P_0, P_2 >$ satisfies the safety algorithm.

Chapter 10

File Systems and Protection

10.1 Protection

10.1.1 Goals of Protection

In one protection model, the computer consists of a collection of objects, hardware and/or software, where each object has a unique name and can be accessed through a well-defined set of operations.
The protection problem is to ensure that each object is accessed correctly and only by the processes that are allowed to do so.

10.1.2 Principles of Protection

The guiding principle of protection is the principle of least privilege, where;

- Programs, users, and systems should be given just enough the privileges to perform their tasks
- Limits damage if process is compromised
- Can be static (during system or process lifetime) or dynamic (changed by process as needed) domain switching, privilege escalation
- "Need to know" is a similar concept regarding access to data.

The grain aspect

- Rough-grained privilege management - easier, simpler, but less privilege levels as they are done in chunks

  Processes either have all privileges of user or of root

- Fine-grained management - more complex, more overhead, but more stringent and protective

  File ACL Lists, RBAC

10.1.3 Domain of Protection

Protection rings seperate functions into domains and order them hierarchically. A generalization of rings is using domains without a hierarchy.
A computer system can be treated as a collection of processes and objects.
Objects in this context can be hardware or software entities.

10.1.3.1 Domain Structure

In order to facilitate this scheme, a process may operate within a protection domain which specifices the resources that the process may access.
Each domain defines a set of objects and the types of operations that may be invoked.
The ability to execute an operation on an object is an access right.
A domain is a collection of access rights, each an ordered pair

(object, set of operations)

For example, if domain D_1 has access rights **(file *F*, read, write)**, then process P_1 in domain D_1 may read and write F.
Domains may share access rights as well, for example, if domain D_2 has access rights **(file *F*, read)**, then process P_2 in domain D_2 may read F.
The association between a process and a domian may be either static or dynamic.
Establishing dynamic protection domains is more complex as it involves domain switching, where the process switches from one domain to another.

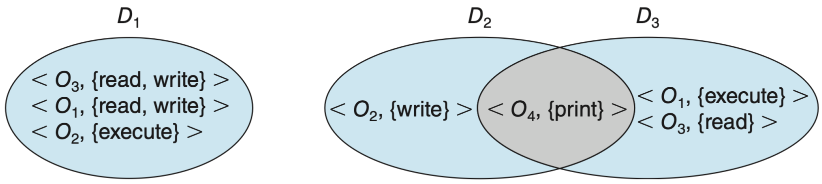


Figure 10.1: System with three protection domains

Domains can be realised in multiple ways

1. Each *user* may be a domain. In this case, the set of objects that can be accessed depends on the identity of the user. Domain switching occurs when the user is changed (login, su, etc.)
2. Each *process* may be a domain. In this case the set of objects that can be accessed depends on the identity of the process. Domain switching occurs when one process sends a message to another process and waits for a response.
3. Each *procedure* may be a domain. In this case the set of objects that can be accessed corresponds to the local variables defined within the procedure. Domain switching occurs when a procedure is called.

10.1.4 Access Matrix

The general model of protection can be viewed abstractly as a matrix, called an access matrix.
Rows represent domains, and columns represent objects.
Each entry contains a set of access rights where the entry `access`(i,j) defines the set of operations that a process in domain D_i can invoke on object O_j.

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Figure 10.2: Access Matrix

The creator of the object dictates the access column for the object.
Can be expanded to dynamic protection

- Operations to add, delete rights (columns)
- Special access rights
  - Owner of O_i
  - Copy op from O_i to O_j (denoted by *)
  - Control D_i can modify D_j access rights
  - Transfer switch domain from D_i to D_j

Note that copy and owner are applicable to an object while control and transfer are applicable to a domain.

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Figure 10.3: Access Matrix with Domains as Objects

This has the two variants;

- A right is copied from `access`(i,j) to `access`(k,j). It is then removed from from `access`(i,j). Therefore, is a transfer not a copy.
- Propogation of `copy` right may be limited. That is, when the right $R*$ is copied from `access`(i,j) to `access`(k,j), only the right R (not $R*$) is created. A process executing in domain D_k cannot further copy the right R.

domain \ object	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

domain \ object	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

Figure 10.4: Access Matrix with Copy Rights

10.1.5 Access Control

Protection can also be applied to non file resources.
Oracle Solaris 10 provides role-based access control (RBAC) to implement least privilege.

- **Privilege** is the right to execute a system call or use an option within a system call
- Can be assigned to processes
- Users assigned **roles** granting access to privileges and programs

  Enable role via password to gain privileges
- Similar to an access matrix

10.2 File System Interface

10.2.1 File Concept

A file is a named collection of related information that is recorded on secondary storage. A logical storage unit that is independent from the physical properties of its storage device. From a user's perspective, a file is the smallest allotment of logical secondary storage (data cannot be written to secondary storage unless they are within a file).
File types include

- Data
  - Numeric
  - Character
  - Binary
- Program

Some files (plain text) do not have formatting while some do.
In general, a file is a sequence of bits, bytes, lines, or records

10.2.1.1 File Attributes

File attributes vary between operating systems, but in general consist of

- **Name** - only information that is kept in human readable form
- **Identifier** - unique tag (number) that identifies file within system
- **Type** - needed for systems that support different types
- **Location** - pointer to file location on device

- **Size** - current size
- **Protection** - controls read, write and execute privileges
- **Time, data, and user identification** - data for protection, security, and usage monitoring

Information about files are kept in the directory structure, which is maintained on secondary storage.

10.2.1.2 File Operations

Files are an **abstract data type** with the following defined operations

- **Create**

  Find space in the file system and then add an entry of the new file into the directory.

- **Write**

  The file system searches the directory to find the file's writing location, writes to that location, then updates the write pointer.

- **Read**

  The file system searches the directory to find the file's read location, reads from that location and then updates the read pointer.

- **Reposition**

  The file system searches the directory to find the file's location, then updates the read and write pointers.

- **Delete**

  The file system searches the directory to find the file's location, then removes the file from the directory and marks the space as available.

- **Truncate**

  The file system searches the directory to find the file's location, then removes the file from the directory and marks the space as available. The file system then updates the file's size attribute.

These primitive operatiosn can then be combined to perform other file operations (can create a file copy operation by creating a new file and then reading from the old and writing to the new)

- **Open**(F_i)

  Searches the directroy structure on disk for entry F_i and moves the content of the entry into internal system table. The entry contains information about the location of the file, its size, and other attributes. The system also allocates resources to manage the open file table.

- **Close**(F_i)

  Writes any cached information belonging to F_i to disk, removes F_i from the open file table, and frees up any table space allocated to F_i in the system.

10.2.1.3 Open Files

Several pieces of data are needed to manage open files

- **Open-file table**

  Tracks open files

- **File pointer**

  Pointer to last read/write location, per process that has the file open

- **File-open count**

  Counter of the number of times a file is open to allow removal of data from the open-file table when the last process closes it

- **Disk location**

  Cache of data access information

- **Access rights**

  Per process access mode information

10.2.1.4 Open File Locking

Provied by some operating systems and file systems and is similar to reader-writer locks

- **Shared lock**

  Similar to a reader lock. Several processes can acquire concurrently

- **Exclusive lock**

  Similar to a writer lock. Only one process can acquire at a time

Mandatory or Advisory

- **Mandatory**

  Access is denied depending on the locks held and requested

- **Advisory**

  Processes can find status of locks and decide what to do

10.2.1.5 File Structure

File types can be used to indicate the internal structure of a file where an empty type menas a sequence of words/bytes.
Simple record structure

- Lines
- Fixed length
- Variable length

Complex structures

- Formatted document
- Relocatable load file

If the operating system suports a file type, it needs to contain the code to support that file structure. Some operating systems (Unix, Windows) impose a minimal number of structures.

Example: Unix considers each file to be a sequence of 8-bit types. No interpretation for these 8 bits

10.2.2 Access Methods

- **Sequential Access**

```
read_next(record);
write_next(record);
reset(); /* Move the file pointer to the beginning of the file
```

- **Direct Access**

```
read(n);
write(n);
position_to(n); /* Move the file pointer to the nth record */
read_next();
write_next();
```

n = relative block number

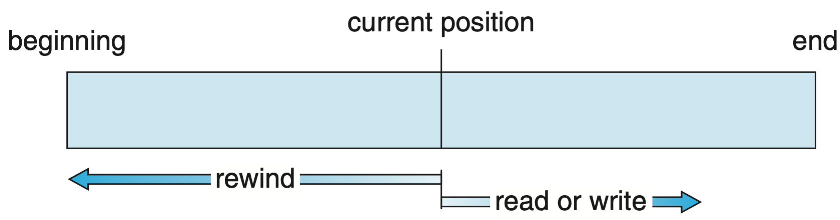


Figure 10.5: Sequential Access

sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp; cp = cp + 1;
write_next	write cp; cp = cp + 1;

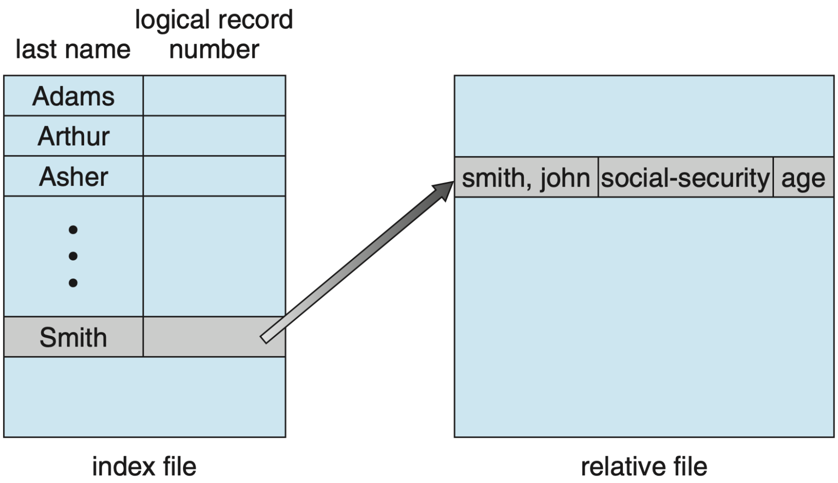
Figure 10.6: Simulation of sequential access on a direct access file

10.2.2.1 Other Access Methods

Access methods can be built on top of the direct access method, generally involving the creation of an index for the file. Keeping the index in memory can speed up locating data to be operating on (UP code + record of data about that item). Alternatively, index (in memory) or index (on disk) if space constraints arise.
IBM Indexed Sequential Access Method (ISAM)

- Small master index which points to disk blocks of secondary index
- Files are kept sorted on a defined key
- All managed on the operating system

VMS operating system provides index and relative files.



10.2.3 File-Systems

10.2.3.1 Directory Structure

Is a collection of nodes containing information about all the files.

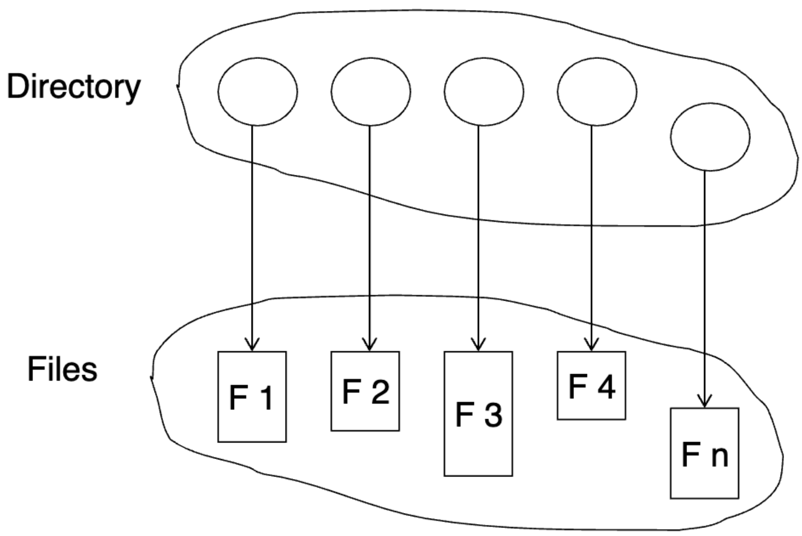


Figure 10.7: Directory Structure

10.2.3.2 Disk Structure

Disks, can be divided into partitions, which can be RAID (Redundant Array of Independent Disks) protected against a failure. They can also be used raw, without a file system/not formatting with a file system. Partitions are also called minidisks, slices. An entity containing a file system is also called a volume, where each volute containing file system also tracks that file system's information in the device directory or volume table of contents.
There are special purpose and general purpose file systems.
Frequently within the same operating system.

10.2.3.3 Types of File Systems

While we focus on general purpose file systems, there are special file systems used by operating systems as well. For example, with Solaris:

- tmpfs

  Temporary file system in memory, volatile and fast

- objfs

  Interface into kernel memory to get kernel symbols for debugging

- ctfs

  Contract file system for managing daemons

- lofs

  Loopback file system allowing one filesystem to be accessed in place of another

- procfs

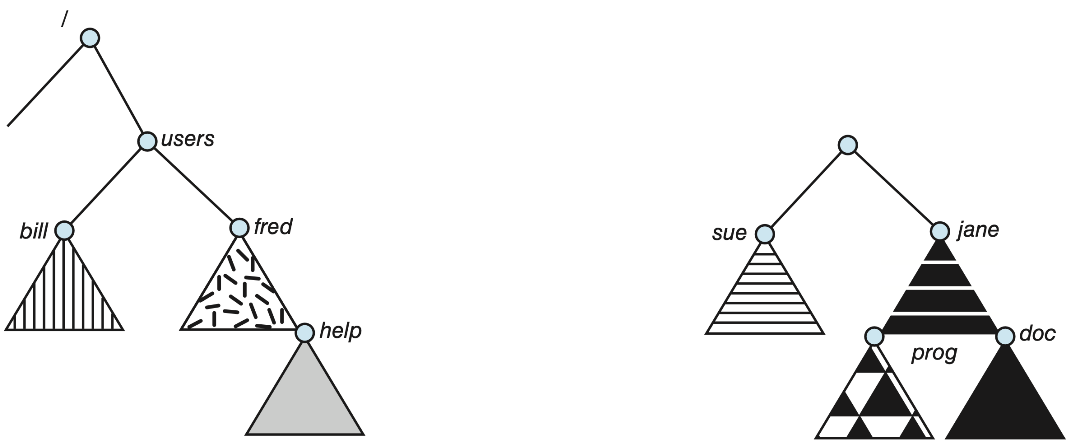
  Kernel interface to process structures

- ufs, zfs

  General purpose file systems

10.2.3.4 File System Mounting

A file system must be mounted before being accessed.
An unmounted file system is mounted at the mount point



(a) Existing System (b) Unmounted Volume

Figure 10.8: File System Mounting

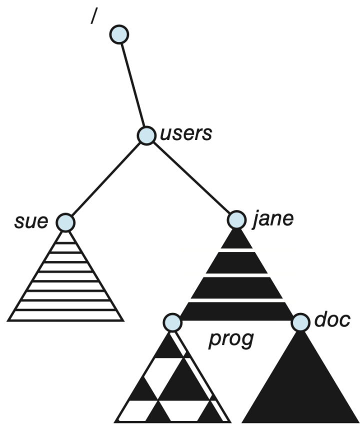


Figure 10.9: Mounted Volume at /users

10.2.4 Directory Structures

10.2.4.1 Operations Performed on Directory

A diretory can be viewed as a data structure that stores information about the files in the directory. The operations that can be performed on a directory are

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

10.2.4.2 Single-Level Directory

Simplest directory structure, where a single directory contains all the files on the disk.

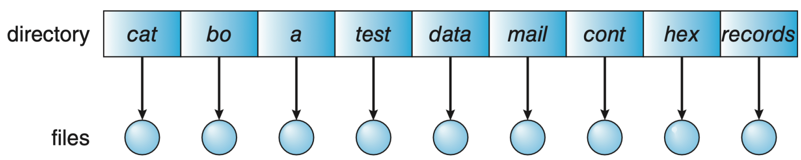


Figure 10.10: Single-Level Directory

10.2.4.3 Two-Level Directory

Seperate directory for each user (user file directory UFD) and one directory for all users (master file directory MFD).

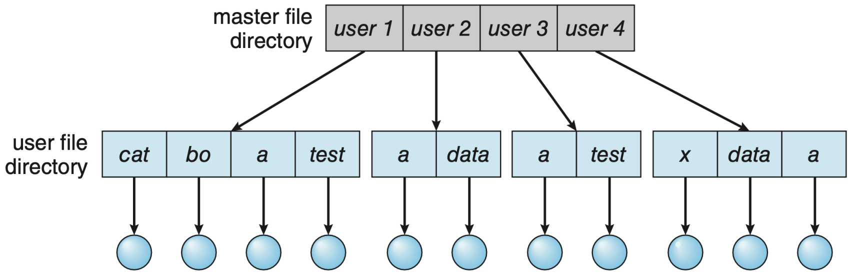


Figure 10.11: Two-Level Directory

10.2.4.4 Tree-Structured Directory

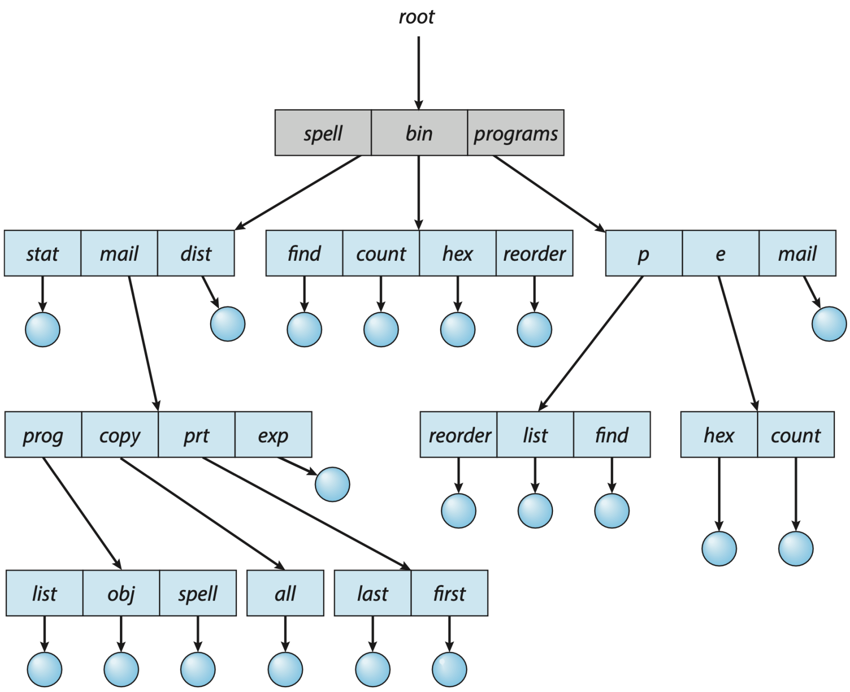


Figure 10.12: Tree-Structured Directory

10.2.4.5 Acyclic Graph Directory

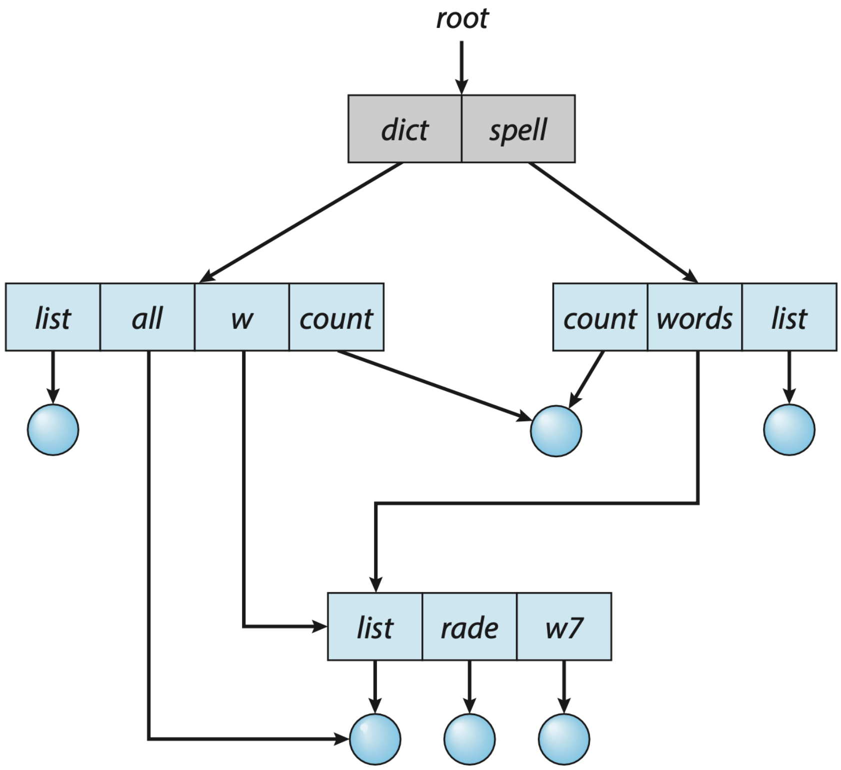


Figure 10.13: Acyclic Graph Directory

10.2.4.6 General Graph Directory

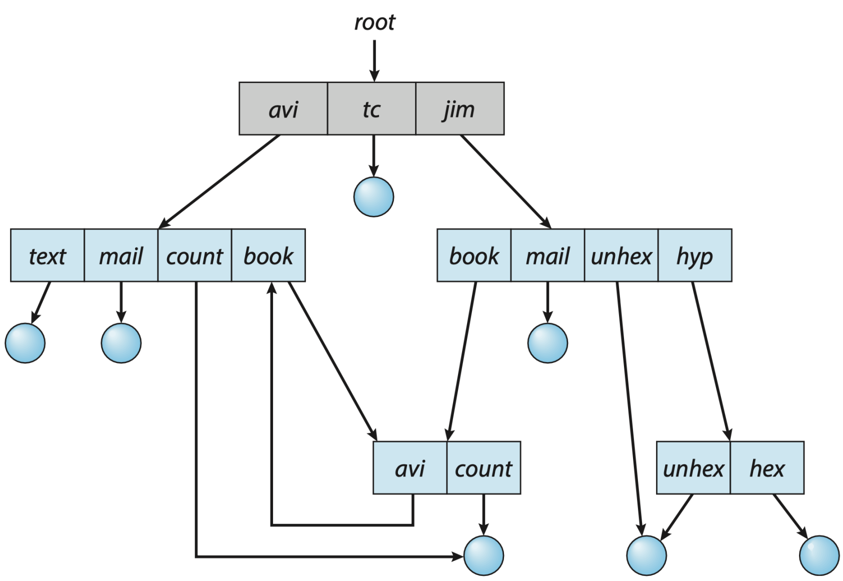


Figure 10.14: General Graph Directory

10.2.5 Protection

File owner/creator must be able to control what can be done and by whom.
The following types of access have been defined for files

- **Read**

- **Write**
- **Execute**
- **Append**
- **Delete**
- **List**

10.2.6 Access Lists and Groups

Modes of access: read, write, execute.
Three user classes in Unix

User	Group ID	rwx
Owner	7	111
Group	6	110
Public	1	001

Figure 10.15: Unix Access Classes

Chapter 11

Virtual Machines

Virtual machines abstract the hardware of a single computer (CPU, memory, disk drives, network interface cards, etc.) into several execution environments, creating the illusion that each environment is running on its own computer. Virtualization creates a layer on which the operating system/applications can run.
Virtual machine implmentations involve seperate components

- **Host** - the underlying hardware that runs the virtual machine
- **Virtual Machine Manager (VMM/Hypervisor)** - software that creates and runs virtual machines by providing an interface identical to the host
- **Guest** - the operating system and/or applications running on the hypervisor

11.1 Hypervisor

11.1.1 Type 0 Hypervisor

Is an older idea with many names, including **bare metal**, **native**, **embedded**, **domains**, and **partitions**.
Hardware features are implemented by firmware. The hypervisor is the first software to run on the hardware and the operating system requires no modifications.
Limited feature set but high performance as each guest gets dedicated hardware.
I/O is difficult as the hypervisor must implement all drivers.
VMM implements a control partition that runs daeomsn that other guests communicate with for shared I/O.
Allows for nesting virtualization, where a guest can run its own hypervisor and create its own guests.

11.1.2 Type 1 Hypervisor

Common in datacenters and cloud computing.
Datacenter managers control and manage operating systems in new, sophisticated ways by controlling the Type 1 hypervisor.
Consolidation of multiple operating systems and apps onto less hardware.
Move guests between systems to balance performance.
Allows for snapshots and cloning.
Rn in kernel mode, taking advantage of hardware protection. WHere the host CPU allows, they use multiple modes to give the guest operating system individual control and improved performance. They are also operating systems of their own, and therefore have their own CPU scheduling, memory management, and I/O subsystems.
Commonly closed source offerings (VMware ESX, Microsoft Hyper-V, Citrix XenServer)

These can run on Type 0 hypervisors but not on other Type 1 hypervisors.

11.1.3 Type 2 Hypervisor

These are less interesting from an operating system's perspective as there is little involvement in these.
Run in the application level as VMMs and are simplyu another process that is run and managed by the host.
The host is unaware that there is a VMM running.
Have limits not associated with other types such as requiring superuser/administrator to access hardware assistance features of modern CPUs and cannot use these features if not given these permissions and have extra overhead of running.
Have the benefit of being able to run on any host operating system and therefore can be used to test non native operating systems and software.

11.1.4 Paravirtualization

11.1.5 Programming-Environment Virtualization

11.1.6 Emulation

11.1.7 Application Containment

List of Figures