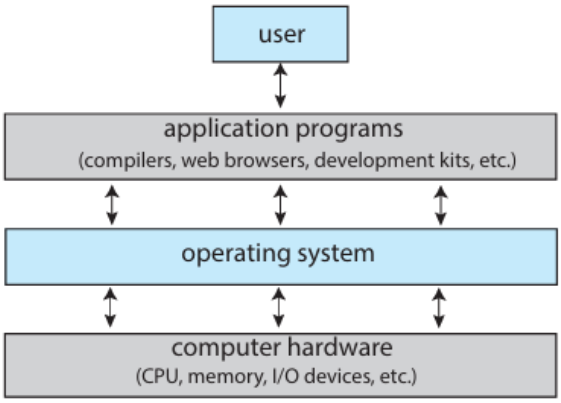
OPERATING SYSTEMS

1.0) INTRODUCTION

1.1) User View



re 1.1 Abstract view of the components of a computer system.

The user doesn't interact with the hardware. They interact with application programs and O.S. interacts with the hardware for the user.

1.2) Interrupts

Interrupt is the way hardware warns the software (such as Operating systems or device drivers) about various events such as:

1. I/O operations
2. Errors
3. Request for service from the device
4. Timer expiration for scheduling

An interrupt can be planned (requested by currently running program) or unplanned (caused by an event that might or might not related to running event). Interrupts are a key part how hardware communicates with software.

Hardware triggers an interrupt by sending a signal to CPU via system bus.

1.3) Storage Structure

The CPU loads information from the memory. So, any programs first should be loaded into memory to run. The first program to run when the power is on is **bootstrap program** (**AKA Boot Loader**). Which loads the OS.

Even though it'd be nice to have all the programs to reside in the main memory for reasons such as small storage size and volatility issues it is not possible. For that reason, we have the secondary storage.

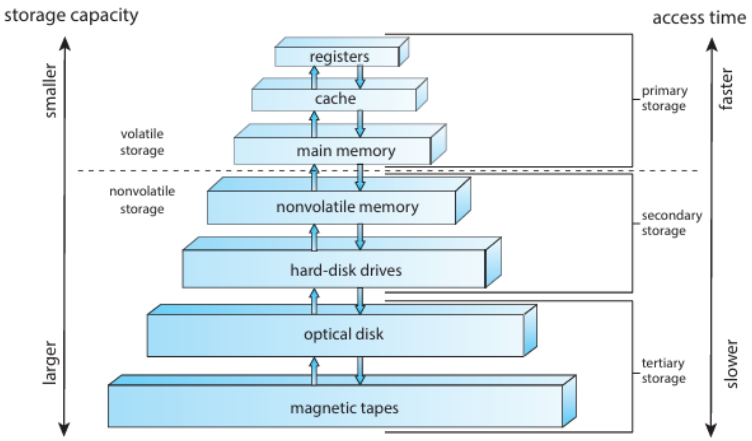


Figure 1.6 Storage-device hierarchy.

Generally, the volatile memory is referred as memory and the non-volatile memory is referred as NVS.

Some Terminology

CPU: Hardware that executes the instructions

Processor: A physical chip that contains one or more CPUs

Core: The basic computation unit of the CPU

Multicore: If includes multiple cores on the same CPU.

Multiprocessor: If includes multiple processors

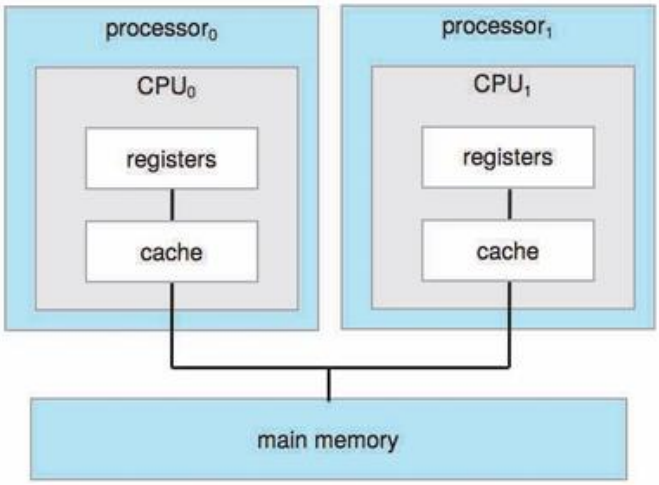


Figure 1.8 Symmetric multiprocessing architecture.

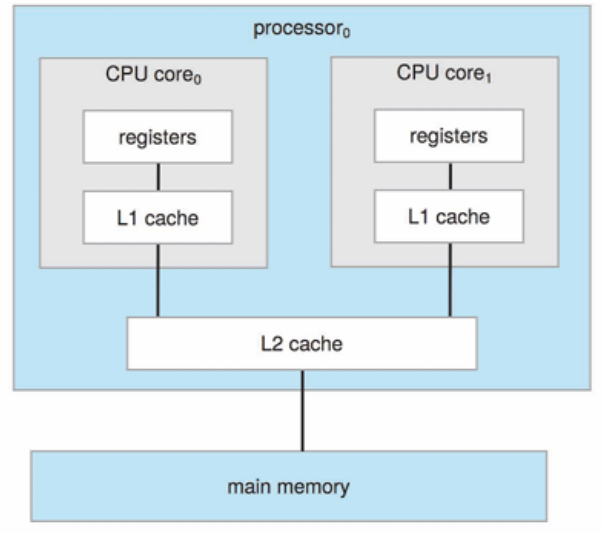


Figure 1.9 A dual-core design with two cores on the same chip.

1.4) Operating-System Operations

When the kernel is loaded and executing, it starts providing services to the system/users. Some services are provided outside the kernel by system programs are loaded into memory at the booting time and they are called **system daemons**. Daemons run entire time the OS is running. In Linux the first system daemon is *systemd* it starts many other daemons. Once it is done the system waits for some event to occur.

Some daemons are:

sshd: Responsible for ssh connections

httpd: Deals with incoming web requests

cron: Executes scheduled tasks.

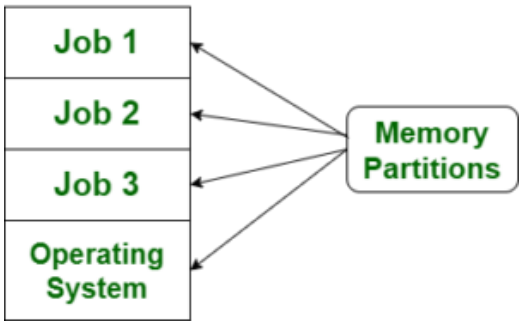
Another form of interrupt is **trap** (or **exception**). They are either caused by an error or by a specific request from a user program (which OS provides by executing a special operation called **system call**).

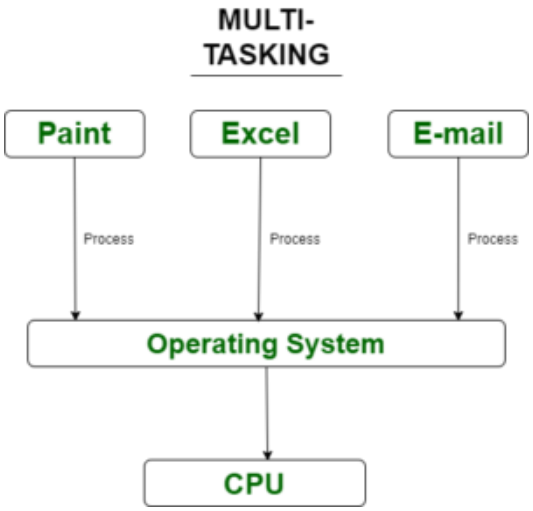
1.5) Multiprogramming and Multitasking

Since the CPU is the most expensive part of a computer, utilizing it is important. We do it via **multiprogramming**. It makes sure that the CPU has always have a process to execute. When a process needs to wait for a task such as I/O operation etc. The CPU switches to another process.

Multitasking is logical extension of multiprogramming. It can be said that it is an advance form of multitasking. In multitasking, the CPU executes multiple process by switching among them. The main difference is the switches occur more frequently, providing the user with a fast *response time*.

Multiprogramming





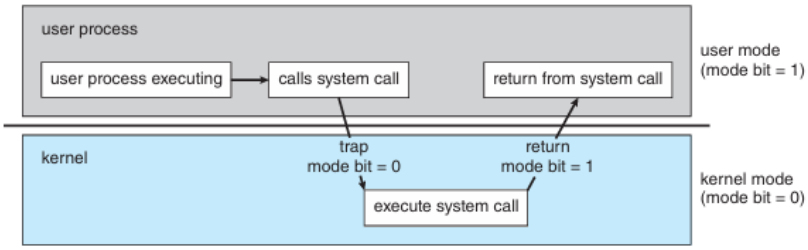
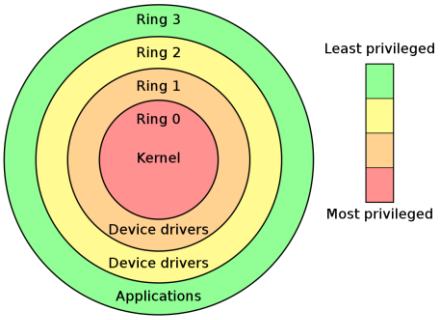


Figure 1.13 Transition from user to kernel mode

How the system calls are executed.



The intel processors have 4 separate **protection rings**. Ring 0 is the kernel mode and ring 3 is for user mode.

1.7) Timer

We must make sure that OS maintains control of CPU. We can't allow a process to get stuck in infinite loop etc. To do that we use **timer**. A timer can be set to interrupt a computer after a specified period.

1.8) Process Management

An operating system is responsible for the following activities connected to process management:

1. Creating and deleting user and system process
2. Scheduling processes and threads on the CPUs
3. Suspending and resuming processes
4. Providing mechanisms for process synchronization
5. Providing mechanisms for process communications

1.9) Memory Management

An operating system is responsible for the following activities connected to memory management:

1. Keep track of which parts of memory is currently used and which process is using them
2. Allocating and deallocating memory space as needed
3. Decide which processes to move in and out of memory

2.0) Operating-System Structures

2.1) System Calls

Almost all of the programs heavily use system calls. But application developers design their programs according to an **API** (**Application Program Interface**). 3 of the most used APIs are:

WINDOWS API for windows systems, POSIX API for POSIC-based systems (which includes virtually all versions of Unix, Linux and macOS) and the JAVA API for java

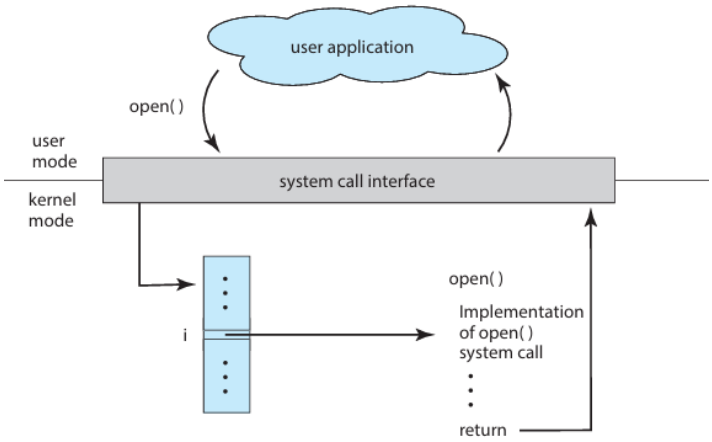


Figure 2.6 The handling of a user application invoking the `open()` system call.

System calls can be grouped into 6 categories. **Process control**, **file management**, **device management**, **information maintenance**, **communications** and **protection**.

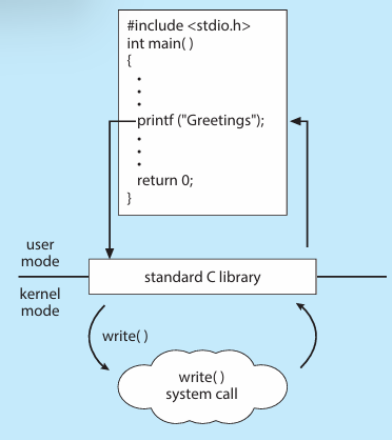
1. Process Control
    - Create and terminate process
    - Load, execute
    - Get process attributes, set process attributes
    - Wait event, signal event
    - Allocate and free memory
2. File Management
    - Create file, Delete File
    - Open, close
    - Read, write, reposition
    - Get file attributes, Set file attributes
3. Device Management
    - Request device, release device
    - Read, write, reposition
    - Get device attributes, set device attributes
    - Logically attach or detach devices
4. Information Maintenance
    - Get time or date, Set time or date
    - Get system data, Set system data
    - Get process, file or device attribute, Set process, file or device attribute
5. Communications
    - Create, delete communication connection
    -
    - Send, receive messages
    - Transfer status information
    - Attach or detach remote devices
6. Protection

- Get file permissions
- Set file permissions

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix
Process control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File management	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device management	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`
	`WriteConsole()`	`write()`
Information maintenance	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`
Communications	`CreatePipe()`	`pipe()`
	`CreateFileMapping()`	`shm_open()`
	`MapViewOfFile()`	`mmap()`
Protection	`SetFileSecurity()`	`chmod()`
	`InitlializeSecurityDescriptor()`	`umask()`
	`SetSecurityDescriptorGroup()`	`chown()`



Quite often two or more processes may share data. To protect the integrity of the data shared OS provides system calls allowing a process to **lock** shared data. The other process can't access to the data until it required the lock. Typically, such system calls include acquire_lock() and release_lock()

A single-CPU computer can execute only one instruction at a time. If a process is running a user program in user mode and needs a system service, it executes a **trap** instruction to transfer control to the operating system. The OS then figures what the calling process wants by inspecting the parameters.

Let's take the system call *read* in our hands. In the "count = read(fd,buffer,nbytes)" This code returns the number of bytes read to the count variable. This value is normally should be

same as nbytes variable passed to the read syscall but it might be smaller. For example, if there are 3 bytes and you try to read 5 bytes it returns 3. Or if it fails to read it returns –1.

When we call read the program first prepares the parameters. In x86_64 CPUs, Linux, macOS use the system C AMD64 ABI **calling convention**. Which means the first 6 parameters will be put to the registers and the rest will be pushed onto the stack.

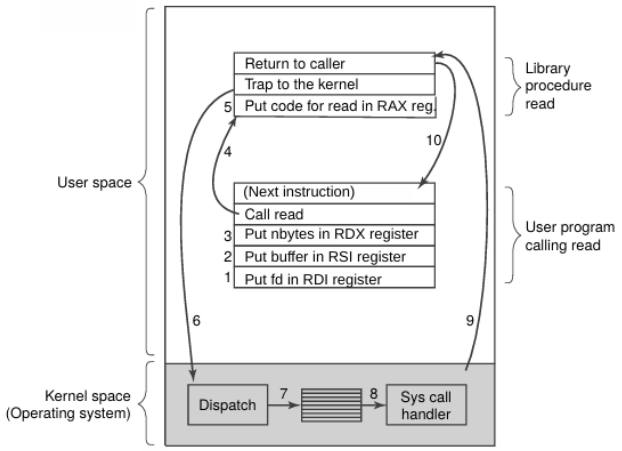


Figure 1-17. The 10 steps in making the system call read(fd, buffer, nbytes).

First and third parameters are passed by value, second by reference. The library procedure generally puts the system-call number in a place where the operating system expects it (Step 5). Then it executes the **trap** instruction to switch from user mode to kernel mode and start executing at a fixed address within the kernel (STEP 6).

Note: A trap instruction in operating systems is a special kind of operation or instruction used to transition from user mode to kernel mode, allowing a user-level program to request services or functionality provided by the operating system's kernel.

The kernel code that starts following the trap examines the syscall number in the RAX register and dispatches to the correct syscall handler. This handler is table of pointers to syscall handlers indexed on syscall number (STEP 7). Afterwards syscall handler runs (STEP 8). And once it is over control is returned to the trap instruction (STEP 9). Returns to the caller (STEP 10) and then continues with the next instructions

You can see some of the system calls below:

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory- and file-system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

To open a file with system call, you need to specify the filename with either relative to the C.W.D. or its absolute path and then you should also specify whether you want to read, write etc. with *O_RDONLY*, *O_WRONLY*, *O_RDWR*. To create a new file, you do *O_CREAT*.

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount, so no umount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

2.2) System Services

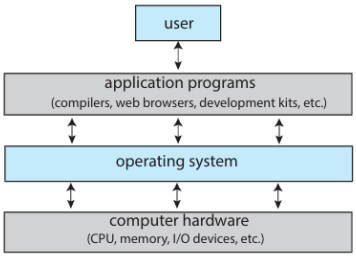


Figure 1.1 Abstract view of the components of a computer system.

The system services lie between OS and application programs. They provide a convenient environment for program development and execution.

2.3) Linkers and Loaders

Usually, a program resides on disk as a binary executable file. To run on a CPU, the program must be brought into memory and placed in the context of a process.

Source files are compiled into object files that are designed to be loaded into any physical memory location, a format known as **relocatable object file**. Next, the **linker** combines these relocatable object files into a single binary executable file.

Next a **loader** loads the binary executable to the memory.

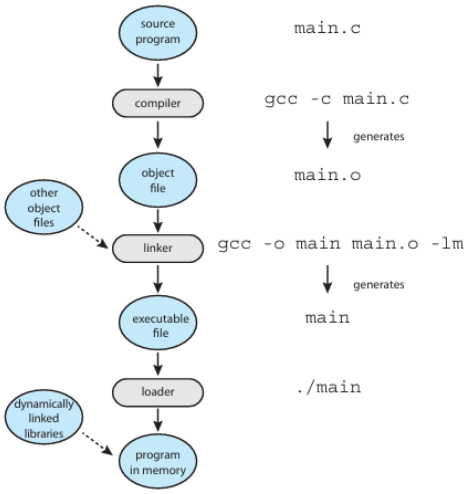


Figure 2.11 The role of the linker and loader.

Let's say we are trying to run an executable in Linux with "./main" The shell creates a new process to run in the program using the fork() system call. Then the shell invokes the loader with exec() system call (it passes the name of the executable to the exec). The loader then loads the specified program into memory using the address space of the newly created process.

Windows uses **DLL**s, which allow for dynamic linking and loading of libraries at runtime, ensuring that only the necessary libraries are loaded into memory when required by a program. For instance, if you wrote a hello_world.c program with unnecessary libraries, thanks to DLLs, the loader doesn't load them into memory. In Linux, the equivalent of DLL files is **.so** files.

Object files and executable files have standard formats that include the compiled machine code and a symbol table containing metadata about functions and variables that are referenced in the code. This standard form is known as **ELF** (**Executable and Linkable Format**).

Main.o is an ELF relocatable file while Main is an ELF executable.

2.4) Why Are Applications OS Specific?

The main reason is that different operating systems have different unique system calls. If we want to make our application to run on multiple systems, we can have:

1. Application can be written in an interpreted language (Python, Ruby etc.) that has multiple interpreter available for multiple OS.
2. The application can be written in, in a language that includes a virtual machine containing the running application such as Java.
3. The application developer might use a standard language or API in which the compiler generates binaries in a machine and OS specific language. The best-known example of it is POSIX API.

2.5) Operating-System Structure

1. Monolithic Structure

   The simplest structure for organizing OS is no structure at all. This approach is known as monolithic approach

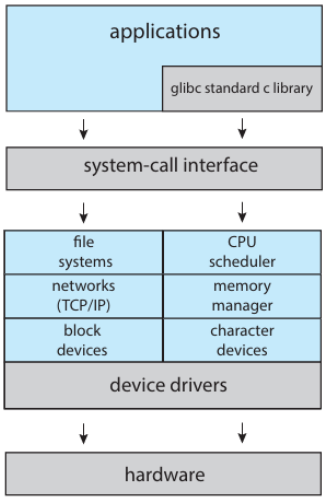


Figure 2.13 Linux system structure.

2. Layered Approach

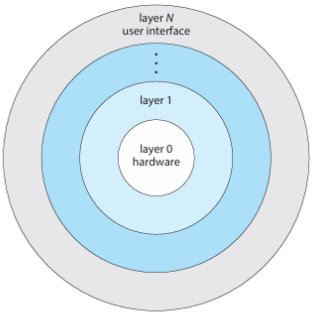


Figure 2.14 A layered operating system.

One of the biggest Monolithic

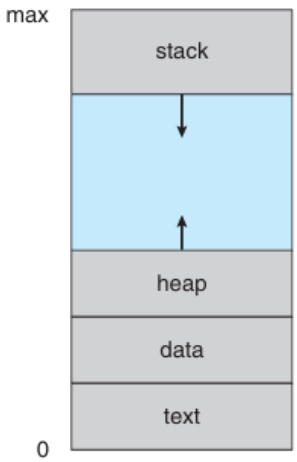
3. Microkernels
4. Modules
5. Hybrid Systems
    a. MacOS and iOS
    b. Android

3.0) PROCESS MANAGEMENT

3.1) Process Concept

Earlier computer systems were batch systems that executed **jobs**, followed by the emergence of time-shared systems that ran **user programs**, or **tasks**. Today's computers are able to run multiple programs at one time, for example word processor, web browser, e-mail package etc. These activities are called processes.

3.2) The Process



The status of the current activity of a process is represented by the value of the **program counter**. In the figure above, the **text** is the executable code, **the data section** is the global

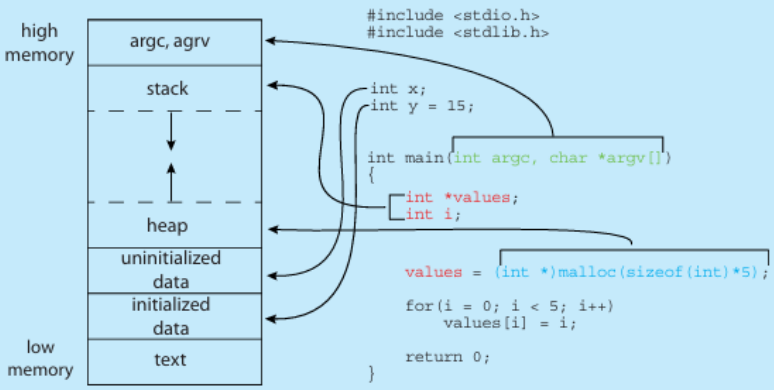
variables, furthermore the data is separated into two, initialized data and uninitialized data. Heap is allocated memory and in stack we hold the addresses of the functions we are going from temporarily.

As it can be understood from above the size of text and data is fixed. However, the heap and stack are able to shrink and grow. Every time we are calling a function an **activation record** containing the functions parameters, local variables and the return address is pushed into the stack.

A more detailed form of memory is:

The figure shown below illustrates the layout of a C program in memory, highlighting how the different sections of a process relate to an actual C program. This figure is similar to the general concept of a process in memory as shown in Figure 3.1, with a few differences:

- The global data section is divided into different sections for (a) initialized data and (b) uninitialized data.
- A separate section is provided for the `argc` and `argv` parameters passed to the `main()` function.



The GNU `size` command can be used to determine the size (in bytes) of some of these sections. Assuming the name of the executable file of the above C program is `memory`, the following is the output generated by entering the command `size memory`:

```
text    data    bss     dec     hex     filename
1158    284     8       1450    5aa     memory
```

The `data` field refers to uninitialized data, and `bss` refers to initialized data. (`bss` is a historical term referring to ***block started by symbol***.) The `dec` and `hex` values are the sum of the three sections represented in decimal and hexadecimal, respectively.

Please know that there are several distinctions between programs and processes. Programs are passive entities while the processes are active entities. A program becomes process when and executable file is loaded into the main memory. BTW, you can load an executable either by double-clicking on executable or from the terminal

3.3) Process State

When a process executes it changes its **state**. A state of a process is the processes current activity. Is it ready for execution, is it executing, is it terminated, is it blocked/waiting.

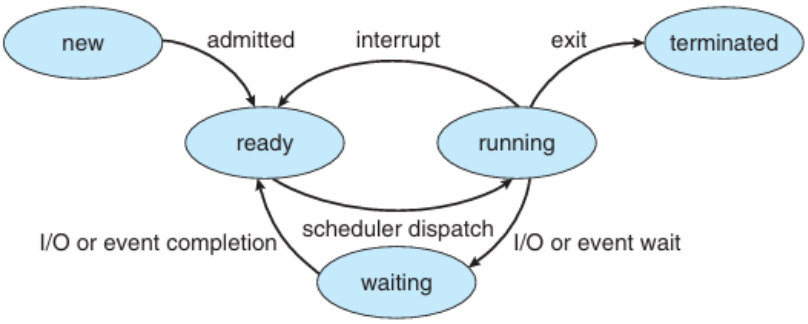


Figure 3.2 Diagram of process state.

3.4) Process Control Block

A **process control block** is also known as **task control block**. It basically holds all the information related to a process.

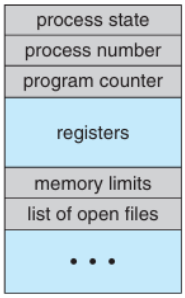


Figure 3.3 Process control block (PCB).

Process state: Process state, states whether the code is new, ready, running, waiting or terminated

Program Counter: The counter indicates the address pf the next instruction to be executed.

CPU Registers: They vary in number and type wrt. the computer architecture.

CPU-scheduling information: Includes process priority, pointers to scheduling queues and any other scheduling parameters.

Memory Management Information:

Accounting information: Amount of CPU and real time used, time limits etc.

I/O status information: This information includes the list of I/O devices allocated to the process, a list of open files, etc.

3.5) Threads

Threads allows a program to do more than one thing. For example, in word processing, if you have a single thread, you can only write you can't do spell checking etc. It runs quasi-parallel, as if they were different processes. But be careful, different threads aren't as independent as different processes. Threads are sometimes called **lightweight processes**.

Per-process items	Per-thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Figure 2-11. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

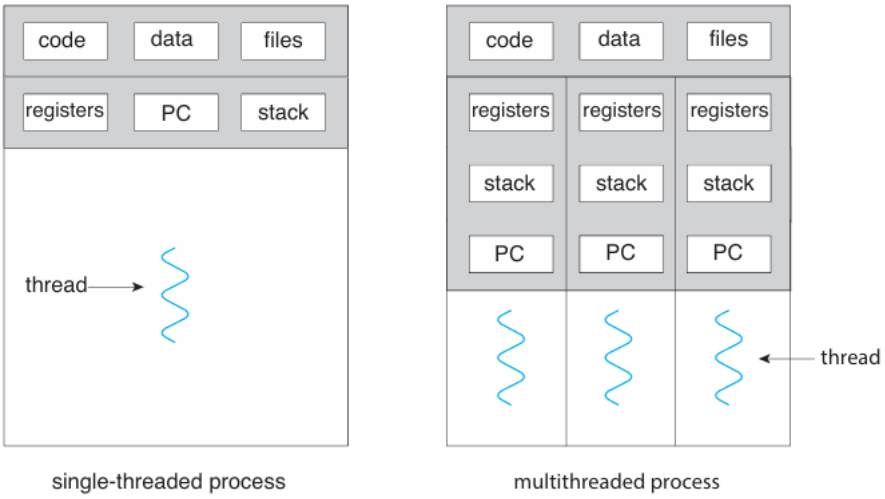


Figure 4.1 Single-threaded and multithreaded processes.

Each thread has their own registers, stack and PC. But everything else, such as heap memory, sockets, file descriptors, global variables are shared. Resource allocated by one thread is visible to others (Since resource allocation is made within the heap memory).

The threads are what is executed. So, the kernel (OS) schedules that. Because they are the actual unit of execution. And kernel decides which thread runs on which CPU schedule and when.

However, if a thread causes an error, then the entire process is terminated. (Because a signal is delivered per process and not per thread.)

3.5.1) Multicore Programming

If a processing chip has multiple computing cores, then it is called **multicore**. If a chip has only one core, then the core will only be able to execute one thread at a time. If every operation is CPU-bound on a single-core CPU, then threads won't provide any performance gain. However, if some operations are I/O-bound, threads can improve performance. If the system has multiple CPUs, parallelism is possible.

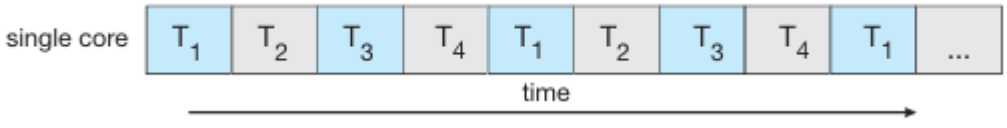


Figure 4.3 Concurrent execution on a single-core system.

However, if a computer has multiple cores, then it can execute different threads.

A concurrent system supports more than one task by allowing all the tasks to make progress. Parallel systems allow the execution of more than one task simultaneously. Before multicore computers to create the illusion of parallelism CPU schedulers switched between

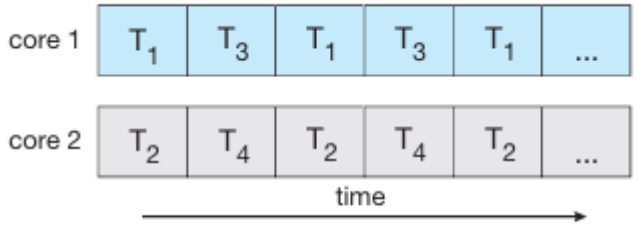


Figure 4.4 Parallel execution on a multicore system.

processes.

3.5.2) Multithreading Models

Support for threads may be provided either at the user level via **user threads** or by the kernel via **kernel threads**. Multithreading is allowing multiple threads in the same process. If multithreads are running on a single-CPU system, the threads take turns running.

3.5.2.1) Many-To-One Model

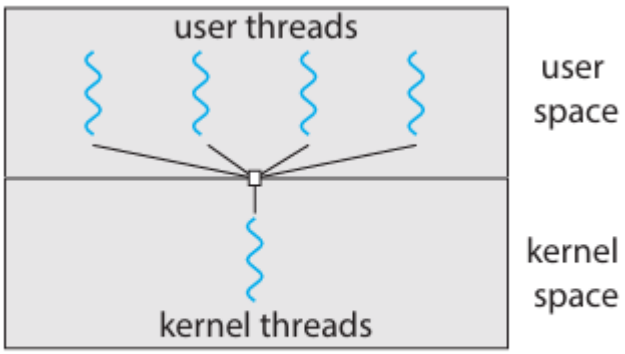


Figure 4.7 Many-to-one model.

Many-to-ne modeling maps many user-level threads to one kernel thread. It is done by the thread library making it efficient. But the entire process will be blocked if one of the threads is blocked. Moreover, since only one thread can access the kernel and multiple threads are unable to run in parallel.

3.5.2.2) One-To-One Models

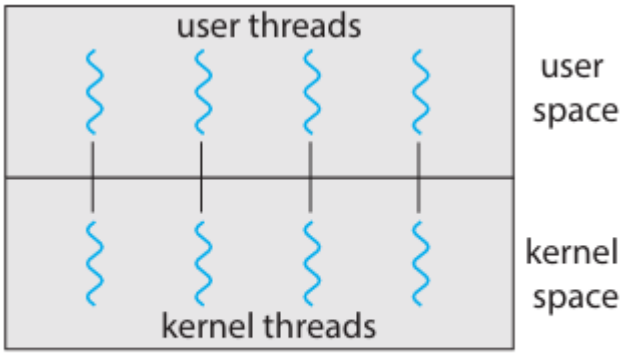
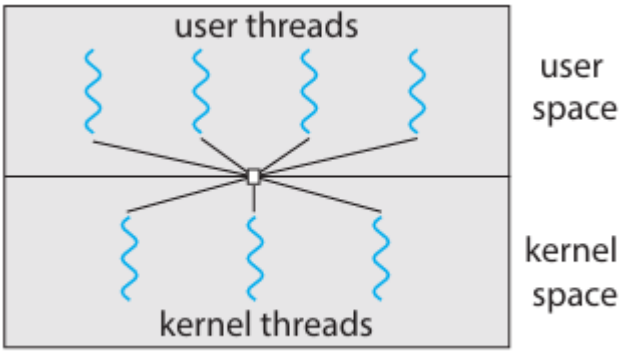


Figure 4.8 One-to-one model.

Maps each user thread to a kernel thread. It provides more concurrency than many-to-one models by allowing different user threads to have different kernel threads. Multiple threads can run on multiple processors. In this model, creating a user thread requires the creation of a kernel thread. And a large number kernel threads may burden the system.

3.5.2.3) Many-To- Many Models



In many to many models the number of kernel threads are less than or equal to number of user threads.

Overall, one-to-many models don't really create parallelism. For the kernel can only schedule one thread at a time. Even though one-to-one models allow for greater concurrency, developers have to be careful not to burden the system. And many-to-many doesn't suffer either of these shortcomings. Developers can create as much kernel threads as possible and the corresponding kernel threads can run in parallel in multiprocessors.

3.5.3) Pthreads

Now, let's check the code below. The tid is the identifier of the thread. Attr is its attribute. With the pthread_create we are specifying the name of a procedure/function for our thread to run.

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

#include <stdlib.h>

int 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 tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   /* set the default attributes of the thread */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

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

/* The thread will execute in 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 4.11 Multithreaded C program using the Pthreads API.

And here are windows threads:

```c
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in 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 the thread identifier */

    /* now wait for the thread to finish */
   WaitForSingleObject(ThreadHandle,INFINITE);

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

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

Figure 4.13 Multithreaded C program using the Windows API.

In Linux systems, fork of a multithreaded process will create only a single thread in the child. To overcome this issue, we can use POSIX thread's function called pthread_atfork().

Thread call	Description
pthread_create	Create a new thread
pthread_exit	Terminate the calling thread
pthread_join	Wait for a specific thread to exit
pthread_yield	Release the CPU to let another thread run
pthread_attr_init	Create and initialize a thread's attribute structure
pthread_attr_destroy	Remove a thread's attribute structure

Figure 2-13. Some of the Pthreads function calls.

Thread Creation Steps:

1. **Define thread**: pthread_t my_thread
2. **If necessary, define attribute**: pthread_t_attr attr
3. **If necessary, initialize attribute**: pthread_attr_init (&attr)
4. **Create thread**: pthread_create(&my_thread, &attr, thread_func, NULL)
5. **If necessary, destroy attribute**: pthread_attr_destroy(&attr)
6. **Join the threads**: pthread_join (my_thread, NULL)

```c
#include <pthread.h>$
#include <stdio.h>$
#include <stdlib.h>$
void* thread_func(void* arg) {$
    printf("Thread is running\n");$
    return NULL;$
}$
int main() {$
    pthread_t my_thread; // Defining thread$
    pthread_attr_t attr; // Defining attribute$
    pthread_attr_init(&attr); // Initializing attribute$
    if (pthread_create(&my_thread, &attr, thread_func, NULL) != 0){ //Thread creation$
        fprintf(stderr, "Error creating thread\n");$
        return 1;$
    }$
    pthread_attr_destroy(&attr); //Destroying attribute$
    if (pthread_join(my_thread, NULL) != 0){ // Joining threads$
        fprintf(stderr, "Error joining thread\n");$
        return 1;$
    }$
    printf("Thread has terminated\n");$
    return 0;$
}$
```

Joinable Threads: Assume that a parent thread creates a child thread. Afterwards, the child thread does its thing and parent thread does its thing. And parent thread tries to join with child thread using pthread_join() function. At this stage parent thread is blocked until the child thread comes back and joins it. Resources of the new thread is only released when is joins the caller thread.

```c
void* child_thread_function(void* arg) {$
    int i = 0;$
    while(i < 5) { // Child performs some tasks$
        printf("Child thread is running i=%d\n", i);$
        sleep(1);$
        i++;$
    }$
    pthread_exit(NULL);$
}$
$
int main(){$
    pthread_t child_thread;$
    pthread_attr_t attr;$
    if (pthread_attr_init(&attr)!=0){ //Initialized the attribute$
        printf("Failed to initialize attribute");$
    }$
    if(pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE)!=0){ // Made it joinable$
        printf("Failed to set it joinable");$
    }$
    if (pthread_create(&child_thread, &attr, child_thread_function, NULL) != 0){ //Threads are joinable by default actually.
        printf(s"Failed to create thread\n");$
        return 1;}$
$
    printf("Parent thread is doing its thing\n"); // Perform some tasks$
    if (pthread_join(child_thread, NULL) != 0) { // Wait for child to terminate and join.$
        printf("Failed to join threads\n");$
        return 1;$
    }$
    printf("Child thread has terminated and resources are released\n");$
    return 0;$
}$
```

In this example the main thread creates a new thread that goes to the child_thread_function. Afterwards, both main and the newly created thread does its tasks. And the main thread waits for newly created thread to return at the pthread_join part of our code.

Detached Threads: However, if a thread is detached it is not joinable. As you can see from the code below.

```c
#include <pthread.h>$
#include <stdio.h>$
void* thread_function(void* arg) {$
    printf("Detached thread is running\n");$
}$
int main() {$
    pthread_t thread;$
    pthread_attr_t attr;$
    pthread_attr_init(&attr); //Initing the attribute$
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED); //We set the thread to be detached$
    pthread_create(&thread, &attr, thread_function, NULL); //Creating the thread$
    printf("Detached thread has been created\n");$
    if (pthread_join(thread, NULL) != 0) { //Wait for child to terminate and join it.$
        fprintf(stderr, "Error joining thread\n");$
        return 1;}$
    return 0;$
}$
```

Please note that a joinable thread can be converted into detached thread while it runs and vice versa.

Joinable To Detach: For example, in the code below you can see a thread that is started as a joinable thread turning into detached.

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

void* thread_function(void* arg) {
    printf("Thread started\n");
    sleep(2); // Simulate some work
    printf("Thread finished\n");
    return NULL;
}

int main() {
    pthread_t thread;
    if(pthread_create(&thread, NULL, thread_function, NULL)!=0){ //Created a joinable thread (It is joinable by default)
        printf("Couldn't create thread\n");
        return 1;
    }

    if(pthread_detach(thread) != 0){ //Detached the thread.
        printf("Error detaching thread\n");
        return 1;
    }
    printf("Thread detached\n");

    sleep(3);
    if(pthread_join(thread,NULL)!=0){
        printf("Failed to join threads\n");
    }
    return 0;
}
```

Important Note About Thread Joins:

Any thread can invoke pthread_join() for any other joinable thread. Not just parent thread.

Important Note About Thread Attributes:

Thread attributes allow developers to customize the behavior of threads to fit in the application requirements. Some of the common thread attributes are:

1. **Detach State**: With pthread_attr_setdetachstate() you can set to either detached or joinable state. And with pthread_attr_getdetachstate you can check whether or not it is detached.
2. **Stack Size**: With pthread_attr_setstacksize you can set stack size and with pthread_attr_getstacksize you can retrieve stack size.
3. **Scheduling Policy**: Usage: “int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);” And the policy is: SCHED_FIFO, SCHED_RR, SCHED_OTHER.
4. **Scheduling Parameters**: Usage: “int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);”

Map-Reduce method is a programming method of dividing and conquering. We basically divide the problem into subproblems and create a worker thread for each. Worker threads are **called mappers**. Mappers work on non-shared data independently. The thread who waits for all worker threads to finish is called **reducer thread**.

Should A Thread Be Joinable Or Detached?

If a thread must return some result to other threads or when some threads are interested in being notified of other thread’s termination then you should make it joinable. For example, map reduce.

But if no return result is expected and nobody bothers about its death then it is detached. Examples are, waiting for user input, waiting for network pkt, TCP server’s worker.

If the threads are managed in the user space, each process needs its **own thread table** to keep track of threads of those processes. If a thread is moved to ready or blocked state, the information needed to restart it is stored in the thread table.

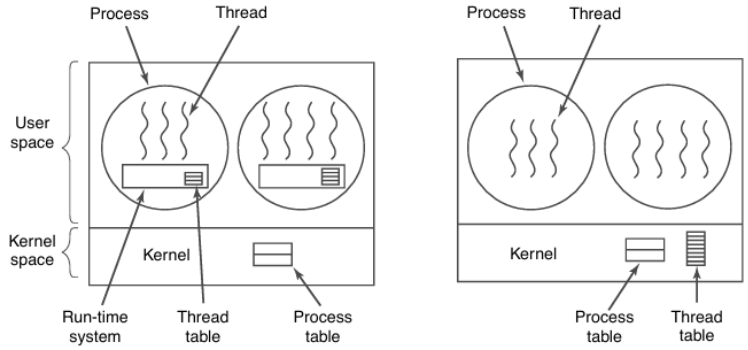


Figure 2-15. (a) A user-level threads package. (b) A threads package managed by the kernel.

If a thread is blocked it calls a run-time system procedure. Procedure checks the necessity of putting thread into blocked state. If yes, it stores the thread's registers in the thread table. Checks the table if it is ready to run, and reloads the machine registers with new thread's saved values. When stack pointer and PC are switched the thread comes life again automatically.

Another common thread call, which is not used below, is thread_yield. It makes thread to voluntarily give up the CPU to let another thread run. When a thread is finished running for the moment it calls the thread_yield. It saves thread's information in the table then calls thread scheduler to run another thread.

In UNIX systems handling situations where reading from a source can cause process the block for the data isn't ready yet. So, we have a **select** system call. This checks if a read operation would block before actually attempting to read. It works like this:

1.) Instead of directly calling 'read', a new procedure calls select. It checks whether the state is safe (Reading won't block anything)

2.) If it is deemed safe, 'read' call is made. Else another thread can run instead.

3.) The system continuously checks to see whether the read operation can be performed safely. Moreover, the code that checks if the read operation is safe is called a **wrapper**.

4.) Although it requires changes to the system call library and thus inefficient, it is necessary if no better options are available.

5.) LINUX OS have more efficient methods for asynchronous I/O. In LINUX we have "epoll".

If a page fault occurs (a page fault is when a process requests data that is not currently in memory), the process is blocked while the required data is fetched from the disk. This is called a page fault. If a thread causes a page fault, the kernel typically blocks the entire process, even if other threads might be runnable.

3.5.4) Implicit Threading

To deal with the troubles of the multi-threading and better support the design of concurrent and parallel applications is using **implicit threading**. Here I'll write about the some of the main approaches.

3.5.4.1) Threads Pools

Let's take a multi-threaded web server in our hands. Whenever a new request comes it needs to create a new thread however, there are some problems with this. To begin with it takes time to create a thread. The second issue is resource exhaustion. Unlimited threads might deplete system resources such as memory or CPU.

Thread pool concept: To resolve these issues, we can use **thread pools**. We pre-create a fixed number of threads at the beginning and these threads wait in pool for tasks.

Request Handling: When a request is submitted, an available thread handles the request, if no thread is available, the request is queued.

And using existing threads is faster than creating new ones. Moreover, limiting the number of threads is important for systems with limited resources. And it allows for different strategies for task execution (example delayed or periodic execution)

Factors such as number of CPUs, physical memory and expected requests are used to determine the pool size. Moreover, dynamic adjustment of pool size based on usage is also used to determine it.

Windows Thread Pools

1. **CreateThreadpool:** Creates a new thread pool.
2. **SetThreadpoolThreadMaximum:** Sets the max. number of threads.
3. **SetThreadpoolThreadMinimum:** Sets the min. number of threads.
4. **SubmitThreadpoolWork:** Submits work to the thread pool.
5. **CreateThreadpoolWork:** Creates a work object that can be submitted to the thread pool.

Java Thread Pools

1. **Single Thread Executor**: 'newSingleThreadExecutor ()' creates a thread pool with one thread. Needs to be used if tasks are to be executed sequentially.
2. **Fixed Thread Method**: 'newFiexdThreadPool (int size)' creates a thread pool with a fixed number of threads. Suitable if concurrent tasks are limited in number.
3. **Cached Thread Method**: 'newCachedThreadPool' creates an unbound thread pool that reuses threads if possible. Ideal for short-lived asynchronous tasks.

3.5.4.2) Fork Join

It is a synchronous threading strategy. Library takes care of creation and joining of the threads. It is good for handling divide & conquer class types of problems. Java also has this.

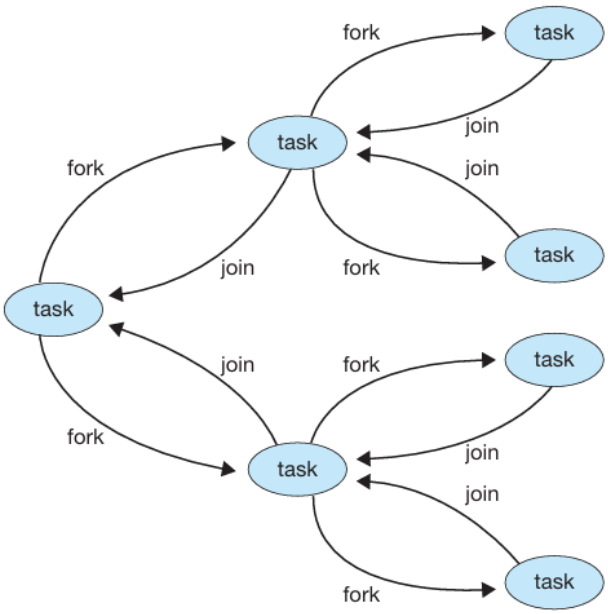


Figure 4.17 Fork-join in Java.

The mentality is basically this:

```
Task(problem)
   if problem is small enough
      solve the problem directly
   else
      subtask1 = fork(new Task(subset of problem)
      subtask2 = fork(new Task(subset of problem)

      result1 = join(subtask1)
      result2 = join(subtask2)

      return combined results
```

We divide a task into 2 tasks. If these sub-problems still too large we continue on dividing it.

3.5.4.3) OpenMP

It allows parallel programming in shared memory. It is used in C/C++, FORTRAN.

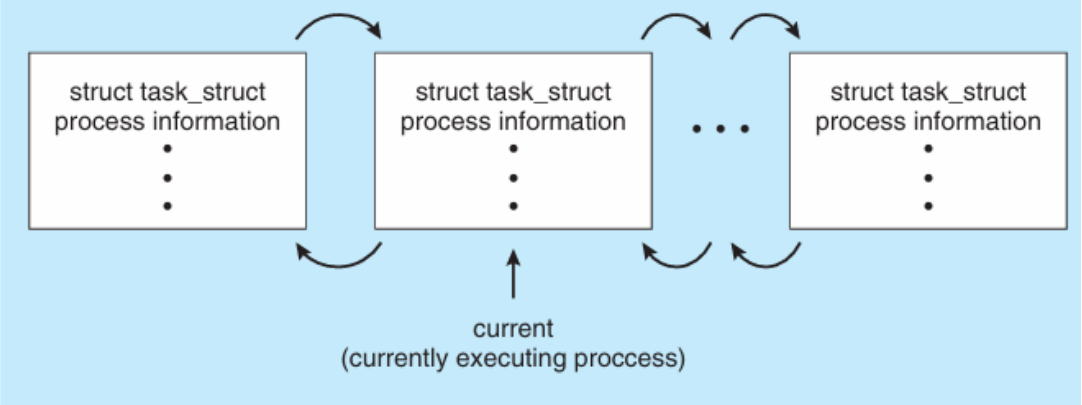
3.5.5) Process Scheduling

Time sharing is switching the CPU among processes so frequently that users thinks its multiprogramming.

Each CPU core can run one process at a time. The **process scheduler** selects an available process for execution.

In the Linux, process control block is represented with task_struct C structure, which is a part of <include(linux/sched.h> include file in the kernel source directory. This structure has all the necessary information for representing a process. It includes:

```
long state;                     /* state of the process */
struct sched_entity se;         /* scheduling information */
struct task_struct *parent;     /* this process's parent */
struct list_head children;      /* this process's children */
struct files_struct *files;     /* list of open files */
struct mm_struct *mm;           /* address space */
```



The number of processes currently in the memory is known as the **degree of multiprogramming**.

I/O-bound process is a process that spends doing more I/O operation than computing and **CPU-bound process** is deals with computation.

3.5.6) Process Operations

3.5.6.1) Process Creation

A process can create other processes and then the processes created can create their own processes resulting in formation of a tree of processes. Almost all operating systems identify the processes with a unique **process identifier** (**pid**). In Linux operating systems, with the fork() system call you can create a new process. *Return for fork is zero for the new (child) process and pid of the child for the parent*.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Figure 3.8 Creating a separate process using the UNIX `fork()` system call.

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

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
       fprintf(stderr, "Create Process Failed");
       return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Figure 3.10 Creating a separate process using the Windows API.

3.5.6.2) Process Termination

A process can die when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call.

Some systems don't allow child to exist without its parent, so if the parent exits it forces child to exit as well in a phenomenon called **cascading termination** which is initiated by OS

3.5.7) Inter-Process Communication

If a process doesn't share data with any other process, it is called *independent*. If it shares it is *cooperating*. Cooperating processes require an **interprocess communication** (**IPC**) mechanism that allows exchange of data. There are 2 models for interprocess communication: **Shared memory** and **message passing**.

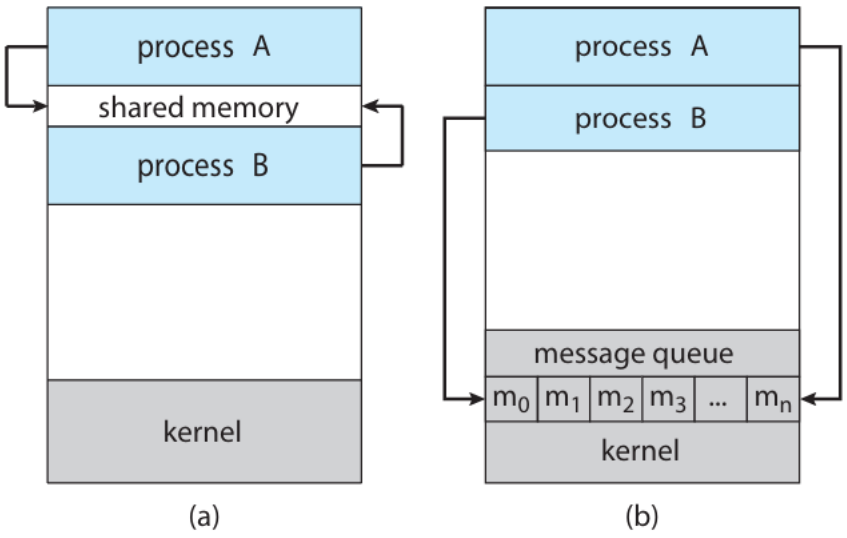


Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

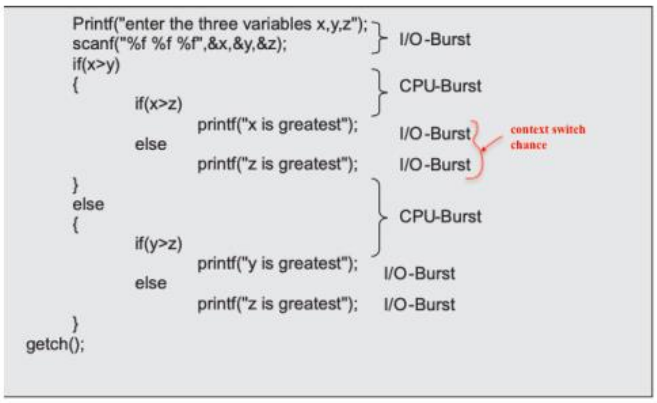
IPC in Shared-Memory Systems requires the communicating processes to establish a region of shared memory. In general, shared memory area resides within the address space of the process creating the shared memory segment.

IPC in Message-Passing Systems requires processes share a region of memory and that the code for accessing and manipulating the shared memory be written explicitly by the application programmer. It allows the synchronization without sharing the same address space.

3.6) CPU Scheduling

3.6.1) Basic Concepts

Since the CPU is one of the most expensive, if not the most expensive, part in a computer it would be wiser to make it optimized. In this part we'll discuss how the multiprogramming maximizes the CPU utilization by making sure it always executes something.



Look at the example above. During the I/O-bursts the CPU doesn't work and therefore wastes time. For situations like this the O.S keeps several processes in the memory at one time. And when a process needs to wait the other process gets executed.

Scheduling is fundamental. Almost all of the computer resources are scheduled before use.

3.6.1.1) CPU-I/O Burst Cycle

If the processes alternate between CPU execution and I/O wait it is called CPU-I/O burst cycle. It starts with CPU burst, followed by I/O burst and ends with the system request to terminate.

An I/O-bound program typically has many short CPU bursts.

An CPU-bound program has a few long CPU bursts.

3.6.1.2) CPU Scheduler

Whenever the CPU is idle the OS must select one of the processes in the ready queue to be executed. Ready queue can be many algorithms and not necessarily FIFO queue.

3.6.1.3) Preemptive and Non-preemptive Scheduling

1. If a process is switched from the running state to waiting state
2. If a process is switched from the running state to ready state
3. If a process is switched from the waiting state to ready state
4. When a process terminates

CPU scheduling decisions come into play. For the situations 1 and 4 there are no scheduling choices yet in situations 2 and 3 there are. When scheduling takes place under first and fourth choices, we say the scheduling is **non-preemptive**. Else they are **preemptive**.

Under non-preemptive scheduling, one the CPU is allocated to a process, *the process keeps the CPU until it releases either by terminating or switching to waiting state*. Almost all modern OS's use preemptive scheduling algorithms but it can result in race conditions when data are shared among several processes.

A non-preemptive kernel will wait for a system call to finish before making a context switch. Even though this ensures that the kernel structure is simple, this kernel-execution model is a poor choice for supporting a real-time computing, where tasks must complete execution within a given time frame. Most modern OSs nowadays are fully preemptive when running in kernel mode.

3.6.1.4) Dispatcher

Dispatcher gives control of the CPU's core to the process selected by the CPU scheduler. This function involves the following:

- Switching context from one process to another.
- Switching to user mode.
- Jumping to the proper location in the user program to resume the program.

Because they are enacted after every context switch, dispatch must be fast. The time it takes for the dispatcher to stop one process and start another is called **dispatcher latency**.

3.6.2) Scheduling Criteria

Different scheduling algorithms have different properties and the choice of a particular algorithm may favor one class of processes over another. We have many criterias to compare CPU-scheduling algorithms. These criterias are:

1. **CPU Utilization**: Since the CPU is the most expensive part of our computer, we want to keep it as busy as possible.
2. **Throughput**: If CPU is executing process, then work is being done. Measure of work is the number of process completed per time unit is called **throughput**.
3. **Turnaround time**: The interval from the time of submission of a process to the time of completion is the turnaround time.
4. **Waiting time**: Amount of time that a process spends waiting in the ready queue.
5. **Response time**: Time from the submission of a request until the first response is produced.

For interactive systems (such as PC desktop or laptop system) it is more important to minimize the variance in the response time than to minimize the average response time.

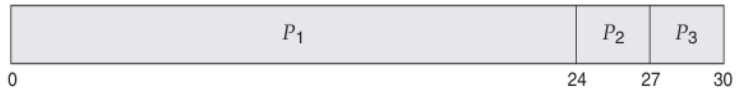
3.6.3) Scheduling Algorithms

3.6.3.1) First-Come, First-Served Scheduling

Implementing FCFS policy is easily managed with a FIFO queue. When a process enters to ready queue, its PCB is linked to the tail of the queue. When CPU is free, it is allocated to the process at the head of the queue.

Even though implementation is easy, the average waiting time is long. Take the example below and its **Gannt chart**:

Process	Burst Time
P_1	24
P_2	3
P_3	3



If the execution is as follows the average waiting time is: (0 + 24 + 27) / 3 = 17. However, if the processes arrive at in P2, P3, P1 order the Gannt chart would be:

And the average waiting time would be: (0 + 3 + 6) / 3 = 3. As you can see the average waiting time in FCFS scheduling is not minimal and may vary.

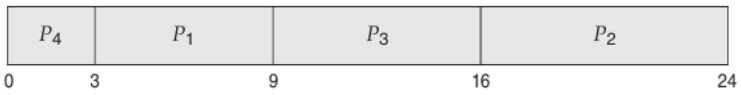
If other processes wait for one big process to be done to get the CPU, this situation is known as the **convoy effect**. This effect results in lower CPU and device utilization. Remember that the **First-Come, First-Served (FCFS)** scheduling algorithm is non-preemptive, meaning once a process gets hold of the CPU, it doesn't release it until it is finished. This can lead to the convoy effect, making FCFS problematic for interactive systems.

3.6.3.2) Shortest-Job-First Scheduling

In this approach we look at the CPU burst time. The smallest one gets the CPU. If two processes have the same CPU burst time, FCFS is used to break the tie. Actually, a better term for this algorithm would be *shortest-next-CPU-burst* algorithm. Since we are looking at the length of the CPU burst.

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



Here the average waiting time is: (0 + 3 + 9 + 16) / 4 = 7. If we were to use FCFS average waiting time would be 10.25 milliseconds.

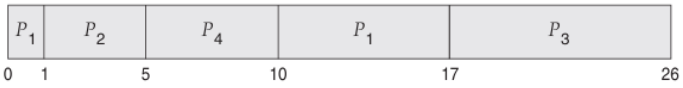
Even though this algorithm is optimal there is no way of knowing the length of the next CPU burst. To solve this problem, we can approximate the SJF scheduling. We generally use **exponential average** to approximate.

The SJF can either be preemptive or non-preemptive. Preemptive SJF is sometimes called **shortest-remaining-time-first** scheduling.

As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



Average waiting time: [(0-0) + (1-1) + (5 - 3) + (10 - 1) + (17 – 2)] / 4 = 6.5 milliseconds. Non-preemptive would result in 7.75 milliseconds.

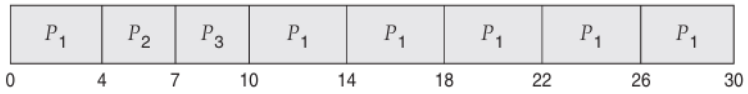
3.6.3.3) Round-Robin Scheduling

Round-robin is similar to FCFS but a preemption is added to enable the system to switch between processes. A small unit of time, called a **time quantum** or **time slice**. A time quantum is generally within 10-100 milliseconds.

The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for one time quantum.

Process	Burst Time
P_1	24
P_2	3
P_3	3

Let's look at the example above. Assume that they all have reached at time zero and the quantum time is 4 milliseconds. Its Gannt chart will look like the following.



Waiting Time (P1) = (0 - 0) + (10 - 4) = 6

Waiting Time (P2) = (4 - 0) = 4

Waiting Time (P3) = (7 - 0) = 7

Average = (6 + 4 + 7) / 3 = 5,66

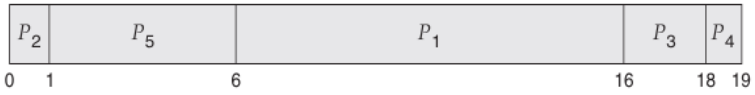
In the Round-Robin scheduling no process is allocated the CPU more than 1 time quantum. (If it is not the only runnable process). Performance of RR depends on time quantum. If the time quantum is too large it will basically be FCFS. If it is too low it will have large number of context switches. Please notice that higher context switches also slow the overall execution.

3.6.3.4) Priority Scheduling

In priority scheduling process with the highest priority gets the CPU.

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Let's take the example above on our hands. Its Gannt chart will look like below.



Priority scheduling can either be preemptive or non- preemptive.

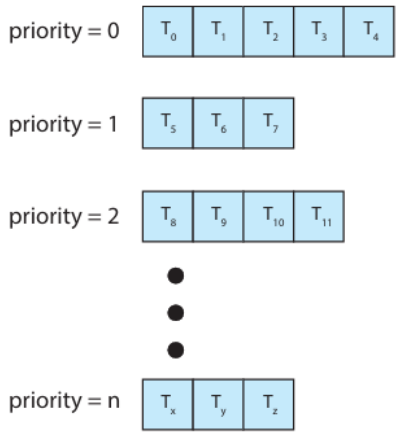
A major problem with priority scheduling is **indefinite blocking** or **starvation**. A process that is ready to run but waits CPU is called blocked. Priority scheduling can leave low-priority

processes blocked indefinitely. Because a system can be heavily loaded with a steady stream of higher-priority processes. A solution to this problem is **aging**. This method gradually increases the priority of processes as they wait in the system for a long time.

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

3.6.3.5) Multilevel Queue Scheduling

- Processes are divided into multiple queues based on some characteristics such as priority, process type, or resource requirements.
- Each queue may have its own scheduling algorithm (e.g., FCFS, RR)
- Queues themselves are scheduled with a priority-based approach, where higher priority queues are given preference over lower priority queues.



A multilevel queue scheduling algorithm can be used to partition processes into several separate queues based on the process type.

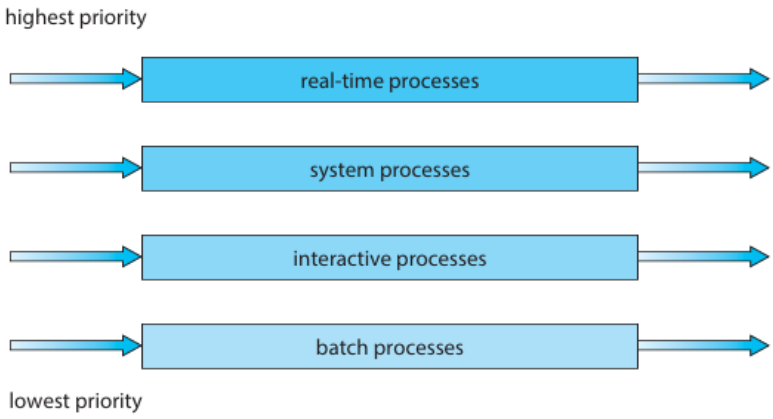


Figure 5.8 Multilevel queue scheduling.

Moreover, there can be scheduling among the queues. Which is commonly implemented as fixed-priority preemptive scheduling.

1. Real-Time processes
2. System processes
3. Interactive processes
4. Batch processes

Each queue has priority over lower-priority queues. For example, no batch queue processes can run before real-time, system and interactive were all empty.

3.6.4) Thread Scheduling

To run a user-level threads must ultimately be mapped to an associated kernel-level thread. But this mapping might be indirect and it might use a lightweight process (LWP).

3.6.4.1) Contention Scope

Contention Scope distinguishes between how user-level and kernel-level threads are scheduled. Here's a detailed explanation with concise notes

PCS (Process Contention Scope):

- User-level threads scheduled by the thread library onto LWPs.
- Competition for CPU among threads of the same process.
- Execution requires OS to schedule LWP's kernel thread on a CPU.

SCS (System Contention Scope):

- Kernel schedules kernel-level threads onto CPUs.
- Competition for CPU among all system threads.
- Used by one-to-one model systems (e.g., Windows, Linux).

Priority in PCS:

- Threads are scheduled based on priority.
- Programmers set thread priorities; some libraries allow priority changes.

- PCS typically preempts for higher-priority threads.
- No guaranteed time slicing for equal-priority threads.

Multi-Processor Scheduling

If multiple CPUs are available, **load sharing** is possible. Traditionally **multiprocessor** was the term to describe the systems that provided multiple physical processors. However, nowadays they are used for multiple CPUs, multithreaded cores, NUMA systems, Heterogeneous multiprocessing.

4.0) PROCESS SYNCHRONIZATION

4.1) Synchronization Tools

4.1.1) Race Conditions

In some operating systems, processes can share common storage that each of them can read and write. (It might be in main memory or in the shared file this doesn't change the problems that arise.) To see the problem let's consider the following problem. A print spooler.

When a process wishes to print a file, it enters the file name in a special **spooler directory**. The **printer daemon** periodically checks if there are anything to print, and if yes it prints and removes the names from directory. Imagine spooler has a very large number of slots. Imagine there are 2 shared variables out and in. Out points to the next file to be printed and in points to the next free slot. Assume the slots 0-3 are empty as they are already printed. Slots 4-6 are full as the figure below.

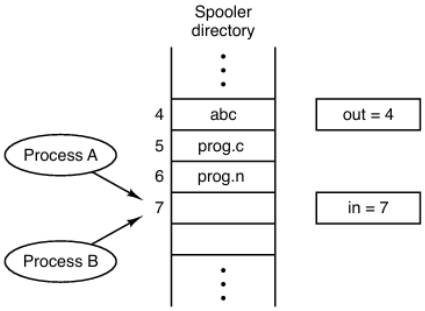


Figure 2-21. Two processes want to access shared memory at the same time.

Now assume the following scenario. Process A reads the "in" and stores the value 7, then clock interrupts the and process A stops. Process B also reads the "in" and stores the value

7. At this moment both are thinking the available free slot is 7. Process B continues and stores the name of its file in slot 7 and updates the “in” to be 8. Eventually process A runs again. It read the value 7 so it erases what the process B put to the slot 7. Situations like these where 2 or more processes are reading or writing in some shared data and the final result depends on who run are called **race conditions**.

4.1.2) The Critical-Section Problem

It aims to design a protocol that the processes can use to synchronize their activity so as to cooperatively share data. Each process must get permission to enter its critical section. Section of code to implement this request is **entry section**. Critical section is followed by an **exit section**. The remaining code is the **remainder section**.

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

A solution to critical-section must satisfy the following requirements:

1. **Mutual Exclusion**: If any process is in the critical section no other process can be executing in it.
2. **Progress**: If no process is in the critical section and some processes wishes to enter it, then only the processes then only the processes that aren’t executing in their remainder sections can participate in deciding which will enter the critical section next.
3. **Bounded Waiting**:

The critical-section problem can be solved easily in a single-core environment with preventing the occurrence of interrupts. We can make instructions run in order without any preemption. But because of the time consumption, inefficiency and the impracticality of the disabling interrupts, this solution is not feasible in the multi-core processors. **Non-preemptive kernels** do not suffer from race conditions as the processes are not stopping until finish.

Mutual exclusion is achieved through:

1. No two processes are inside their critical regions simultaneously.
2. We can’t make any assumption about speeds or the number of CPUs
3. No process running outside the critical section may block any process
4. No process should wait forever to enter the critical section.

4.1.2.1) Mutual Exclusion with Busy Waiting

4.1.2.1.1) Disabling Interrupts

On a single process system, the simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it. With interrupts disabled, no clock interrupts can occur.

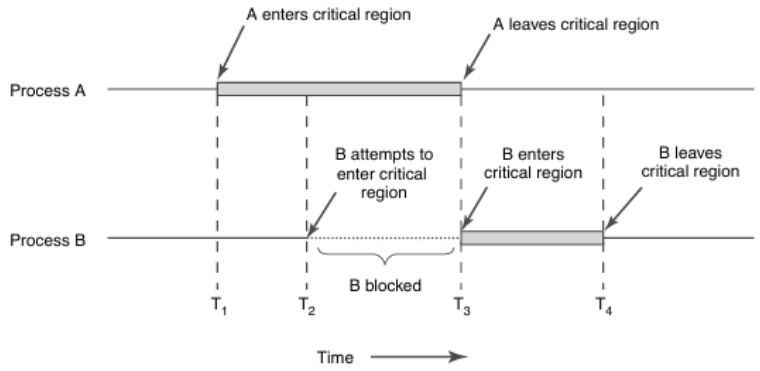


Figure 2-22. Mutual exclusion using critical regions.

The CPU is only switches from one process to another as a result of interrupt. If the interrupts are disabled the processes can examine and update the shared memory without any fear that any other process will intervene. This approach is generally deemed unattractive because it is unwise to give user processes the power to turn off interrupts.

Moreover, if the system is multiprocessor, disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue and can access to the shared area.

4.1.2.1.2) Lock Variables

When a process wants to enter its critical region, it first tests the lock. Lock is initially 0. If the lock is still zero it sets the lock to 1 and enters the critical region.

4.1.2.1.3) Strict Alternation

```
while (TRUE) {
    while (turn != 0) { }      /* loop */
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1) { }      /* loop */
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Figure 2-23. A proposed solution to the critical-region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

In the while(turn) part of the code the code does **busy waiting**. It should be avoided as it wastes the CPU time. A lock that uses bust waiting is called a **spin lock**.

4.1.2.1.4) Peterson's Solution

Peterson's solution

```
                        turn = 0;
                        flag[0] = false;
                        flag[1] = false;
P0                                          P1
while(true){                                while(true){
  flag[0] = true;                             flag[1] = true;
  turn =1;                                    turn = 0;
  while (flag[1] and turn ==1) do no-op;        while (flag[0] and turn ==0) do no-op;
    CS                                          CS
  flag[0] = false;                            flag[1] = false;
  remainder section;                          remainder section;
}                                           }
```

Peterson's solution only enforces mutual exclusion when both processes want to enter the critical section simultaneously. In the strict alternation the faster processes can be blocked unnecessarily to enter to the critical section.

4.1.2.1.5) The TSL Instruction

This approach requires help from the hardware. Computers who are designed with the multiprocessors in mind generally has an instruction like:

TSL RX, LOCK

It can be said that it is like a non-preemptive strict alternation on hardware. It is atomic (once it is started it can't be stopped).

Initially lock variable is set to 0, indicating that the resource is available. If a thread/process wishes to enter a critical section it should first sets the lock to 1 with the TSL instruction. If the lock isn't 0 it should retry or wait. After the execution of critical section is over, thread/process resets the lock variable to 0.

```
enter_region:
    TSL REGISTER,LOCK       | copy lock to register and set lock to 1
    CMP REGISTER,#0         | was lock zero?
    JNE enter_region        | if it was not zero, lock was set, so loop
    RET                     | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0            | store a 0 in lock
    RET                     | return to caller
```

Figure 2-25. Entering and leaving a critical region using the TSL instruction.

4.1.2.1.6) Sleep and Wakeup

All the solutions so far require busy waiting. And busy waiting wastes the CPU time. Moreover, busy-waiting can cause **priority inversion problem**. What it means is this, assume that H is high priority and L is low priority. H is supposed to run whenever it is ready. But if the L is in the critical region H has to busy-wait.

With sleep and wakeup signals we can avoid wasting CPU time. Sleep is a system call that causes the caller to block (it suspends until another process wakes it up)

4.1.2.1.6.1) The Producer-Consumer Problem

AKA **bounded-buffer** problem. Two processes share a common fixed size buffer. Producer puts information and customer takes it out. If the producer tries to put information to a buffer that is already full then producer goes to sleep and to be awakened when the

customer has removed one or more items. Similarly, the customer goes to sleep if no information is left and woken up when new information is added.

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

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


void consumer(void)
{
     int item;

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

Figure 2-27. The producer-consumer problem with a fatal race condition.

From time-to-time sleep or wakeup signals can be lost, this can prevent buffer issues. Moreover, there is a phenomenon called **spurious wake-ups**. A thread waiting on a condition can be wake up without being explicitly signaled for several reasons such as hardware interrupts, race conditions or issues within the operating system scheduler.

4.1.2.1.7) Mutexes

Mutex is short for mutual exclusion. We use mutex locks to protect critical sections and preventing race conditions. The acquire() function acquires the lock and release() function releases the lock.

```
while (true) {

     acquire lock

          critical section

     release lock

          remainder section

}
```

The definition of `acquire()` is as follows:

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;
}
```

The definition of `release()` is as follows:

```
release() {
   available = true;
}
```

The mutex lock you see above is called **spinlock**. Because the process spins while waiting for the lock (busy-waiting). In reality the **mutex_lock** is used to enter the critical section and **mutex_unlock** to leave it.

Thread call	Description
pthread_mutex_init	Create a mutex
pthread_mutex_destroy	Destroy an existing mutex
pthread_mutex_lock	Acquire a lock or block
pthread_mutex_trylock	Acquire a lock or fail
pthread_mutex_unlock	Release a lock

Figure 2-31. Some of the Pthreads' calls relating to mutexes.

Instead of busy waiting, mutexes can use **thread_yield** to give up the CPU to another thread when the mutex is busy.

Mutexes might use busy waiting if the expected wait time is very short. More sophisticated mutex implementations avoid busy waiting by blocking the thread. The thread is put to sleep and moved to a waiting queue, to be awaken when the mutex is released. This is called **blocking mutexes**. In Linux systems, futexes are used for blocking mutexes.

Please note that the key difference between blocking mutexes and the sleep and wakeup solution is that in the blocking mutexes, the waiting threads are waken up when the mutex becomes available and in the sleep and wakeup solution it requires a wakeup signal which can be prone to timing issues such as lost wakeups.

4.1.2.1.8) Semaphores

Edward Dijkstra suggested using integer variable to count the number of wakeups saved for future use. He proposed a new variable type he called **semaphore**. Dijkstra proposed having two operations on semaphores called down and up. Down operations checks whether the value is greater than 0. If yes, it decrements the value and moves on. Else the process is put to sleep. Before completing the down for the moment. Checking the value, changing it and possibly going to sleep is done in a single indivisible **atomic action**.

How the semaphores work?

- Basic Operation
  .1. Wait (P or down)
    - Decrements the semaphore value
    - If the semaphore value is greater than zero, the operation is successfully completed.
    - Else the process is put to sleep until the value is greater than zero

.2. Signal (V or Up)
    - Increments the semaphore value.
    - If there exists, processes sleeping one of them are woken up to proceed.

- Atomicity:
    .1. Semaphore operations are atomic. They are executed in single, indivisible action. This is crucial to prevent other processes from modifying the semaphore during these operations.

Types Of Semaphores

1. Binary Semaphores:
    - Can only take values 0 or 1.
    - They are similar to mutex locks. They use mutual exclusion to ensure only one process accesses a critical section at a time.
2. Counting Semaphores:
    - Can take non-negative integer values.
    - Used to control access to a resource that has multiple instances, such as pool of connections.

Semaphores avoids busy waiting, unlike spinlocks, semaphores put processes to sleep when they can't proceed. Semaphores effectively handle synchronization issues like mutual exclusion, producer-consumer problems, and reader-writer problems.

Binary Semaphore Example:

```c
sem_t binary_sem;

void* thread_function(void* arg) {
    sem_wait(&binary_sem); // Acquire the semaphore
    printf("Thread %ld entering critical section.\n", (long)arg);
    sleep(1); // Simulate some work in the critical section
    printf("Thread %ld leaving critical section.\n", (long)arg);
    sem_post(&binary_sem); // Release the semaphore
    return NULL;
}

int main() {
    pthread_t threads[5];

    sem_init(&binary_sem, 0, 1); // Initialize the semaphore with a value of 1

    for (long i = 0; i < 5; i++) {
        pthread_create(&threads[i], NULL, thread_function, (void*)i);
    }

    for (int i = 0; i < 5; i++) {
        pthread_join(threads[i], NULL);
    }

    sem_destroy(&binary_sem); // Destroy the semaphore
    return 0;
}
```

Counting Semaphores

```c
#define RESOURCE_COUNT 3

sem_t counting_sem;

void* thread_function(void* arg) {
    sem_wait(&counting_sem); // Acquire the semaphore (decrement the count)
    printf("Thread %ld acquired a resource.\n", (long)arg);
    sleep(1); // Simulate some work with the resource
    printf("Thread %ld releasing a resource.\n", (long)arg);
    sem_post(&counting_sem); // Release the semaphore (increment the count)
    return NULL;
}

int main() {
    pthread_t threads[5];

    sem_init(&counting_sem, 0, RESOURCE_COUNT); // Initialize the semaphore

    for (long i = 0; i < 5; i++) {
        pthread_create(&threads[i], NULL, thread_function, (void*)i);
    }

    for (int i = 0; i < 5; i++) {
        pthread_join(threads[i], NULL);
    }

    sem_destroy(&counting_sem); // Destroy the semaphore
    return 0;
}
```

Unnamed semaphores are generally used within the related process. They are used for threads whereas **named semaphores** are used for unrelated processes.

4.1.2.1.9) Monitors

Monitors are high-level synchronization constructs that simplify the design and implementation of concurrent programs. They provide mechanism to achieve mutual exclusion and coordinate the execution of multiple threads, making sure that only one thread can access the shared resource at any given time.

Key Concepts of Monitors

1. Mutual Exclusion:
    - When a process enters a monitor, other processes trying to enter are blocked until the first one leaves the monitor.
2. Condition Variables:
    - Condition variables are used within the monitors to manage the synchronization of processes. They allow processes to wait for certain conditions to be true and to signal other processes when those conditions change.
    - Primary operations are ‘wait’ and ‘signal’
3. Wait and Signal Operations:
    - Wait: A process that executes wait operation on a condition variable is suspended until another process signals that condition.

- Signal: A signal operation resumes one of the processes (if any) that was suspended on the condition variable. If no processes are suspended, the signal operation has no effect

How monitors work?

They encapsulate shared variables and procedures that operate on these variables. The monitor guarantees that procedures won't interfere with each other by ensuring that only one process can be active within the monitor at any time.

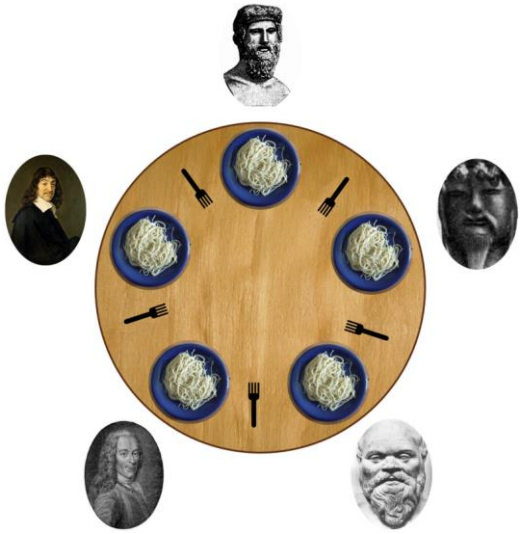
Monitors have several advantages such as simplicity, encapsulation and compiler support. Yet they are language dependent and have limited flexibility and they are originally designed for single-CPU.

4.2) DEADLOCKS

We will refer to the objects granted as **resources**. Resource can be a hardware device, piece of information etc.

A resource can be preemptable or non-preemptable. A **preemptable** resource is one that can be taken away from the process owning it and a **non-preemptable** means you can't take the resource away once it is assigned. Whether a resource is preemptable depends on the context. Memory for example is preemptable however low-end devices that don't have swapping or paging are non-preemptable.

4.2.1) The Dining Philosophers Problem



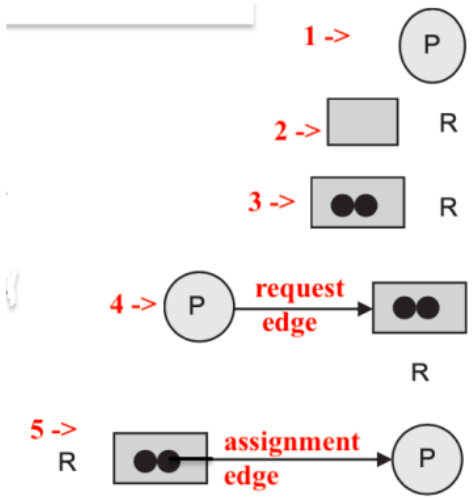
There are 5 philosophers and 5 forks. To eat the special spaghetti a person should hold one fork. All of the philosophers start to take one of the forks. Assume that it starts with Descartes above. All the philosophers are taking one fork in order at a time. Each philosopher got a fork but all of them are waiting for another fork and since none of them start eating (it requires two forks) the process can't proceed. This situation is known as the dining philosopher's problem. The situation they are ended up in is called **deadlock**. More official definition of deadlock is: *A set of process is deadlocked if each process in the set is*

waiting for an event that only another process in the set can cause. And the lack of resources left are called **starvation**.

There are 4 conditions that must be held for deadlock to occur:

1. Mutual Exclusion: Only once process can use the resource at a certain moment
2. Hold and Wait: All processes are holding the resources and waiting for other resources
3. No preemption: Once a resource is held by a process, we can't take it out.
4. Circular Wait: In a set of processes, each process holds a resource needed by the next one.

4.2.2) Deadlock Modelling



We can model a deadlock with graphs.

In the figure above:

1. Circular Node **->** Process
2. Rectangular Node **->** Resources
3. Dots in the rectangular node **->** Each dot is a resource instance
4. An edge from process to resource **->** A request edge
5. An edge from resource dot to a process **->** Assignment edge

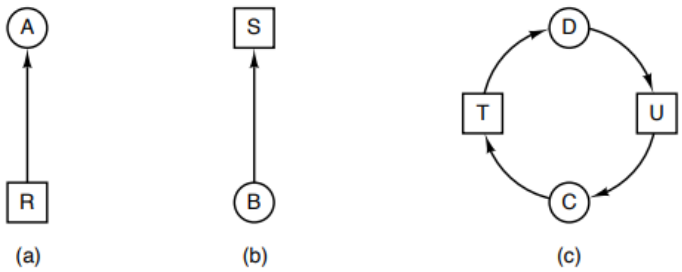


Figure 6-6. Resource -allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

- **Question 3: Draw the RAG of P1→ R1, P2→ R3, R2→ P1, R1→P3, P4→R3, R1→P4 ?**

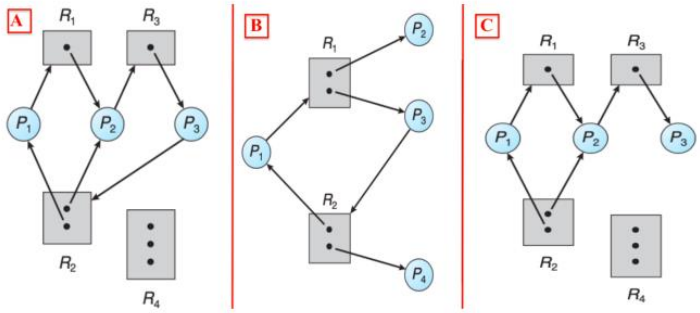


Figure 3: Question 1- RAG: A) with deadlock, B) Cycle but no deadlock, C) normal

In general, 4 strategies are used to deal with deadlocks

1. Just ignore the problem.
2. Detection and recovery. After they occur detect them, and take action to resolve.
3. Dynamic avoidance by careful resource allocation.
4. Prevention, by structurally negating one of the four conditions.

4.2.3) Deadlock Detection & Recovery

The second technique doesn't do anything to stop deadlocks from occurring. But tries to detect and resolve after it occurred.

1. Process A holds R and wants S.
2. Process B holds nothing but wants T.
3. Process C holds nothing but wants S.
4. Process D holds U and wants S and T.
5. Process E holds T and wants V.
6. Process F holds W and wants S.
7. Process G holds V and wants U.

The graph for the conditions above is below:

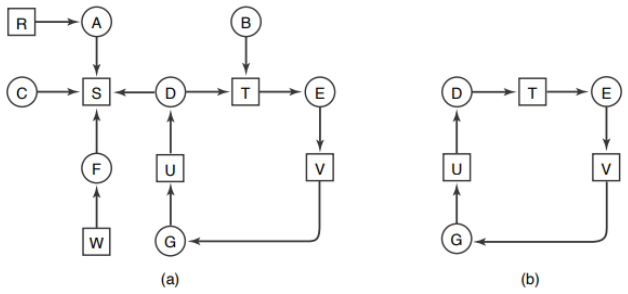


Figure 6-8. (a) A resource graph. (b) A cycle extracted from (a).

As it can be seen there is a clear deadlock. It is fairly easy to detect a deadlock visually. For detecting it algorithmically we generally use cycle detection algorithms.

4.2.4) Deadlock detection with multiple resources of each type

We need a different approach to detect deadlock if we have multiple copies of the same resource exists.

E is the **existing resource vector**. A is the **available resource vector**. And “m” is the number of resource classes, “n” is the number of processes. And now we need 2 matrices, R is the **request matrix** and C the **current allocation matrix**.

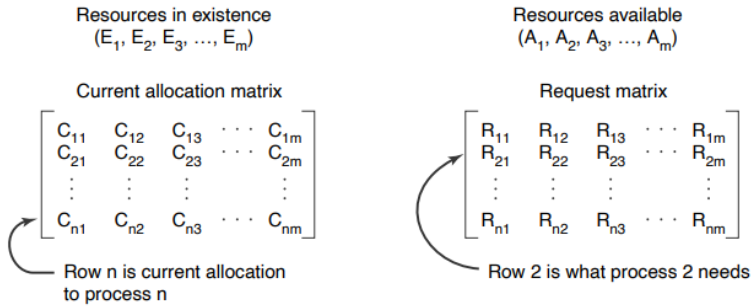


Figure 6-9. The four data structures needed by the deadlock detection algorithm.

HERE: If requested < available then do the below

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

And repeat the process over and over where I wrote HERE. If no process exists the program terminates.

4.2.5) Recovery From Deadlock

1. **Recovery through Preemption**: If possible, temporarily take a resource from its current owner and give it to another process.
2. **Recovery through Rollback**: If the system designers and machine operators know that deadlocks are likely, they can arrange to have processes **checkpointed**. Meaning its state is written to a file so it can be restarted later. For this to be effective new checkpoints shouldn’t overwrite but should write to new files. When a deadlock is detected, it is easy to see which resources are needed. To do the recovery, to do this recovery we roll back to the time they acquired that resource by starting at one of its earlier checkpoints.
3. **Recovery through Killing Processes**: Crudest yet simplest way to break a deadlock is to kill one or more processes. One possibility is to kill a process in the cycle. With this the other processes might continue but it isn’t guaranteed. This can continue until the cycle is broken.

4.2.6) Deadlock Avoidance

The banker’s algorithm is the most used algorithm when it comes to deadlock avoidance.

Process	Tape drives	Plotters	Printers	Cameras
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Printers	Cameras
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still assigned

E = (6342)
P = (5322)
A = (1020)

You have resources assigned and resources waiting for assignment. If he waiting is less than the available you should do basically what we did at the "**Deadlock detection with multiple resources of each type** "part.

4.2.7) Deadlock Prevention

1. **Attacking the Mutual-Exclusion Condition**: Mutual exclusion means at least one of the resources must be non-shareable. Shareable resources don't require mutual exclusion thus can't involve in a deadlock. Read-only files are a good example on this. More than one processes can reach a file if they are using read only instructions. However, this strategy is not applicable for all resources for some resources are intrinsically non-sharable.
2. **Attacking the Hold and Wait**: If we prevent processes that hold resources from waiting for more resources, we can eliminate deadlocks. One way of achieving this is requiring all processes to request all their resources before execution but the thing is they may not know about the resources they need before execution. Another way to break hold and wait condition is to require a process requesting a resource to first temporarily release all the resources it currently holds and then try to get everything at once.
3. **Attacking the No-Preemption Condition**: Some resources can be virtualized to avoid non-preemption. Spooling printer output to the SSD or disk and allowing only the printer daemon access to real printer eliminates deadlocks involving printer.
4. **Attacking the Circular Wait Condition**: Look at the figure below. One way to break circular wait is to force processes to request resources in numerical order. For example, in the figure below a process might first request printer and then the tape driver (because 2 < 4) but it can't request first plotter then printer (because 3 > 2) With this rule we make sure there is no cycles.

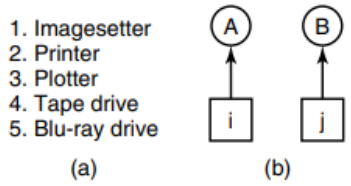


Figure 6-16. (a) Numerically ordered resources. (b) A resource graph.

Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

Figure 6-17. Summary of approaches to deadlock prevention.

Check after 493 / 1185

5.0) MAIN MEMORY

5.1) Background

Main memory and the registers built into each processing core are the only general-purpose storage the CPU can directly access. There are machine instructions that take memory addresses but none takes disk addresses. So, any instruction in execution must be in direct-access storage devices.

The CPUs can access registers in one cycle or *CPU clock*. Completing a memory access may take many cycles of the CPU clock. The processor normally needs to **stall**, since it doesn't have the data required to complete the instruction. The remedy is adding a fast memory between CPU and main memory, like **cache** memory.

We aren't only concerned with speed but also the validity. Operating system should be protected from the user processes and user processes should be protected from other user processes.

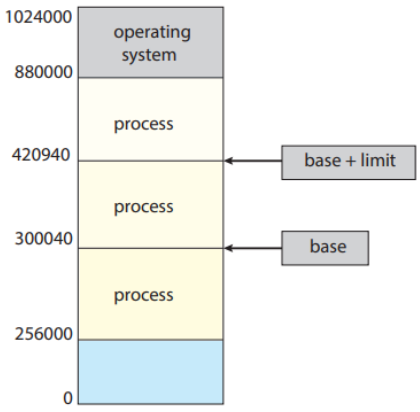


Figure 9.1 A base and a limit register define a logical address space.

Base register holds the smallest legal physical memory address and the **limit register** specifies the size of the range.

Protection of memory space is achieved by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating system memory ends up with trap to the operating system.

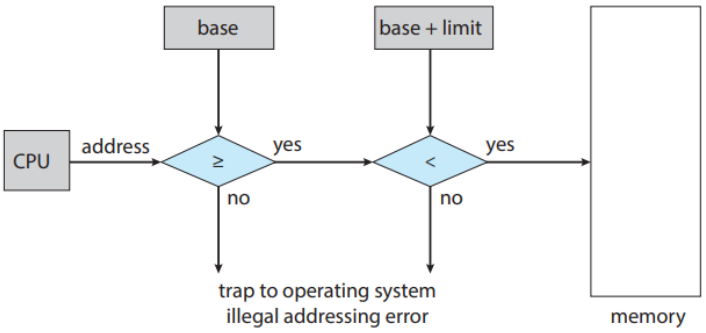


Figure 9.2 Hardware address protection with base and limit registers.

5.2) Address Binding

Programs run on a disk as a binary executable file. To run it we should first bring it to the memory and place within the context of a process where it becomes eligible for execution. After its execution other processes reclaim the memory.

Address binding is the process of mapping instructions and data to memory addresses. It occurs at different stages: Compile time, load time and execution time. Address binding basically takes the addresses in the secondary storage and maps them to main storage (RAM etc.).

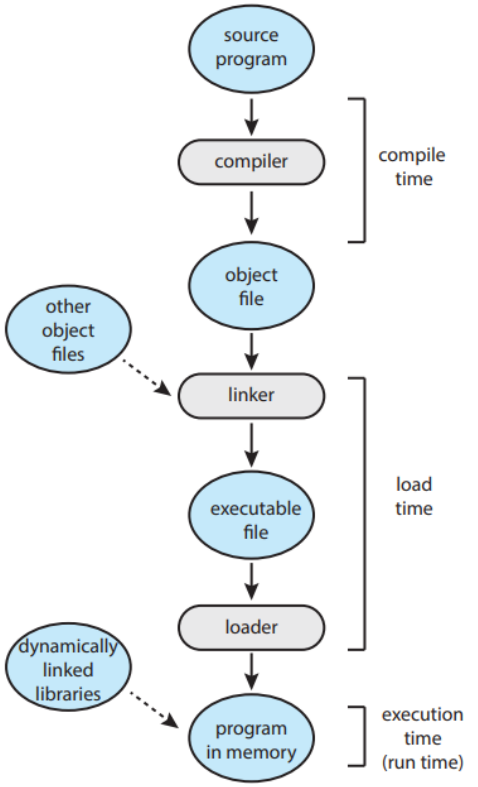


Figure 9.3 Multistep processing of a user program.

5.2.1) Stages of address binding

1. Compile Time
    - If you know at compile time where the process resides in the memory, then **absolute code** can be generated. For example, if a program is always loaded at address ‘R’, the compiler generated code starting at ‘R’. But if the starting location changes, recompilation is needed.
2. Load Time
    - If where the process resides in memory isn’t known at compile time, then the compiler must generate **relocatable code**.
3. Execution Time
    - If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

5.3) Logical Versus Physical Address Space

An address generated by the CPU during the execution time is referred as a **logical address**. Whereas an address seen by the memory unit, which is the one loaded into the **memory-address register** of the memory, is commonly referred as **physical address**.

Please note that logical address and virtual address is used interchangeably.

Logical address space = Set of all logical addresses generated by a program.

Physical address space = Set of all physical addresses corresponding to these logical addresses.

Memory Management Unit = It does run-time mapping from virtual to physical address.

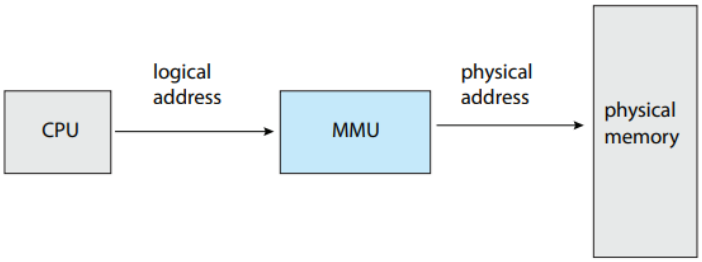


Figure 9.4 Memory management unit (MMU).

Let’s call the base register **relocation register** for now. Value in the relocation register is added to every address generated by a user process at the time the address is sent to memory. For example, if the base is at 14000 then an attempt by the user to address location 0 is relocated to location 14000. And access to 346 is relocated to 14346.

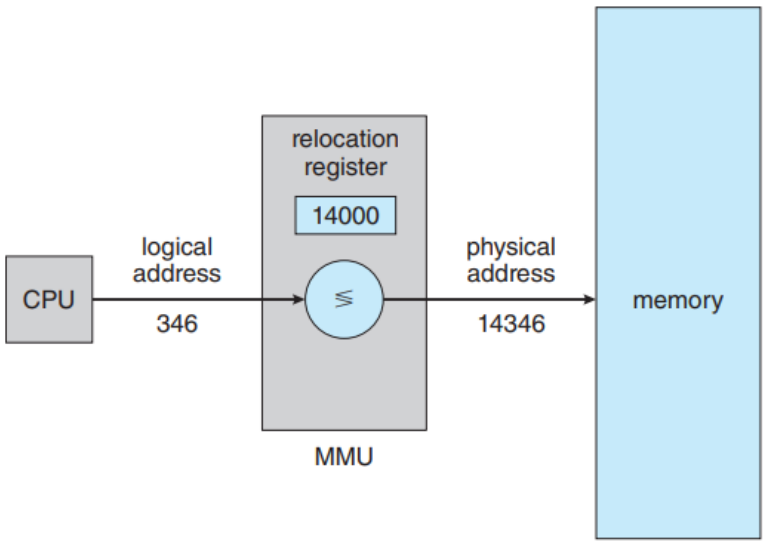


Figure 9.5 Dynamic relocation using a relocation register.

5.4) Dynamic Loading

The size of a process has been limited to the size of physical memory. We use **dynamic loading** to achieve a better memory-space utilization. In dynamic loading, a routine is not loaded until it is called. All of them are kept on disk in a relocatable load format. Main advantage of dynamic loading is that a routine is loaded only when it is needed. This method is useful when large amounts of code are needed to handle infrequent occurring cases such as error routines.

5.5) Dynamic linking and shared libraries

Dynamically linked libraries (**DLLs**) are system libraries that are linked to user programs when the programs are run. Some operating systems support only **static linking** (System libraries are treated like any other object module and are combined by the loader into the binary program image.)

Dynamic linking on the other hand is like dynamic loading but it is usually used for system libraries such as <stdlib.h> library. Without it, each program on a system must include a copy of its language library and this can waste main memory.

A second advantage of DLLs is that these libraries can be shared among multiple processes and because of this they are also known as **shared libraries** and extensively used in Windows and Linux systems.

When a program references a routine that is in the dynamic library, the loader locates the DLL, loads into memory if necessary. It then adjusts addresses that reference functions in the dynamic library to the location in memory where the DLL is stored.

Unlike dynamic loading, dynamic linking and shared libraries requires OS help.

If the processes in memory are protected from one another, then the operating system is the only entity that can check to see whether the needed routine is in another process's memory space or that can allow multiple processes to access the same memory addresses.

5.6) Contiguous Memory Allocation

This method was a one early memory allocation method. Memory is generally divided into 2 parts. One for operating systems and one for user processes. We can either put the operating system in low or high memory addresses. Decision depends on factors such as location of the interrupt vector etc. In many O.S.s operating system is out into high memory.

In the contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.

5.7) Memory protection

Aims to prevent processes from accessing the memory that they don't own. Doing so ensures the operating system and other processes are protected from unintended or malicious access.

Each logical address generated by the CPU is checked to ensure that it falls within the range specified by the limit register. The MMU then add the base address from the relocation register to the logical address to form the physical address. This mapped physical address is then used to access memory.

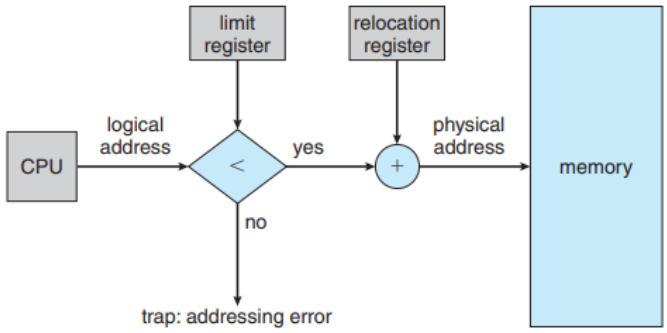


Figure 9.6 Hardware support for relocation and limit registers.

When the CPU scheduler selects a process for execution, the dispatcher loads the correct values into the relocation and limit registers. Every address generated by the process is checked against these registers. If an address is out of range, a trap is generated to trap the process.

This allows the operating system's size to change dynamically, which is useful for managing memory efficiently.

5.8) Memory Allocation

Operating system keeps a table indicating which parts of memory are available and which are not. This is called **variable partition** scheme. Initially all the memory is available for user processes and is considered one large block of available memory, called a **hole**. Eventually as processes are loaded and terminated, the available memory becomes fragmented into holes of various sizes.

If there is not enough memory space to satisfy the demands of an arriving process, we can do one of the 2 things

1. Reject the process: Provide an error message
2. Wait queue: Put it in a waiting queue until sufficient memory becomes available

The procedure to allocate memory is a particular instance of the general **dynamic storage allocation problem**. It concerns how to satisfy a memory allocation request from a list of free holes. Some of the most used solutions for this problem are **first-fit**, **best-fit** and **worst-fit**.

1. **First Fit**: Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes. The searching stops as soon as we find a hole large enough.
2. **Best Fit**: Allocate the smallest hole that is big enough. This algorithm searches the entire memory.
3. **Worst Fit**: Allocate the largest hole. This algorithm searches the entire memory.

5.9) Fragmentation

When processes are loaded and removed from the memory, the free memory is broken into little pieces. **External fragmentation** is when there is enough total memory space to satisfy a request but the available spaces aren't contiguous. Storage is fragmented into a large number of small holes.

Statistical analysis on first-fit algorithms shows that about half of the total memory blocks (external fragmentation) are lost to fragmentation. This is called **50-percent rule**.

If the fragmentation is unused memory that is internal to a partition it is called **internal fragmentation**.

One solution for external fragmentation is **compaction**. We shuffle the memory contents so as to place all free memory together in one large block.

5.10) Paging

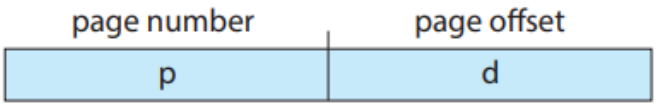
Paging is a memory management scheme that permits a process's physical address space to be non-contiguous. Main benefit of *paging is that it avoids external fragmentation* and the associated need for compaction.

5.10.1) Basic Method

We basically breaking physical memory into fixed-sized blocks called **frames** and we break the logical memory into blocks of the same size called **pages**.

When a process is to be executed, its pages are loaded into any available memory frames from their source. The backing store is divided into fixed-sized blocks that are the *same size* as the memory frames.

Every address generated by the CPU is divided into two parts: a **page number** (**p**) and a **page offset** (**d**)



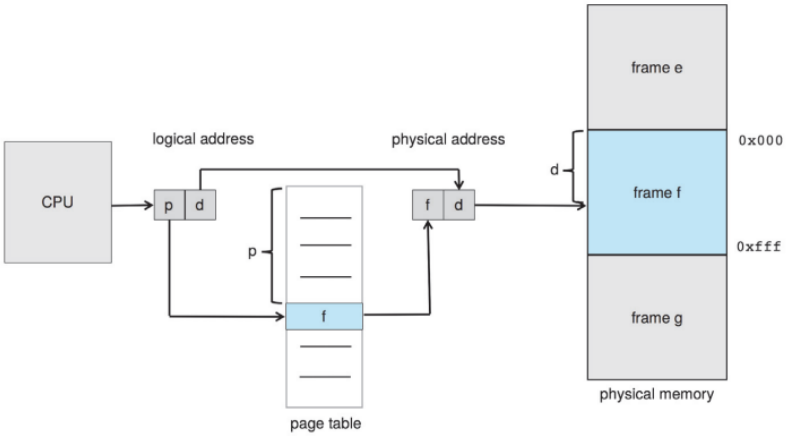


Figure 9.8 Paging hardware.

In the figure above you can see how the paging works. The MMU does the following to translate the logical address generated by the CPU to a physical address:

1. Extract the page number p and use it as an index in the page table.
2. Extract the frame number from the corresponding page number.
3. Replace the page number “p” in the logical address with the frame number “f “.

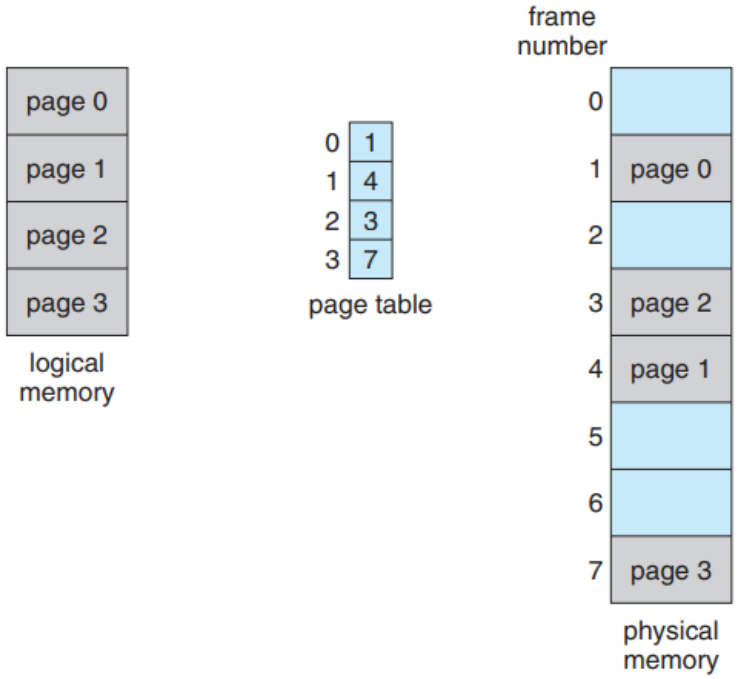


Figure 9.9 Paging model of logical and physical memory.

For page 0, we should go to 0 in the page table. And it gives us the frame number 1. Go to physical memory 1 and here the page 0. And so on.

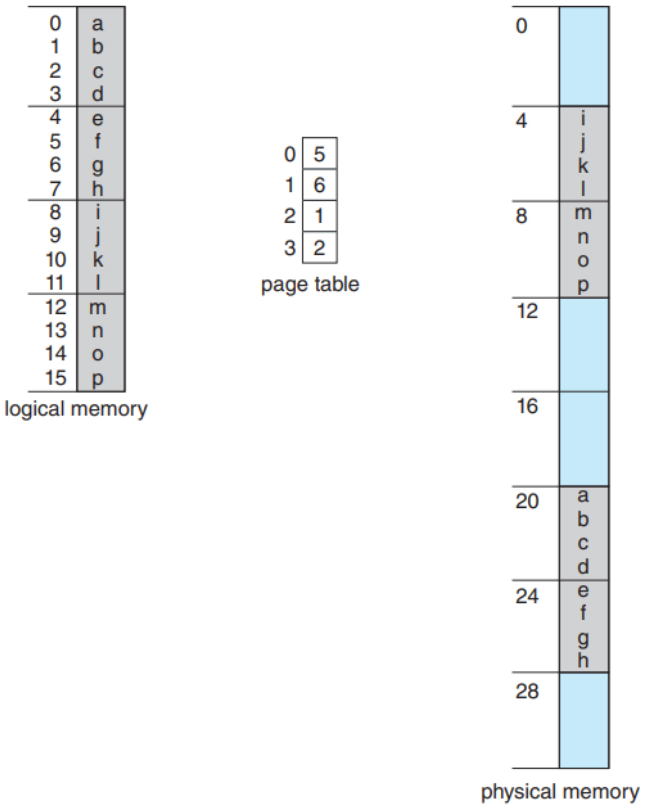


Figure 9.10 Paging example for a 32-byte memory with 4-byte pages

If process size is independent of page size, we expect internal fragmentation to average one-half page per process. This consideration suggests that small page sizes are desirable. With the growth of average process or data sets the pages size increased and today they are either 4KB or 8KB in size.

To obtain page size on Linux system we can use the system call *getpagesize()*. Or we can enter *genconf PAGESIZE* command.

When a process arrives in a system to be executed, its size, expressed in pages is examined. Each page of the process needs one frame. So, if the process needs n pages, at least n frames should be available in the memory. If so, the process is loaded into one of the allocated frames.

Paging separates the programmers view of the memory and the actual physical memory. The programmer views the memory as a one single space only containing this one program. But in reality, the user program is scattered throughout physical memory

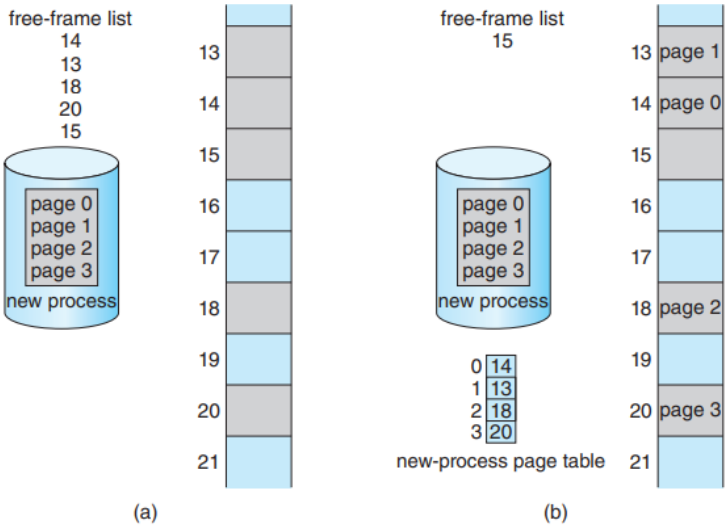


Figure 9.11 Free frames (a) before allocation and (b) after allocation.

Operating system is managing physical memory and therefore it should be aware of the allocation details of the physical memory. Like which frames are allocated, which are available, how many total frames are there and so on. This information is generally kept in a single, system-wide data structure called a **frame table**. Frame table has one entry for each physical page frame, indicating its situation, free or allocated and if allocated to which page of which process

5.10.2) Hardware Support

Simplest way of doing page tables with the hardware implementation is implementing it as a set of dedicated high-speed hardware registers. Usage of registers for page table is satisfactory if the page table is reasonably small. But most O.S.s support much larger page tables. Making the use of fast registers infeasible. Rather the page table is kept in main memory and a **page-table base register** (**PTBR**) points to the page table.

5.10.3) Translation Look-Aside Buffer

Although storing the page in main memory can yield faster context switches, it might case slower memory access times.

The standard solution for this is to use a special, small, fast-lookup hardware cache called **translation look-aside buffer** (**TLB**)

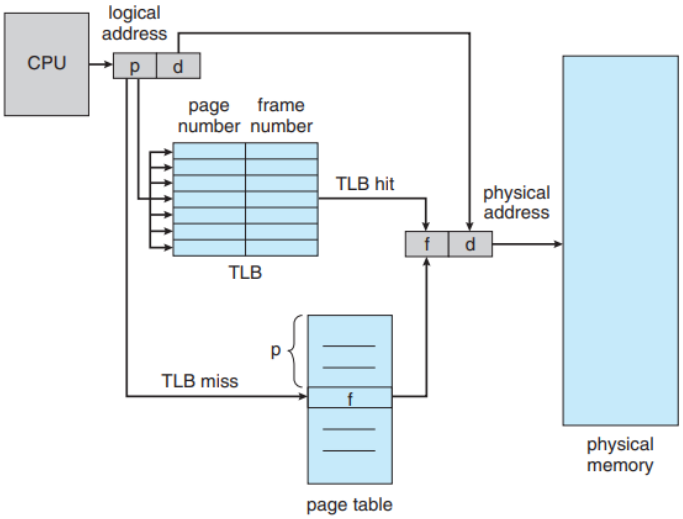


Figure 9.12 Paging hardware with TLB.

The CPU looks for the page address at the TLB first. TLB has 2 parts. One keeps the page number and one keeps the frame number. If the item is found (**TLB Hit**) the correct physical address is created and found in the physical memory. If the page number couldn't be found in the TLB (**TLB Miss**) address translation proceeds following the step.

5.10.4) Protection

Memory protection in a paged environment is accomplished by protection bit associated with each frame. One bit can define a page to be read-write or read-only.

One additional bit is generally attached to each entry in the page table. A **valid-invalid** bit. When this bit is set to *valid*, the associated page is in the process's logical address space and is thus legal.

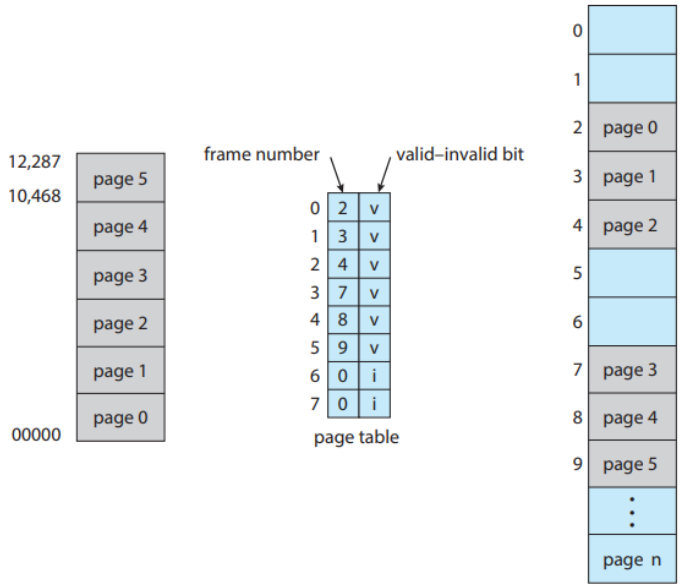


Figure 9.13 Valid (v) or invalid (i) bit in a page table.

If the valid-invalid bit is set to invalid, the computer will trap to the O.S. (invalid page reference)

5.10.5) Shared Pages

An advantage of paging is *sharing* common code. One of the most used libraries are stdlib.h or libc. Assume that libc is 2 MB and system has 40 user processes using libc. So, instead of loading libc 40 times, thus using 80 MB, the system can share it.

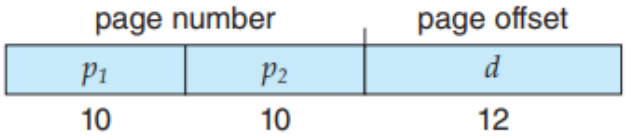
5.11) Structure of the Page Table

5.11.1) Hierarchical Paging

With large logical address spaces (2^32 or 2^64) the page table can become very large. Hierarchical paging is used to manage large page tables. This technique involves paging the page table to itself and dividing it into smaller, more manageable pieces.

Two level paging: We divide the logical address into multiple parts. Outer page table, inner page table and page offset.

In a 32-bit address example we can do the two-level paging as follows:



P1 is the outer page number

P2 is the inner page number

D is the page offset.

Because this address translation works from the outer page table to inward, it is known as a **forward-mapped** page table.

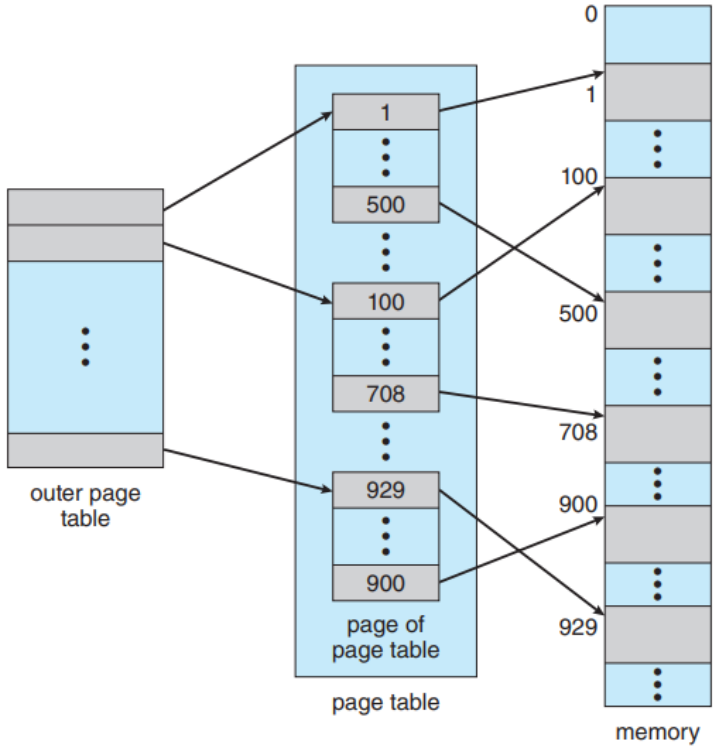
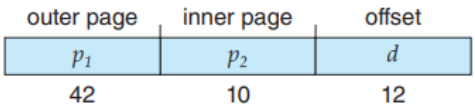


Figure 9.15 A two-level page-table scheme.



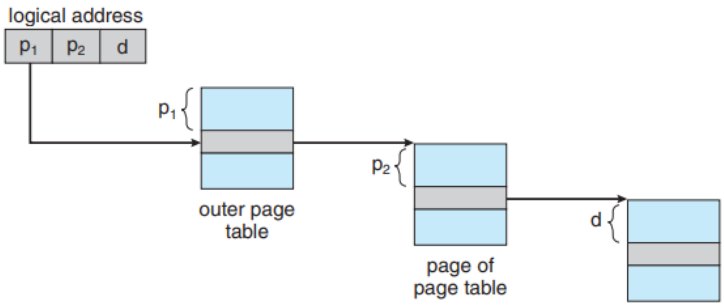


Figure 9.16 Address translation for a two-level 32-bit paging architecture.

5.11.2) Hashed page tables

One approach for handling address spaces larger than 32 bits is to use a **hashed page table**. With the hash value being the virtual page number.

This algorithm works as follows:

Virtual page number in the virtual address is hashed into the hash table. Virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding frame is used, else subsequent entries in the linked list are searched for a matching virtual page number.

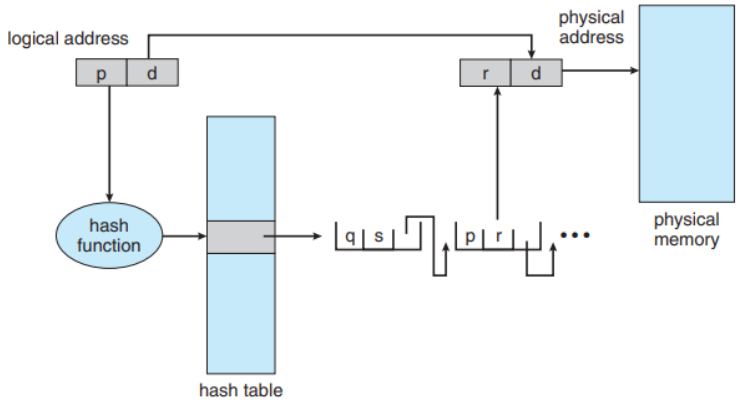


Figure 9.17 Hashed page table.

5.11.3) Swapping

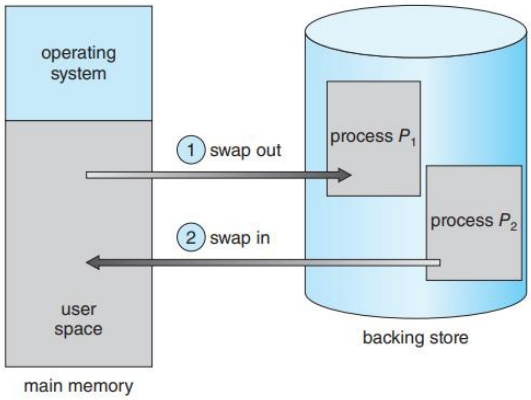


Figure 9.19 Standard swapping of two processes using a disk as a backing store.

A process or a portion of it can be temporarily **swapped** out of the memory to a **backing store** and then brought back into memory for continued execution. Swapping allows the total physical address space of all processes to exceed the real physical memory.

5.11.3.1) Standard Swapping

Standard swapping is just moving the entire process between main and backing store. The backing storage used is generally fast secondary storage.

When a process or part of it is swapped to the backing storage, the data structures associated with the process must be written to the backing store so that it can later be swapped back.

Advantage of the standard swapping is that it allows physical memory to be oversubscribed, so that the system can accommodate more processes than actual physical memory.

5.11.3.2) Swapping with Paging

Standard swapping generally is not used in modern OSs. Most modern systems use swapping with paging rather standard swapping.

In fact, *swapping* generally refers to standard swapping and *paging* refers to swapping with paging. A **page out** operation moves a page from memory to the backing store. Reverse is known as **page in**.

Just read the pages between 484- 490.

6.0) VIRTUAL MEMORY

6.1) Background

Even though it is logical and necessary to have the entire program in the memory. But a closer look shows us that entire program is not needed.

For instance, programs often have a code to handle error cases which rarely occurs. Arrays, list, tables are often allocated more memory that necessary. In these conditions they busy the memory unnecessarily.

But even in cases where entire program is needed not all of the program might be needed at the same time. If we can execute a program that is not in memory fully, we can have these benefits:

1. Physical memory constraint would be over as users would be able to write programs for an extremely large *virtual* addresses.
2. Because each program does take less space, more programs can be loaded.
3. Less I/O would be needed to load or swap portions of programs into memory, so each program would run faster

So, a running program not being entirely in the memory would benefit both system and user.

Virtual memory involves separation of logical memory as perceived by developers from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.

The **virtual address space** of a process refers to the logical view of how a process is store in memory.

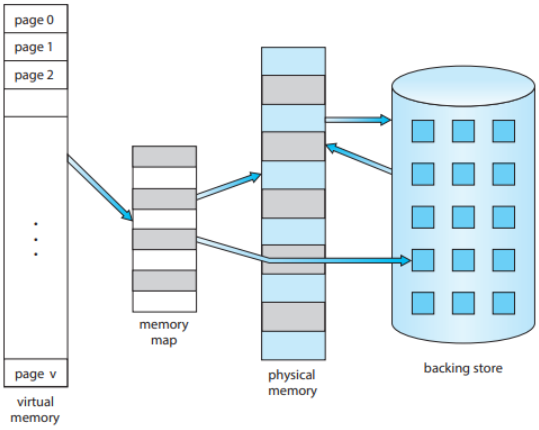


Figure 10.1 Diagram showing virtual memory that is larger than physical memory.

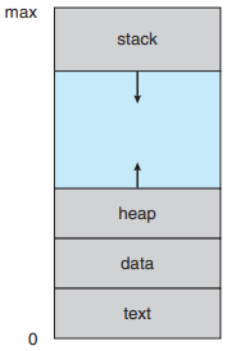


Figure 10.2 Virtual address space of a process in memory.

The large blank space between the heap and stack is part of the virtual address space. But it requires physical pages only if the heap and stack grow. Virtual address space has holes, known as **sparse** address spaces.

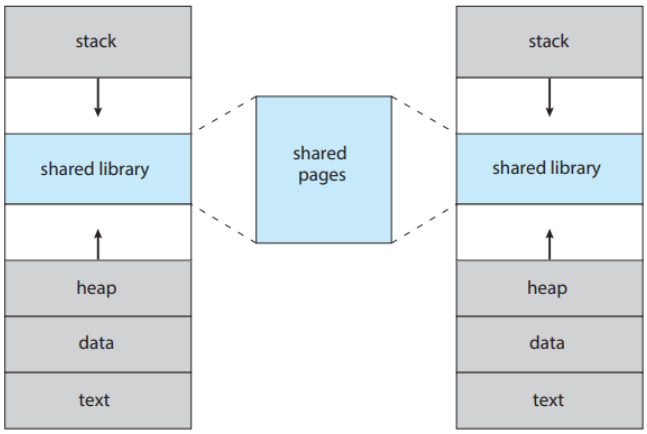


Figure 10.3 Shared library using virtual memory.

Virtual memory allows files and memory to be shared by two or more processes through page sharing. Many system libraries can be shared by several processes. This has several advantages:

- Even if different processes consider it their own virtual address space, the actual pages that reside on physical memory are shared by all.

- Moreover, processes can share memory (remember the race condition notes). Virtual memory allows a process to create a region of memory that it can share.
- Pages can be shared during the process creation with fork() syscall, speeding the process creation.

6.2) Demand Paging

We may not need to have entire process in the memory as it might not be necessary. A solution for this is **demand paging** and is commonly used in virtual memory systems. The general concept behind demand paging is *loading the* pages *only when they are needed*.

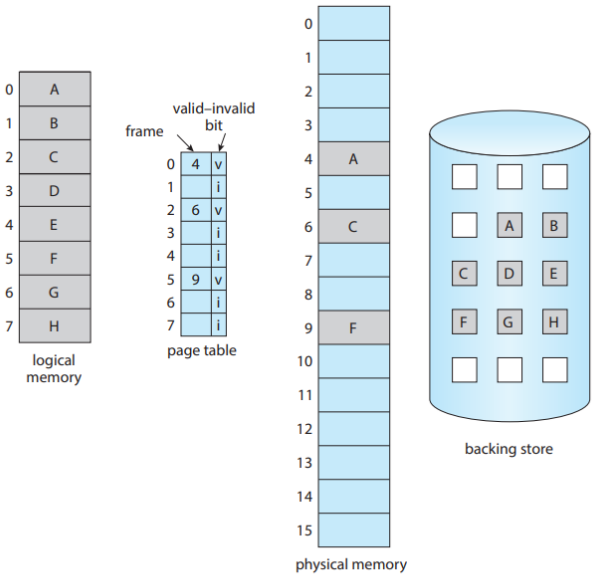


Figure 10.4 Page table when some pages are not in main memory.

As a result, as the process is executing some of the pages will be in secondary storage. And here comes the valid-invalid bit. If the bit is set to “valid” the memory is both legal and in the memory. If the bit is “invalid” it is either not in the logical address space or valid but in the secondary storage.

If process tries to access a page that wasn’t brought into the memory is called **page fault**. The paging hardware will notice that the invalid bit is set and trap to the operating system. This trap is the result of the operating system’s failure to bring the desired page into memory. The procedure is as follows:

1. Check an internal table (Generally kept with the PCB) for this process to determine whether the reference was a valid or an invalid memory access. If the reference is valid or not.
2. If the reference is invalid (AKA it doesn’t exist) we terminate the process. If it is valid but not brought in that page, we now page it in.
3. We find a free frame.
4. We schedule a secondary storage operation to read the desired page into the newly allocated frame.
5. When the storage read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in the territory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page as if it was always in the memory.

The process can be seen in the figure below.

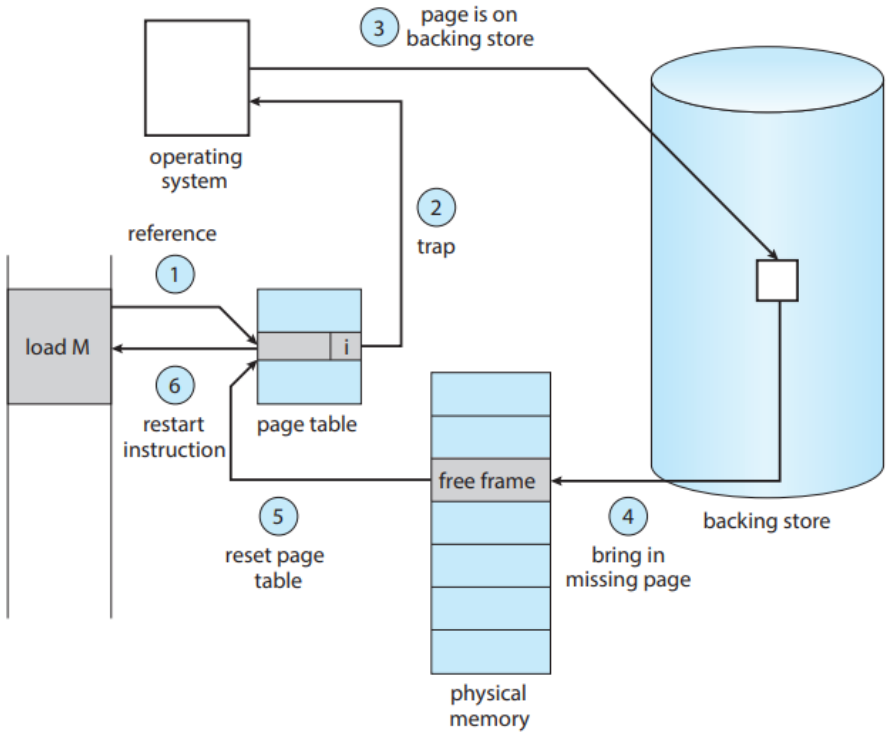


Figure 10.5 Steps in handling a page fault.

Pure demand paging is one extreme case. We start executing process with NO pages in the memory and we bring page into memory only when it is required.

The crucial requirement for demand paging is the ability to restart any instruction after a page fault. Because we save the state of the interrupted process when the page fault occurs, we have to restart exactly the same place and state but this time the requested page is in the memory.

6.2.1) Free-Frame List

Many of the O.S.s maintain a **free-frame list**. This is a pool of free frame lists for satisfying such requests. O.S.s allocate free frames using a technique known as **zero-fill-on-demand**.

Figure 10.6 List of free frames.

When a system starts up, all available memory is placed on the free-frame list and as the frames are requested, the size of it shrinks. At some point list become NULL and then we repopulate it which'll be discussed later.

6.3) Copy-on-Write

Traditionally the fork() system call copies the parent's address space for the child. This duplicates the pages belonging to the parent. Instead, we can use **copy-on-writing**. It allows the parent and child process to share the same pages initially. Before process1 makes any changes, the pages are this:

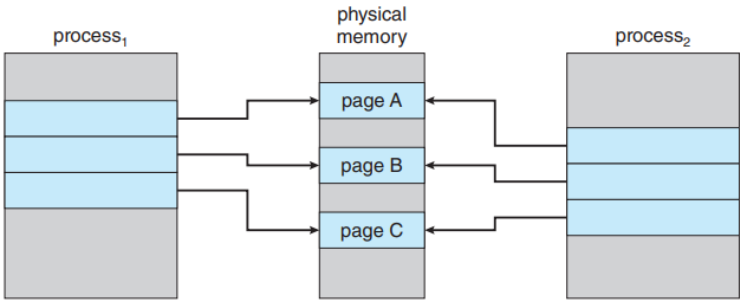


Figure 10.7 Before process 1 modifies page C.

Now assume that child process attempts to modify a page (let's say page C). The OS will obtain a frame from the free-frame list and create a copy of the page. Mapping it to the child's address space. After that the child process *modifies the page copied* not the original one. Please note that only the pages that are modified are copied un-modified pages are shared between parent and child. You can see the modification below:

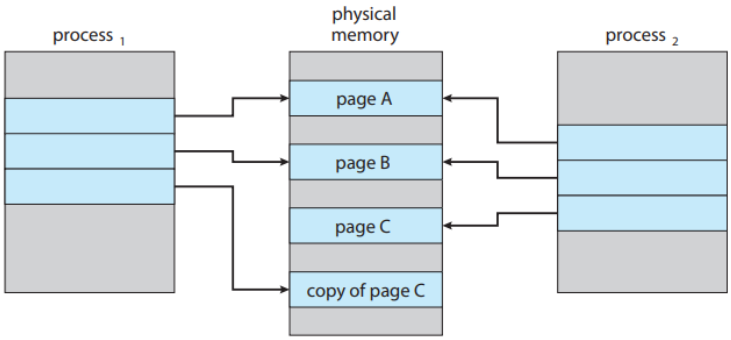


Figure 10.8 After process 1 modifies page C.

6.4) Page Replacement

The increase of multiprogramming comes with the **over-allocating** of the memory. If we are to run 6 processes and all of them have ten pages but actually only using five of them, we will have higher CPU utilization.

Assume that the page fault occurred. OS first determines where the desired page is resides on the secondary storage. However, if there are no frames on the free-frame list (all memory in use) operating system has several options at this point.

1. It could terminate the process: Not the best option
2. It could use standard swapping
3. **Page replacement**

6.4.1) Basic Page Replacement

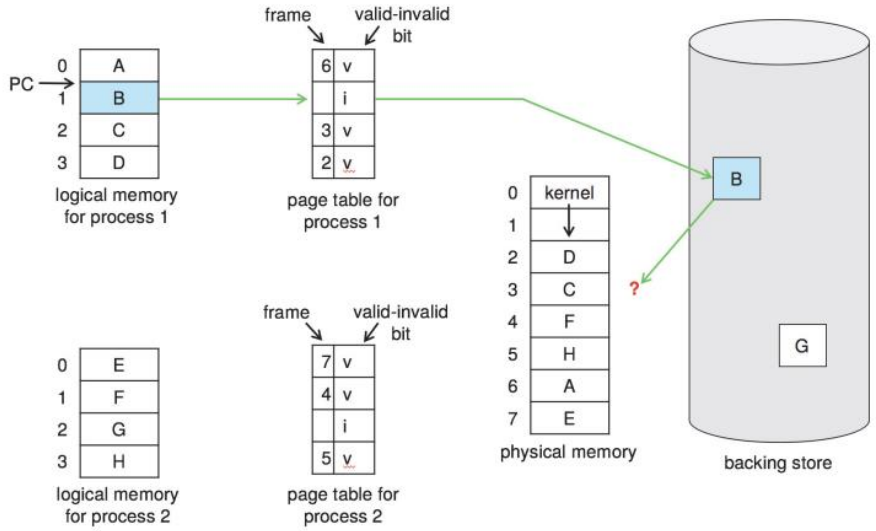


Figure 10.9 Need for page replacement.

Page replacement works as follows: If no frame is free, we find the one that is not currently used and free that. We can free a frame by writing its contents to swap space and changing the page table to indicate that the page is no longer in memory.

1. Find the location of the desired page in the secondary storage.
2. Find the free frame.
    a. If there is a free frame use it
    b. If there is no free frame, use a page replacement algorithm to select a **victim frame**.
    c. Write the victim frame to secondary storage, change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame, change the page and frame tables.
4. Continue the process from where the page fault occurred.

If no frames are free, two pages transfers are required. This situation doubles the page-fault service time and increases the access time. This situation can be reduced by using a **modifying bit** (also called **dirty bit**). When this schema is used, each page or frame has a modify bit. When this schema is used, each page or frame has a modifying bit associated with it in the hardware. It is set whenever any bit for a page is written into, indicating it has been modified.

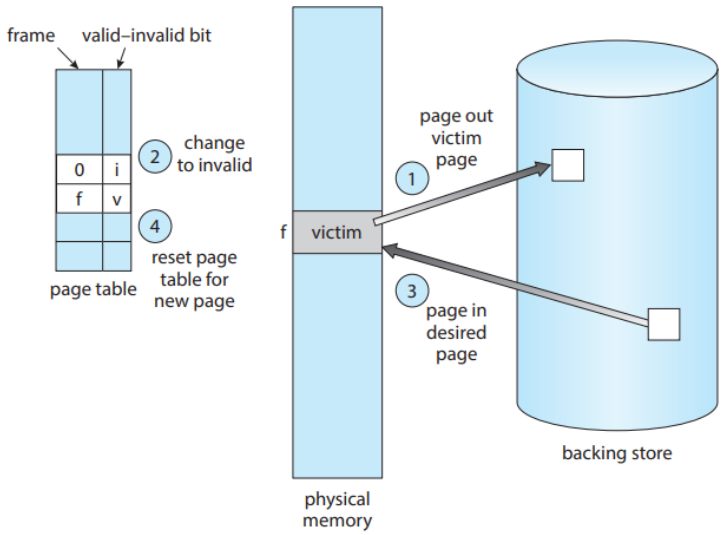


Figure 10.10 Page replacement.

With demand paging the size of the logical address space is no longer constrained by physical memory. If a process has 20 pages, we can execute it in 10 frames simply by using demand paging and using replacement algorithm to find a free frame when necessary.

But we have to solve 2 problems. We must develop a **frame-allocation algorithm** and a **page-replacement algorithm**. If we have multiple processes in memory, we must decide how many frames to allocate to each process. And when a page replacement is needed, we must select the frames that are to be replaced.

We evaluate an algorithm by running it on a particular string of memory references and compute the number of page faults. The string of memory references is called a **reference string**.

6.4.2) FIFO Page Replacement

Simplest page-replacement algorithm is first-in, first-out (FIFO). A FIFO replacement algorithm associates with each page the time when that page was brought into memory. *When a page must be replaced, the oldest page is chosen*. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page brought into memory, we insert it at the tail of the queue.

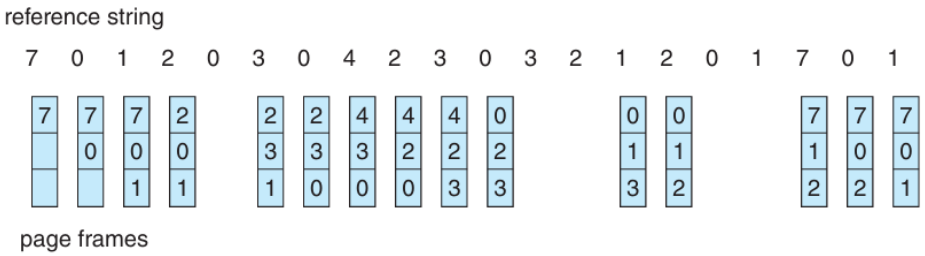


Figure 10.12 FIFO page-replacement algorithm.

When the 2 came at step 4, the memory was full. And there is no “2” in the memory so we removed the oldest frame from the memory which is 7 and then put 2.

Afterwards, when 0 came, 0 was in the memory so no need to put it.

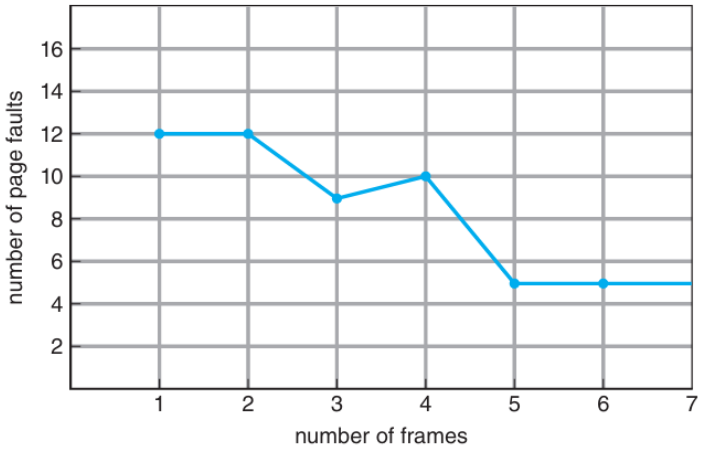


Figure 10.13 Page-fault curve for FIFO replacement on a reference string.

As you can see above when the frame size is 4, the number of page faults increased. This is called **belady's anomaly**.

6.4.3) Optimal Page Replacement

When the belady's algorithm is discovered, we started the search for **optimal page-replacement algorithm**. The algorithm that causes the lowest page-fault rate for all algorithms.

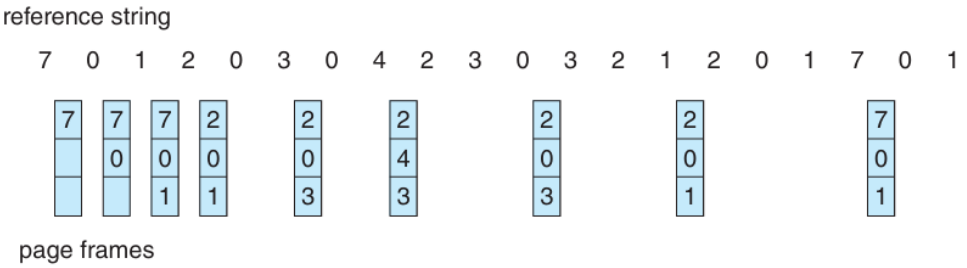


Figure 10.14 Optimal page-replacement algorithm.

(What we do is basically, if there exists empty frame, put the page. Else we take a note of the frames and we look at the ***right*** side, find the farthest away frame and change that)

Unfortunately, it is hard to implement as you need to know the future knowledge of the reference string.

6.4.4) LRU Page Replacement

(What we do is basically, if there exists empty frame, put the page. Else we take a note of the frames and we look at the ***left*** side, find the farthest away frame and change that)

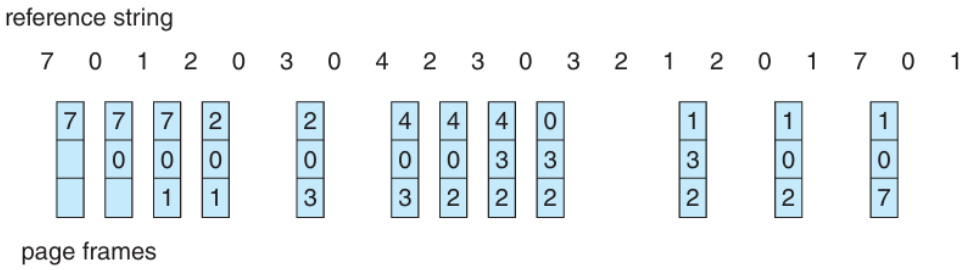


Figure 10.15 LRU page-replacement algorithm.

The LRU is often used and considered a good algorithm. The major problem is *how* to implement. An LRU page-replacement algorithm might require a hardware solution. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

1. **Counters**: Each page-table entry has a time-of-use field. A logical clock is incremented every memory reference. When a page is referenced, the clock value is copied to the time-of-use field. Page with the smallest time value will be replaced but it should consider overflow of the clock.
2. **Stack**: A stack of page numbers is maintained. When a page is referenced, it is removed from stack and placed on top. With this we ensure the least recently used is always at the bottom. Best implemented using a doubly linked list with head and tail pointers. It is more suitable for software or microcode implementations.

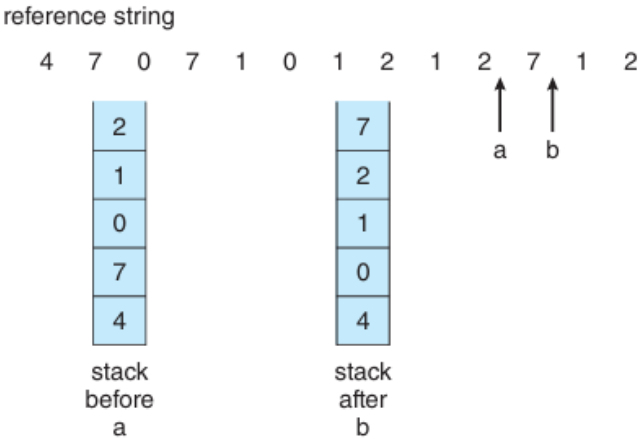


Figure 10.16 Use of a stack to record the most recent page references.

The LRU replacement doesn't suffer from belady's anomaly.

6.4.5) LRU- Approximation Page Replacement

Not many computer systems provide sufficient hardware support for LRU page replacement. If a system doesn't provide a hardware support, then other page algorithms must be used. But many systems provide help in the form of **reference bit**. It is set by the hardware whenever page is referenced and with it, we can determine which pages are referenced.

6.4.6) Additional-Reference-Bits Algorithm

Uses an 8-bit byte for each page. A timer interrupt shifts the reference bit into the high order bit of the byte at regular intervals. This creates a history of the page usage over the last eight periods. Pages are recent uses (lower numbers) are replaced first. If the bits are cleared over time, this algorithm degenerates into **second-chance page-replacement algorithm**.

6.4.7) Second-Chance Algorithm

It is basically FIFO replacement algorithm. checks the reference bit of the selected page. If the bit is 0, the page is replaced and the page gets a second chance.

One way to implement second-chance algorithm (also called as clock algorithm) is as circular queue. A pointer indicates which page to be replaced next. Pointer advances until finding a page with an “0” reference bit. As it advances it clears the bits.

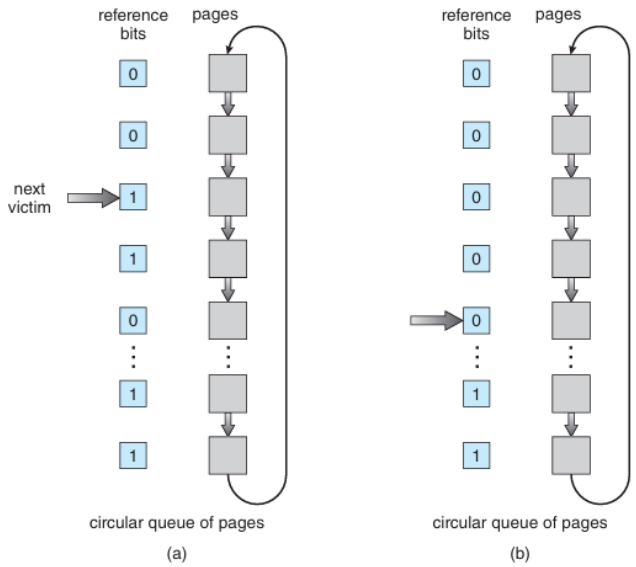


Figure 10.17 Second-chance (clock) page-replacement algorithm.

6.4.8) Enhanced Second-Chance Algorithm

Considers both reference and modify bit. Pages are categorized into 4 classes:

1. (0,0) Neither recently used nor modified. Best page to replace
2. (0,1) Not recently used but modified. Not best because page will need to written out before replacement
3. (1,0) Recently used but clean. Probably will be used again
4. (1,1) Recently used and modified. Probably will be used again soon.

6.4.9) Counting-Based Page Replacement

Uses a counter to track the number of references for each page.

1. **Least Frequently Used (LFU)** page-replacement algorithm requires that the page with the smallest count to be replaced. But a problem occurs if the page is used extensively during the initialization and not later one. One solution for this is to shift the counts by 1 bit at regular intervals.
2. **Most Frequently Used (MFU)** assumes that the page with the smallest count was probably just brought in and hasn't been used.

Neither of them is common.

6.4.10) Page-Buffering Algorithms

Maintains a pool of free frames. On a page fault, a free frame is allocated to the desired page before writing out the victim page. Keeps a list of modified pages to write them to secondary storage when idle. Can remember contents of frames to reuse them directly if needed again.

6.5) Allocation of Frames and Design Issues for Paging Systems

6.5.1) Minimum Number of Frames

Every process needs a minimum number of frames to execute efficiently. The number is constrained by the architecture and the need to hold all pages referenced by a single instruction. For example, intel 32 and 64 bits need at least 6 frames for specific instructions. Each process should receive minimum number of frames because otherwise page faults might increase.

6.5.2) Allocation Algorithms

Equal allocation divides available frames equally among processes. For example, with 93 frames and 5 processes, each process gets 18 frames.

Proportional allocation on the other hand allocates frames based on the size of each process's virtual memory. Ensuring more of a need base distribution.

Equal allocation might seem fair at first but it makes little to no sense to give equal frames to 10-KB and 300-KB process.

6.5.3) Local versus Global Allocation Policies

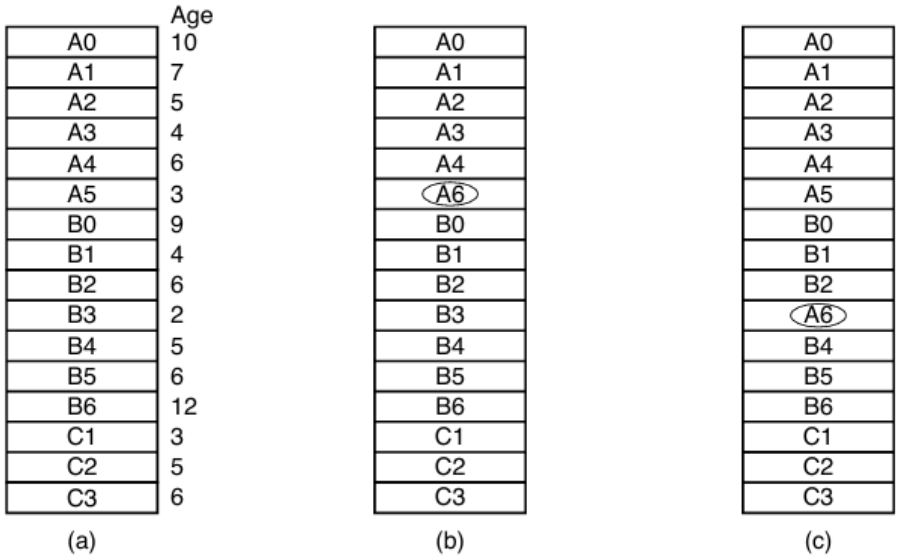


Figure 3-22. Local versus global page replacement. (a) Original configuration. (b) Local page replacement. (c) Global page replacement.

Take a look at figure (a). There are 3 processes. A, B and C make up the set of runnable processes. Suppose A gets a page fault and we use least-recently-used (LRU) page-replacement algorithms. If it *only looks at the pages of A*, the page with the lowest age value is A5. And the system changes that so the situation at (b) occurs. This is said to be **local** page replacement algorithm. Local algorithms effectively correspond to allocating every process a *fixed fraction* of the memory.

Global algorithms on the other hand, dynamically allocate the page frames among the runnable processes. Thus, the number of page frames assigned to each process varies in time.

In general, global algorithms work better. Especially if the working set size can vary a lot over the lifetime of a process. A program that causes page fault in every few instructions is called **thrashing**. If a local algorithm is used working set grows, thrashing will occur. Moreover, if the working set shrinks, local algorithms waste memory.

If global algorithm is used, it might be possible to use proportional allocation. But the allocation has to be updated dynamically. One way to manage this is using **PFF** (Page Fault Frequency). It tells when to increase or decrease the process' page allocation but doesn't say which page to replace.

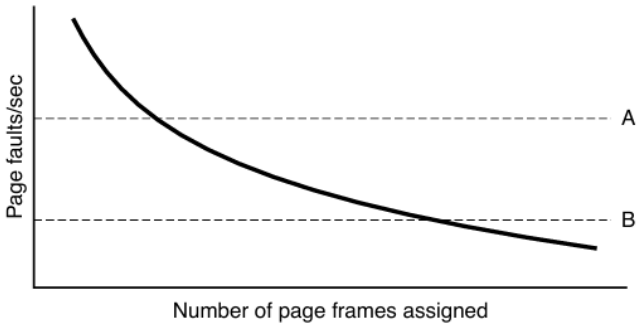


Figure 3-23. Page fault rate as a function of the number of page frames assigned.

Note: There are 2 types of page faults **minor** or **major** (In windows they are referred as **soft** or **hard**). Major fault is if we refer a page that isn't in the memory. Minor is if we don't have a logical mapping to an existing page. In Linux systems, the number of major faults is very low and minors are higher. Meaning that, they are taking advantage of shared libraries, as a library is loaded into memory, subsequent page faults are only minor faults.

6.5.4) Load Control

To implement a global page-replacement algorithm, we can implement a strategy that keeps the amount of free memory above a minimum threshold. When it drops below it, a kernel routine is triggered that begins reclaiming pages from all processes in the system. Such routines are known are **reapers**. In Linux, if the amount of free memory falls to *very* low levels, **out-of-memory** (**OOM**) **killer** routine gets into play and selects a process to terminate. Each process has an OOM score, a higher score means it is more likely to be killed by OOM killer. (OOM scores can be seen in the /proc file system) OOM scores are calculated by the amount of memory a process is using.

We can also swap the processes to secondary memory as discussed earlier

But an important thing to know is that there is another smart technique called **deduplication** or **same page merging**. The system checks the memory to see if two pages have the exact same content. If yes, instead of storing both physical frames, the operating system removes one of the duplicates and modifies the page table mappings. The frame is shared copy-on write. But the second we write to the page; a fresh copy is made.

6.5.5) Cleaning Policy

It is made by the **paging daemon**. Even though it sleeps for most of the time, it is periodically woken up to inspect the memory. If the available frames are too low it begins to select pages to evict using the replacement algorithms.

6.5.6) Non-Uniform Memory Access

On **non-uniform memory access** (**NUMA**) systems with multiple CPUs a CPU can access some sections of the main memory faster than it can access other parts. How CPUs and memory are interconnected causes these differences.

These systems are made up of multiple CPUs and each of them have their own local memory. CPUs can access its local memory faster than memory local to another CPU.

NUMA systems are always slower.

6.5.7) Shared Libraries

In modern operating systems, there are many large libraries used by many processes, for example, multiple I/O and graphics libraries. Statically binding all these libraries to every executable program on nonvolatile storage would make them bloated.

Instead, a technique called **shared libraries** (In windows they are called **DLL**s or **Dynamic Link Libraries**) are used.

But what is the idea behind shared libraries? In a traditional linking when a program is linked, one or more object files are names in the command to the linker, for example this UNIX command: “ld *.o -lc -lm “. Any functions called in the object fields but not present there are called **undefined externals**. They are looked after in the libraries and included in the executable binary if found. For example, for printf function to work in C, write system call is needed. If the write is not included in the library, linker will look for it and include it when found. When the linker is done, an executable file ONLY has the *necessary functions*. Remember that when the codes are compiled, they are written to secondary storage at first. So, if we didn’t have linking, we would waste huge space both in primary and in secondary storage.

6.6) Thrashing

What if a process has too little frames. Meaning what if the number of frames isn’t enough to support pages. The process page faults of course. We page-replace it but since the number of frames are too little it quickly faults again.

This high page fault and paging activity is called **thrashing**.

6.6.1) Cause of Thrashing

A global page replacement algorithm replaces the pages regardless of their process. Assume that after a while process needs more frames so it starts faulting and taking frames from other processes. But the processes whose frames are taken also needs those pages. So, they also start to fault. As a result, ready queue empties and CPU utilization decreases.

When the CPU scheduler observes the decrease in the CPU utilization it *increases* the degree of multiprogramming. And now the new processes request frames from the running processes causing more page fault and longer wait time. As a result, the CPU utilization drops further and CPU scheduler increases the degree of multiprogramming. No process is getting executed for all the processes are spending their time paging.

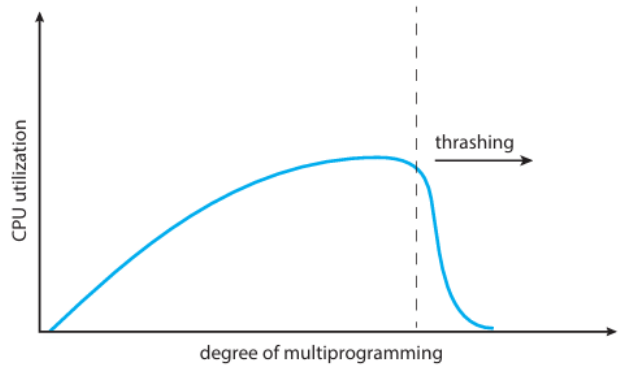


Figure 10.20 Thrashing.

As you can see above, as the degree of multiprogramming is increasing the CPU utilization increases up to a point but after a point it drops significantly.

One way of *limiting* the effects of thrashing is to *use local page replacement algorithms*. Because local page-replacement algorithms don't steal frame from other processes but uses their own. But this doesn't entirely solve the problem. Just slows it down.

6.7) Allocating Kernel Memory

Kernel memory is often allocated from a free-memory pool, which is different from the list used to satisfy user-mode processes. There are 2 main reasons for that

1. Kernel requests memory for data structures of varying sizes, some of which are less than a page in size. And as you can guess this causes fragmentation, so the kernel must use memory conservatively. This is especially important for many operating systems don't subject the kernel code or data to the paging system
2. Page allocated by the user-mode processes don't have to be in main memory contiguously but kernel-mode processes might need to reside in physically contiguous pages.

The "buddy system" and "slab allocation" are 2 strategies to allocate kernel memory.

6.7.1) Buddy System

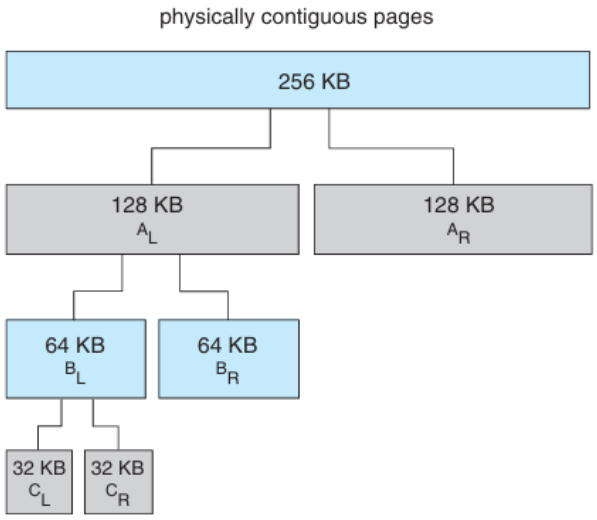


Figure 10.26 Buddy system allocation.

I think the figure above is clear enough

6.7.2) Slab Allocation

A slab is made up of one or more physically contiguous pages. A **cache** consists of one or more slabs. There is a single cache for the data structure representing process descriptors, a separate cache for file objects etc. Each cache is filled objects that are instances of the kernel data structure the cache represents.

The slab-allocation uses caches to store kernel objects. When a cache is created, a number of objects are allocated to the cache. Number of objects depends the size of the slab. When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache.

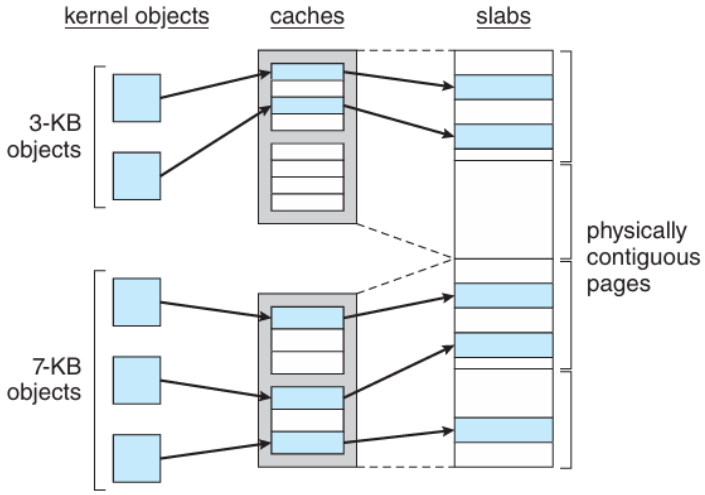


Figure 10.27 Slab allocation.

Slabs provide 2 main benefits:

1. No memory wasted due to fragmentation. It is not an issue for each unique kernel data structure has an associated cache and each cache is made up of one or more slabs that are divided into chunks the size of the objects being represented. Thus, when the kerel requests memory for an object, the slab allocator returns the exact amount of memory required to represent the object.
2. Memory requests can be satisfied more rapidly. Because the objects are created in advance.

6.8) Operating-System Examples on How Linux and Windows Manages Virtual Memory

6.8.1) Linux

Linux holds 2 types of page lists to manage memory. An *active_list* and *inactive_list*. The active list indicates the pages that are in use. Each page has an *accessed* page that is set if page is referred. Periodically the accessed bits for pages in the active_list are reset and over time the least recently used page will be at the front of the active list. For illustration, look at the figure below.

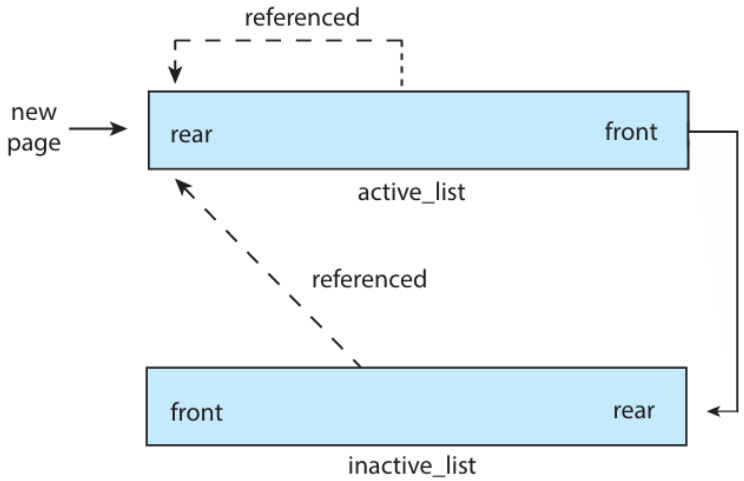


Figure 10.29 The Linux `active_list` and `inactive_list` structures.

Each of the lists are kept in balance. If active grows a lot, front of it is moved to the inactive_list.

Linux kernel has a page-out daemon process *kswapd* that periodically awakens and checks the amount of free memory in the system. If free memory falls below a certain threshold this daemon scans the inactive_list and reclaims them for the free list.

6.8.2) Windows

In 32-bit systems, default virtual address space is 2GB but it can be extended to 3 GB and it can support 4GB of physical memory.

In 64-bit systems, Windows 10 has 128TB virtual address space and it can support 24TB of physical memory

Windows 10 implements virtual memory using *demand paging with clustering*. A strategy that recognizes the locality of memory references and therefore handles page faults by bringing not only the faulting page but also pages immediately preceding and following it.

A key component of memory management in Windows 10 is working set management. Each process is allocated a set of minimum 50 and maximum of 345 pages. These are called **working-set minimum** and **working-set maximum**.

7.0) FILE SYSTEMS

7.1) File System Interface

7.1.1) File Concept

To define a logical storage unit, operating system abstracts its storage devices from physical properties. This is called the **file**. The files are mapped to the physical devices by the operating system.

7.1.1.1) File Operations

Some of the most common system calls relating to files are:

- **Create**: First, we found a space in the file system for the file and then an entry for the new file must be made in a directory.
- **Opening a File**: Except create and delete, all file operations require open() first. If successful, this system call returns a file handle.
- **Reading a File**: We need to use a system call that specifies the file handle and where the next block of the file should be put. The system needs to keep a **read pointer** to the location in the file.
- **Repositioning Within a File**: Also known as file **seek**.
- **Deleting a File**: We first search for the directory the file lives in. First, we release all file space, so that it can be reused by other files, and erase or mark as free the directory entry.
- **Truncating a File**:
- **Writing**:
- **Append**:
- **Get attributes**: For example, the UNIX *make* program is commonly used to manage software development projects consisting of many source files. When make is called it examines the modification times of all the source and object files and arranges for the minimum number of compilations required to bring everything up to date. To do its job, it must look at the attributes, namely, the modification times.

- **Set attributes**:

7.1.1.2) File Types

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Figure 13.3 Common file types.

The UNIX system uses a **magic number** stored at the beginning of some binary files to indicate the data type of the file.

7.1.1.3) File Structure

Files can be structured in several ways

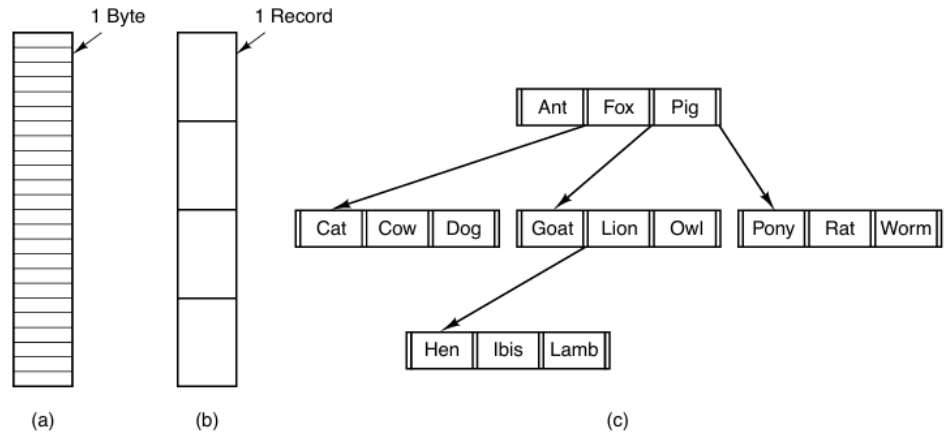


Figure 4-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

Both UNIX and Windows use byte sequence.

In the record sequence, file is a sequence of fixed-length records, each with some internal structure.

In the tree, a file consists of a tree, not necessarily in the same length, each contains a **key**. Field in a fixed position in the record.

7.1.2) Access Methods

7.1.2.1) Sequential Access

In the early systems only access method was **sequential access**. In these systems, a process could read all the records in a file in order. The read_next() operation reads the next portion of the file and advances the file pointer. Similarly, we also have write_next() operation.

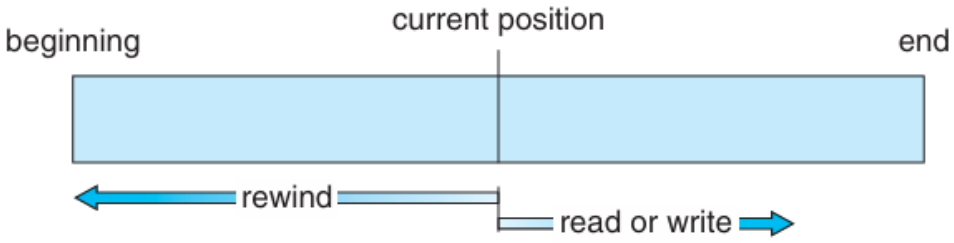


Figure 13.4 Sequential-access file.

7.1.2.2) Sequential Access

Another method is **direct access** (or **relative access**). In this method a file is made up of fixed-length **logical records** that allow programs to read and write records rapidly in no order.

Disk model of a file allows random access to any file block and the direct access-method is based on that. Files whose records can be read in any order is called **random-access files**. For direct-access, the file is viewed as a numbered sequence of blocks or records. So, we can read block 14, then 53 etc. There is no restriction on order of reading/writing.

In here, we have read() and write() functions because the next is not really the next.

7.1.3) Directory Structure

7.1.3.1) Single-Level Directory

It is the simplest directory structure. All files are contained in the same directory.

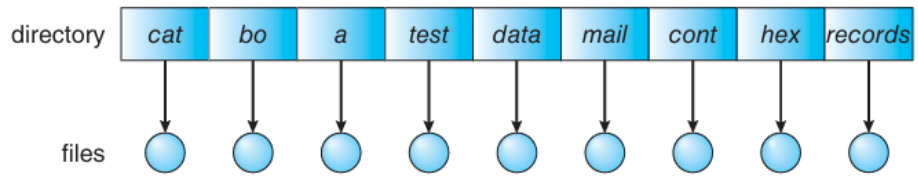


Figure 13.7 Single-level directory.

Even though it is simple it has limitations, for example, if a system is multi-user. All files must have a unique name.

7.1.3.2) Two-Level Directory

In the two-level directory structure, each user has his own **user file directory** (**UFD**). Each UFD lists the files of a single user. When a user job starts or when user logs in, **master file directory** (**MFD**) which is indexed by user name or account id is found. Later, if a user refers to a directory only its UFD is searched.

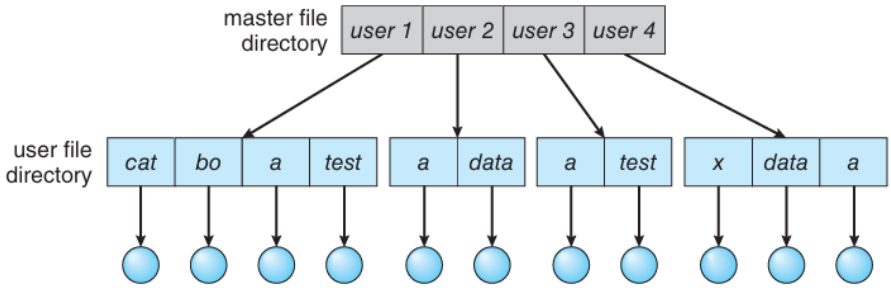


Figure 13.8 Two-level directory structure.

7.1.3.3) Tree-Structure Directory

Tree structure allows users to create their own sub-directories and organize their files accordingly. Tree has a root directory and every file in the system has a unique path name.

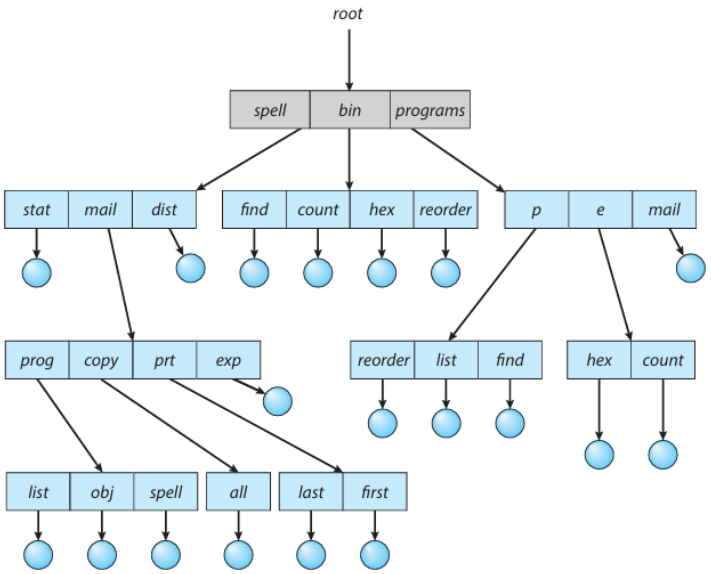


Figure 13.9 Tree-structured directory structure.

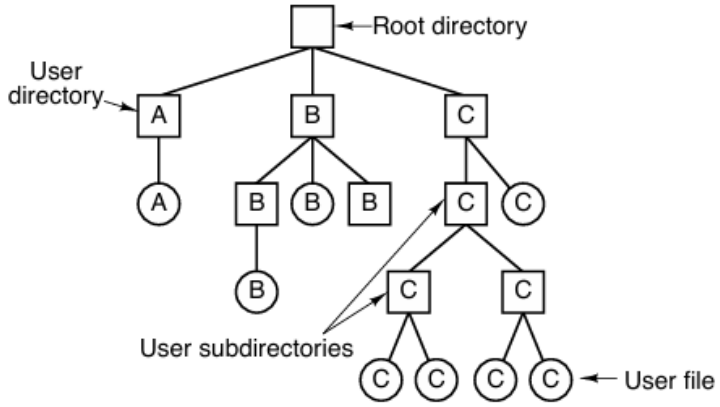


Figure 4-8. A hierarchical directory system.

7.1.3.4) Acyclic-Graph Directories

Unlike tree structures, this approach allows the *sharing* of a subdirectory at will. Please note that shared files are not copies but the original.

Shared files or subdirectories can be implemented via several way. One of the ways is to creating a **link**. Link is a pointer to a file or subdirectory. Another way is to simply duplicating all information about them in both sharing directories.

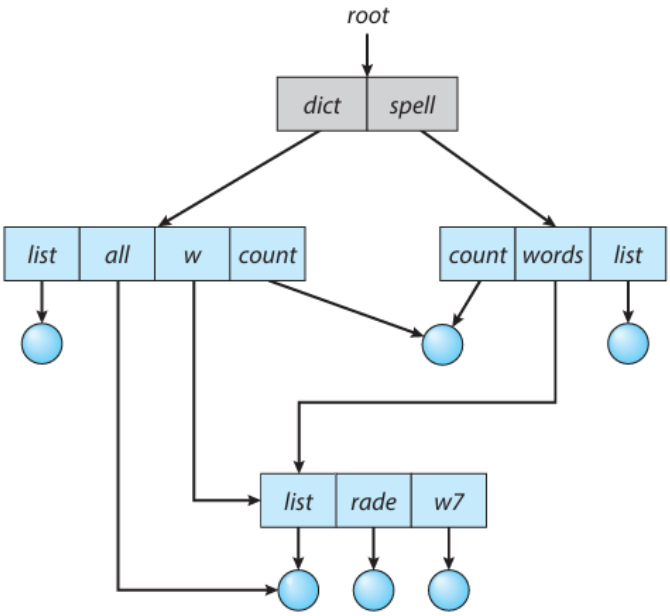


Figure 13.10 Acyclic-graph directory structure.

7.1.4) Memory-Mapped Files

Each file access requires a system call and a disk access. If we use virtual memory techniques this is called **memory mapping** a file. Memory-mapped files allow applications to map a file's contents to virtual memory, facilitating shared memory usage between processes.

7.1.4.1) Basic Mechanism

We map a disk block to a page to achieve memory mapping. Initial access to the file proceeds through ordinary demand paging which causes page faults. Read and writes to file are handled as standard memory accesses. Manipulating the files in memory speeds up file access and usage.

Generally, the systems save the changes upon the closure of the file and not immediately. When the file is closed, all memory-mapped data are written back to the file on the secondary storage.

Multiple processes might be allowed to map the same file concurrently, to allow sharing of data.

7.1.4.2) Shared Memory in the Windows API

In the Windows API if we want to create a region of shared memory, we first create a file for it to be mapped. And then establish a **view** of the mapped file in the process's virtual address space. A second process can open and create a view of the mapped file in its virtual address space.

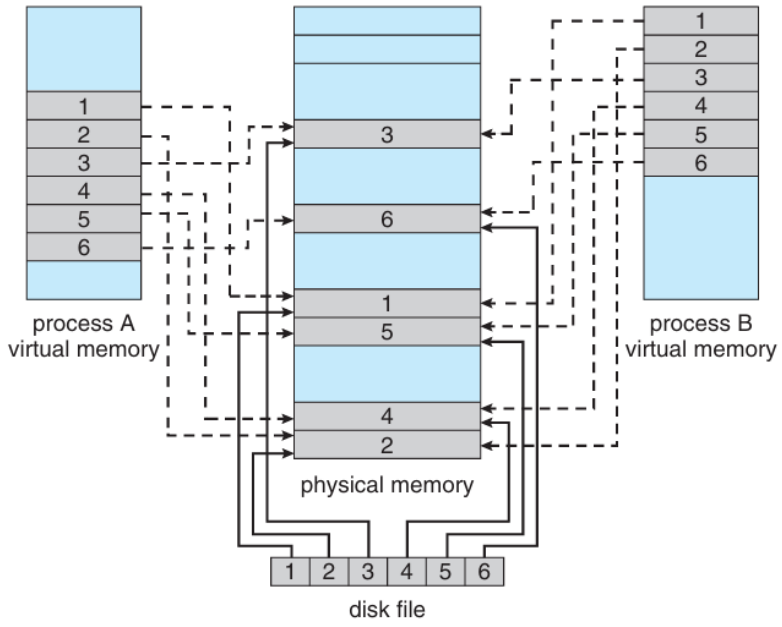


Figure 13.13 Memory-mapped files.

The producer first creates a shared-memory object using the memory-mapping features available in Windows API. The producer then writes a message to shared memory. Afterwards, consumer opens a mapping to the shared memory object and read the message.

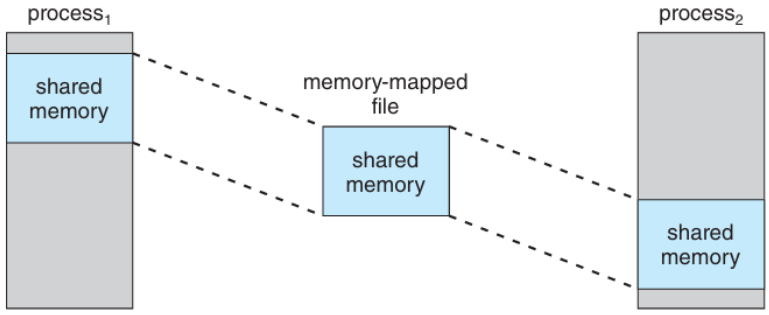


Figure 13.14 Shared memory using memory-mapped I/O.

In windows, a process first opens a file to be mapped via the CreateFile() function, which returns a **HANDLE** to the opened file.

NOTE: A handle is an abstract reference to a system resource, such as a file, windows or a thread. They're used in Windows.

Handles provide abstraction, when a process interacts with a file via a handle, it doesn't need to know the specifics of how the operating system manages file access.

Handles encapsulate the resource information and provide a way for the operating system to perform security checks and resource management.

Pointers references to a specific memory address, whereas handles indirect references managed by the operating system. When a process requests access to a resource, OS

creates a handle for that resource. Handle is generally an integer or a pointer that uniquely identifies the resource within the context of the process.

OS maintains a resource (or handle) table for each process. This table maps handles to the actual system resource. So, when a process creates a handle, it can reach to the handle table instead of the direct memory address

---

The process then creates a mapping of this file HANDLE using the CreateFileMapping() function.

Once the file mapping is done, process establishes a view of the mapped file in its virtual address space with the MapViewOfFile() function. The view of the mapped file represents the portion of the file being mapped in the virtual address pace of the process.

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

int main(int argc, char *argv[])
{
   HANDLE hFile, hMapFile;
   LPVOID lpMapAddress;

   hFile = CreateFile("temp.txt", /* file name */
      GENERIC_READ | GENERIC_WRITE, /* read/write access */
      0, /* no sharing of the file */
      NULL, /* default security */
      OPEN_ALWAYS, /* open new or existing file */
      FILE_ATTRIBUTE_NORMAL, /* routine file attributes */
      NULL); /* no file template */

   hMapFile = CreateFileMapping(hFile, /* file handle */
      NULL, /* default security */
      PAGE_READWRITE, /* read/write access to mapped pages */
      0, /* map entire file */
      0,
      TEXT("SharedObject")); /* named shared memory object */

   lpMapAddress = MapViewOfFile(hMapFile, /* mapped object handle */
      FILE_MAP_ALL_ACCESS, /* read/write access */
      0, /* mapped view of entire file */
      0,
      0);

   /* write to shared memory */
   sprintf(lpMapAddress,"Shared memory message");

   UnmapViewOfFile(lpMapAddress);
   CloseHandle(hFile);
   CloseHandle(hMapFile);
}
```

Figure 13.15 Producer writing to shared memory using the Windows API.

The call to create CreateFileMapping() creates a **named shared-memory object** called SharedObject. The consumer will communicate using this shared-memory segment by creating a mapping to the same named object.

The MapViewOfFile() function returns a pointer to the shared-memory object

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

int main(int argc, char *argv[])
{
   HANDLE hMapFile;
   LPVOID lpMapAddress;

   hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS, /* R/W access */
      FALSE, /* no inheritance */
      TEXT("SharedObject")); /* name of mapped file object */

   lpMapAddress = MapViewOfFile(hMapFile, /* mapped object handle */
      FILE_MAP_ALL_ACCESS, /* read/write access */
      0, /* mapped view of entire file */
      0,
      0);

   /* read from shared memory */
   printf("Read message %s", lpMapAddress);

   UnmapViewOfFile(lpMapAddress);
   CloseHandle(hMapFile);
}
```

Figure 13.16 Consumer reading from shared memory using the Windows API.

7.2) File System Implementation

7.2.1) File-System Structure

Most of the secondary storage needs, which the file systems are maintained, are fulfilled with disks.

1. Re-writability: A disk can be rewritten in place. When can read, modify and then write it to the same block
2. Direct Access: Any block of information can be accessed directly by disks.

These characteristics make them convenient for use.

I/O transfers between memory and mass storage is performed via *blocks* for I/O efficiency.

Device drivers can be taught as a translator. It takes high level commands and gives output consisting low-level, hardware specific, instructions which is used by hardware controller to interface I/O device and rest of the system.

Layered File System Design:

1. **I/O control** contains device drivers and interrupt handlers for information transfer between memory and disk.
2. The **basic file system** (In Linux it's called "block I/O subsystem") needs only to issue generic commands to the appropriate device driver to read and write blocks on the storage device.
3. **File organization module** manages files and logical blocks. Also has free-space manager for unallocated blocks.
4. **Logical file system** manages the metadata information, directory structure, and **file control blocks** (**FCB**) that contains info about the file, including ownership, permission etc. Also responsible for ensuring file protection.

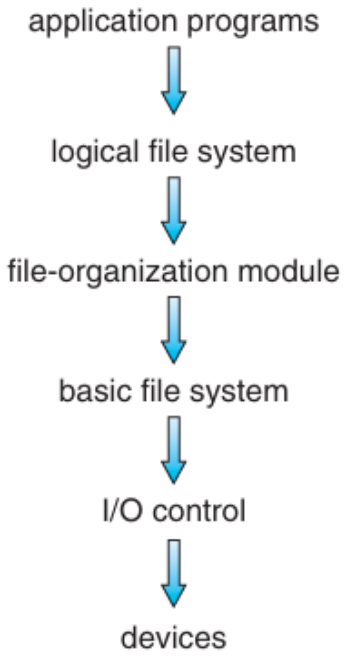


Figure 14.1 Layered file system.

UNIX uses **UNIX File System** (**UFS**) and Windows uses **NTFS**

Many disks can be divided up into partitions. The layout depends on the computer. The old computers had a BIOS and a master boot record and modern computers have UEFI-based systems.

7.2.1.1) The Master Boot Record

In older systems, the first sector of the disk, known as the Master Boot Record (MBR), is used for booting the computer. The MBR contains the partition table at its end, which holds the starting and ending addresses of each partition.

When we boot the computer, it reads and executes the MBR. The MBR locates the active partition and reads and executes its first block, known as the Volume Boot Record (VBR) or boot sector. The VBR contains code that loads the OS contained in that partition. For consistency, every partition starts with a boot block, even if it does not contain a bootable OS.

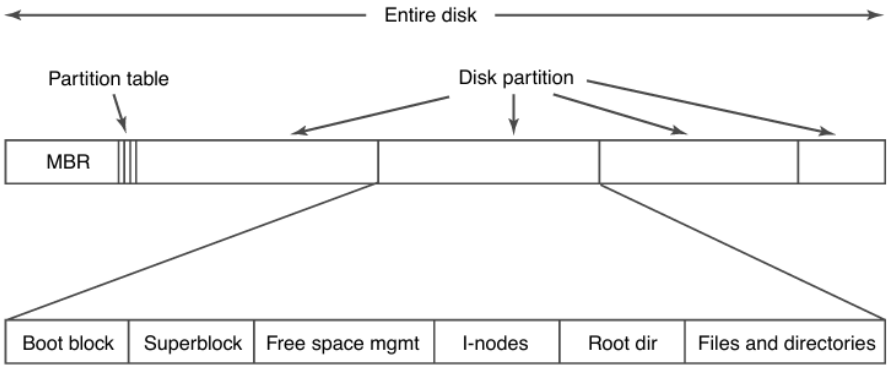


Figure 4-10. A possible file-system layout.

7.2.1.2) Unified Extensible Firmware Interface (UEFI)

UEFI looks at the second block of the device for the partition table. It keeps the first block to basically pointing to the second block in case a software expects an MBR there.

The **GPT** (**GUID** [Globally Unique] **Partition Table**) holds information about the locations of the various partitions on the disk. UEFI keeps a backup of GUID at the last block.

GPT contains the starting and ending of each partition. When the GPT is found, the firmware is good to read file systems of specific types.

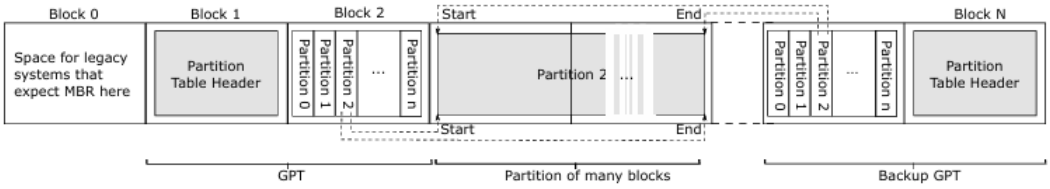


Figure 4-11. Layout for UEFI with partition table.

7.2.2) File-System Operations

The OS has system calls like open() and close() to request access to file contents.

- The information needed for OS to boot an operating system can be found in the **boot control block** (per volume). It's generally the first block that holds the OS. In UFS, it is called the **boot block** and in NTFS it's called **partition boot sector**.
- Volume details such as the number of blocks in a volume, size of blocks, free-block count and free-block pointers are held in **volume control block**.
- A directory structure (per file system) is used to organize the files. In UFS, this includes file names and associated inode numbers, in NTFS they are stored in the master file table.

Several of the in-memory structures are:

1. **Mount Table**: Contains information about each mounted volume.
2. **Directory-Structure Cache**: Holds info about recently accessed directories, might include pointers to the volume table for mounted volumes

3. **System-Wide Open-File Table**: Contains the copy of FCB of each open file, along with relevant info.
4. **Per-Process Open-File Table**: Contains pointers to the appropriate entries in the system-wide open-file table for all files open by the process
5. **Buffers**: Used to hold file-system blocks during read and write operations

Creating and managing files:

- When a new file is created, a new FCB is allocated. If FCBs are pre-allocated, an FCB is chosen from the set of free FCBs.
- The directory is read into memory, updated with the new file name and FCB, and written back to the file system

7.2.3) Directory Implementation

The directory allocation & management algorithms affect the efficiency, performance and reliability of file systems significantly.

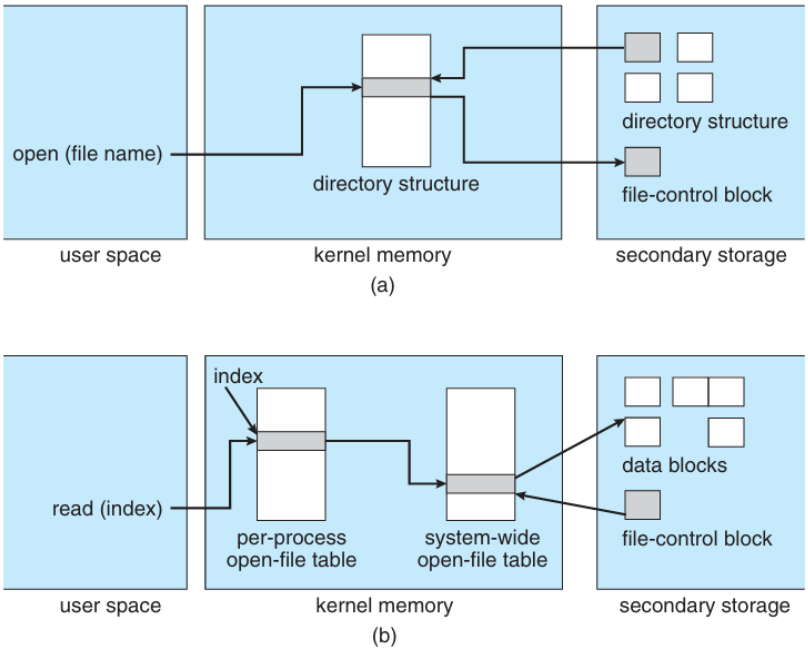


Figure 14.3 In-memory file-system structures. (a) File open. (b) File read.

7.2.3.1) Linear List

Simple to program, time-consuming to execute.

To add, first it makes sure that no file has the same name within the same directory. Then we add a new entry at the end of the directory.

To delete we search the directory for the file and if found release the space allocated to it.

Since this approach involves linear search, it is slow. Moreover, most of the OSs holds a cache for recently used directories. A sorted list allows a binary search and decreases the average search time.

7.2.3.2) Hash Table

Hash table takes a value computed for a filename and returns a pointer to the file name in the linear list. By doing so, it decreases the search time significantly.

One hardship about hash tables is that assume we have 64 entries. Hash table converts file names into integers from 0 to 64. But if we add 65th entry the size is increase to 128.

7.2.4) Allocation Methods

There are 3 allocation methods. Contiguous, linked and indexed allocation.

7.2.4.1) Contiguous Allocation

It requires that each file occupy set of contiguous blocks. It's the easiest to implement. A disk with 1-KB blocks, then a file with 50-KB would be allocated 50 consecutive blocks.

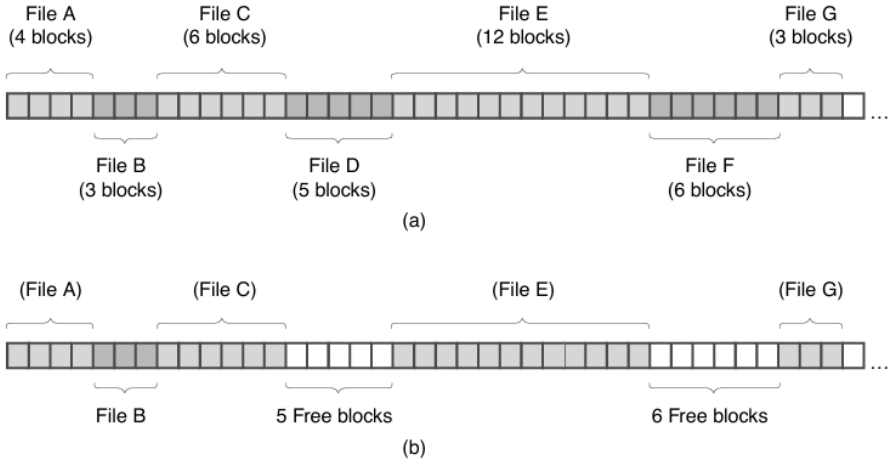


Figure 4-12. (a) Contiguous allocation of disk space for seven files. (b) The state of the disk after files *D* and *F* have been removed.

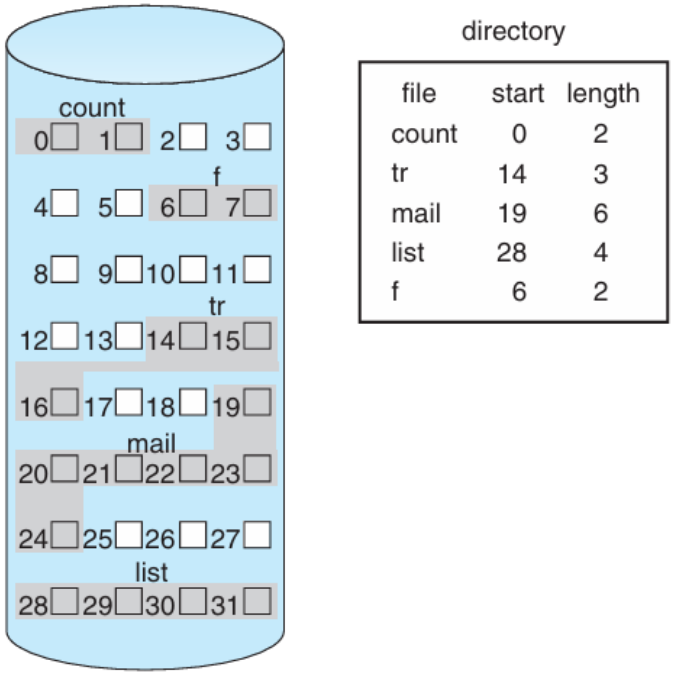


Figure 14.4 Contiguous allocation of disk space.

It provides a fast access for sequential and direct access. But finding a space for a file might be hard (Especially if it is big) because of the external fragmentation.

Moreover, external fragmentation can cause storage loss. One strategy to prevent loss to copy entire system onto another device, free the original device and then copy the files back.

7.2.4.2) Linked-List Allocation

Each file is a linked list of blocks, scattered anywhere on the device. Each block points to the next block. This solves the contiguous allocations' problems. And it is easy to grow as blocks can be added anywhere.

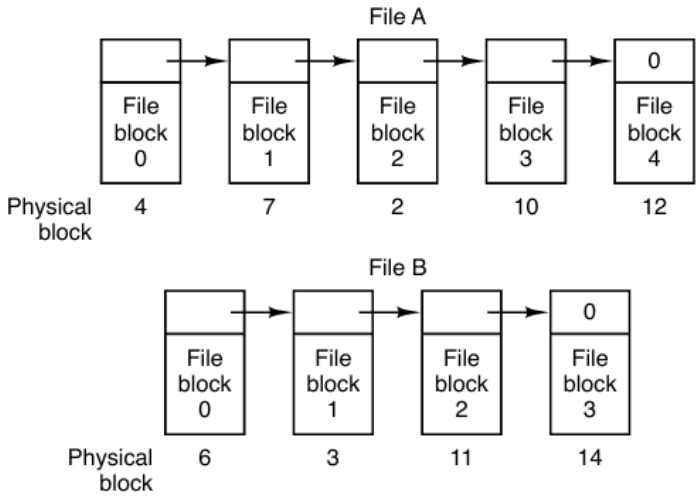


Figure 4-13. Storing a file as a linked list of disk blocks.

An important thing is the use of **FAT** (**File Allocation Table**). FAT is a variation used by MS-DOS and OS/2, storing pointers in a table instead of blocks. *FAT used like a linked list*.

Directory entry points to the block number of the first block of the table. The pointed block, points to the next entry and so on, until there is nothing left to point.

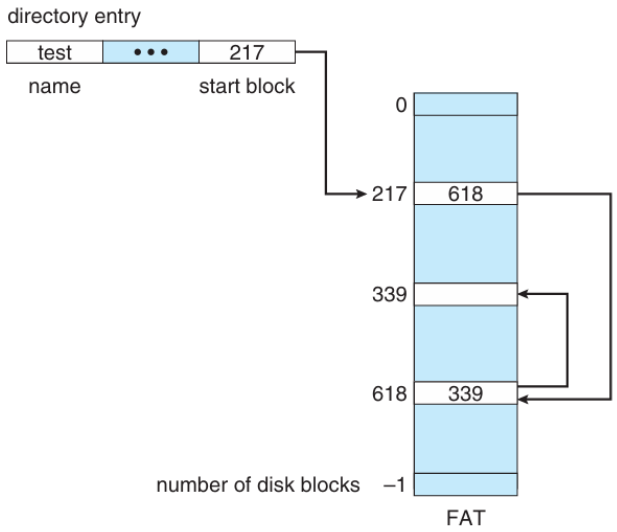


Figure 14.6 File-allocation table.

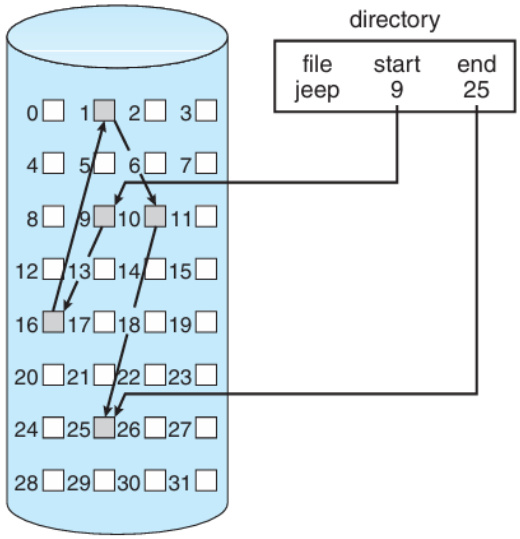


Figure 14.5 Linked allocation of disk space.

However, since it requires sequentially following pointers it is inefficient for direct access. And broken links can cause data loss.

7.2.4.3) Indexed Allocation

In the absence of FAT, pointers are scattered around the blocks. With the **indexed allocation** we have all the pointers into one location called **index location**.

7.2.5) Free-Space Management

To keep track of the free disk space, the system maintains a **free-space list**. It records all free device blocks that aren't allocated to some file or directory.

To create a new file, we check the free-space list for the required length of space. This space is then removed from the free-space list and the allocation is made. Also, when a file is deleted, its space is added to free-space list.

7.2.5.1) Bit Vector

Each block is represented as 1 bit. If the block is free the bit is 1, else it is 0.

For example, if 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space **bitmap**/**bit vector** would be:

00111100111111000110000001110000 0...

7.2.5.2) Linked List

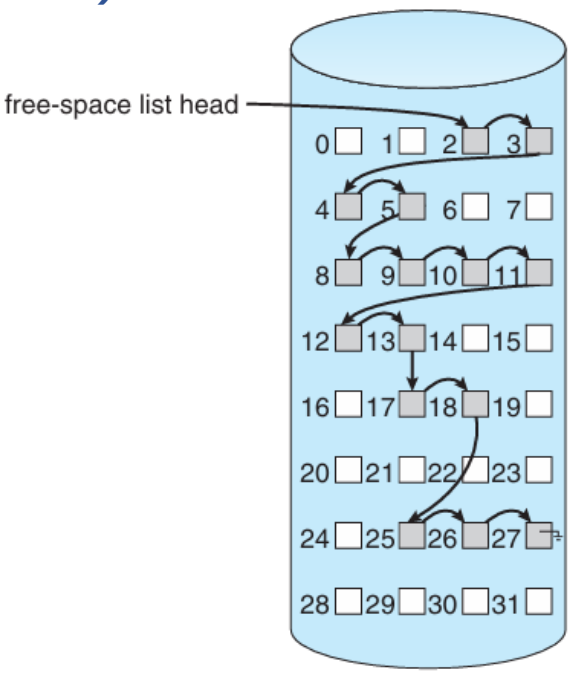


Figure 14.9 Linked free-space list on disk.

Figure is explanatory enough.

7.2.5.3) Grouping

7.2.6) Efficiency and Performance

Some systems have a separate section of the main memory for a **buffer cache**. Blocks kept in buffer cache are assumed to be reused again shortly.

Whereas some other systems cache file data using a **page cache**. Page cache uses virtual memory techniques to cache file data as pages.

Caching files via virtual address is more efficient than using physical disk blocks.

Some system (Unix, Linux and Windows) uses page both process pages and file data to cache, it is known as **unified virtual memory**.

7.2.7) Recovery

To avoid data loss or inconsistency, we should take care of files & directories in the main memory and storage volume. A system crash can cause inconsistencies among directory structures, free-block pointers and free FCB pointers.

To optimize the I/O performance, OS might cache updates which can lead to inconsistencies if crashes occur before cached updates are written to storage.

Crashes, bugs in file-system implementation, device controllers and user applications can cause corruption in file system. There are several ways to deal with it. They are:

7.2.7.1) Consistency Checking

To detect a corruption, file system scans all the metadata on each file system. Scan confirms or denies the consistency of the system. But if this scan is made when the system is booting it might take hours.

The **consistency checker**, *system program such a fsck in UNIX*, compares the data in the directory structure and other metadata with the state on storage and tries to fix any inconsistency it finds.

7.2.7.2) Log-Structured File Systems

It is inspired by database recovery algorithms. Each metadata update is treated as a transaction. If a system crashes, transactions in the log file can be replayed to maintained consistency.

7.3) File-System Internals

7.3.1) File Systems

A computer has millions and even billions of files and they are stored on random-access storage devices such as hard disk drives, optical disks and nonvolatile memory devices.

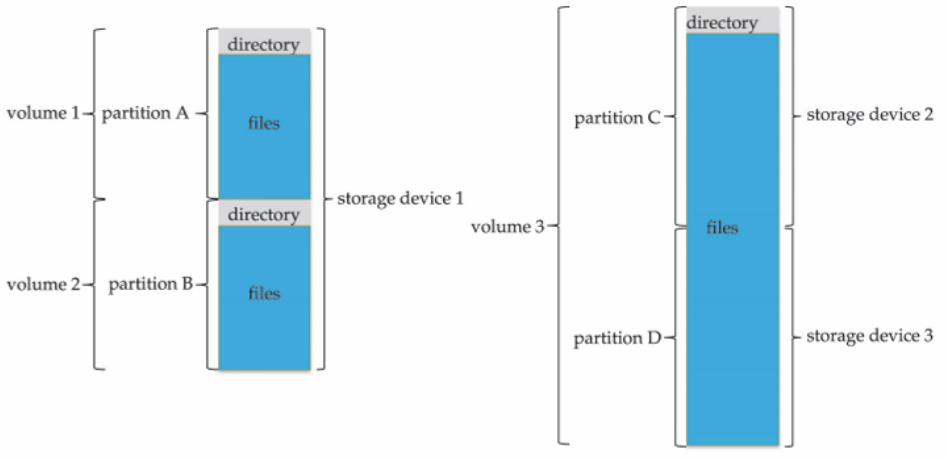


Figure 15.1 A typical storage device organization.

- **tmpfs**—a "temporary" file system that is created in volatile main memory and has its contents erased if the system reboots or crashes
- **objfs**—a "virtual" file system (essentially an interface to the kernel that looks like a file system) that gives debuggers access to kernel symbols
- **ctfs**—a virtual file system that maintains "contract" information to manage which processes start when the system boots and must continue to run during operation
- **lofs**—a "loop back" file system that allows one file system to be accessed in place of another one
- **procfs**—a virtual file system that presents information on all processes as a file system
- **ufs, zfs**—general-purpose file systems

7.3.2) File Systems Mounting

A file system should be mounted before it can be available to process on the system.

Mounting is straightforward. OS is given the name of the device and the mount point (the location within the file structure where the file system is to be attached). Some OS expects the file-system type to be provided while others can determine themselves. Mounting point is generally an empty directory.

For example, in UNIX systems, home directories might be mounted as /home.