

**Case Study of Windows XP Operating System**

**Course Name :** Operating System

**Course Title :** CSE315

**Section :** 02 & 03

**Prepared By:**

Mahmudul Hasan Sheepu (1931349)

Kaushik Dey Joy (1821818) (sec-2)

Zarin Tasnim Titly (1820428)

Nohori Jobaida (1931154)

**Submitted To:**

Mohammad Noor Nabi

**Date:** 21.12.2023

Table of Contents

[Introduction 4](#_Toc27441)

[What is Windows Operating System? 4](#_Toc27442)

[Windows XP 4](#_Toc27443)

[Design Goals/Principles 5](#_Toc27444)

[Operating-System Structure 6](#_Toc27445)

[Components of Windows XP system/OS 6](#_Toc27446)

[Shells of the Operating System 7](#_Toc27447)

[Programming Interface 8](#_Toc27448)

[Process Management: 8](#_Toc27449)

[Interprocess Communication: 9](#_Toc27450)

[Memory Management: 9](#_Toc27451)

[**Virtual memory:** 9](#_Toc27452)

[Process Creation, Termination & Communication 10](#_Toc27453)

[Process Creation 10](#_Toc27454)

[Process Termination 10](#_Toc27455)

[PROCESS COMMUNICATION 12](#_Toc27456)

[Process State Diagram & Process Management System-Calls 13](#_Toc27457)

[Process State Diagram 13](#_Toc27458)

[Process Management Calls 14](#_Toc27459)

[Process Versus Thread 15](#_Toc27460)

[User Level Thread & Kernel Level Thread: Models 16](#_Toc27461)

[User Level Threads 16](#_Toc27462)

[Kernel Level Threads 16](#_Toc27463)

[Multithreading Models 17](#_Toc27464)

[Process/Thread Scheduling Parameters 19](#_Toc27465)

[Process/Thread Scheduling Algorithm 20](#_Toc27466)

[Synchronization Tools 21](#_Toc27467)

[System generation and booting process 22](#_Toc27468)

[Generic Boot Procedure 22](#_Toc27469)

[Windows XP Boot Components 23](#_Toc27470)

[Windows XP Boot Sequence 24](#_Toc27471)

[Virtual memory management 25](#_Toc27472)

[File system 27](#_Toc27473)

[Internal Layout 27](#_Toc27474)

[Recovery 27](#_Toc27475)

[Security 28](#_Toc27476)

[Volume Management and Fault Tolerance 28](#_Toc27477)

[Volume Set 28](#_Toc27478)

[Stripe Set 29](#_Toc27479)

[Stripe Set with Parity 29](#_Toc27480)

[Disk Mirroring 30](#_Toc27481)

[Compression 31](#_Toc27482)

[Reparse Points 31](#_Toc27483)

# Introduction

## What is Windows Operating System?

It is an Operating System that has a set of programs that controls and coordinates the use of computer hardware among various application programs. It provides an environment within which user can execute programs. **To put it simply, it manages communication between your software and hardware**. Without the operating system (often referred to as the “OS”), the software would not function.

## Windows XP

Windows XP is an operating system **introduced in 2001** from Microsoft's Windows family of operating systems, the previous version of Windows being Windows Me. The "XP" in Windows **XP stands for eXPerience**.

Microsoft’s XP release was an important product since Windows 95, and it changed their whole game. Along with a redesigned look and new feel to the user interface, **the new operating system was built on the Windows 2000 kernel**, giving users a more stable and reliable environment than previous versions of Windows.

Windows XP was made available in in two versions, Windows XP Home and Windows XP Professional. The company **focused on mobility, such as technology allowed for in 2001, and included plug and play features for connecting to wireless networks**. The operating system utilizes the 802.11x wireless security standard.

The initial Windows XP release was followed by the release of Windows XP 64Bit Edition (v2002), Windows XP Media Center Edition and Windows XP 64-Bit Edition (v2003).

**Windows is distributed under a closed source license**. Closed source follows the following key philosophies:

* The source code can only be altered by Microsoft, and only viewed by few selected customers like big companies.
* Only Microsoft can make modifications to the OS and choose who can use and share their codes with.
* The copyright of distribution belongs to the Microsoft and updates are also provided by them.

**The OS is comprised of a number of pieces:**

* **The Bootloader:** The software that manages the boot process of your computer. For most users, this will simply be a splash screen that pops up and eventually goes away to boot into the operating system.
* **The kernel:** The kernel is the core of the system and manages the CPU, memory, and peripheral devices. The kernel is the “lowest” level of the OS.
* **Daemons:** These are background services (printing, sound, scheduling, etc.) that either start up during boot, or after you log into the desktop.
* **The Shell:** You’ve probably heard of the Windows command line. This is the shell – a command process that allows you to control the computer via commands typed into a text interface.
* **Graphical Server:** This is the sub-system that displays the graphics on your monitor. It is commonly referred to as the X server or just “X”.
* **Desktop Environment:** This is the piece of the puzzle that the users actually interact with. There are many desktop environments to choose from (Unity,

GNOME, Cinnamon, Enlightenment, KDE, XFCE etc.). Each desktop environment includes built-in applications (such as file managers, configuration tools, web browsers, games etc.).

* **Applications:** Desktop environments do not offer the full array of apps. Windows offers thousands upon thousands of high-quality software titles that can be easily found and installed.

# Design Goals/Principles

* Uses a **micro-kernel architecture.**
* **Merge corporate and consumer codebases** – Windows XP Professional and Windows XP Home Edition – Built on same codebase.
* **Security** – Kerberos – Access control lists – Internet Connection Firewall
* **Scalability** – Multiprocessor support - Symmetric multiprocessing – Windows XP

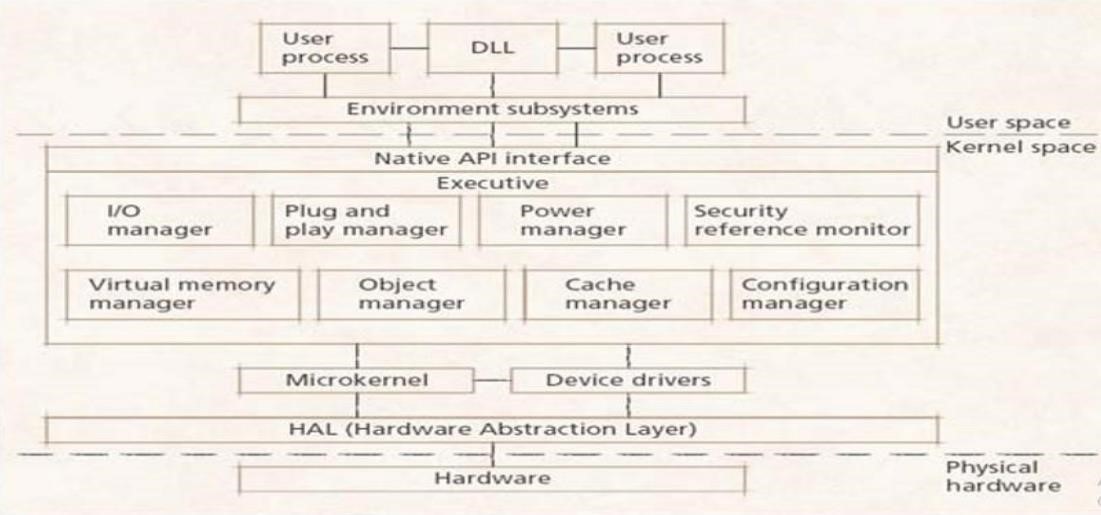
64-Bit Edition – Windows XP Embedded

* **User-friendly** – New GUI – More multimedia and network support
* **Fast boot** – 30 seconds after cold boot – 20 seconds from hibernation – 5 seconds from standby
* **Compatibility** - with MS-DOS and MS-Windows applications
* **Portability** - software can be transferred from one machine or system to another.
* **POSIX compliance** – a set of standards that define how to develop programs for UNIX – APIs
* **International support**

# Operating-System Structure

**Windows XP is Layered system of modules with two modes:**

**-> Protected mode**: consists of Hardware abstraction layer (HAL), kernel, executive **-> User mode:** consists of collection of subsystems



**Fig:Windows XP architecture**

As we can see from the figure, **Environmental subsystems emulate different operating systems** while **Protection subsystems provide security functions**.

# Components of Windows XP system/OS

Windows XP Operating System **has primarily five components**:

– **HAL**

* Interacts with hardware, drives device components on mainboard.
* Abstracts hardware specifics that differ between systems of the same architecture

**-Microkernel**

* Basic system mechanisms
* Thread scheduling, interrupt dispatching, low-level processor synchronization, recovery after a power failure etc.
* Abstracts hardware specifics that differ between architectures
* **Device drivers** 
  + Control peripheral devices.

* **Executive** 
  + Administers the main operating system subsystems, such as the file system, I/O subsystem, and system memory.
  + Native API – Environment subsystems
  + Provide a specific computing environment for user-mode processes ex: Win32, SFU, WOW64. – DLLs
  + User-mode modules that processes can dynamically link
  + Environment subsystems’ APIs are

**DLLs – System services**

* + Like Linux daemons: processes that execute in the background at all times  ex: Task Manager, Computer Browser, etc.

# Shells of the Operating System

Computers understand the language of 0's and 1's called binary language. In early days of computing, instructions are provided using binary language, which is difficult for all of us, to read and write. So, **in OS there is special program called Shell. Shell accepts your instruction or commands in English (mostly) and if it is a valid command, it is passed to kernel.**

* **Shell is a user program**. It is simply a window without any graphical decorations which is a command language interpreter that executes commands read from the standard input device (keyboard) or from a file.
* Shell is not part of system kernel, but uses the system kernel to execute programs, create files etc. Below we can see a figure that describes a typical command shell.



As seen in the figure: You type commands, using the keyboard, and you see the results of the commands displayed as text. There are no icons, and you can use the mouse only in limited circumstances**. A command shell is not a good user interface for casual users.** However, **experienced users find that the command shell is indispensable for many tasks.**

# Programming Interface

In Windows XP, a process gains access to a kernel object named **XXX** by calling the **CreateXXX** function to open a handle to **XXX**; the handle is unique to that process and a handle can be closed by calling the **CloseHandle** function; the system may delete the object if the count of processes using the object drops to 0.

XP provides three ways to share objects between processes:

* A child process inherits a handle to the object.
* One process gives the object a name when it is created, and the second process opens that name.
* **DuplicateHandle** function; gives a handle to process and the handle’s value a second process can get a handle to the same object, and thus share it.

## Process Management:

Process is started via the **CreateProcess** routine which loads any dynamic link libraries that are used by the process and creates a primary thread. Additional threads can be created by the **CreateThread** function. Every dynamic link library or executable file that is loaded into the address space of a process is identified by an instance handle Scheduling in Win32 utilizes four priority classes:

* **IDLE\_PRIORITY\_CLASS** (priority level 4)
* **NORMAL\_PRIORITY\_CLASS** (level 8 — typical for most processes)
* **HIGH\_PRIORITY\_CLASS** (level 13)
* **REALTIME\_PRIORITY\_CLASS** (level 24)

To provide performance levels needed for interactive programs, XP has a special scheduling rule for processes in the **NORMAL\_PRIORITY\_CLASS.** XP distinguishes between the foreground process that is currently selected on the screen, and the background processes that are not currently selected. When a process moves into the foreground, XP increases the scheduling quantum by some factor, typically 3.

The kernel dynamically adjusts the priority of a thread depending on whether it is I/Obound or CPU-bound. To synchronize the concurrent access to shared objects by threads, the kernel provides synchronization objects, such as semaphores and mutexes.

In addition, threads can synchronize by using the **WaitForSingleObject** or

**WaitForMultipleObjects** functions. Another method of synchronization in the Win32 API is the critical section.

A fiber is user-mode code that gets scheduled according to a user-defined scheduling algorithm. Only one fiber at a time is permitted to execute, even on multiprocessor hardware. XP includes fibers to facilitate the porting of legacy UNIX applications that are written for a fiber execution model.

## Interprocess Communication:

Win32 applications can have interprocess communication by sharing kernel objects. An alternate means of interprocess communications is message passing, which is particularly popular for Windows GUI applications. One thread sends a message to another thread or to a window. A thread can also send data with the message.

Every Win32 thread has its own input queue from which the thread receives messages. This is more reliable than the shared input queue of 16-bit windows, because with separate queues, one stuck application cannot block input to the other applications.

## Memory Management:

# Virtual memory:

There are two functions which enables the application to determine the virtual address at which the memory is allocated.

* **VirtualAlloc** reserves or commits virtual memory.
* **VirtualFree** decommits or releases the memory.

An application can use memory by memory mapping a file into its address space. Multistage process: Two processes share memory by mapping the same file into their virtual memory.

A heap in the Win32 environment is a region of reserved address space. Win 32 process is created with a 1 MB default heap. Access is synchronized to protect the heap’s space allocation data structures from damage by concurrent updates by multiple threads. Because functions that rely on global or static data typically fail to work properly in a multithreaded environment, the thread-local storage mechanism allocates global storage on a per-thread basis. The mechanism provides both dynamic and static methods of creating thread-local storage.

# Process Creation, Termination & Communication

Process Creation and Process termination are used to create and terminate processes, respectively. Details about these are given as follows:

## Process Creation

A process may be created in the system for different operations. Some of the events that lead to process creation are as follows:

* User request for process creation
* system Initialization
* Batch job initialization
* Execution of a process creation system call by a running process

A process may be created by another process using fork(). The creating process is called the parent process and the created process is the child process. A child process can have only one parent, but a parent process may have many children. Both the parent and child processes have the same memory image, open files, and environment strings. However, they have distinct address spaces

## Process Termination

Processes terminate either voluntarily through an exit system call or involuntarily as the result of a signal. Some of the causes of process termination are as follows:

* A process may be terminated after its execution is naturally completed. This process leaves the processor and releases all its resources.
* A child process may be terminated if its parent process requests for its termination.
* A process can be terminated if it tries to use a resource that it is not allowed to.

For example - A process can be terminated for trying to write into a read only file.

* If an I/O failure occurs for a process, it can be terminated. For example - If a process requires the printer and it is not working, then the process will be terminated.
* In most cases if a parent process is terminated then its child processes are also terminated. This is done because the child process cannot exist without the parent process.
* If a process requires more memory than is currently available in the system, then it is terminated because of memory scarcity. In either case, process termination causes a status code to be returned to the parent of the terminating process (if the parent still exists). This termination status is returned through the wait system call. The wait call permits an application to request the status of both stopped and terminated processes.

Finally, the wait interface allows a process to request status codes without blocking. Within the kernel, a process terminates by calling the exit() routine. The exit() routine first kills off any other threads associated with the process.

The termination of other threads is done as follows:

* Any thread entering the kernel from userspace will thread\_exit() when it traps into the kernel.
* Any thread already in the kernel and attempting to sleep will return immediately with EINTR or EAGAIN, which will force them back out to user space, freeing resources as they go. When the thread attempts to return to user space, it will instead hit exit().

The exit() routine then cleans up the process’s kernel-mode execution state by doing the following:

* Canceling any pending timers
* Releasing virtual-memory resources
* Closing open descriptors
* Handling stopped or traced child processes with the kernel-mode state reset, the process is then removed from the list of active processes—the allproc list— and is placed on the list of zombie processes pointed to by zombproc.

The process state is changed to show that no thread is currently running. The exit() routine then does the following:

* Records the termination status in the p\_xstat field of the process structure • Bundles up a copy of the process’s accumulated resource usage (for accounting purposes) and hangs this structure from the field of the process structure • Notifies the deceased process’s parent. Finally, after the parent has been notified, the cpu\_exit() routine frees any machine-dependent process resources and arranges for a final context switch from the process. The wait call works by searching a process’s descendant processes for ones that have entered the ZOMBIE state (e.g., that have terminated).

## PROCESS COMMUNICATION

The Windows operating system provides mechanisms for facilitating communications and data sharing between applications. Collectively, the activities enabled by these mechanisms are called inter-process communications (IPC).

Inter Process Communication or IPC as name suggests, is used to share data between two applications or processes. The processes can be on the same computer or somewhere else in the network.

Windows operating system supports various techniques for IPC, these are:

* **COM**: Component Object Model offers a platform to interact in Server and Client pattern between processes. COM server can be a local server or InProcess server. There can be also multiple COM clients which interact with the COM server and exchange data.

* **Copy Data**: Windows provides a message i.e., WM\_COPYDATA which enables a process to share data with another process. It can be used with SendMessage API of win32 and COPYDATASTRUCT is used as a parameter. This message is used in case of local computer only.

* **DDE**: Dynamic Data Exchange is a protocol that contains a set of guidelines and rules to send data across processes. A process can use SendMessage API with

WM\_DDE\_INITIATE or WM\_DDE\_ACK message sent in response to WM\_DDE\_INITIATE message. It uses shared memory to exchange data.

* **File Mapping**: File Mapping, a fast communication mechanism between processes and gives an efficient way to use the file content in the virtual memory or by accessing the memory sharing. In this IPC, the data of the file is treated as a part of the address space of the process so that process can easily access the address of the content.

* **Pipes**: Pipes can be used as both single and bi-directional data sharing mechanism. Windows supports two types of pipes i.e., pipe and anonymous pipe. Anonymous pipes can be used in the same network or between the related processes only, while named pipe can be used over a network within different processes. Pipe can be considered as a FIFO queue where one end acts as a server and other as the client.

* **Sockets**: Socket is an efficient way to send and receive data over the network and on the local computer. It uses multiple protocols like TCP/IP and UDP. It is used with the combination of machine IP and Port address where the data can be transported.

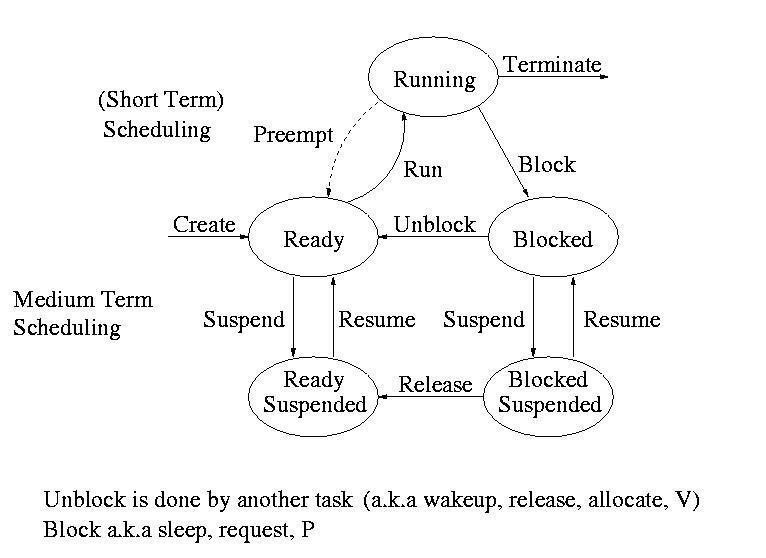
Once you decide on insertion of IPC in your application then you need to decide on which IPC you require. Also, there are several aspects you need to think about IPC for local network or IPC will work on same OS or different operating systems or Performance matters or not.

# Process State Diagram & Process Management System-Calls

## Process State Diagram

PROCESS STATE in the operating system can be in any of the following states:

* NEW- The process is being created.
* READY- The process is waiting to be assigned to a processor.
* RUNNING- Instructions are being executed.
* WAITING- The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
* TERMINATED- The process has finished execution

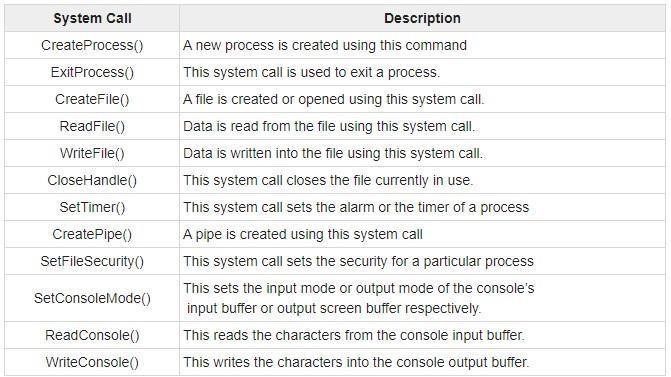


As from the above figure we can see, **Process is created** and sent to ready where it waits to be assigned to a processor. It is suspended and after it is assigned, it is resumed and is **running where instructions are being executed** and now suddenly in some case if an **interrupt occurs** which **has higher** **priority** hence **the process is blocked** and kept **waiting** after the interrupt has been executed, process is resumed as **unblock is done by another task such as release or wakeup**. Later the process has finished executing and **is terminated.**

## Process Management Calls

**System calls in Windows are used for file system control, process control, interprocess communication, main memory management, I/O device handling, security etc.** The programs interact with the Windows operating system using the system calls. **Since system calls are the only way to access the kernel, all the programs requiring resources must use system calls.**

Details about some of the important system calls in Windows are given as follows-



**Process control**: CreateProcess(), ExitProcess(), WaitForSingleObject()

**Device manipulation**: SetConsoleMode(), ReadConsole(), WriteConsole()

**File manipulation**: CreateFile(), ReadFile(), WriteFile(), CloseHandle()

**Information maintenance**: GetCurrentProcessID(), SetTimer(), Sleep()

**Communication:** CreatePipe(), CreateFileMapping(), MapViewOfFile()

**Protection:** SetFileSecurity(),InitlializeSecurityDescriptor(), SetSecurityDescriptorGroup

()

##### Process Versus Thread

|  |  |  |
| --- | --- | --- |
| **Definition** | Process means a program is in execution. | Thread means a segment of a process. |
| **Lightweight** | The process is not Lightweight. | Threads are Lightweight. |
| **Termination time** | The process takes more time to terminate. | The thread takes less time to terminate. |
| **Creation time** | It takes more time for creation. | It takes less time for creation. |
| **Communication** | Communication between processes needs more time compared to thread. | Communication between threads requires less time compared to processes. |
| **Context switching time** | It takes more time for context switching. | It takes less time for context switching. |
| **Resource** | Process consumes more resources. | Thread consumes fewer resources. |
| **Treatment by OS** | Different process is treading separately by OS. | All the level peer threads are treated as a single task by OS. |
| **Memory** | The process is mostly isolated. | Threads share memory. |
| **Sharing** | It does not share data | Threads share data with each other. |

# User Level Thread & Kernel Level Thread: Models

Threads are implemented in following two ways: − • User Level Threads − User managed threads.

• Kernel Level Threads − Operating System managed threads acting on kernel, an operating system core.

### User Level Threads

In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.

**Advantages**

* Thread switching does not require Kernel mode privileges.
* User level thread can run on any operating system.
* Scheduling can be application specific in the user level thread.
* User level threads are fast to create and manage.

**Disadvantages**

* In a typical operating system, most system calls are blocking.
* Multithreaded application cannot take advantage of multiprocessing.

### Kernel Level Threads

In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals’ threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

**Advantages**

* Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
* If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
* Kernel routines themselves can be multithreaded.

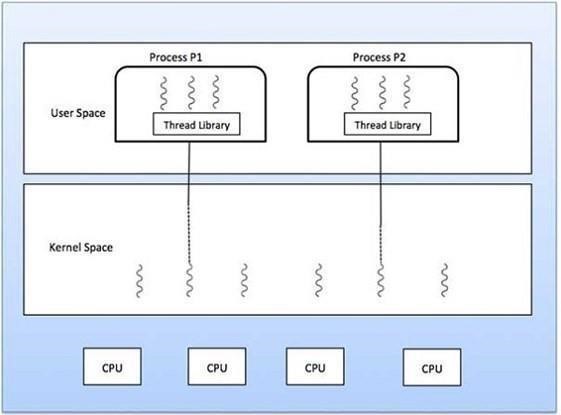
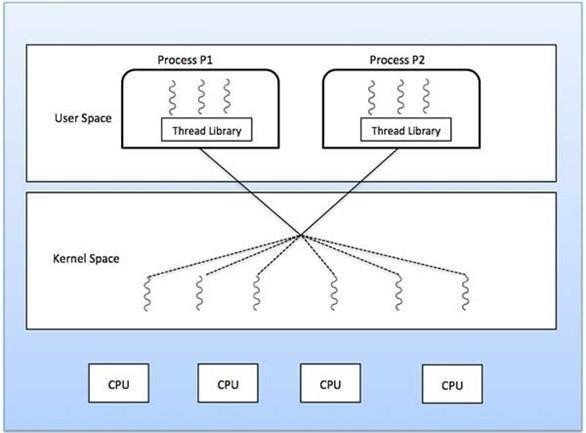
**Disadvantages**  • Kernel threads are generally slower to create and manage than the user threads.

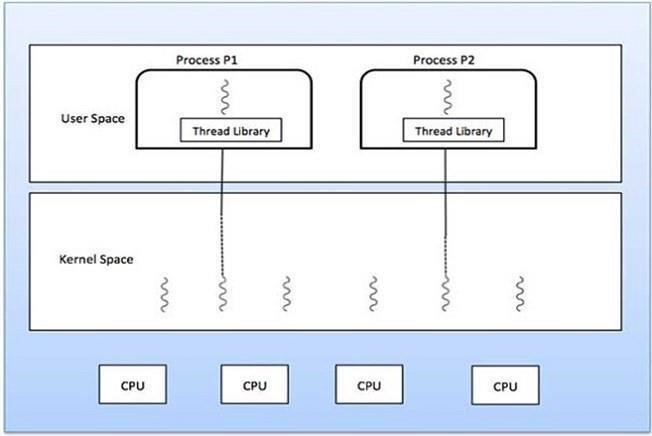
* Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

### Multithreading Models

Some operating system provides a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types:

1. Many to many relationships.
2. Many to one relationship.
3. One to one relationship.





**Many to Many Model**

Model The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.

The following diagram above shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.

**Many to One Model**

Many-to-one model maps many user levels threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the manytoone relationship modes.

**One to One Model**

There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, Windows NT and windows 2000 use one to one relationship model.

# Process/Thread Scheduling Parameters

1. **Scheduling parameters.**

Process priority, amount of CPU time consumed recently, amount of time spent sleeping recently. Together, these are used to determine which process to run next.

1. **Memory image.**

Pointers to the text, data, and stack segments, or page tables. If the text segment is shared, the text pointer points to the shared text table. When the process is not in memory, information about how to find its parts on disk is here too.

1. **Signals.**

Pointers to the text, data, and stack segments, or page tables. If the text segment is shared, the text pointer points to the shared text table. When the process is not in memory, information about how to find its parts on disk is here too.

1. **Machine registers**.

When a trap to the kernel occurs, the machine registers (including the floatingpoint ones, if used) are saved here.

1. **System call state.**

Information about the current system call, including the parameters, and results.

1. **Accounting.**

Pointer to a table that keeps track of the user and system CPU time used by the process. Some systems also maintain limits here on the amount of CPU time a process may use, the maximum size of its stack, the number of page frames it may consume, and other items.

1. **Kernel stack.**

A fixed stack for use by the kernel part of the process.

1. **Miscellaneous.**

Current process state, event being waited for, if any, time until alarm clock goes off, PID, PID of the parent process, and user and group identification.

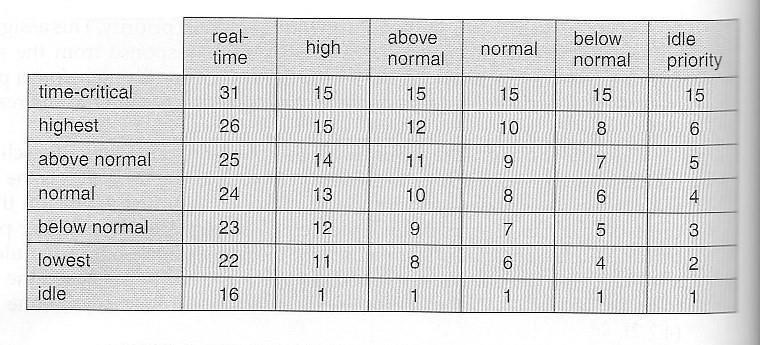
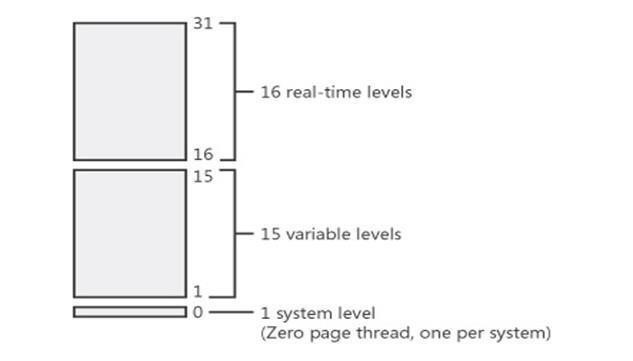
# Process/Thread Scheduling Algorithm

**NT-based versions of Windows** use a CPU scheduler based on a **multilevel feedback queue, with 32 priority levels defined**. It is intended to meet the following design requirements for multimode systems:

* Give preference to short jobs.
* Give preference to I/O bound processes.
* Quickly establish the nature of a process and schedule the process accordingly. All processes receive a priority boost after a wait event, but **processes that have experienced a keyboard I/O wait get a larger boost** thanthose that have experienced a disk I/O wait. “**Foreground” processes given higher priority**.

As we know, Windows XP uses a quantum-based, preemptive priority scheduling algorithm. It uses a 32-level priority scheme to determine the order of **thread execution, divided into two classes - variable class from 1 to 15 and real-time class from 16 to 31,** (plus a thread at priority 0 managing memory)

->There is also a special idle thread that is scheduled when no other threads are



->Win XP identifies 7 priority classes (rows on the table above), and 6 relative priorities within each class (columns.)

->Processes are also each given a base priority within their priority class. When variable class processes consume their entire time quanta, then their priority gets lowered, but not below their base priority.

->Processes in the foreground (active window) have their scheduling quanta multiplied by 3, to give better response to interactive processes in the foreground.

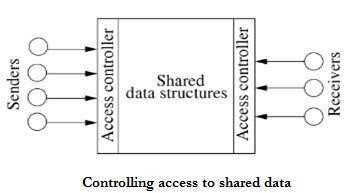
# Synchronization Tools

Each process executes its own operations on shared variables sequentially, as specified by its own program. Nevertheless, different processes may execute their operations on the same shared variable concurrently. That is, operation executions of different processes may overlap, and they may affect one another.

* Each operation on a shared variable, when executed indivisibly, transforms the variable from one consistent value to another. However, when the ready. 22 operations arc executed concurrently on a shared variable, the consistency of its values may not the guaranteed.
* **The behaviors of operation executions on shared variables must, be predictable for effective inter-process communication**. Thus, operation executions on shared variables may need to be coordinated to ensure their consistency semantics.

* **Coordination of accesses to shared variables is called synchronization**. A synchronization solution coordinates accesses from processes to shared variables.

As shown in Figure, where all accesses to shared variables are channeled through access controllers.



**The controllers do the coordination**. Most operating systems implement a few different synchronization schemes for process coordination purposes. Each scheme supports a set of primitives. The **primitives are used when it is absolutely necessary to have orderly executions of operations on shared variables in a particular manner.**

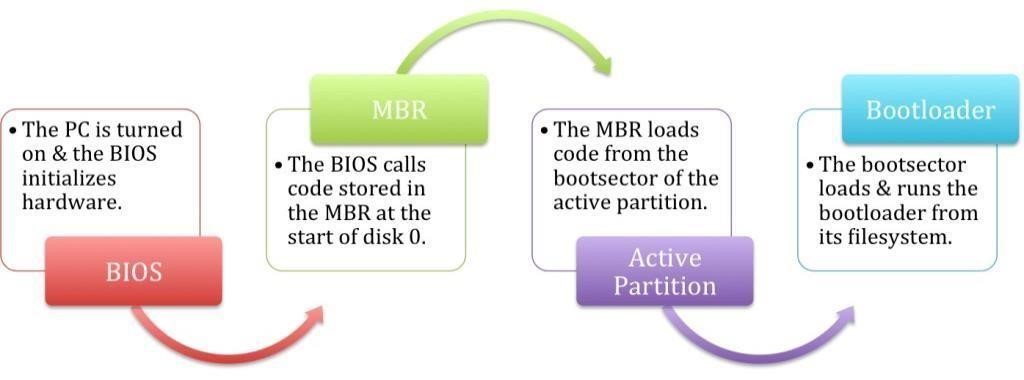
# System generation and booting process

**About the Windows XP boot process**

At first, we outline the basic components involved in the process of booting up a PC running Windows XP, covering the initial process involved in going from power up to loading NTLDR, and then the subsequent procedure and events involved in displaying the boot menu listing available OS entries and booting into the selected item.

## Generic Boot Procedure

Before we can take a detailed look at the process by which Windows XP’s boot process works, it is important to first understand the basic principles that precede any bootloader, starting with when the PC is first powered up. You can read up on a more detailed explanation of the standard BIOS boot-up sequence here, but the following is enough to get going with:



As can be seen, there is a chain of handoffs beginning at the time you power up your PC; **starting from the BIOS,** continuing through to the **MBR, bootsector,** and then, finally, the **bootloader.**

There are **two different reasons for this seemingly-convoluted chain**. The first is for **portability**: you can take out and replace any one of the components powering/facilitating these four steps without needing to replace the others, as all interactions are abstracted to a set of agreed-upon behaviors — and this is especially important because each of these individual layers is traditionally manufactured, written, designed, or developed by a different entity.

The second is **a more technical limitation**: the amount of space available for the executable bootloader code (read: how big and complicated the bootloader is allowed to be) is severely restricted in the first stages, and increases as you go up the ladder. The bootsector is similarly restricted in size (though the exact limits depend on the filesystem of the active partition), and it’s only after you reach the final stage of the actual bootloader file that you’re free to write lengthy boot code.

This lengthy procedure is responsible for everything up to the point that the Windows XP bootloader, NTLDR (short for NT loader) is finally located and called by the code in the bootsector of the NTFS or FAT32 boot partition.

##### Windows XP Boot Components

**Once the boot-related code in the bootsector of the active partition is called**, it will **search for a file called NTLDR in its root directory** (the partition itself is expected to be of NTFS or FAT32 format). NTLDR itself, while traditionally given all the praise and glory, does not do its job alone and is, at best, only a third of the solution.

**NTLDR**

NTLDR is the name of the **bootloader used by Microsoft Windows**, starting from Windows NT and continuing through Windows 2000, Windows XP, and Windows Server 2003. With the release of Windows Vista, NTLDR is no longer used and is now supplanted by **BOOTMGR**, the new Windows bootloader. It’s normally located in the root directory of the active partition on the first boot drive. With typically Windows XP installations, this is usually C:\ and the same partition that Windows itself is installed to, though that definitely does not have to be the case. Especially in cases where more than one OS is installed on the same computer (on different partitions), NTLDR takes heavy advantage of this feature.

**BOOT.INI**

Boot.ini is a plain-text file located in the root of the active partition on the first boot drive. **NTLDR reads boot configuration information from the boot.ini file,** and uses it to display and then **boot into the various operating systems**. BOOT.INI1 can contain references to Windows NT-based operating systems (Windows NT, Windows 2000, Windows XP, Windows Server 2003) as well as entries pointing to “chainloaders” (bootloaders for other operating systems saved to a file) to load non-Windows OSes. **This is where new operating systems are added to the bootloader** and from where the boot options/parameters of existing entries may be modified or replaced.

**NTDETECT.COM**

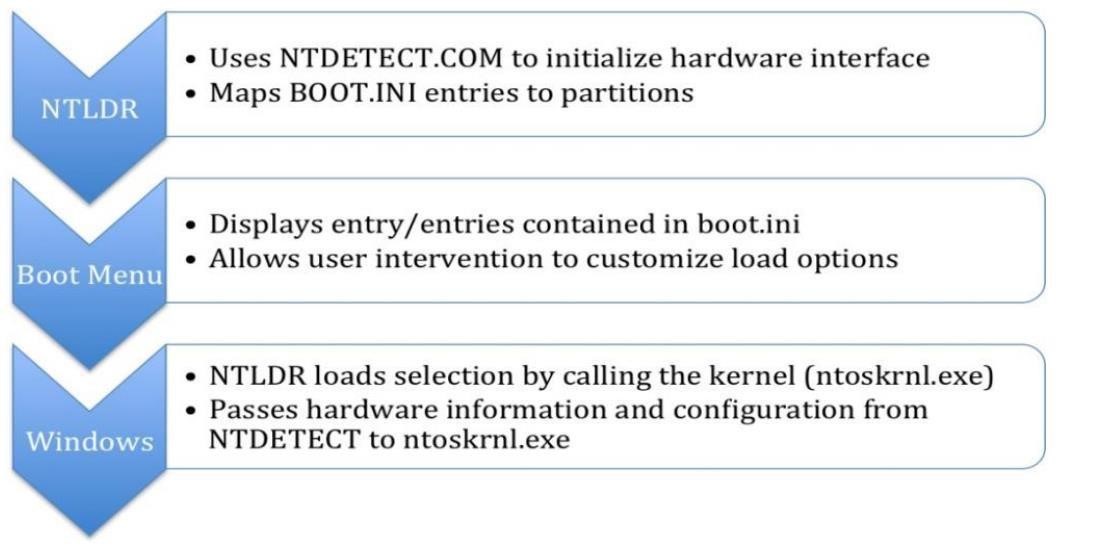
The ntdetect.com file is part of NT-based operating systems of Windows **and it’s used during boot up to detect hardware components needed to start the operating system**. At the startup process the **NTLDR starts and ntdetect.com collects hardware information that will be sent to the ntoskrnl.exe file** (Windows kernel). NTDETECT.COM is found in all NT-based operating systems: Windows XP, 2003 and Windows Vista. It collects this type of hardware information:

* video adapters
* hardware date and time
* keyboards
* hard drives
* mouse
* and others (bus and adapter types, floppy disks etc.)

Information and configuration collected and generated by NTDETECT are passed along to the Windows NT kernel (ntoskrnl.exe) by NTLDR when an operating system is selected. While the Windows kernel has other means of identifying and communicating with the underlying hardware on your PC (via the HAL, or hardware abstraction layer), the information from NTDETECT is crucial in facilitating first access to the disks and other hardware.

### Windows XP Boot Sequence

As mentioned above, **NTLDR uses both BOOT.INI and NTDETECT.COM to ultimately load a Windows installation from a local disk.** The following chart summarizes the boot procedure, highlighting the interactions between the various components of the Windows XP boot system:

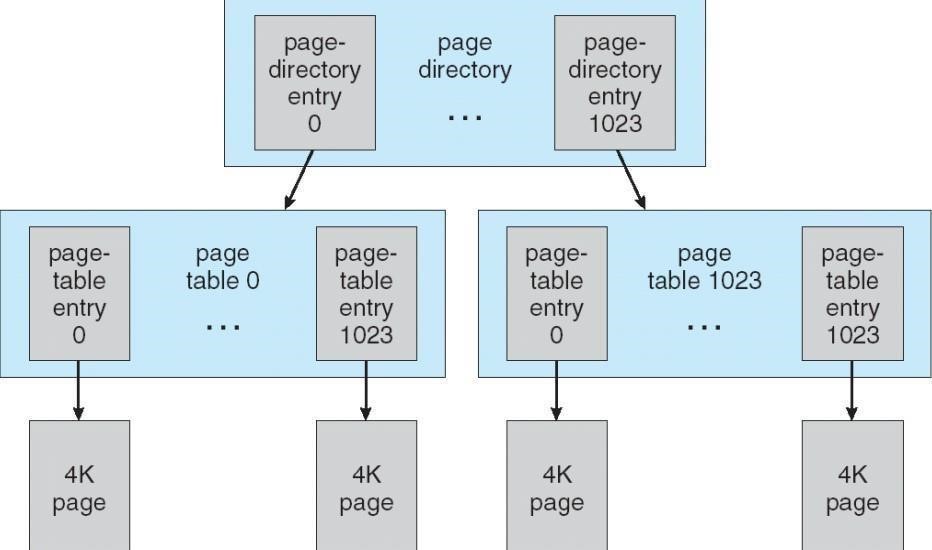


# Virtual memory management

The design of the VM manager assumes that the underlying hardware supports virtual to physical mapping a paging mechanism, transparent cache coherence on multiprocessor systems, and virtual addressing aliasing. The VM manager in XP uses a page-based management scheme with a page size of 4 KB.

The XP VM manager uses a two step process to allocate memory:

* The first step reserves a portion of the process’s address space
* The second step commits the allocation by assigning space in the 2000 paging file.

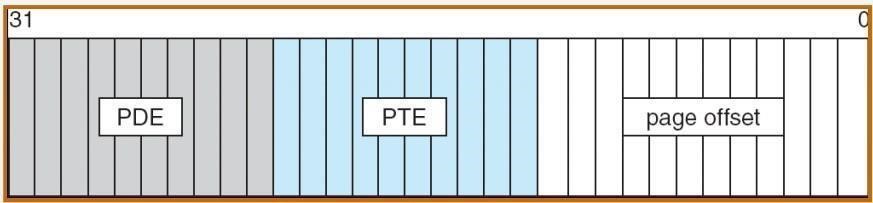


The virtual address translation in XP uses several data structures:

* Each process has a page directory that contains 1024 page directory entries of size 4 bytes.
* Each page directory entry points to a page table which contains 1024 page table entries (PTEs) of size 4 bytes.
* Each PTE points to a 4 KB page frame in physical memory.

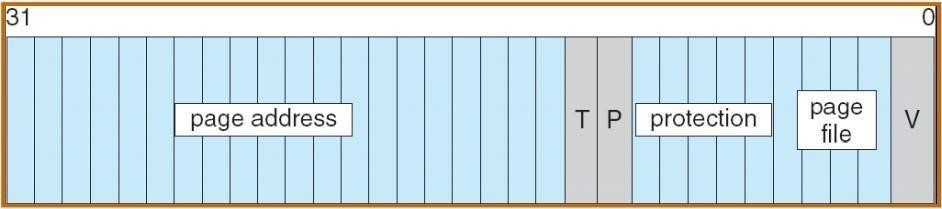
A 10-bit integer can represent all the values from 0 to 1023, therefore, can select any entry in the page directory, or in a page table. This property is used when translating a virtual address pointer to a bye address in physical memory. A page can be in one of six states: valid, zeroed, free standby, modified, and bad.

**Virtual-to-Physical Address Translation:**



10-bits for page directory entry (PDE), 20-bits for page table entry (PTE), and 12bits for byte offset in page.

**Page File Page-Table Entry:**



5 bits for page protection, 20 bits for page frame address, 4 bits to select a paging file, and 3 bits that describe the page state. V = 0.

# File system

The fundamental structure of the XP file system (NTFS) is a volume, created by the XP disk administrator utility which is based on a logical disk partition. It may occupy a portion of a disk, an entire disk, or span across several disks. All metadata, such as information about the volume, is stored in a regular file. NTFS uses clusters as the underlying unit of disk allocation:

* A cluster is a number of disk sectors that is a power of two.
* Because the cluster size is smaller than for the 16-bit FAT file system, the amount of internal fragmentation is reduced.

#### Internal Layout

NTFS uses logical cluster numbers (LCNs) as disk addresses. A file in NTFS is not a simple byte stream, as in MS-DOS or UNIX, rather, it is a structured object consisting of attributes. Every file in NTFS is described by one or more records in an array stored in a special file called the Master File Table (MFT). Each file on an NTFS volume has a unique ID called a file reference.

* 64-bit quantity that consists of a 48-bit file number and a 16-bit sequence number.
* Can be used to perform internal consistency checks. The NTFS name space is organized by a hierarchy of directories; the index root contains the top level of the B+ tree.

#### Recovery

This scheme does not guarantee that all the user file data can be recovered after a crash, just that the file system data structures (the metadata files) are undamaged and reflect some consistent state prior to the crash. The log is stored in the third metadata file at the beginning of the volume. The logging functionality is provided by the XP log file service.

#### Security

Security of an NTFS volume is derived from the XP object model. Each file object has a security descriptor attribute stored in this MFT record. This attribute contains the access token of the owner of the file, and an access control list that states the access privileges that are granted to each user that has access to the file.

#### Volume Management and Fault Tolerance

**FtDisk**, the fault tolerant disk driver for XP, provides several ways to combine multiple SCSI disk drives into one logical volume. Logically concatenate multiple disks to form a large logical volume, a volume set.

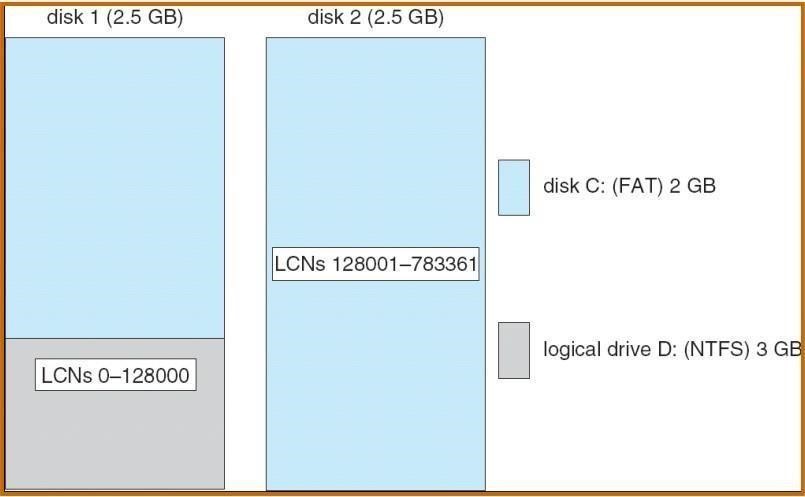
Interleave multiple physical partitions in round-robin fashion to form a stripe set (also called RAID level 0, or “disk striping”). Variation: stripe set with parity, or RAID level

5.

Disk mirroring, or RAID level 1, is a robust scheme that uses a mirror set — two equally sized partitions on two disks with identical data contents.

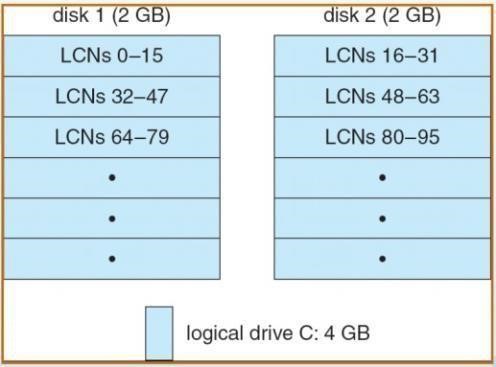
To deal with disk sectors that go bad**, FtDisk,** uses a hardware technique called sector sparing and NTFS uses a software technique called cluster remapping.

#### Volume Set



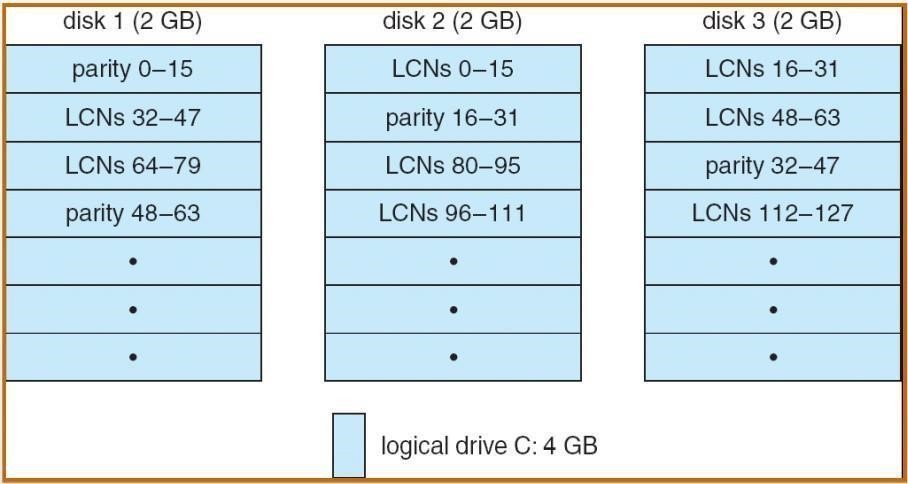
One way to combine multiple disks is to concatenate them logically to form a large logical volume, as shown in the diagram above. In Windows XP, this logical volume, called a volume set, can consist of up to 32 physical partitions. A volume set that contains an NTFS volume can be extended without disturbance of the data already stored in the file system. The bitmap metadata on the NTFS volume are simply extended to cover the newly added space. NTFS continues to use the same LCN mechanism that it uses for a single physical disk, and the FtDisk driver supplies the mapping from a logical-volume offset to the offset on one particular disk.

#### Stripe Set



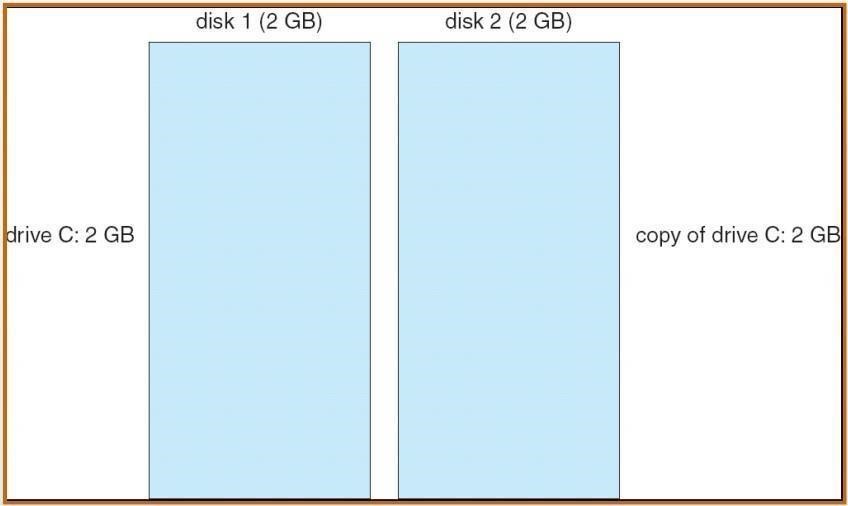
Another way to combine multiple physical partitions is to interleave their blocks in roundrobin fashion to form a strip set, as shown in the above diagram. This scheme is also called RAID level 0, or disk striping. FtDisk uses a stripe size of 64 KB. The first 64 KB of the logical volume are stored in the first physical 30 partition, the second 64 KB in the second physical partition, and so on, until each partition has contributed 64 KB of space. Then, the allocation wraps ar01md to the first disk, allocating the second 64-KB block. A stripe set forms one large logical volume, but the physical layout can improve the I/0 bandwidth, because, for a large I/0, all the disks can transfer data in parallel.

#### Stripe Set with Parity



A variation of this idea is the stripe set with parity, which is shown in the diagram above. This scheme is also called RAID level 5. Suppose that a stripe set has eight disks. Seven of the disks will store data stripes, with one data stripe on each disk, and the eighth disk will store a parity stripe for each data stripe. The parity stripe contains the byte-wise exclusive or of the data stripes. If any one of the eight stripes is destroyed, the system can reconstruct the data by calculating the exclusive or of the remaining seven. This ability to reconstruct data makes the disk array much less likely to lose data in case of a disk failure. Notice that an update to one data stripe also requires recalculation of the parity stripe. Seven concurrent writes to seven different data stripes thus require updates to seven parity stripes. If the parity stripes were all on the same disk, that disk could have seven times the I/0 load of the data disks. To avoid creating this bottleneck, we spread the parity stripes over all the disks by assigning them in round-robin style. To build a stripe set with parity, we need a minimum of three equal-sized partitions located on three separate disks.

#### Disk Mirroring



An even more robust scheme is called disk mirroring, or RAID level 1; it is depicted in the diagram above. A mirror set comprises two equal-sized partitions on two disks. When an application writes data to a mirror set, FtDisk writes the data to both partitions, so that the data contents of the two partitions are identical. If one partition fails, FtDisk has another copy safely stored on the mirror. Mirror sets can also improve performance, because read requests can be split between the two mirrors, giving each mirror half of the workload. To protect against the failure of a disk controller, we can attach the two disks of a mirror set to two separate disk controllers. This arrangement is called a duplex set.

#### Compression

To compress a file, NTFS divides the file’s data into compression units, which are blocks of 16 contiguous clusters.

For sparse files, NTFS uses another technique to save space:

* Clusters that contain all zeros are not actually allocated or stored on disk.
* Instead, gaps are left in the sequence of virtual cluster numbers stored in the MFT entry for the file.
* When reading a file, if a gap in the virtual cluster numbers is found, NTFS just zerofills that portion of the caller’s buffer

#### Reparse Points

A reparse point returns an error code when accessed. The reparse data tells the I/O manager what to do next. Reparse points can be used to provide the functionality of UNIX mounts. Reparse points can also be used to access files that have been moved to offline storage.