# Case Study 1: LINUX

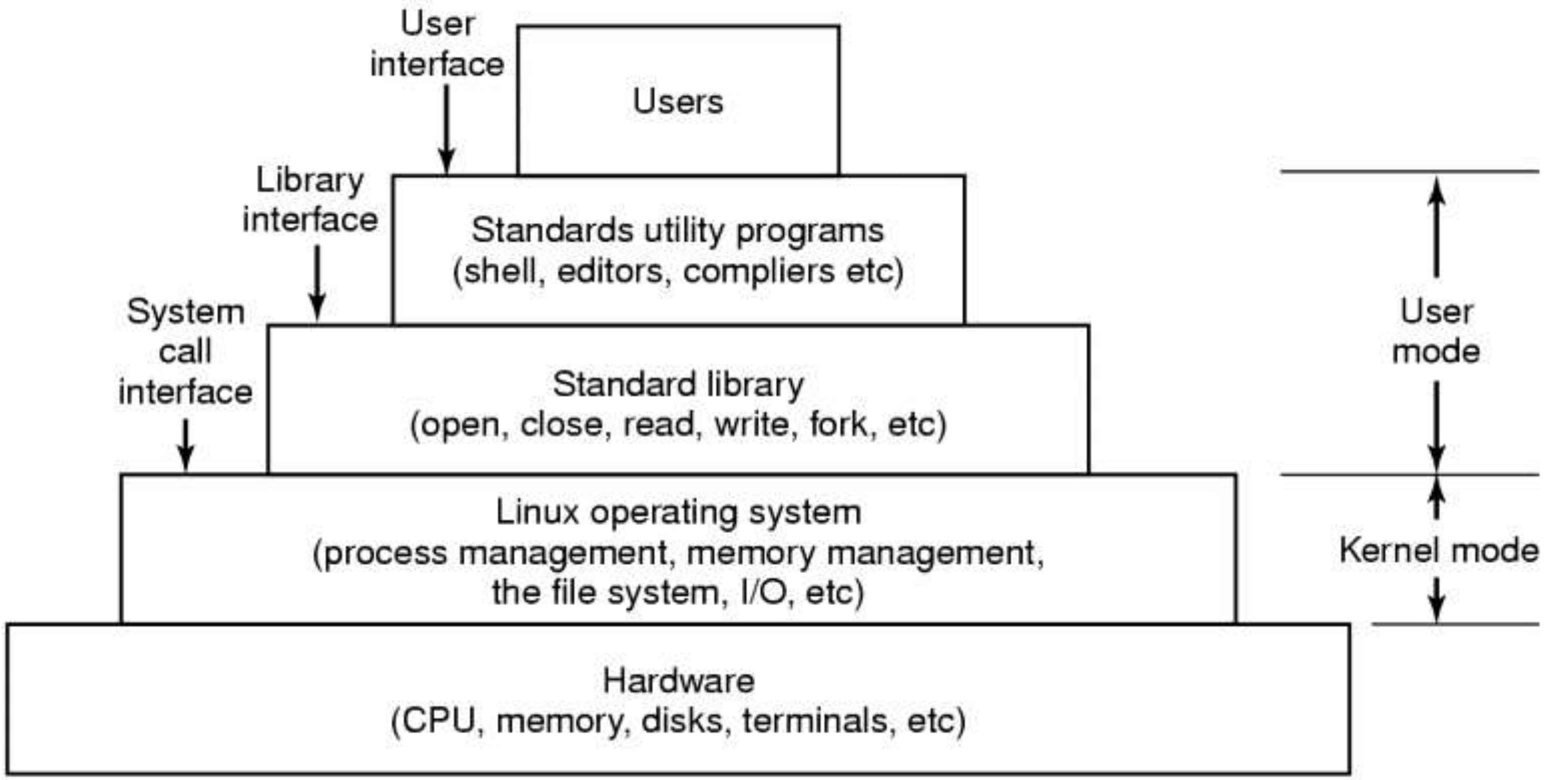
# History of UNIX and Linux

- UNICS
- PDP-11 UNIX
- Portable UNIX
- Berkeley UNIX
- Standard UNIX
- MINIX
- Linux

# UNIX/Linux Goals

- Designed by programmers, for programmers
- Designed to be:
  - Simple
  - Elegant
  - Consistent
  - Powerful
  - Flexible

# Interfaces to Linux




The layers in a Linux system.

# Linux Utility Programs (1)

Categories of utility programs:

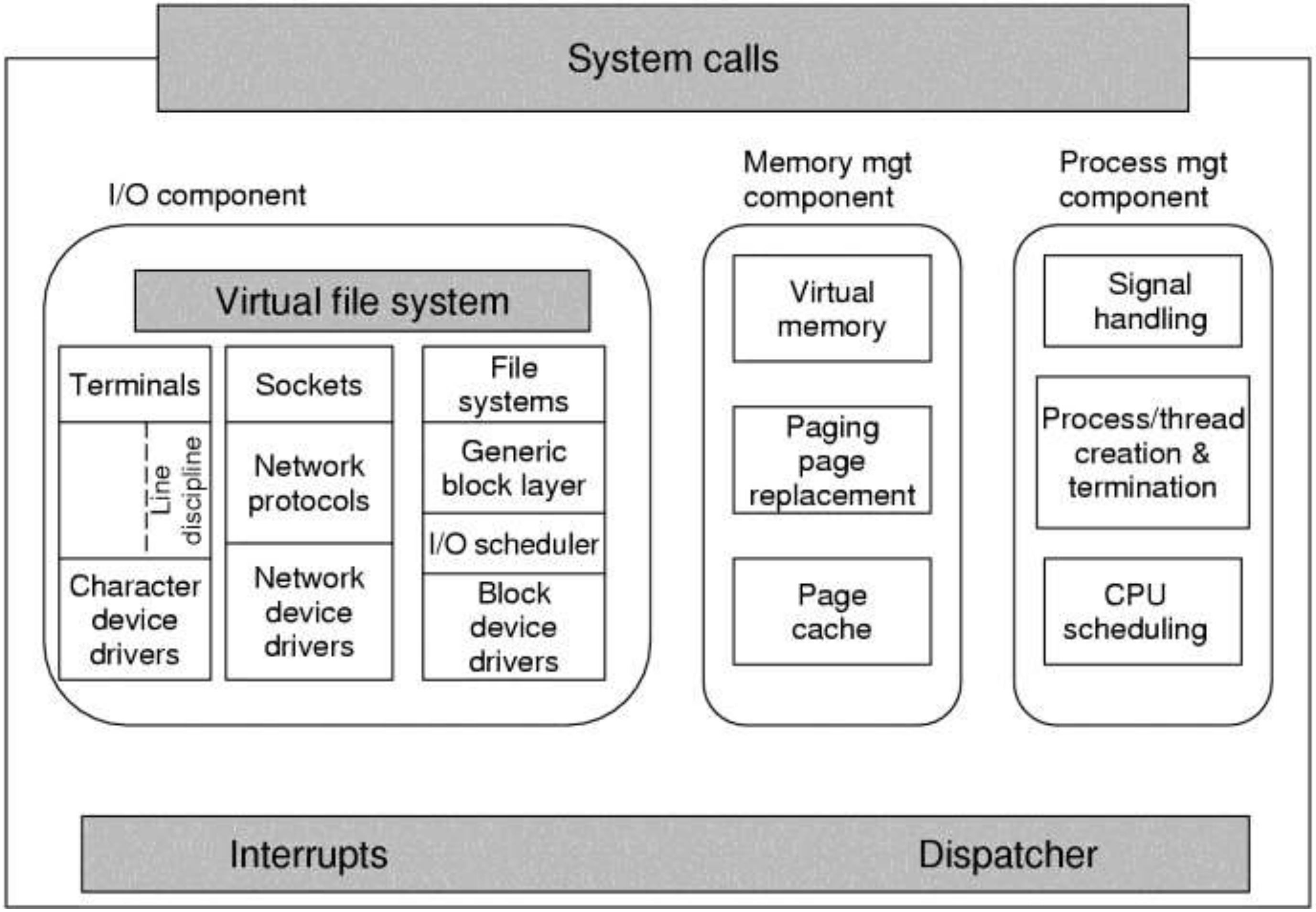
- File and directory manipulation commands.
- Filters.
- Program development tools, such as editors and compilers.
- Text processing.
- System administration.
- Miscellaneous.

# Linux Utility Programs (2)

| Program | Typical use |
|---|---|
| cat | Concatenate multiple files to standard output |
| chmod | Change file protection mode |
| cp | Copy one or more files |
| cut | Cut columns of text from a file |
| grep | Search a file for some pattern |
| head | Extract the first lines of a file |
| ls | List directory |
| make | Compile files to build a binary |
| mkdir | Make a directory |
| od | Octal dump a file |
| paste | Paste columns of text into a file |
| pr | Format a file for printing |
| ps | List running processes |
| rm | Remove one or more files |
| rmdir | Remove a directory |
| sort | Sort a file of lines alphabetically |
| tail | Extract the last lines of a file |
| tr | Translate between character sets |

A few of the common Linux utility programs required by POSIX.

# Kernel Structure




Structure of the Linux kernel

# Processes in Linux

```
pid = fork( );                  /* if the fork succeeds, pid > 0 in the parent */
if (pid < 0) {
        handle_error( );        /* fork failed (e.g., memory or some table is full) */
} else if (pid > 0) {
                                /* parent code goes here. /*/
} else {
                                /* child code goes here. /*/
}
```

Process creation in Linux.

# Signals in Linux (1)

| Signal | Cause |
|---|---|
| SIGABRT | Sent to abort a process and force a core dump |
| SIGALRM | The alarm clock has gone off |
| SIGFPE | A floating-point error has occurred (e.g., division by 0) |
| SIGHUP | The phone line the process was using has been hung up |
| SIGILL | The user has hit the DEL key to interrupt the process |
| SIGQUIT | The user has hit the key requesting a core dump |
| SIGKILL | Sent to kill a process (cannot be caught or ignored) |
| SIGPIPE | The process has written to a pipe which has no readers |
| SIGSEGV | The process has referenced an invalid memory address |
| SIGTERM | Used to request that a process terminate gracefully |
| SIGUSR1 | Available for application-defined purposes |
| SIGUSR2 | Available for application-defined purposes |

The signals required by POSIX.

# Process Management System Calls in Linux

| System call | Description |
| --- | --- |
| pid = fork( ) | Create a child process identical to the parent |
| pid = waitpid(pid, &statloc, opts) | Wait for a child to terminate |
| s = execve(name, argv, envp) | Replace a process' core image |
| exit(status) | Terminate process execution and return status |
| s = sigaction(sig, &act, &oldact) | Define action to take on signals |
| s = sigreturn(&context) | Return from a signal |
| s = sigprocmask(how, &set, &old) | Examine or change the signal mask |
| s = sigpending(set) | Get the set of blocked signals |
| s = sigsuspend(sigmask) | Replace the signal mask and suspend the process |
| s = kill(pid, sig) | Send a signal to a process |
| residual = alarm(seconds) | Set the alarm clock |
| s = pause( ) | Suspend the caller until the next signal |

Some system calls relating to processes.

# A Simple Linux Shell

```
while (TRUE) {                               /* repeat forever /*/
    type_prompt( );                          /* display prompt on the screen */
    read_command(command, params);           /* read input line from keyboard */

    pid = fork( );                           /* fork off a child process */
    if (pid < 0) {
        printf("Unable to fork0);            /* error condition */
        continue;                            /* repeat the loop */
    }

    if (pid != 0) {
        waitpid (-1, &status, 0);            /* parent waits for child */
    } else {
        execve(command, params, 0);          /* child does the work */
    }
}
```

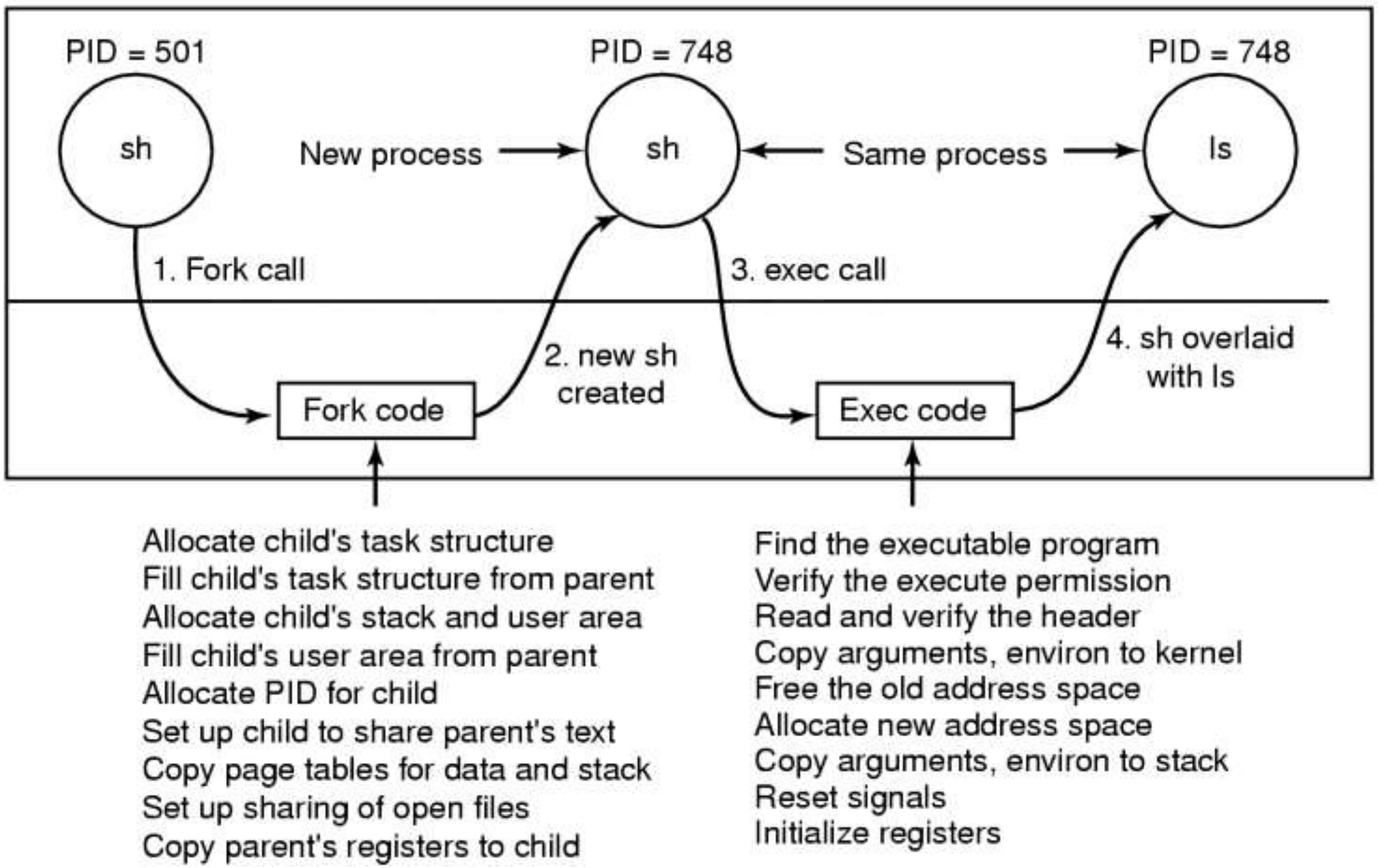
A highly simplified shell.

# Implementation of Processes and Threads

Categories of information in the process descriptor:

- Scheduling parameters
- Memory image
- Signals
- Machine registers
  System call state
- File descriptor table
- Accounting
- Kernel stack
- Miscellaneous

# Implementation of Exec




The steps in executing the command *ls* typed to the shell.

# The Clone System Call

| Flag | Meaning when set | Meaning when cleared |
|---|---|---|
| CLONE_VM | Create a new thread | Create a new process |
| CLONE_FS | Share umask, root, and working dirs | Do not share them |
| CLONE_FILES | Share the file descriptors | Copy the file descriptors |
| CLONE_SIGHAND | Share the signal handler table | Copy the table |
| CLONE_PID | New thread gets old PID | New thread gets own PID |
| CLONE_PARENT | New thread has same parent as caller | New thread's parent is caller |

Bits in the sharing_flags bitmap.

# Scheduling in Linux (1)

Three classes of threads for scheduling purposes:

- Real-time FIFO.
- Real-time round robin.
- Timesharing.

# Scheduling in Linux (2)

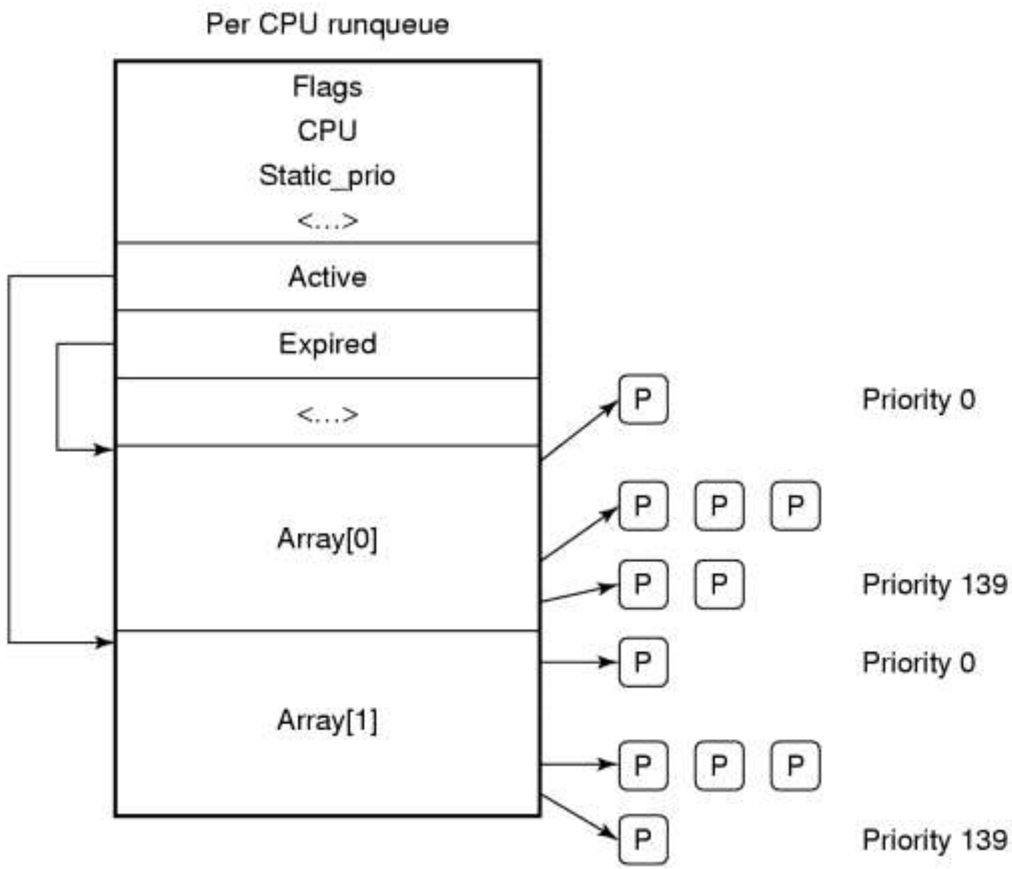
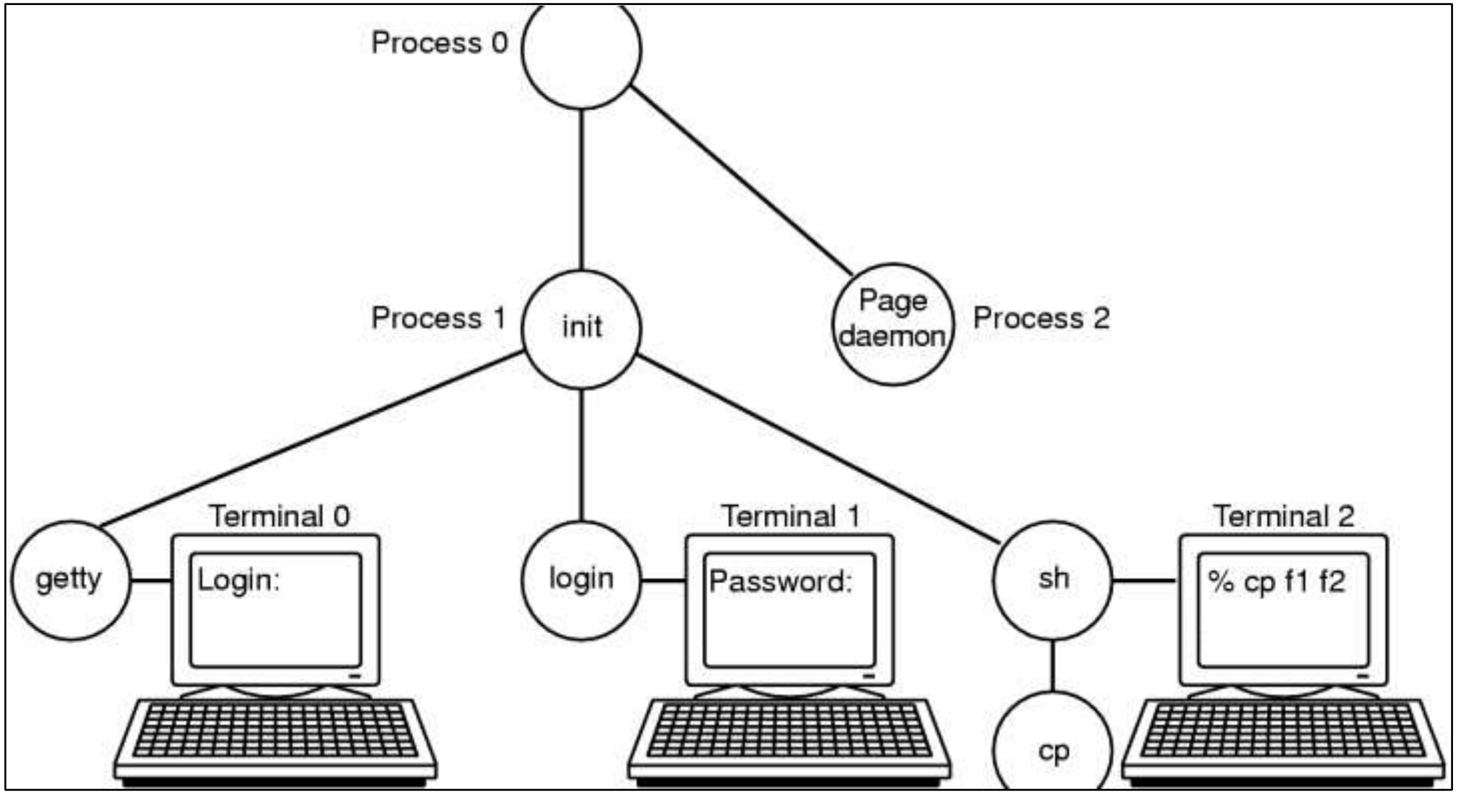


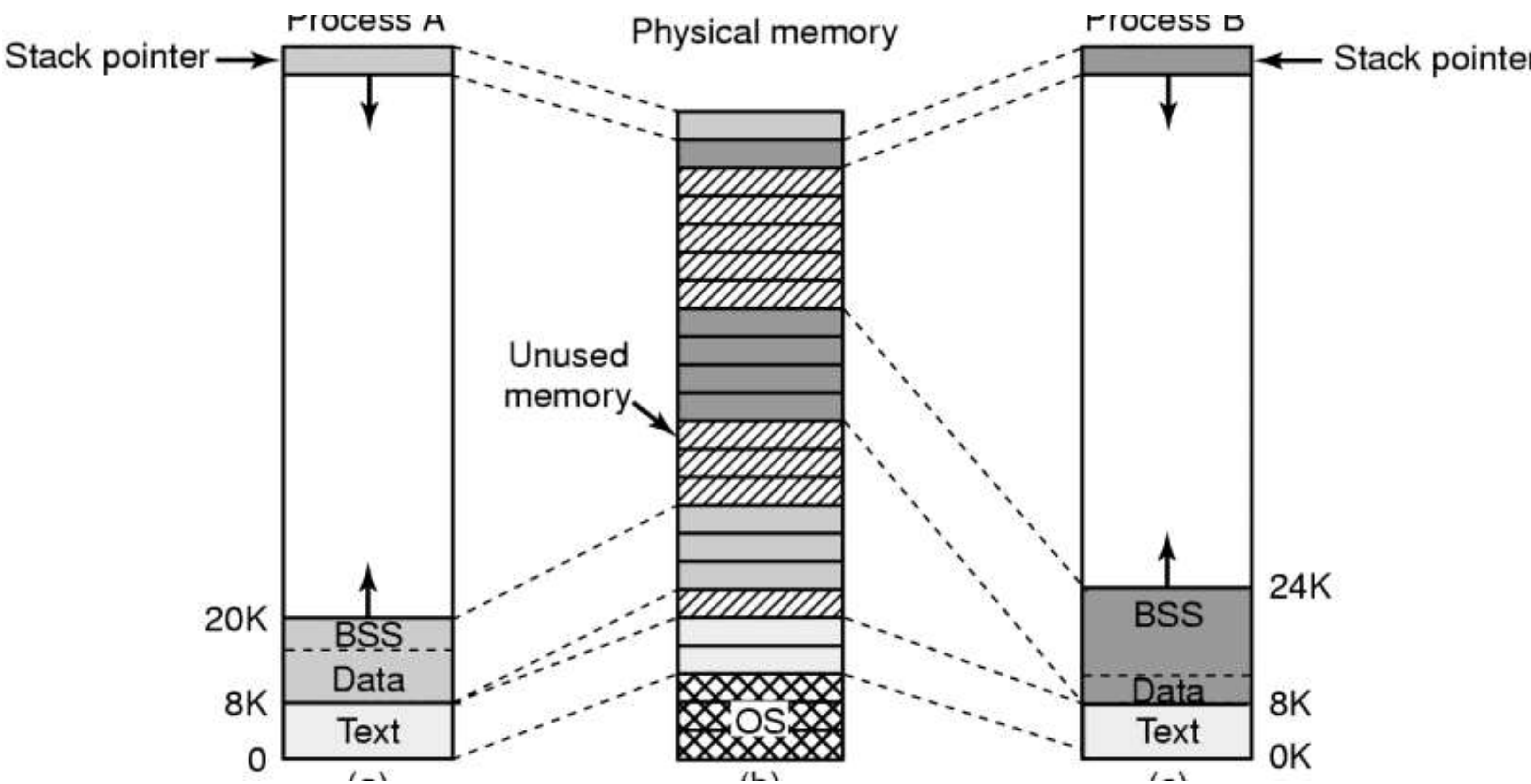

Illustration of Linux runqueue and priority arrays.

# Booting Linux



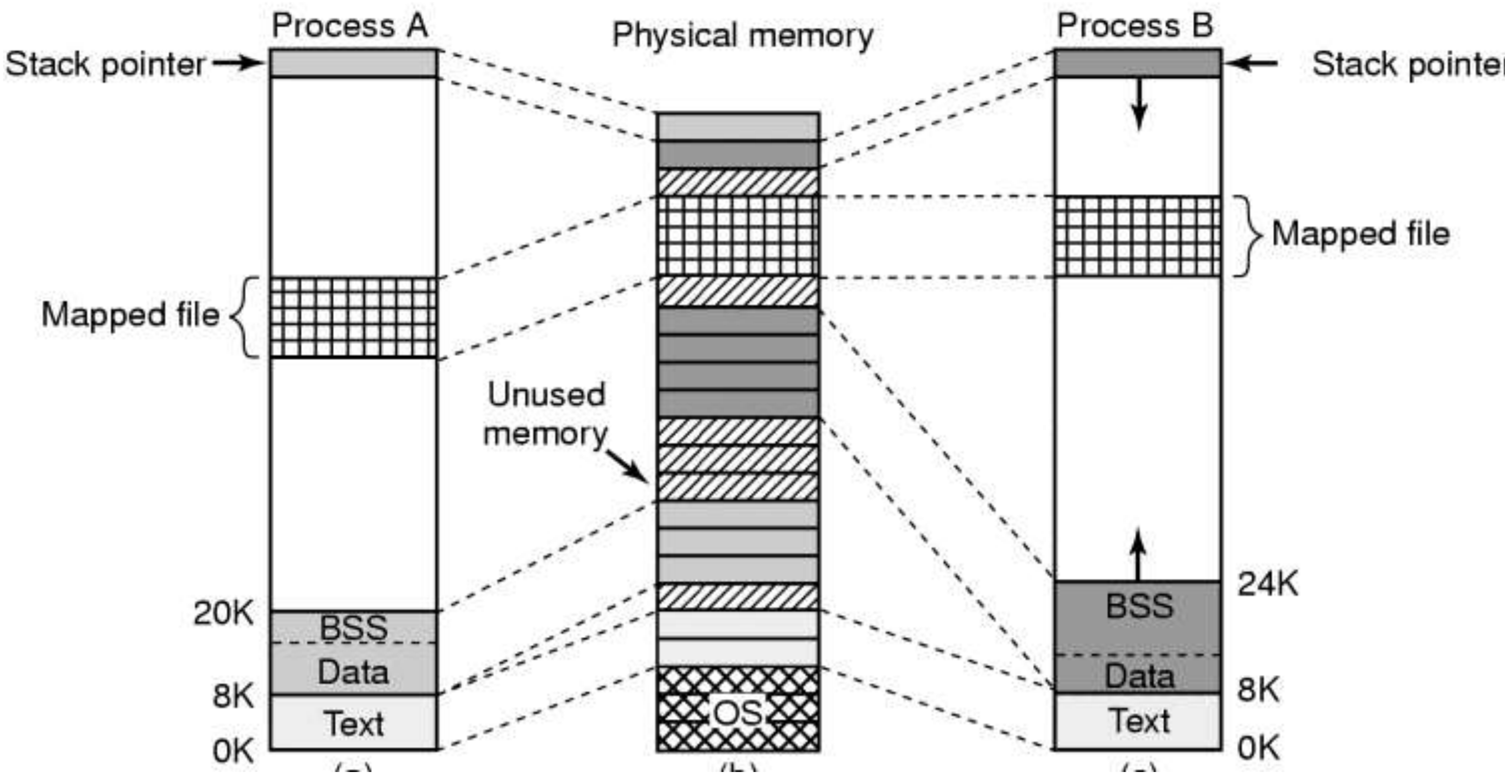

The sequence of processes used to boot some Linux systems.

# Memory Management in Linux (1)




(a) Process A’s virtual address space. (b) Physical memory. (c) Process B’s virtual address space.

# Memory Management in Linux (2)




Two processes can share a mapped file.

# Memory Management System Calls in Linux

| System call | Description |
| --- | --- |
| s = brk(addr) | Change data segment size |
| a = mmap(addr, len, prot, flags, fd, offset) | Map a file in |
| s = unmap(addr, len) | Unmap a file |

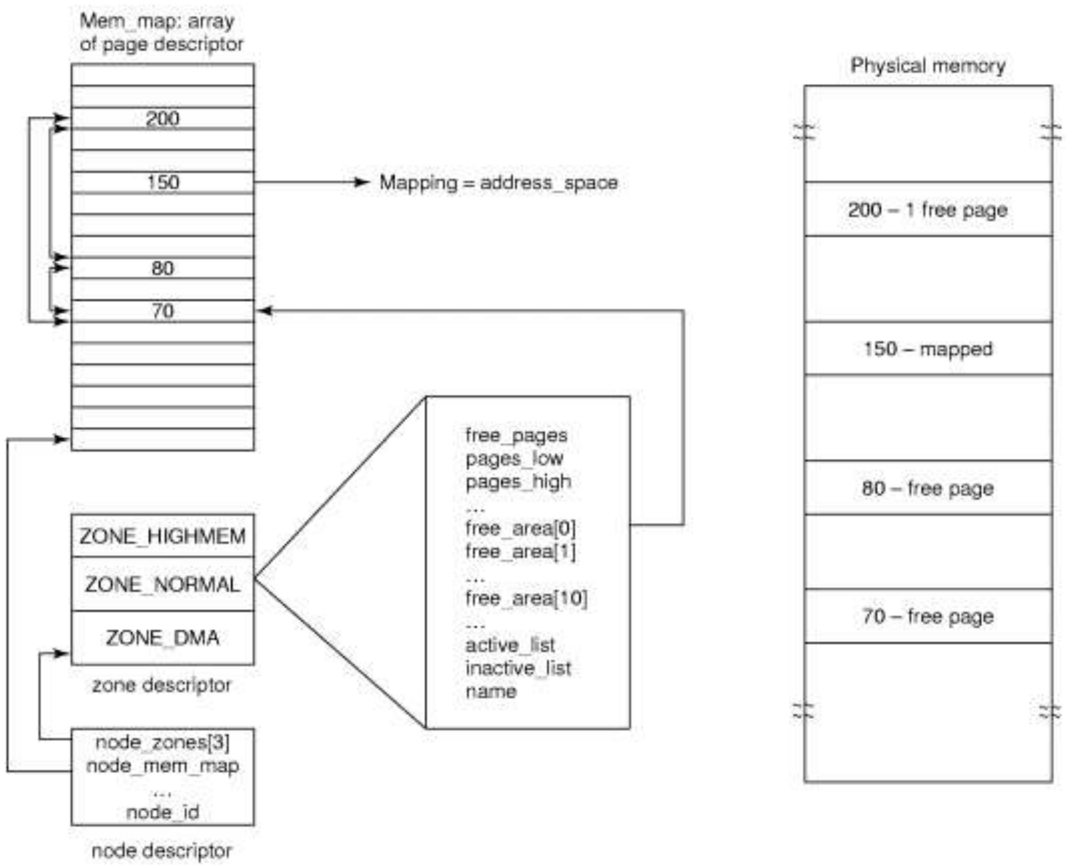
Some system calls relating to memory management.

# Physical Memory Management (1)

Linux distinguishes between three memory zones:

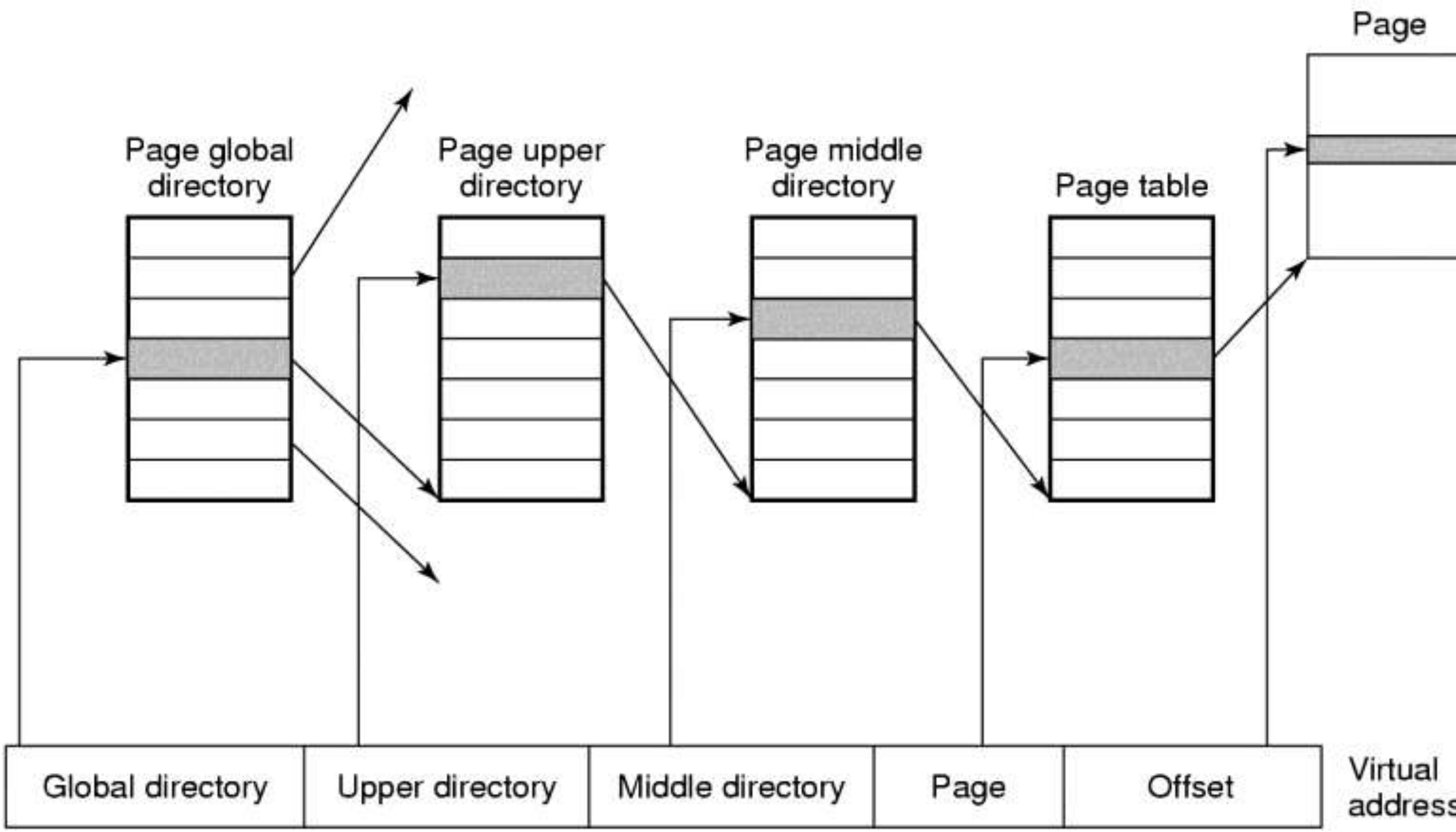
- ZONE_DMA - pages that can be used for DMA operations.
- ZONE_NORMAL - normal, regularly mapped pages.
- ZONE_HIGHMEM - pages with high-memory addresses, which are not permanently mapped.

# Physical Memory Management (2)



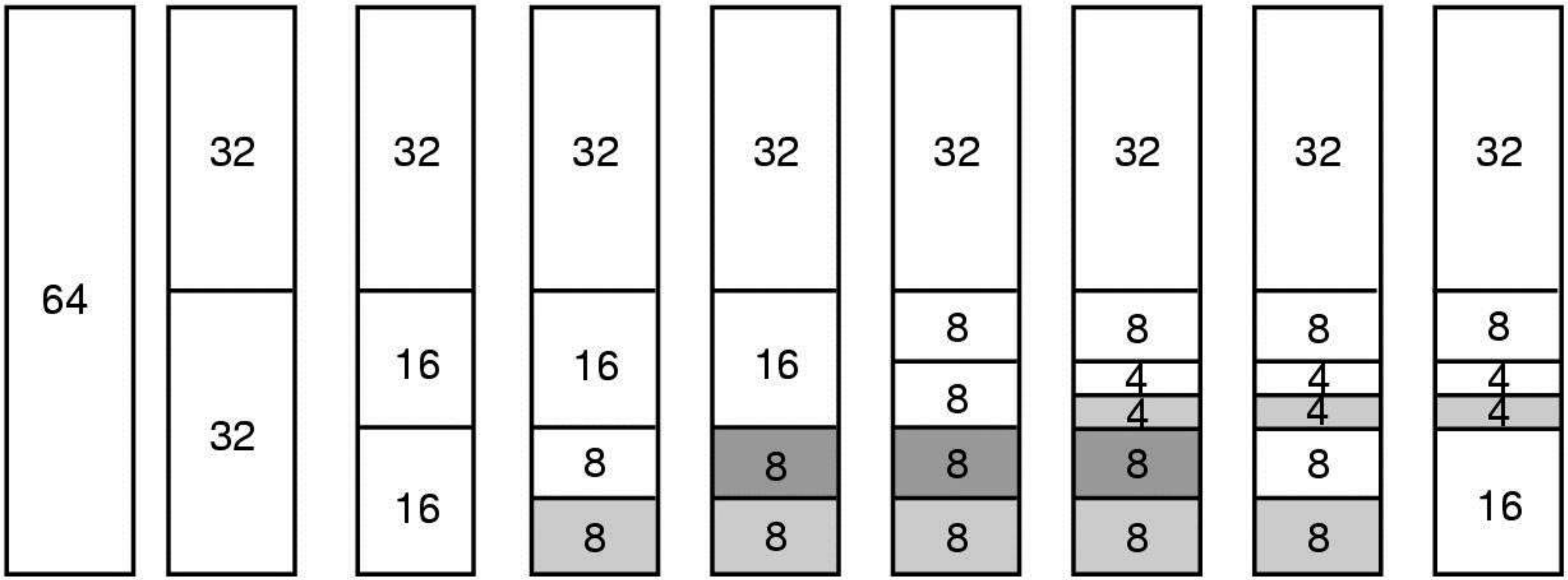

Linux main memory representation.

# Physical Memory Management (3)



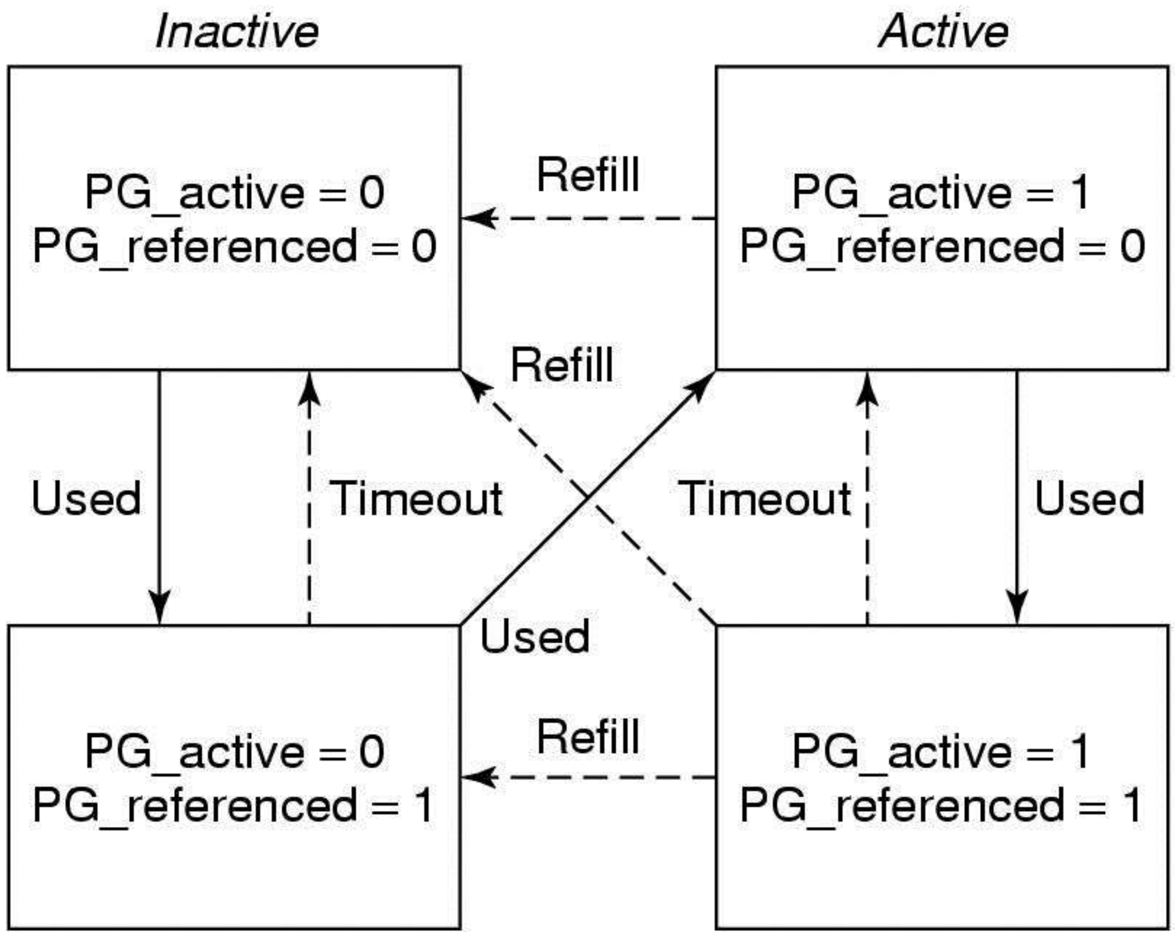

Linux uses four-level page tables.

# Memory Allocation Mechanisms



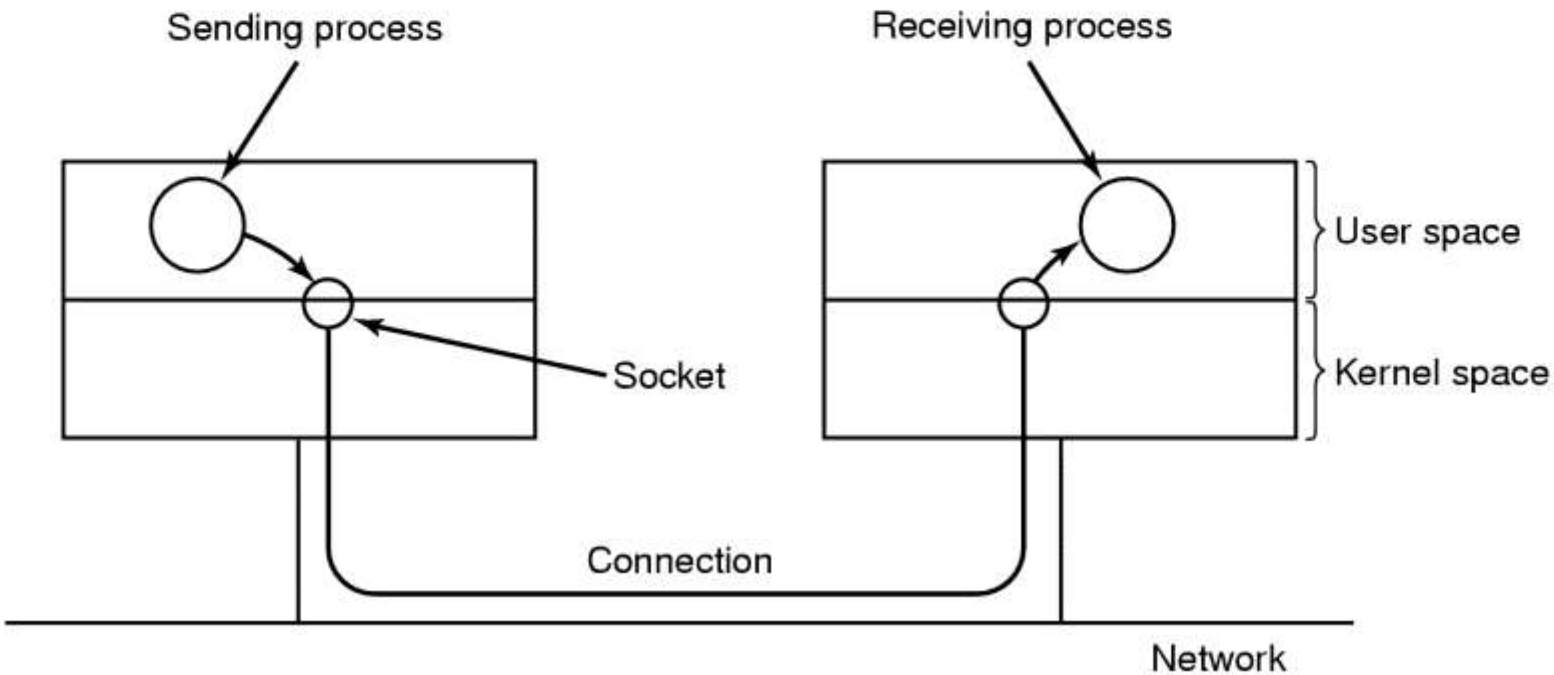

Operation of the buddy algorithm.

# The Page Replacement Algorithm




Page states considered in the page frame replacement algorithm.

# Networking (1)




The uses of sockets for networking.

# Networking (2)

Types of networking:

- Reliable connection-oriented byte stream.
- Reliable connection-oriented packet stream.
- Unreliable packet transmission.

# Input/Output System Calls in Linux

| Function call | Description |
|---|---|
| s = cfsetospeed(&termios, speed) | Set the output speed |
| s = cfsetispeed(&termios, speed) | Set the input speed |
| s = cfgetospeed(&termios, speed) | Get the output speed |
| s = cfgtetispeed(&termios, speed) | Get the input speed |
| s = tcsetattr(fd, opt, &termios) | Set the attributes |
| s = tcgetattr(fd, &termios) | Get the attributes |

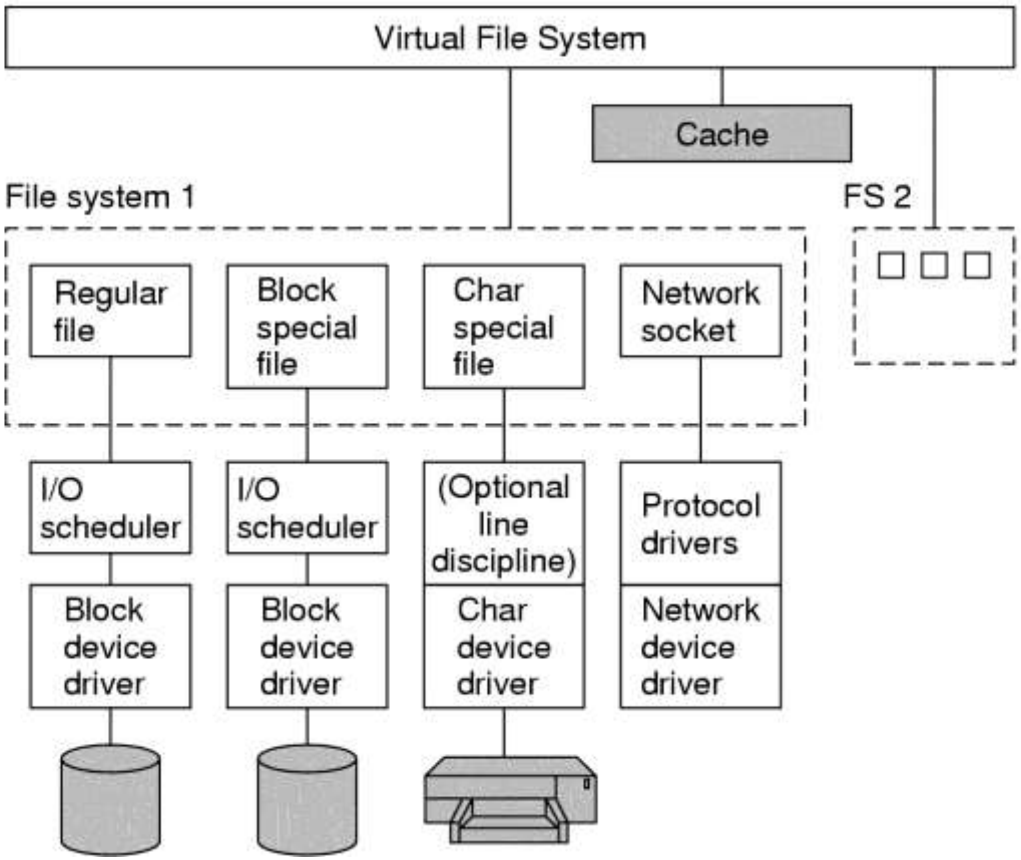
The main POSIX calls for managing the terminal.

# The Major Device Table

| Device | Open | Close | Read | Write | Ioctl | Other |
|---|---|---|---|---|---|---|
| Null | null | null | null | null | null | ... |
| Memory | null | null | mem_read | mem_write | null | ... |
| Keyboard | k_open | k_close | k_read | error | k_ioctl | ... |
| Tty | tty_open | tty_close | tty_read | tty_write | tty_ioctl | ... |
| Printer | lp_open | lp_close | error | lp_write | lp_ioctl | ... |

Some of the file operations supported for typical character devices.

# Implementation of Input/Output in Linux



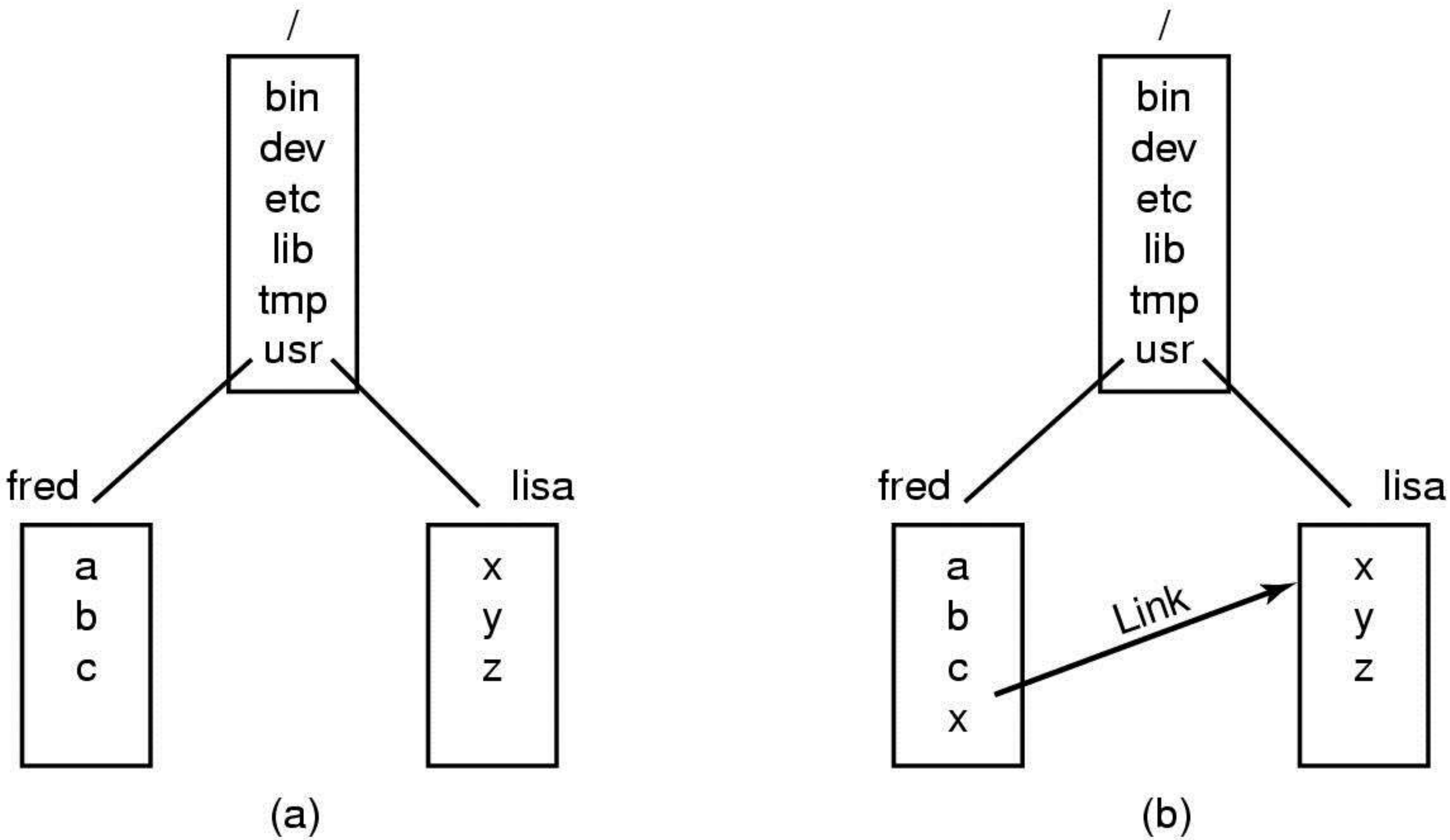

The Linux I/O system showing one file system in detail.

# The Linux File System (1)

| Directory | Contents |
| --- | --- |
| bin | Binary (executable) programs |
| dev | Special files for I/O devices |
| etc | Miscellaneous system files |
| lib | Libraries |
| usr | User directories |

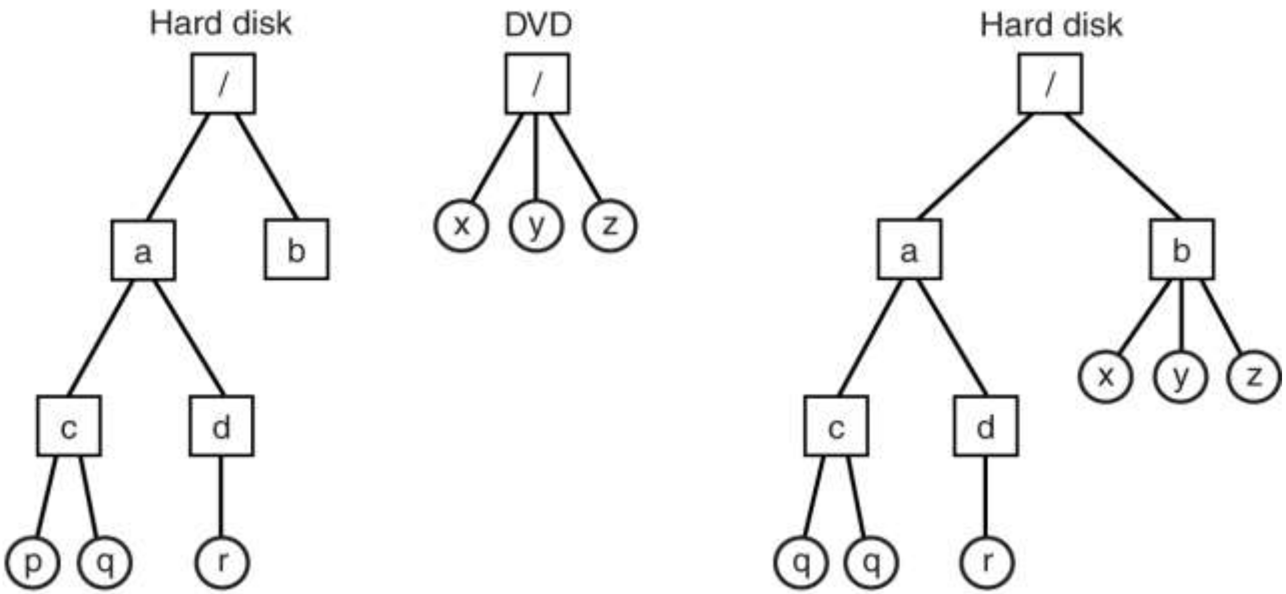
Some important directories found in most Linux systems.

# The Linux File System (2)



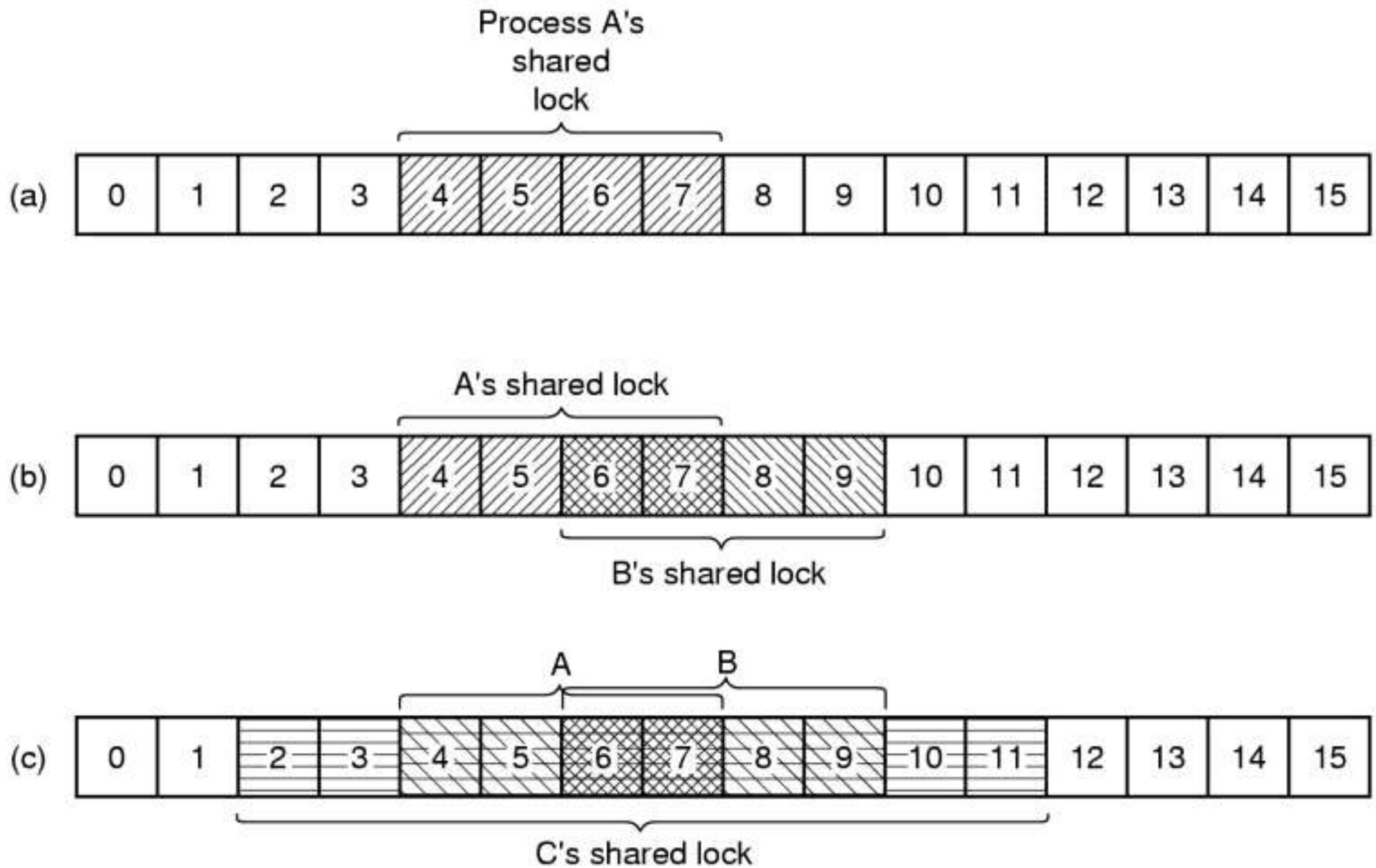

(a) Before linking. (b) After linking.

# The Linux File System (3)




(a) Separate file systems. (b) After mounting.

# The Linux File System (4)




(a) A file with one lock. (b) Addition of a second lock. (c) A third lock.

# File System Calls in Linux (1)

| System call | Description |
|---|---|
| fd = creat(name, mode) | One way to create a new file |
| 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 |
| s = fstat(fd, &buf) | Get a file's status information |
| s = pipe(&fd[0]) | Create a pipe |
| s = fcntl(fd, cmd, ...) | File locking and other operations |

System calls relating to files.

# File System Calls in Linux (2)

| |
|---|
| Device the file is on |
| I-node number (which file on the device) |
| File mode (includes protection information) |
| Number of links to the file |
| Identity of the file's owner |
| Group the file belongs to |
| File size (in bytes) |
| Creation time |
| Time of last access |
| Time of last modification |

The fields returned by the **stat** system call.

# File System Calls in Linux (3)

| System call | Description |
|---|---|
| s = mkdir(path, mode) | Create a new directory |
| s = rmdir(path) | Remove a directory |
| s = link(oldpath, newpath) | Create a link to an existing file |
| s = unlink(path) | Unlink a file |
| s = chdir(path) | Change the working directory |
| dir = opendir(path) | Open a directory for reading |
| s = closedir(dir) | Close a directory |
| dirent = readdir(dir) | Read one directory entry |
| rewinddir(dir) | Rewind a directory so it can be reread |

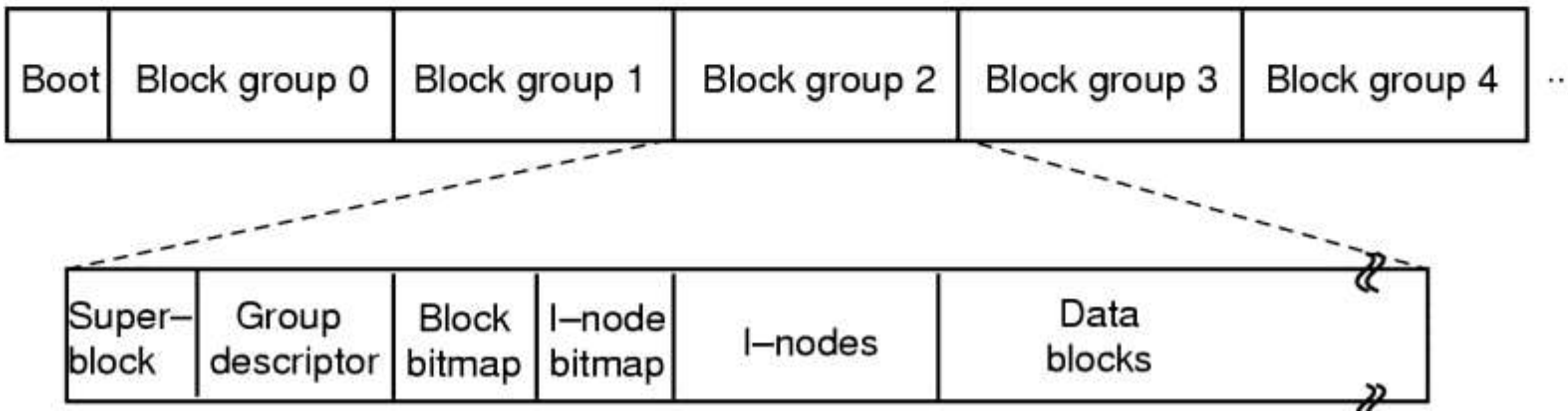
System calls relating to directories.

# The Linux Virtual File System

| Object | Description | Operation |
|---|---|---|
| Superblock | specific filesystem | read_inode, sync_fs |
| Dentry | directory entry, single component of a path | create, link |
| I-node | specific file | d_compare, d_delete |
| File | open file associated with a process | read, write |

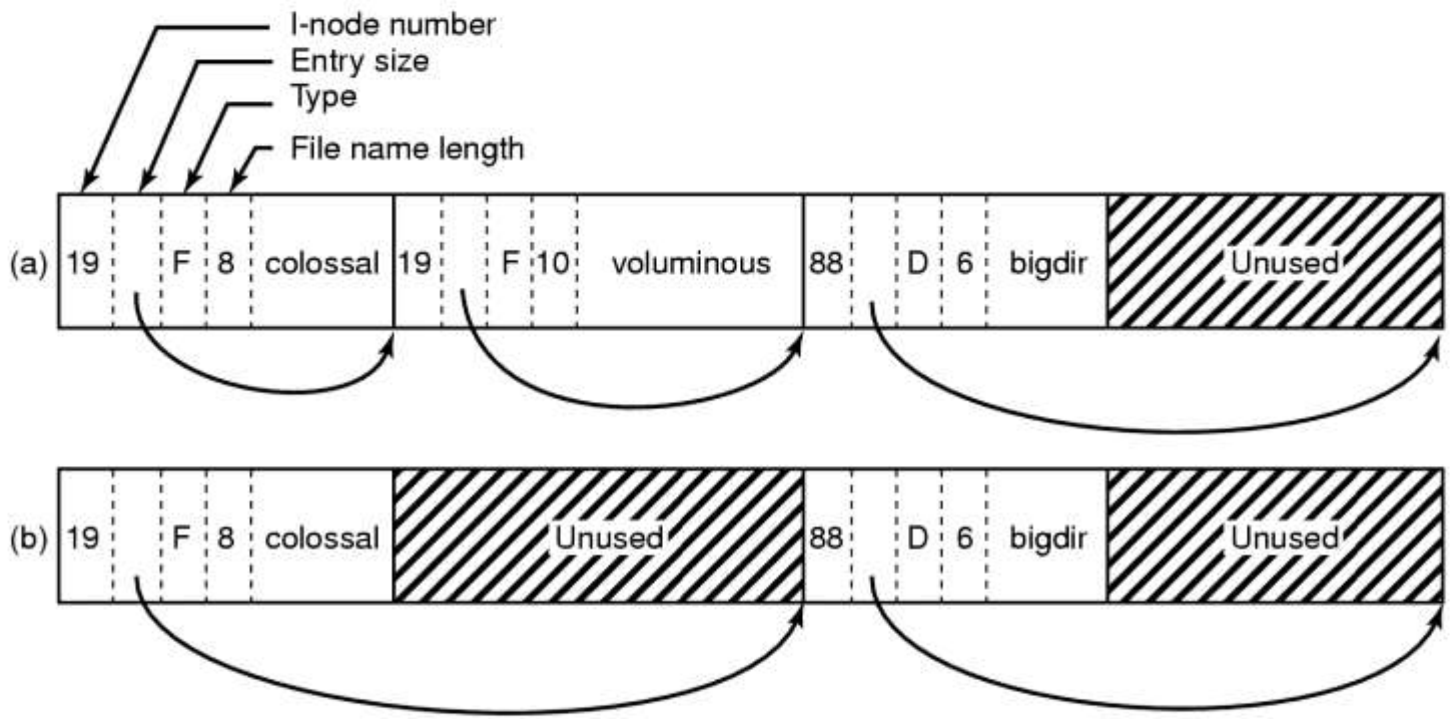
File system abstractions supported by the VFS.

# The Linux Ext2 File System (1)




Disk layout of the Linux ext2 file system.

# The Linux Ext2 File System (2)



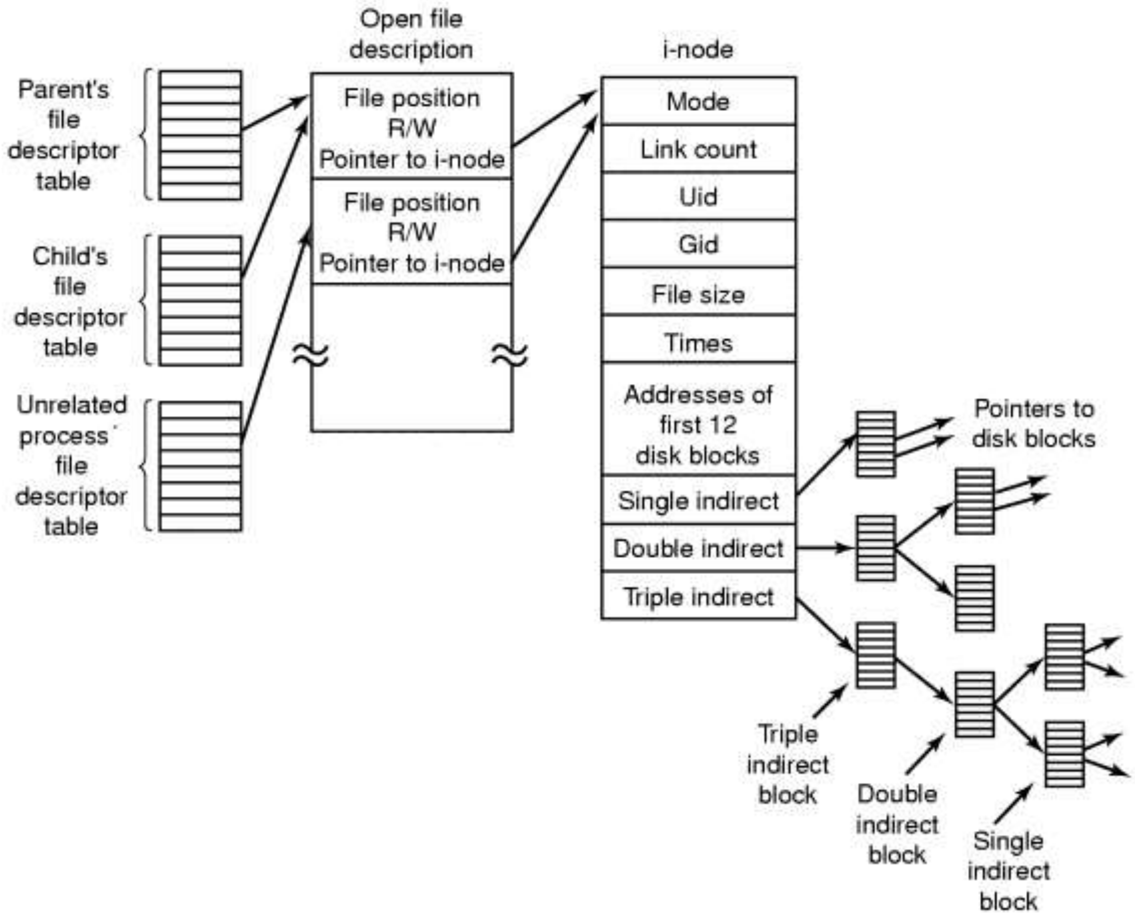

(a) A Linux directory with three files. (b) The same directory after the file voluminous has been removed.

# The Linux Ext2 File System (3)

| Field | Bytes | Description |
|---|---|---|
| Mode | 2 | File type, protection bits, setuid, setgid bits |
| Nlinks | 2 | Number of directory entries pointing to this i-node |
| Uid | 2 | UID of the file owner |
| Gid | 2 | GID of the file owner |
| Size | 4 | File size in bytes |
| Addr | 60 | Address of first 12 disk blocks, then 3 indirect blocks |
| Gen | 1 | Generation number (incremented every time i-node is reused) |
| Atime | 4 | Time the file was last accessed |
| Mtime | 4 | Time the file was last modified |
| Ctime | 4 | Time the i-node was last changed (except the other times) |

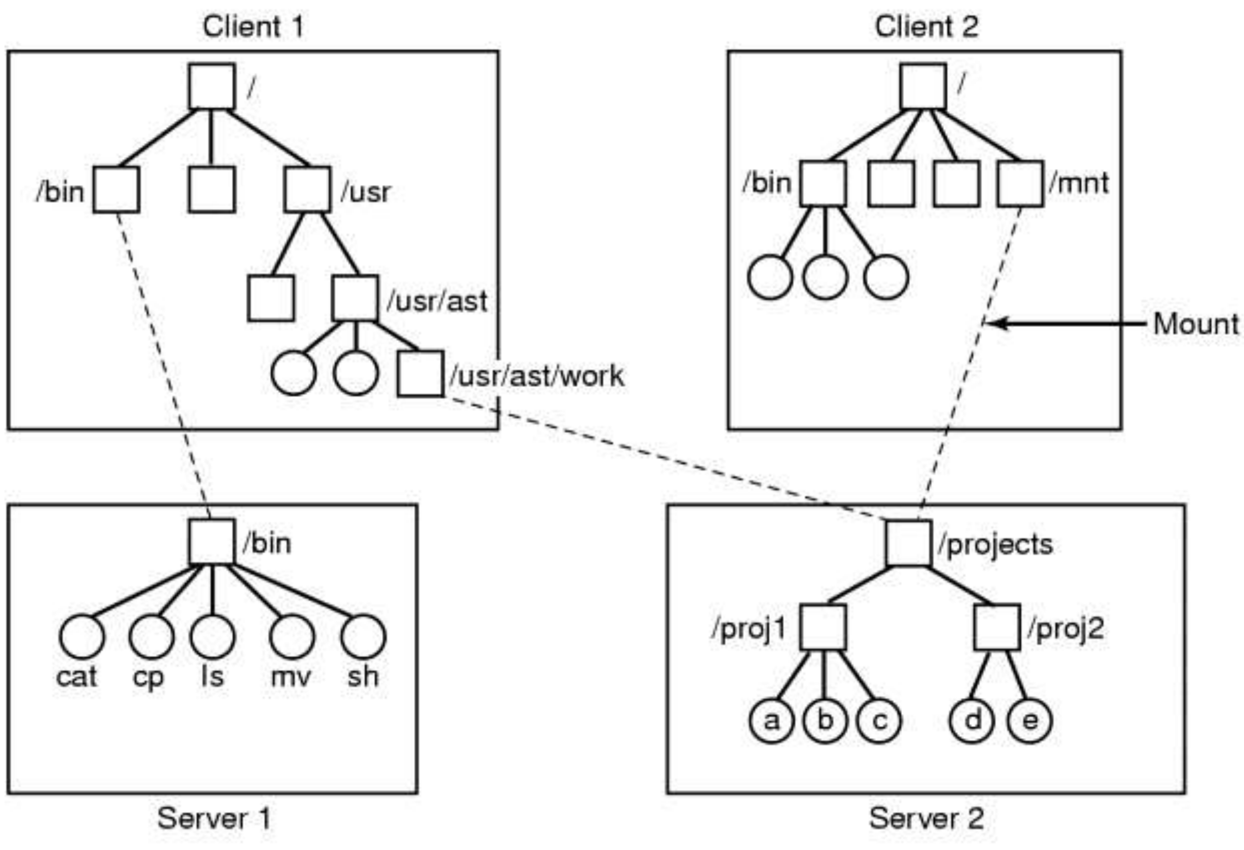
Some fields in the i-node structure in Linux

# The Linux Ext2 File System (4)



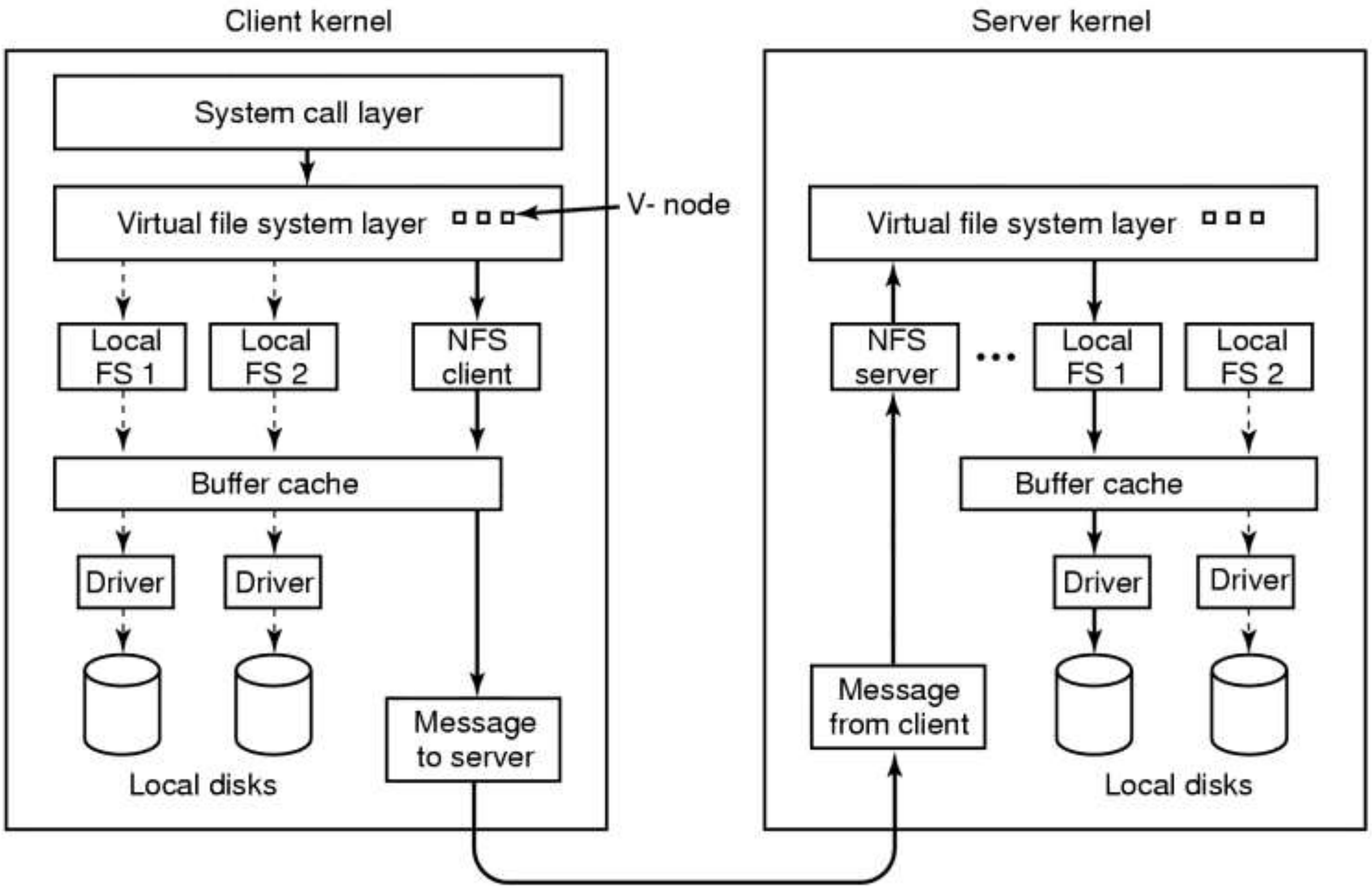

The relation between the file descriptor table, the open file description table, and the i-node table.

# NFS Protocols




Examples of remote mounted file systems. Directories shown as squares, files shown as circles.

# NFS Implementation




The NFS layer structure

# Security In Linux

| Binary | Symbolic | Allowed file accesses |
|---|---|---|
| 111000000 | rwx–––––– | Owner can read, write, and execute |
| 111111000 | rwxrwx––– | Owner and group can read, write, and execute |
| 110100000 | rw–r––––– | Owner can read and write; group can read |
| 110100100 | rw–r––r–– | Owner can read and write; all others can read |
| 111101101 | rwxr–xr–x | Owner can do everything, rest can read and execute |
| 000000000 | ––––––––– | Nobody has any access |
| 000000111 | ––––––rwx | Only outsiders have access (strange, but legal) |

Some example file protection modes.

# Security System Calls in Linux

| System call | Description |
|---|---|
| s = chmod(path, mode) | Change a file's protection mode |
| s = access(path, mode) | Check access using the real UID and GID |
| uid = getuid( ) | Get the real UID |
| uid = geteuid( ) | Get the effective UID |
| gid = getgid( ) | Get the real GID |
| gid = getegid( ) | Get the effective GID |
| s = chown(path, owner, group) | Change owner and group |
| s = setuid(uid) | Set the UID |
| s = setgid(gid) | Set the GID |

system calls relating to security.

# Case Study 2: Windows 8

# History of Windows through Windows 8.1

| Year | MS–DOS | MS-DOS based Windows | NT-based Windows | Modern Windows | Notes |
|---|---|---|---|---|---|
| 1981 | 1.0 | | | | Initial release for IBM PC |
| 1983 | 2.0 | | | | Support for PC/XT |
| 1984 | 3.0 | | | | Support for PC/AT |
| 1990 | | 3.0 | | | Ten million copies in 2 years |
| 1991 | 5.0 | | | | Added memory management |
| 1992 | | 3.1 | | | Ran only on 286 and later |
| 1993 | | | NT 3.1 | | |
| 1995 | 7.0 | 95 | | | MS-DOS embedded in Win 95 |
| 1996 | | | NT 4.0 | | |
| 1998 | | 98 | | | |
| 2000 | 8.0 | Me | 2000 | | Win Me was inferior to Win 98 |
| 2001 | | | XP | | Replaced Win 98 |
| 2006 | | | Vista | | Vista could not supplant XP |
| 2009 | | | 7 | | Significantly improved upon Vista |
| 2012 | | | | 8 | First Modern version |
| 2013 | | | | 8.1 | Microsoft moved to rapid releases |

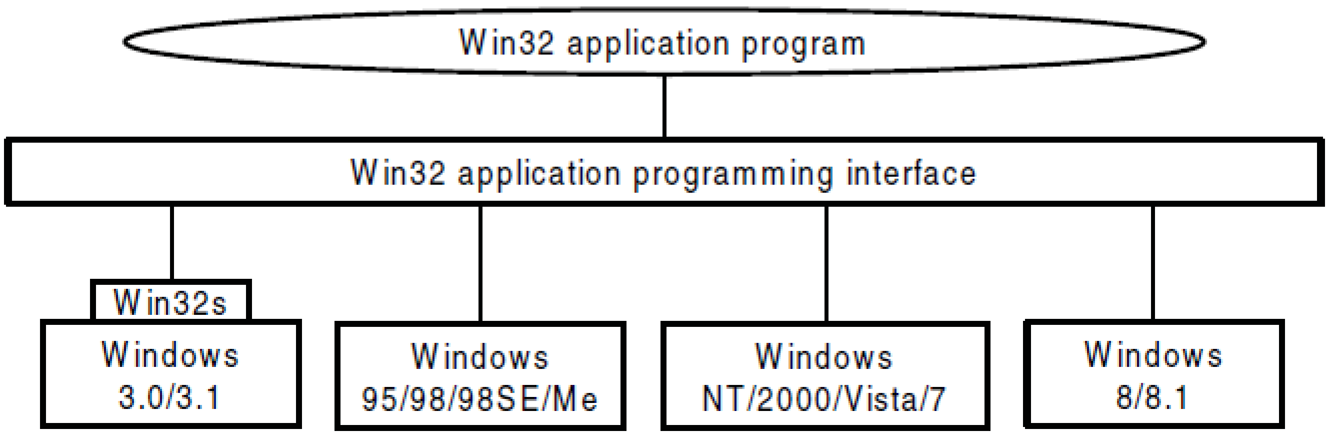
Major releases in the history of Microsoft operating systems for desktop PCs.

# 2000s: NT-based Windows (1)

| Year | DEC operating system | Characteristics |
|---|---|---|
| 1973 | RSX-11M | 16-bit, multiuser, real-time, swapping |
| 1978 | VAX/VMS | 32-bit, virtual memory |
| 1987 | VAXELAN | Real-time |
| 1988 | PRISM/Mica | Canceled in favor of MIPS/Ultrix |

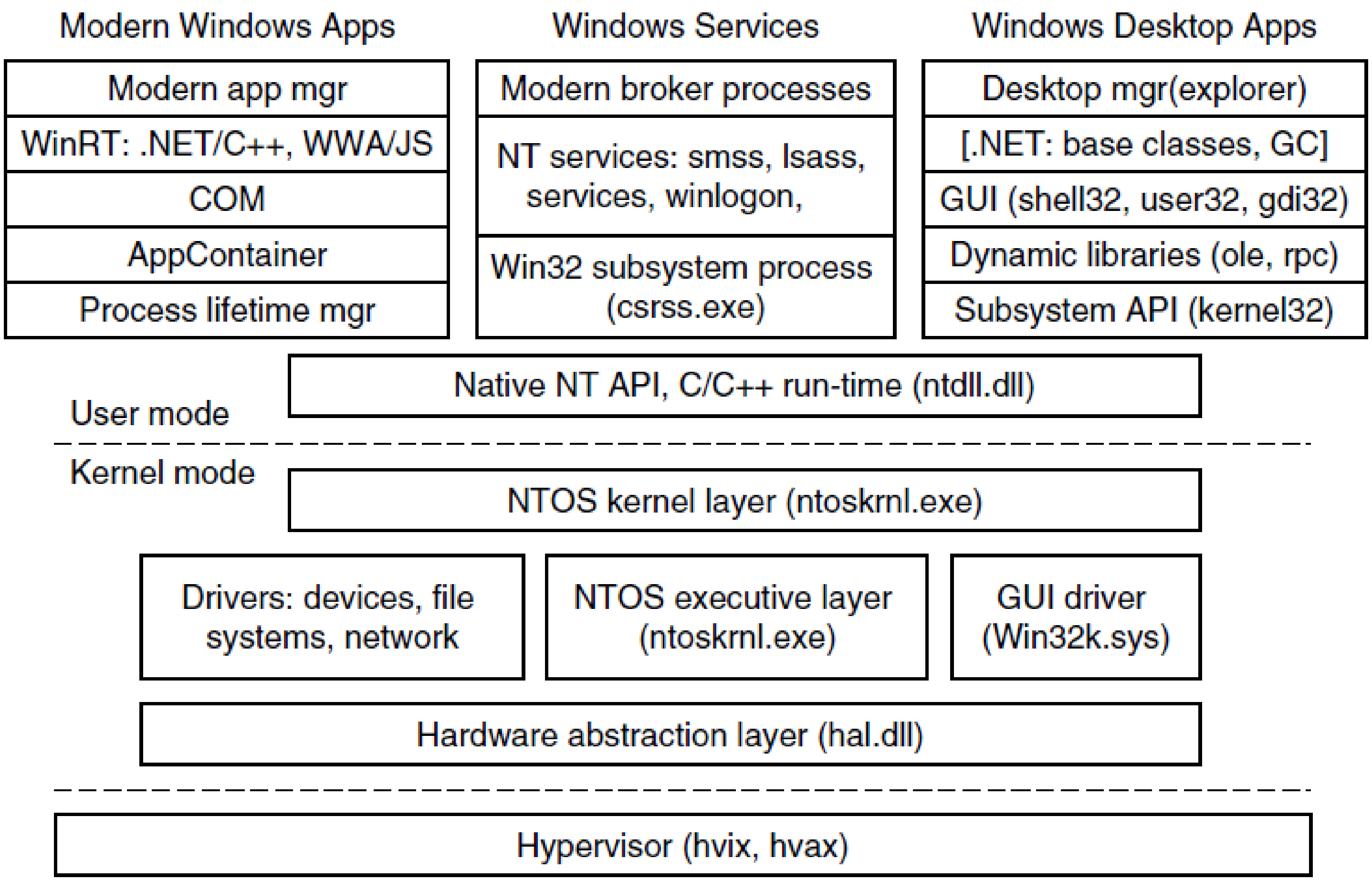
DEC Operating Systems developed by Dave Cutler.

# 2000s: NT-based Windows (2)



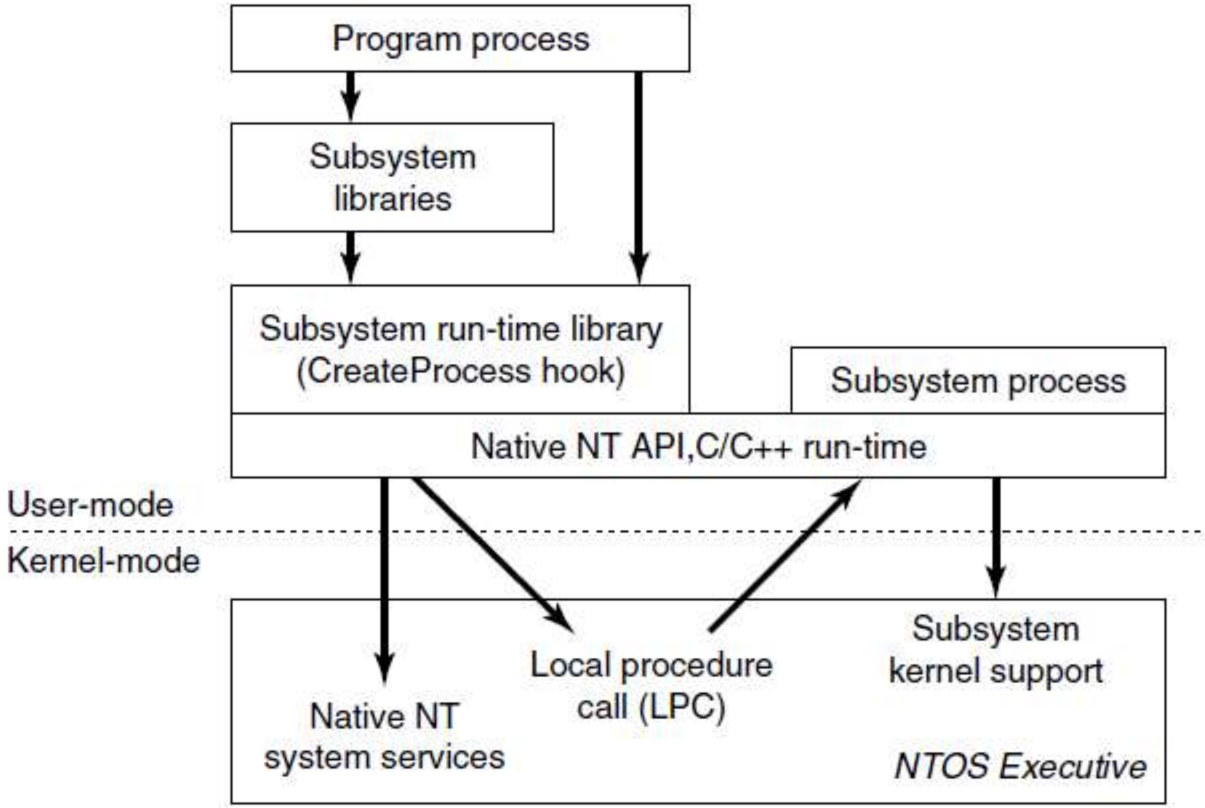

The Win32 API allows programs to run on almost all versions of Windows.

# Programming Windows (1)




The programming layers in Modern Windows

# Programming Windows (2)




The components used to build NT subsystems.

# The Native NT Application Programming Interface (1)

| Object category | Examples |
|---|---|
| Synchronization | Semaphores, mutexes, events, IPC ports, I/O completion queues |
| I/O | Files, devices, drivers, timers |
| Program | Jobs, processes, threads, sections, tokens |
| Win32 GUI | Desktops, application callbacks |

Common categories of kernel-mode object types.

# The Native NT Application Programming Interface (2)

| |
|---|
| NtCreateProcess(&ProcHandle, Access, SectionHandle, DebugPortHandle, ExceptPortHandle, ...) |
| NtCreateThread(&ThreadHandle, ProcHandle, Access, ThreadContext, CreateSuspended, ...) |
| NtAllocateVirtualMemory(ProcHandle, Addr, Size, Type, Protection, ...) |
| NtMapViewOfSection(SectHandle, ProcHandle, Addr, Size, Protection, ...) |
| NtReadVirtualMemory(ProcHandle, Addr, Size, ...) |
| NtWriteVirtualMemory(ProcHandle, Addr, Size, ...) |
| NtCreateFile(&FileHandle, FileNameDescriptor, Access, ...) |
| NtDuplicateObject(srcProcHandle, srcObjHandle, dstProcHandle, dstObjHandle, ...) |

Examples of native NT API calls that use handles to manipulate objects across process boundaries.

# The Win32 Application Programming Interface

| Win32 call | Native NT API call |
|---|---|
| CreateProcess | NtCreateProcess |
| CreateThread | NtCreateThread |
| SuspendThread | NtSuspendThread |
| CreateSemaphore | NtCreateSemaphore |
| ReadFile | NtReadFile |
| DeleteFile | NtSetInformationFile |
| CreateFileMapping | NtCreateSection |
| VirtualAlloc | NtAllocateVirtualMemory |
| MapViewOfFile | NtMapViewOfSection |
| DuplicateHandle | NtDuplicateObject |
| CloseHandle | NtClose |

Examples of Win32 API calls and the native NT API calls that they wrap.

# The Windows Registry (1)

| Hive file | Mounted name | Use |
|---|---|---|
| SYSTEM | HKLM \SYSTEM | OS configuration information, used by kernel |
| HARDWARE | HKLM \HARDWARE | In-memory hive recording hardware detected |
| BCD | HKLM \BCD* | Boot Configuration Database |
| SAM | HKLM \SAM | Local user account information |
| SECURITY | HKLM \SECURITY | lsass' account and other security information |
| DEFAULT | HKEY_USERS \.DEFAULT | Default hive for new users |
| NTUSER.DAT | HKEY_USERS \<user id> | User-specific hive, kept in home directory |
| SOFTWARE | HKLM \SOFTWARE | Application classes registered by COM |
| COMPONENTS | HKLM \COMPONENTS | Manifests and dependencies for sys. components |

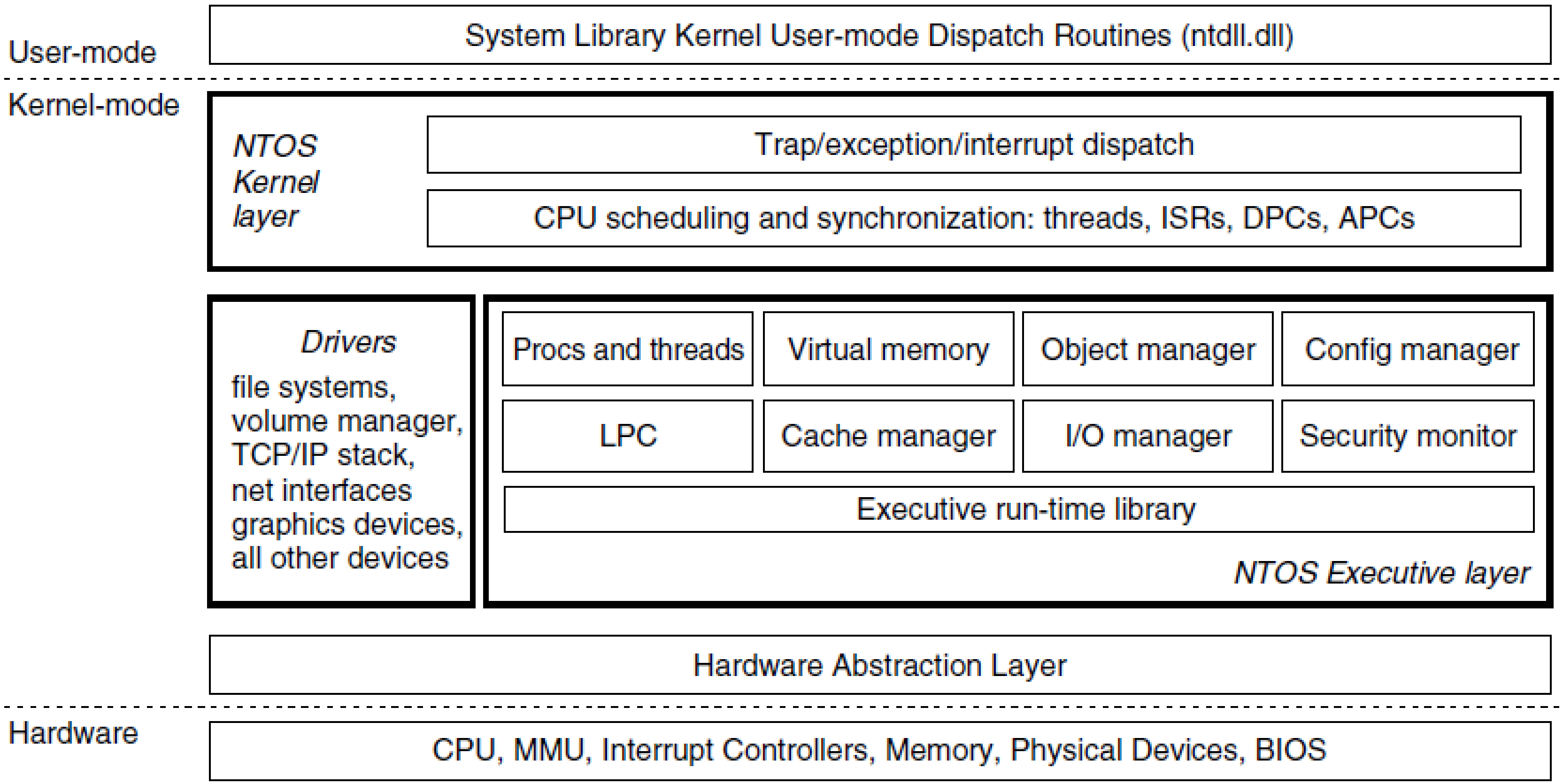
The registry hives in Windows. HKLM is a short-hand for *HKEY LOCAL MACHINE.*

# The Windows Registry (2)

| Win32 API function | Description |
|---|---|
| RegCreateKeyEx | Create a new registry key |
| RegDeleteKey | Delete a registry key |
| RegOpenKeyEx | Open a key to get a handle to it |
| RegEnumKeyEx | Enumerate the subkeys subordinate to the key of the handle |
| RegQueryValueEx | Look up the data for a value within a key |

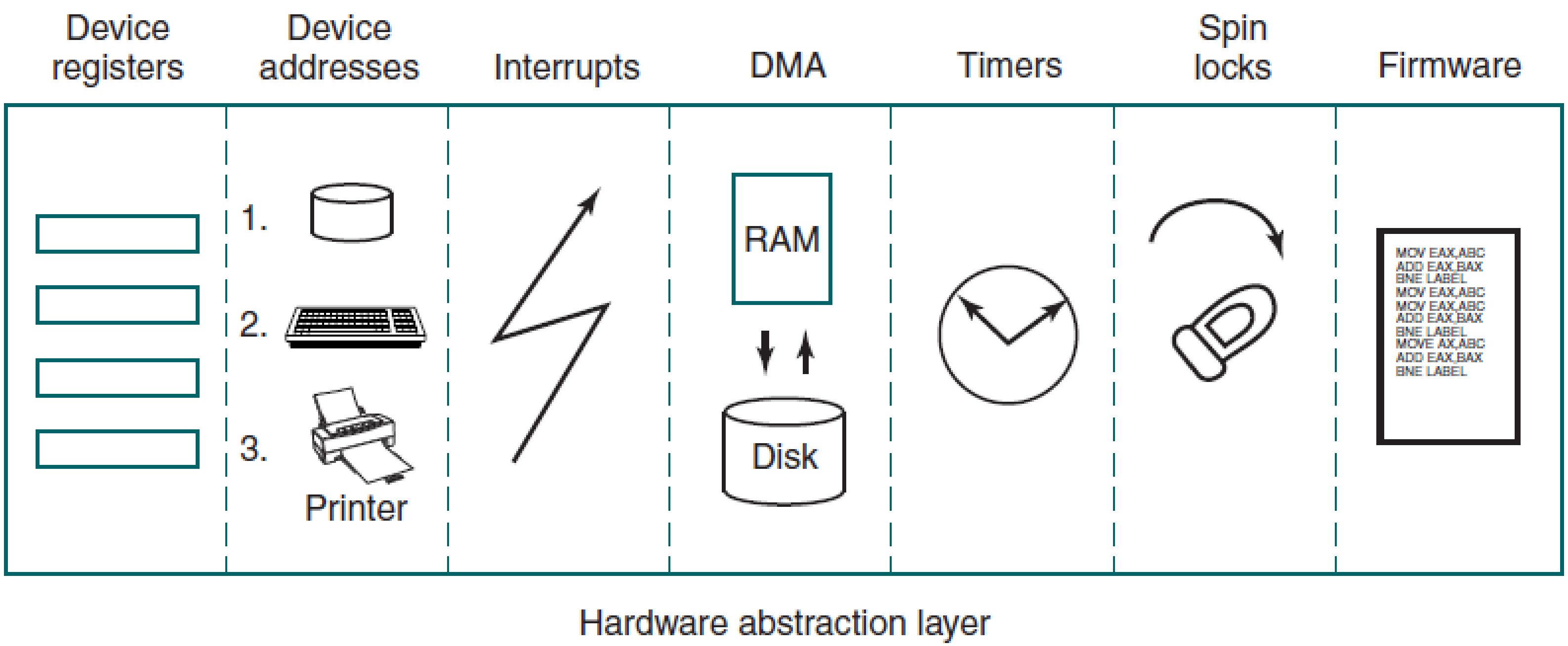
Some of the Win32 API calls for using the registry

# Operating System Structure




Windows kernel-mode organization.

# The Hardware Abstraction Layer

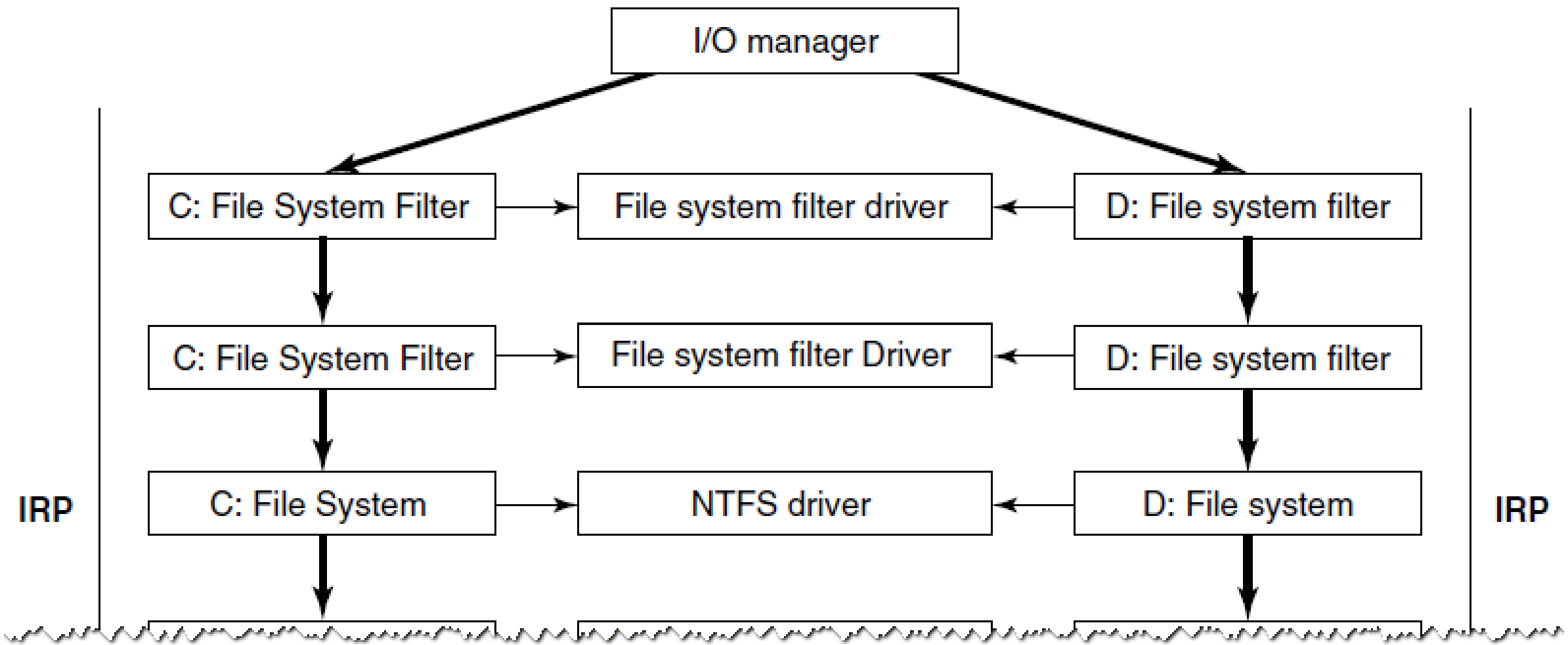



Some of the hardware functions the HAL manages

# Dispatcher Objects

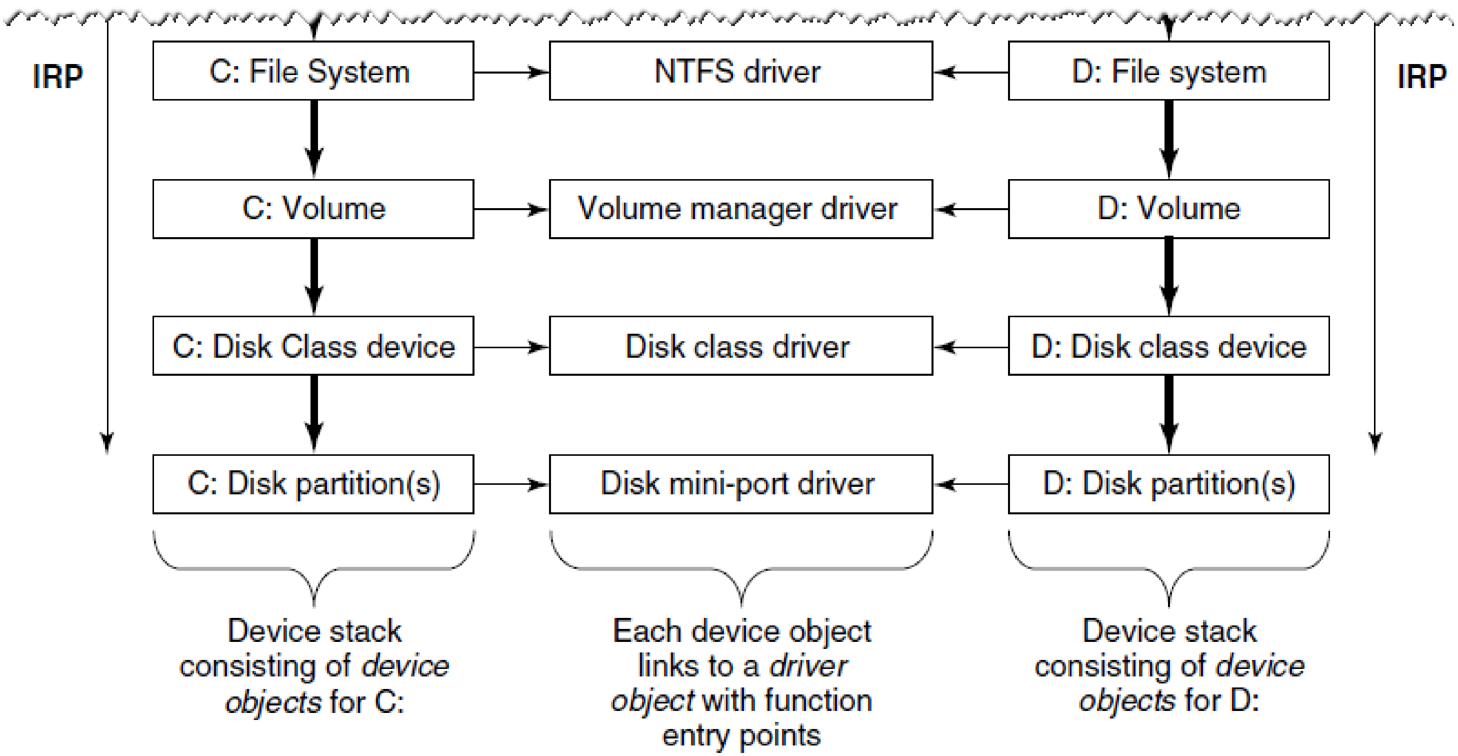

*dispatcher_header* data structure embedded in many executive objects (*dispatcher objects*).

# The Device Drivers (1)



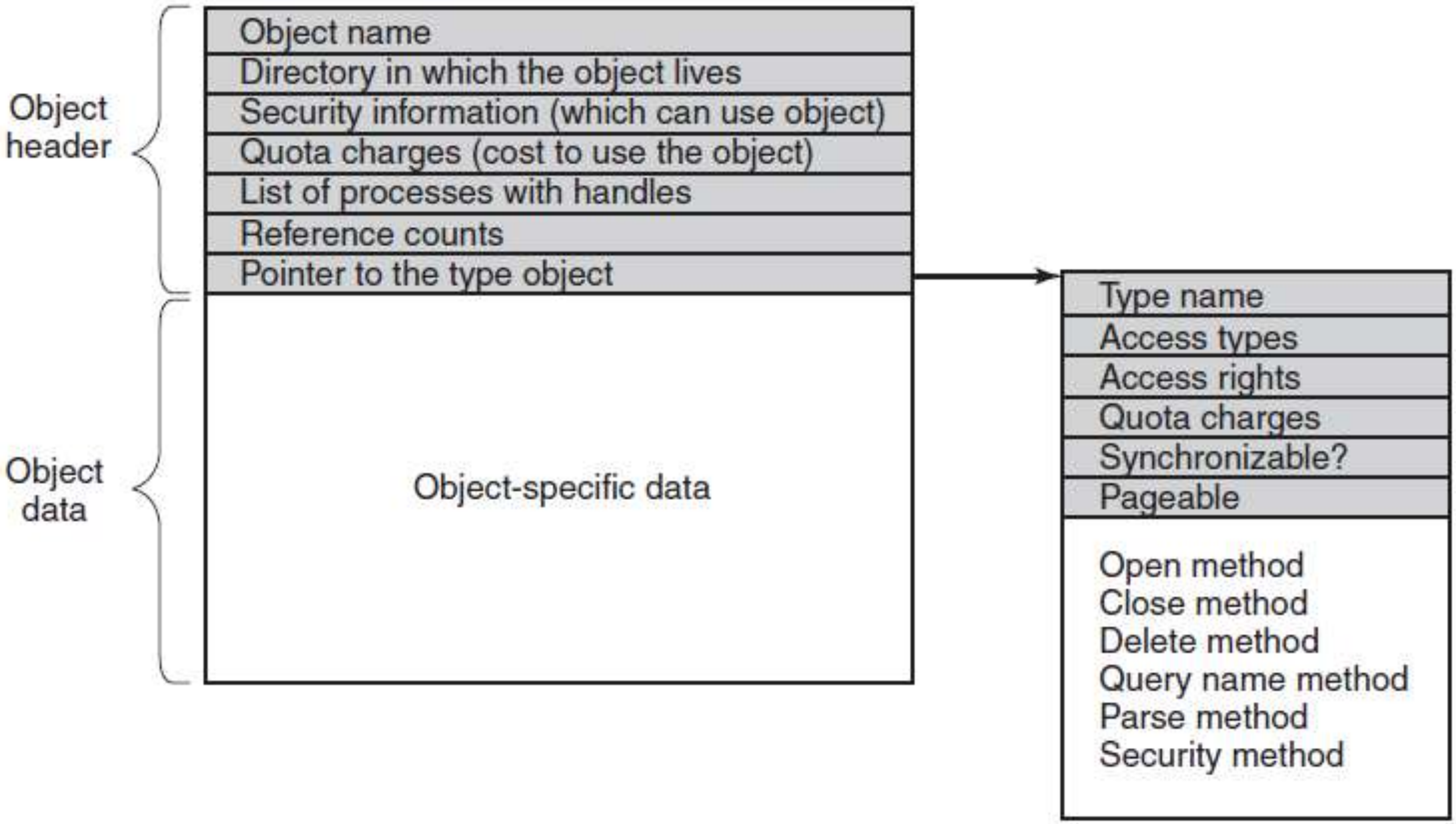

Simplified depiction of device stacks for two NTFS file volumes. The I/O request packet is passed from down the stack. The appropriate routines from the associated drivers are called at each level in the stack. The device stacks themselves consist of device objects allocated specifically to each stack.

# The Device Drivers (2)



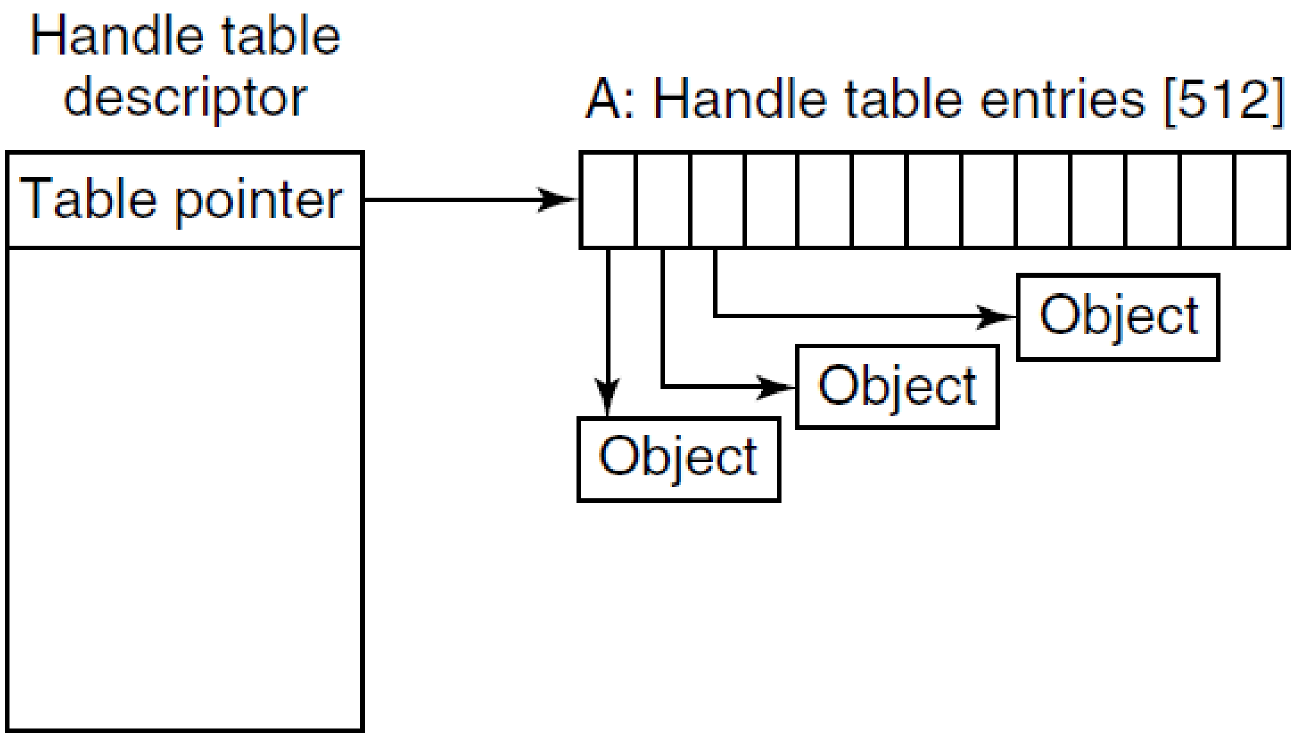

Simplified depiction of device stacks for two NTFS file volumes. The I/O request packet is passed from down the stack. The appropriate routines from the associated drivers are called at each level in the stack. The device stacks themselves consist of device objects allocated specifically to each stack.

# Implementation of the Object Manager



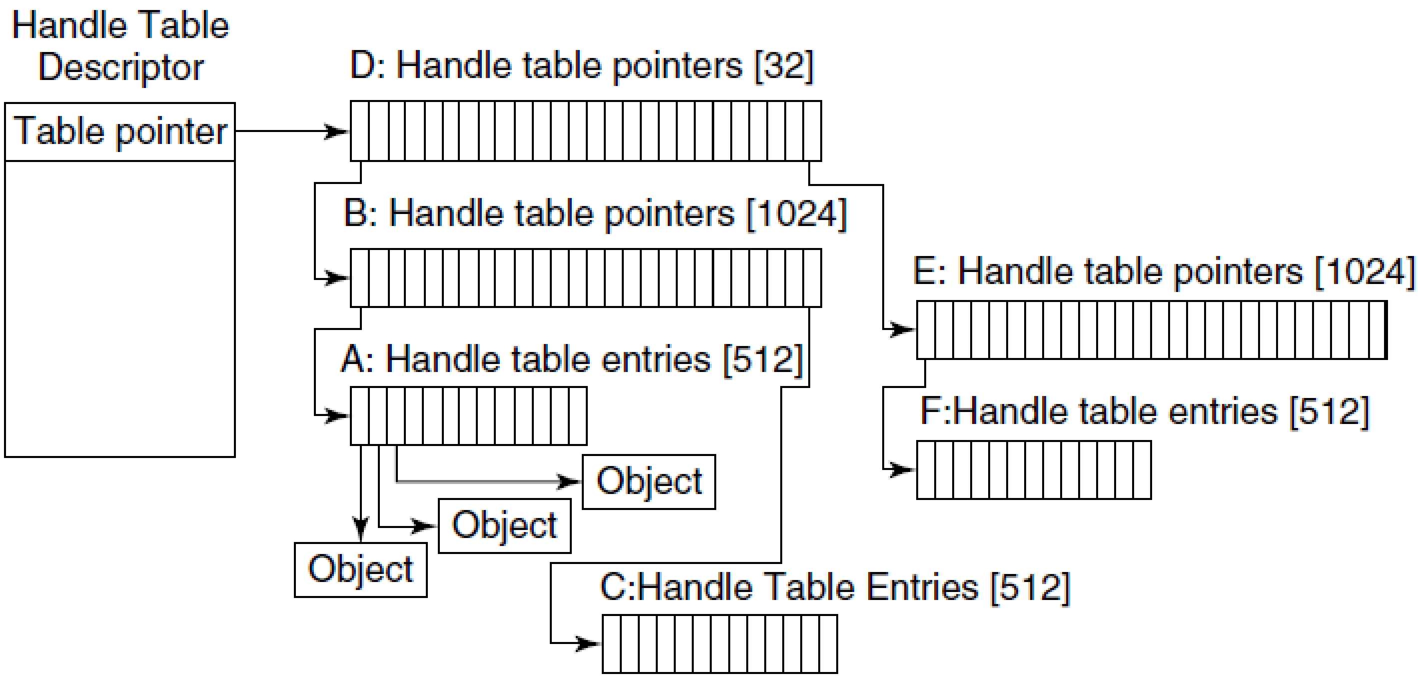

The structure of an executive object managed by the object manager

# Handles (1)




Handle table data structures for a minimal table using a single page for up to 512 handles.

# Handles (2)




Handle table data structures for a maximal table of up to 16 million handles.

# The Object Name Space (1)

| Procedure | When called | Notes |
|---|---|---|
| Open | For every new handle | Rarely used |
| Parse | For object types that extend the namespace | Used for files and registry keys |
| Close | At last handle close | Clean up visible side effects |
| Delete | At last pointer dereference | Object is about to be deleted |
| Security | Get or set object's security descriptor | Protection |
| QueryName | Get object's name | Rarely used outside kernel |

The object procedures supplied when specifying a new object type.

# The Object Name Space (2)

| Directory | Contents |
|---|---|
| \?? | Starting place for looking up MS-DOS devices like C: |
| \DosDevices | Official name of \??, but really just a symbolic link to \?? |
| \Device | All discovered I/O devices |
| \Driver | Objects corresponding to each loaded device driver |
| \ObjectTypes | The type objects such as those listed in Fig. 11-22 |
| \Windows | Objects for sending messages to all the Win32 GUI windows |
| \BaseNamedObjects | User-created Win32 objects such as semaphores, mutexes, etc. |
| \Arcname | Partition names discovered by the boot loader |
| \NLS | National Language Support objects |
| \FileSystem | File system driver objects and file system recognizer objects |
| \Security | Objects belonging to the security system |
| \KnownDLLs | Key shared libraries that are opened early and held open |

Some typical directories in the object name space.

# The Object Name Space (3)

Use of *parse* procedure:

1. Executive component passes Unicode pathname for namespace
2. Object manager searches through directories and symbolic links
3. Object manager calls the *Parse* procedure for object type
4. I/O manager creates IRP, allocate file object, send request to stack of I/O devices

# The Object Name Space (4)

Use of *parse* procedure:

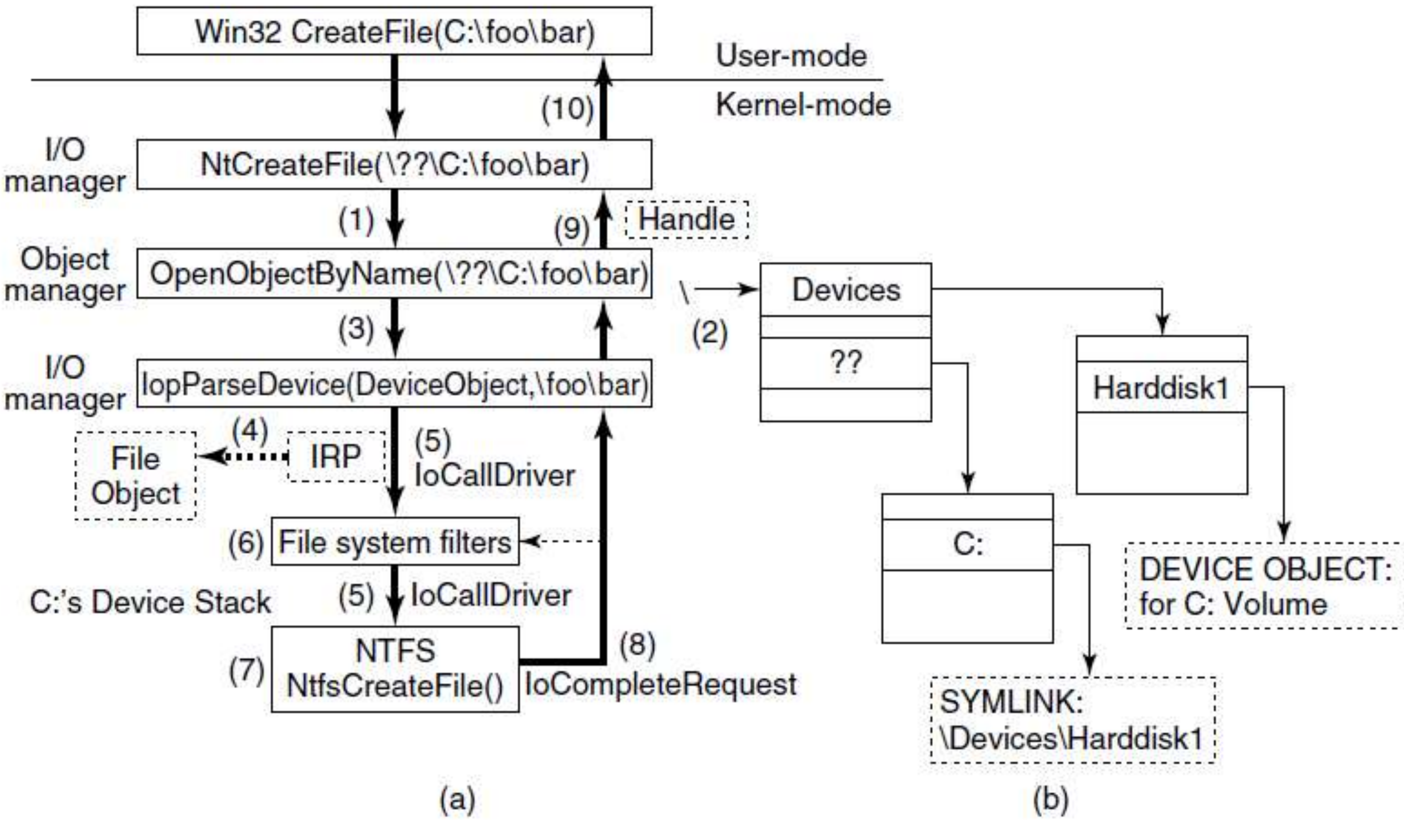
5. IRP passed down the I/O stack until it reaches device object representing the file system instance
6. Device objects encountered as the IRP heads toward the file system represent file system filter drivers
7. File system device object has a link to file system driver object

# The Object Name Space (5)

Use of *parse* procedure:

8. NTFS fills in file object and returns it to I/O manager, which returns back up through all devices on the stack
9. Object manager is finished with its namespace lookup
10. Final step is to return back to the user-mode caller

# The Object Name Space (6)




I/O and object manager steps for creating/opening a file and getting back a file handle.

# The Object Name Space (7)

| Type | Description |
|---|---|
| Process | User process |
| Thread | Thread within a process |
| Semaphore | Counting semaphore used for interprocess synchronization |
| Mutex | Binary semaphore used to enter a critical region |
| Event | Synchronization object with persistent state (signaled/not) |
| ALPC Port | Mechanism for interprocess message passing |
| Timer | Object allowing a thread to sleep for a fixed time interval |
| Queue | Object used for completion notification on asynchronous I/O |
| Open file | Object associated with an open file |
| Access token | Security descriptor for some object |

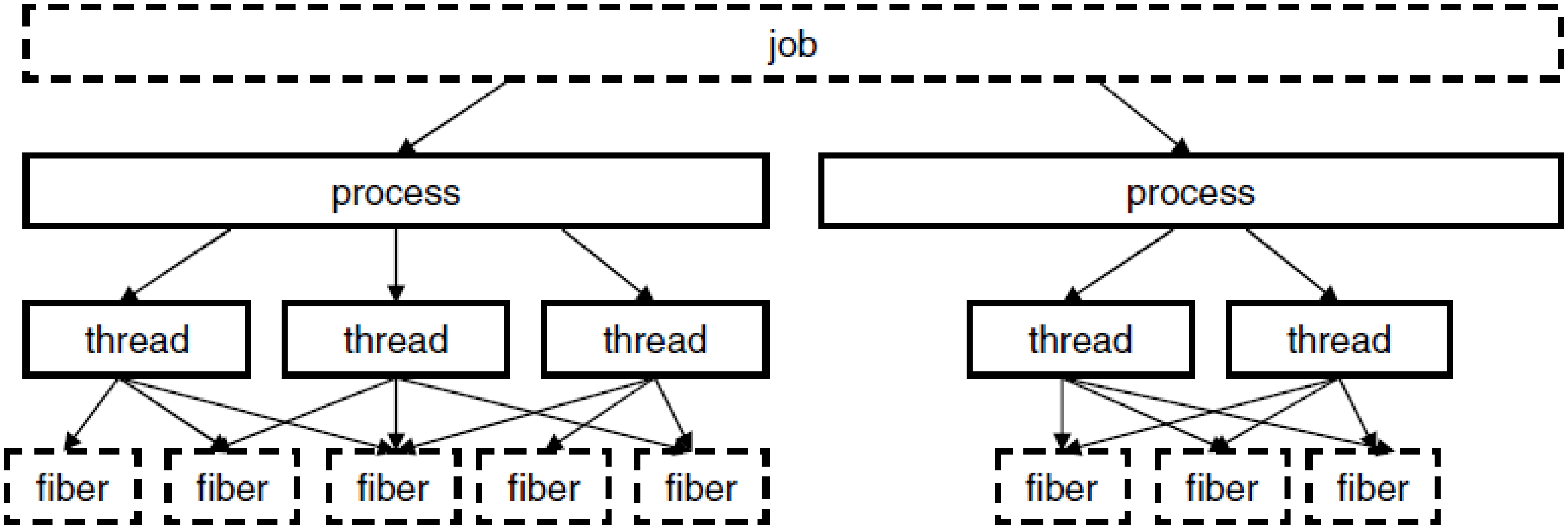
Some common executive object types managed by the object manager.

# The Object Name Space (8)

| | |
|---|---|
| Queue | Object used for completion notification on asynchronous I/O |
| Open file | Object associated with an open file |
| Access token | Security descriptor for some object |
| Profile | Data structure used for profiling CPU usage |
| Section | Object used for representing mappable files |
| Key | Registry key, used to attach registry to object manager namespace |
| Object directory | Directory for grouping objects within the object manager |
| Symbolic link | Refers to another object manager object by pathname |
| Device | I/O device object for a physical device, bus, driver, or volume instance |
| Device driver | Each loaded device driver has its own object |

Some common executive object types managed by the object manager.

# Jobs, Processes, Threads, Fibers




The relationship between jobs, processes, threads and fibers. Jobs and fibers are optional; not all processes are in jobs or contain fibers.

# Thread Pools and User-Mode Scheduling (1)

Key elements of UMS implementation:

1. User-mode switching
2. Re-entering the user-mode scheduler
3. System call completion

# Thread Pools and User-Mode Scheduling (2)

| Name | Description | Notes |
|---|---|---|
| Job | Collection of processes that share quotas and limits | Used in AppContainers |
| Process | Container for holding resources | |
| Thread | Entity scheduled by the kernel | |
| Fiber | Lightweight thread managed entirely in user space | Rarely used |
| Thread Pool | Task-oriented programming model | Built on top of threads |
| User-mode Thread | Abstraction allowing user-mode thread switching | An extension of threads |

Basic concepts used for CPU and resource management.

# Job, Process, Thread, and Fiber Management API Calls (1)

Differences from UNIX:

1. Actual search path for finding program to execute buried in library code for Win32, but managed more explicitly in UNIX.
2. Current working directory is a kernel-mode concept in UNIX but a user-mode string in Windows.
3. UNIX parses command line and passes array of parameters, while Win32 leaves argument parsing up to individual program.

# Job, Process, Thread, and Fiber Management API Calls (2)

Differences from UNIX:

4. Inheritance of file descriptors in UNIX a property the handle. In Windows is also property of handle to process creation.
5. New processes directly passed information about primary window in Win32. Passed as parameters to GUI applications in UNIX.
6. Windows does not have SETUID bit as property of the executable, but one process can create a process that runs as a different user (with proper token)

# Job, Process, Thread, and Fiber Management API Calls (3)

Differences from UNIX:

7. Process and thread handle returned from Windows can be used at any time to modify new process/thread in many ways.
   UNIX only makes modifications to new process between fork and exec calls, and only in limited ways

# Implementation of Processes and Threads (1)

| **Win32 API Function** | **Description** |
|---|---|
| CreateProcess | Create a new process |
| CreateThread | Create a new thread in an existing process |
| CreateFiber | Create a new fiber |
| ExitProcess | Terminate current process and all its threads |
| ExitThread | Terminate this thread |
| ExitFiber | Terminate this fiber |
| SwitchToFiber | Run a different fiber on the current thread |
| SetPriorityClass | Set the priority class for a process |
| SetThreadPriority | Set the priority for one thread |
| CreateSemaphore | Create a new semaphore |
| CreateMutex | Create a new mutex |
| OpenSemaphore | Open an existing semaphore |
| OpenMutex | Open an existing mutex |

Some of the Win32 calls for managing processes, threads, and fibers.

# Implementation of Processes and Threads (2)

| | |
|---|---|
| CreateSemaphore | Create a new semaphore |
| CreateMutex | Create a new mutex |
| OpenSemaphore | Open an existing semaphore |
| OpenMutex | Open an existing mutex |
| WaitForSingleObject | Block on a single semaphore, mutex, etc. |
| WaitForMultipleObjects | Block on a set of objects whose handles are given |
| PulseEvent | Set an event to signaled then to nonsignaled |
| ReleaseMutex | Release a mutex to allow another thread to acquire it |
| ReleaseSemaphore | Increase the semaphore count by 1 |
| EnterCriticalSection | Acquire the lock on a critical section |
| LeaveCriticalSection | Release the lock on a critical section |
| WaitOnAddress | Block until the memory is changed at the specified address |
| WakeByAddressSingle | Wake the first thread that is waiting on this address |
| WakeByAddressAll | Wake all threads that are waiting on this address |
| InitOnceExecuteOnce | Ensure that an initialize routine executes only once |

Some of the Win32 calls for managing processes, threads, and fibers.

# Scheduling (1)

Conditions that invoke scheduling

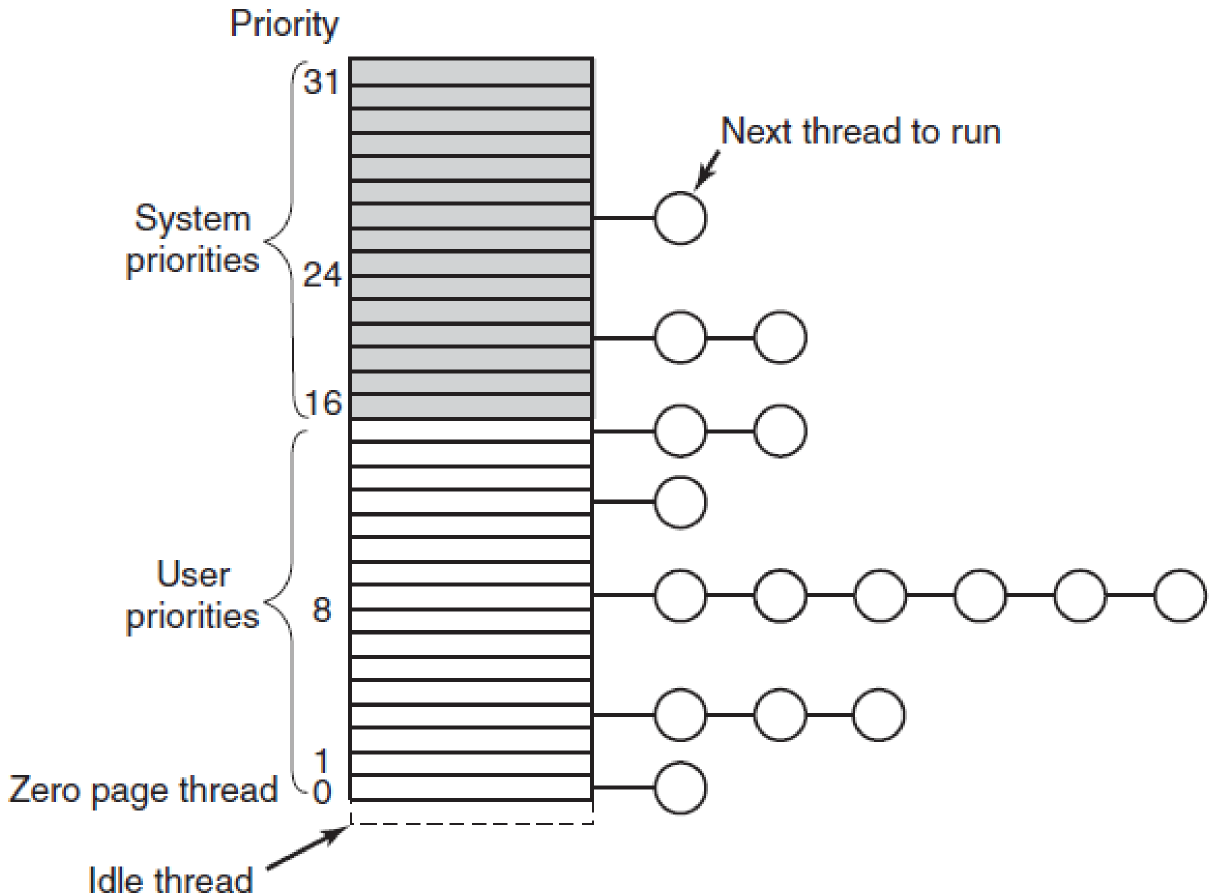
1. A running thread blocks on a semaphore, mutex, event, I/O, etc.
2. Thread signals an object (e.g., does an up on a semaphore).
3. The quantum expires.
4. An I/O operation completes.
5. A timed wait expires.

# Scheduling (2)

| | | Win32 process class priorities | | | | | |
|---|---|---|---|---|---|---|---|
| | | Real-time | High | Above Normal | Normal | Below Normal | Idle |
| Win32 thread priorities | Time critical | 31 | 15 | 15 | 15 | 15 | 15 |
| | Highest | 26 | 15 | 12 | 10 | 8 | 6 |
| | Above normal | 25 | 14 | 11 | 9 | 7 | 5 |
| | Normal | 24 | 13 | 10 | 8 | 6 | 4 |
| | Below normal | 23 | 12 | 9 | 7 | 5 | 3 |
| | Lowest | 22 | 11 | 8 | 6 | 4 | 2 |
| | Idle | 16 | 1 | 1 | 1 | 1 | 1 |

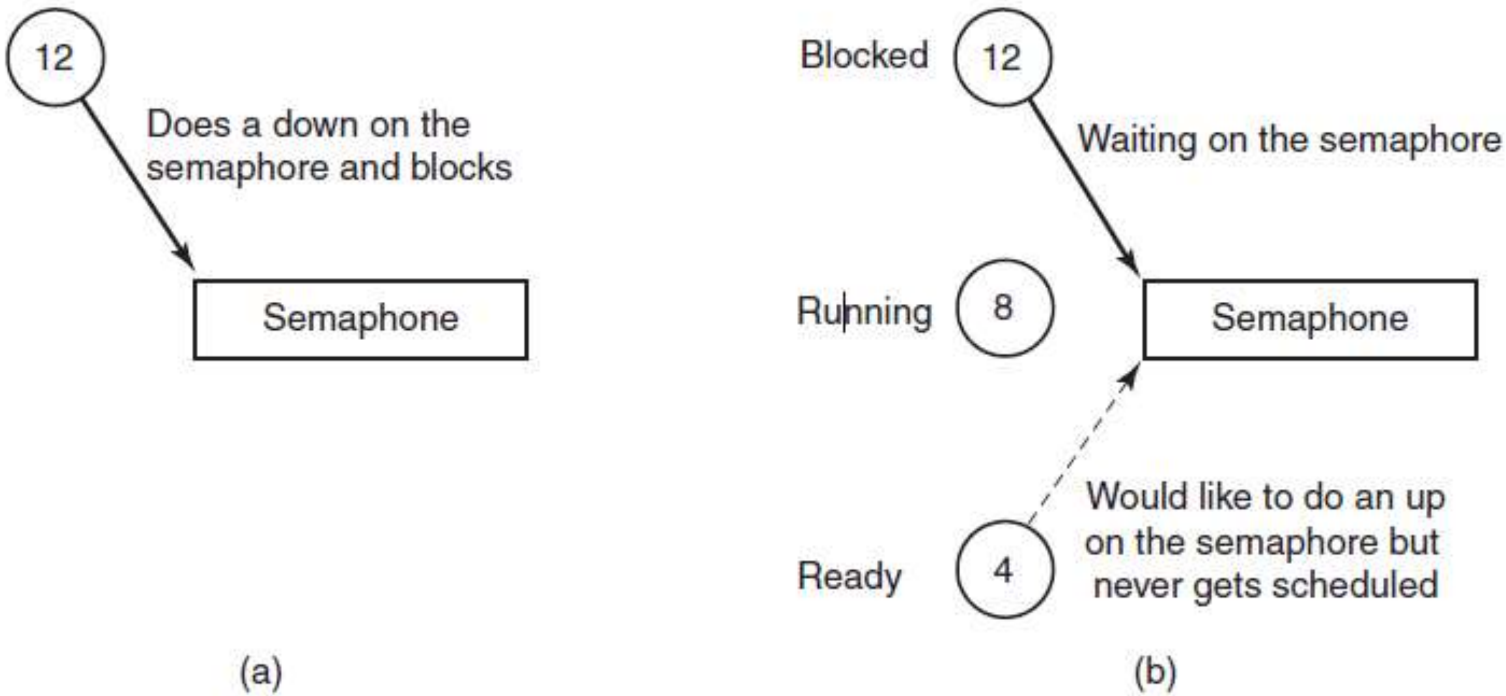
Mapping of Win32 priorities to Windows priorities.

# Scheduling (3)



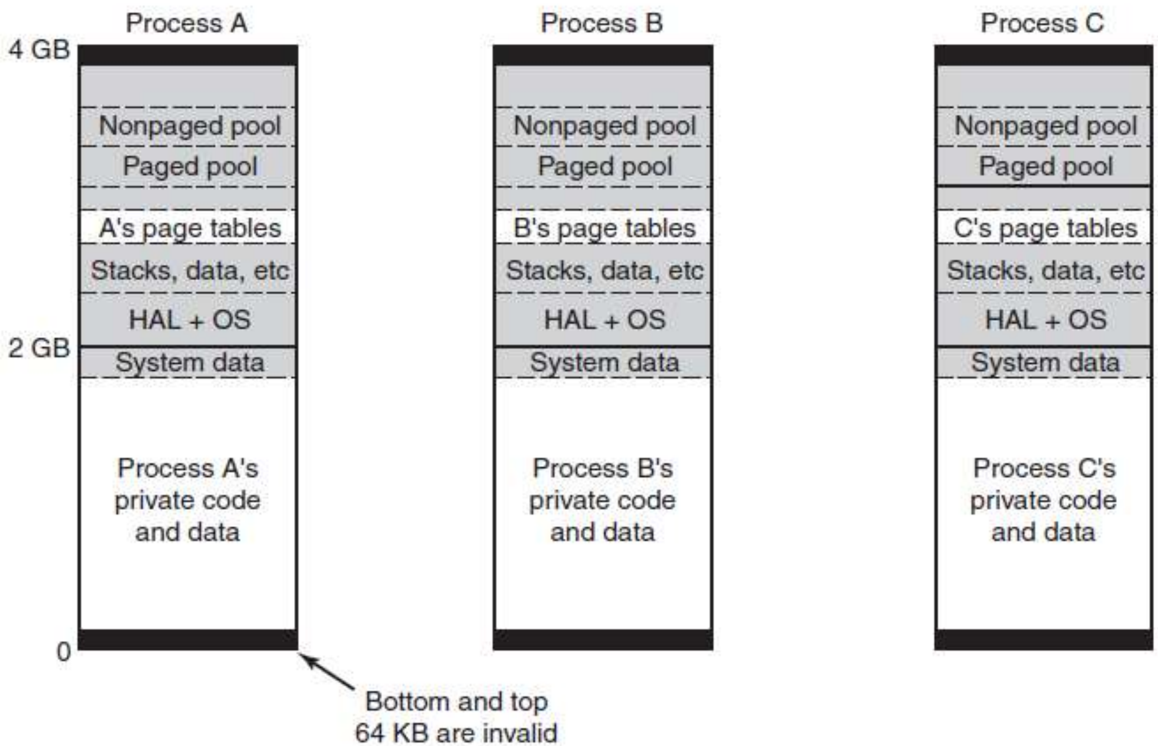

Windows supports 32 priorities for threads.

# Scheduling (4)




An example of priority inversion.

# Memory Management Fundamental Concepts



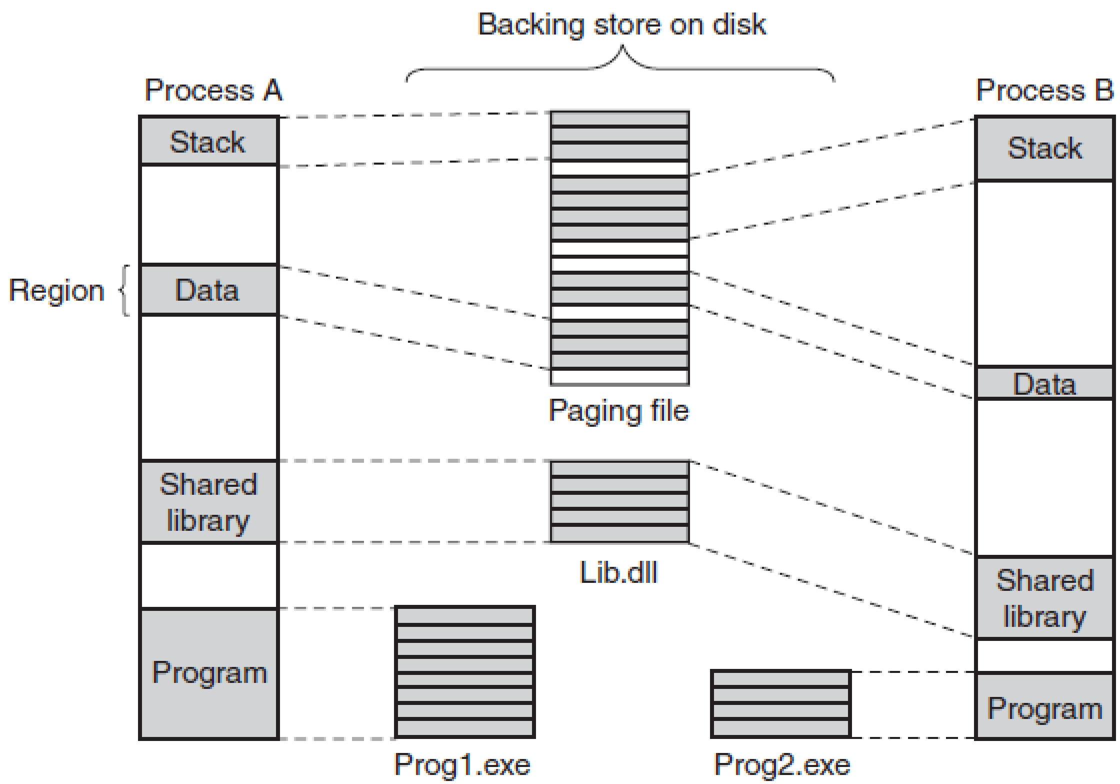

Virtual address space layout for three user processes on the x86. The white areas are private per process. The shaded areas are shared among all processes.

# Memory-Management System Calls

| Win32 API function | Description |
|---|---|
| VirtualAlloc | Reserve or commit a region |
| VirtualFree | Release or decommit a region |
| VirtualProtect | Change the read/write/execute protection on a region |
| VirtualQuery | Inquire about the status of a region |
| VirtualLock | Make a region memory resident (i.e., disable paging for it) |
| VirtualUnlock | Make a region pageable in the usual way |
| CreateFileMapping | Create a file mapping object and (optionally) assign it a name |
| MapViewOfFile | Map (part of) a file into the address space |
| UnmapViewOfFile | Remove a mapped file from the address space |
| OpenFileMapping | Open a previously created file mapping object |

The principal Win32 API functions for managing virtual memory in Windows.

# Implementation of Memory Management




Mapped regions with their shadow pages on disk. The *lib.dll* file is mapped into two address spaces at the same time.

# Page Fault Handling (1)

| 63 | 62 52 | 51 12 | 11 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NX | AVL | Physical page number | AVL | G | PAT | D | A | PCD | PWT | U/S | R/W | P |

NX No eXecute
AVL AVaiLable to the OS
G Global page
PAT Page Attribute Table
D Dirty (modified)
A Accessed (referenced)

PCD Page Cache Disable
PWT Page Write-Through
U/S User/Supervisor
R/W Read/Write access
P Present (valid)

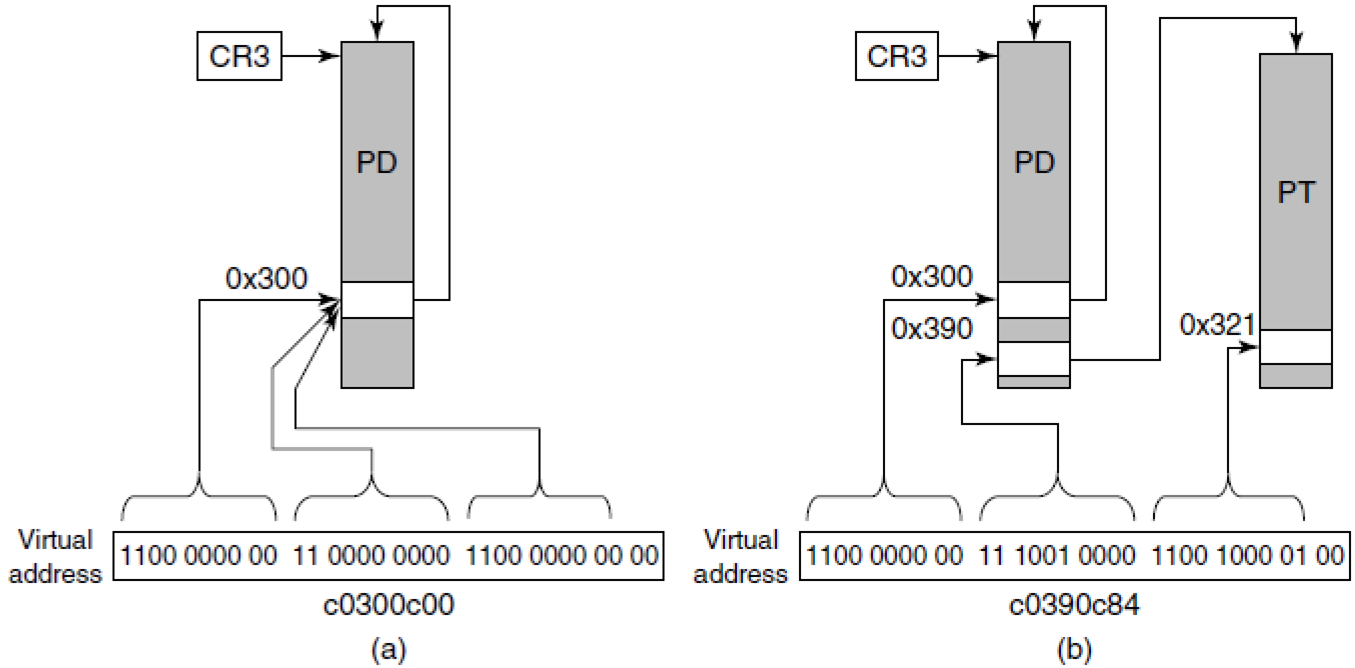
A page table entry (PTE) for a mapped page on the Intel x86 and AMD x64 architectures.

# Page Fault Handling (2)

Categories of page faults:

1. The page referenced is not committed.
2. Attempted access to a page in violation of the permissions.
3. A shared copy-on-write page was about to be modified.
4. The stack needs to grow.
5. The page referenced is committed but not currently mapped in.

# The Page Replacement Algorithm (1)




Self-map: PD[0xc0300000>>22] is PD (page-directory)
Virtual address (a): (PTE *)(0xc0300c00) points to PD[0x300] which is the self-map page directory entry
Virtual address (b): (PTE *)(0xc0390c84) points to PTE for virtual address 0xe4321000

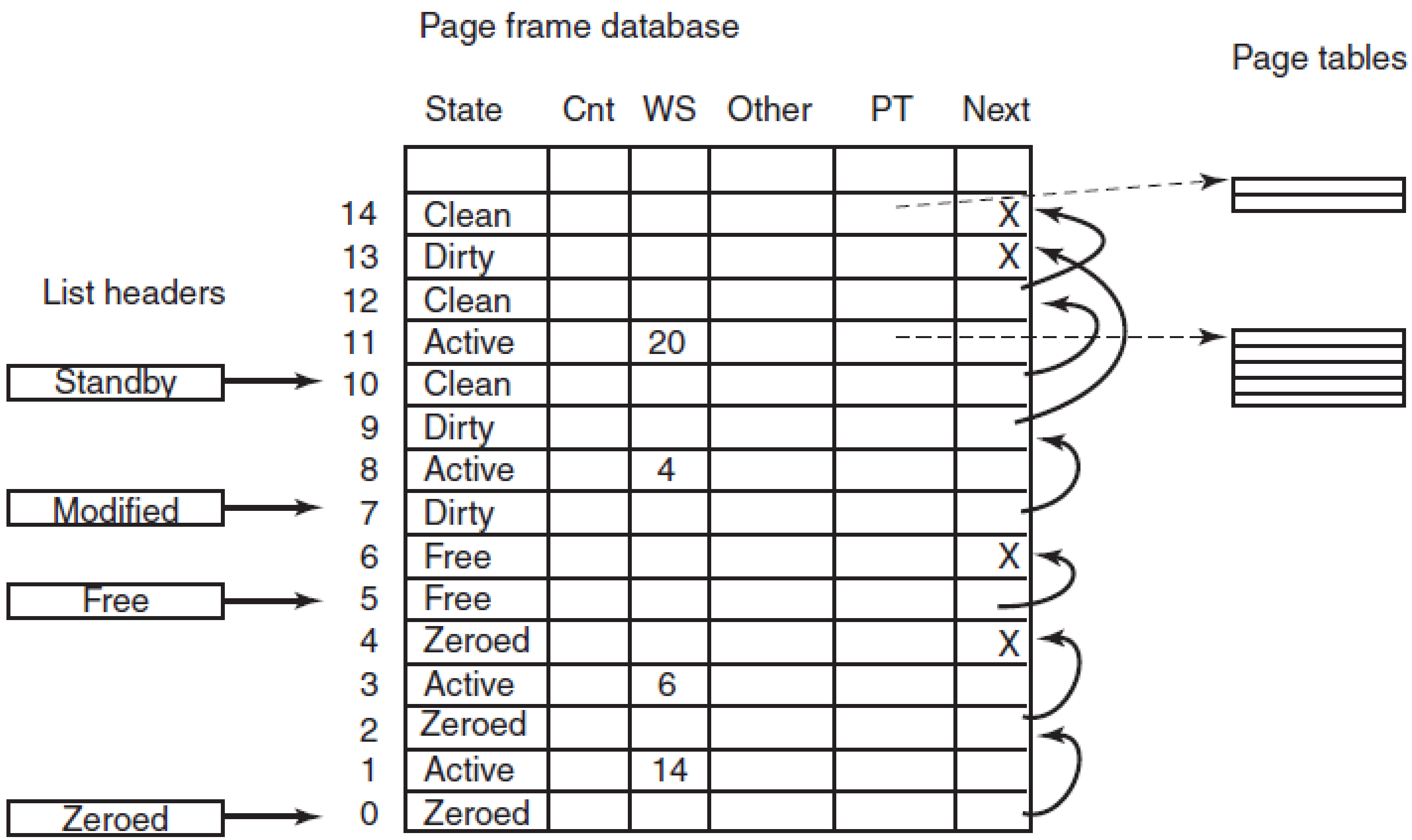
The Windows self-map entries are used to map the physical pages of the page tables and page directory into kernel virtual addresses (shown for 32-bit PTEs).

# The Page Replacement Algorithm (2)

Three levels of activity by the working-set manager:

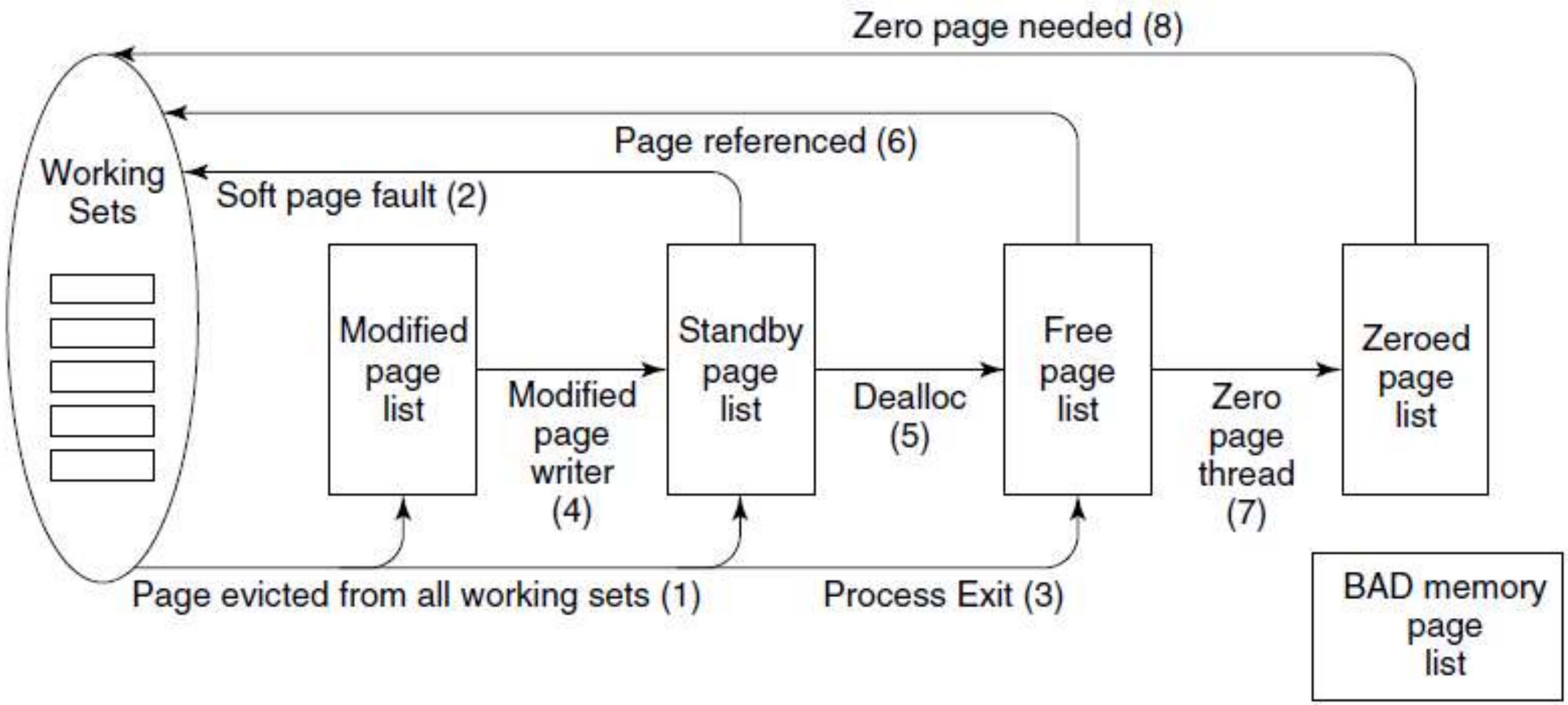
1. Lots of memory available
2. Memory getting tight
3. Memory is tight

# Windows Physical Memory Management (1)




Some of the major fields in the page frame database for a valid page.

# Windows Physical Memory Management (2)



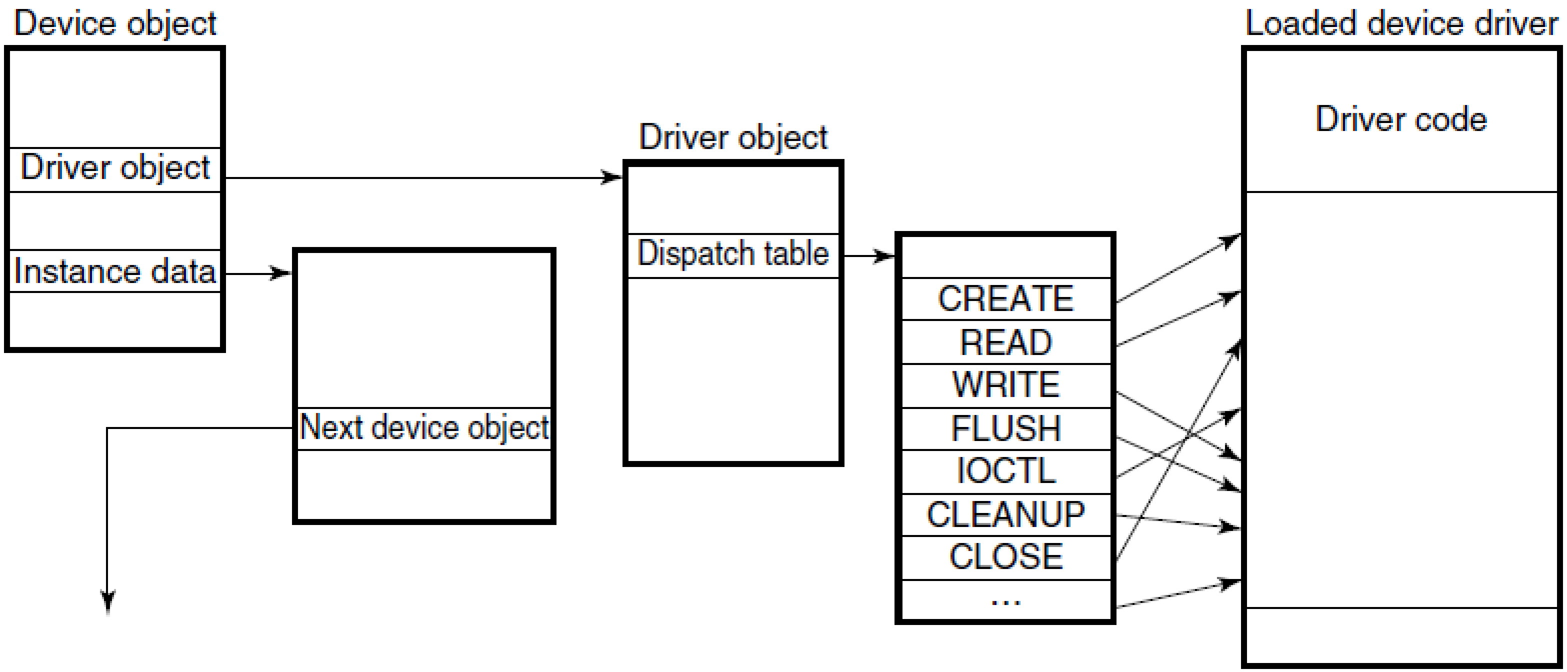

The various page lists and the transitions between them.

# Input/Output API Calls

| I/O system call | Description |
|---|---|
| NtCreateFile | Open new or existing files or devices |
| NtReadFile | Read from a file or device |
| NtWriteFile | Write to a file or device |
| NtQueryDirectoryFile | Request information about a directory, including files |
| NtQueryVolumeInformationFile | Request information about a volume |
| NtSetVolumeInformationFile | Modify volume information |
| NtNotifyChangeDirectoryFile | Complete when any file in the directory or sub-tree is modified |
| NtQueryInformationFile | Request information about a file |
| NtSetInformationFile | Modify file information |
| NtLockFile | Lock a range of bytes in a file |
| NtUnlockFile | Remove a range lock |
| NtFsControlFile | Miscellaneous operations on a file |
| NtFlushBuffersFile | Flush in-memory file buffers to disk |
| NtCancelIoFile | Cancel outstanding I/O operations on a file |
| NtDeviceIoControlFile | Special operations on a device |

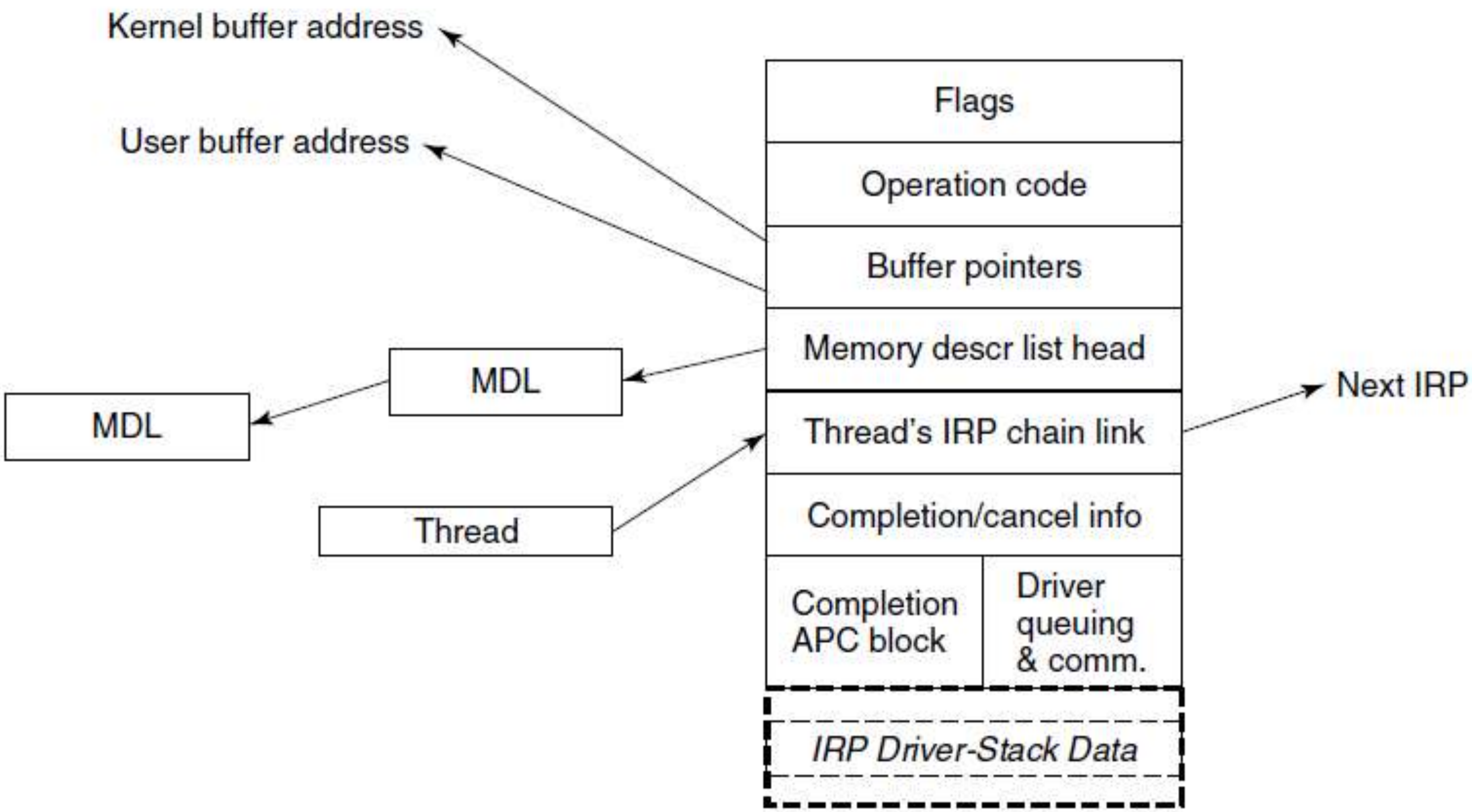
Native NT API calls for performing I/O.

# Device Drivers




A single level in a device stack.

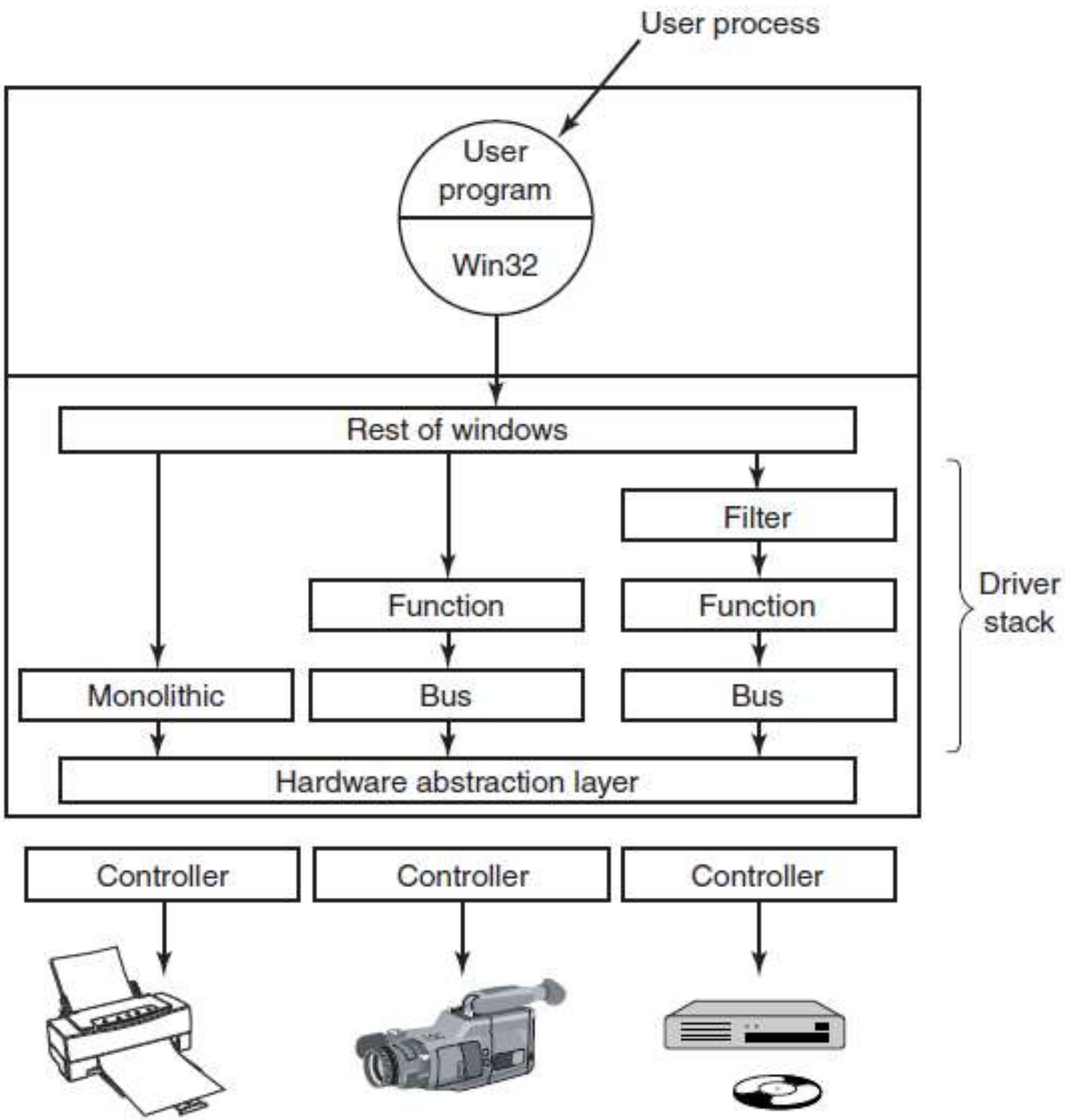
# I/O Request Packets



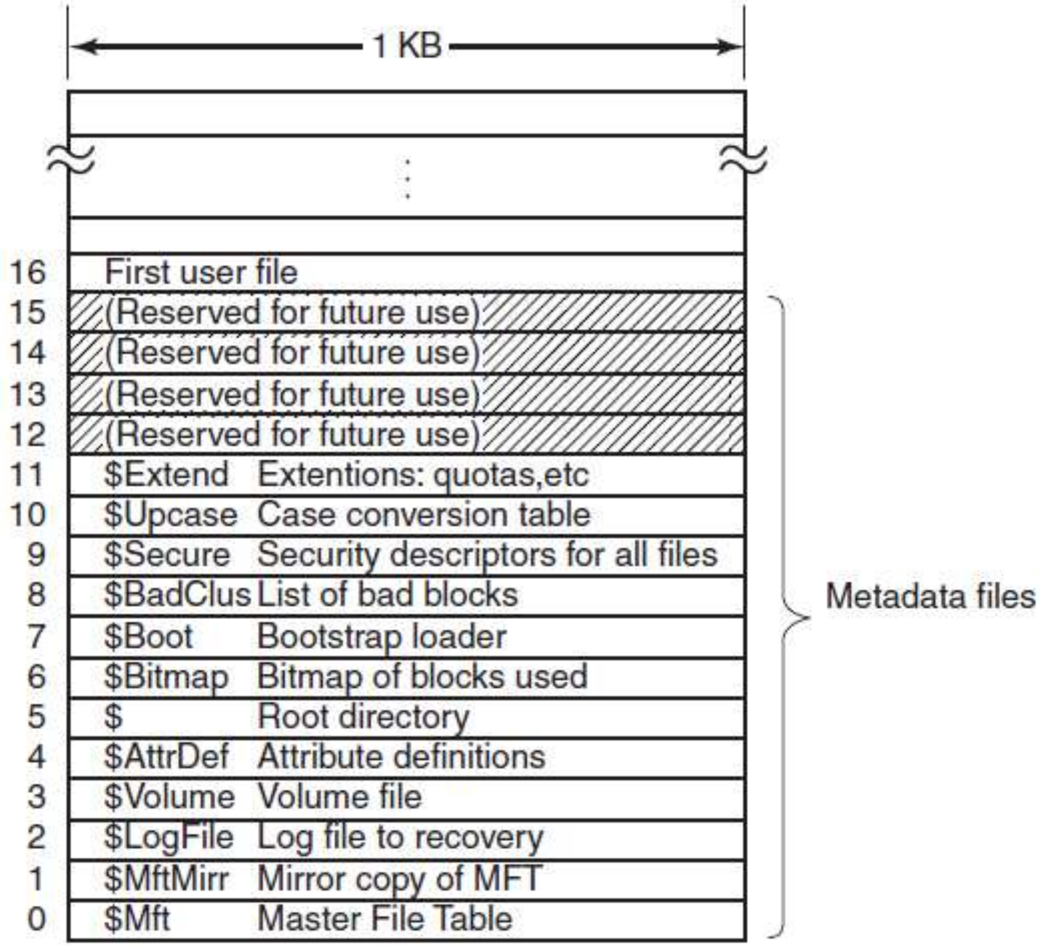

The major fields of an I/O Request Packet.

# Device Stacks

Windows allows drivers to be stacked to work with a specific instance of a device. The stacking is represented by device objects.



# Implementation of the NT File System (1)



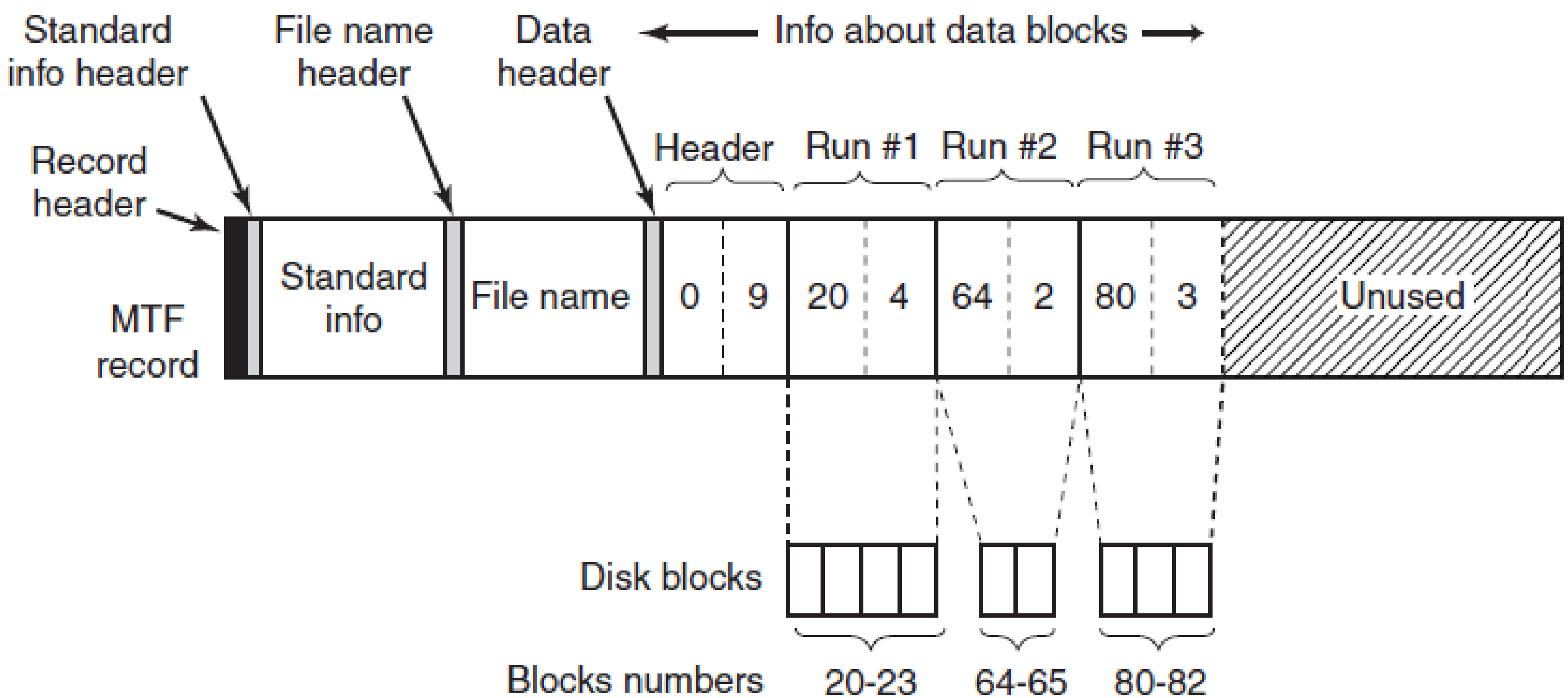

The NTFS master file table.

# Implementation of the NT File System (2)

| Attribute | Description |
|---|---|
| Standard information | Flag bits, timestamps, etc. |
| File name | File name in Unicode; may be repeated for MS-DOS name |
| Security descriptor | Obsolete. Security information is now in $Extend$Secure |
| Attribute list | Location of additional MFT records, if needed |
| Object ID | 64-bit file identifier unique to this volume |
| Reparse point | Used for mounting and symbolic links |
| Volume name | Name of this volume (used only in $Volume) |
| Volume information | Volume version (used only in $Volume) |
| Index root | Used for directories |
| Index allocation | Used for very large directories |
| Bitmap | Used for very large directories |
| Logged utility stream | Controls logging to $LogFile |
| Data | Stream data; may be repeated |

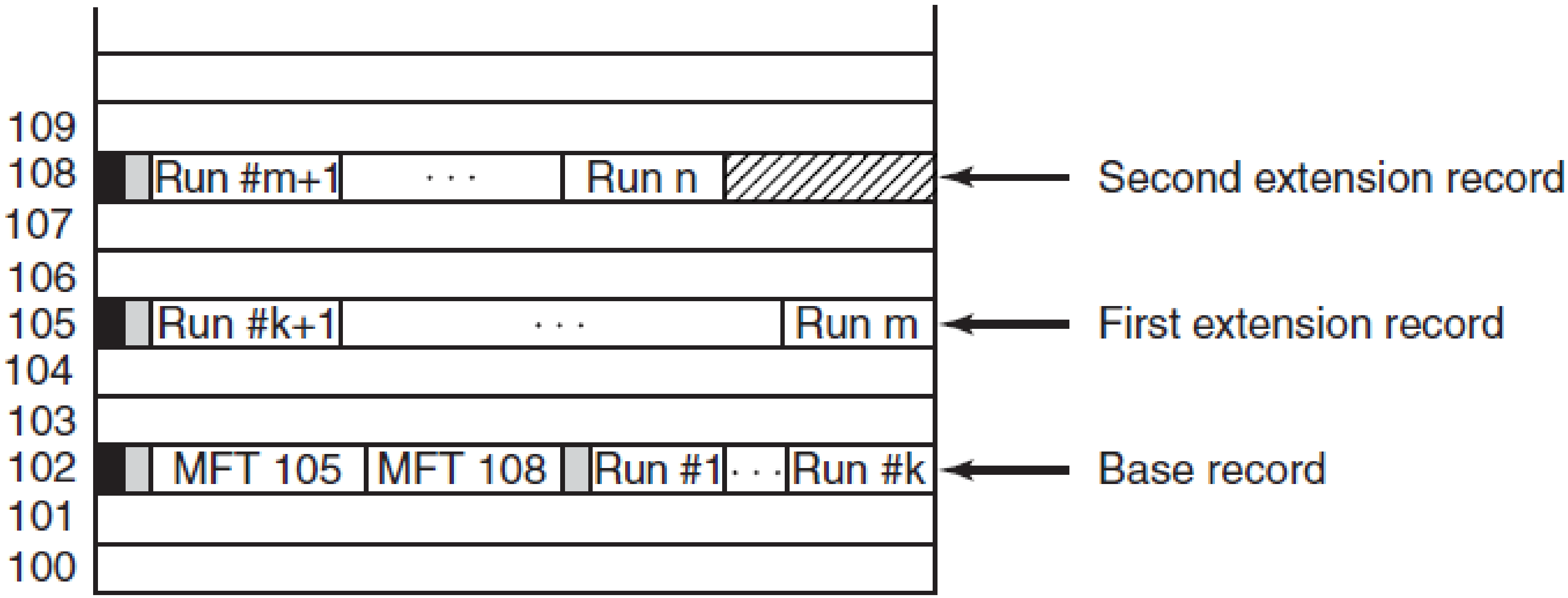
The attributes used in MFT records.

# Storage Allocation (1)



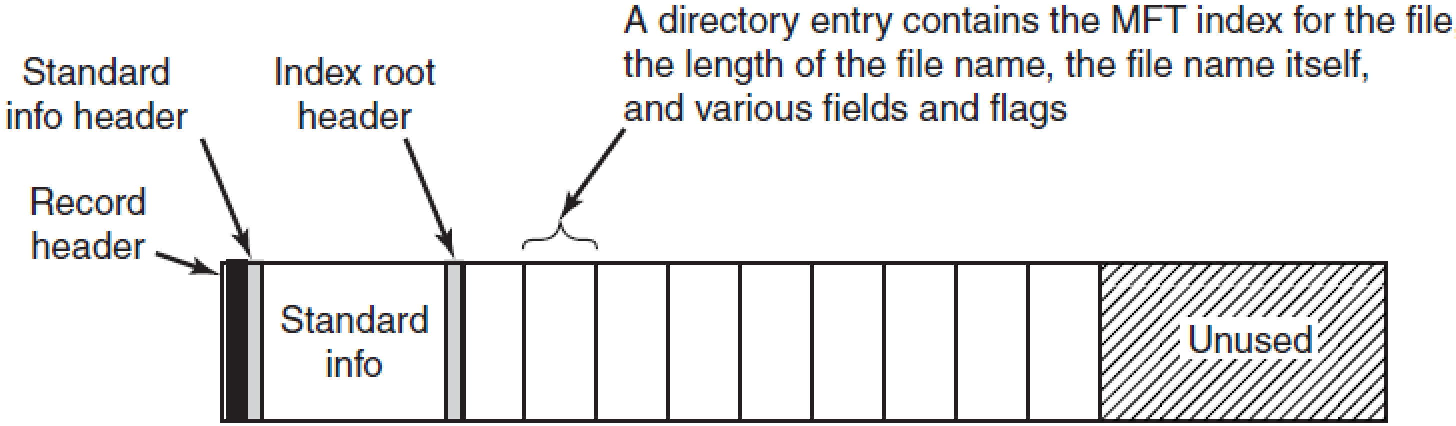

An MFT record for a three-run, nine-block stream.

# Storage Allocation (2)



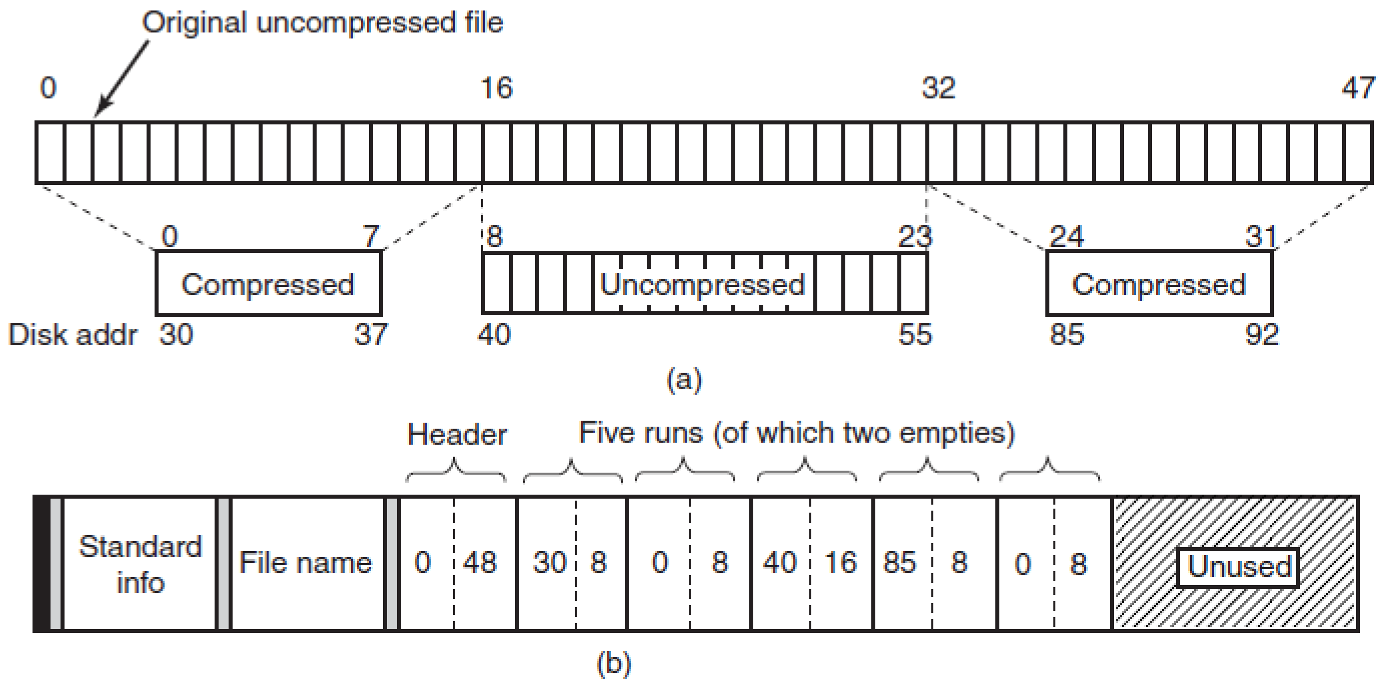

A file that requires three MFT records to store all its runs.

# Storage Allocation (3)




The MFT record for a small directory.

# File Compression



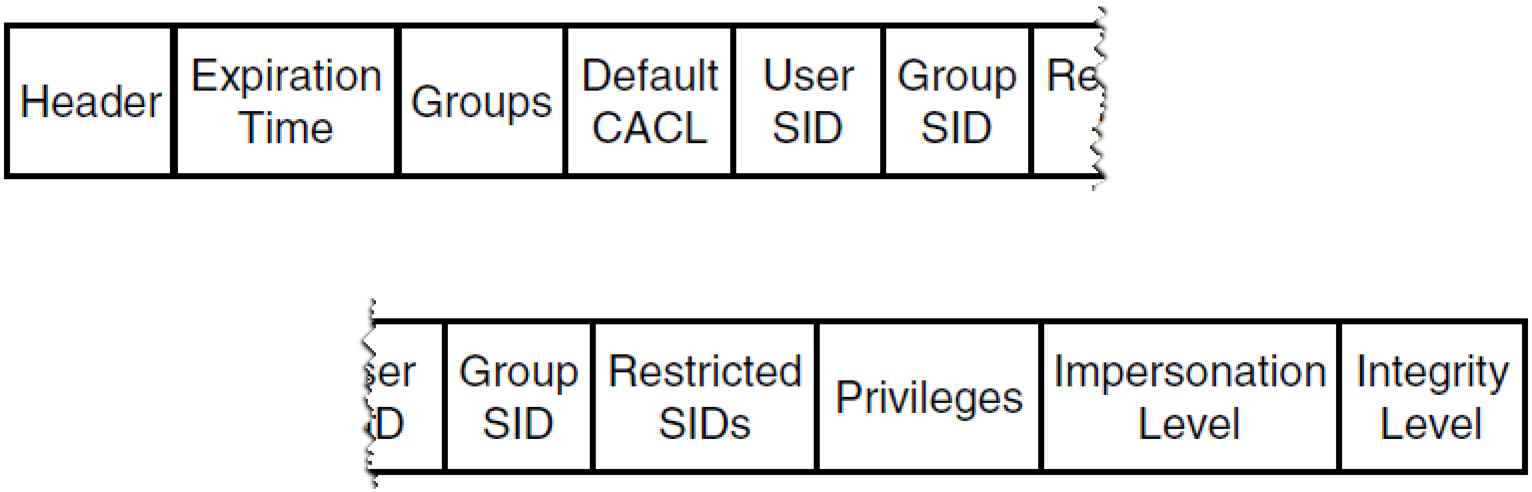

(a) An example of a 48-block file being compressed to 32 blocks.
(b) The MFT record for the file after compression.

# Security in Windows 8

Security properties inherited from NT:

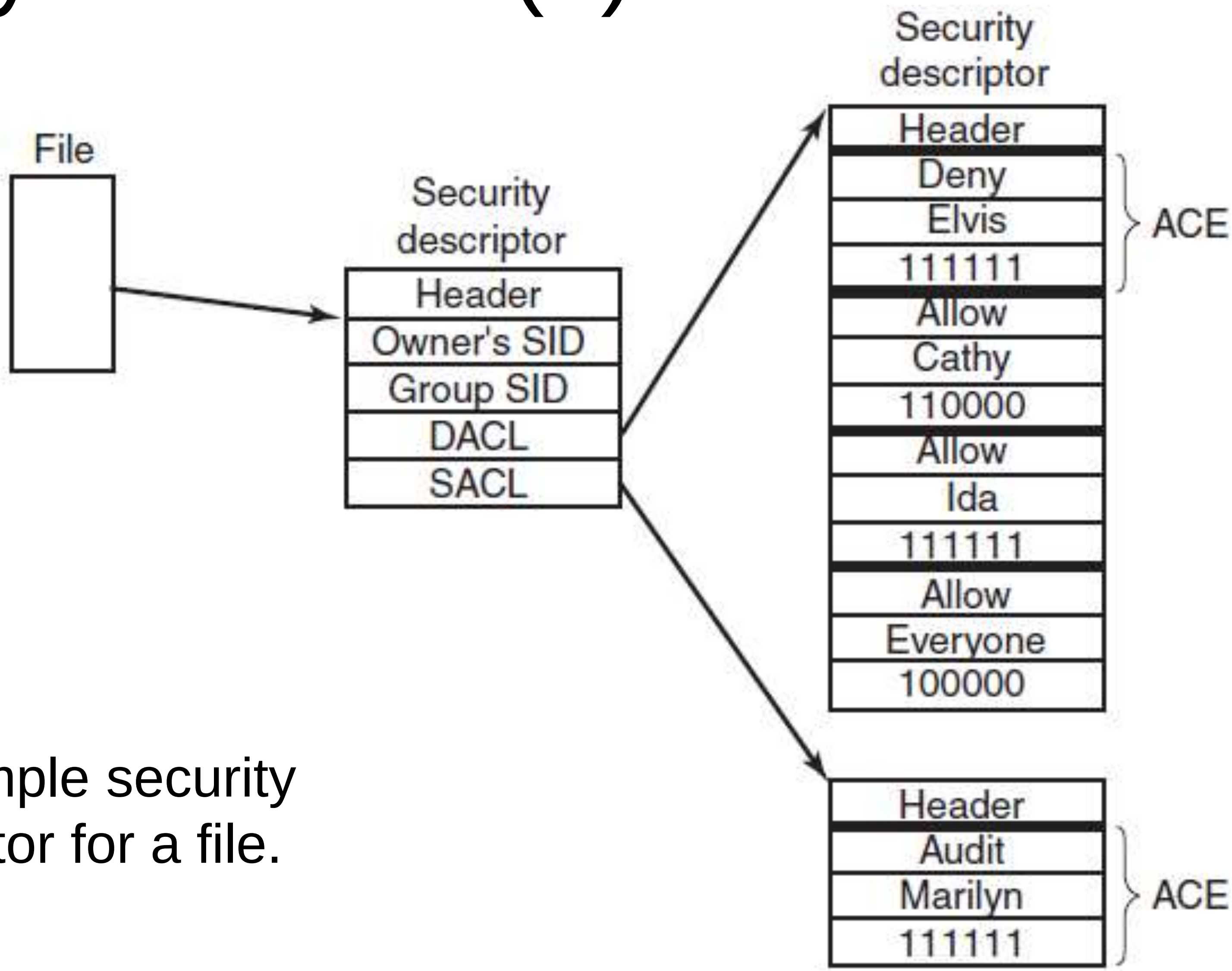
1. Secure login with antispoofing measures.
2. Discretionary access controls.
3. Privileged access controls.
4. Address space protection per process.
5. New pages must be zeroed before being mapped in.
6. Security auditing.

# Fundamental Security Concepts




Structure of an access token.

# Security API Calls (1)




An example security descriptor for a file.

# Security API Calls (2)

| Win32 API function | Description |
|---|---|
| InitializeSecurityDescriptor | Prepare a new security descriptor for use |
| LookupAccountSid | Look up the SID for a given user name |
| SetSecurityDescriptorOwner | Enter the owner SID in the security descriptor |
| SetSecurityDescriptorGroup | Enter a group SID in the security descriptor |
| InitializeAcl | Initialize a DACL or SACL |
| AddAccessAllowedAce | Add a new ACE to a DACL or SACL allowing access |
| AddAccessDeniedAce | Add a new ACE to a DACL or SACL denying access |
| DeleteAce | Remove an ACE from a DACL or SACL |
| SetSecurityDescriptorDacl | Attach a DACL to a security descriptor |

The principal Win32 API functions for security.

# Security Mitigations

| Mitigation | Description |
|---|---|
| /GS compiler flag | Add canary to stack frames to protect branch targets |
| Exception hardening | Restrict what code can be invoked as exception handlers |
| NX MMU protection | Mark code as non-executable to hinder attack payloads |
| ASLR | Randomize address space to make ROP attacks difficult |
| Heap hardening | Check for common heap usage errors |
| VTGuard | Add checks to validate virtual function tables |
| Code Integrity | Verify that libraries and drivers are properly cryptographically signed |
| Patchguard | Detect attempts to modify kernel data, e.g. by root kits |
| Windows Update | Provide regular security patches to remove vulnerabilities |
| Windows Defender | Built-in basic antivirus capability |

Some of the principal security mitigations in Windows.