6

THE OPERATING SYSTEM

MACHINE LEVEL

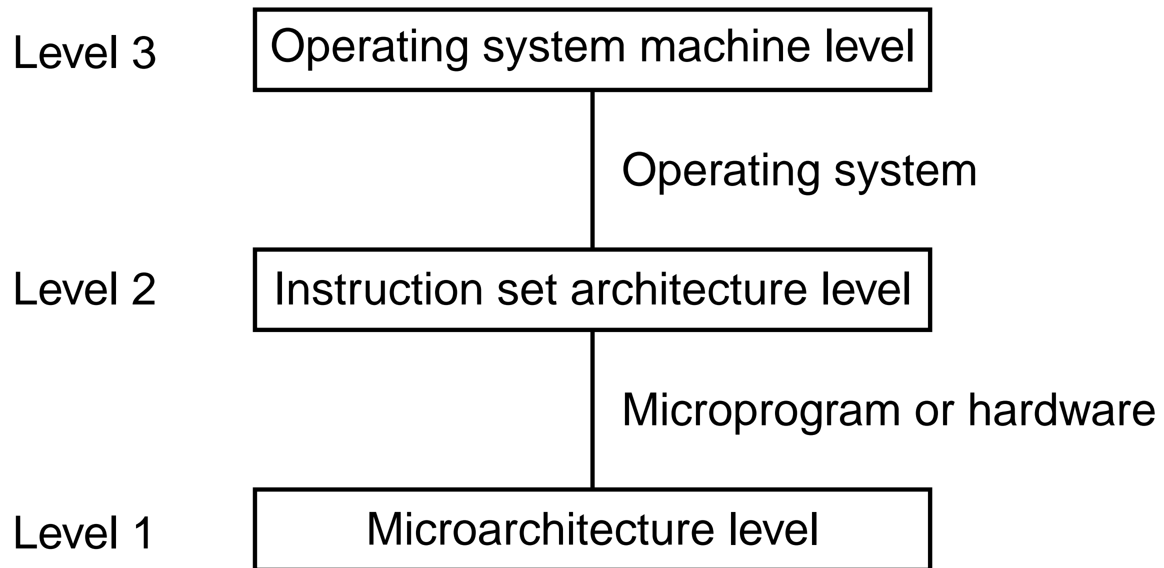

Figure 6-1. Positioning of the operating system machine level.

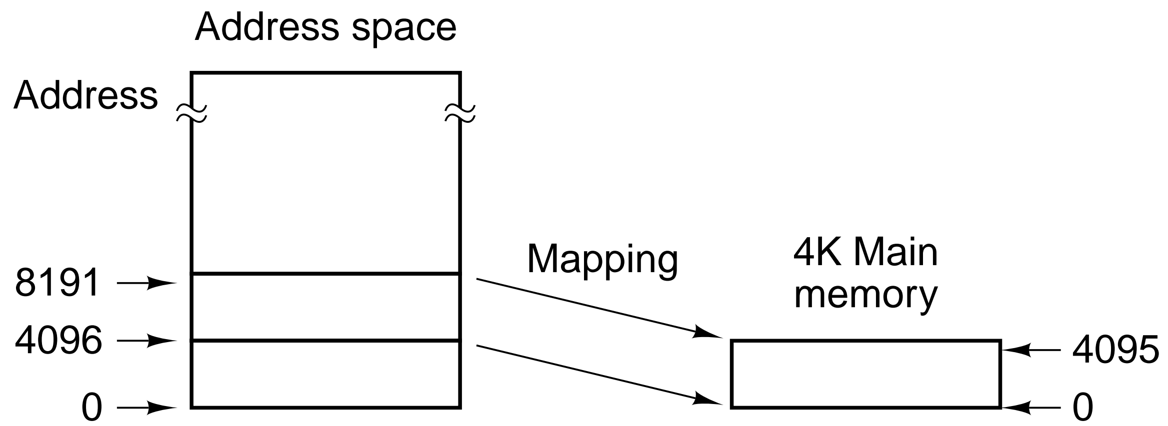

Figure 6-2. A mapping in which virtual addresses 4096 to 8191 are mapped onto main memory addresses 0 to 4095.

Page	Virtual addresses
≈	≈
15	61440 – 65535
14	57344 – 61439
13	53248 – 57343
12	49152 – 53247
11	45056 – 49151
10	40960 – 45055
9	36864 – 40959
8	32768 – 36863
7	28672 – 32767
6	24576 – 28671
5	20480 – 24575
4	16384 – 20479
3	12288 – 16383
2	8192 – 12287
1	4096 – 8191
0	0 – 4095

(a)

Page frame	Bottom 32K of main memory Physical addresses
7	28672 – 32767
6	24576 – 28671
5	20480 – 24575
4	16384 – 20479
3	12288 – 16383
2	8192 – 12287
1	4096 – 8191
0	0 – 4095

(b)

Figure 6-3. (a) The first 64K of virtual address space divided into 16 pages, with each page being 4K. (b) A 32K main memory divided up into eight page frames of 4K each.

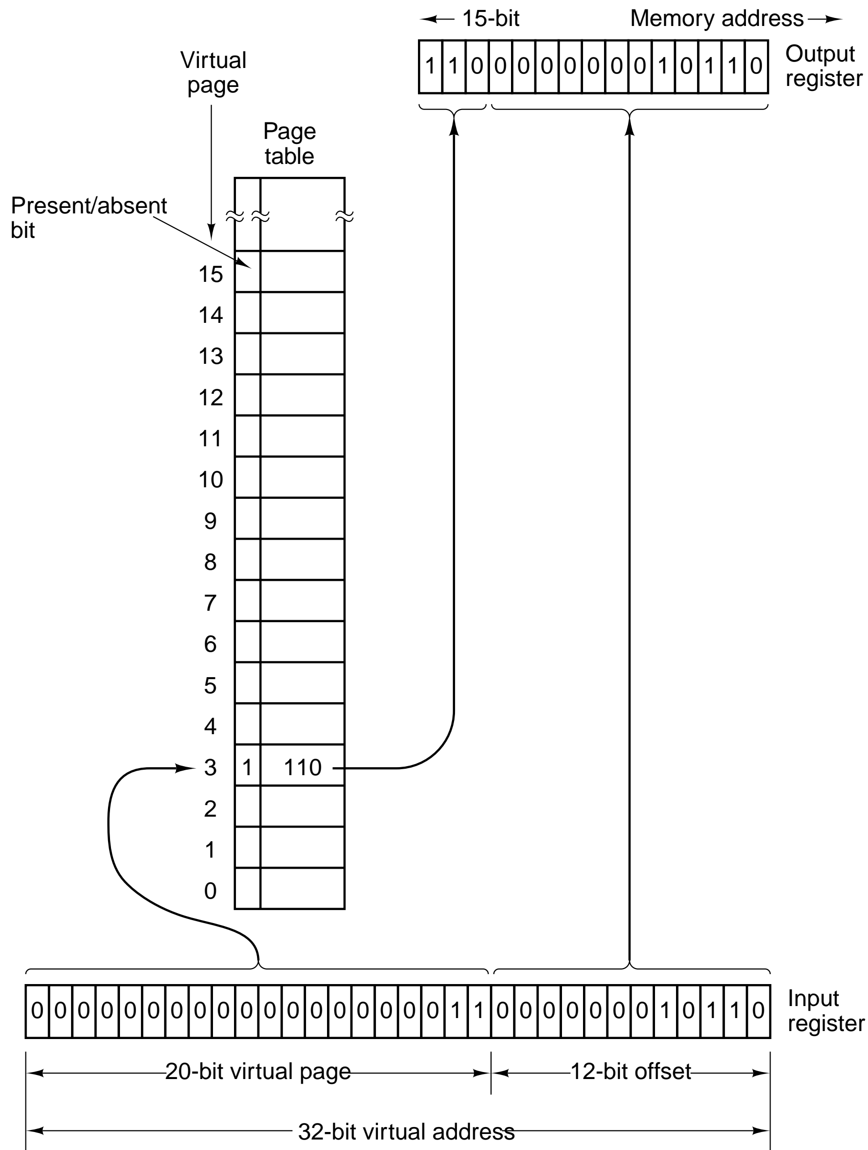

Figure 6-4. Formation of a main memory address from a virtual address.

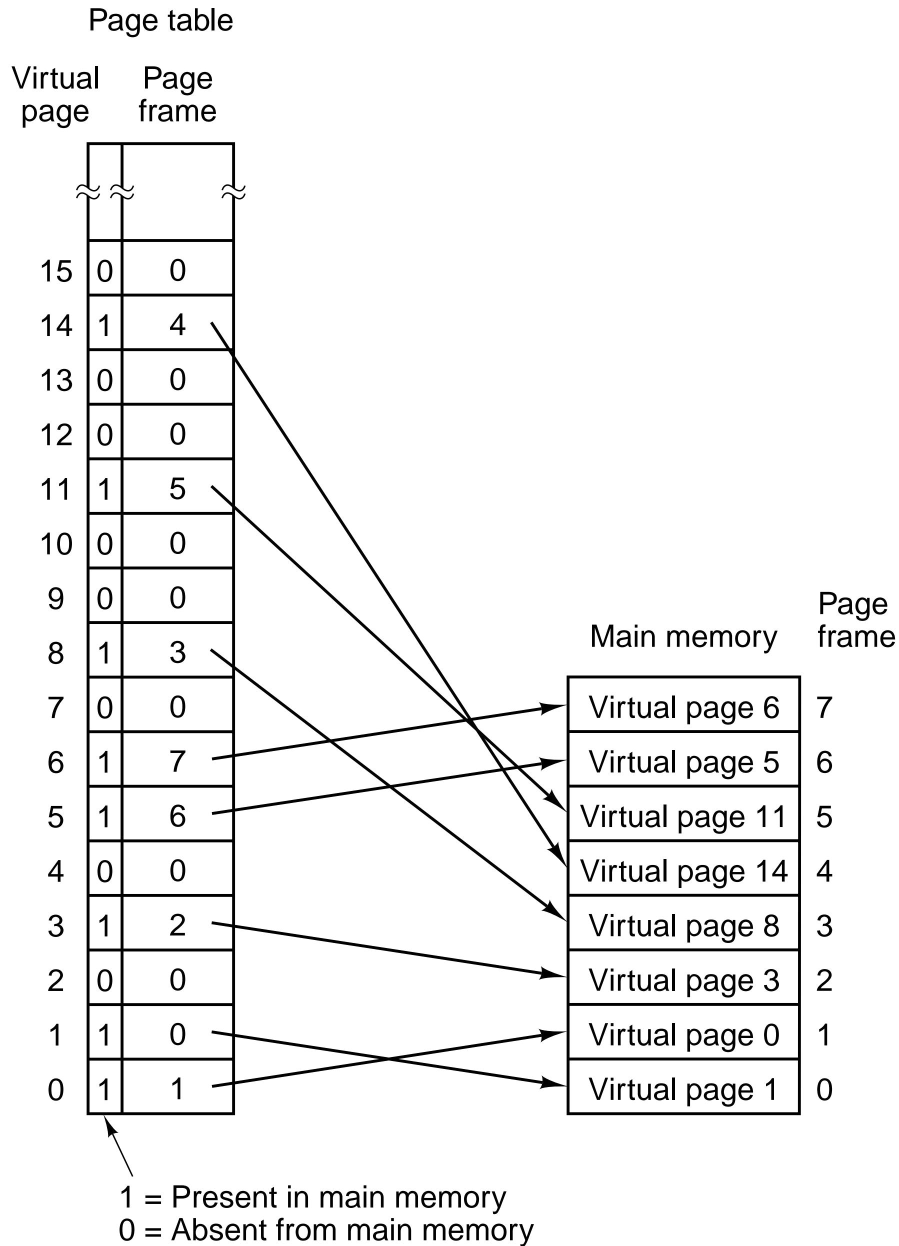

Figure 6-5. A possible mapping of the first 16 virtual pages onto a main memory with eight page frames.

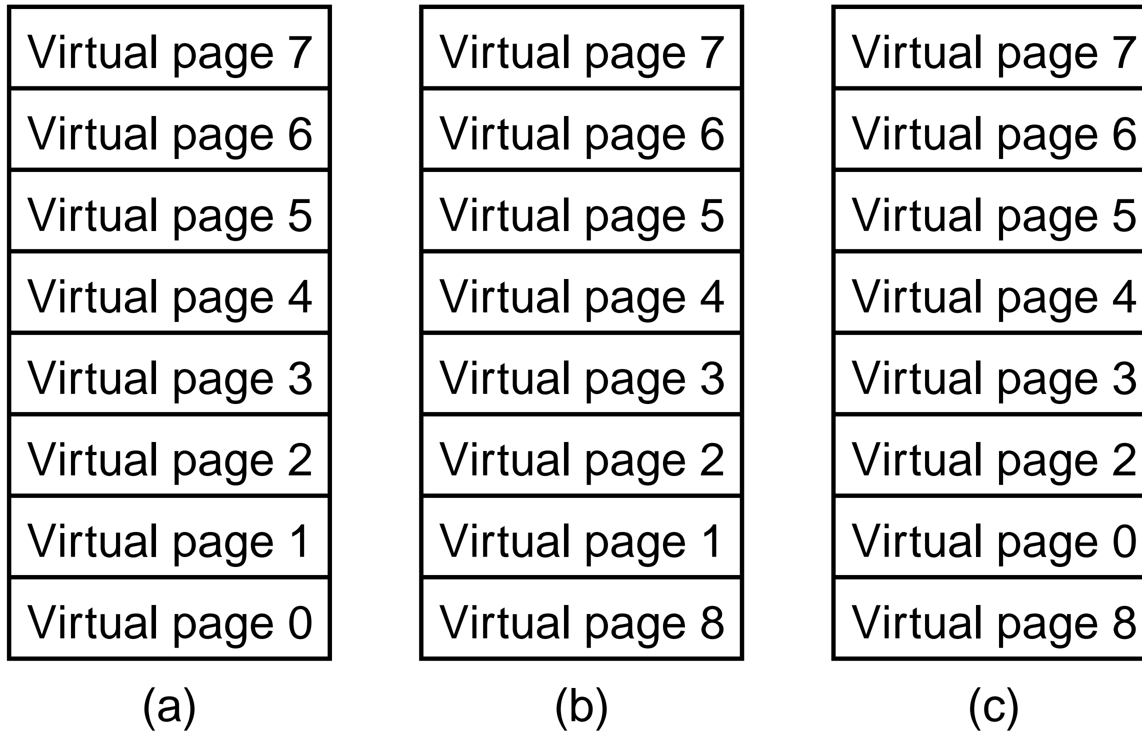

Figure 6-6. Failure of the LRU algorithm.

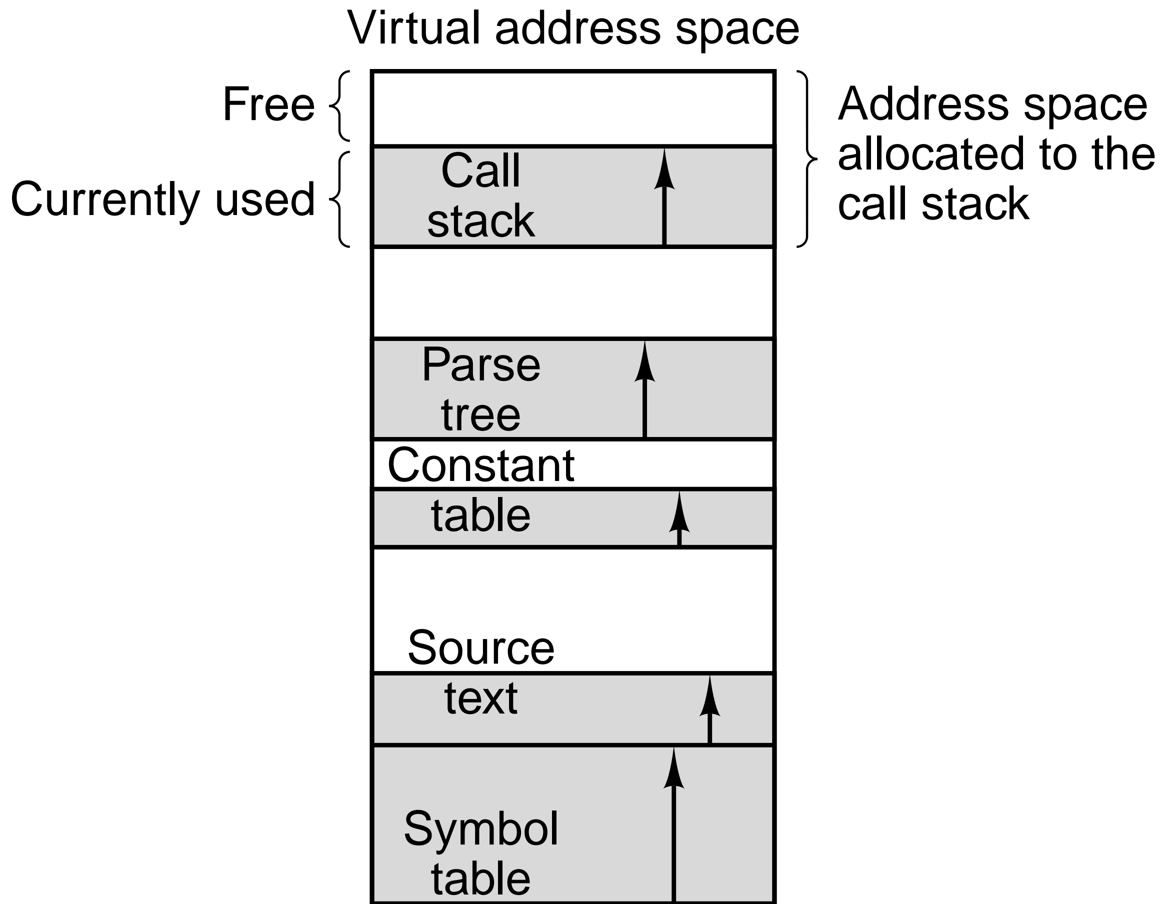

Figure 6-7. In a one-dimensional address space with growing tables, one table may bump into another.

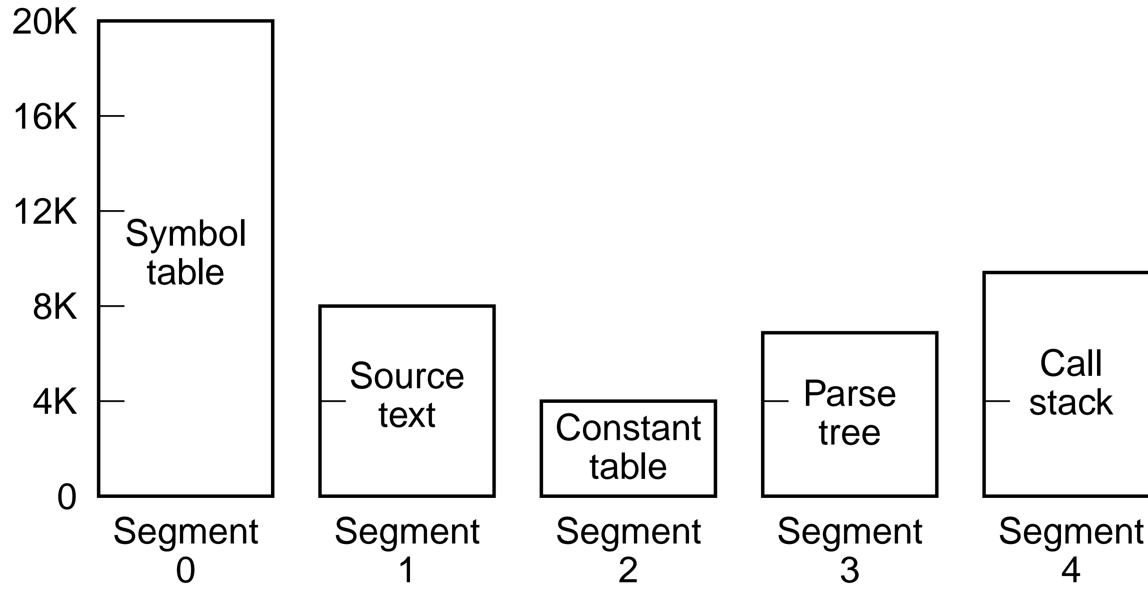

Figure 6-8. A segmented memory allows each table to grow or shrink independently of the other tables.

Consideration	Paging	Segmentation
Need the programmer be aware of it?	No	Yes
How many linear addresses spaces are there?	1	Many
Can virtual address space exceed memory size?	Yes	Yes
Can variable-sized tables be handled easily?	No	Yes
Why was the technique invented?	To simulate large memories	To provide multiple address spaces

Figure 6-9. Comparison of paging and segmentation.

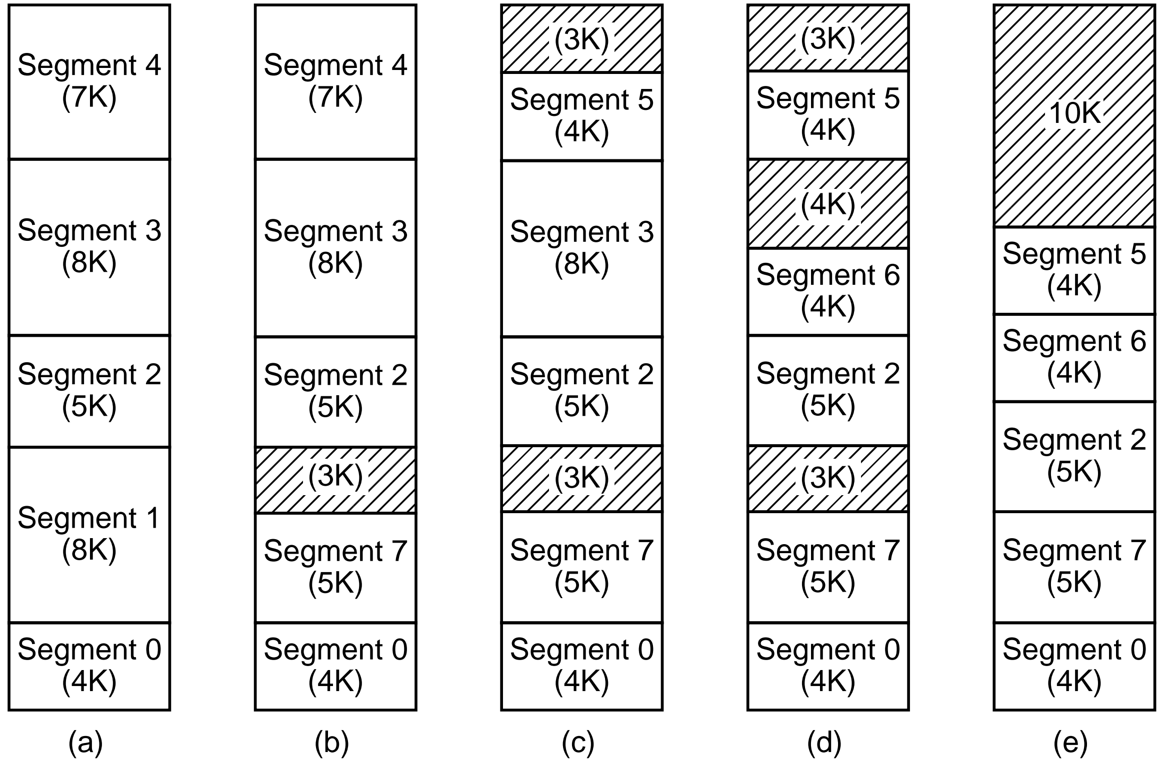

Figure 6-10. (a)-(d) Development of external fragmentation (e) Removal of the external fragmentation by compaction.

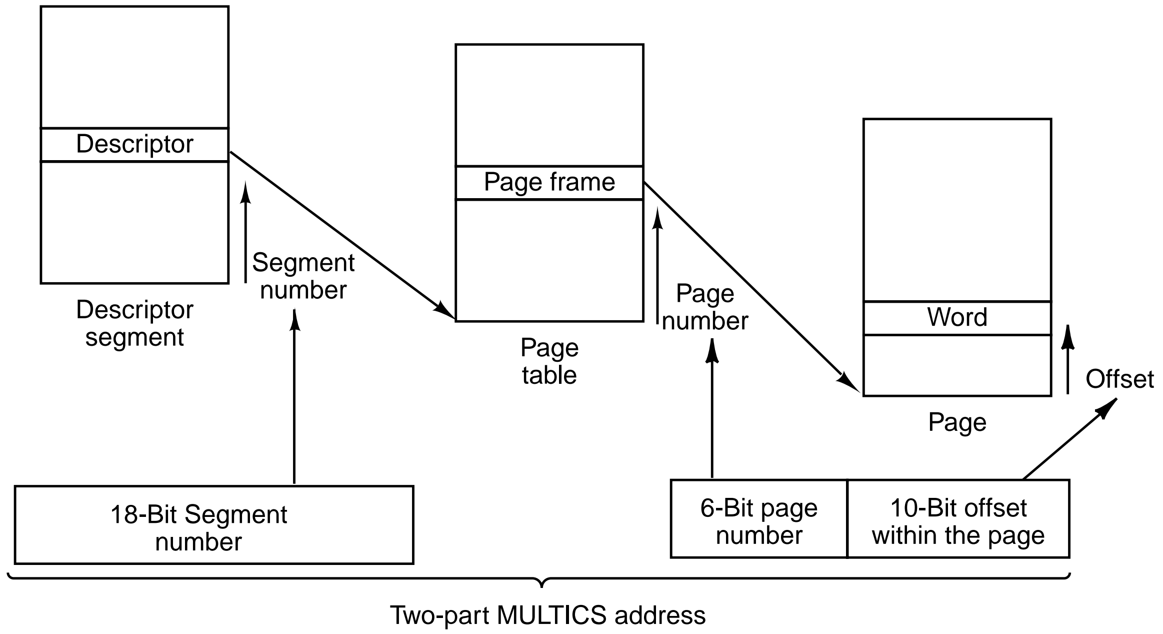

Figure 6-11. Conversion of a two-part MULTICS address into a main memory address.

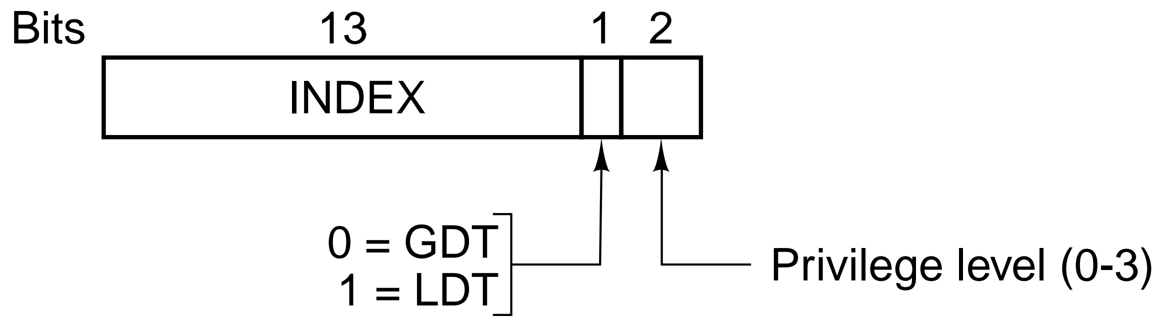

Figure 6-12. A Pentium II selector.

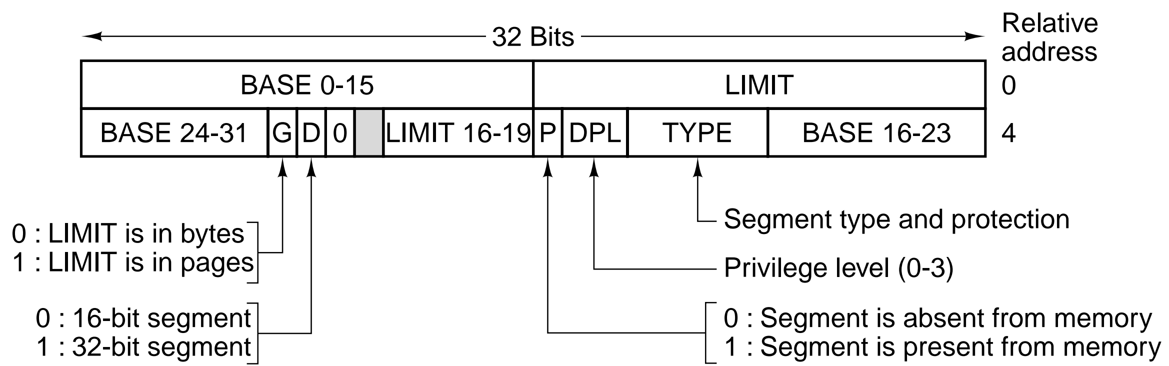

Figure 6-13. A Pentium II code segment descriptor. Data segments differ slightly.

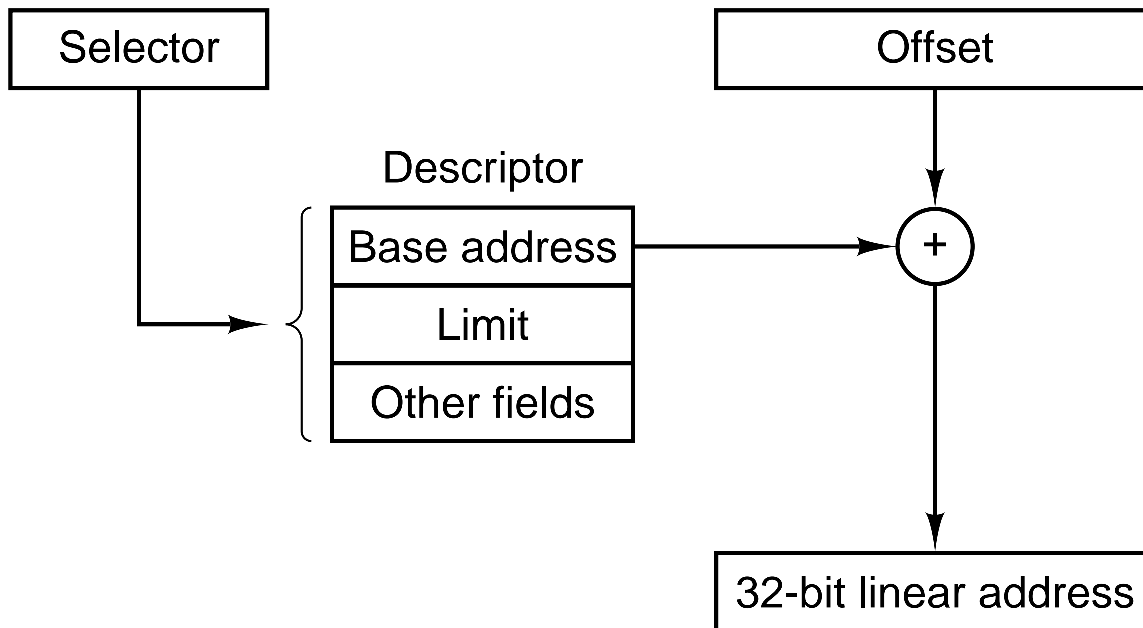

Figure 6-14. Conversion of a (selector, offset) pair to a linear address.

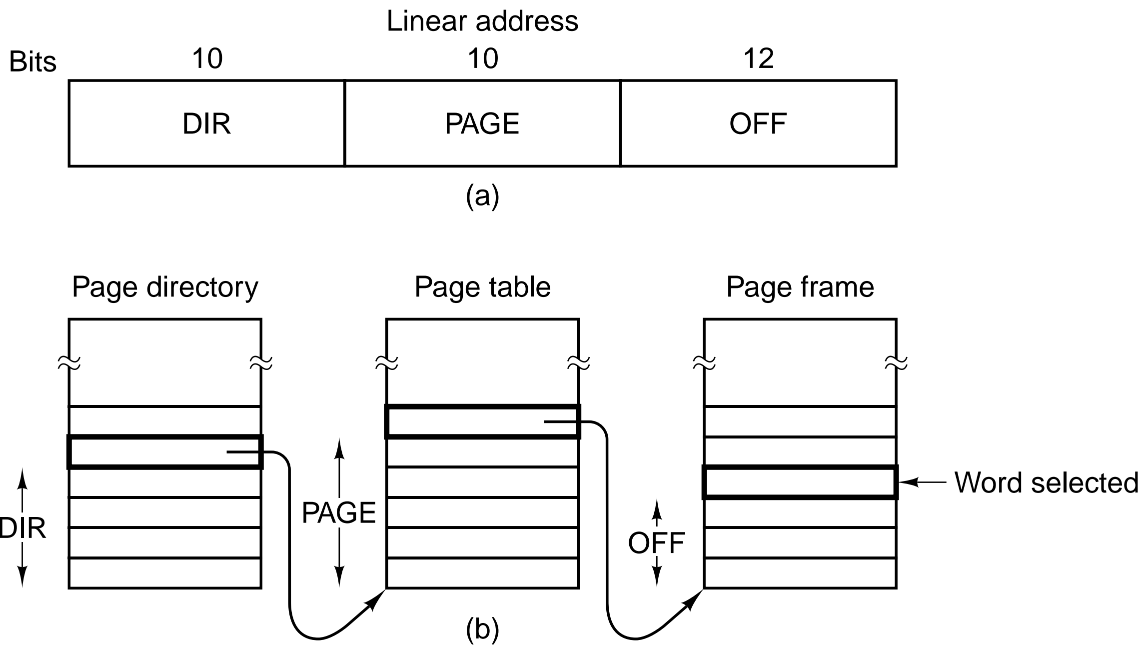

Figure 6-15. Mapping of a linear address onto a physical address.

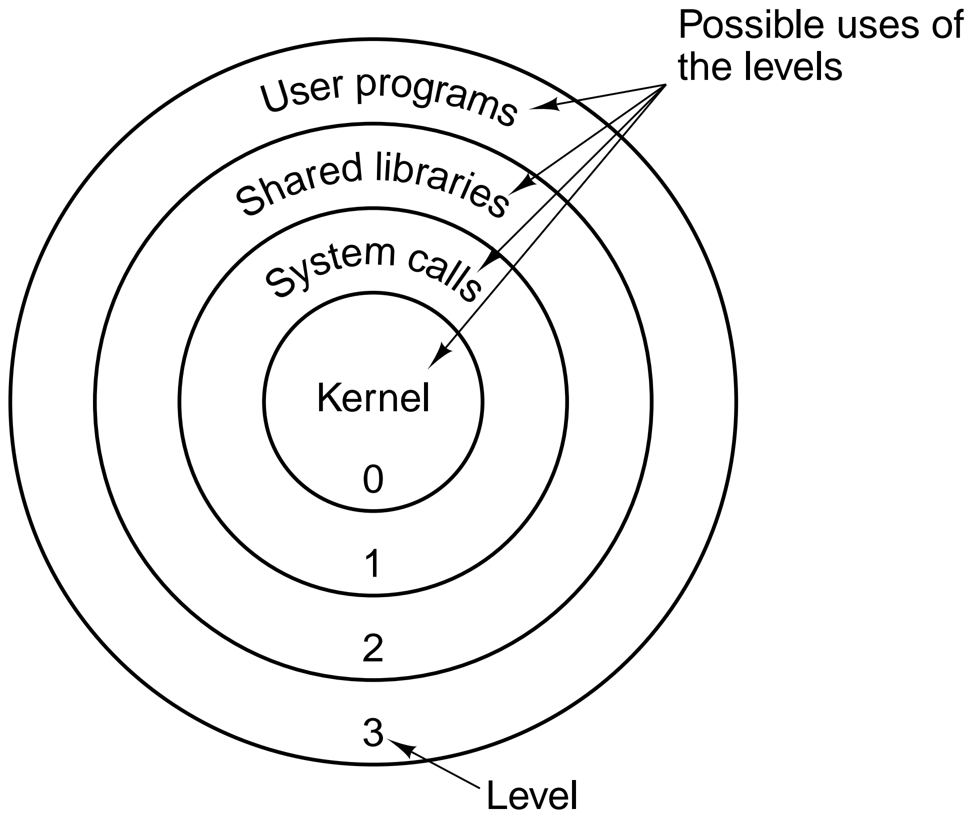

Figure 6-16. Protection on the Pentium II.

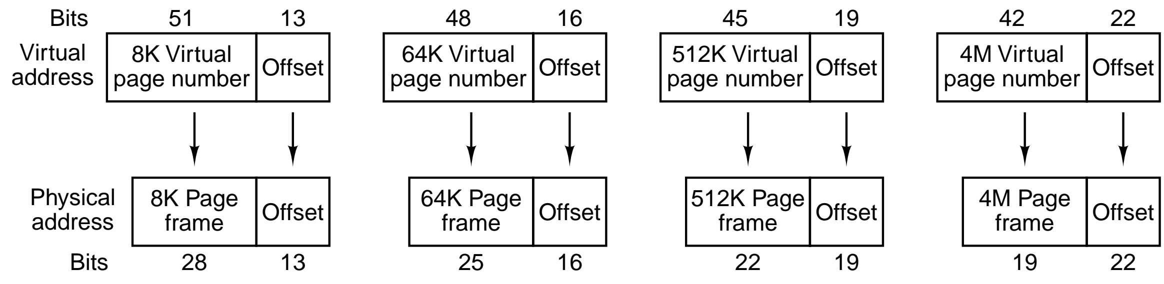

Figure 6-17. Virtual to physical mappings on the UltraSPARC.

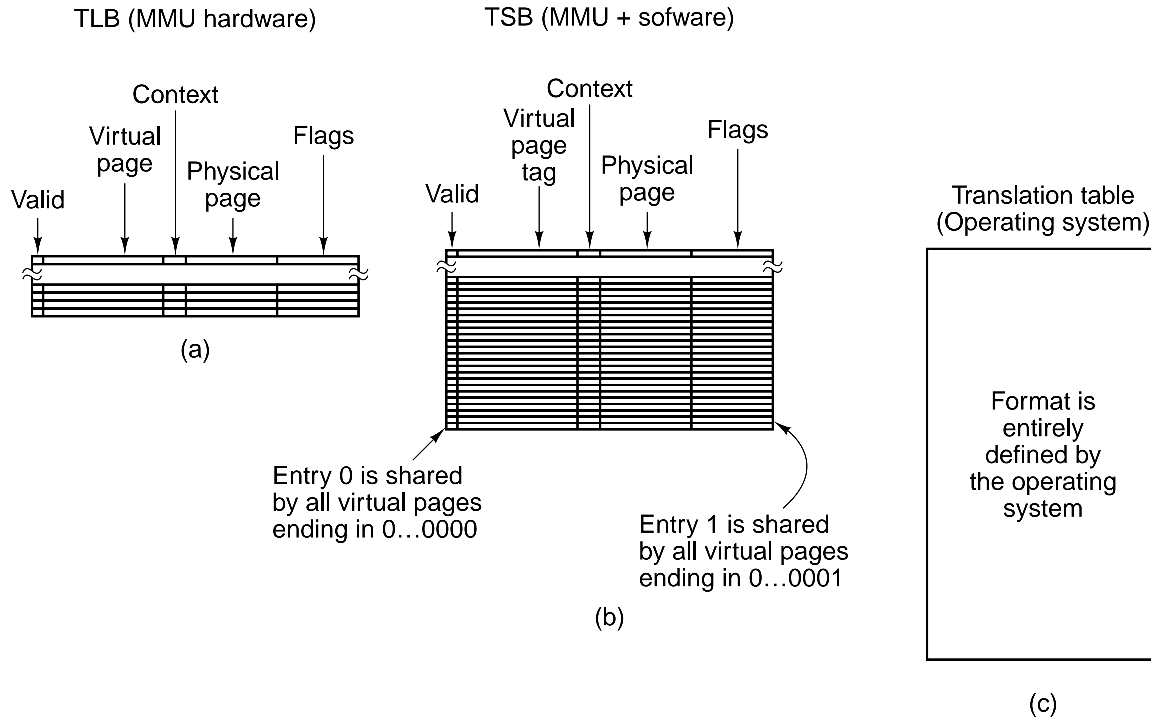

Figure 6-18. Data structures used in translating virtual addresses on the UltraSPARC. (a) TLB. (b) TSB. (c) Translation table.

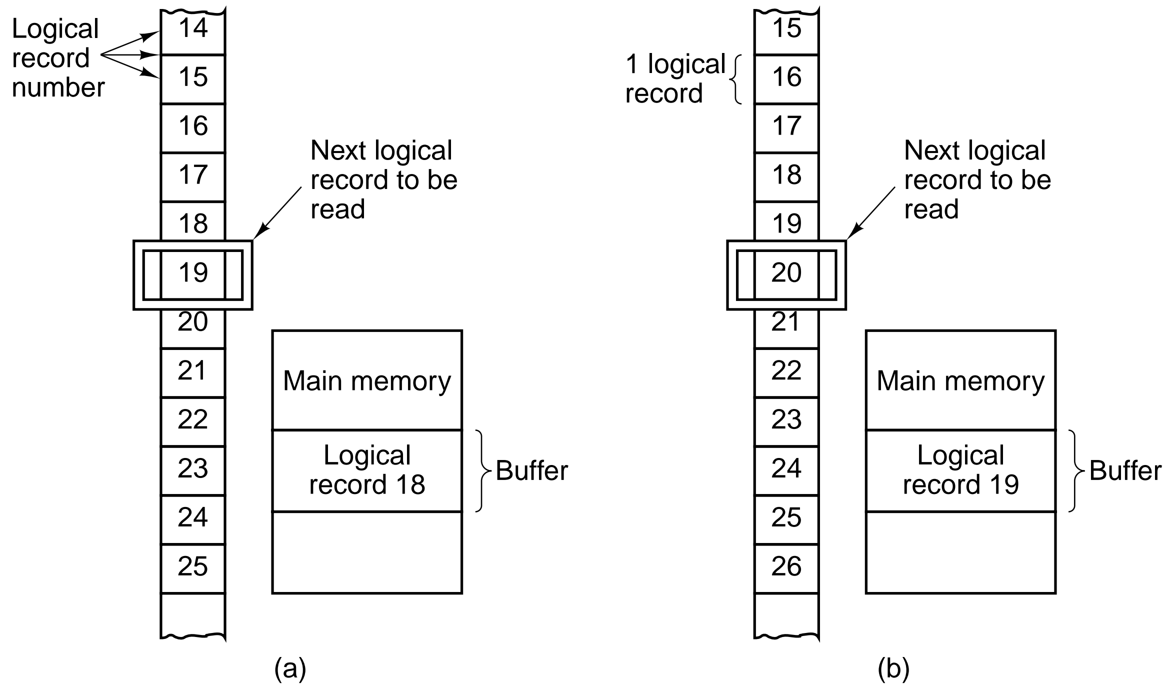

Figure 6-19. Reading a file consisting of logical records. (a) Before reading record 19. (b) After reading record 19.

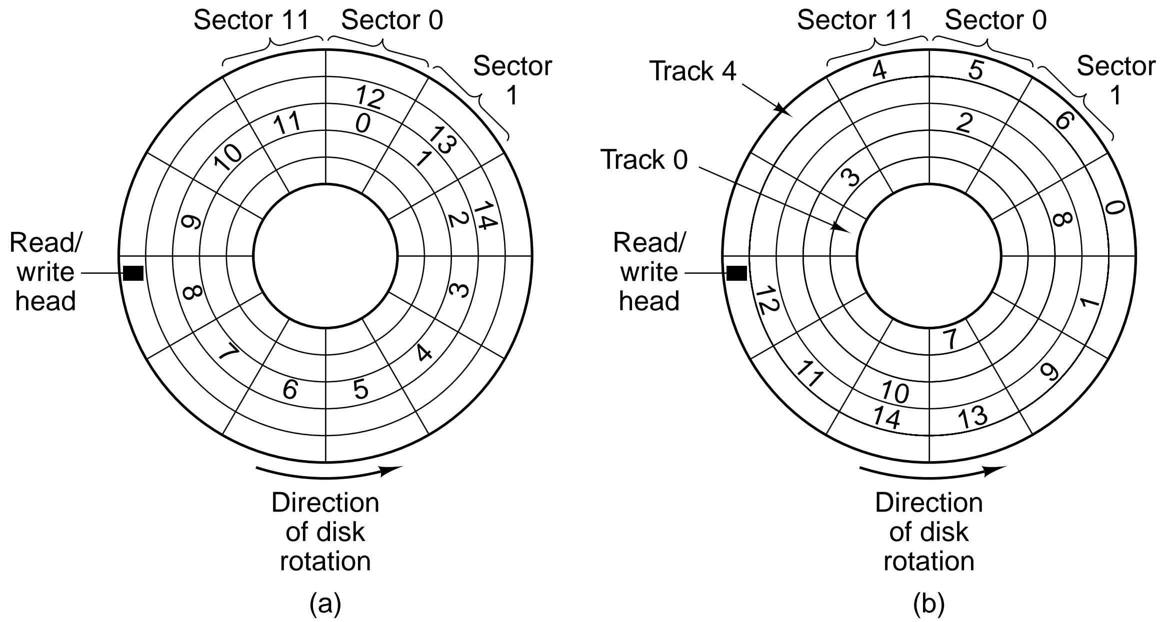

Figure 6-20. Disk allocation strategies. (a) A file in consecutive sectors. (b) A file not in consecutive sectors.

Track	Sector	Number of sectors in hole
0	0	5
0	6	6
1	0	10
1	11	1
2	1	1
2	3	3
2	7	5
3	0	3
3	9	3
4	3	8

(a)

	Sector											
Track	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0
2	1	0	1	0	0	0	1	0	0	0	0	0
3	0	0	0	1	1	1	1	1	1	0	0	0
4	1	1	1	0	0	0	0	0	0	0	0	1

(b)

Figure 6-21. Two ways of keeping track of available sectors. (a) A free list. (b) A bit map.

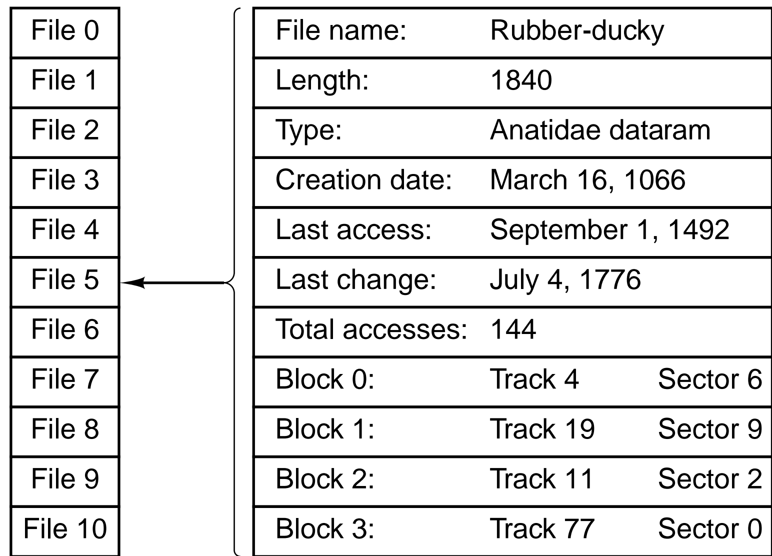

Figure 6-22. (a) A user file directory. (b) The contents of a typical entry in a file directory.

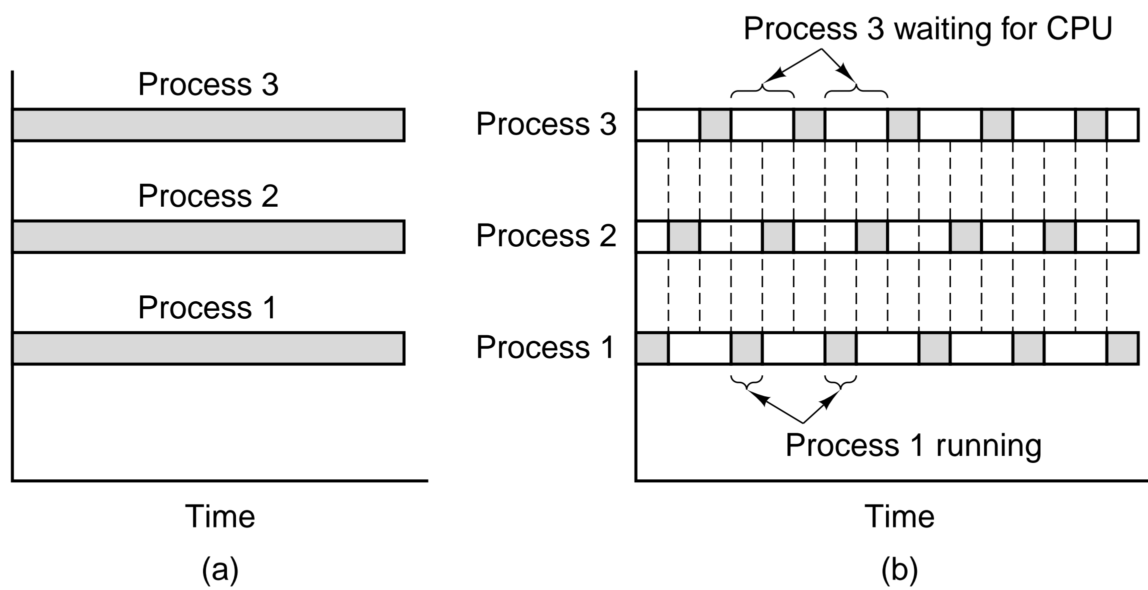

Figure 6-23. (a) True parallel processing with multiple CPUs. (b) Parallel processing simulated by switching one CPU among three processes.

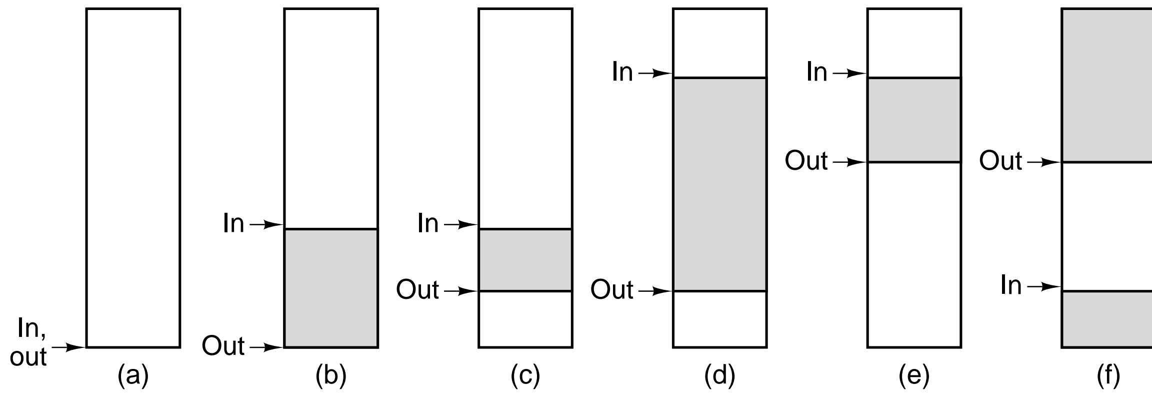

Figure 6-24. Use of a circular buffer.

```
public class m {
   final public static int BUF_SIZE = 100; // buffer runs from 0 to 99
   final public static long MAX_PRIME = 100000000000L; // stop here
   public static int in = 0, out = 0;          // pointers to the data
   public static long buffer[ ] = new long[BUF_SIZE];// primes stored here
   public static producer p;                   // name of the producer
   public static consumer c;                   // name of the consumer

   public static void main(String args[ ]) { // main class
      p = new producer( );                     // create the producer
      c = new consumer( );                     // create the consumer
      p.start( );                              // start the producer
      c.start( );                              // start the consumer
   }

   // This is a utility function for circularly incrementing in and out
   public static int next(int k) {if (k < BUF_SIZE − 1) return(k+1); else return(0);}
}

class producer extends Thread {             // producer class
   public void run( ) {                        // producer code
      long prime = 2;                          // scratch variable

      while (prime < m.MAX_PRIME) {
         prime = next_prime(prime);         // statement P1
         if (m.next(m.in) == m.out) suspend( );       // statement P2
         m.buffer[m.in] = prime;            // statement P3
         m.in = m.next(m.in);               // statement P4
         if (m.next(m.out) == m.in) m.c.resume( );   // statement P5
      }
   }

   private long next_prime(long prime){ ... } // function that computes next prime
}

class consumer extends Thread {            // consumer class
   public void run( ) {                        // consumer code
      long emirp = 2;                          // scratch variable

      while (emirp < m.MAX_PRIME) {
         if (m.in == m.out) suspend( );     // statement C1
         emirp = m.buffer[m.out];           // statement C2
         m.out = m.next(m.out);             // statement C3
         if (m.out == m.next(m.next(m.in))) m.p.resume( );// statement C4
         System.out.println(emirp);         // statement C5
      }
   }
}
```

Figure 6-25. Parallel processing with a fatal race condition.

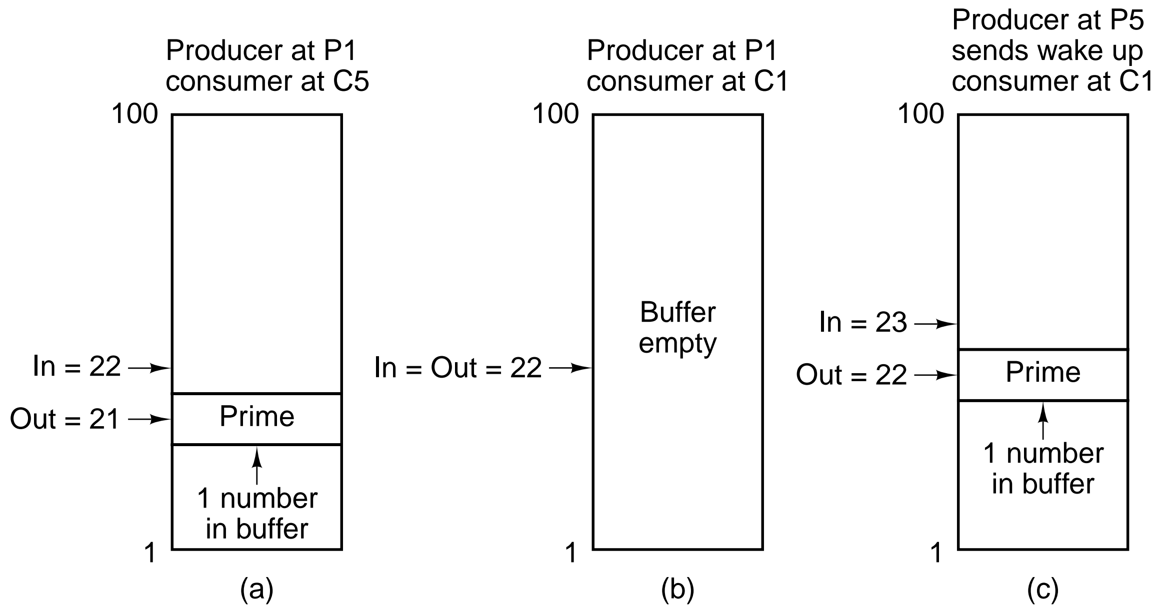

Figure 6-26. Failure of the producer-consumer communication mechanism.

Instr	Semaphore = 0	Semaphore > 0
Up	Semaphore=semaphore+1; if the other process was halted attempting to complete a down instruction on this semaphore, it may now complete the down and continue running	Semaphore=semaphore+1
Down	Process halts until the other process ups this semaphore	Semaphore=semaphore−1

Figure 6-27. The effect of a semaphore operation.

```
public class m {
   final public static int BUF_SIZE = 100; // buffer runs from 0 to 99
   final public static long MAX_PRIME = 100000000000L; // stop here
   public static int in = 0, out = 0;          // pointers to the data
   public static long buffer[ ] = new long[BUF_SIZE];// primes stored here
   public static producer p;                   // name of the producer
   public static consumer c;                   // name of the consumer
   public static int filled = 0, available = 100;        // semaphores

   public static void main(String args[ ]) { // main class
      p = new producer( );                     // create the producer
      c = new consumer( );                     // create the consumer
      p.start( );                              // start the producer
      c.start( );                              // start the consumer
   }

   // This is a utility function for circularly incrementing in and out
   public static int next(int k) {if (k < BUF_SIZE − 1) return(k+1); else return(0);}
}

class producer extends Thread {                // producer class
   native void up(int s); native void down(int s);     // methods on semaphores
   public void run( ) {                        // producer code
      long prime = 2;                          // scratch variable

      while (prime < m.MAX_PRIME) {
         prime = next_prime(prime);            // statement P1
         down(m.available);                    // statement P2
         m.buffer[m.in] = prime;               // statement P3
         m.in = m.next(m.in);                  // statement P4
         up(m.filled);                         // statement P5
      }
   }

   private long next_prime(long prime){ ... } // function that computes next prime
}

class consumer extends Thread {                // consumer class
   native void up(int s); native void down(int s);     // methods on semaphores
   public void run( ) {                        // consumer code
      long emirp = 2;                          // scratch variable

      while (emirp < m.MAX_PRIME) {
         down(m.filled);                       // statement C1
         emirp = m.buffer[m.out];              // statement C2
         m.out = m.next(m.out);                // statement C3
         up(m.available);                      // statement C4
         System.out.println(emirp);            // statement C5
      }
   }
}
```

Figure 6-28. Parallel processing using semaphores.

Category	Some examples
File management	Open, read, write, close, and lock files
Directory management	Create and delete directories; move files around
Process management	Spawn, terminate, trace, and signal processes
Memory management	Share memory among processes; protect pages
Getting/setting parameters	Get user, group, process ID; set priority
Dates and times	Set file access times; use interval timer; profile execution
Networking	Establish/accept connection; send/receive message
Miscellaneous	Enable accounting; manipulate disk quotas; reboot the system

Figure 6-29. A rough breakdown of the UNIX system calls.

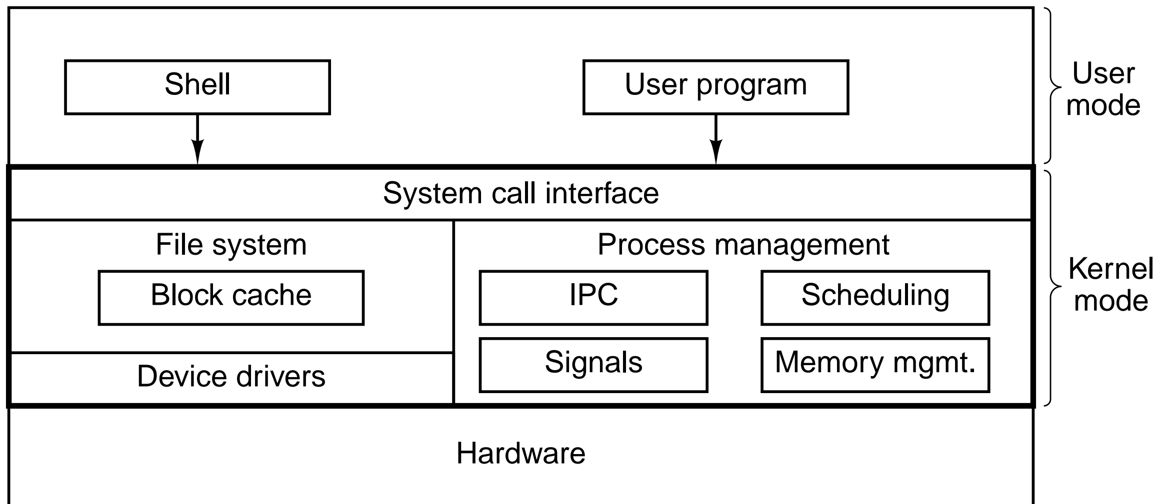

Figure 6-30. The structure of a typical UNIX system.

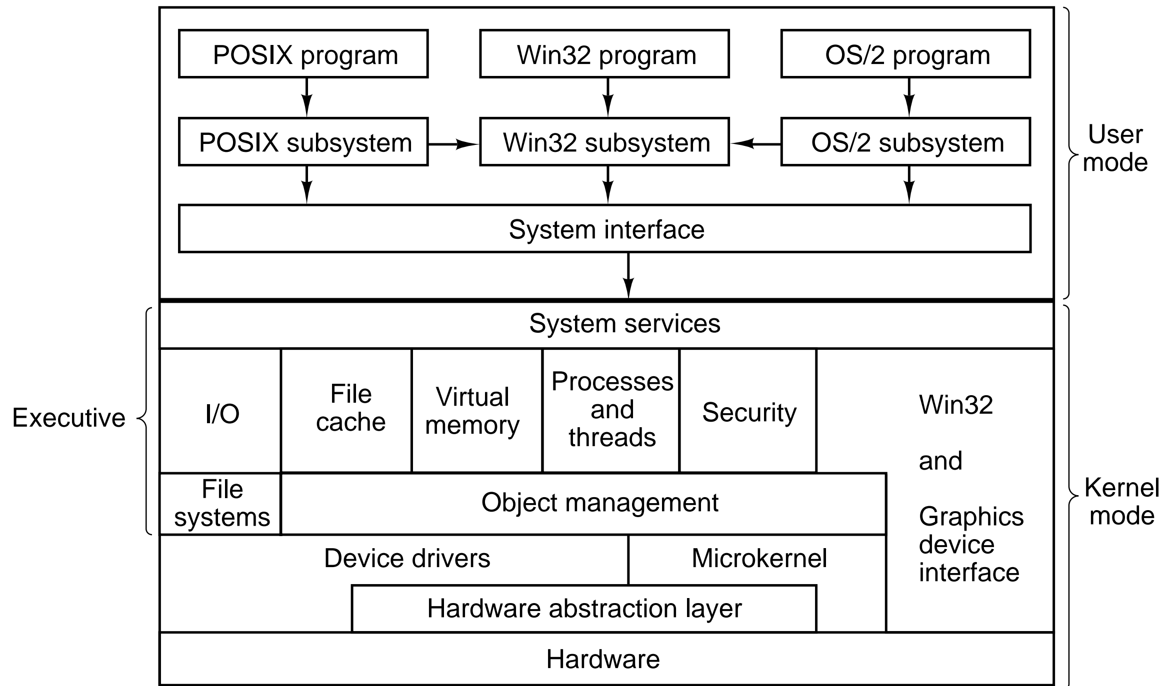

Figure 6-31. The structure of Windows NT.

Item	Windows 95/98	NT 5.0
Win32 API?	Yes	Yes
Full 32-bit system?	No	Yes
Security?	No	Yes
Protected file mappings?	No	Yes
Sep. addr space for each MS-DOS program?	No	Yes
Plug and play?	Yes	Yes
Unicode?	No	Yes
Runs on	Intel 80x86	80x86, Alpha
Multiprocessor support?	No	Yes
Re-entrant code inside OS?	No	Yes
Some critical OS data writable by user?	Yes	No

Figure 6-32. Some differences between versions of Windows.

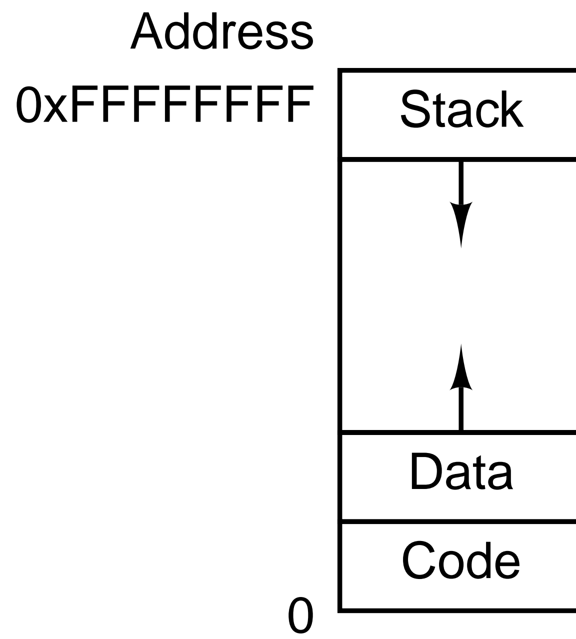

Figure 6-33. The address space of a single UNIX process.

API function	Meaning
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

Figure 6-34. The principal API functions for managing virtual memory in Windows NT.

System call	Meaning
creat(name, mode)	Create a file; *mode* specifies the protection mode
unlink(name)	Delete a file (assuming that there is only 1 link to it)
open(name, mode)	Open or create a file and return a file descriptor
close(fd)	Close a file
read(fd, buffer, count)	Read *count* bytes into *buffer*
write(fd, buffer, count)	Write *count* bytes from *buffer*
lseek(fd, offset, w)	Move the file pointer as required by *offset* and *w*
stat(name, buffer)	Return information about a file
chmod(name, mode)	Change the protection mode of a file
fcntl(fd, cmd, ...)	Do various control operations such as locking (part of) a file

Figure 6-35. The principal UNIX file system calls.

```
// Open the file descriptors
infd = open(″data″, 0);
outfd = creat(″newf″, ProtectionBits);

// Copy loop
do {
    count = read(infd, buffer, bytes);
    if (count > 0) write(outfd, buffer, count);
} while (count > 0);

// Close the files
close(infd);
close(outfd);
```

Figure 6-36. A program fragment for copying a file using the UNIX system calls. This fragment is in C because Java hides the low-level system calls and we are trying to expose them.

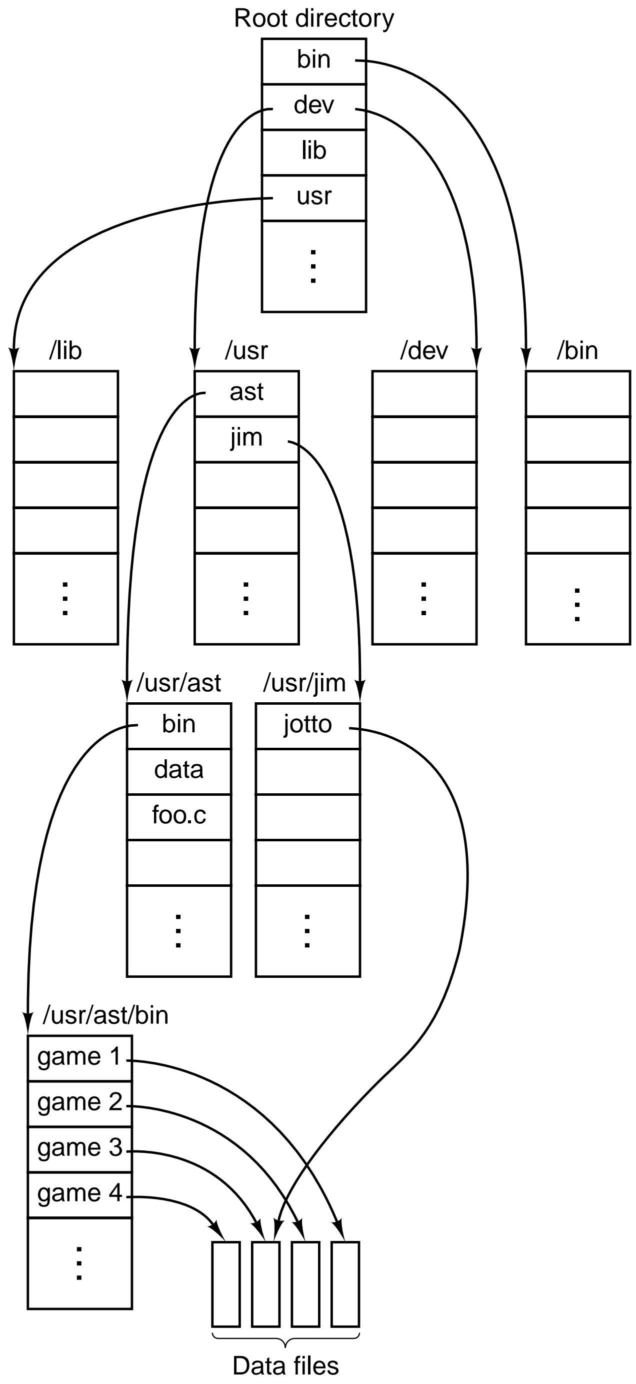

Figure 6-37. Part of a typical UNIX directory system.

System call	Meaning
mkdir(name, mode)	Create a new directory
rmdir(name)	Delete an empty directory
opendir(name)	Open a directory for reading
readdir(dirpointer)	Read the next entry in a directory
closedir(dirpointer)	Close a directory
chdir(dirname)	Change working directory to *dirname*
link(name1, name2)	Create a directory entry *name2* pointing to *name1*
unlink(name)	Remove *name* from its directory

Figure 6-38. The principal UNIX directory management calls.

API function	UNIX	Meaning
CreateFile	open	Create a file or open an existing file; return a handle
DeleteFile	unlink	Destroy an existing file
CloseHandle	close	Close a file
ReadFile	read	Read data from a file
WriteFile	write	Write data to a file
SetFilePointer	lseek	Set the file pointer to a specific place in the file
GetFileAttributes	stat	Return the file properties
LockFile	fcntl	Lock a region of the file to provide mutual exclusion
UnlockFile	fcntl	Unlock a previously locked region of the file

Figure 6-39. The principal Win32 API functions for file I/O. The second column gives the nearest UNIX equivalent.

```
// Open files for input and output.
inhandle = CreateFile("data", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
outhandle = CreateFile("newf", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL, NULL);

// Copy the file.
do {
        s = ReadFile(inhandle, buffer, BUF_SIZE, &count, NULL);
        if (s > 0 && count > 0) WriteFile(outhandle, buffer, count, &ocnt, NULL);
while (s > 0 && count > 0);

// Close the files.
CloseHandle(inhandle);
CloseHandle(outhandle);
```

Figure 6-40. A program fragment for copying a file using the Windows NT API functions. This fragment is in C because Java hides the low-level system calls and we are trying to expose them.

API function	UNIX	Meaning
CreateDirectory	mkdir	Create a new directory
RemoveDirectory	rmdir	Remove an empty directory
FindFirstFile	opendir	Initialize to start reading the entries in a directory
FindNextFile	readdir	Read the next directory entry
MoveFile		Move a file from one directory to another
SetCurrentDirectory	chdir	Change the current working directory

Figure 6-41. The principal Win32 API functions for directory management. The second column gives the nearest UNIX equivalent, when one exists.

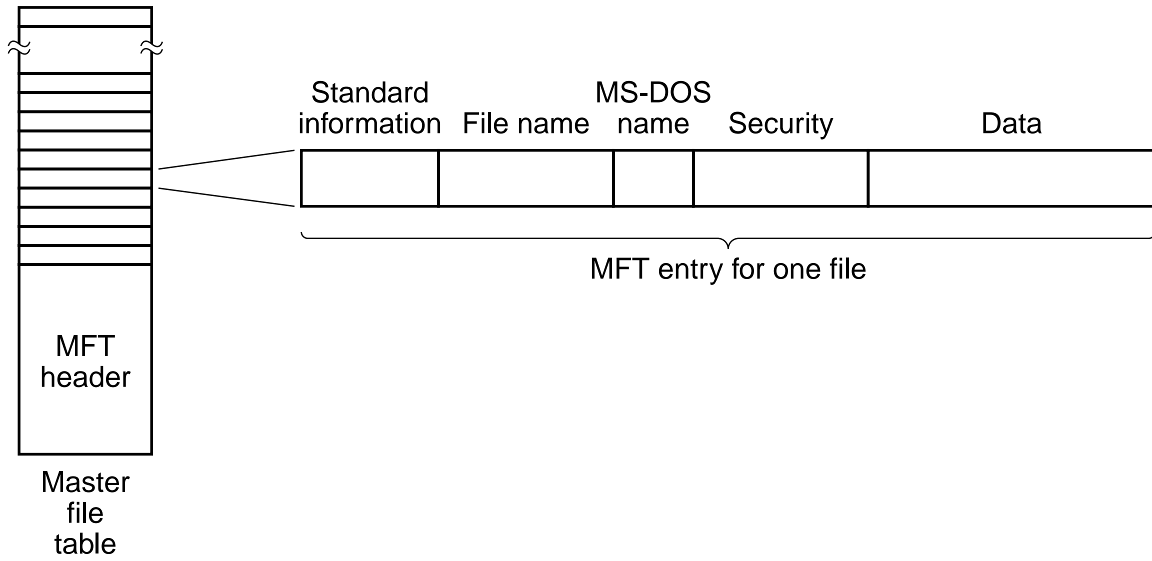

Figure 6-42. The Windows NT master file table.

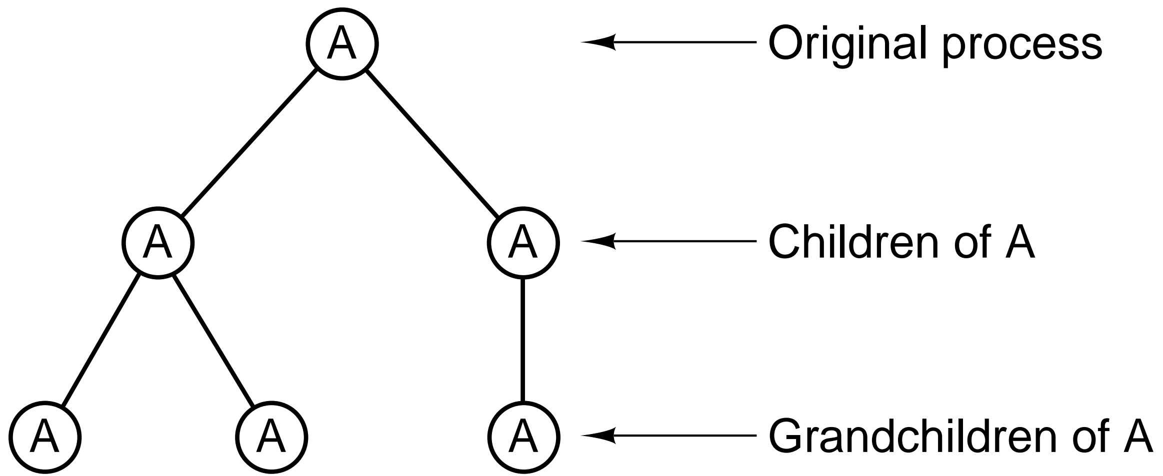

Figure 6-43. A process tree in UNIX.

Thread call	Meaning
pthread_create	Create a new thread in the caller's address space
pthread_exit	Terminate the calling thread
pthread_join	Wait for a thread to terminate
pthread_mutex_init	Create a new mutex
pthread_mutex_destroy	Destroy a mutex
pthread_mutex_lock	Lock a mutex
pthread_mutex_unlock	Unlock a mutex
pthread_cond_init	Create a condition variable
pthread_cond_destroy	Destroy a condition variable
pthread_cond_wait	Wait on a condition variable
pthread_cond_signal	Release one thread waiting on a condition variable

Figure 6-44. The principal POSIX thread calls.