OPERATING SYSTEMS: DESIGN AND IMPLEMENTATION

Second Edition

ANDREW S. TANENBAUM
Vrije Universiteit
Amsterdam, The Netherlands

ALBERT S. WOODHULL
Hampshire College
Amherst, Massachusetts

PRENTICE HALL
Upper Saddle River, NJ 07458

1

INTRODUCTION

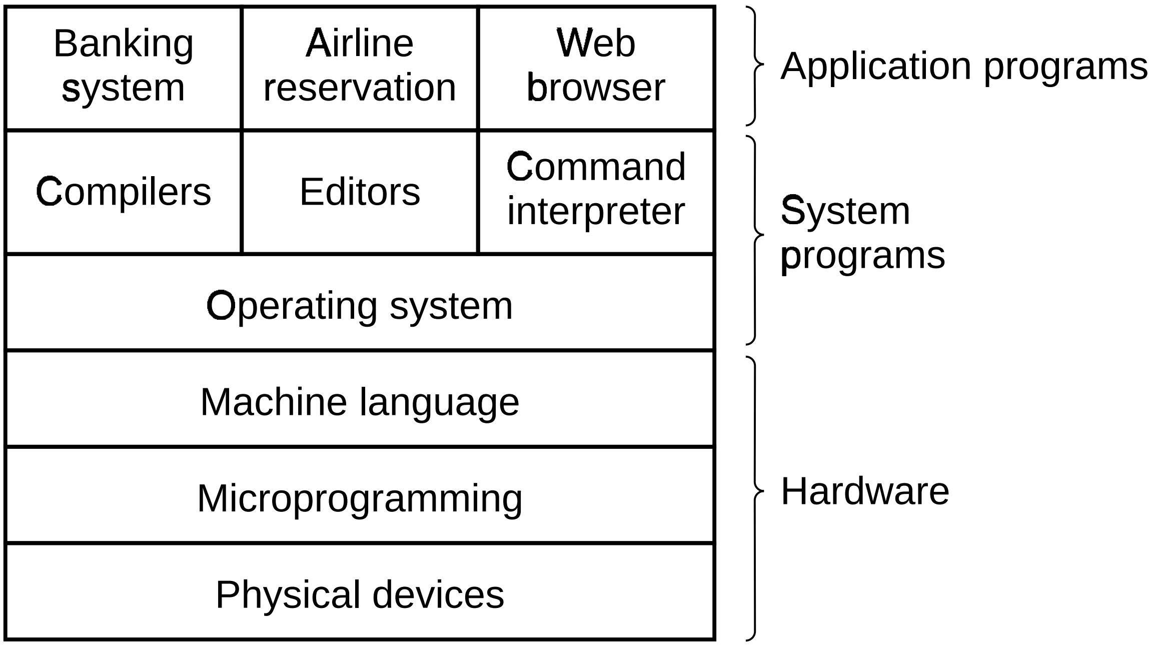


Figure 1-1. A computer system consists of hardware, system programs, and application programs.

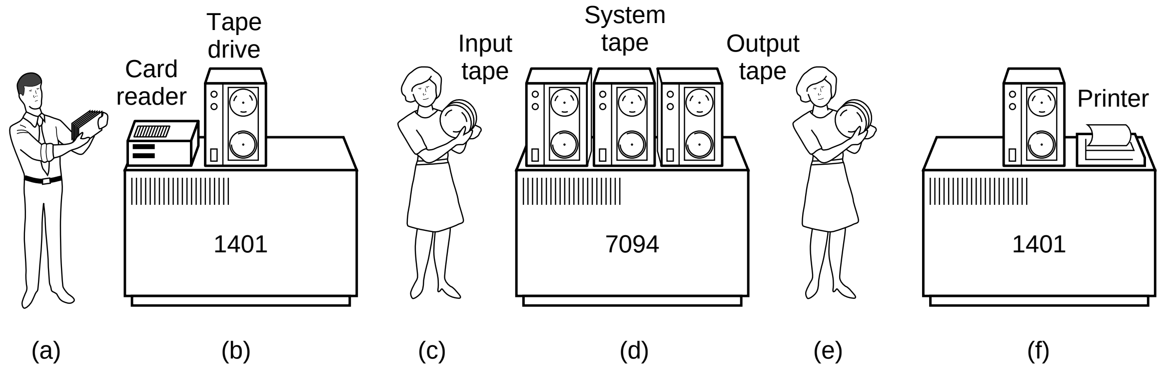


Figure 1-2. An early batch system. (a) Programmers bring cards to 1401. (b) 1401 reads batch of jobs onto tape. (c) Operator carries input tape to 7094. (d) 7094 does computing. (e) Operator carries output tape to 1401. (f) 1401 prints output.

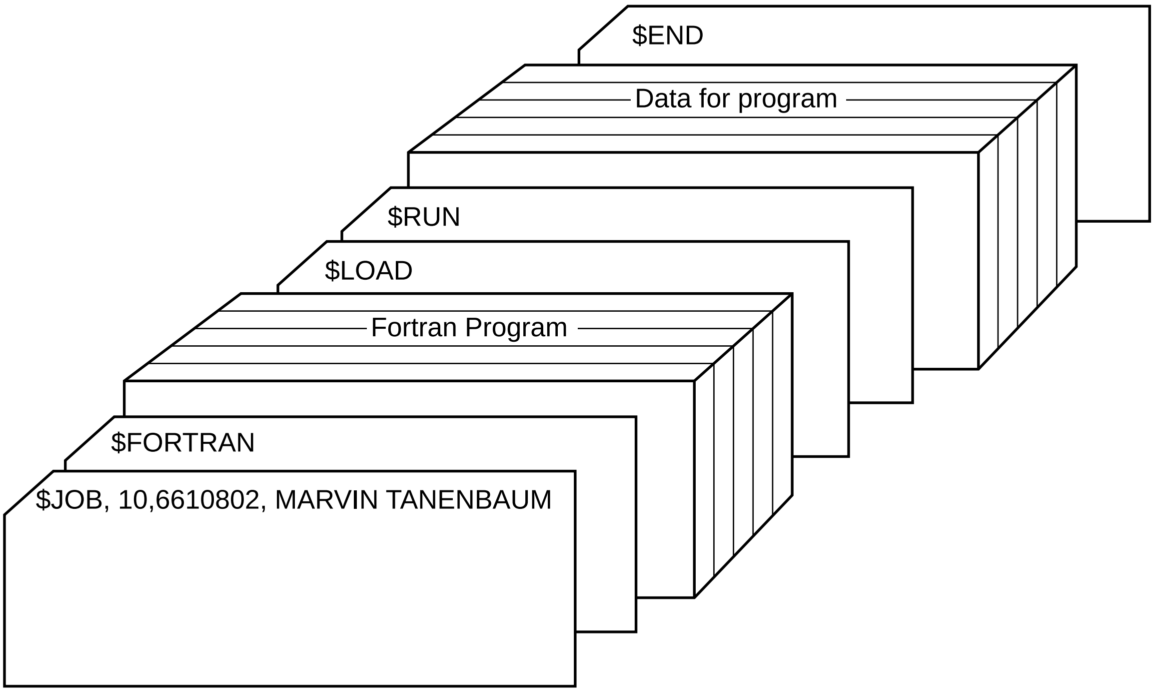


Figure 1-3. Structure of a typical FMS job.

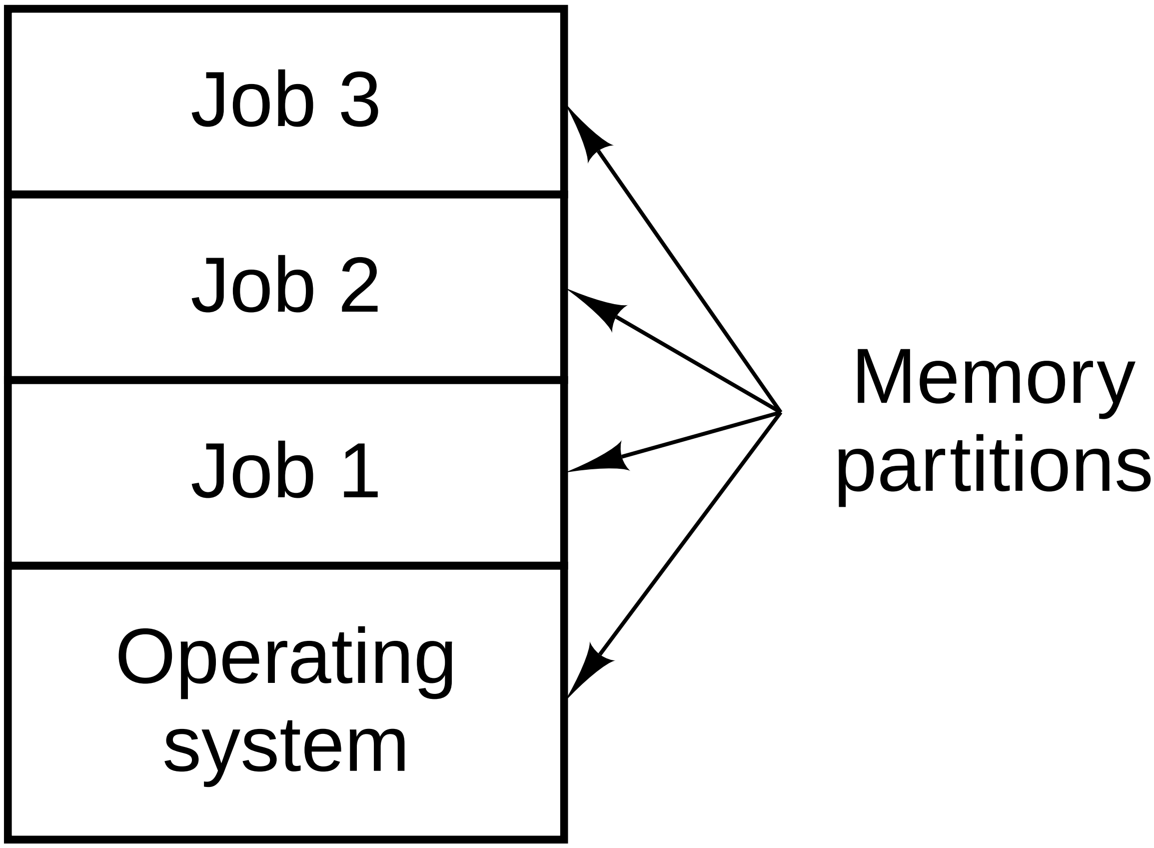


Figure 1-4. A multiprogramming system with three jobs in memory.

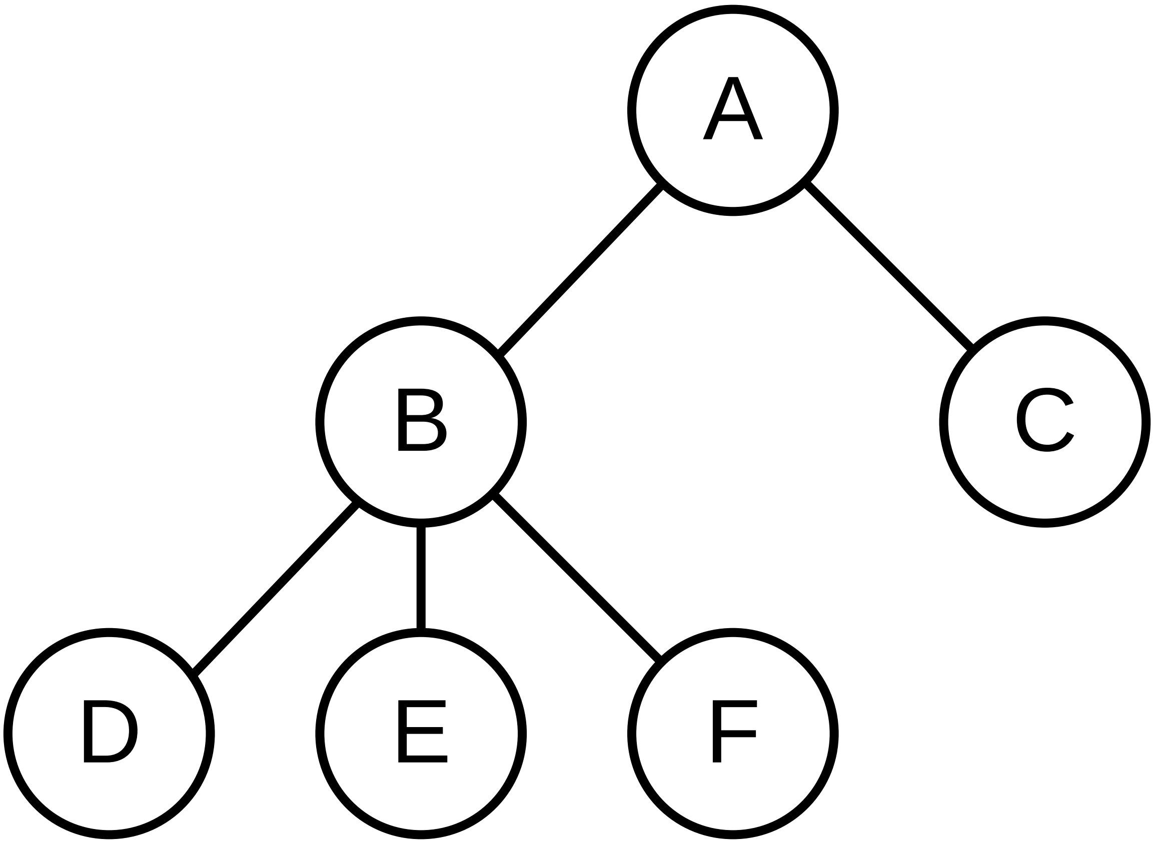


Figure 1-5. A process tree. Process *A* created two child processes, *B* and *C*. Process *B* created three child processes, *D*, *E*, and *F*.

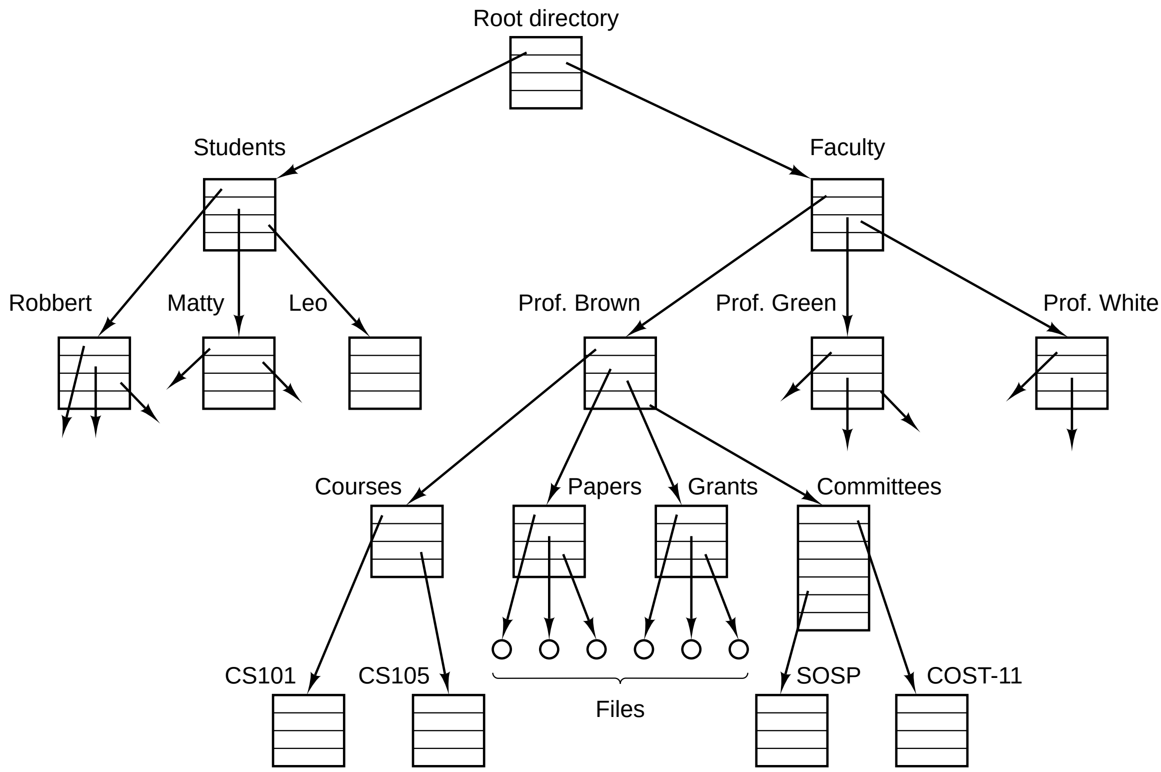


Figure 1-6. A file system for a university department.

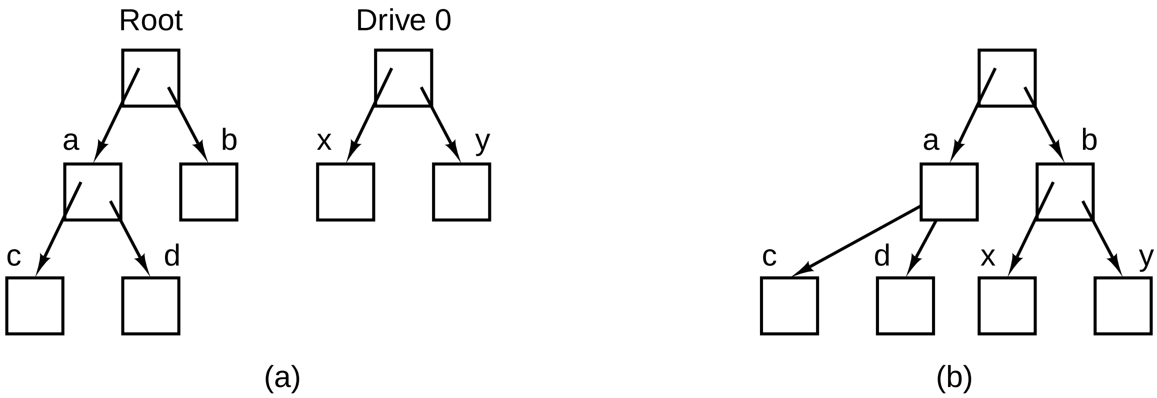


Figure 1-7. (a) Before mounting, the files on drive 0 are not accessible. (b) After mounting, they are part of the file hierarchy.

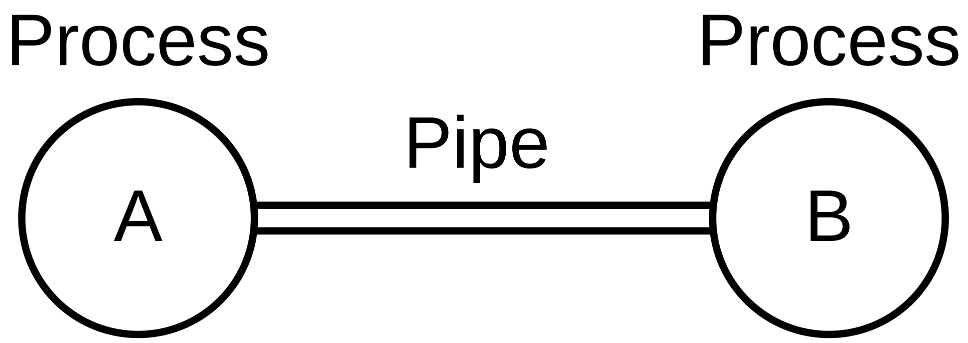


Figure 1-8. Two processes connected by a pipe.

Process management	pid = fork()	Create a child process identical to the parent
	pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
	s = wait(&status)	Old version of waitpid
	s = execve(name, argv, envp)	Replace a process core image
	exit(status)	Terminate process execution and return status
	size = brk(addr)	Set the size of the data segment
	pid = getpid()	Return the caller's process id
	pid = getpgrp()	Return the id of the caller's process group
	pid = setsid()	Create a new session and return its process group id
	l = ptrace(req, pid, addr, data)	Used for debugging
Signals	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
File Management	fd = creat(name, mode)	Obsolete way to create a new file
	fd = mknod(name, mode, addr)	Create a regular, special, or directory i-node
	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
	pos = 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
	fd = dup(fd)	Allocate a new file descriptor for an open file
	s = pipe(&fd[0])	Create a pipe
	s = ioctl(fd, request, argp)	Perform special operations on a file
	s = access(name, amode)	Check a file's accessibility
	s = rename(old, new)	Give a file a new name
	s = fcntl(fd, cmd, ...)	File locking and other operations
Directory & File System Management	s = mkdir(name, mode)	Create a new directory
	s = rmdir(name)	Remove an empty directory
	s = link(name1, name2)	Create a new entry, name2, pointing to name1
	s = unlink(name)	Remove a directory entry
	s = mount(special, name, flag)	Mount a file system
	s = umount(special)	Unmount a file system
	s = sync()	Flush all cached blocks to the disk
	s = chdir(dirname)	Change the working directory
	s = chroot(dirname)	Change the root directory
Protection	s = chmod(name, mode)	Change a file's protection bits
	uid = getuid()	Get the caller's uid
	gid = getgid()	Get the caller's gid
	s = setuid(uid)	Set the caller's uid
	s = setgid(gid)	Set the caller's gid
	s = chown(name, owner, group)	Change a file's owner and group
	oldmask = umask(complmode)	Change the mode mask
Time Management	seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970
	s = stime(tp)	Set the elapsed time since Jan. 1, 1970
	s = utime(file, timep)	Set a file's "last access" time
	s = times(buffer)	Get the user and system times used so far

Figure 1-9. The MINIX system calls.

```
while (TRUE) {                          /* repeat forever */
  read_command(command, parameters);/* read input from terminal */

  if (fork() != 0) {                    /* fork off child process */
      /* Parent code. */
      waitpid(–1, &status, 0);          /* wait for child to exit */
  } else {
      /* Child code. */
      execve(command, parameters, 0);/* execute command */
  }
}
```

Figure 1-10. A stripped-down shell. Throughout this book, *TRUE* is assumed to be defined as 1.

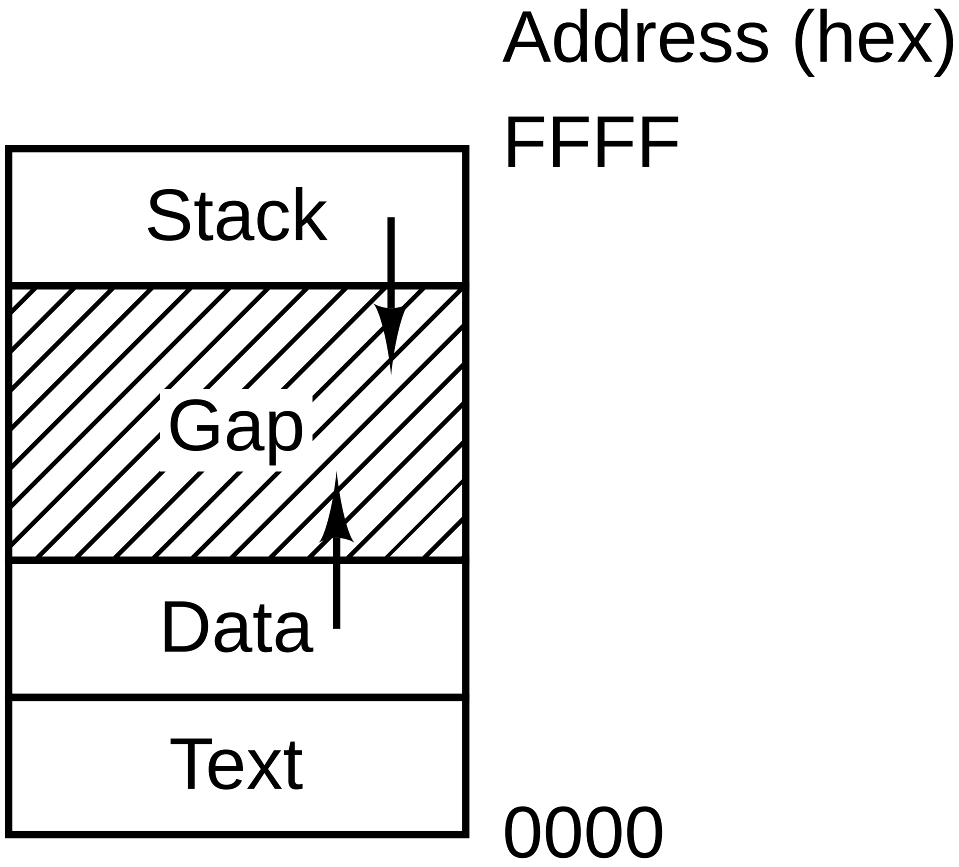


Figure 1-11. Processes have three segments: text, data, and stack. In this example, all three are in one address space, but separate instruction and data space is also supported.

```
struct stat {
    short st_dev;                /* device where i-node belongs */
    unsigned short st_ino;       /* i-node number */
    unsigned short st_mode;      /* mode word */
    short st_nlink;              /* number of links */
    short st_uid;                /* user id */
    short st_gid;                /* group id */
    short st_rdev;               /* major/minor device for special files */
    long st_size;                /* file size */
    long st_atime;               /* time of last access */
    long st_mtime;               /* time of last modification */
    long st_ctime;               /* time of last change to i-node */
};
```

Figure 1-12. The structure used to return information for the STAT and FSTAT system calls. In the actual code, symbolic names are used for some of the types.

```
#define STD_INPUT 0         /* file descriptor for standard input */
#define STD_OUTPUT 1        /* file descriptor for standard output */

pipeline(process1, process2)
char *process1, *process2;  /* pointers to program names */
{
  int fd[2];

  pipe(&fd[0]);                /* create a pipe */
  if (fork() != 0) {
       /* The parent process executes these statements. */
       close(fd[0]);           /* process 1 does not need to read from pipe */
       close(STD_OUTPUT);      /* prepare for new standard output */
       dup(fd[1]);             /* set standard output to fd[1] */
       close(fd[1]);           /* this file descriptor not needed any more */
       execl(process1, process1, 0);
  } else {
       /* The child process executes these statements. */
       close(fd[1]);           /* process 2 does not need to write to pipe */
       close(STD_INPUT);       /* prepare for new standard input */
       dup(fd[0]);             /* set standard input to fd[0] */
       close(fd[0]);           /* this file descriptor not needed any more */
       execl(process2, process2, 0);
  }
}
```

Figure 1-13. A skeleton for setting up a two-process pipeline.

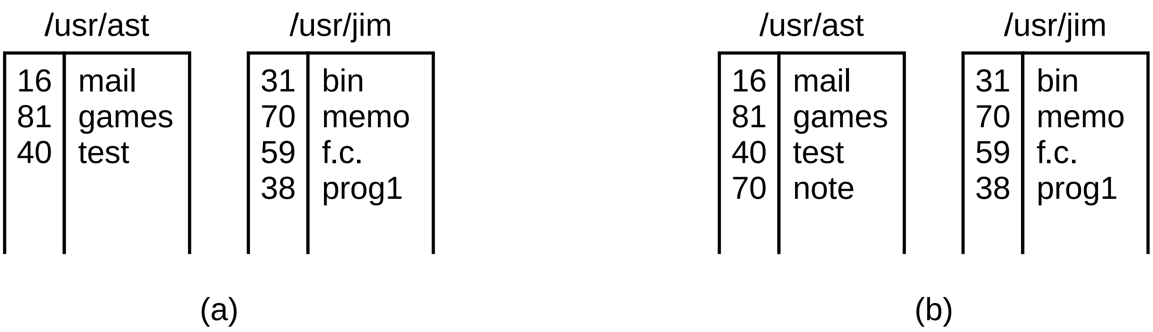


Figure 1-14. (a) Two directories before linking */usr/jim/memo* to ast's directory. (b) The same directories after linking.

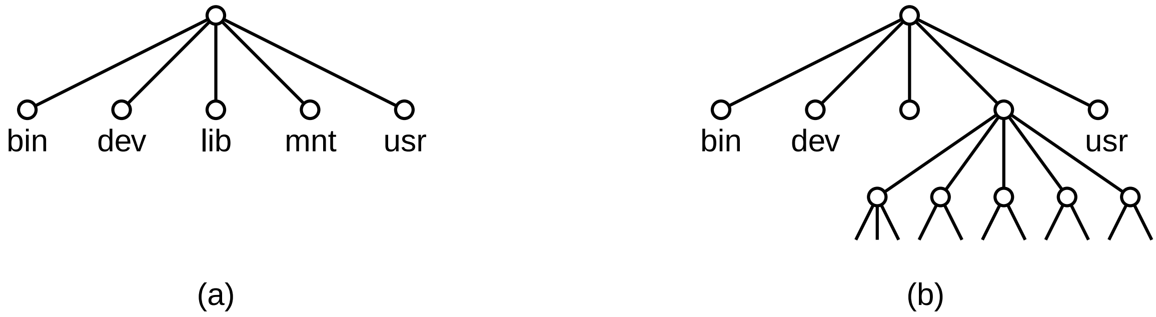


Figure 1-15. (a) File system before the mount. (b) File system after the mount.

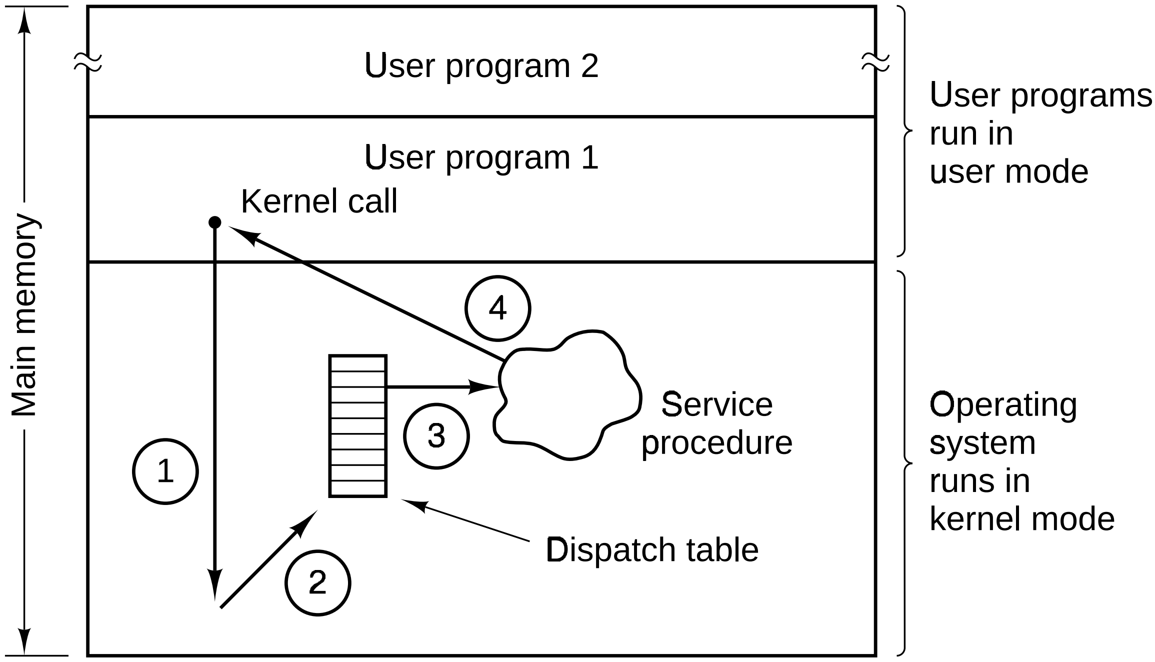


Figure 1-16. How a system call can be made: (1) User program traps to the kernel. (2) Operating system determines service number required. (3) Operating system calls service procedure. (4) Control is returned to user program.

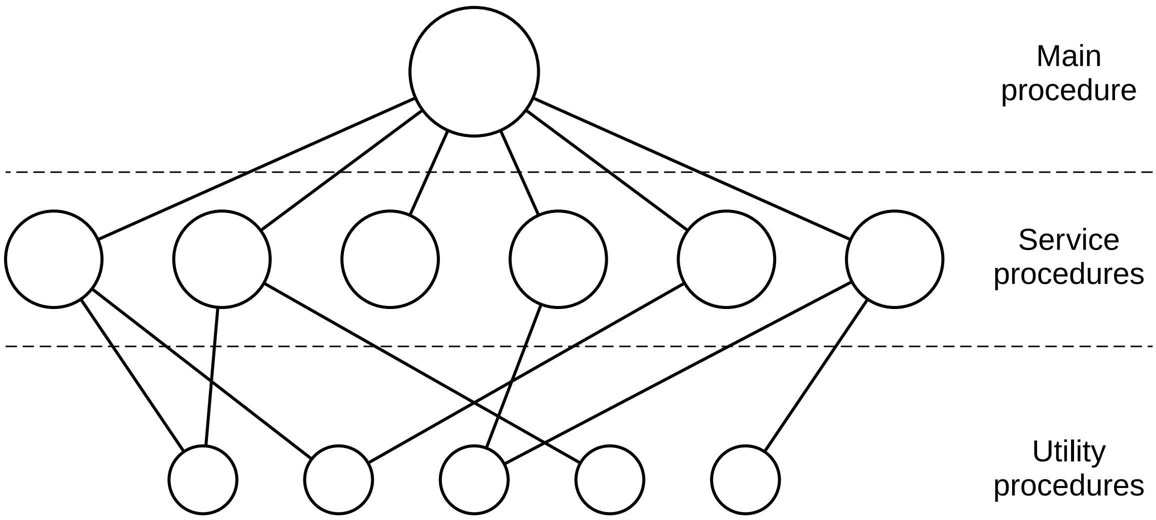


Figure 1-17. A simple structuring model for a monolithic system.

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

Figure 1-18. Structure of the THE operating system.

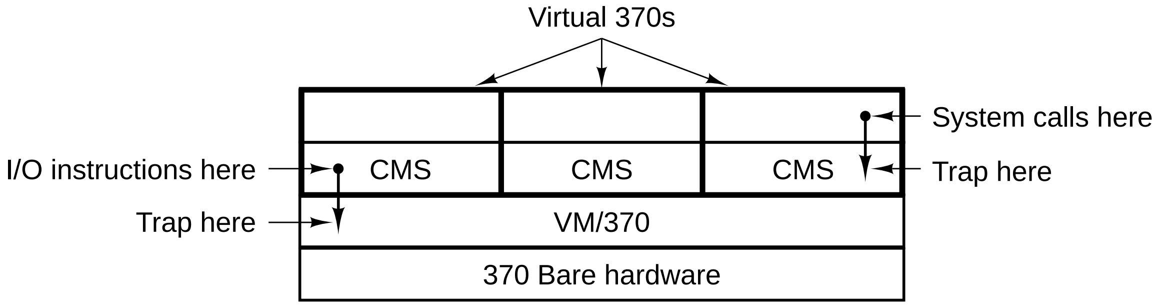


Figure 1-19. The structure of VM/370 with CMS.

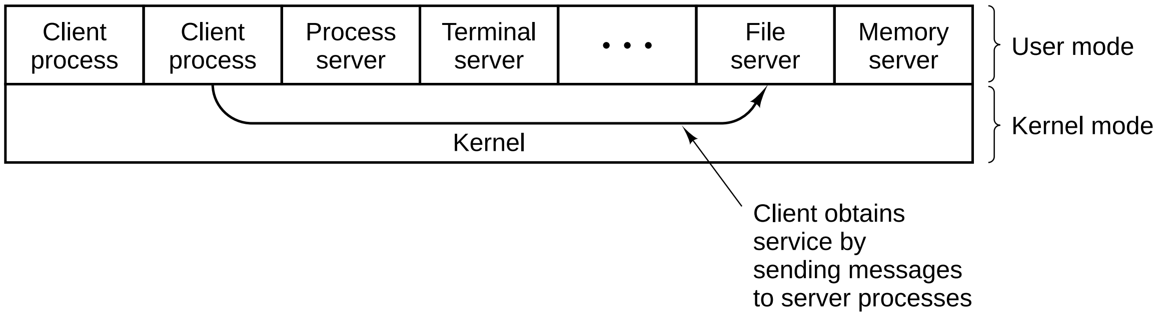


Figure 1-20. The client-server model.

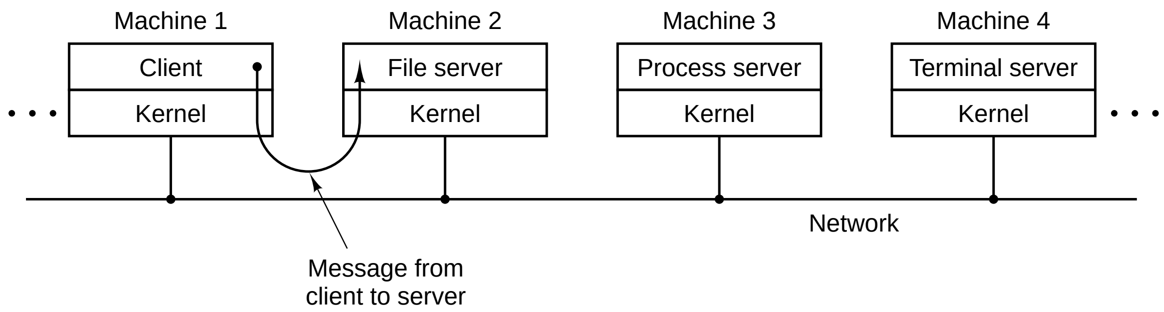


Figure 1-21. The client-server model in a distributed system.

2

PROCESSES

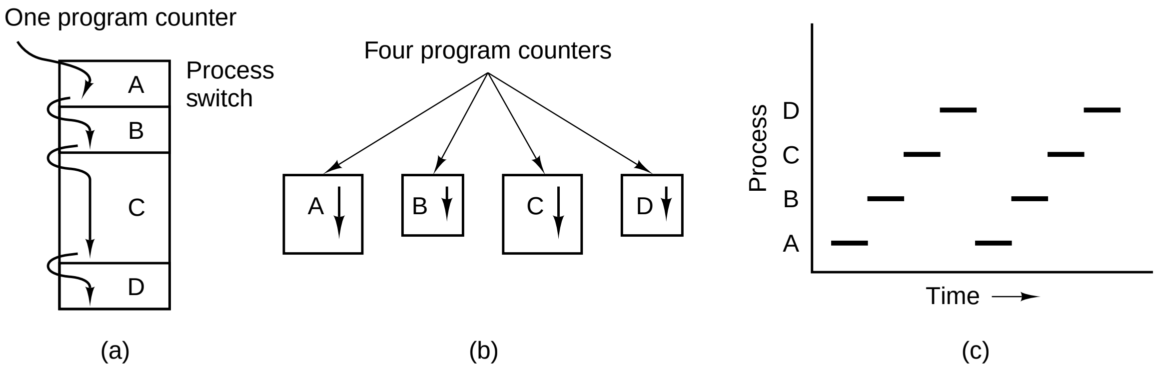


Figure 2-1. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at any instant.

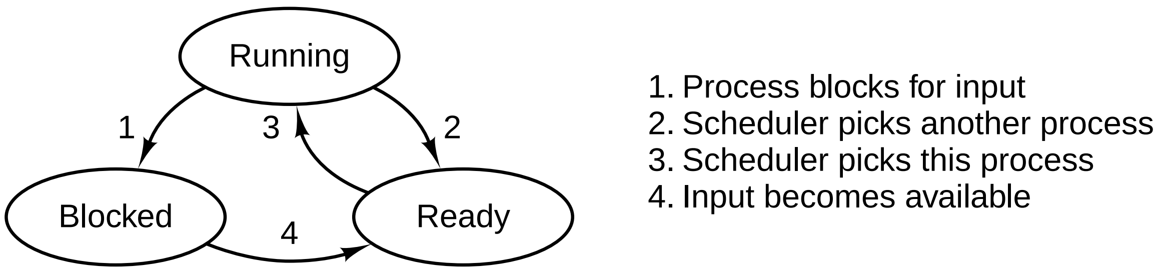


Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

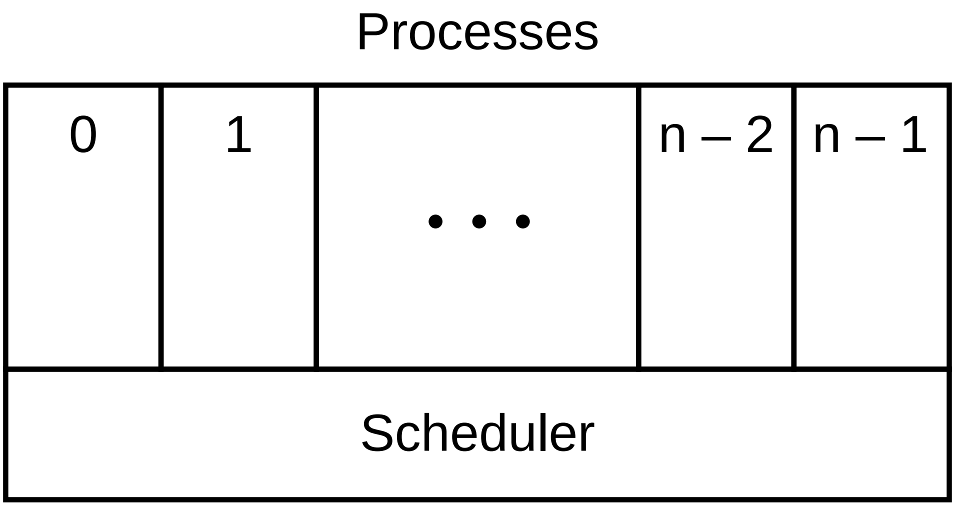


Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

Process management	Memory management	File management
Registers	Pointer to text segment	UMASK mask
Program counter	Pointer to data segment	Root directory
Program status word	Pointer to bss segment	Working directory
Stack pointer	Exit status	File descriptors
Process state	Signal status	Effective uid
Time when process started	Process id	Effective gid
CPU time used	Parent process	System call parameters
Children’s CPU time	Process group	Various flag bits
Time of next alarm	Real uid	
Message queue pointers	Effective	
Pending signal bits	Real gid	
Process id	Effective gid	
Various flag bits	Bit maps for signals	
	Various flag bits	

Figure 2-4. Some of the fields of the MINIX process table.

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler marks waiting task as ready.
7. Scheduler decides which process is to run next.
8. C procedure returns to the assembly code.
9. Assembly language procedure starts up new current process.

Figure 2-5. Skeleton of what the lowest level of the operating system does when an interrupt occurs.

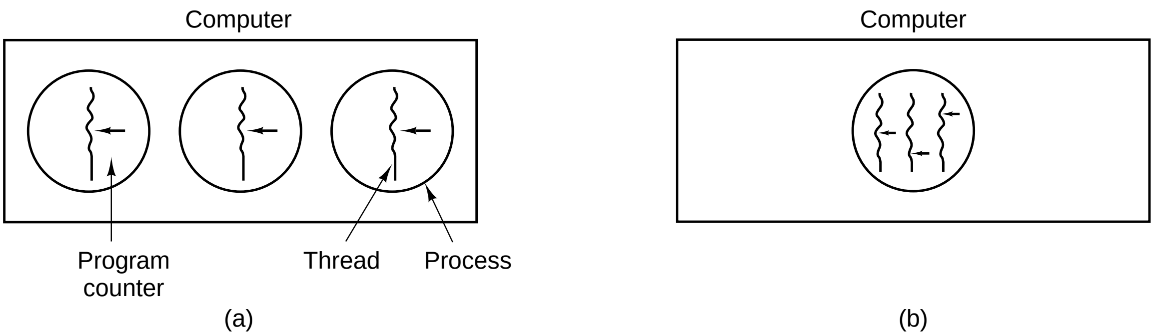


Figure 2-6. (a) Three processes each with one thread. (b) One process with three threads.

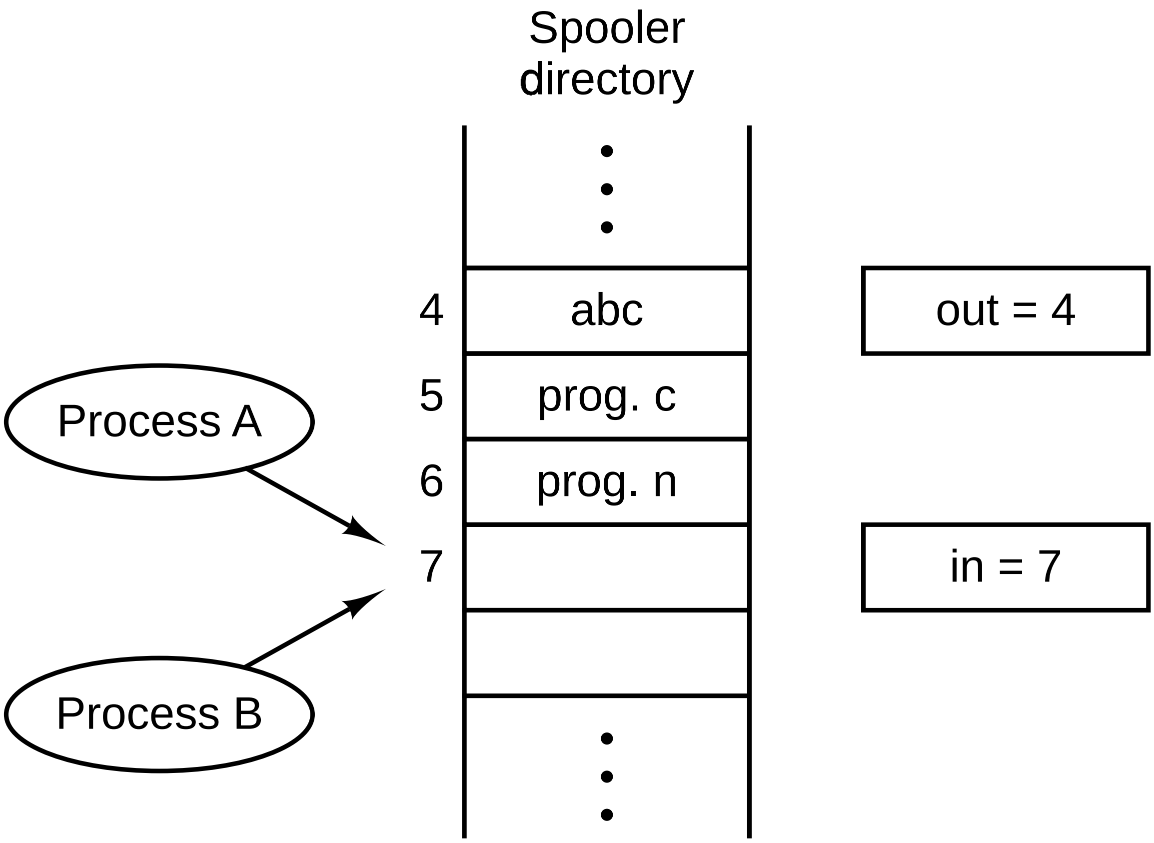


Figure 2-7. Two processes want to access shared memory at the same time.

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

(a)

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

(b)

Figure 2-8. A proposed solution to the critical region problem.

```
#define FALSE                    0
#define TRUE                     1
#define N    2                   /* number of processes */

int turn;                        /* whose turn is it? */
int interested[N];               /* all values initially 0 (FALSE) */

void enter_region(int process);  /* process is 0 or 1 */
{
  int other;                     /* number of the other process */

  other = 1 – process;           /* the opposite of process */
  interested[process] = TRUE;    /* show that you are interested */
  turn = process;                /* set flag */
  while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)   /* process: who is leaving */
{
  interested[process] = FALSE;   /* indicate departure from critical region */
}
```

Figure 2-9. Peterson's solution for achieving mutual exclusion.

```
enter_region:
        tsl register,lock          | copy lock to register and set lock to 1
        cmp register,#0            | was lock zero?
        jne enter_region           | if it was non zero, lock was set, so loop
        ret                        | return to caller; critical region entered

leave_region:
        move lock,#0               | store a 0 in lock
        ret                        | return to caller
```

Figure 2-10. Setting and clearing locks using TSL.

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

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


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

Figure 2-11. The producer-consumer problem with a fatal race condition.

```
#define N 100                    /* number of slots in the buffer */
typedef int semaphore;           /* semaphores are a special kind of int */
semaphore mutex = 1;             /* controls access to critical region */
semaphore empty = N;             /* counts empty buffer slots */
semaphore full = 0;              /* counts full buffer slots */

void producer(void)
{
  int item;

  while (TRUE) {                 /* TRUE is the constant 1 */
        produce_item(&item);     /* generate something to put in buffer */
        down(&empty);            /* decrement empty count */
        down(&mutex);            /* enter critical region */
        enter_item(item);        /* put new item in buffer */
        up(&mutex);              /* leave critical region */
        up(&full);               /* increment count of full slots */
  }
}

void consumer(void)
{
  int item;

  while (TRUE) {                 /* infinite loop */
        down(&full);             /* decrement full count */
        down(&mutex);            /* enter critical region */
        remove_item(&item);      /* take item from buffer */
        up(&mutex);              /* leave critical region */
        up(&empty);              /* increment count of empty slots */
        consume_item(item);      /* do something with the item */
  }
}
```

Figure 2-12. The producer-consumer problem using semaphores.

```
monitor example
   integer i;
   condition c;

   procedure producer(x);
   ⋮
   end;

   procedure consumer(x);
   ⋮
   end;
end monitor;
```

Figure 2-13. A monitor.

```
monitor ProducerConsumer
  condition full, empty;
  integer count;
  procedure enter;
  begin
    if count = N then wait(full);
    enter_item;
    count := count + 1;
    if count = 1 then signal(empty)
  end;
  procedure remove;
  begin
    if count = 0 then wait(empty);
    remove_item;
    count := count – 1;
    if count = N – 1 then signal(full)
  end;
  count := 0;
end monitor;
procedure producer;
begin
  while true do
  begin
    produce_item;
    ProducerConsumer.enter
  end
end;
procedure consumer;
begin
  while true do
  begin
    ProducerConsumer.remove;
    consume_item
  end
end;
```

Figure 2-14. The producer-consumer problem with monitors.

```
#define N 100                          /* number of slots in the buffer */

void producer(void)
{
  int item;
  message m;                           /* message buffer */

  while (TRUE) {
        produce_item(&item);           /* generate something to put in buffer */
        receive(consumer, &m);         /* wait for an empty to arrive */
        build_message(&m, item);       /* construct a message to send */
        send(consumer, &m);            /* send item to consumer */
  }
}

void consumer(void)
{
  int item, i;
  message m;

  for (i = 0; i < N; i++) send(producer, &m);/* send N empties */
  while (TRUE) {
        receive(producer, &m);         /* get message containing item */
        extract_item(&m, &item);       /* extract item from message */
        send(producer, &m);            /* send back empty reply */
        consume_item(item);            /* do something with the item */
  }
}
```

Figure 2-15. The producer-consumer problem with *N* messages.

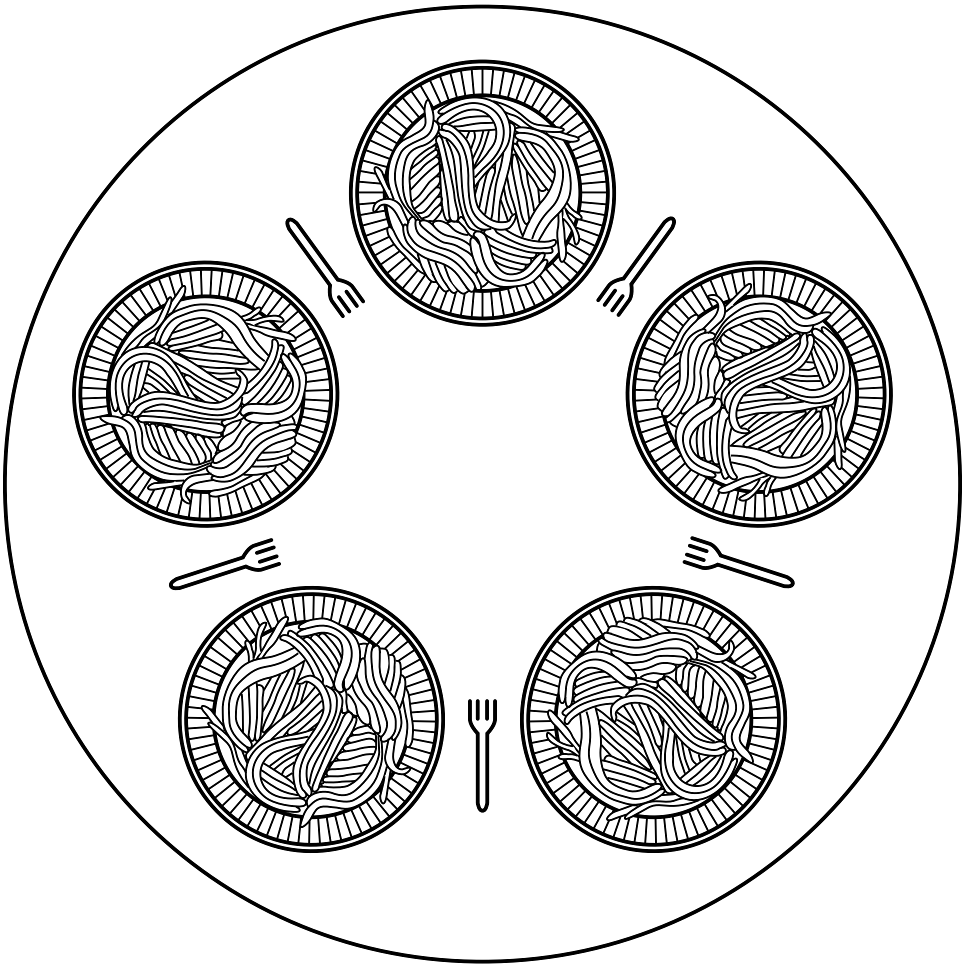


Figure 2-16. Lunch time in the Philosophy Department.

```
#define N 5                       /* number of philosophers */

void philosopher(int i)           /* i: philosopher number, from 0 to 4 */
{
  while (TRUE) {
        think();                  /* philosopher is thinking */
        take_fork(i);             /* take left fork */
        take_fork((i+1) % N);     /* take right fork; % is modulo operator */
        eat();                    /* yum-yum, spaghetti */
        put_fork(i);              /* put left fork back on the table */
        put_fork((i+1) % N);      /* put right fork back on the table */
  }
}
```

Figure 2-17. A nonsolution to the dining philosophers problem.

```
#define N             5               /* number of philosophers */
#define LEFT     (i-1)%N              /* number of i's left neighbor */
#define RIGHT    (i+1)%N              /* number of i's right neighbor */
#define THINKING      0               /* philosopher is thinking */
#define HUNGRY        1               /* philosopher is trying to get forks */
#define EATING        2               /* philosopher is eating */

typedef int semaphore;                /* semaphores are a special kind of int */
int state[N];                         /* array to keep track of everyone's state */
semaphore mutex = 1;                  /* mutual exclusion for critical regions */
semaphore s[N];                       /* one semaphore per philosopher */

void philosopher(int i)               /* i: philosopher number, from 0 to N-1 */
{   while (TRUE) {                    /* repeat forever */
        think();                      /* philosopher is thinking */
        take_forks(i);                /* acquire two forks or block */
        eat();                        /* yum-yum, spaghetti */
        put_forks(i);                 /* put both forks back on table */
  }
}

void take_forks(int i)                /* i: philosopher number, from 0 to N-1 */
{  down(&mutex);                      /* enter critical region */
  state[i] = HUNGRY;                  /* record fact that philosopher i is hungry */
  test(i);                            /* try to acquire 2 forks */
  up(&mutex);                         /* exit critical region */
  down(&s[i]);                        /* block if forks were not acquired */
}

void put_forks(i)                     /* i: philosopher number, from 0 to N-1 */
{  down(&mutex);                      /* enter critical region */
  state[i] = THINKING;                /* philosopher has finished eating */
  test(LEFT);                         /* see if left neighbor can now eat */
  test(RIGHT);                        /* see if right neighbor can now eat */
  up(&mutex);                         /* exit critical region */
}

void test(i)                          /* i: philosopher number, from 0 to N-1 */
{  if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
  }
}
```

Figure 2-18. A solution to the dining philosopher's problem.

```
typedef int semaphore;          /* use your imagination */
semaphore mutex = 1;            /* controls access to 'rc' */
semaphore db = 1;               /* controls access to the data base */
int rc = 0;                     /* # of processes reading or wanting to */

void reader(void)
{
  while (TRUE) {                /* repeat forever */
        down(&mutex);           /* get exclusive access to 'rc' */
        rc = rc + 1;            /* one reader more now */
        if (rc == 1) down(&db); /* if this is the first reader ... */
        up(&mutex);             /* release exclusive access to 'rc' */
        read_data_base();       /* access the data */
        down(&mutex);           /* get exclusive access to 'rc' */
        rc = rc - 1;            /* one reader fewer now */
        if (rc == 0) up(&db);   /* if this is the last reader ... */
        up(&mutex);             /* release exclusive access to 'rc' */
        use_data_read();        /* noncritical region */
  }
}


void writer(void)
{
  while (TRUE) {                /* repeat forever */
        think_up_data();        /* noncritical region */
        down(&db);              /* get exclusive access */
        write_data_base();      /* update the data */
        up(&db);                /* release exclusive access */
  }
}
```

Figure 2-19. A solution to the readers and writers problem.

Figure 2-20. The sleeping barber.

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
  while (TRUE) {
        down(customers);            /* go to sleep if # of customers is 0 */
        down(mutex);                /* acquire access to 'waiting' */
        waiting = waiting - 1;      /* decrement count of waiting customers */
        up(barbers);                /* one barber is now ready to cut hair */
        up(mutex);                  /* release 'waiting' */
        cut_hair();                 /* cut hair (outside critical region) */
  }
}

void customer(void)
{
  down(mutex);                      /* enter critical region */
  if (waiting < CHAIRS) {           /* if there are no free chairs, leave */
        waiting = waiting + 1;      /* increment count of waiting customers */
        up(customers);              /* wake up barber if necessary */
        up(mutex);                  /* release access to 'waiting' */
        down(barbers);              /* go to sleep if # of free barbers is 0 */
        get_haircut();              /* be seated and be serviced */
  } else {
        up(mutex);                  /* shop is full; do not wait */
  }
}
```

Figure 2-21. A solution to the sleeping barber problem.

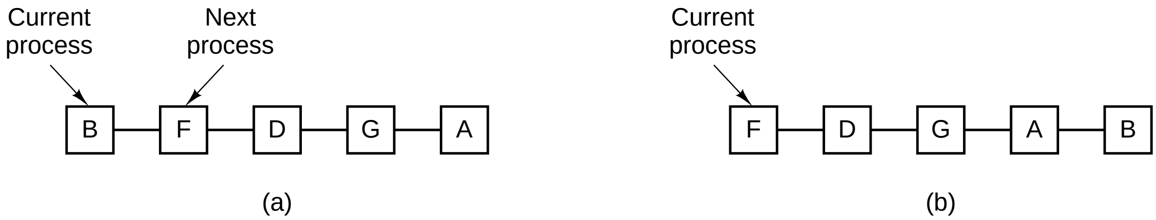


Figure 2-22. Round robin scheduling. (a) The list of runnable processes. (b) The list of runnable processes after *B* uses up its quantum.

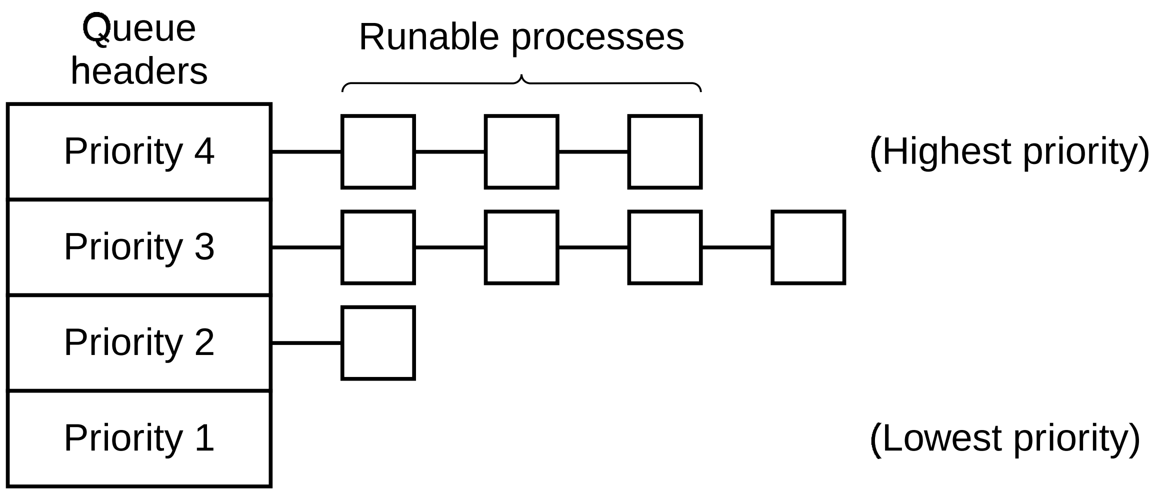


Figure 2-23. A scheduling algorithm with four priority classes.

Figure 2-24. An example of shortest job first scheduling.

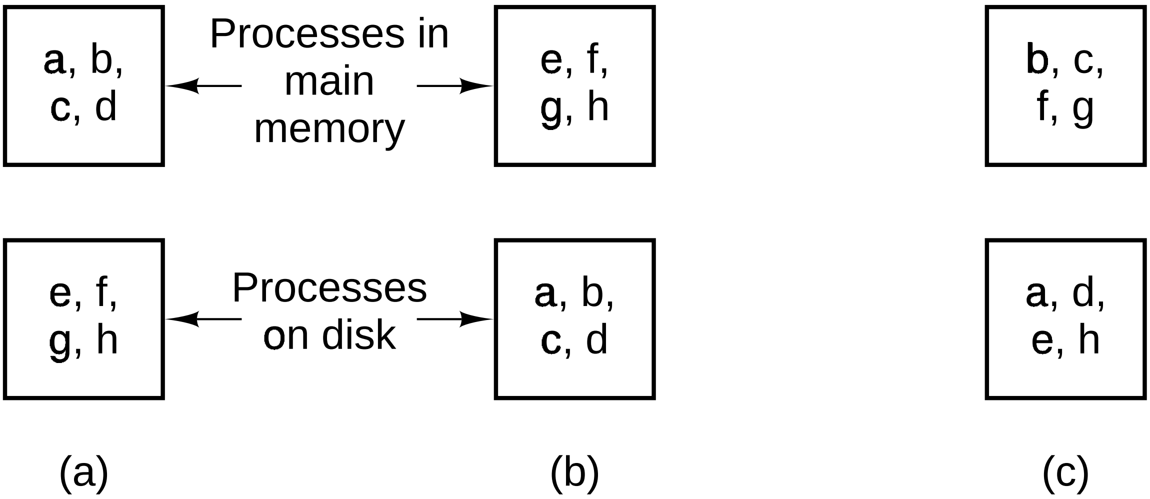


Figure 2-25. A two-level scheduler must move processes between disk and memory and also choose processes to run from among those in memory. Three different instants of time are represented by (a), (b), and (c) .

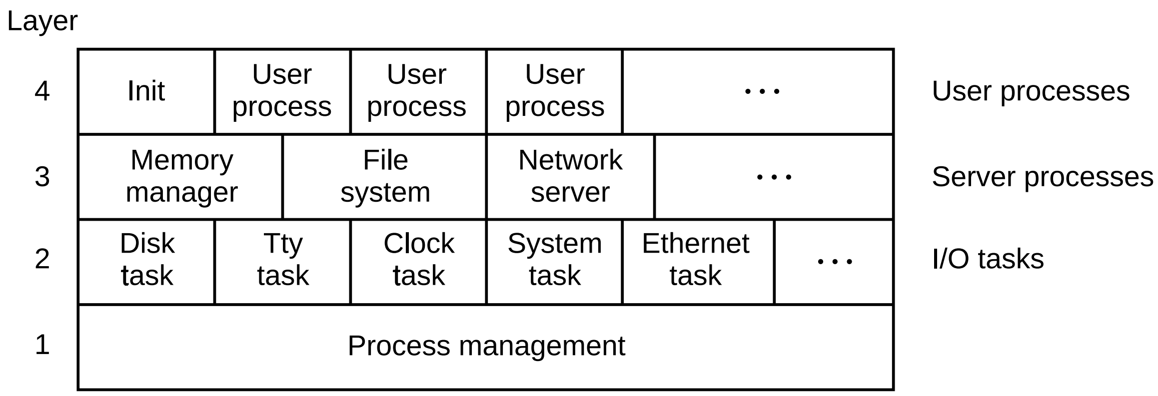


Figure 2-26. MINIX is structured in four layers.

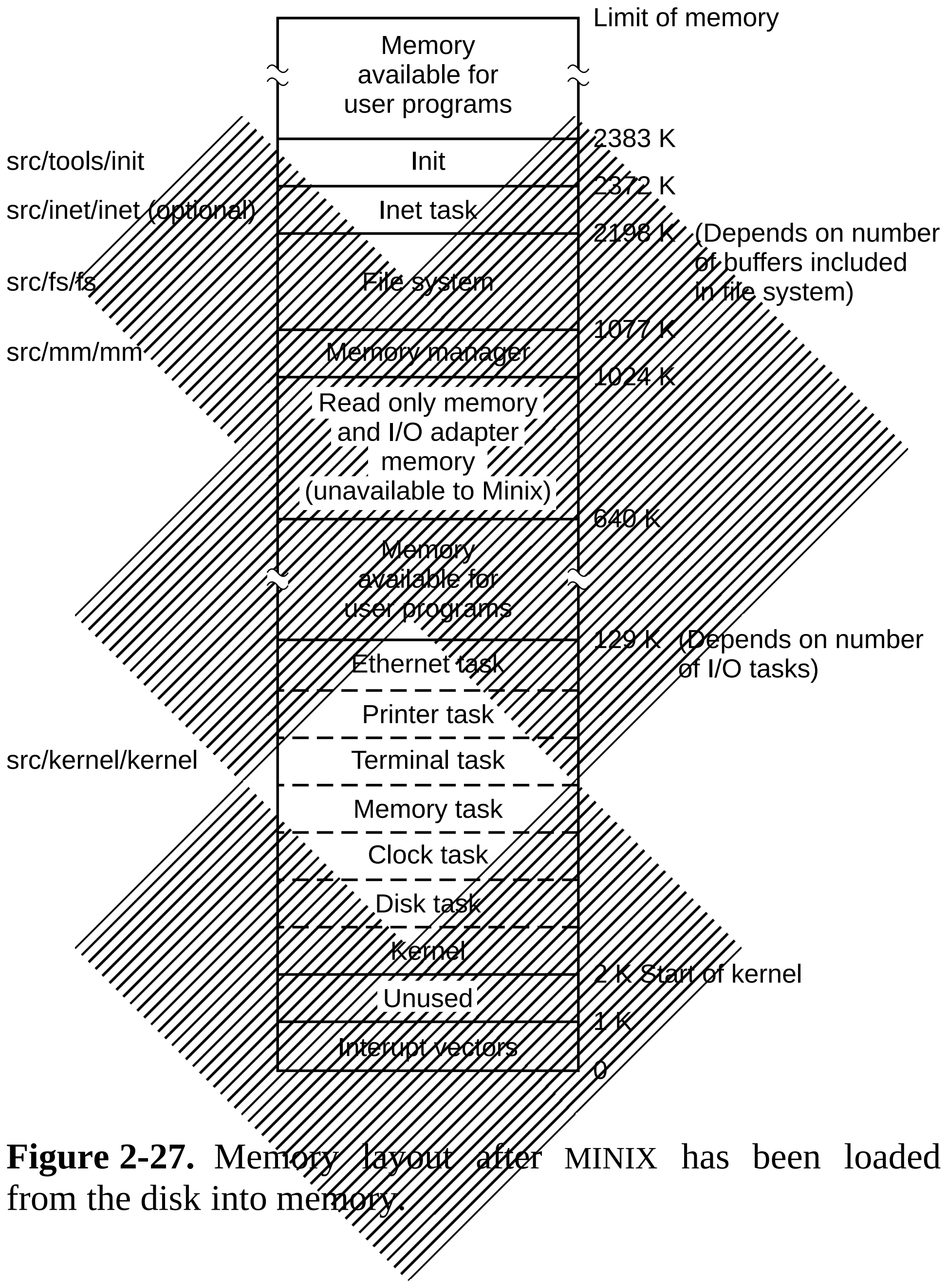


Figure 2-27. Memory layout after MINIX has been loaded from the disk into memory.

```
#include <minix/config.h>   /* MUST be first */
#include <ansi.h>           /* MUST be second */
#include <sys/types.h>
#include <minix/const.h>
#include <minix/type.h>
#include <limits.h>
#include <errno.h>
#include <minix/syslib.h>
```

Figure 2-28. Part of a master header which ensures inclusion of header files needed by all C source files.

Type	16-Bit MINIX	32-Bit MINIX
gid_t	8	8
dev_t	16	16
pid_t	16	32
ino_t	16	32

Figure 2-29. The size, in bits, of some types on 16-bit and 32-bit systems.

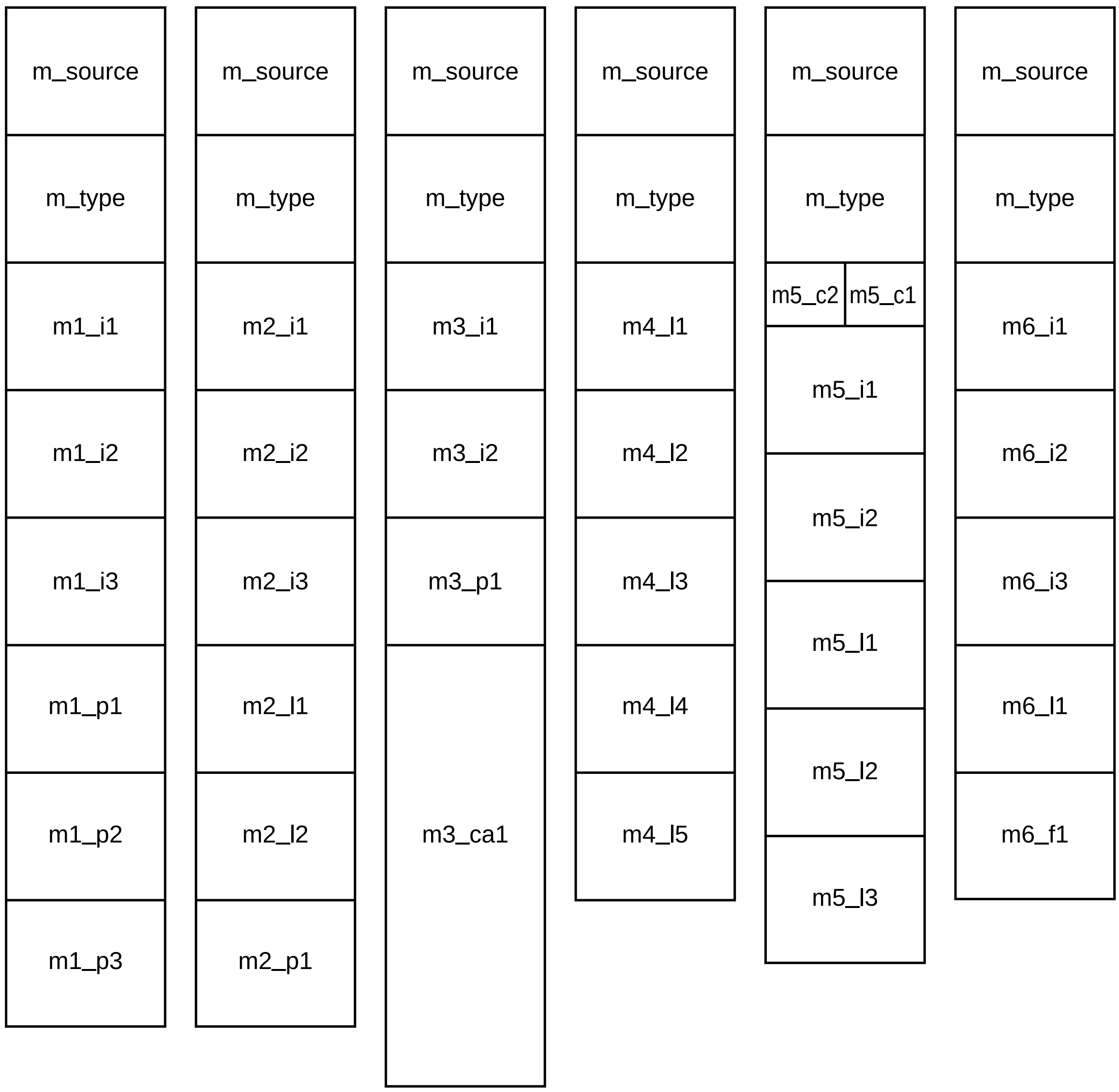


Figure 2-30. The six messages types used in MINIX. The sizes of message elements will vary, depending upon the architecture of the machine; this diagram illustrates sizes on a machine with 32-bit pointers, such as the Pentium (Pro).

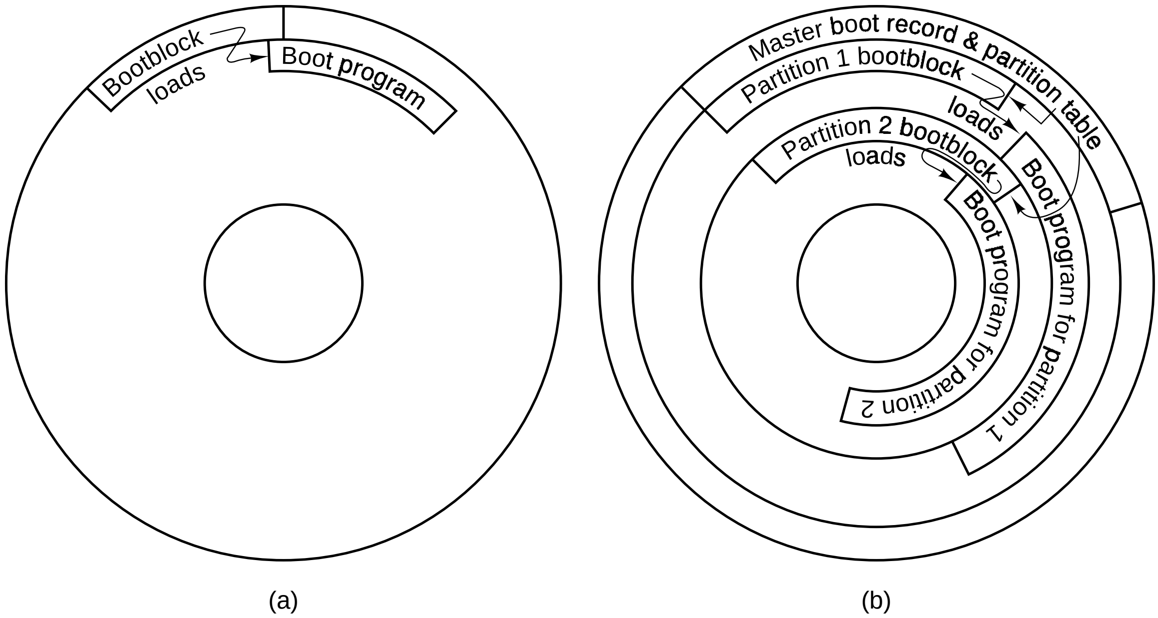


Figure 2-31. Disk structures used for bootstrapping. (a) Unpartitioned disk. The first sector is the bootblock. (b) Partitioned disk. The first sector is the master boot record.

```
#include <minix/config.h>
#if _WORD_SIZE == 2
#include "mpx88.s"
#else
#include "mpx386.s"
#endif
```

Figure 2-32. How alternative assembly language source files are selected.

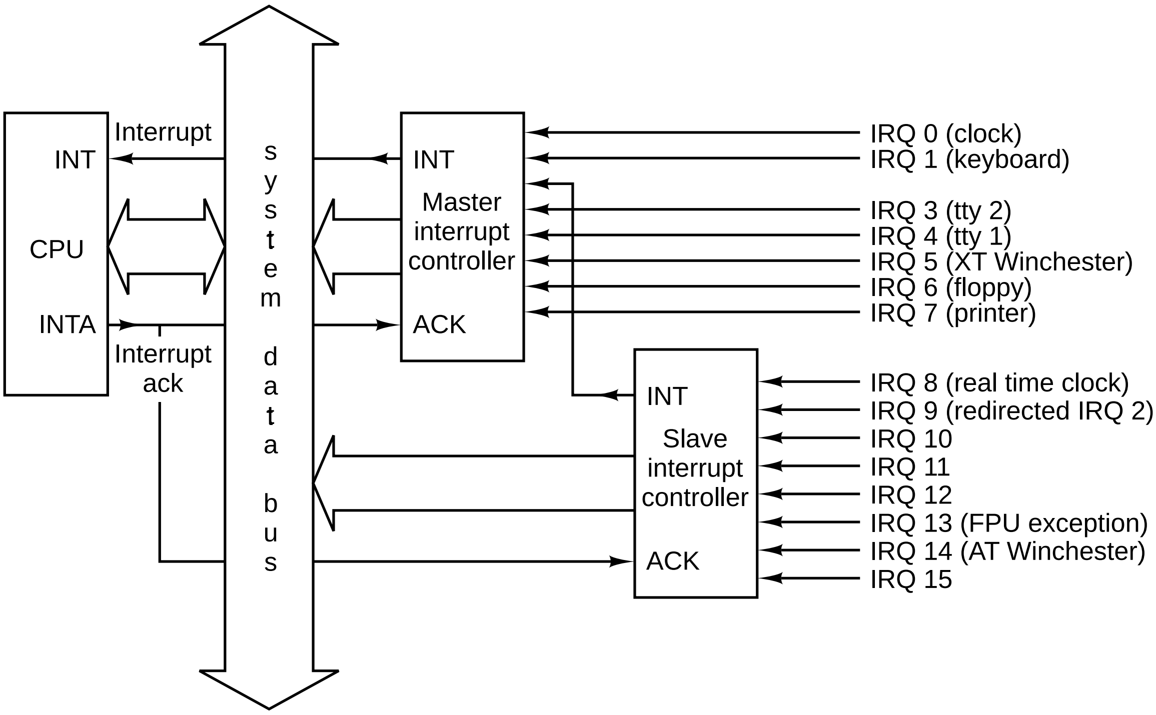


Figure 2-33. Interrupt processing hardware on a 32-bit Intel PC.

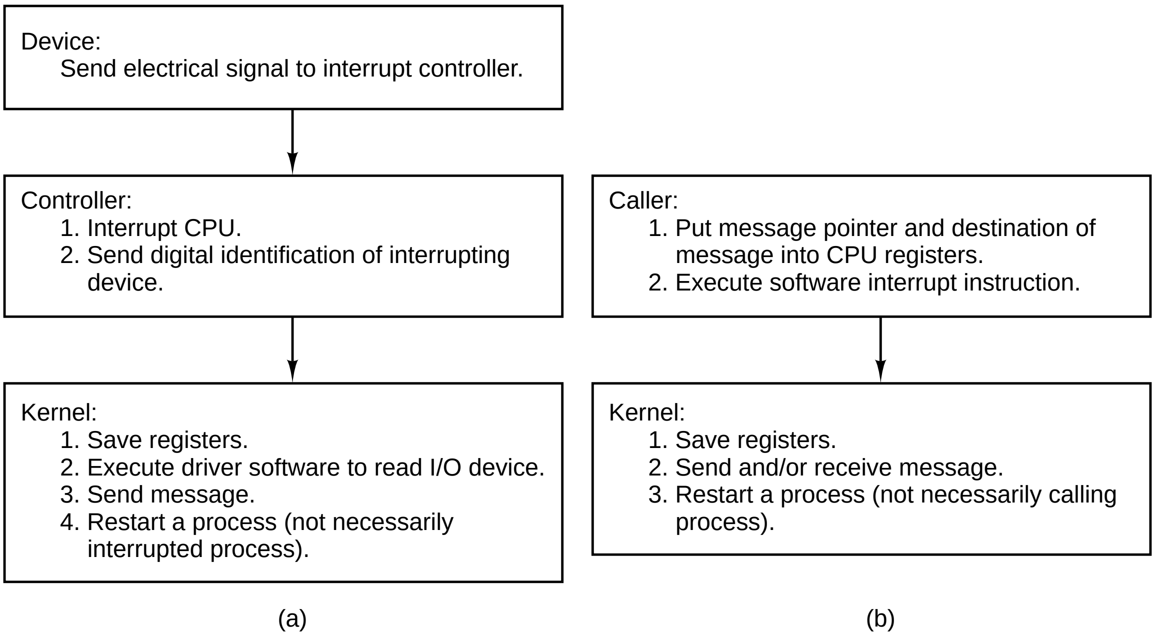


Figure 2-34. (a) How a hardware interrupt is processed. (b) How a system call is made.

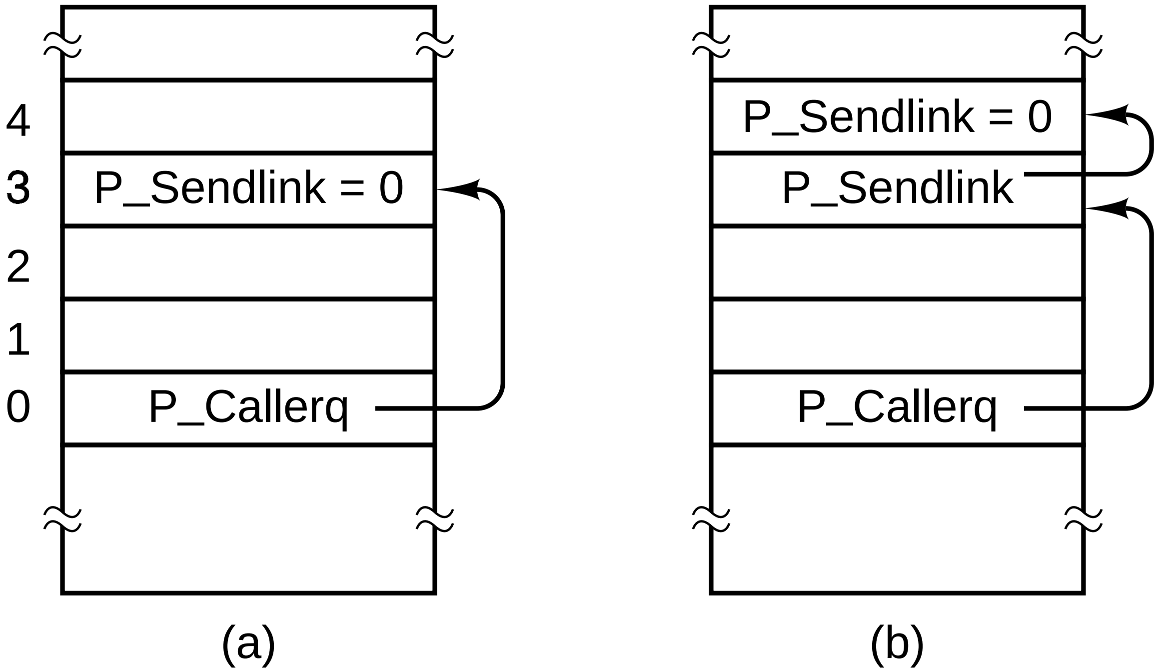


Figure 2-35. Queueing of processes trying to send to process 0.

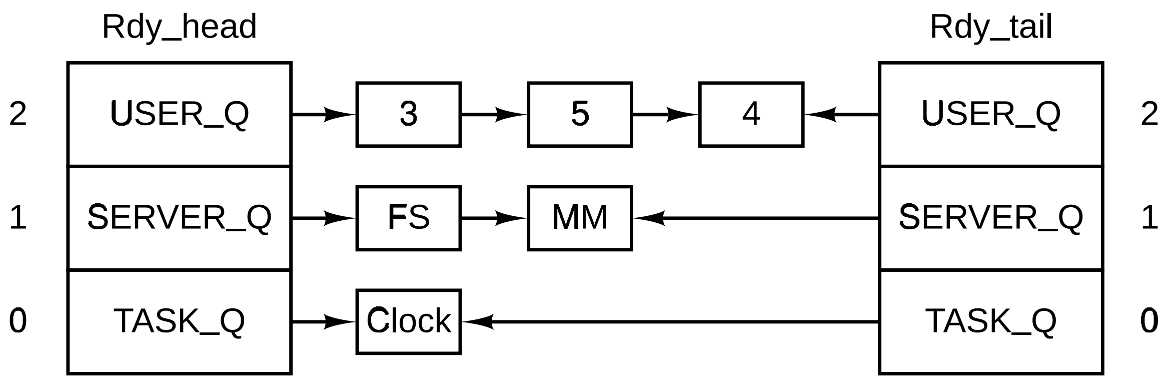


Figure 2-36. The scheduler maintains three queues, one per priority level.

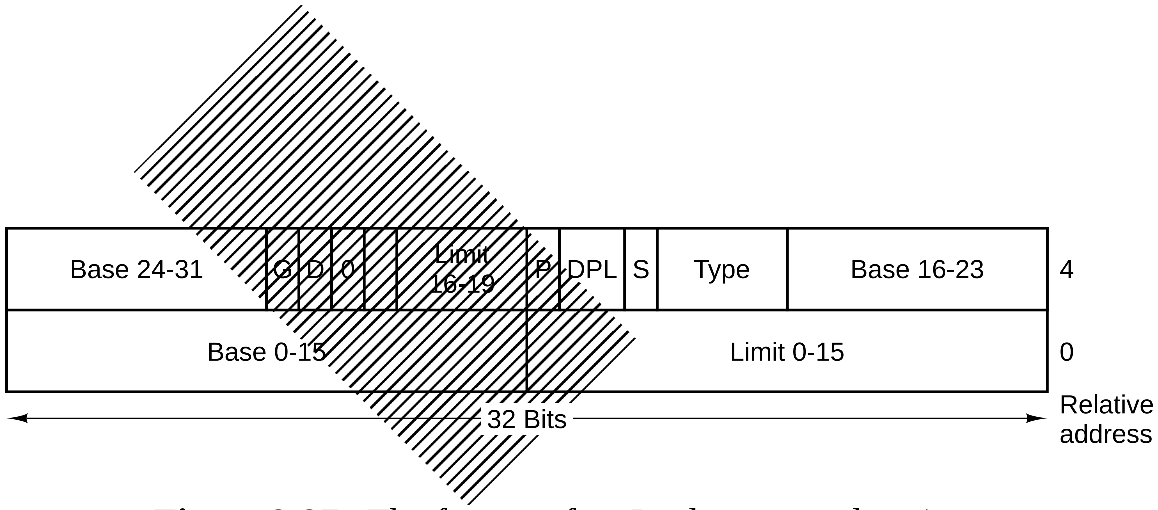


Figure 2-37. The format of an Intel segment descriptor.

3

INPUT/OUTPUT

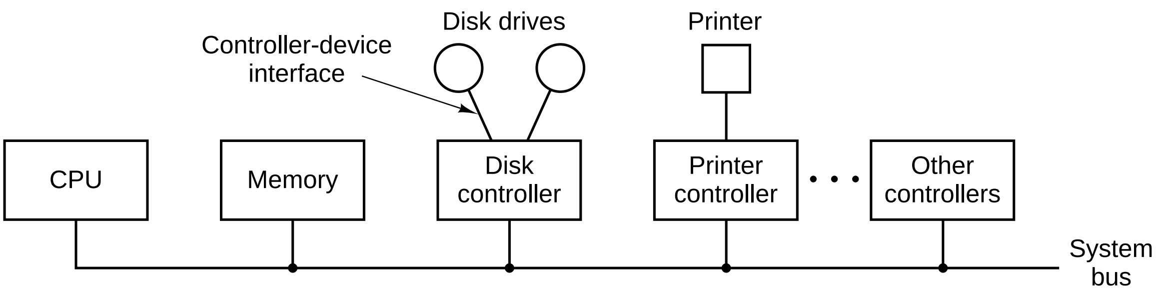


Figure 3-1. A model for connecting the CPU, memory, controllers, and I/O devices.

I/O controller	I/O address	Hardware IRQ	Interrupt vector
Clock	040 – 043	0	8
Keyboard	060 – 063	1	9
Hard disk	1F0 – 1F7	14	118
Secondary RS232	2F8 – 2FF	3	11
Printer	378 – 37F	7	15
Floppy disk	3F0 – 3F7	6	14
Primary RS232	3F8 – 3FF	4	12

Figure 3-2. Some examples of controllers, their I/O addresses, their hardware interrupt lines, and their interrupt vectors on a typical PC running MS-DOS.

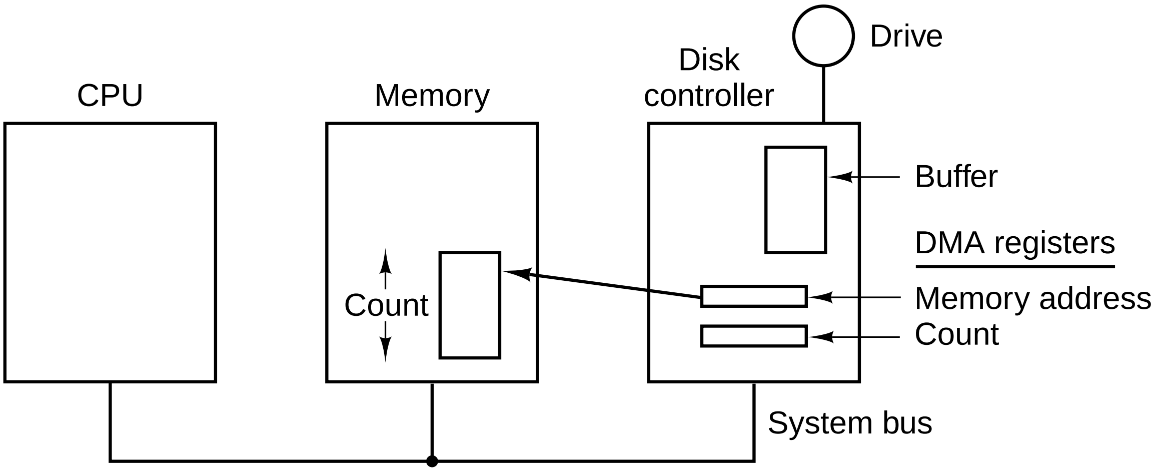


Figure 3-3. A DMA transfer is done entirely by the controller.

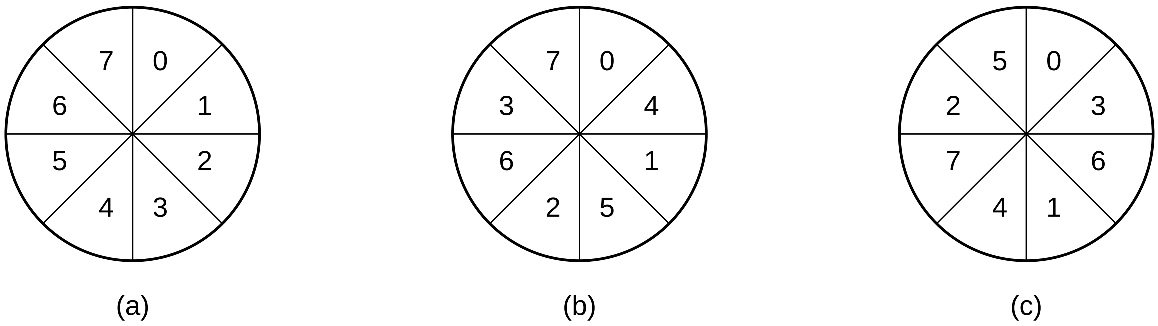


Figure 3-4. (a) No interleaving. (b) Single interleaving. (c) Double interleaving.

Uniform interfacing for device drivers
Device naming
Device protection
Providing a device-independent block size
Buffering
Storage allocation on block devices
Allocating and releasing dedicated devices
Error reporting

Figure 3-5. Functions of the device-independent I/O software.

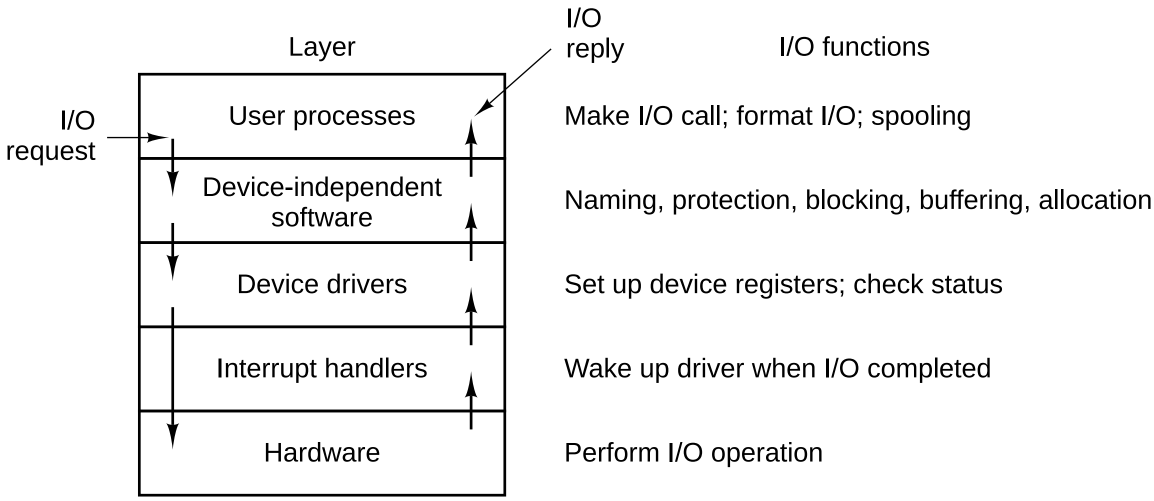


Figure 3-6. Layers of the I/O system and the main functions of each layer.

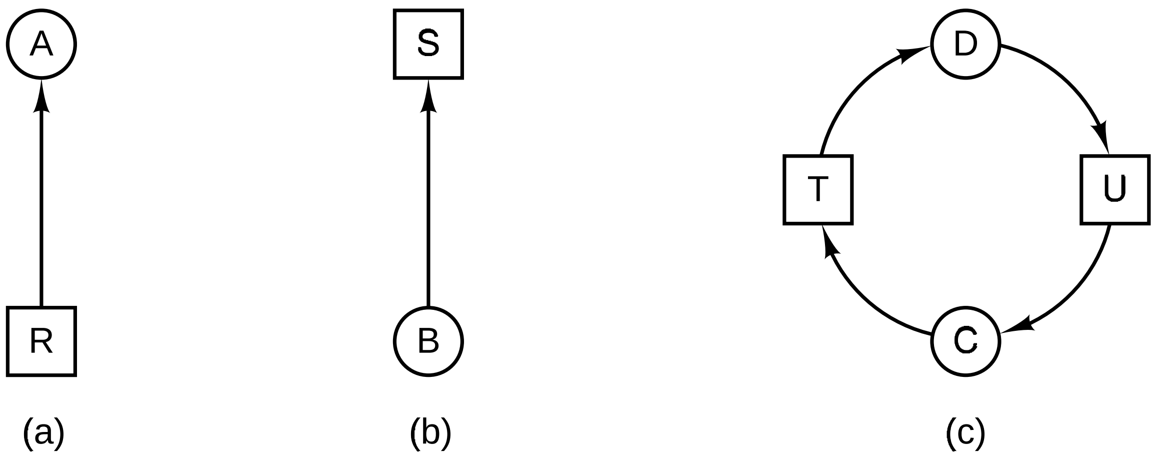


Figure 3-7. Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

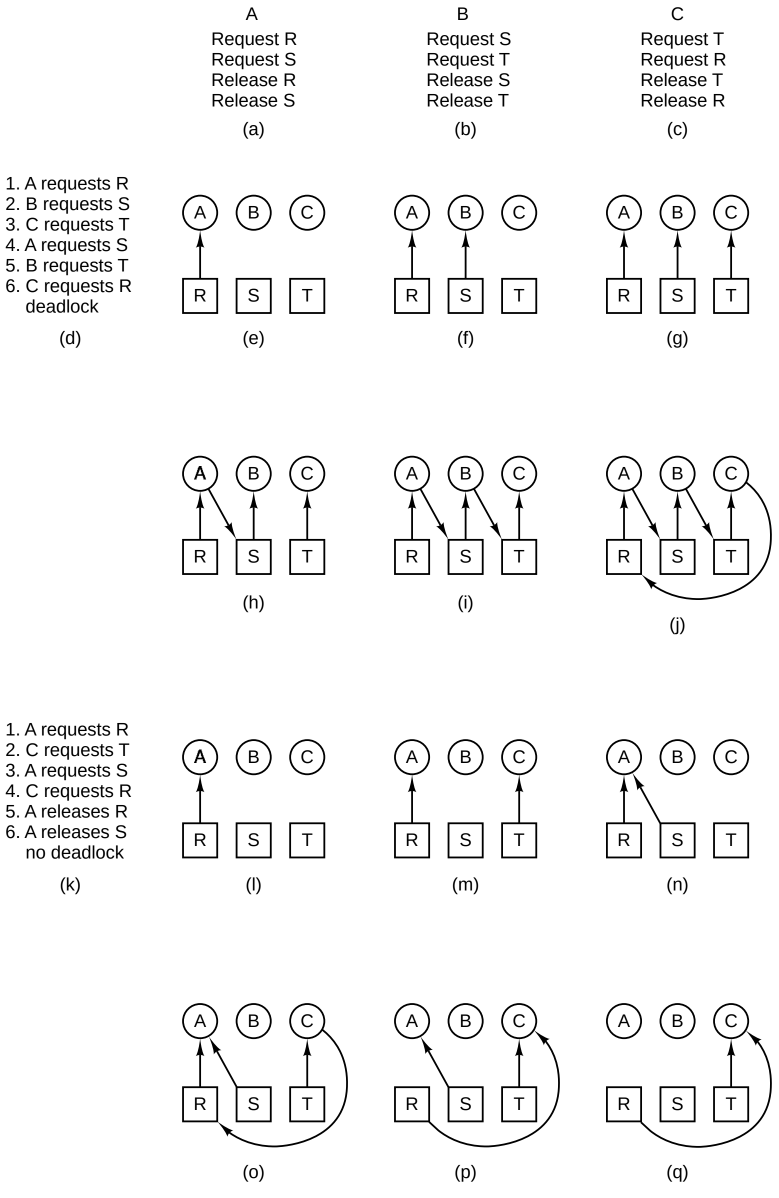


Figure 3-8. An example of how deadlock occurs and how it can be avoided.

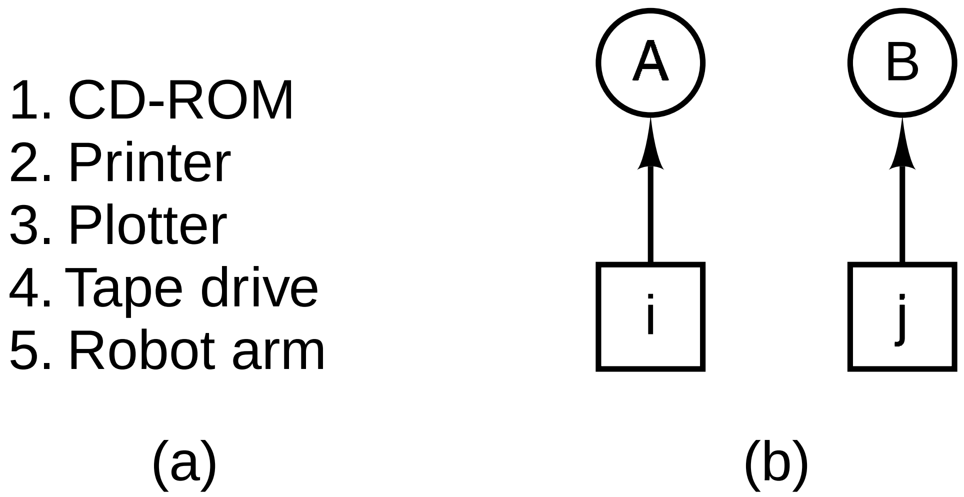


Figure 3-9. (a) Numerically ordered resources. (b) A resource graph.

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

Figure 3-10. Summary of approaches to deadlock prevention.

Name	Used	Maximum
Andy	0	6
Barbara	0	5
Marvin	0	4
Suzanne	0	7

Available: 10

(a)

Name	Used	Maximum
Andy	1	6
Barbara	1	5
Marvin	2	4
Suzanne	4	7

Available: 2

(b)

Name	Used	Maximum
Andy	1	6
Barbara	2	5
Marvin	2	4
Suzanne	4	7

Available: 1

(c)

Figure 3-11. Three resource allocation states: (a) Safe. (b) Safe. (c) Unsafe.

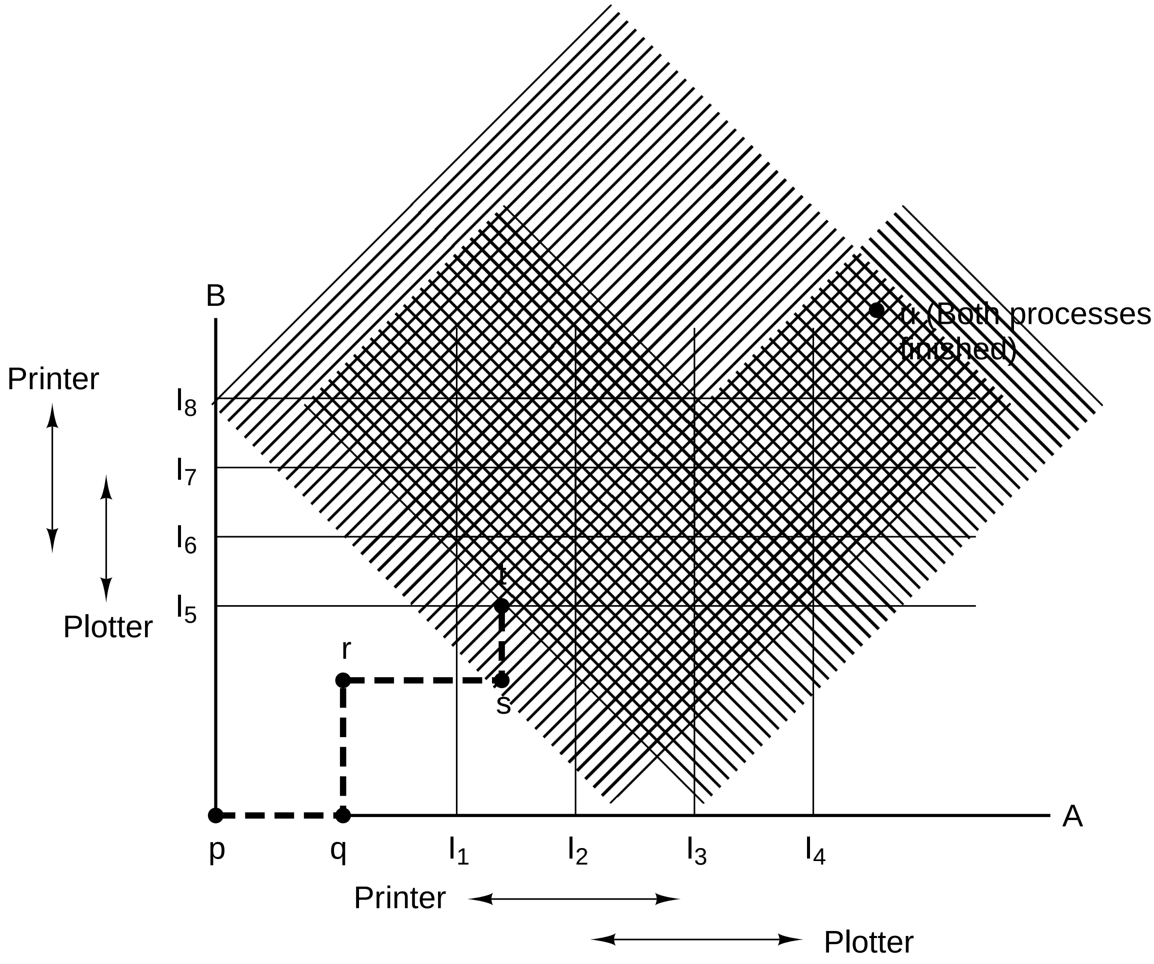


Figure 3-12. Two process resource trajectories.

Process	Tape drives	Plotters	Printers	CD-ROMS
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Printers	CD-ROMS
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

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

Figure 3-13. The banker's algorithm with multiple resources.

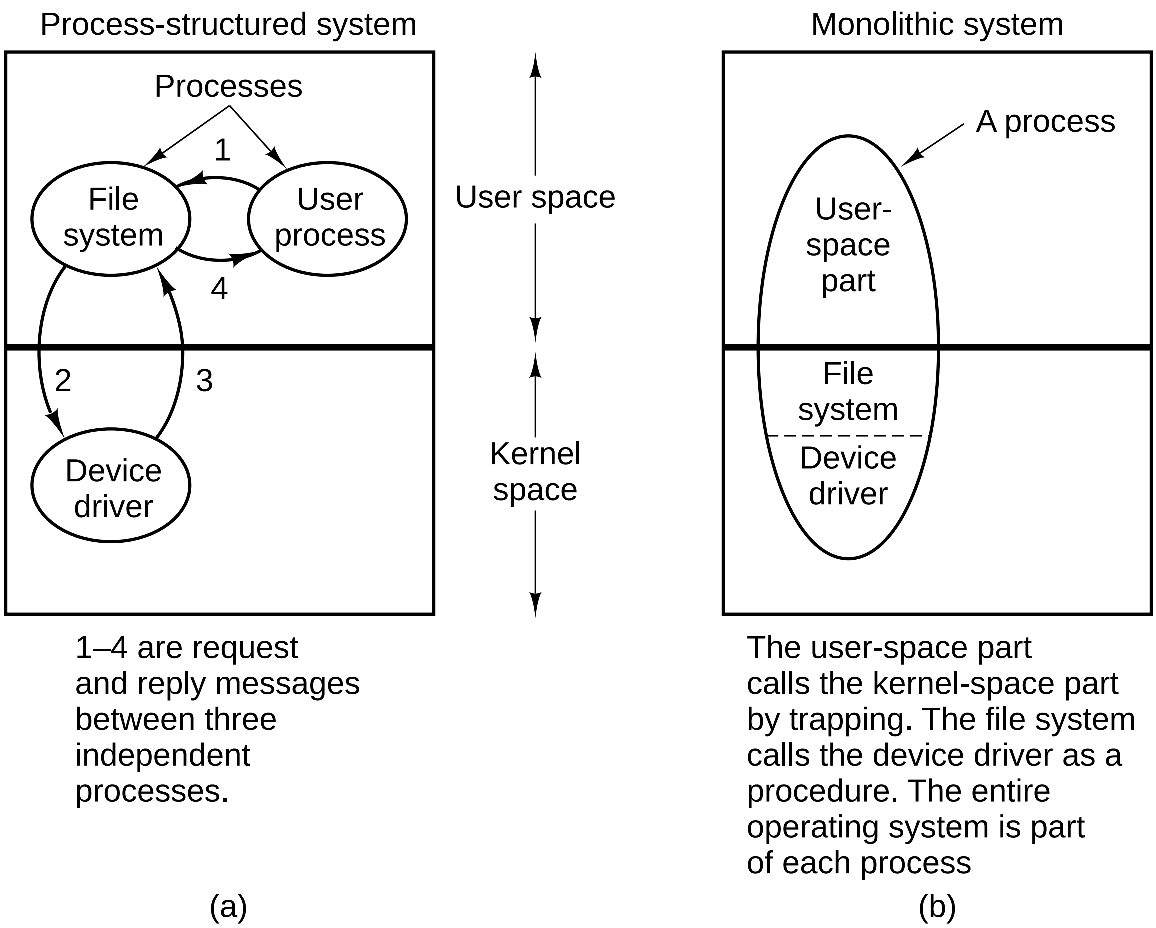


Figure 3-14. Two ways of structuring user-system communication.

Requests		
Field	**Type**	**Meaning**
m.m_type	int	Operation requested
m.DEVICE	int	Minor device to use
m.PROC_NR	int	Process requesting the I/O
m.COUNT	int	Byte count or ioctl code
m.POSITION	long	Position on device
m.ADDRESS	char*	Address within requesting process

Replies		
Field	**Type**	**Meaning**
m.m_type	int	Always TASK_REPLY
m.REP_PROC_NR	int	Same as PROC_NR in request
m.REP_STATUS	int	Bytes transferred or error number

Figure 3-15. Fields of the messages sent by the file system to the block device drivers and fields of the replies sent back.

```
message mess;                   /* message buffer */

void io_task() {
  initialize();                 /* only done once, during system init. */
  while (TRUE) {
        receive(ANY, &mess);/* wait for a request for work */
        caller = mess.source;/* process from whom message came */
        switch(mess.type) {
            case READ:         rcode = dev_read(&mess); break;
            case WRITE:        rcode = dev_write(&mess); break;
            /* Other cases go here, including OPEN, CLOSE, and IOCTL */
             default:  rcode = ERROR;
        }
        mess.type = TASK_REPLY;
        mess.status = rcode; /* result code */
        send(caller, &mess); /* send reply message back to caller */
  }
}
```

Figure 3-16. Outline of the main procedure of an I/O task.

```
message mess;                    /* message buffer */

void shared_io_task(struct driver_table *entry_points) {
/* initialization is done by each task before calling this */
  while (TRUE) {
        receive(ANY, &mess);
        caller = mess.source;
        switch(mess.type) {
            case READ:         rcode = (*entry_points−>dev_read)(&mess); break;
            case WRITE:        rcode = (*entry_points−>dev_write)(&mess); break;
            /* Other cases go here, including OPEN, CLOSE, and IOCTL */
            default:    rcode = ERROR;
        }
        mess.type = TASK_REPLY;
        mess.status = rcode;    /* result code */
        send(caller, &mess);
  }
}
```

Figure 3-17. A shared I/O task main procedure using indirect calls.

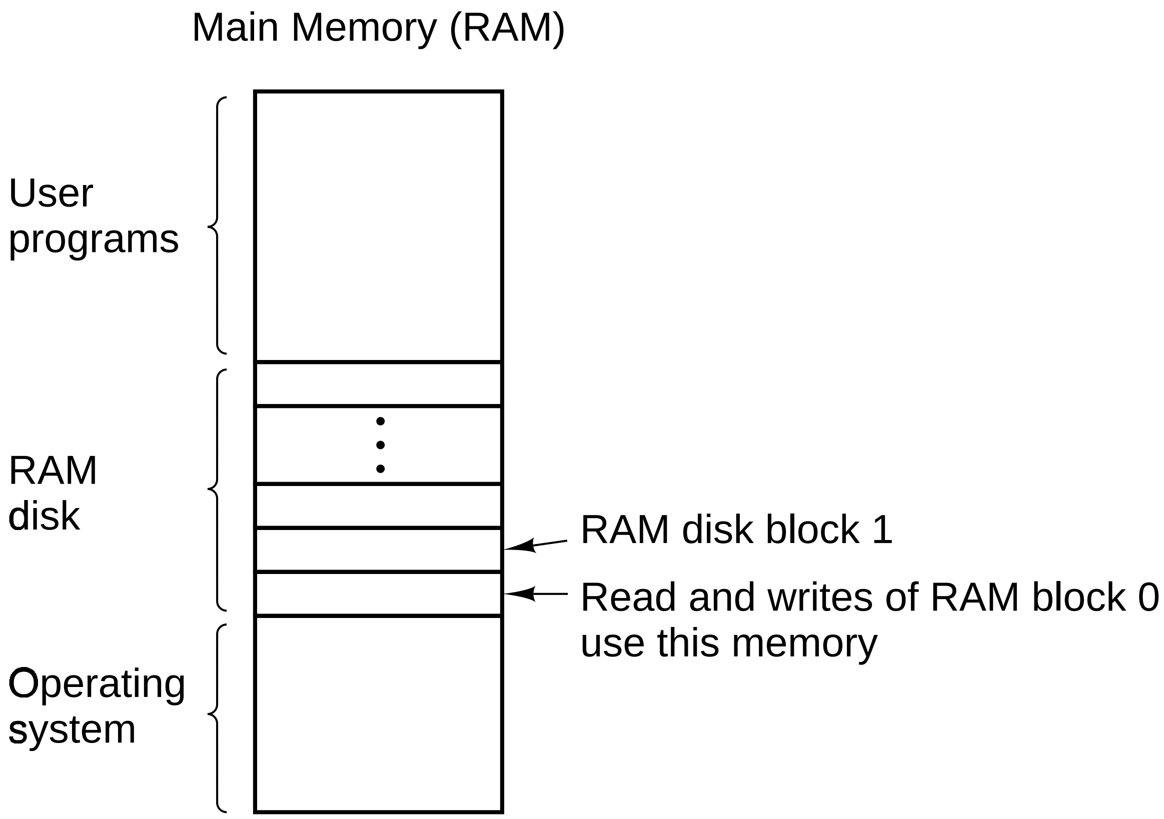


Figure 3-18. A RAM disk.

Parameter	IBM 360-KB floppy disk	WD 540-MB hard disk
Number of cylinders	40	1048
Tracks per cylinder	2	4
Sectors per track	9	252
Sectors per disk	720	1056384
Bytes per sector	512	512
Bytes per disk	368640	540868608
Seek time (adjacent cylinders)	6 msec	4 msec
Seek time (average case)	77 msec	11 msec
Rotation time	200 msec	13 msec
Motor stop/start time	250 msec	9 sec
Time to transfer 1 sector	22 msec	53 μsec

Figure 3-19. Disk parameters for the original IBM PC 360-KB floppy disk and a Western Digital WD AC2540 540-MB hard disk.

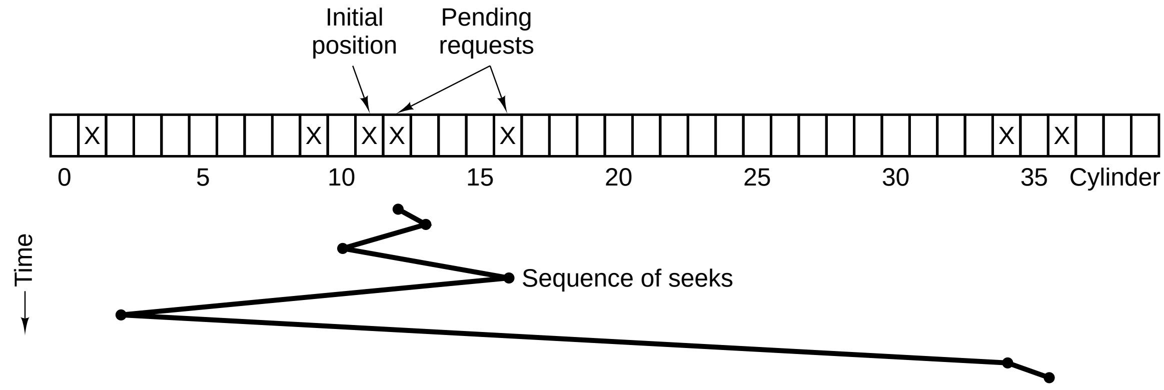


Figure 3-20. Shortest Seek First (SSF) disk scheduling algorithm.

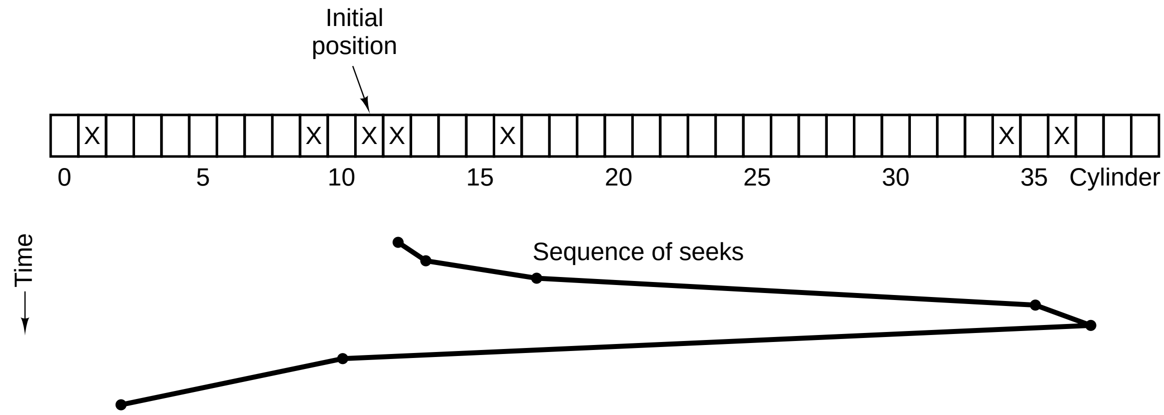


Figure 3-21. The elevator algorithm for scheduling disk requests.

Register	Read Function	Write Function
0	Data	Data
1	Error	Write Precompensation
2	Sector Count	Sector Count
3	Sector Number (0-7)	Sector Number (0-7)
4	Cylinder Low (8-15)	Cylinder Low (8-15)
5	Cylinder High (16-23)	Cylinder High (16-23)
6	Select Drive/Head (24-27)	Select Drive/Head (24-27)
7	Status	Command

(a)

7	6	5	4	3	2	1	0
1	LBA	1	D	HS3	HS2	HS1	HS0

LBA: 0 = Cylinder/Head/Sector Mode
1 = Logical Block Addressing Mode
D: 0 = master drive
1 = slave drive
HSn: CHS mode: Head Select in CHS mode
LBA mode: Block select bits 24 - 27

(b)

Figure 3-22. (a) The control registers of an IDE hard disk controller. The numbers in parentheses are the bits of the logical block address selected by each register in LBA mode. (b) The fields of the Select Drive/Head register.

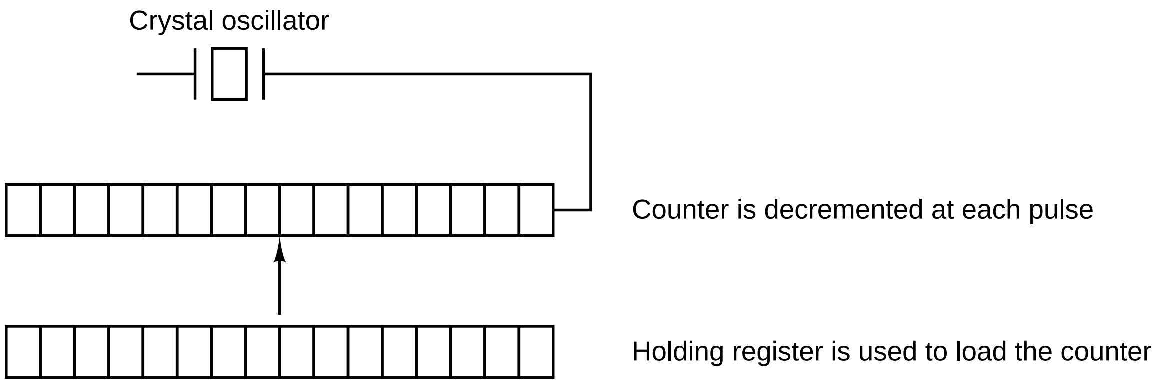


Figure 3-23. A programmable clock.

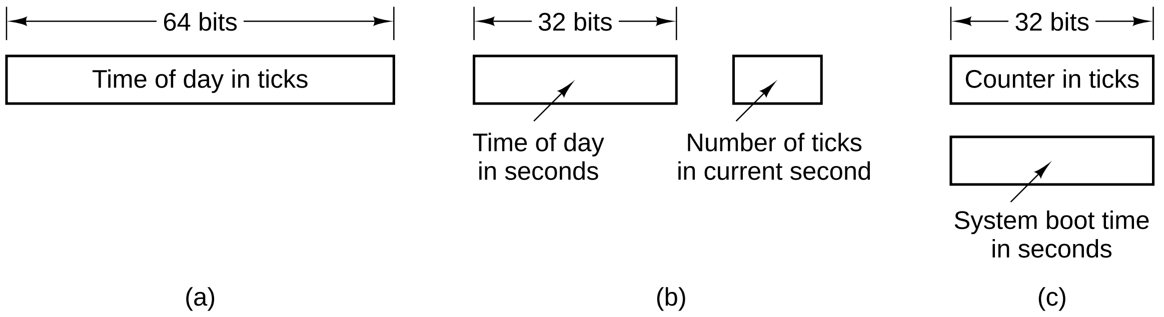


Figure 3-24. Three ways to maintain the time of day.

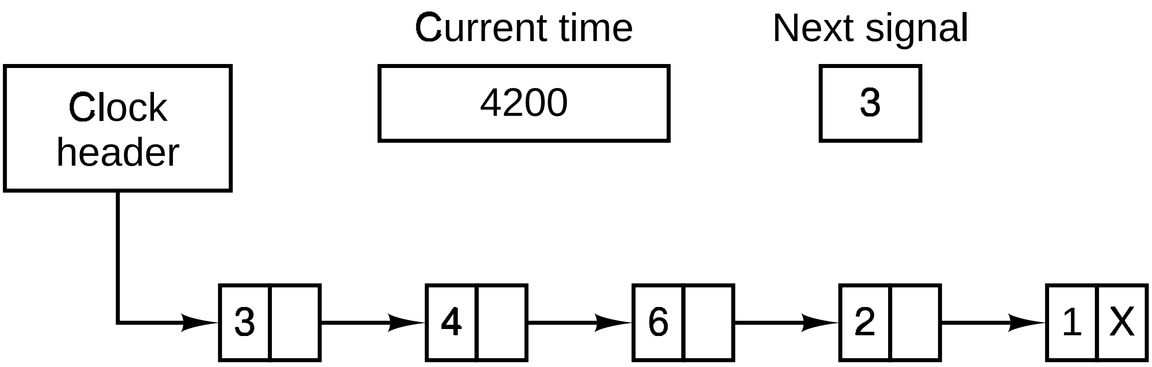


Figure 3-25. Simulating multiple timers with a single clock.

Service	Access	Response	Clients
Gettime	System call	Message	Any process
Uptime	System call	Message	Any process
Uptime	Function call	Function value	Kernel or task
Alarm	System call	Signal	Any process
Alarm	System call	Watchdog activation	Task
Synchronous alarm	System call	Message	Server process
Milli_delay	Function call	Busy wait	Kernel or task
Milli_elapsed	Function call	Function value	Kernel or task

Figure 3-26. The clock code supports a number of time-related services.

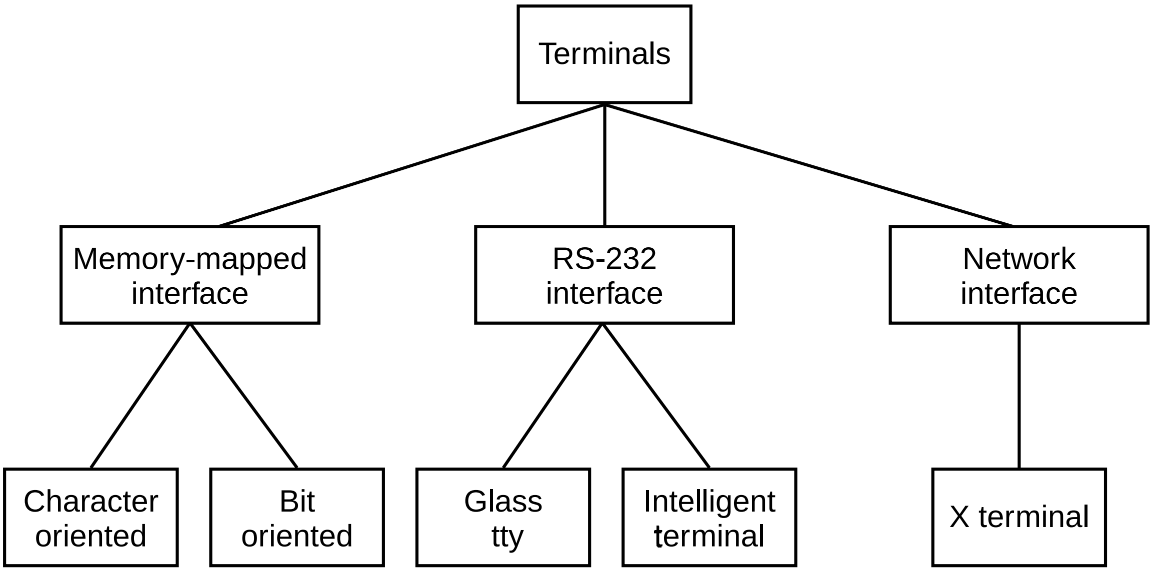


Figure 3-27. Terminal types.

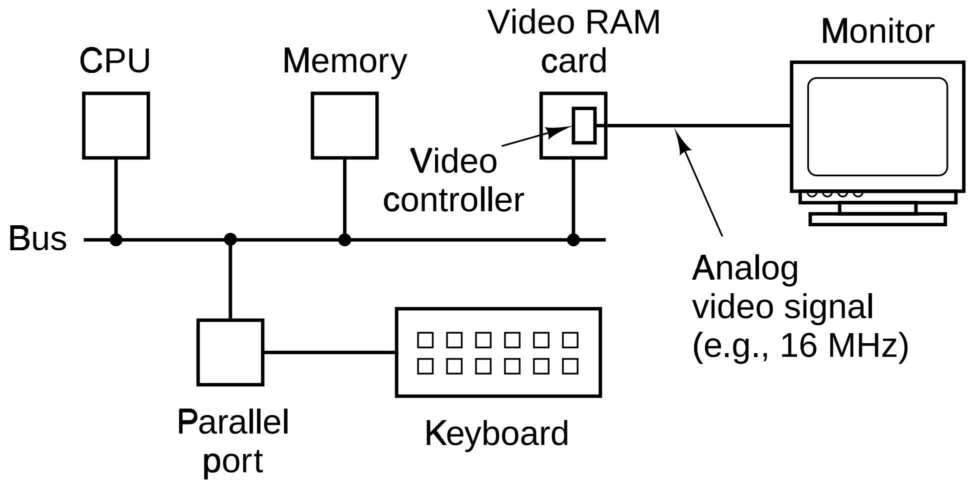


Figure 3-28. Memory-mapped terminals write directly into video RAM.

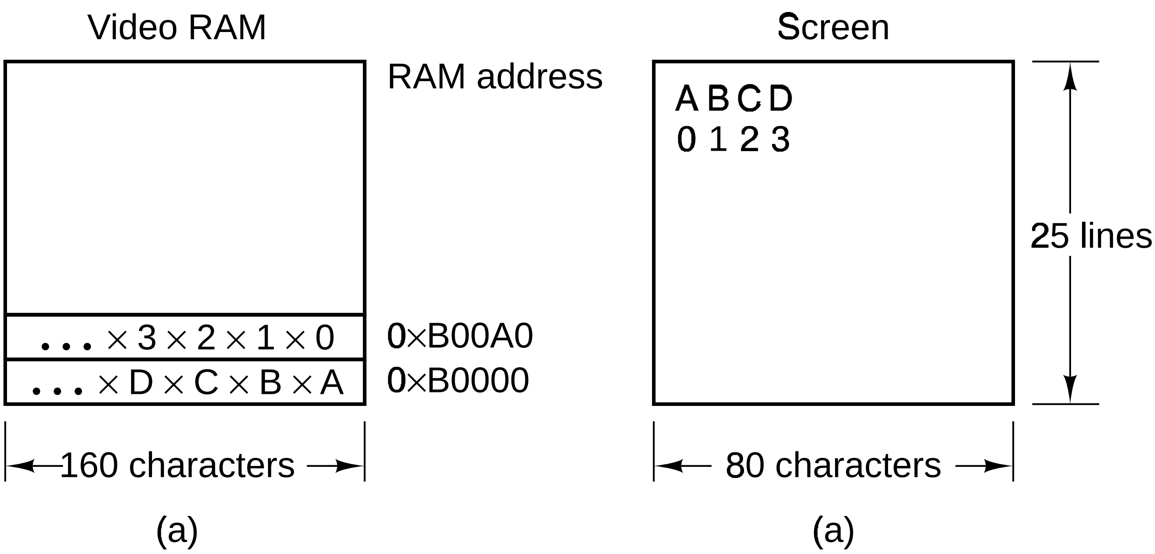


Figure 3-29. (a) A video RAM image for the IBM monochrome display. (b) The corresponding screen. The ×s are attribute bytes.

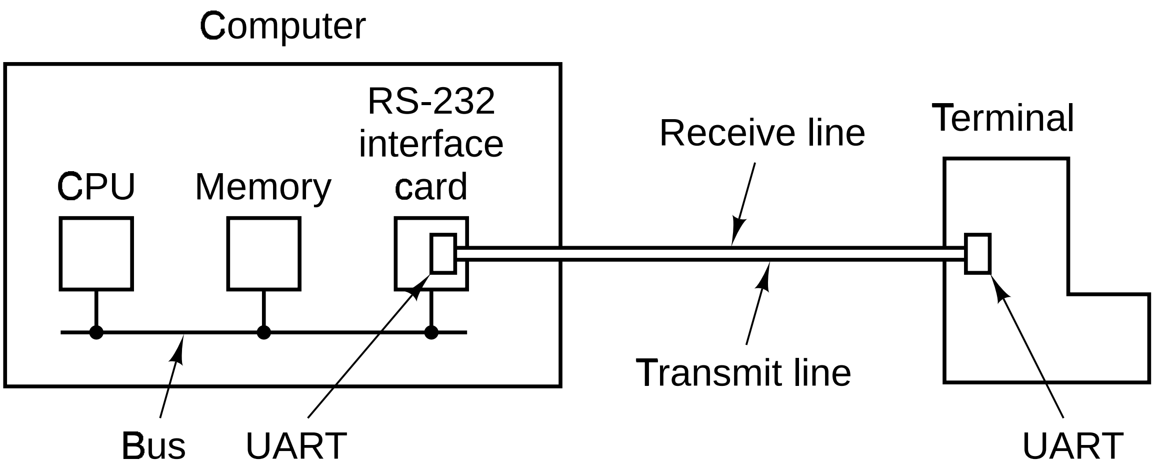


Figure 3-30. An RS-232 terminal communicates with a computer over a communication line, one bit at a time. The computer and the terminal are completely independent.

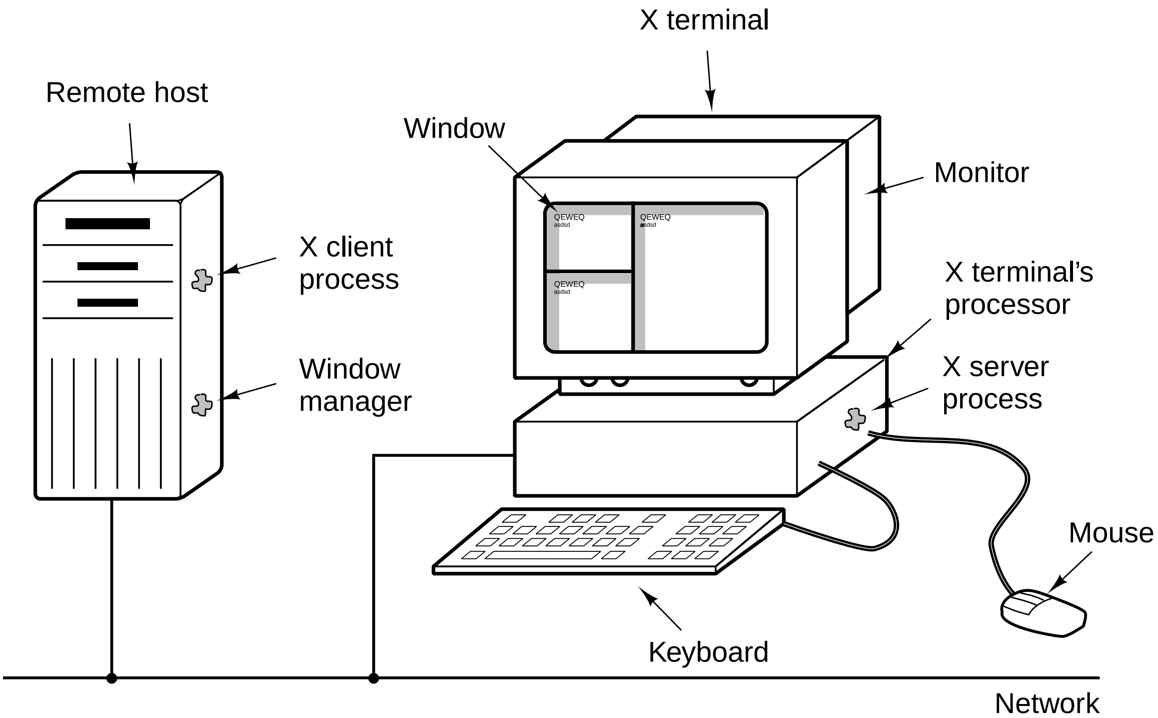


Figure 3-31. Clients and servers in the M.I.T. X Window System.

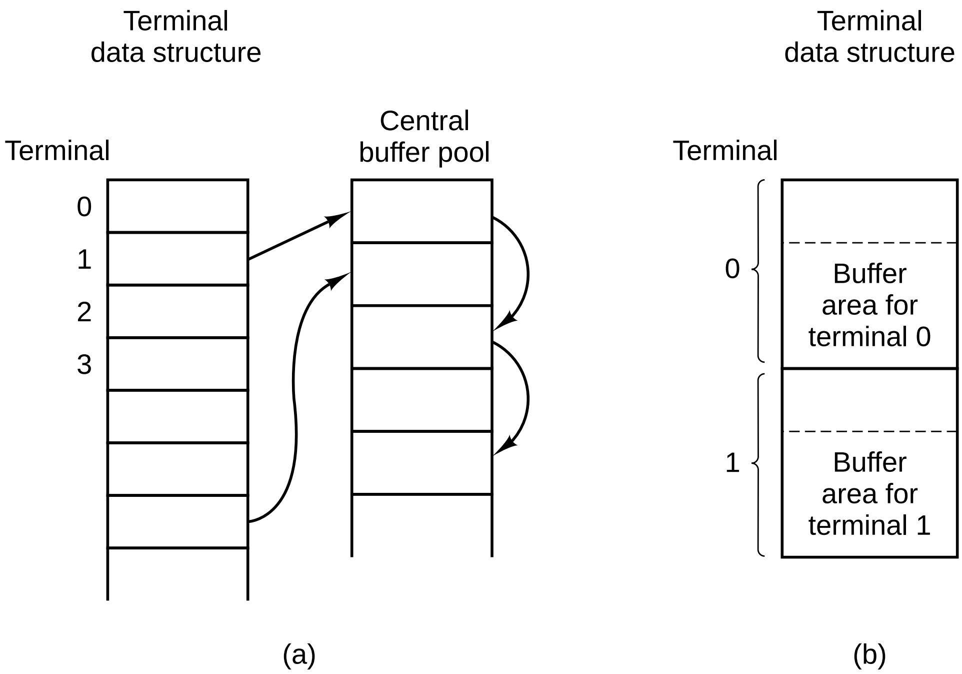


Figure 3-32. (a) Central buffer pool. (b) Dedicated buffer for each terminal.

Character	POSIX name	Comment
CTRL-D	EOF	End of file
	EOL	End of line (undefined)
CTRL-H	ERASE	Backspace one character
DEL	INTR	Interrupt process (SIGINT)
CTRL-U	KILL	Erase entire line being typed
CTRL-\	QUIT	Force core dump (SIGQUIT)
CTRL-Q	START	Start output
CTRL-S	STOP	Stop output
CTRL-R	REPRINT	Redisplay input (MINIX extension)
CTRL-V	LNEXT	Literal next (MINIX extension)
CTRL-O	DISCARD	Discard output (MINIX extension)
CTRL-M	CR	Carriage return (unchangeable)
CTRL-J	NL	Linefeed (unchangeable)

Figure 3-33. Characters that are handled specially in canonical mode.

```
struct termios {
  tcflag_t c_iflag;              /* input modes */
  tcflag_t c_oflag;              /* output modes */
  tcflag_t c_cflag;              /* control modes */
  tcflag_t c_lflag;              /* local modes */
  speed_t  c_ispeed;             /* input speed */
  speed_t  c_ospeed;             /* output speed */
  cc_t c_cc[NCCS];               /* control characters */
};
```

Figure 3-34. The termios structure. In MINIX tc_flag_t is a short, speed_t is an int, cc_t is a char.

	TIME = 0	TIME > 0
MIN = 0	Return immediately with whatever is available, 0 to N bytes	Timer starts immediately. Return with first byte entered or with 0 bytes after timeout
MIN > 0	Return with at least MIN and up to N bytes. Possible indefinite block.	Interbyte timer starts after first byte. Return N bytes if received by timeout, or at least 1 byte at timeout. Possible indefinite block

Figure 3-35. *MIN* and *TIME* determine when a call to read returns in noncanonical mode. *N* is the number of bytes requested.

Escape sequence	Meaning
ESC [*n* A	Move up *n* lines
ESC [*n* B	Move down *n* lines
ESC [*n* C	Move right *n* spaces
ESC [*n* D	Move left *n* spaces
ESC [*m* ; *n* H	Move cursor to (*m*,*n*)
ESC [*s* J	Clear screen from cursor (0 to end, 1 from start, 2 all)
ESC [*s* K	Clear line from cursor (0 to end, 1 from start, 2 all)
ESC [*n* L	Insert *n* lines at cursor
ESC [*n* M	Delete *n* lines at cursor
ESC [*n* P	Delete *n* chars at cursor
ESC [*n* @	Insert *n* chars at cursor
ESC [*n* m	Enable rendition *n* (0=normal, 4=bold, 5=blinking, 7=reverse)
ESC M	Scroll the screen backward if the cursor is on the top line

Figure 3-36. The ANSI escape sequences accepted by the terminal driver on output. ESC denotes the ASCII escape character (0x1B), and *n*, *m*, and *s* are optional numeric parameters.

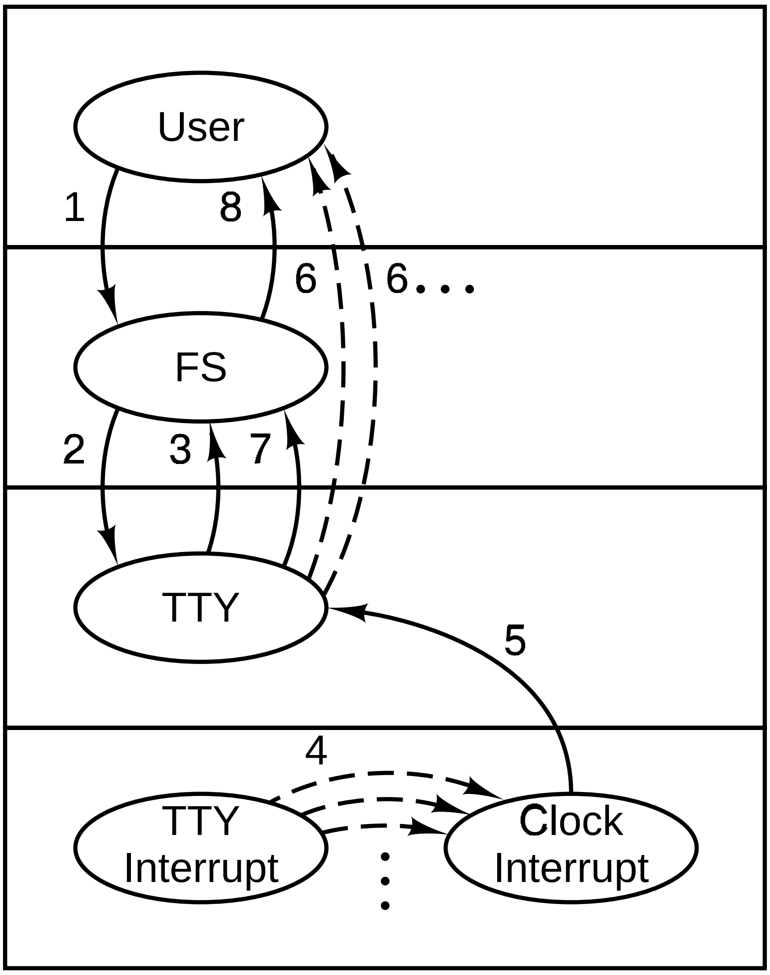


Figure 3-37. Read request from terminal when no characters are pending. FS is the file system. TTY is the terminal task. The interrupt handler for the terminal queues characters as they are entered, but it is the clock interrupt handler that awakens TTY.

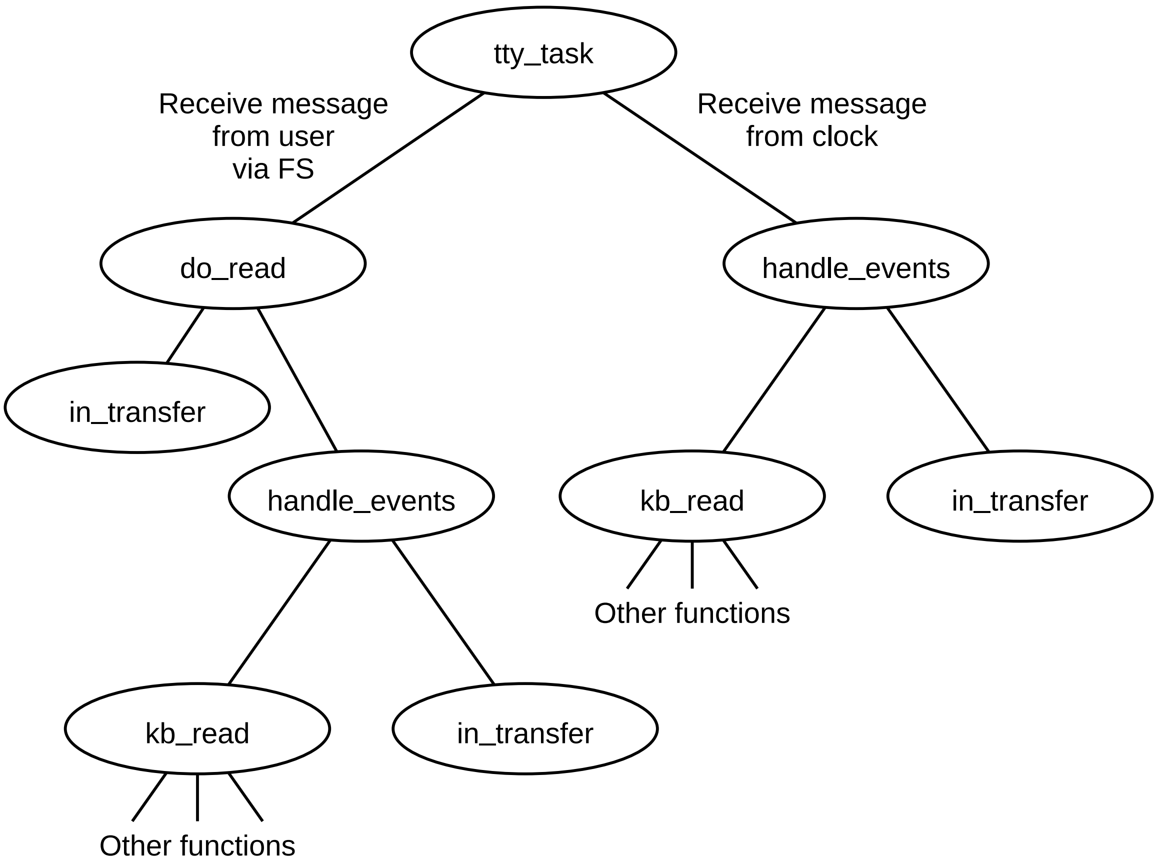


Figure 3-38. Input handling in the terminal driver. The left branch of the tree is taken to process a request to read characters. The right branch is taken when a character-has-been-typed message is sent to the driver.

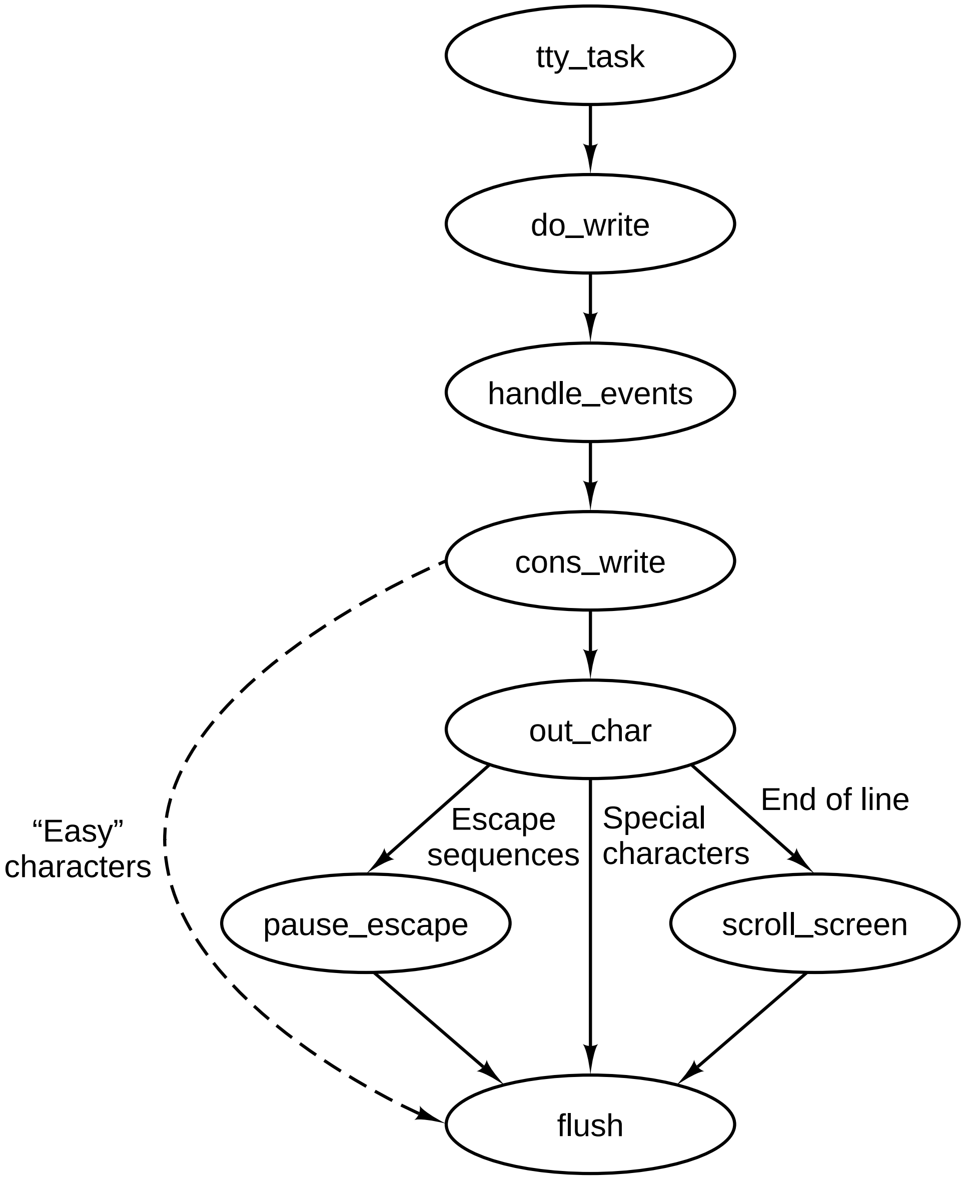


Figure 3-39. Major procedures used on terminal output. The dashed line indicates characters copied directly to *ramqueue* by *cons_write*.

Field	Meaning
c_start	Start of video memory for this console
c_limit	Limit of video memory for this console
c_column	Current column (0-79) with 0 at left
c_row	Current row (0-24) with 0 at top
c_cur	Offset into video RAM for cursor
c_org	Location in RAM pointed to by 6845 base register

Figure 3-40. Fields of the console structure that relate to the current screen position.

Scan code	Character	Regular	SHIFT	ALT1	ALT2	ALT+SHIFT	CTRL
00	none	0	0	0	0	0	0
01	ESC	C('[')	C('[')	CA('[')	CA('[')	CA('[')	C('[')
02	'1'	'1'	'!'	A('1')	A('1')	A('!')	C('A')
13	'='	'='	'+'	A('=')	A('=')	A('+')	C('@')
16	'q'	L('q')	'Q'	A('q')	A('q')	A('Q')	C('Q')
28	CR/LF	C('M')	C('M')	CA('M')	CA('M')	CA('M')	C('J')
29	CTRL	CTRL	CTRL	CTRL	CTRL	CTRL	CTRL
59	F1	F1	SF1	AF1	AF1	ASF1	CF1
127	???	0	0	0	0	0	0

Figure 3-41. A few entries from a keymap source file.

Field	Default values
c_iflag	BRKINT ICRNL IXON IXANY
c_oflag	OPOST ONLCR
c_cflag	CREAD CS8 HUPCL
c_lflag	ISIG IEXTEN ICANON ECHO ECHOE

Figure 3-42. Default termios flag values.

POSIX function	POSIX operation	IOCTL type	IOCTL parameter
tcdrain	(none)	TCDRAIN	(none)
tcflow	TCOOFF	TCFLOW	int=TCOOFF
tcflow	TCOON	TCFLOW	int=TCOON
tcflow	TCIOFF	TCFLOW	int=TCIOFF
tcflow	TCION	TCFLOW	int=TCION
tcflush	TCIFLUSH	TCFLSH	int=TCIFLUSH
tcflush	TCOFLUSH	TCFLSH	int=TCOFLUSH
tcflush	TCIOFLUSH	TCFLSH	int=TCIOFLUSH
tcgetattr	(none)	TCGETS	termios
tcsetattr	TCSANOW	TCSETS	termios
tcsetattr	TCSADRAIN	TCSETSW	termios
tcsetattr	TCSAFLUSH	TCSETSF	termios
tcsendbreak	(none)	TCSBRK	int=duration

Figure 3-43. POSIX calls and IOCTL operations.

0	V	D	N	c	c	c	c	7	6	5	4	3	2	1	0

V:	IN_ESC, escaped by LNEXT (CTRL-V)
D:	IN_EOF, end of file (CTRL-D)
N:	IN_EOT, line break (NL and others)
cccc:	count of characters echoed
7:	Bit 7, may be zeroed if ISTRIP is set
6-0:	Bits 0-6, ASCII code

Figure 3-44. The fields in a character code as it is placed into the input queue.

42	35	170	18	38	38	24	57	54	17	182	24	19	38	32	28

Figure 3-45. Scan codes in the input buffer, with corresponding key presses below, for a line of text entered at the keyboard. L+, L-, R+, and R- represent, respectively, pressing and releasing the left and right Shift keys. The code for a key release is 128 more than the code for a press of the same key.

Key	Scan code	"ASCII"	Escape sequence
Home	71	0x101	ESC [H
Up Arrow	72	0x103	ESC [A
Pg Up	73	0x107	ESC [V
–	74	0x10A	ESC [S
Left Arrow	75	0x105	ESC [D
5	76	0x109	ESC [G
Right Arrow	77	0x106	ESC [C
+	78	0x10B	ESC [T
End	79	0x102	ESC [Y
Down Arrow	80	0x104	ESC [B
Pg Dn	81	0x108	ESC [U
Ins	82	0x10C	ESC [@

Figure 3-46. Escape codes generated by the numeric keypad. When scan codes for ordinary keys are translated into ASCII codes the special keys are assigned ''pseudo ASCII'' codes with values greater than 0xFF.

Key	Purpose
F1	Display process table
F2	Display details of process memory use
F3	Toggle between hardware and software scrolling
F5	Show Ethernet statistics (if network support compiled)
CF7	Send SIGQUIT, same effect as CTRL-\
CF8	Send SIGINT, same effect as DEL
CF9	Send SIGKILL, same effect as CTRL-U

Figure 3-47. The function keys detected by *func_key()*.

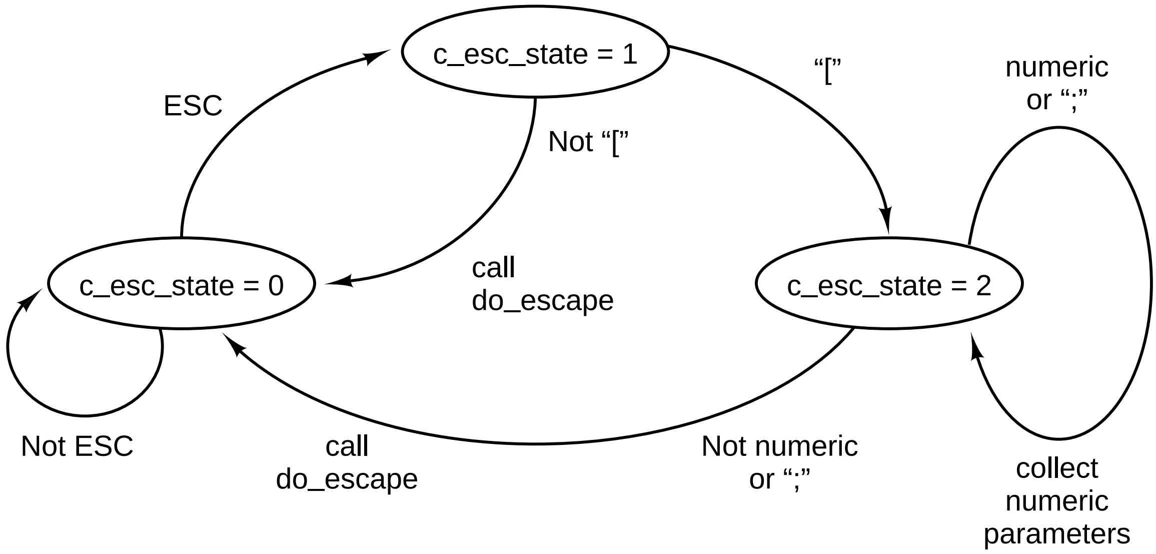


Figure 3-48. Finite state machine for processing escape sequences.

Registers	Function
10 – 11	Cursor size
12 – 13	Start address for drawing screen
14 – 15	Cursor position

Figure 3-49. Some of the 6845's registers.

Message type	From	Meaning
SYS_FORK	MM	A process has forked
SYS_NEWMAP	MM	Install memory map for a new process
SYS_GETMAP	MM	MM wants memory map of a process
SYS_EXEC	MM	Set stack pointer after EXEC call
SYS_XIT	MM	A process has exited
SYS_TIMES	FS	FS wants a process' execution times
SYS_ABORT	Both	Panic: MINIX is unable to continue
SYS_SENDSIG	MM	Send a signal to a process
SYS_SIGRETURN	MM	Cleanup after completion of a signal.
SYS_KILL	FS	Send signal to a process after KILL call
SYS_ENDSIG	MM	Cleanup after a signal from the kernel
SYS_COPY	Both	Copy data between processes
SYS_VCOPY	Both	Copy multiple blocks of data between processes
SYS_GBOOT	FS	Get boot parameters
SYS_MEM	MM	MM wants next free chunk of physical memory
SYS_UMAP	FS	Convert virtual address to physical address
SYS_TRACE	MM	Carry out an operation of the PTRACE call

Figure 3-50. The message types accepted by the system task.

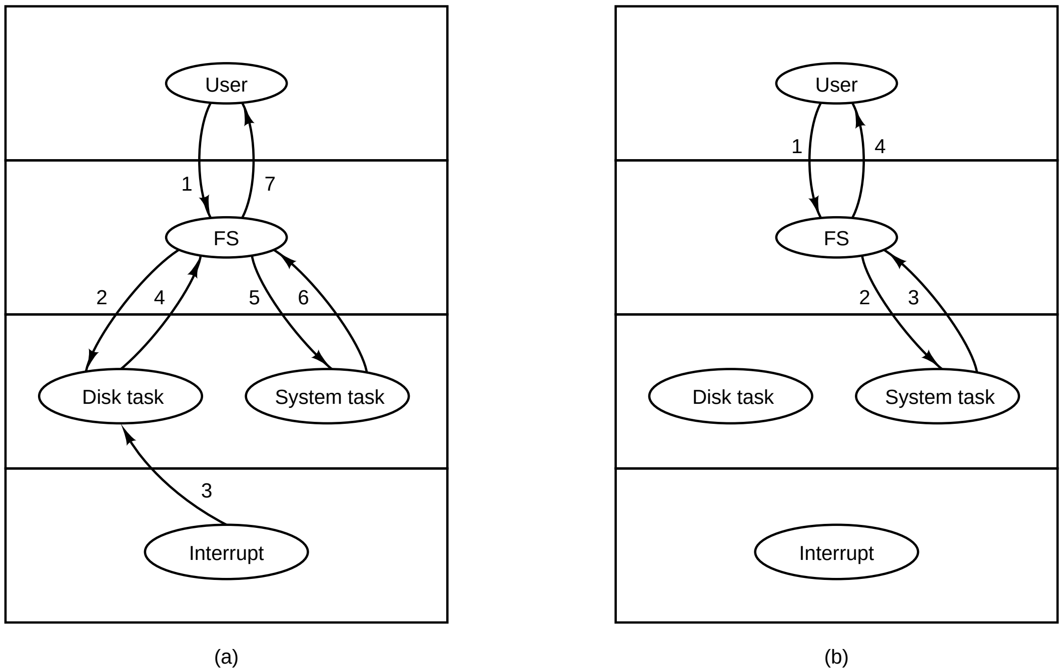


Figure 3-51. (a) Worst case for reading a block requires seven messages. (b) Best case for reading a block requires four messages.

4

MEMORY MANAGEMENT

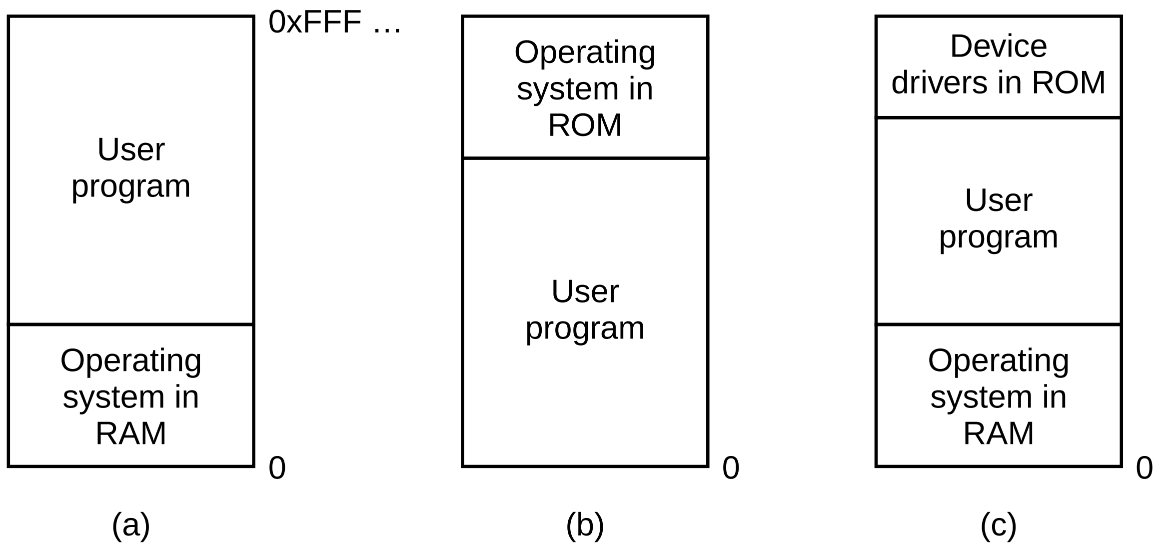


Figure 4-1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

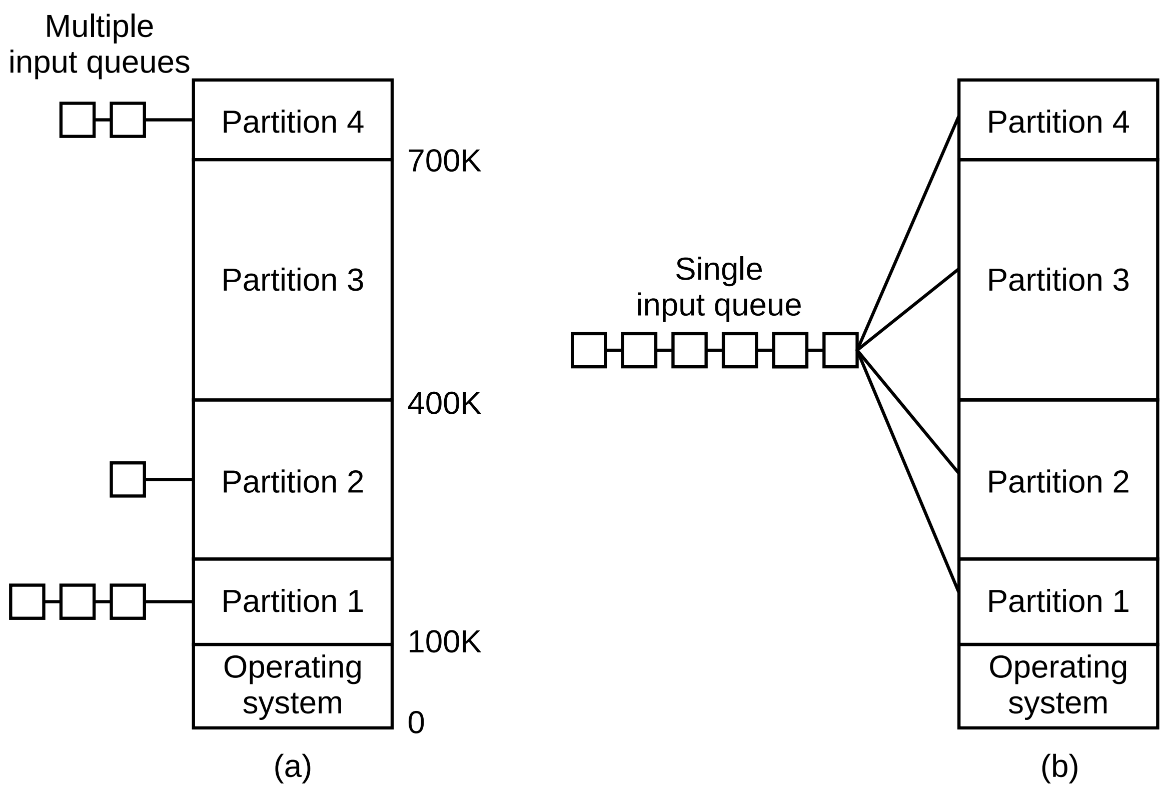


Figure 4-2. (a) Fixed memory partitions with separate input queues for each partition. (b) Fixed memory partitions with a single input queue.

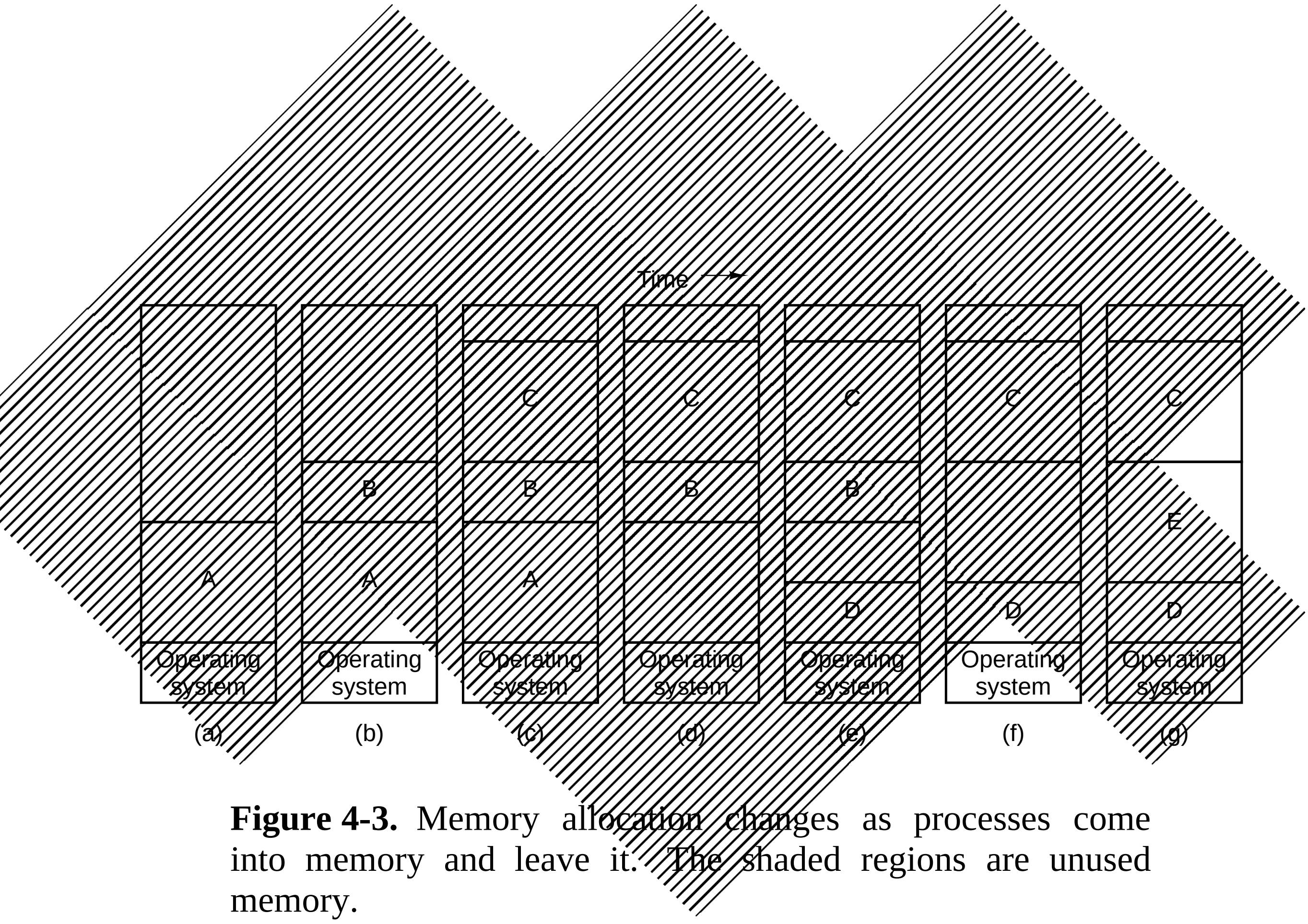


Figure 4-3. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

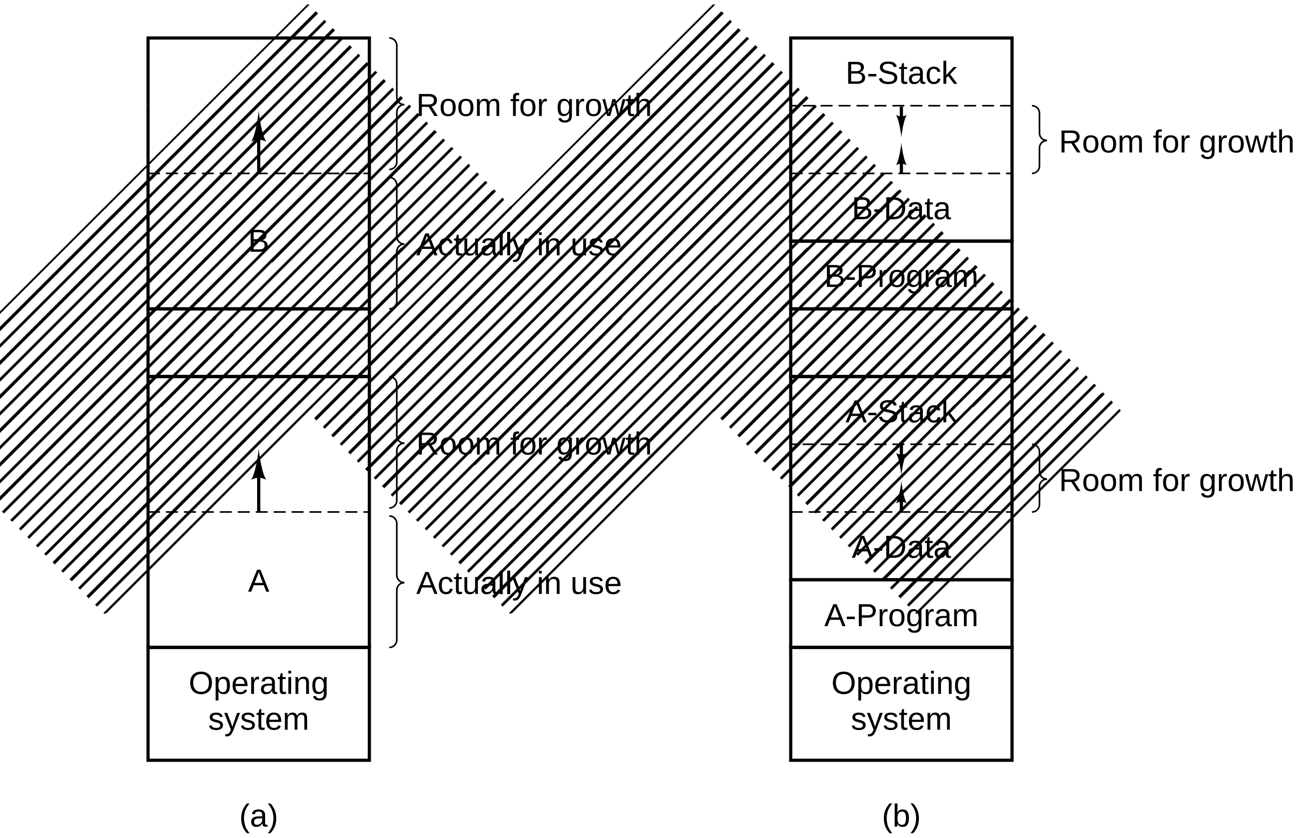


Figure 4-4. (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

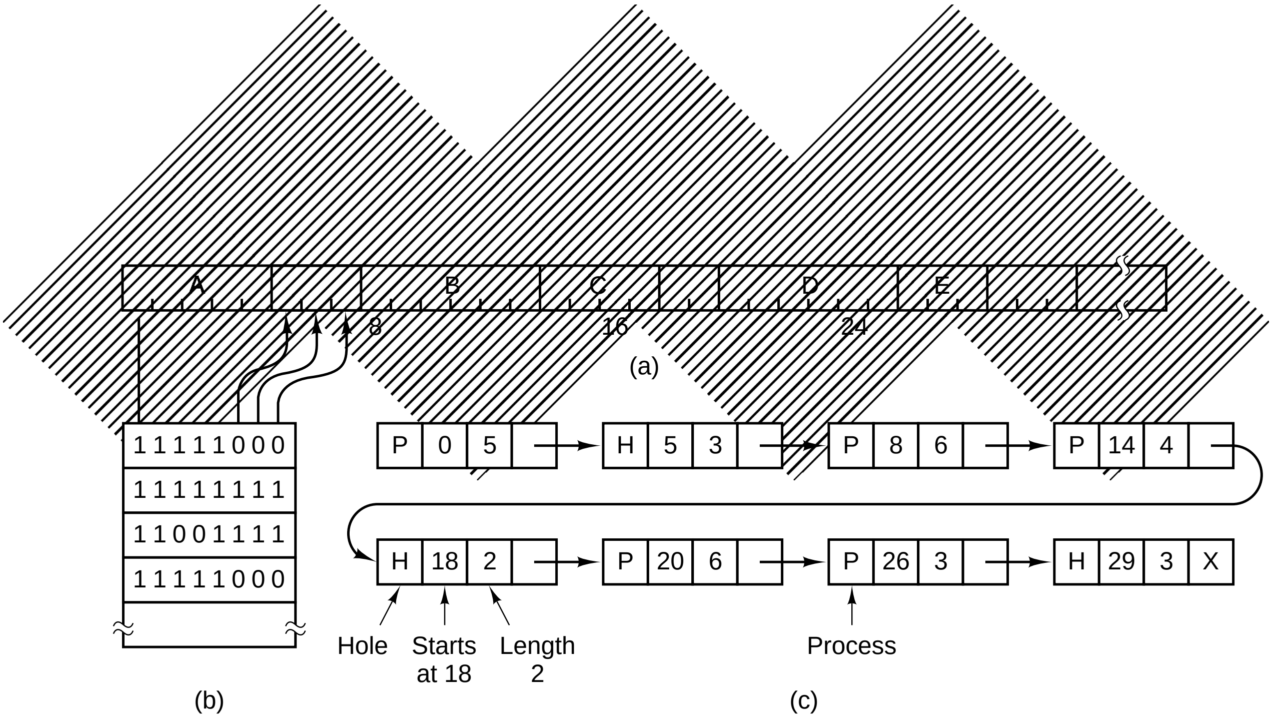


Figure 4-5. (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bit map) are free. (b) The corresponding bit map. (c) The same information as a list.

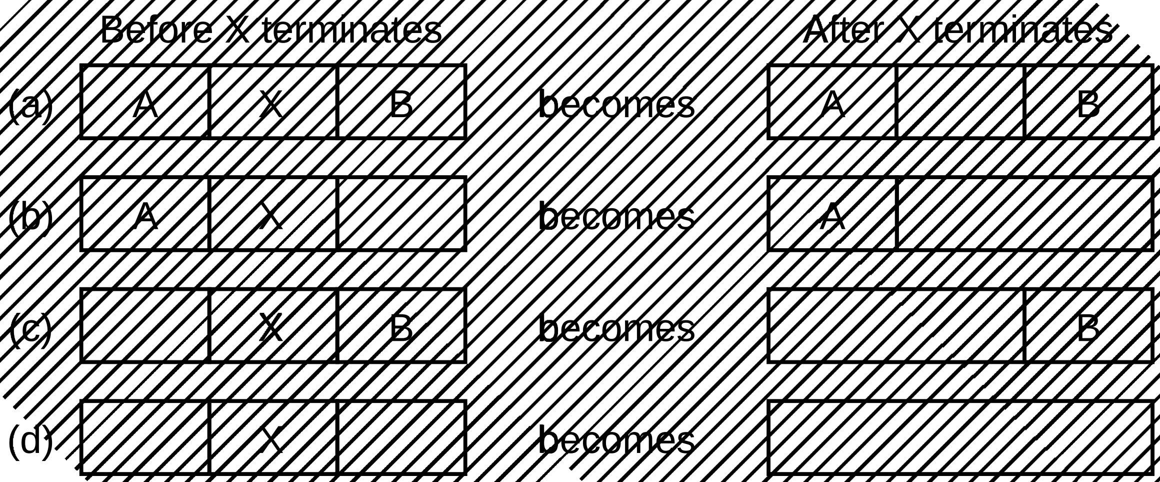


Figure 4-6. Four neighbor combinations for the terminating process, *X*.

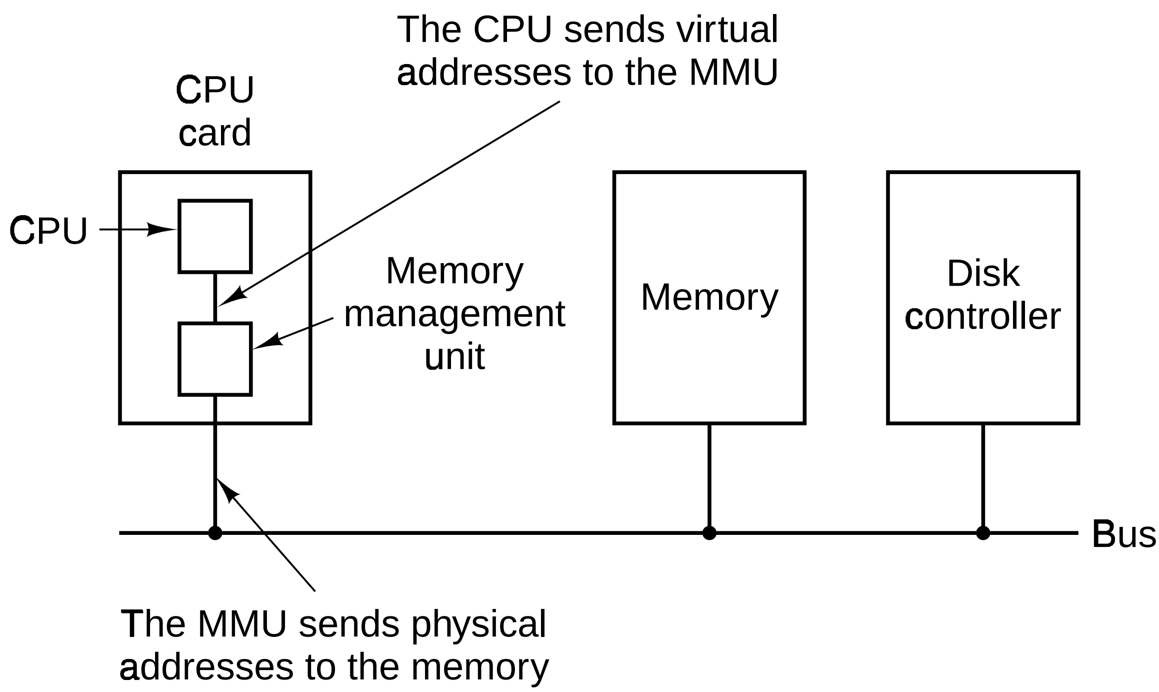


Figure 4-7. The position and function of the MMU.

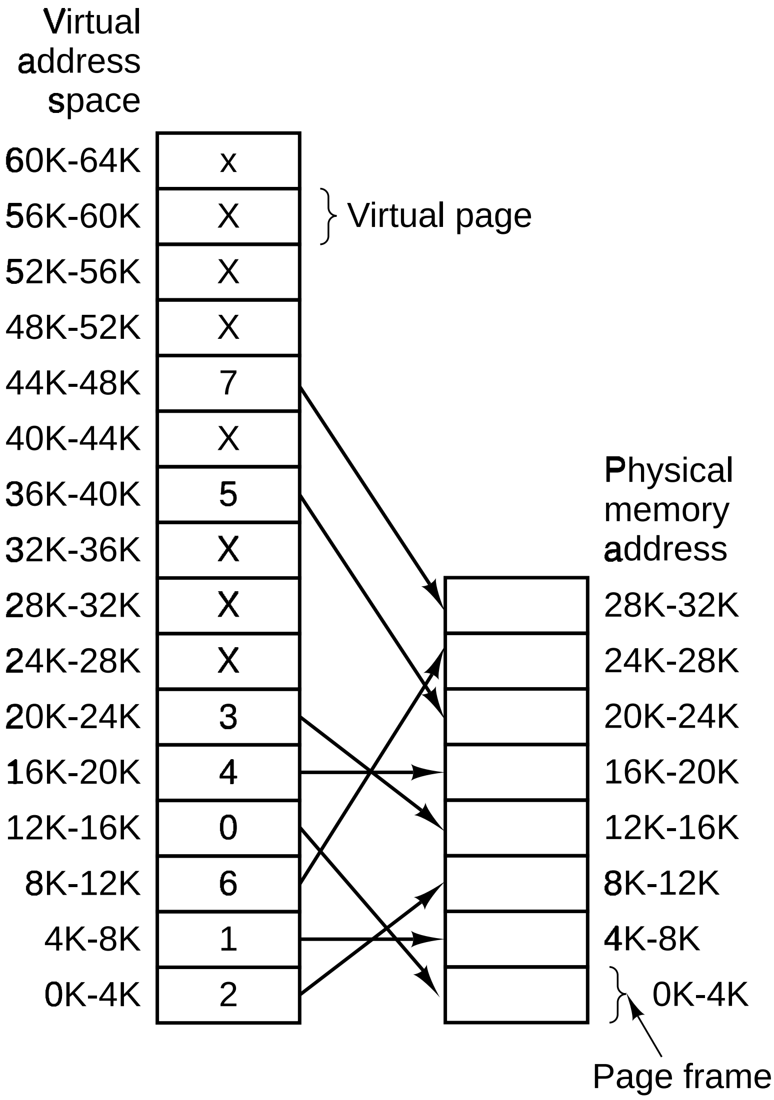


Figure 4-8. The relation between virtual addresses and physical memory addresses is given by the page table.

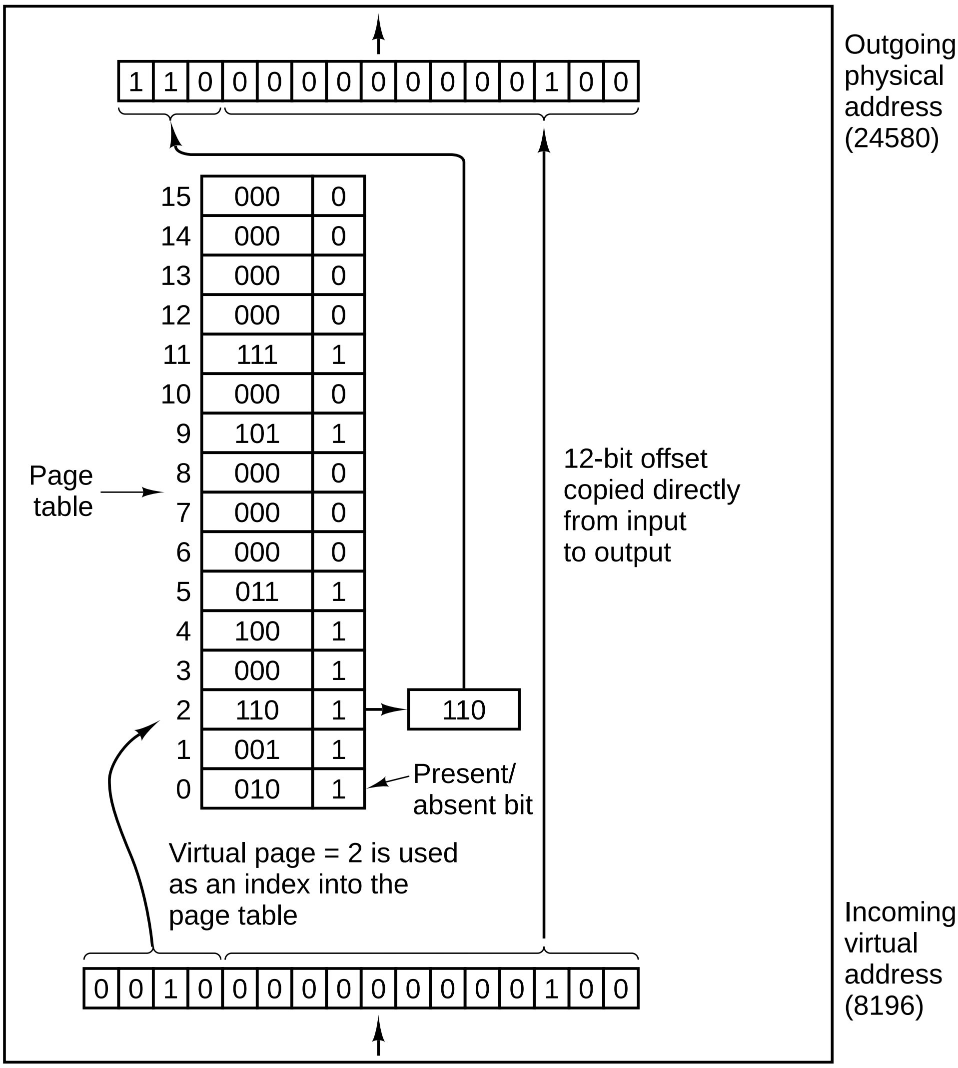


Figure 4-9. The internal operation of the MMU with 16 4K pages.

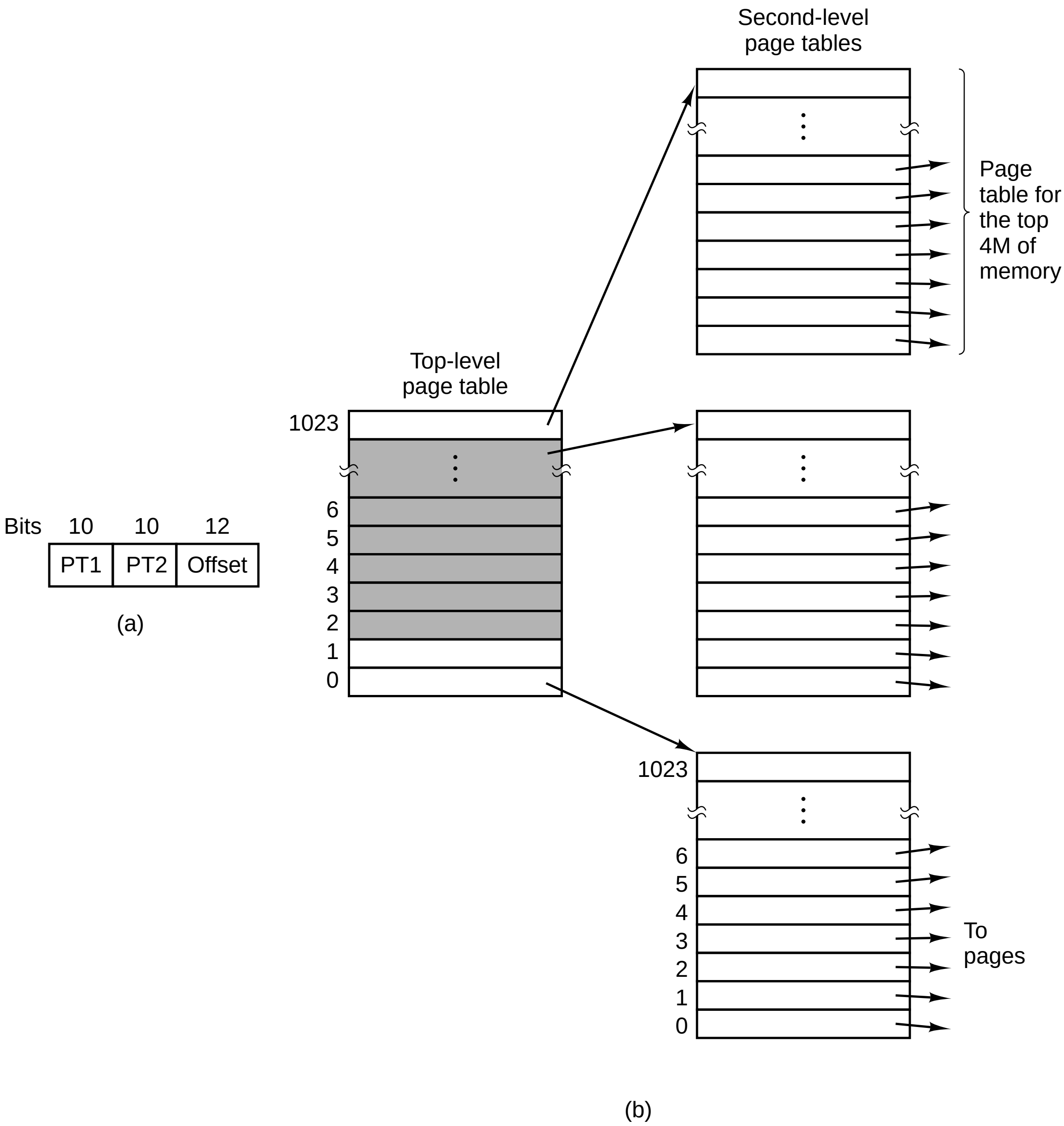


Figure 4-10. (a) A 32-bit address with two page table fields. (b) Two-level page tables.

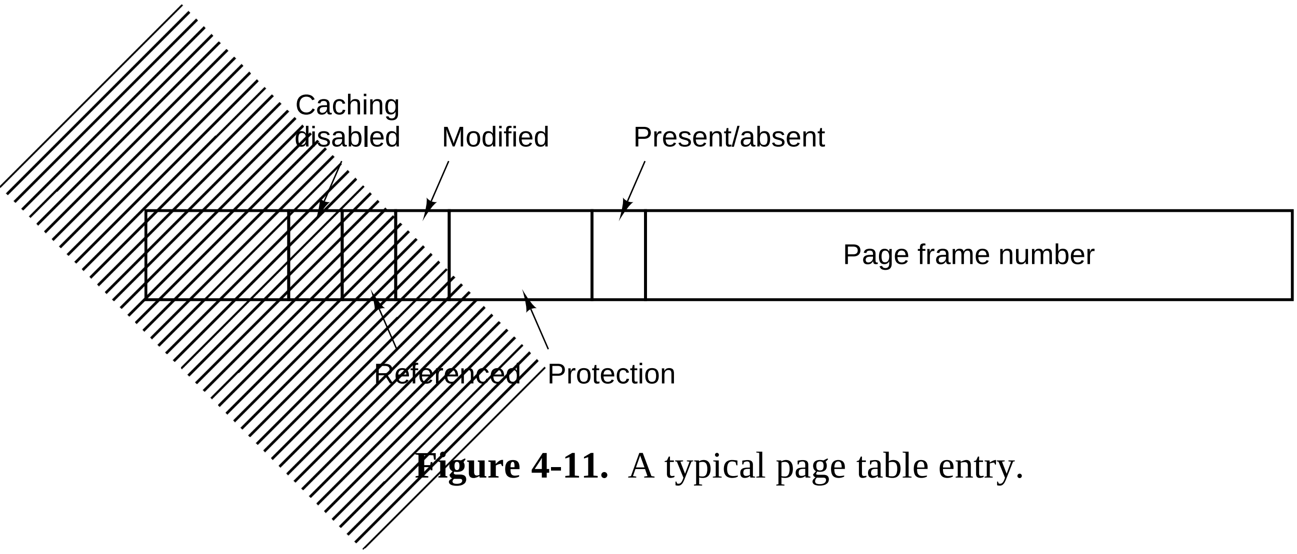


Figure 4-11. A typical page table entry.

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figure 4-12. A TLB to speed up paging.

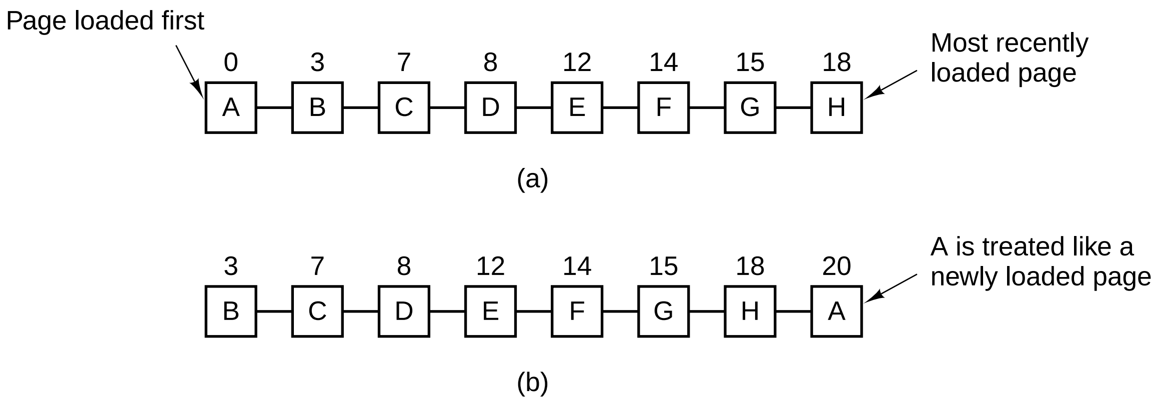


Figure 4-13. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and *A* has its *R* bit set.

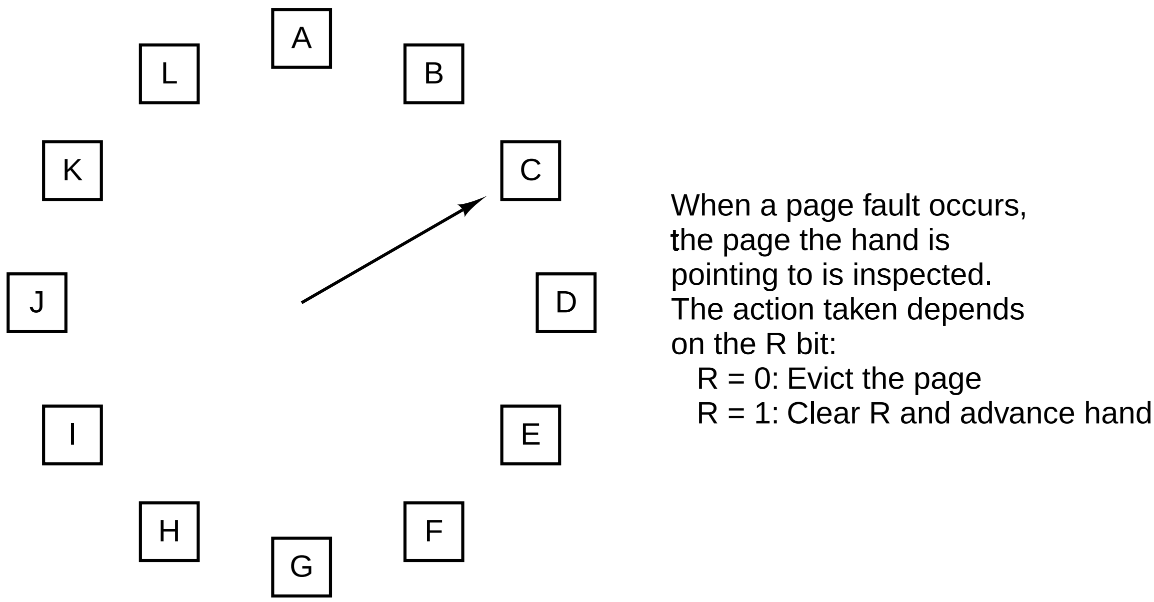


Figure 4-14. The clock page replacement algorithm.

Page

	0	1	2	3
0	0	1	1	1
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

(a)

Page

0	1	2	3
0	0	1	1
1	0	1	1
0	0	0	0
0	0	0	0

(b)

Page

0	1	2	3
0	0	0	1
1	0	0	1
1	1	0	1
0	0	0	0

(c)

Page

0	1	2	3
0	0	0	0
1	0	0	0
1	1	0	0
1	1	1	0

(d)

Page

0	1	2	3
0	0	0	0
1	0	0	0
1	1	0	1
1	1	0	0

(e)

0	0	0	0
1	0	1	1
1	0	0	1
1	0	0	0

(f)

0	1	1	1
0	0	1	1
0	0	0	1
0	0	0	0

(g)

0	1	1	0
0	0	1	0
0	0	0	0
1	1	1	0

(h)

0	1	0	0
0	0	0	0
1	1	0	1
1	1	0	0

(i)

0	1	0	0
0	0	0	0
1	1	0	0
1	1	1	0

(j)

Figure 4-15. LRU using a matrix.

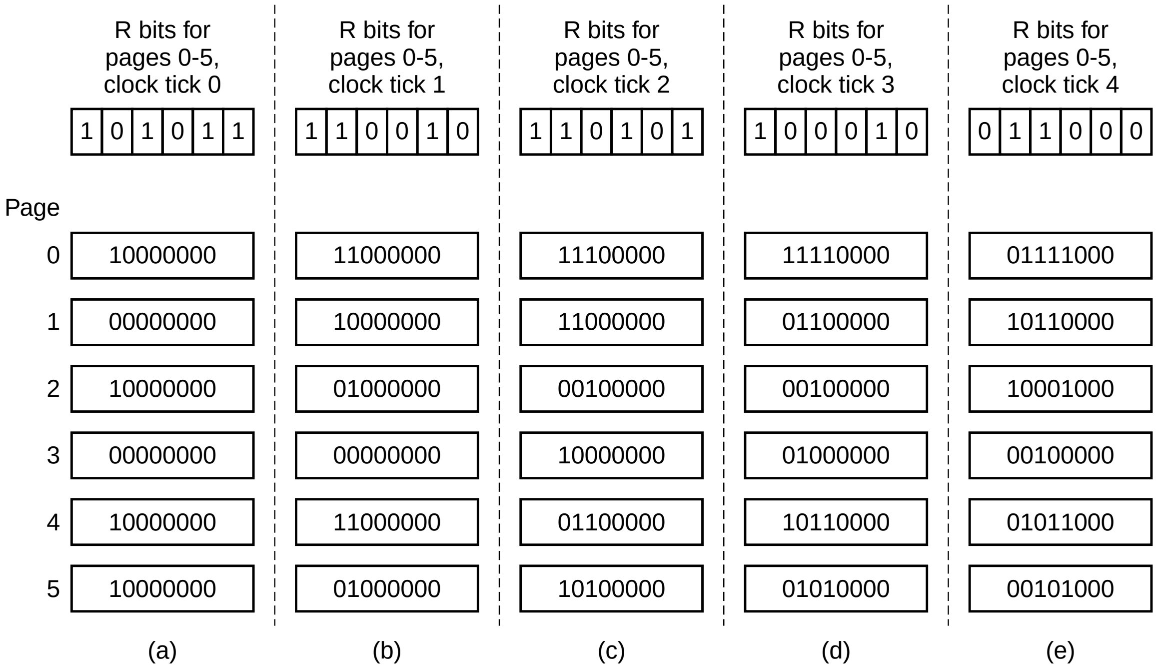


Figure 4-16. The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).

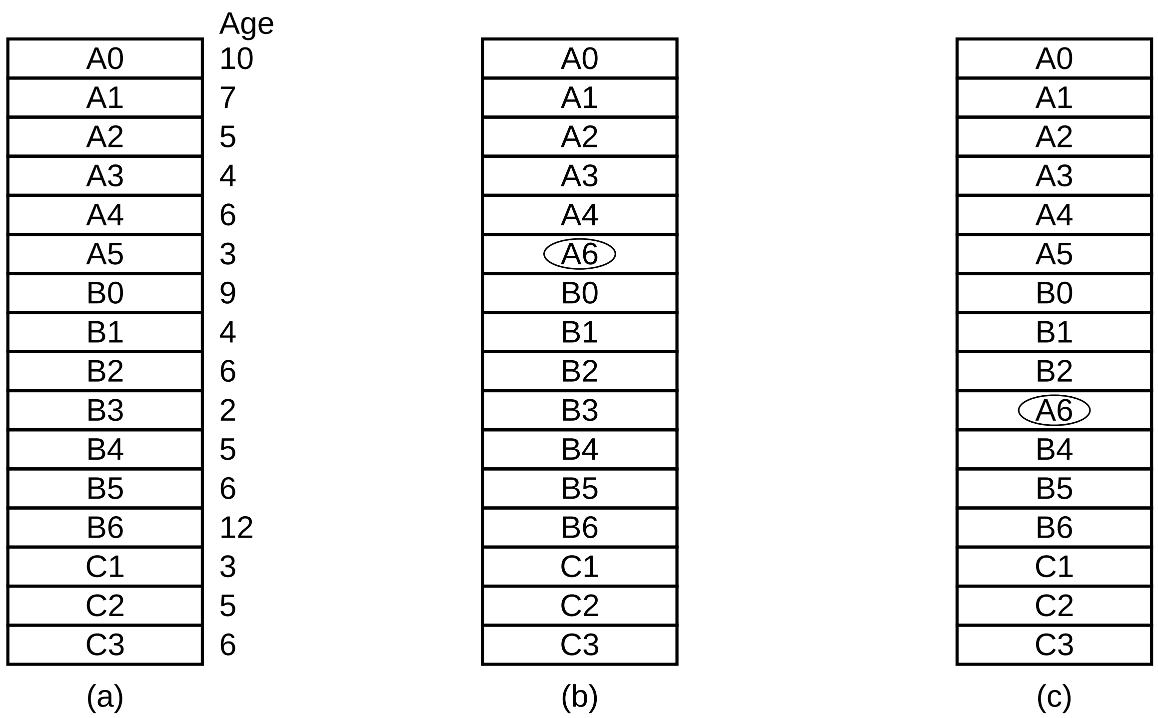


Figure 4-17. Local versus global page replacement. (a) Original configuration. (b) Local page replacement. (c) Global page replacement.

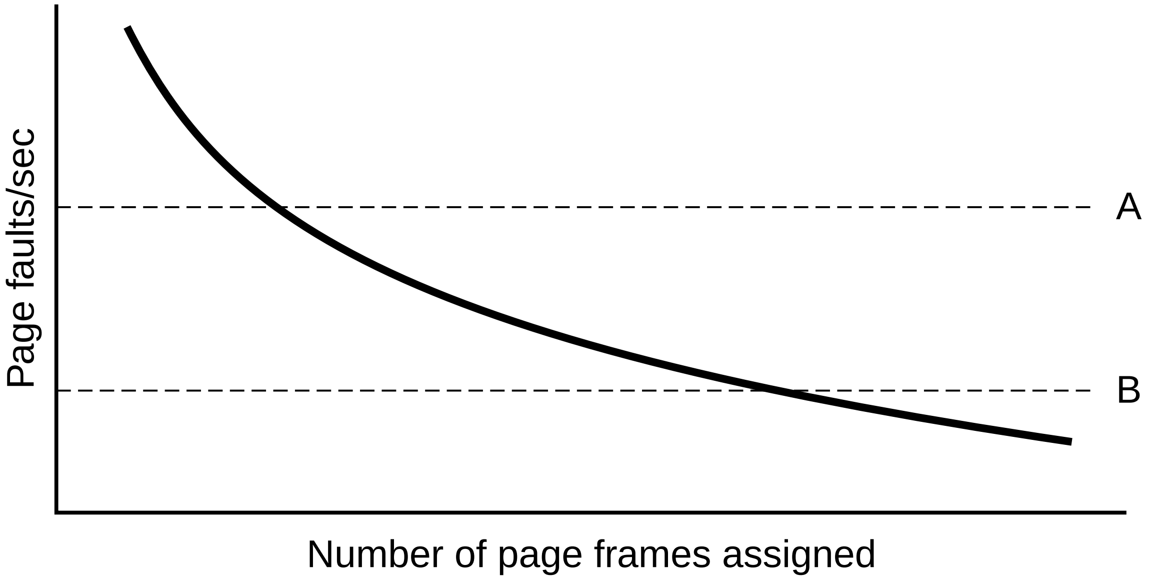


Figure 4-18. Page fault rate as a function of the number of page frames assigned.

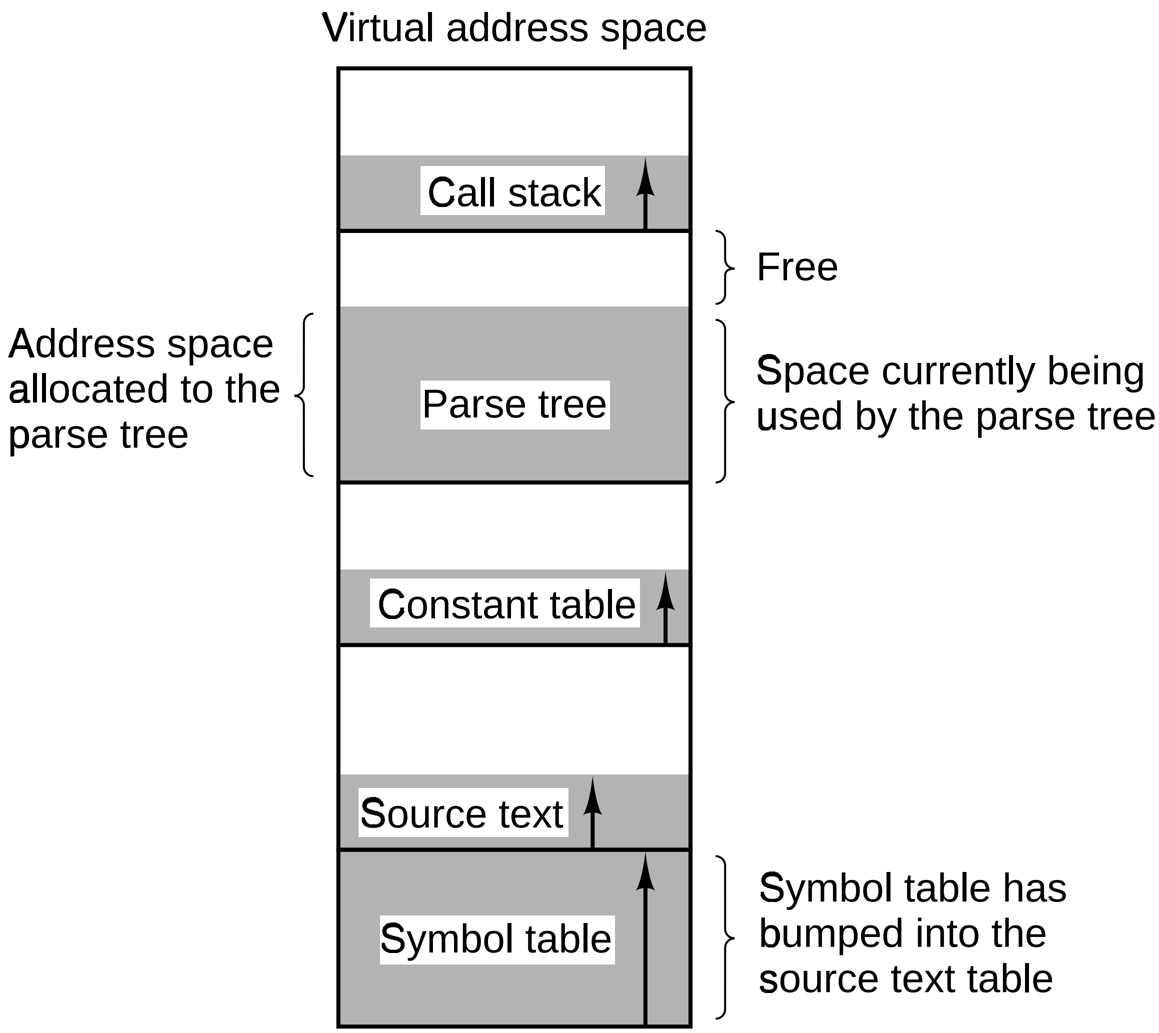


Figure 4-19. In a one-dimensional address space with growing tables, one table may bump into another.

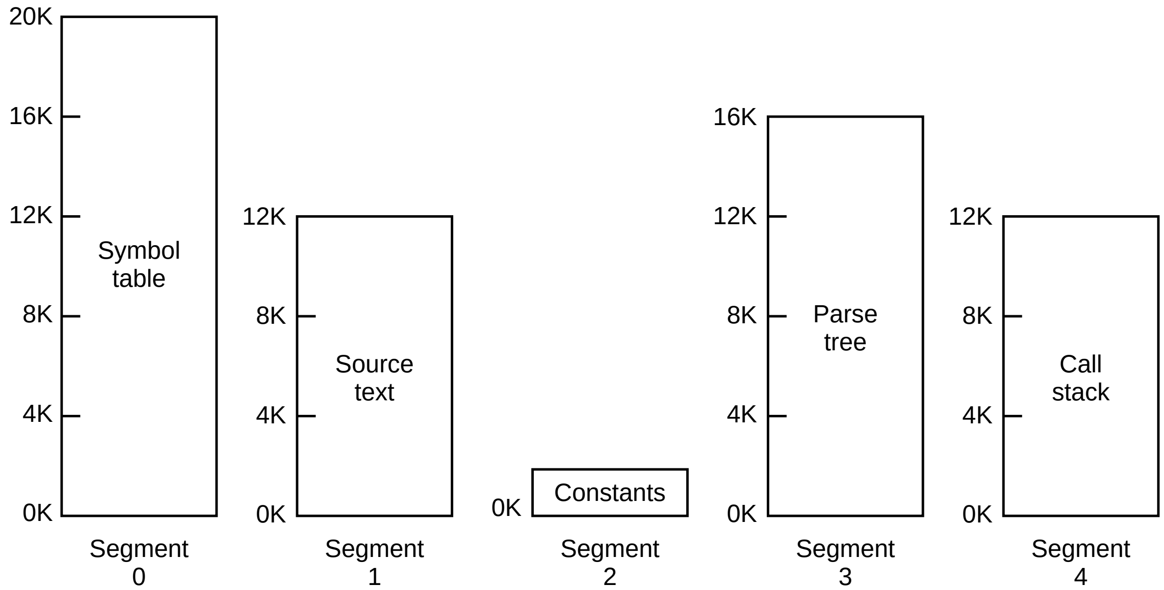


Figure 4-20. A segmented memory allows each table to grow or shrink independently of the other tables.

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Figure 4-21. Comparison of paging and segmentation.

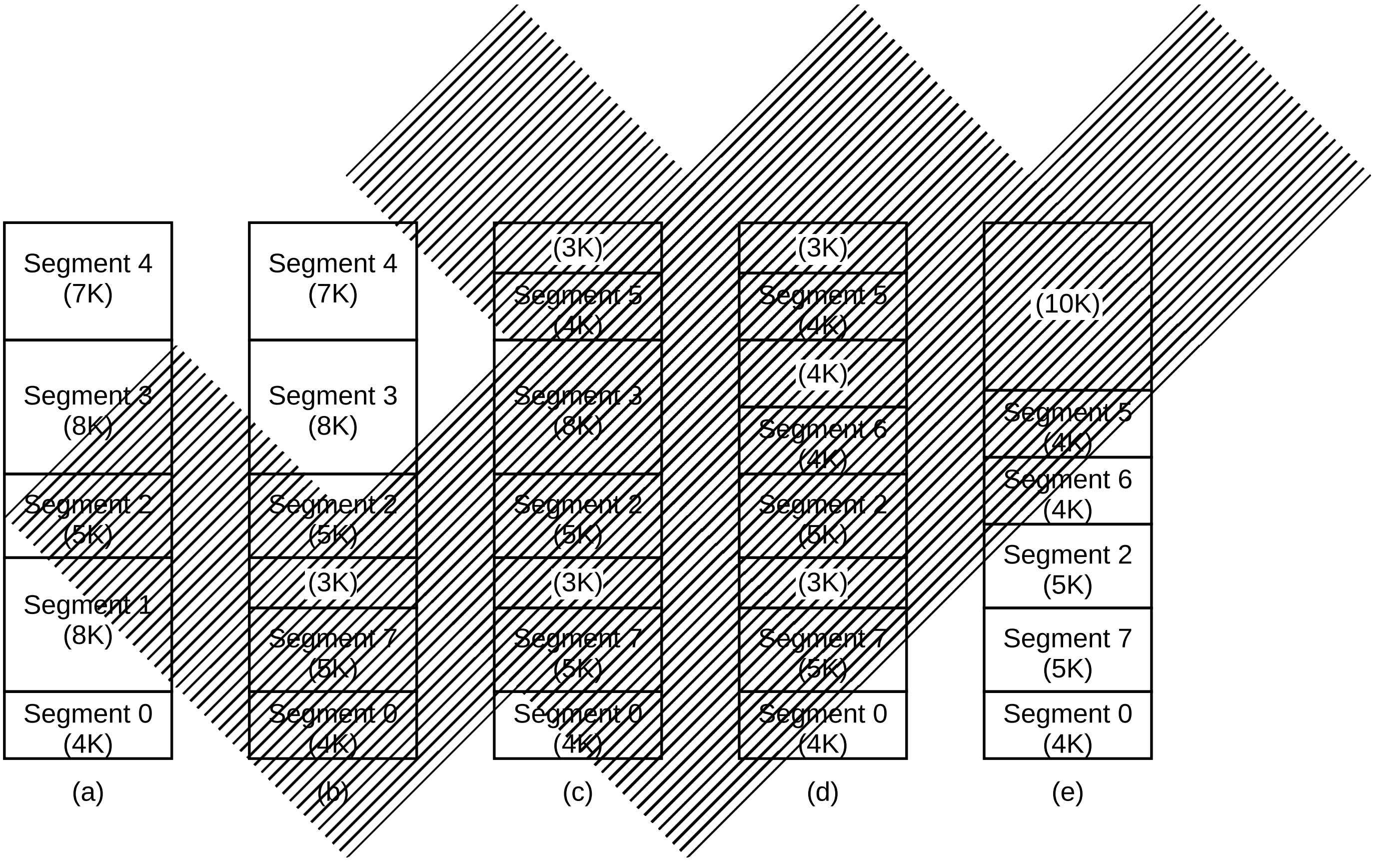


Figure 4-22. (a)-(d) Development of checkerboarding. (e) Removal of the checkerboarding by compaction.

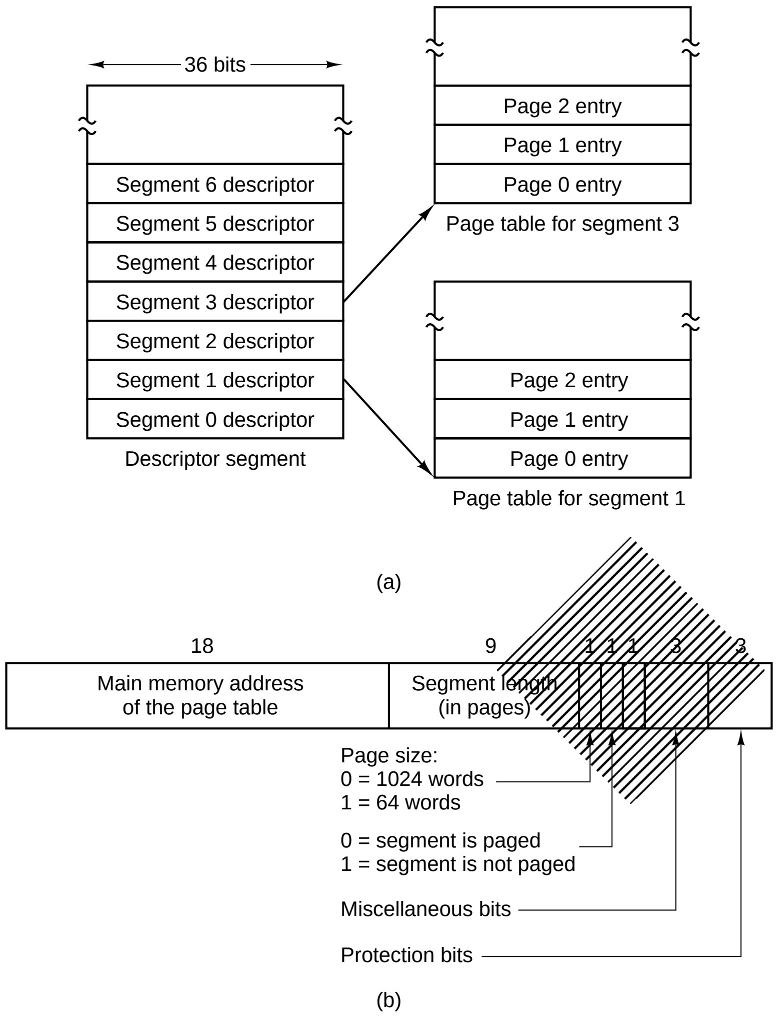


Figure 4-23. The MULTICS virtual memory. (a) The descriptor segment points to the page tables. (b) A segment descriptor. The numbers are the field lengths.

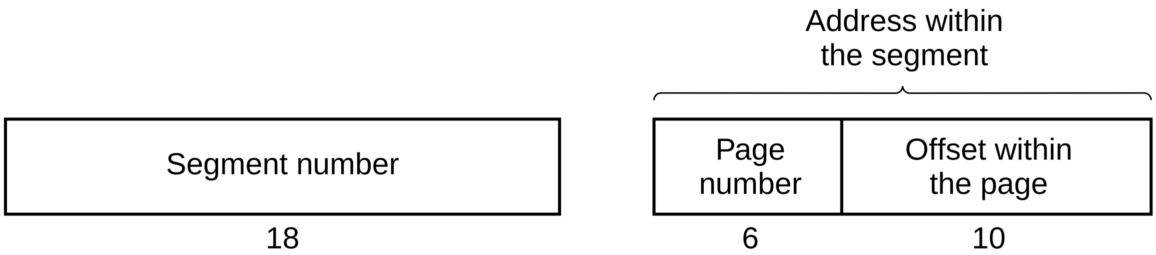


Figure 4-24. A 34-bit MULTICS virtual address.

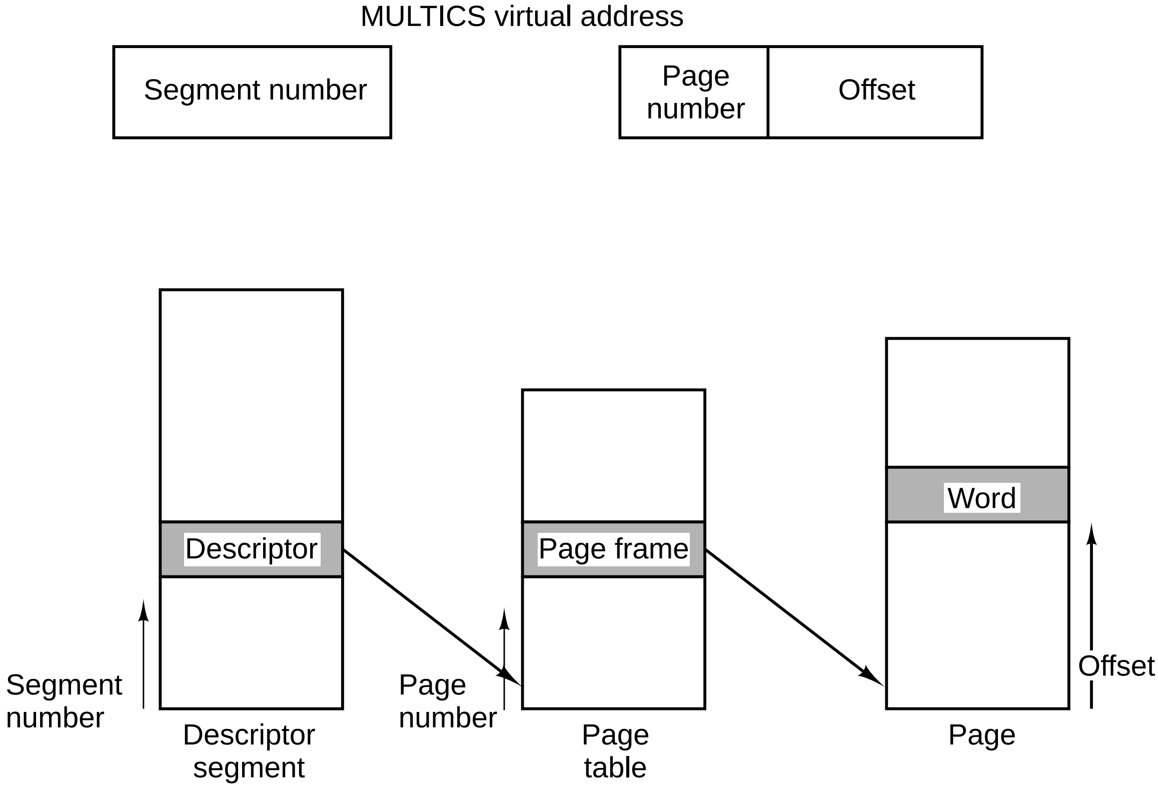


Figure 4-25. Conversion of a two-part MULTICS address into a main memory address.

Comparison field					
Segment number	Virtual page	Page frame	Protection	Age	Is this entry used?
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

Figure 4-26. A simplified version of the MULTICS TLB. The existence of two page sizes makes the actual TLB more complicated.

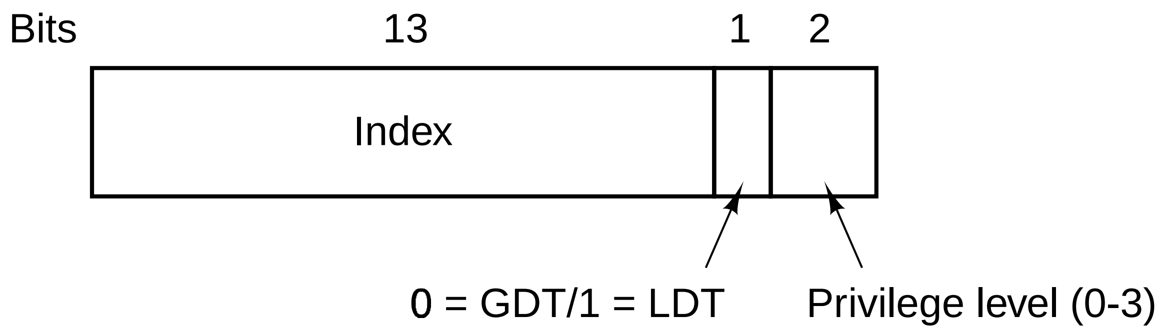


Figure 4-27. A Pentium selector.

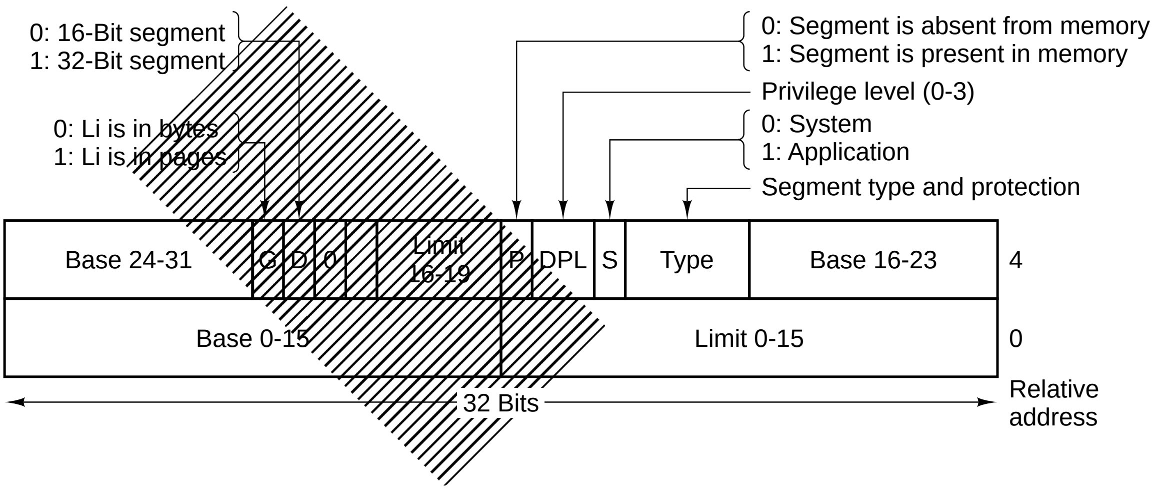


Figure 4-28. Pentium code segment descriptor. Data segments differ slightly.

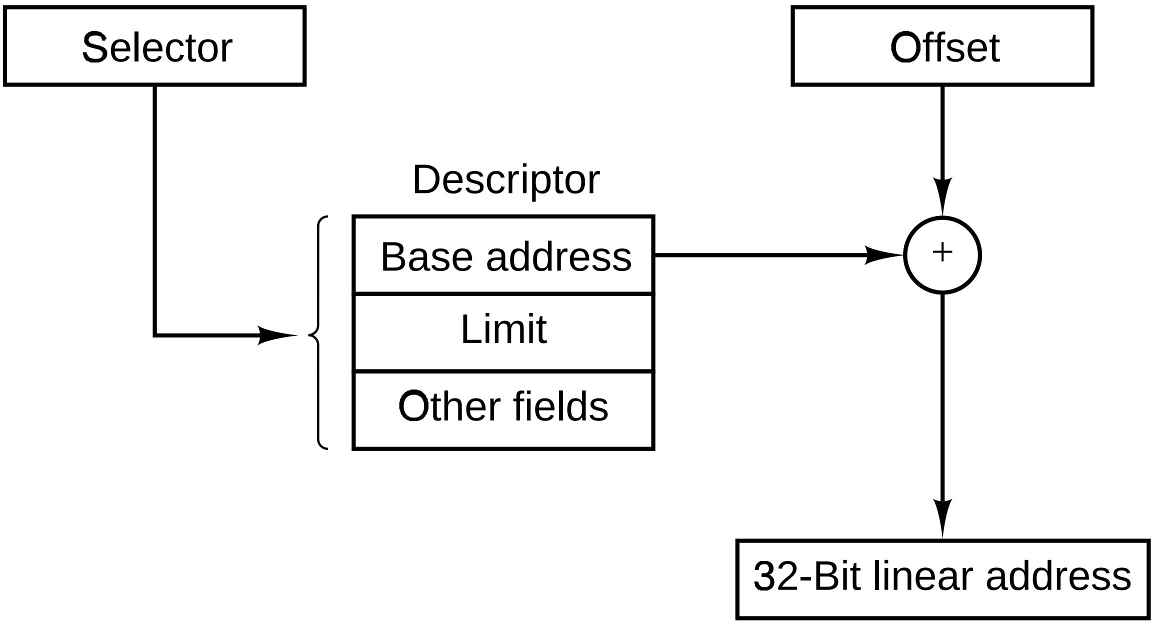


Figure 4-29. Conversion of a (selector, offset) pair to a linear address.

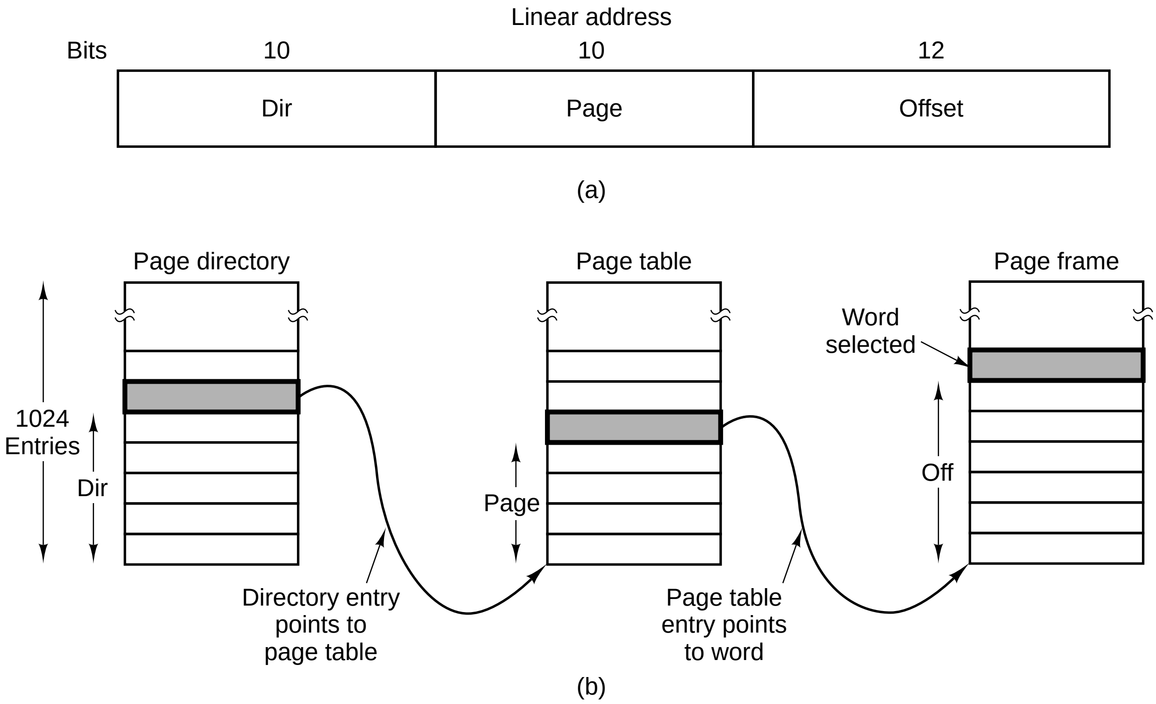


Figure 4-30. Mapping of a linear address onto a physical address.

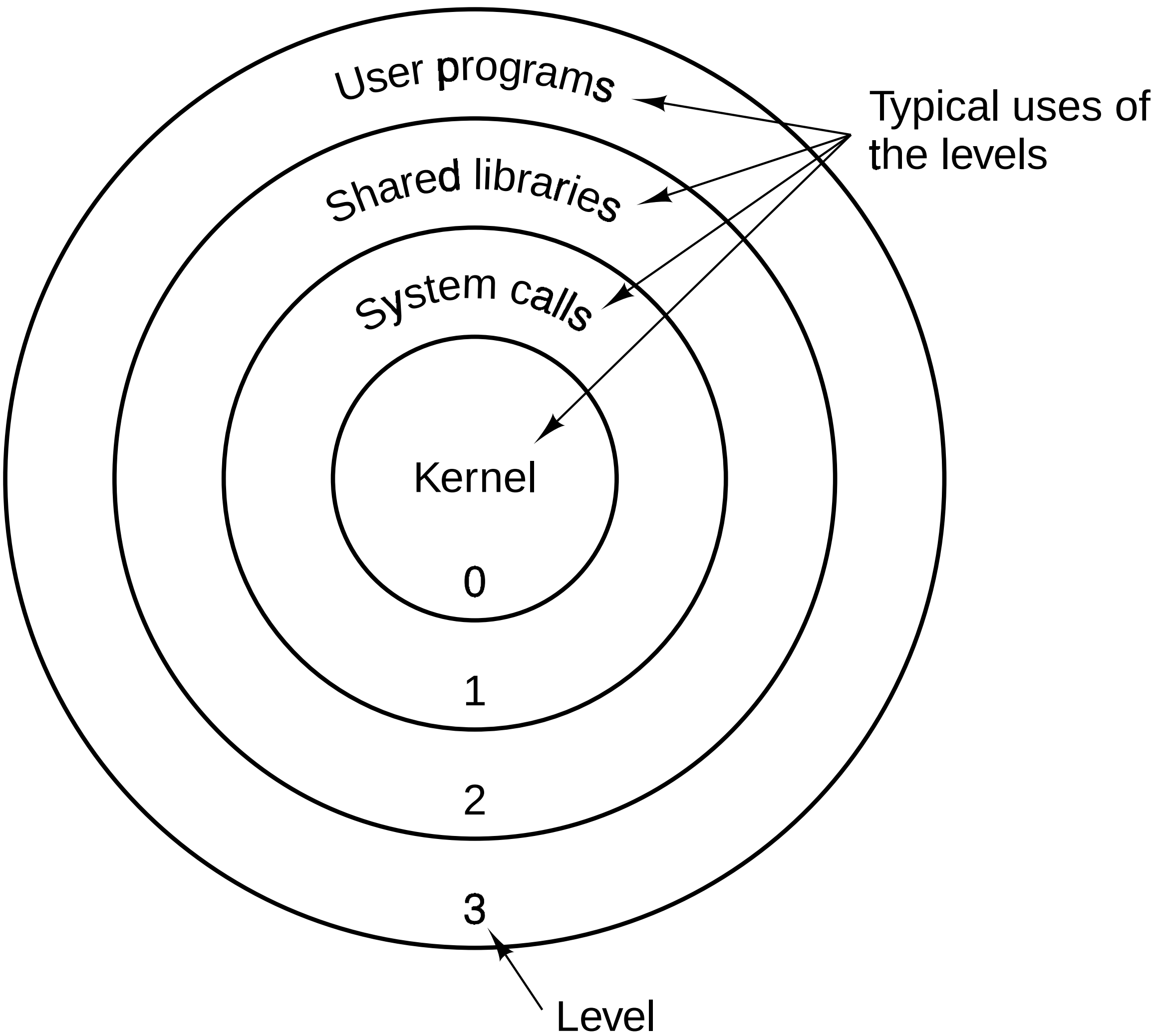


Figure 4-31. Protection on the Pentium.

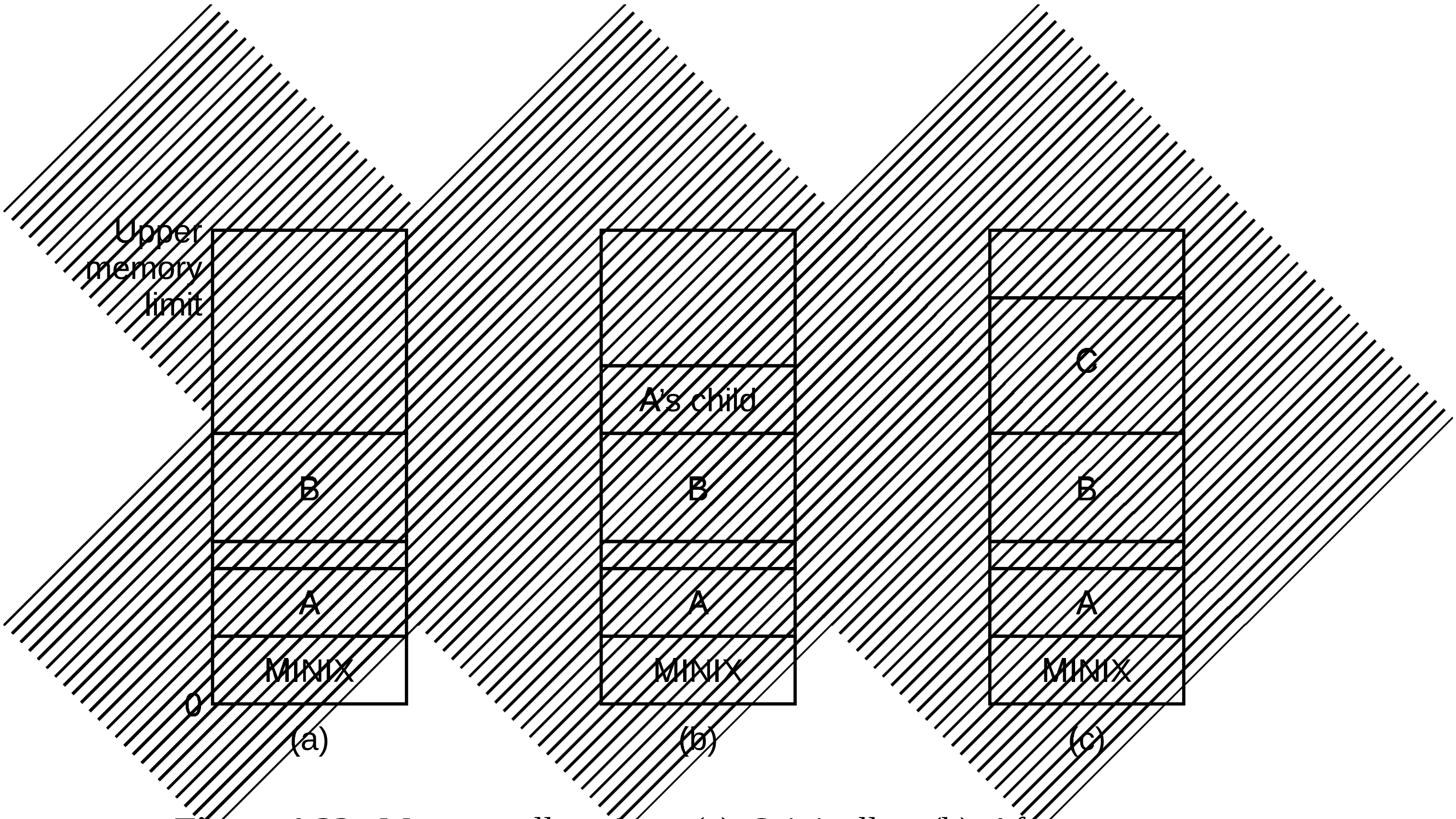


Figure 4-32. Memory allocation. (a) Originally. (b) After a SY FORK. (c) After the child does an EXEC . The shaded regions are unused memory. The process is a common I&D one.

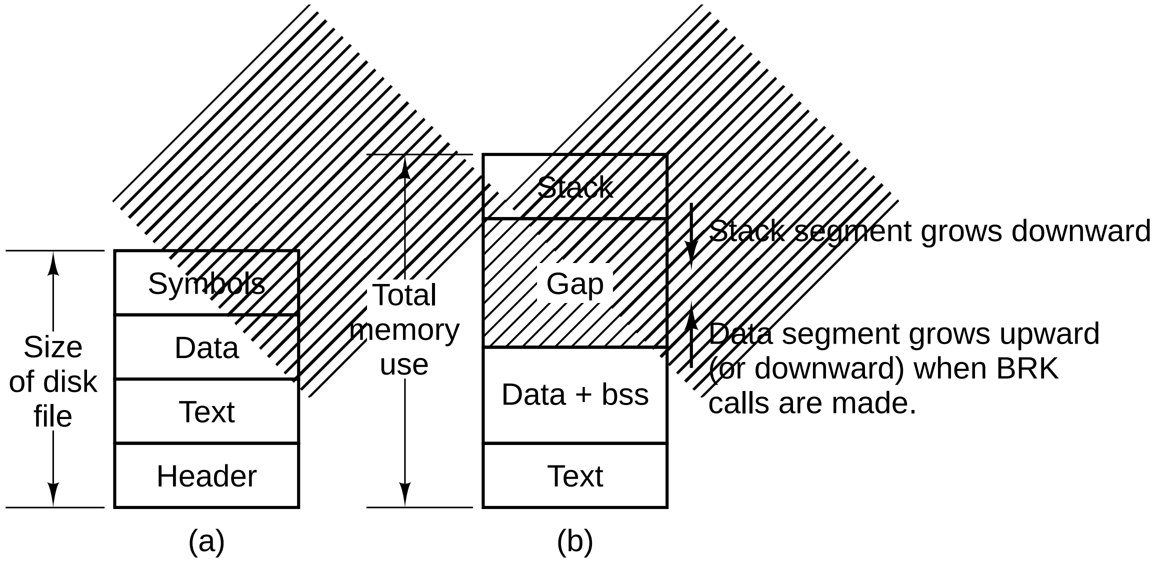
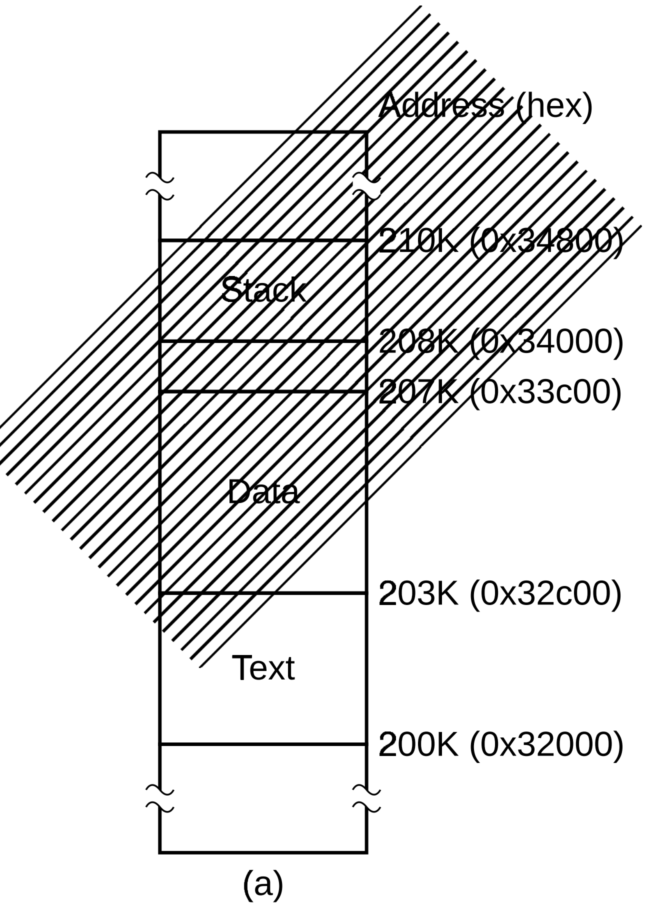


Figure 4-33. (a) A program as stored in a disk file. (b) Internal memory layout for a single process. In both parts of the figure the lowest disk or memory address is at the bottom and the highest address is at the top.

Message type	Input parameters	Reply value
FORK	(none)	Child's pid, (to child: 0)
EXIT	Exit status	(No reply if successful)
WAIT	(none)	Status
WAITPID	(none)	Status
BRK	New size	New size
EXEC	Pointer to initial stack	(No reply if successful)
KILL	Process identifier and signal	Status
ALARM	Number of seconds to wait	Residual time
PAUSE	(none)	(No reply if successful)
SIGACTION	Sig. number, action, old action	Status
SIGSUSPEND	Signal mask	(No reply if successful)
SIGPENDING	(none)	Status
SIGMASK	How, set, old set	Status
SIGRETURN	Context	Status
GETUID	(none)	Uid, effective uid
GETGID	(none)	Gid, effective gid
GETPID	(none)	Pid, parent pid
SETUID	New uid	Status
SETGID	New gid	Status
SETSID	New sid	Process group
GETPGRP	New gid	Process group
PTRACE	Request, pid, address, data	Status
REBOOT	How (halt, reboot, or panic)	(No reply if successful)
KSIG	Process slot and signals	(No reply)

Figure 4-34. The message types, input parameters, and reply values used for communicating with the memory manager.



	Virtual	Physical	Length
Stack	0x20	0x340	0x8
Data	0	0x320	0x1c
Text	0	0x320	0

(b)

	Virtual	Physical	Length
Stack	0x14	0x340	0x8
Data	0	0x32c	0x10
Text	0	0x320	0xc

(c)

Figure 4-35. (a) A process in memory. (b) Its memory representation for nonseparate I and D space. (c) Its memory representation for separate I and D space.

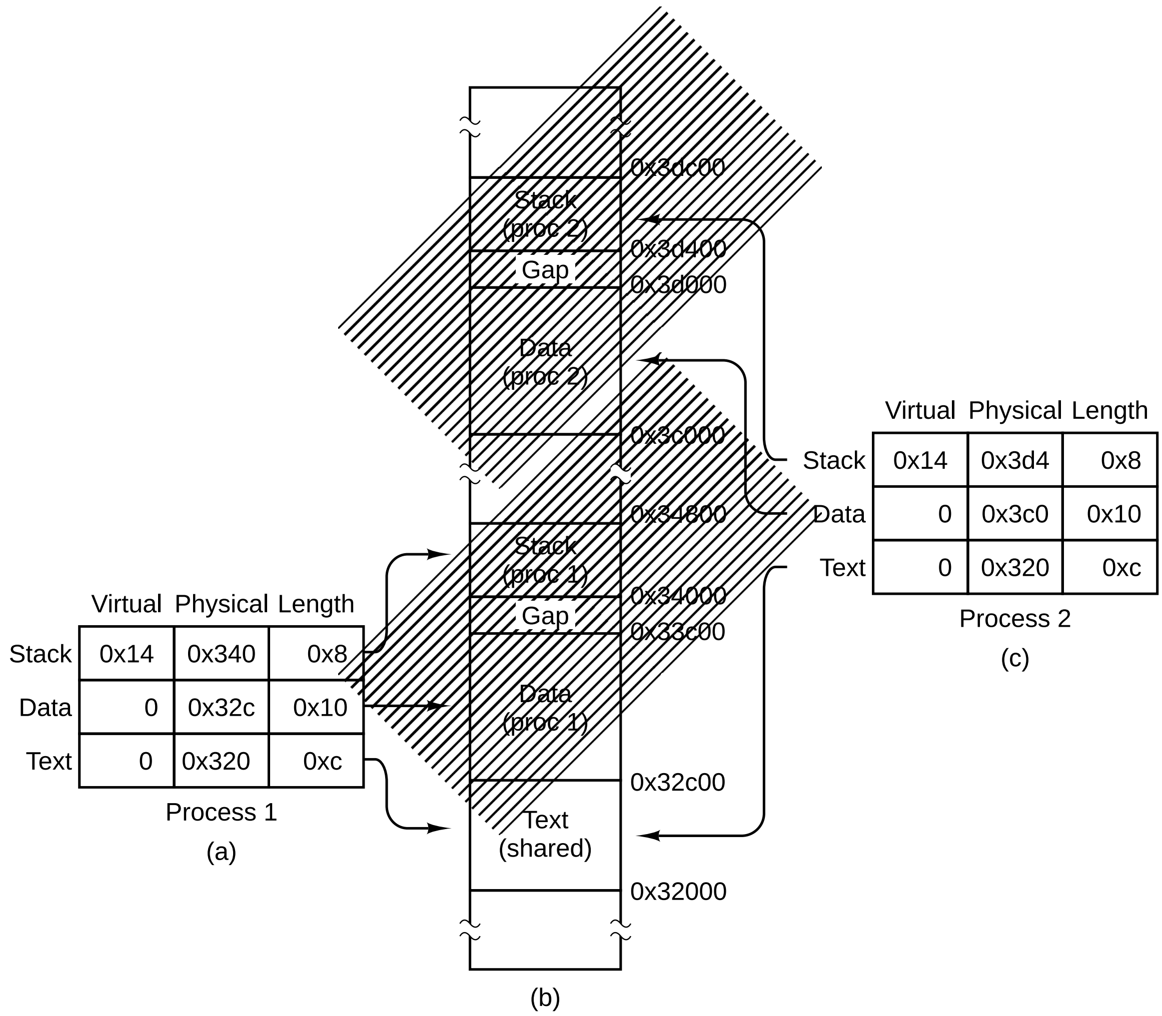


Figure 4-36. (a) The memory map of a separate I and D space process, as in the previous figure. (b) The layout in memory after a second process starts, executing the same program image with shared text. (c) The memory map of the second process.

1. Check to see if process table is full.
2. Try to allocate memory for the child’s data and stack.
3. Copy the parent’s data and stack to the child’s memory.
4. Find a free process slot and copy parent’s slot to it.
5. Enter child’s memory map in process table.
6. Choose a pid for the child.
7. Tell kernel and file system about child.
8. Report child’s memory map to kernel.
9. Send reply messages to parent and child.

Figure 4-37. The steps required to carry out the FORK system call.

1. Check permissions—is the file executable?
2. Read the header to get the segment and total sizes.
3. Fetch the arguments and environment from the caller.
4. Allocate new memory and release unneeded old memory.
5. Copy stack to new memory image.
6. Copy data (and possibly text) segment to new memory image.
7. Check for and handle setuid, setgid bits.
8. Fix up process table entry.
9. Tell kernel that process is now runnable.

Figure 4-38. The steps required to carry out the EXEC system call.

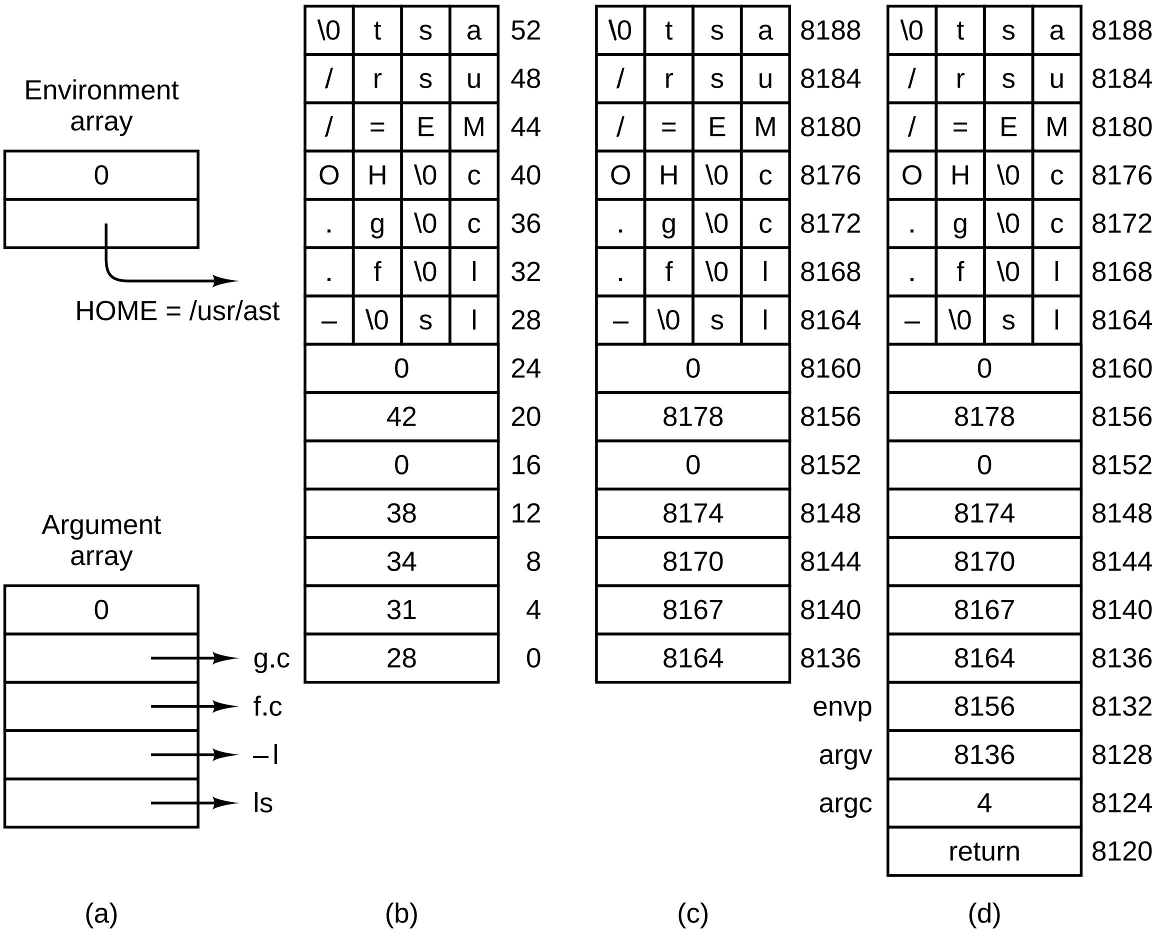


Figure 4-39. (a) The arrays passed to *execve*. (b) The stack built by *execve*. (c) The stack after relocation by the memory manager. (d) The stack as it appears to *main* at the start of execution.

```
push            ecx! push environ
push            edx! push argv
push            eax! push argc
call  _main     ! main(argc, argv, envp)
push            eax! push exit status
call  _exit
hlt             ! force a trap if exit fails
```

Figure 4-40. The key part of the C run-time, start-off routine.

Signal	Description	Generated by
SIGHUP	Hangup	KILL system call
SIGINT	Interrupt	Kernel
SIGQUIT	Quit	Kernel
SIGILL	Illegal instruction	Kernel (*)
SIGTRAP	Trace trap	Kernel (M)
SIGABRT	Abnormal termination	Kernel
SIGFPE	Floating point exception	Kernel (*)
SIGKILL	Kill (cannot be caught or ignored)	KILL system call
SIGUSR1	User-defined signal # 1	Not supported
SIGSEGV	Segmentation violation	Kernel (*)
SIGUSR2	User defined signal # 2	Not supported
SIGPIPE	Write on a pipe with no one to read it	Kernel
SIGALRM	Alarm clock, timeout	Kernel
SIGTERM	Software termination signal from kill	KILL system call
SIGCHLD	Child process terminated or stopped	Not supported
SIGCONT	Continue if stopped	Not supported
SIGSTOP	Stop signal	Not supported
SIGTSTP	Interactive stop signal	Not supported
SIGTTIN	Background process wants to read	Not supported
SIGTTOU	Background process wants to write	Not supported

Figure 4-41. Signals defined by POSIX and MINIX. Signals indicated by (*) depend upon hardware support. Signals marked (M) are not defined by POSIX, but are defined by MINIX for compatibility with older programs. Several obsolete names and synonyms are not listed here.

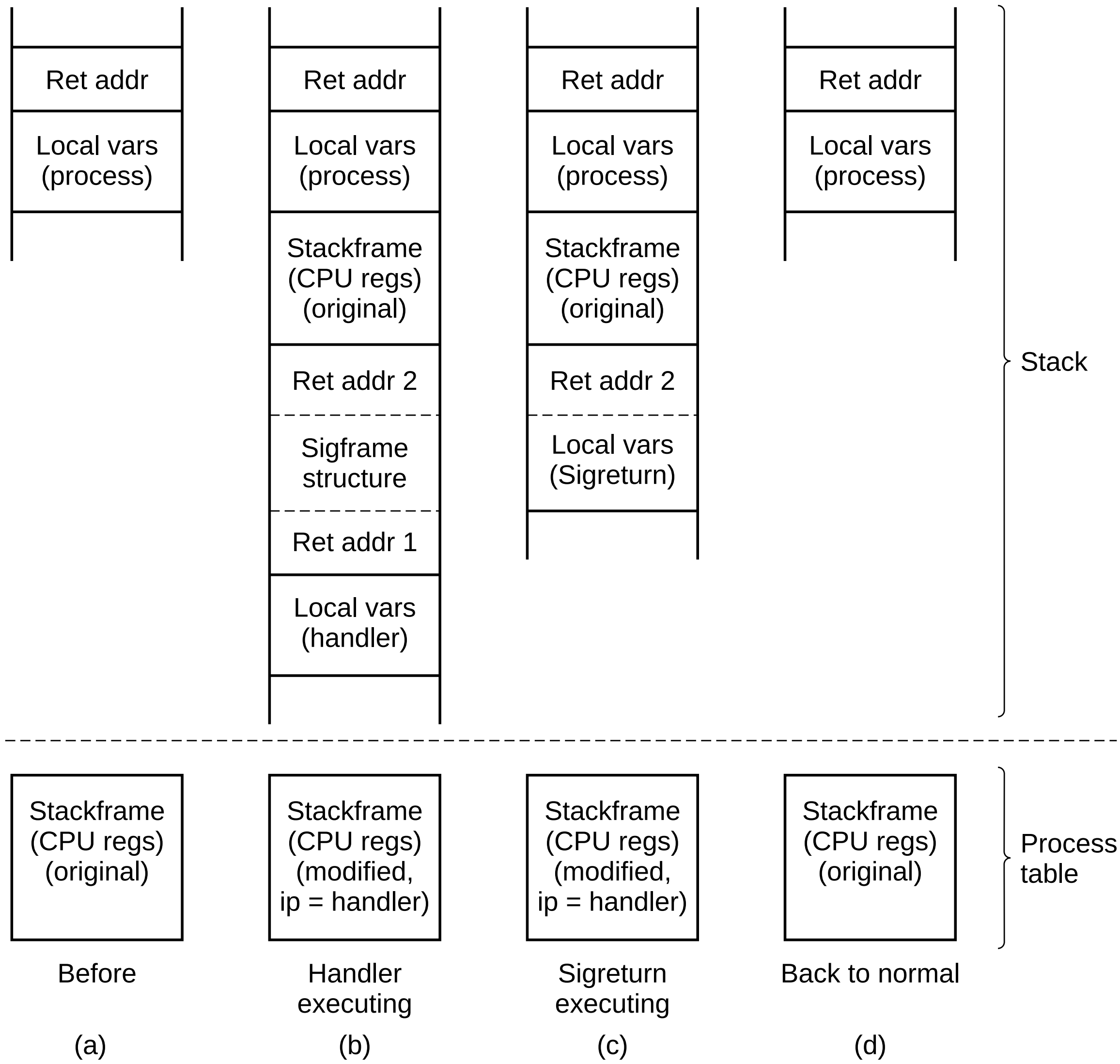


Figure 4-42. A process' stack (above) and its stackframe in the process table (below) corresponding to phases in handling a signal. (a) State as process is taken out of execution. (b) State as handler begins execution. (c) State while SIGRETURN is executing. (d) State after SIGRETURN completes execution.

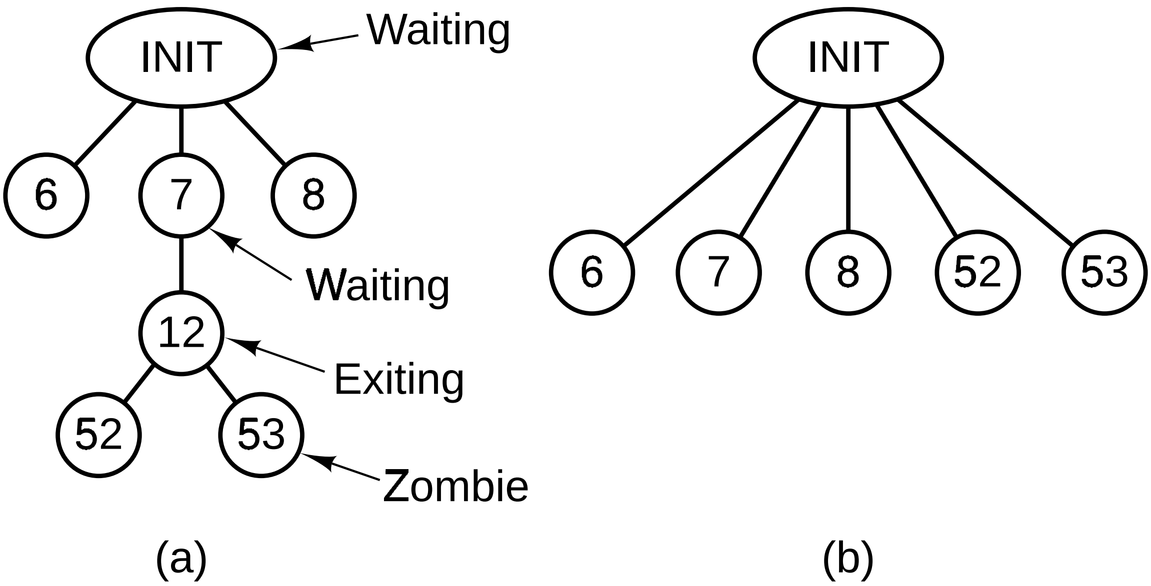


Figure 4-43. (a) The situation as process 12 is about to exit. (b) The situation after it has exited.

System call	Purpose
SIGACTION	Modify response to future signal
SIGPROCMASK	Change set of blocked signals
KILL	Send signal to another process
ALARM	Send ALRM signal to self after delay
PAUSE	Suspend self until future signal
SIGSUSPEND	Change set of blocked signals, then PAUSE
SIGPENDING	Examine set of pending (blocked) signals
SIGRETURN	Clean up after signal handler

Figure 4-44. System calls relating to signals.

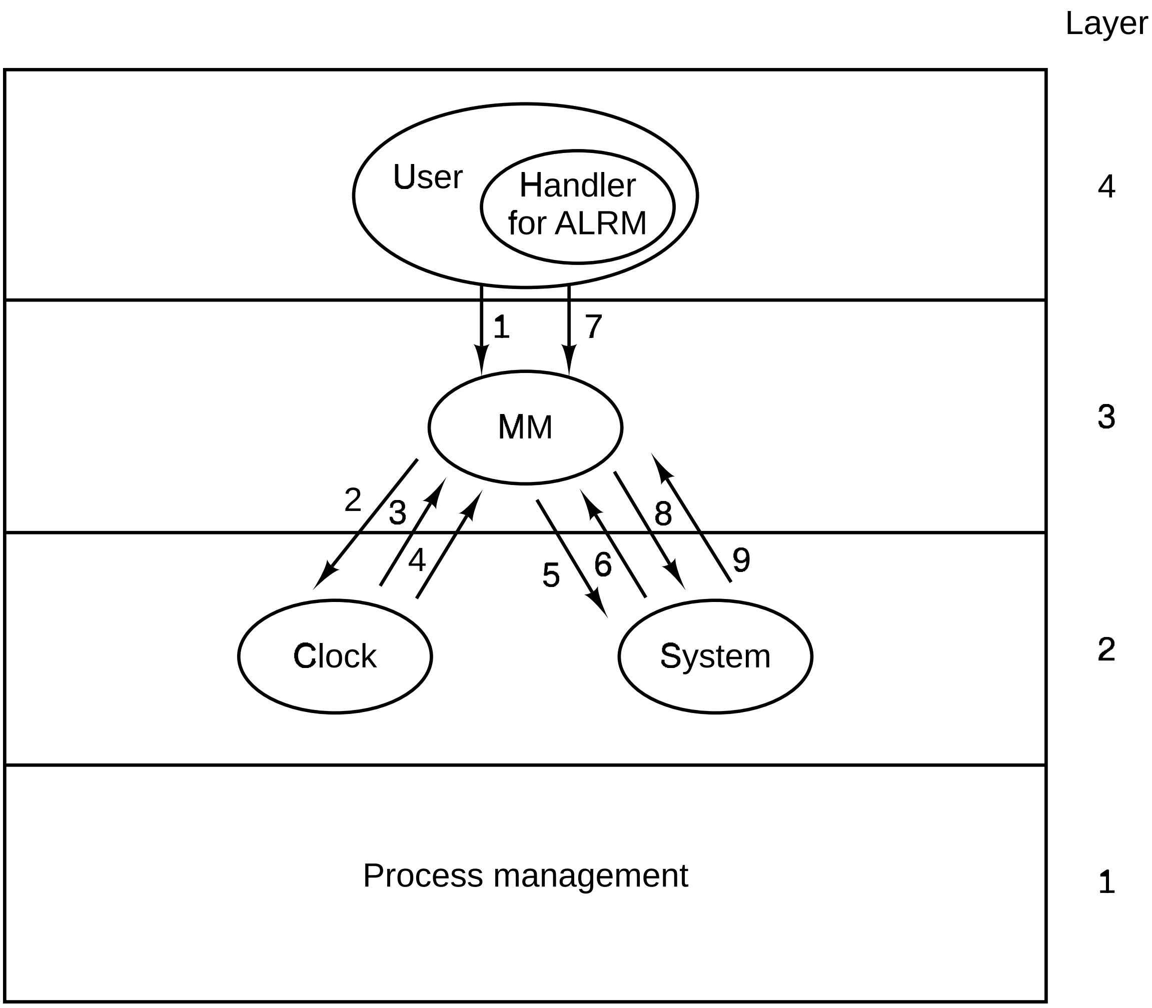


Figure 4-45. Messages for an alarm. The most important are: (1) User does ALARM. (4) After the set time has elapsed, the signal arrives. (7) Handler terminates with call to SIGRETURN. See text for details.

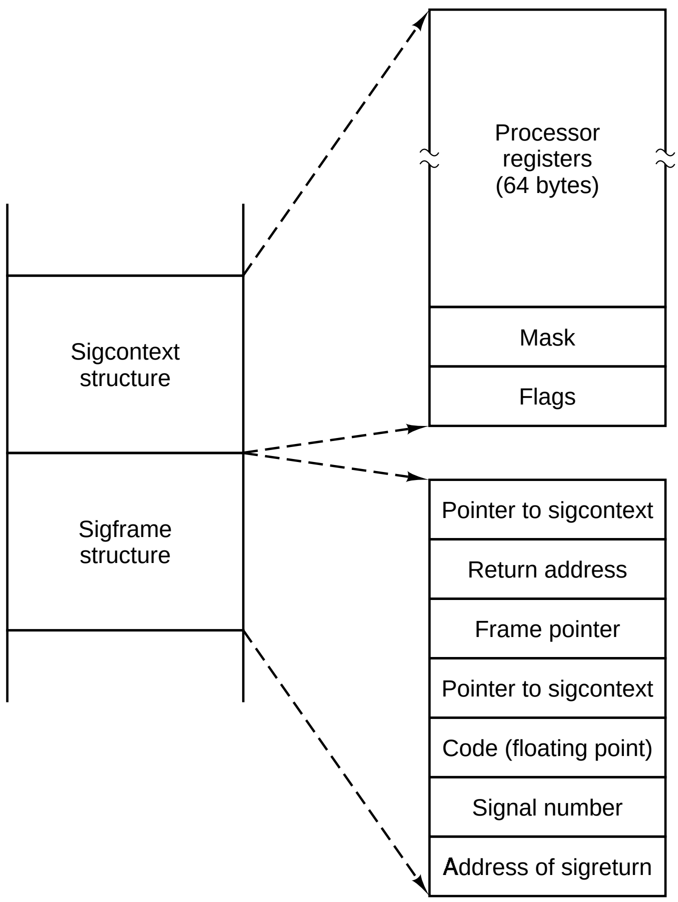


Figure 4-46. The sigcontext and sigframe structures pushed on the stack to prepare for a signal handler. The processor registers are a copy of the stackframe used during a context switch.

System Call	Description
GETUID	Return real and effective uid
GETGID	Return real and effective gid
GETPID	Return pids of process and its parent
SETUID	Set caller’s real and effective uid
SETGID	Set caller’s real and effective gid
SETSID	Create new session, return pid
GETPGRP	Return ID of process group

Figure 4-47. The system calls supported in *mm/getset.c*.

Command	Description
T_STOP	Stop the process
T_OK	Enable tracing by parent for this process
T_GETINS	Return value from text (instruction) space
T_GETDATA	Return value from data space
T_GETUSER	Return value from user process table
T_SETINS	Set value in instruction space
T_SETDATA	Set value in data space
T_SETUSER	Set value in user process table
T_RESUME	Resume execution
T_EXIT	Exit
T_STEP	Set trace bit

Figure 4-48. Debugging commands supported by *mm/trace.c*.

5

FILE SYSTEMS

Extension	Meaning
file.bak	Backup file
file.c	C source program
file.f77	Fortran 77 program
file.gif	Compuserve Graphical Interchange Format image
file.hlp	Help file
file.html	World Wide Web HyperText Markup Language document
file.mpg	Movie encoded with the MPEG standard
file.o	Object file (compiler output, not yet linked)
file.ps	PostScript file
file.tex	Input for the TEX formatting program
file.txt	General text file
file.zip	Compressed archive

Figure 5-1. Some typical file extensions.

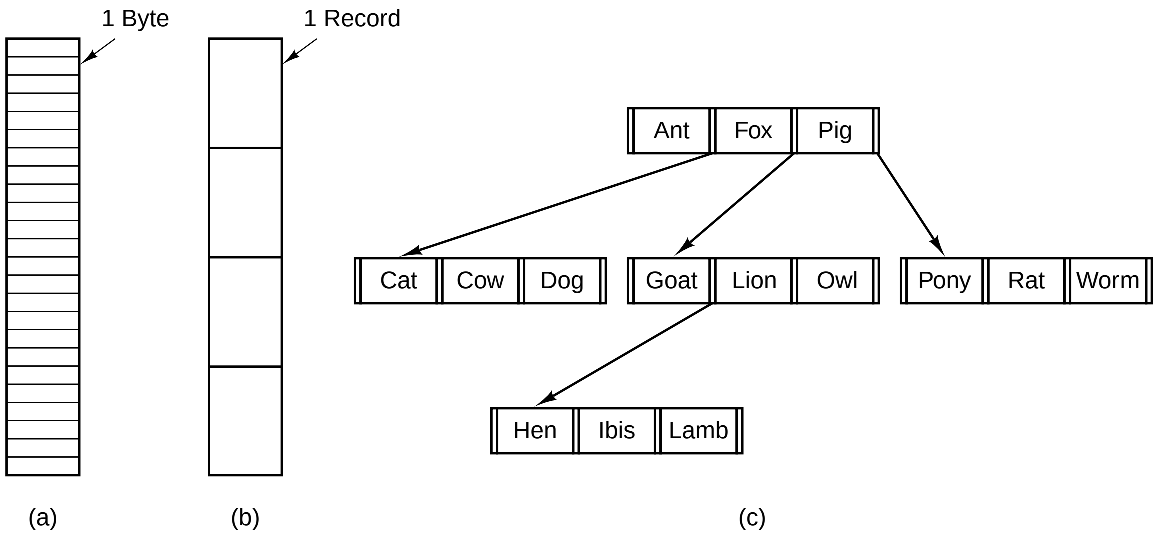


Figure 5-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

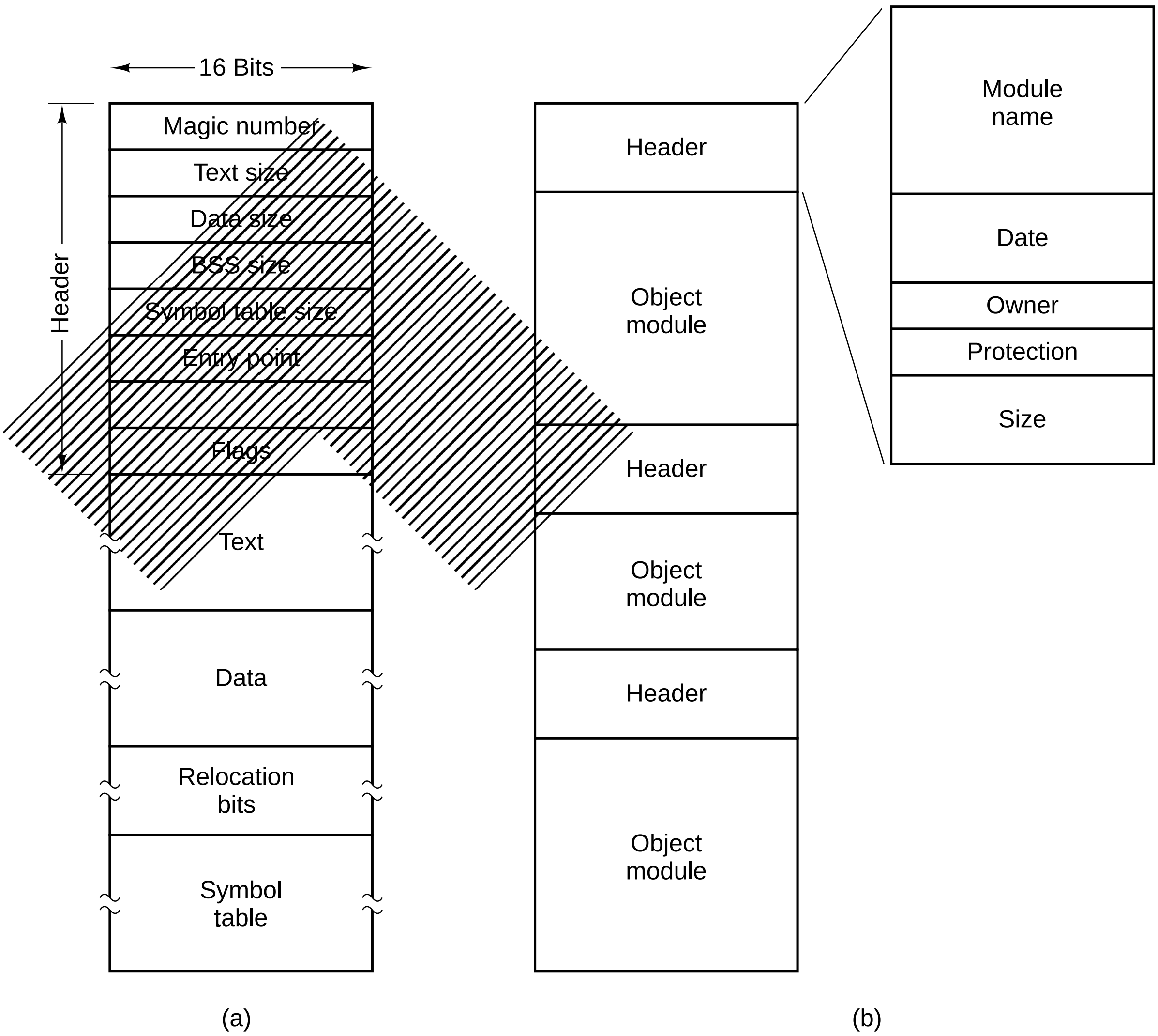


Figure 5-3. (a) An executable file. (b) An archive.

Field	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	Id of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Figure 5-4. Some possible file attributes.

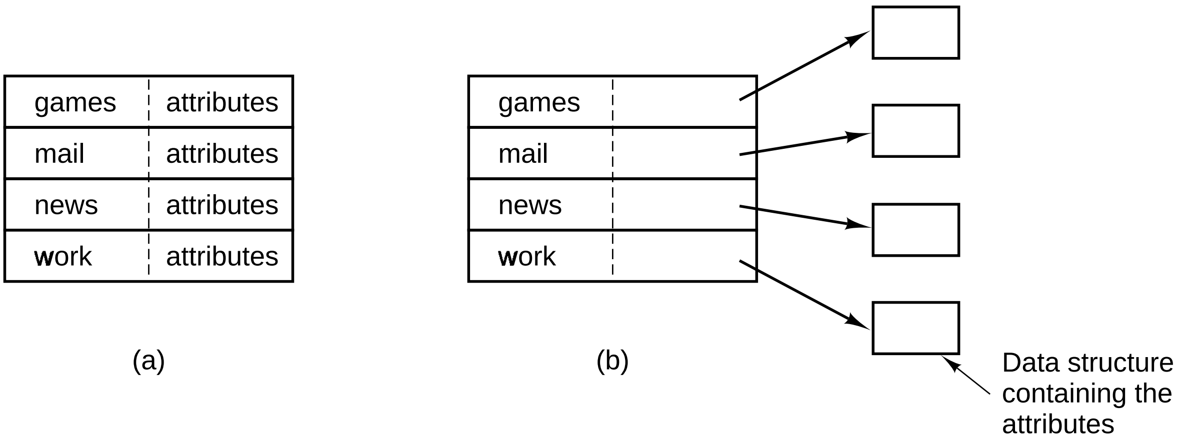


Figure 5-5. (a) Attributes in the directory entry. (b) Attributes elsewhere.

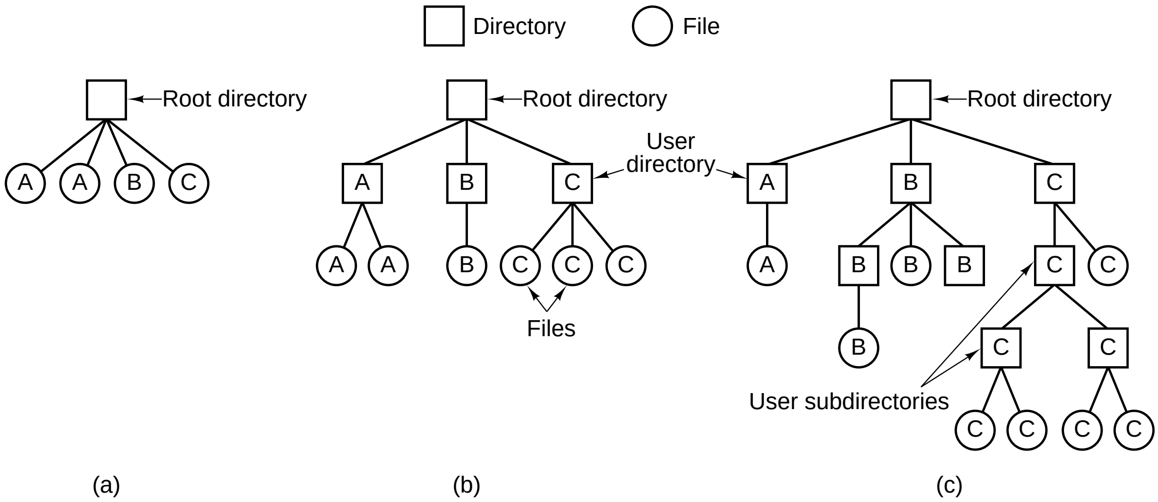


Figure 5-6. Three file system designs. (a) Single directory shared by all users. (b) One directory per user. (c) Arbitrary tree per user. The letters indicate the directory or file's owner.

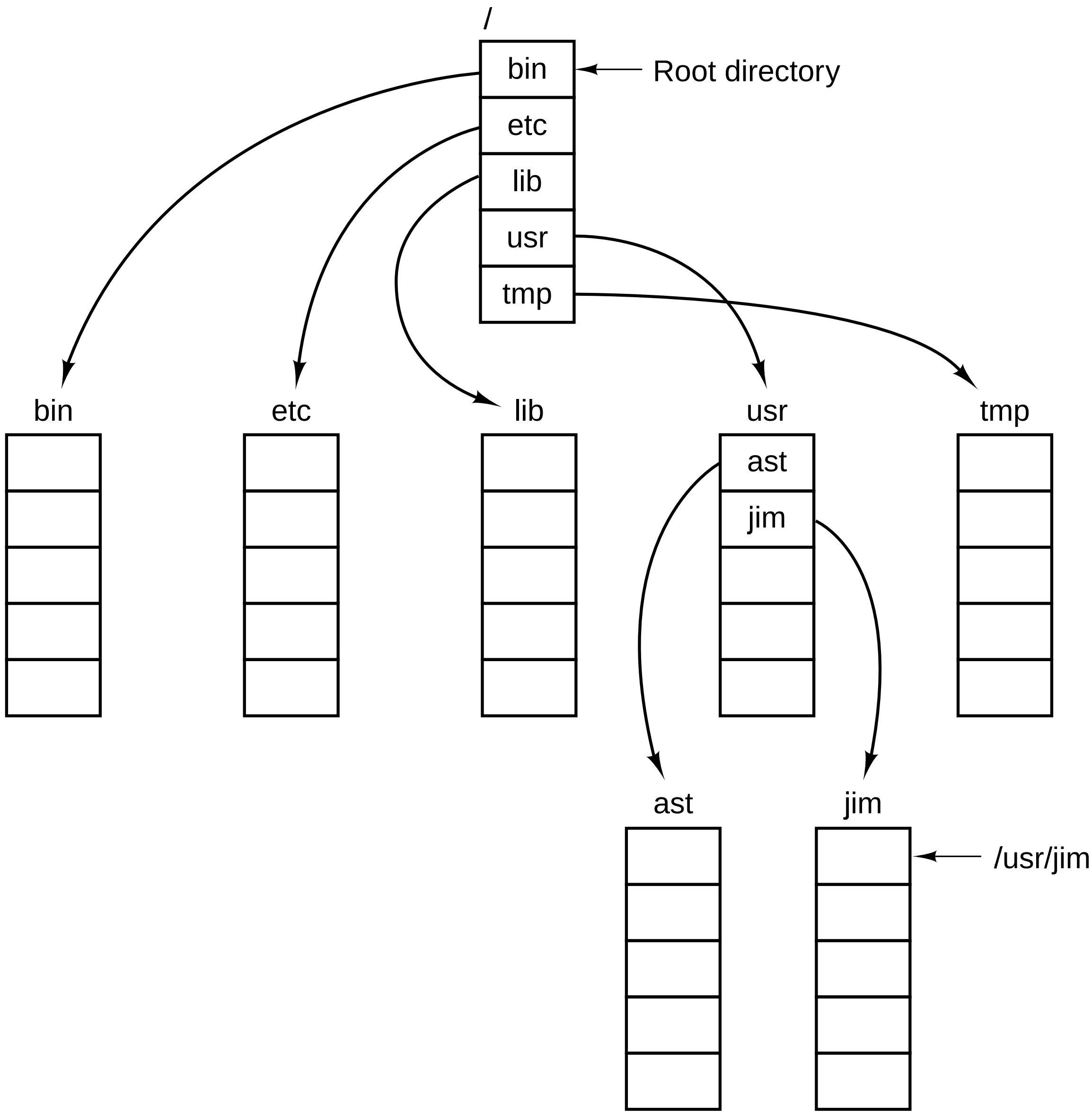


Figure 5-7. A UNIX directory tree.

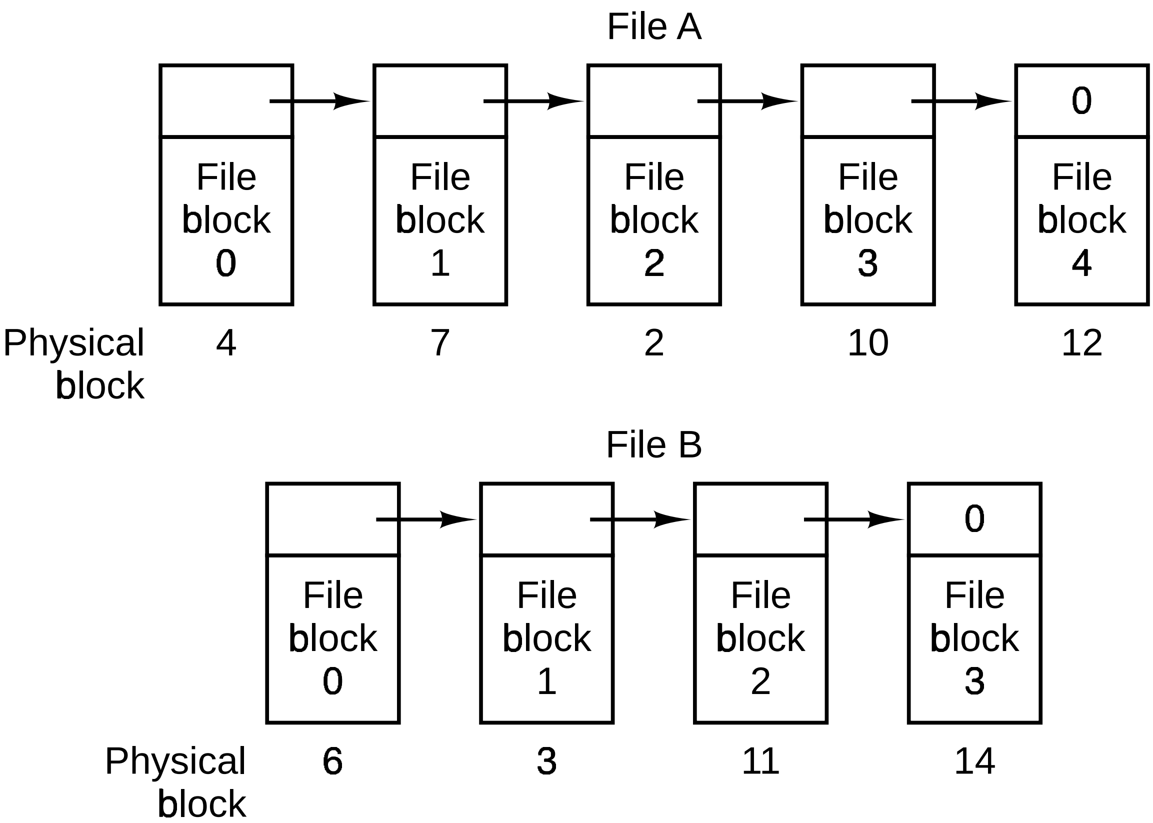


Figure 5-8. Storing a file as a linked list of disk blocks.

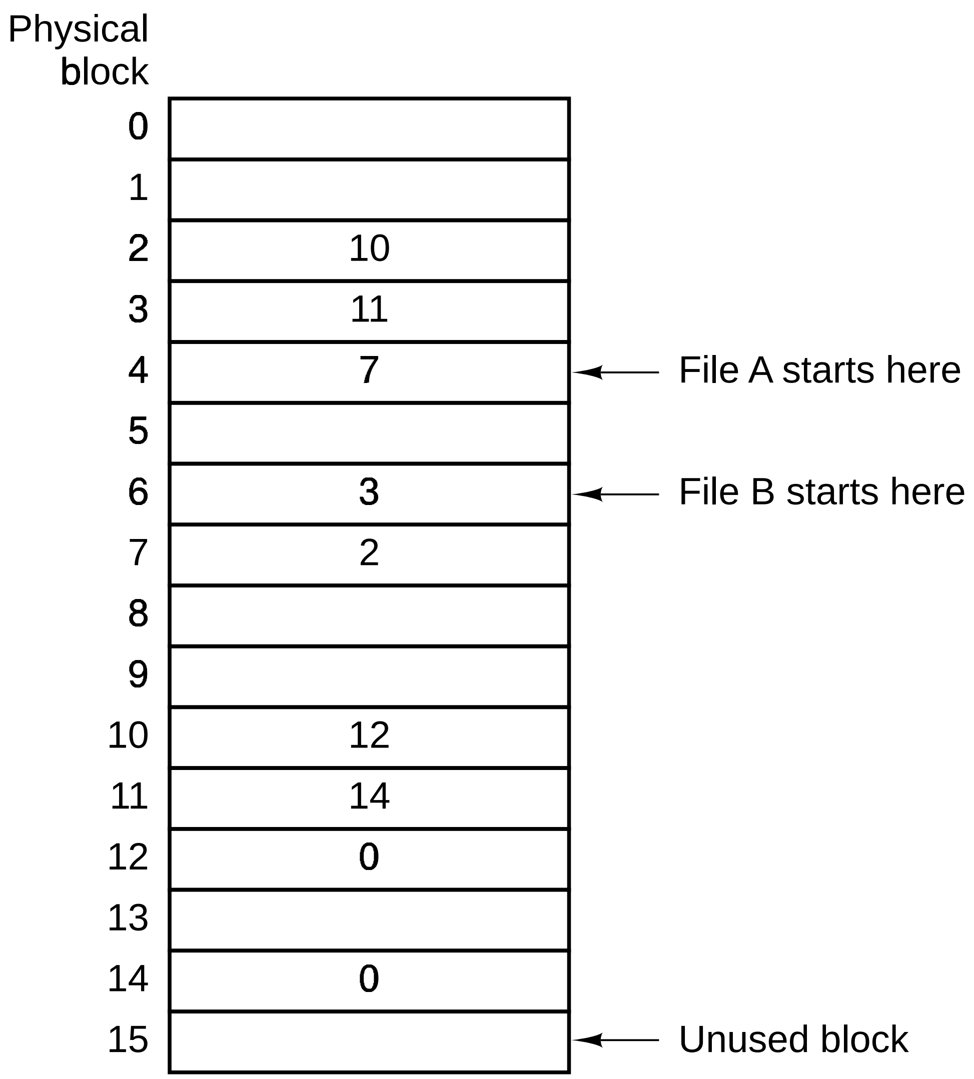


Figure 5-9. Linked list allocation using a table in main memory.

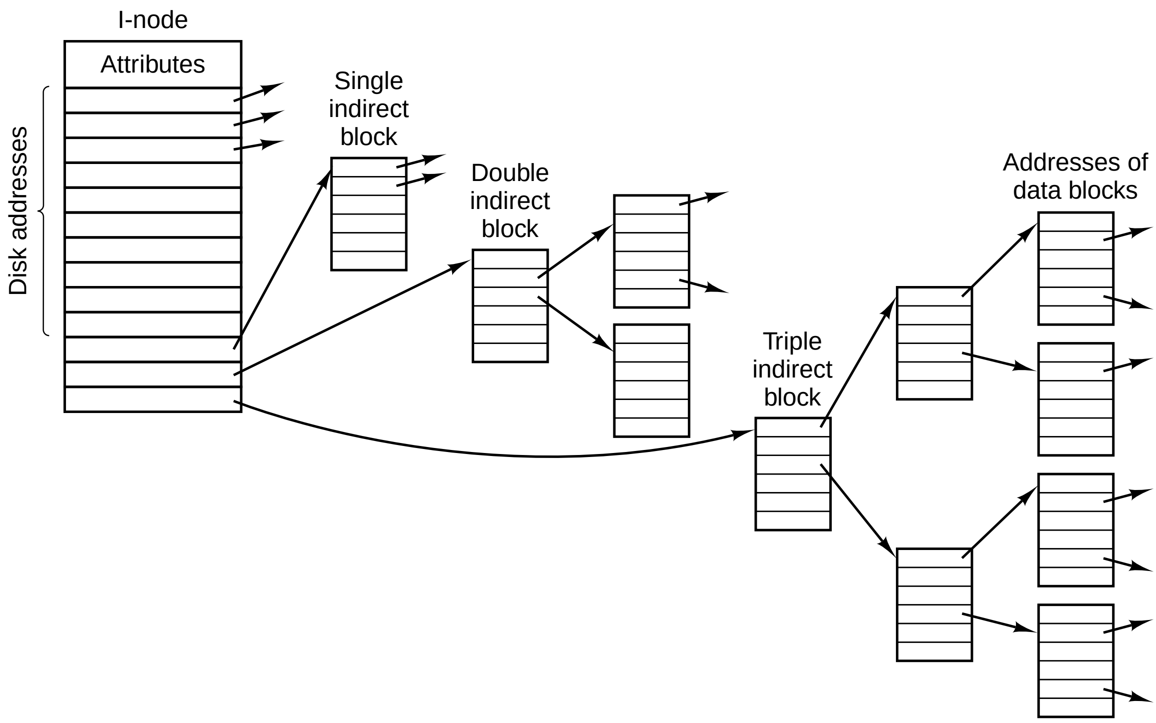


Figure 5-10. An i-node.

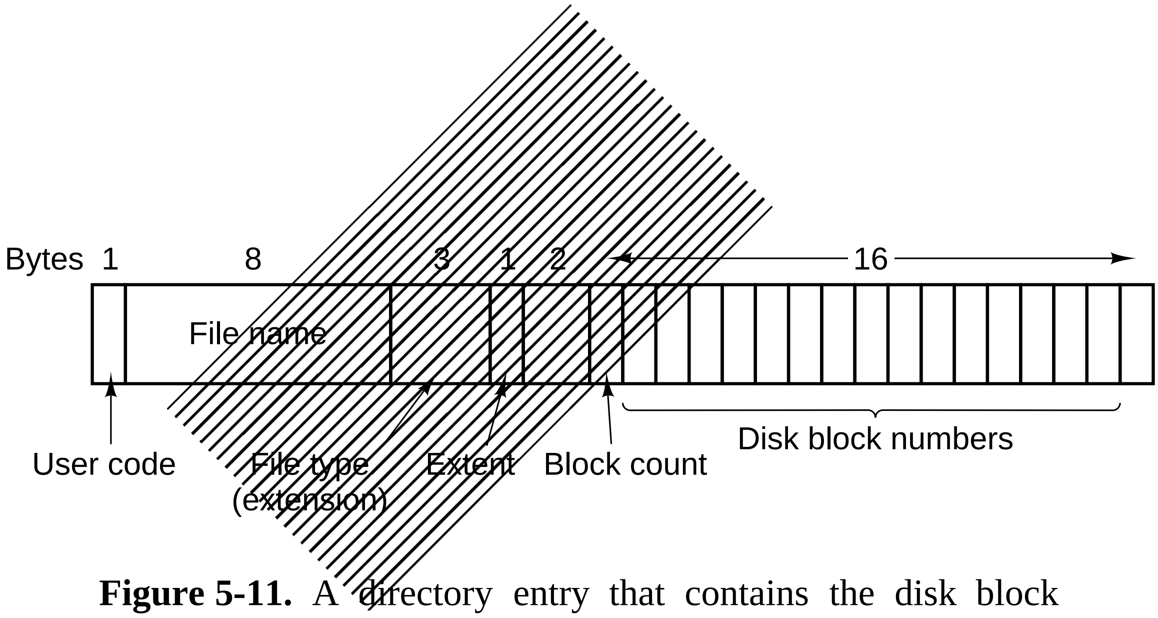


Figure 5-11. A directory entry that contains the disk block numbers for each file.

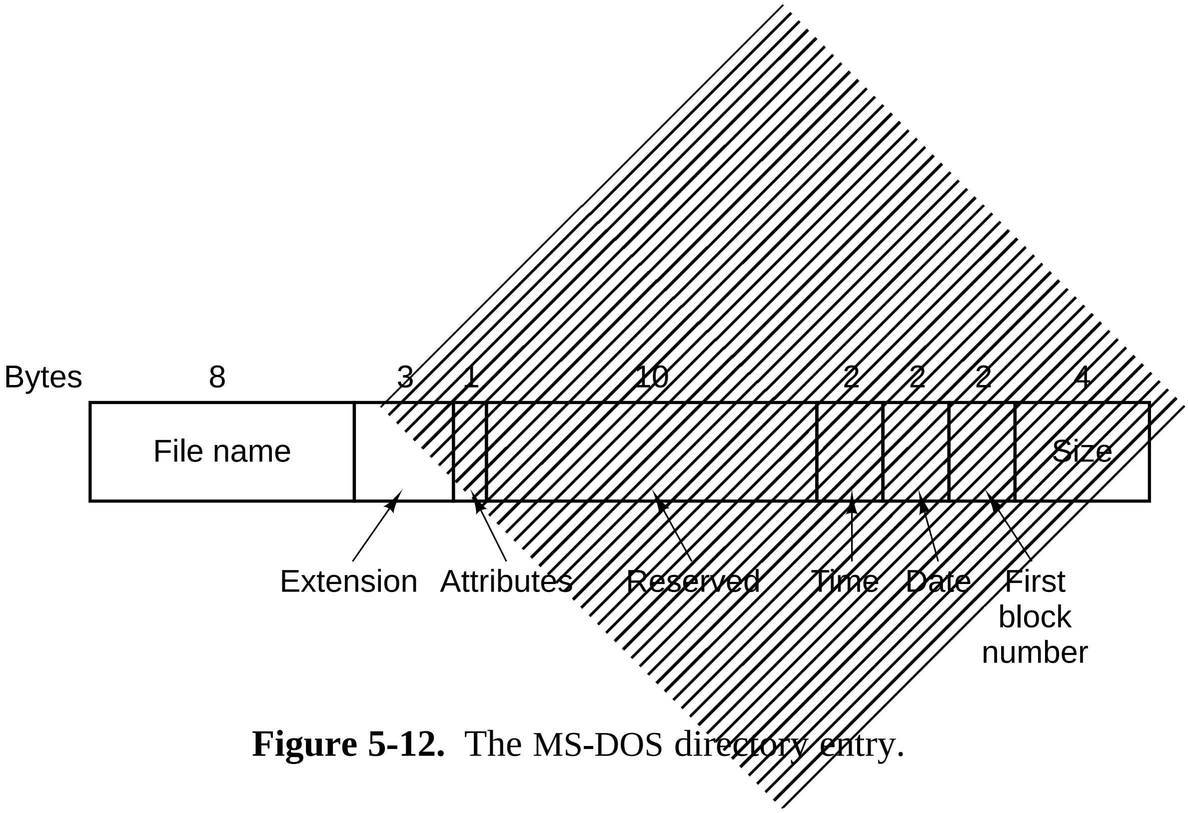


Figure 5-12. The MS-DOS directory entry.

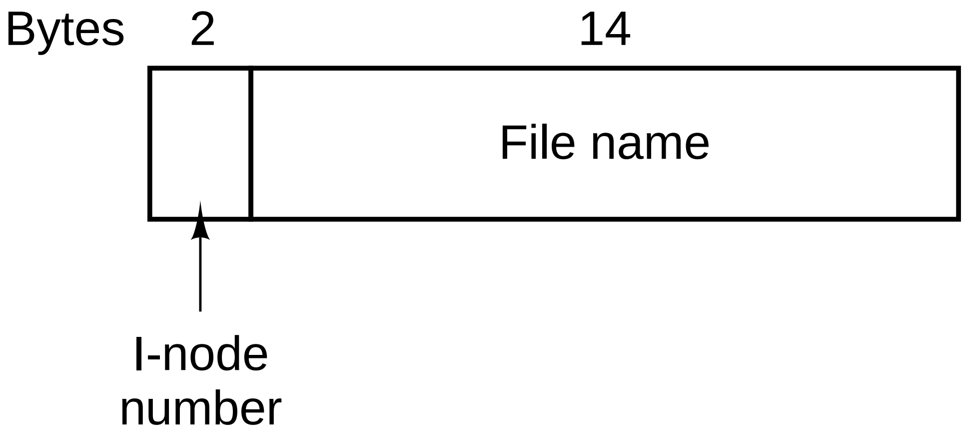


Figure 5-13. A UNIX directory entry.

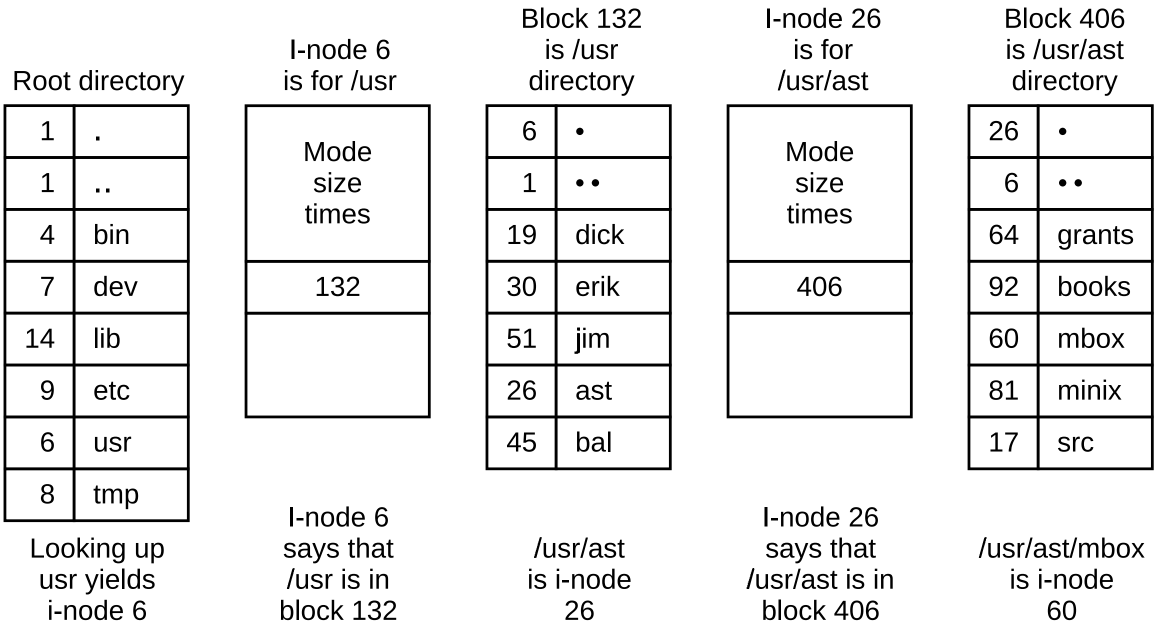


Figure 5-14. The steps in looking up */usr/ast/mbox*.

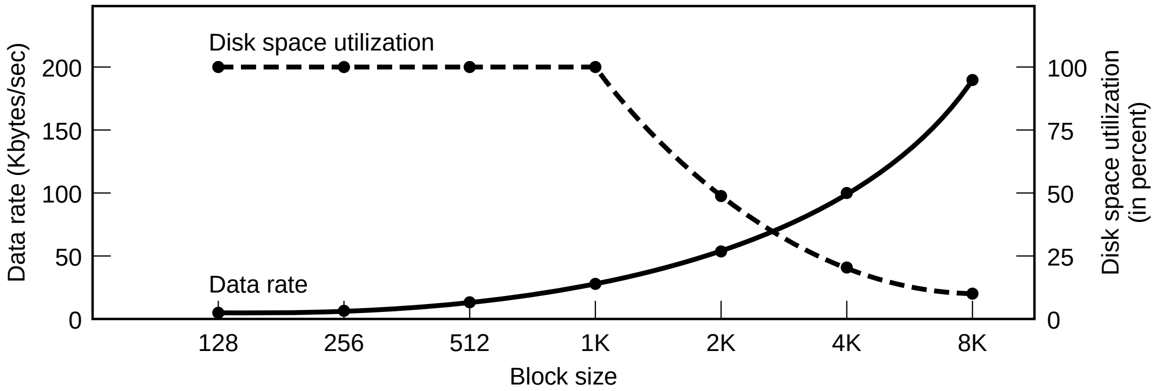


Figure 5-15. The solid curve (left-hand scale) gives the data rate of a disk. The dashed curve (right-hand scale) gives the disk space efficiency. All files are 1K.

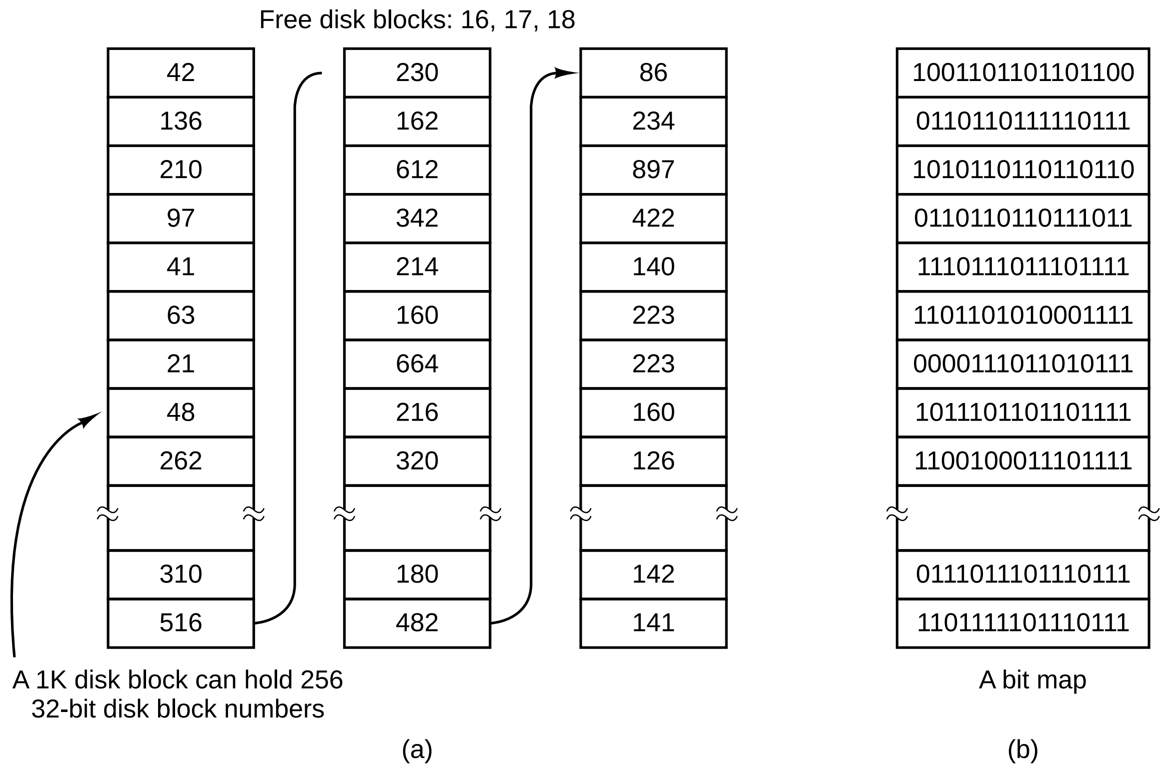


Figure 5-16. (a) Storing the free list on a linked list. (b) A bit map.

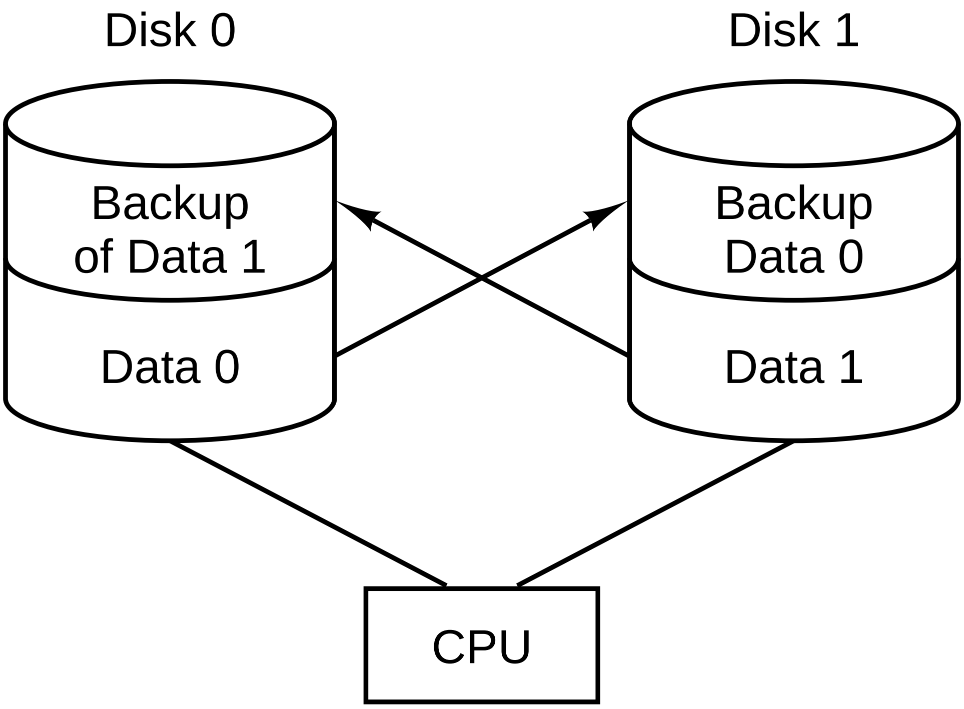


Figure 5-17. Backing up each drive on the other one wastes half the storage.

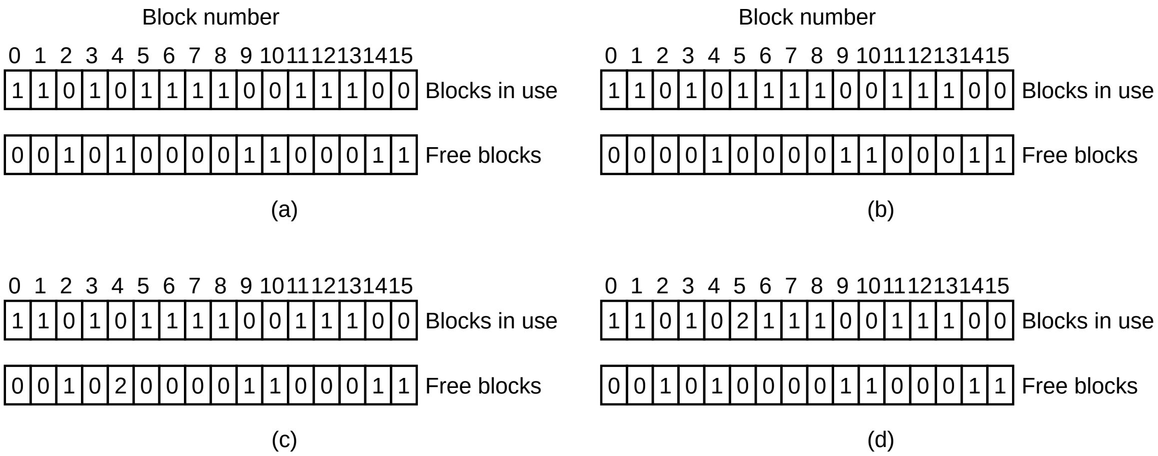


Figure 5-18. File system states. (a) Consistent. (b) Missing block. (c) Duplicate block in free list. (d) Duplicate data block.

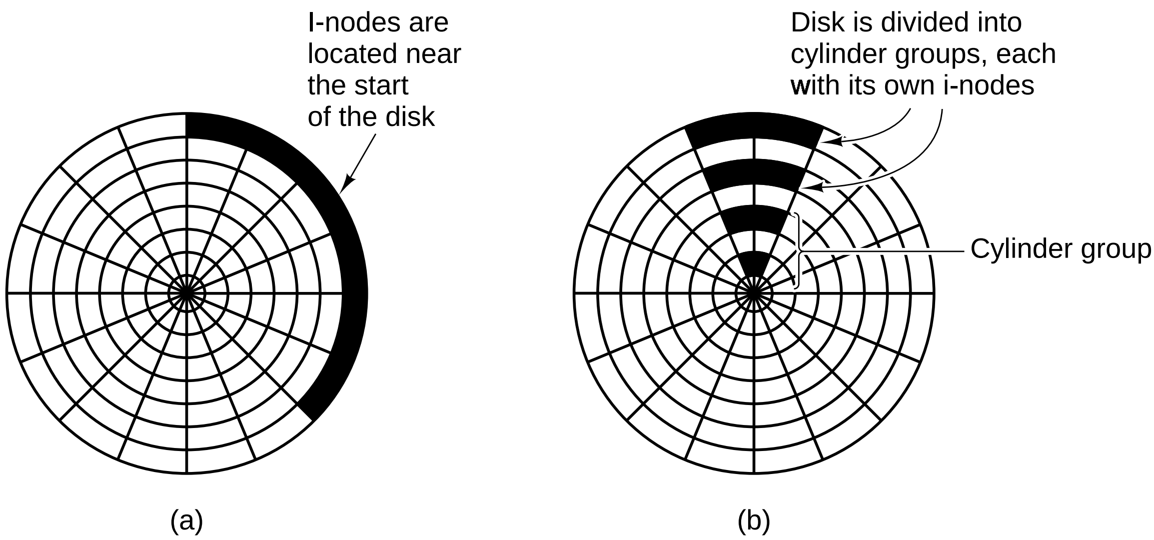


Figure 5-19. (a) I-nodes placed at the start of the disk. (b) Disk divided into cylinder groups, each with its own blocks and i-nodes.

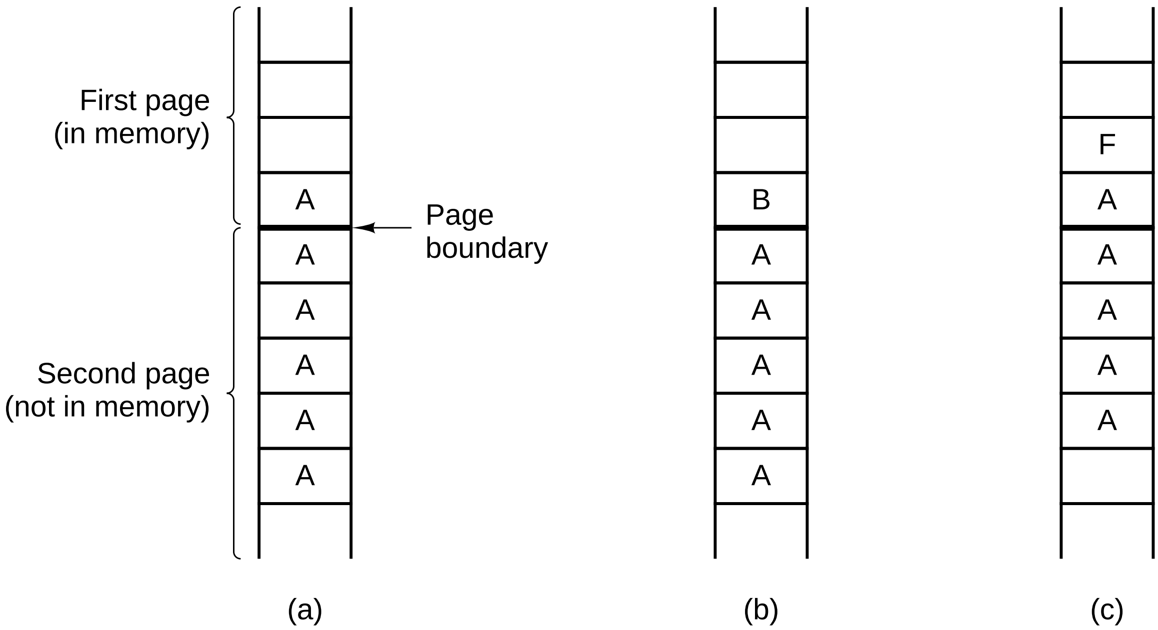


Figure 5-20. The TENEX password problem.

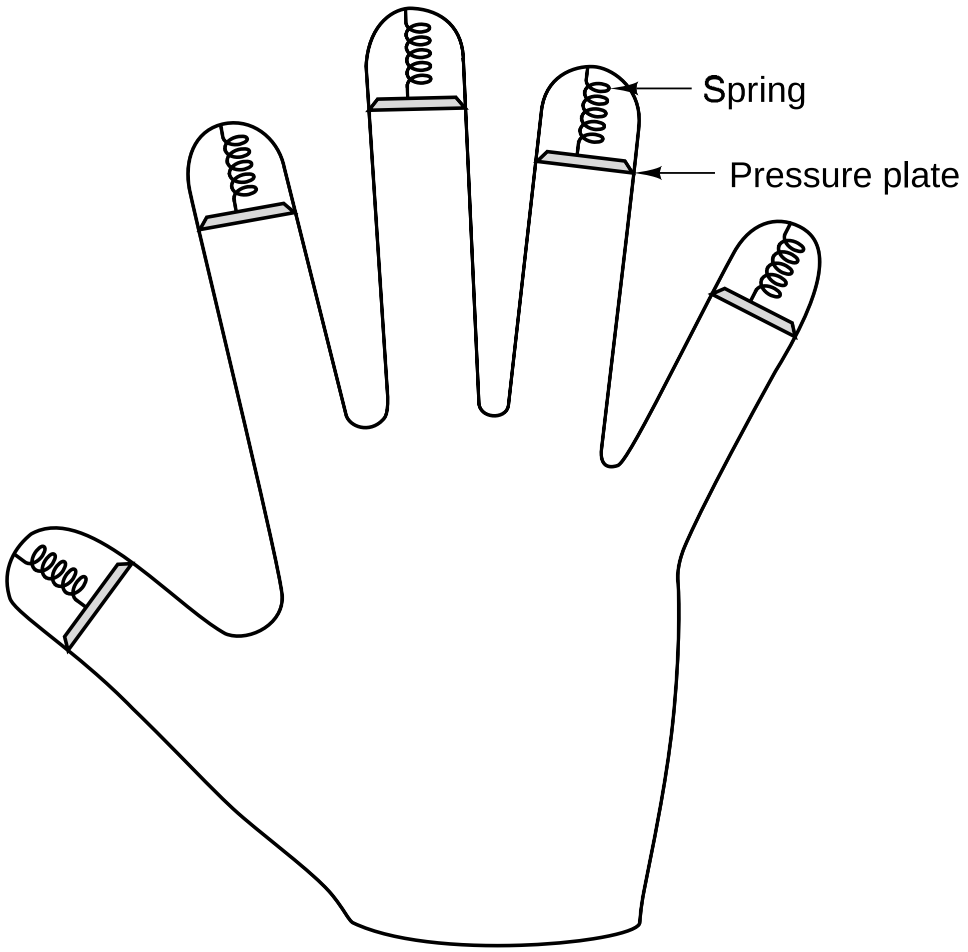


Figure 5-21. A device for measuring finger length.

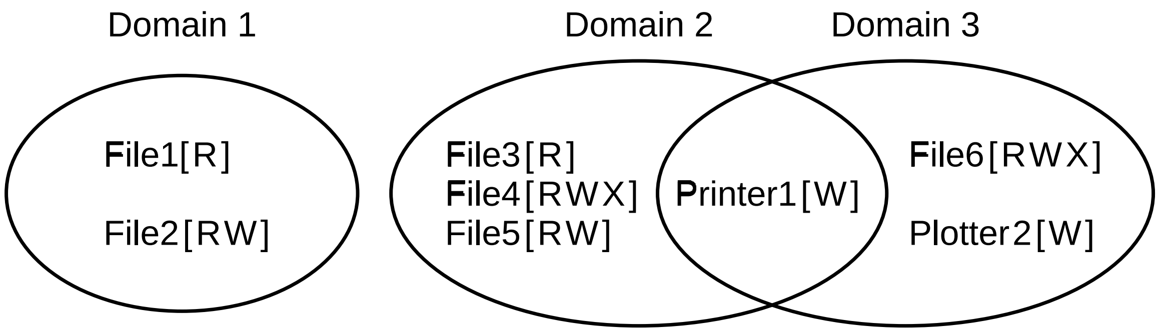


Figure 5-22. Three protection domains.

Domain \ Object	File1	File2	File3	File4	File5	File6	Printer1	Plotter2
1	Read	Read Write						
2			Read	Read Write Execute	Read Write		Write	
3						Read Write Execute	Write	Write

Figure 5-23. A protection matrix.

Domain	File1	File2	File3	File4	File5	File6	Printer1	Plotter2	Domain1	Domain2	Domain3
1	Read	Read Write								Enter	
2			Read	Read Write Execute	Read Write		Write				
3						Read Write Execute	Write	Write			

Figure 5-24. A protection matrix with domains as objects.

#	Type	Rights	Object
0	File	R – –	Pointer to File3
1	File	RWX	Pointer to File4
2	File	RW–	Pointer to File5
3	Pointer	–W–	Pointer to Printer1

Figure 5-25. The capability list for domain 2 in Fig. 5-23.

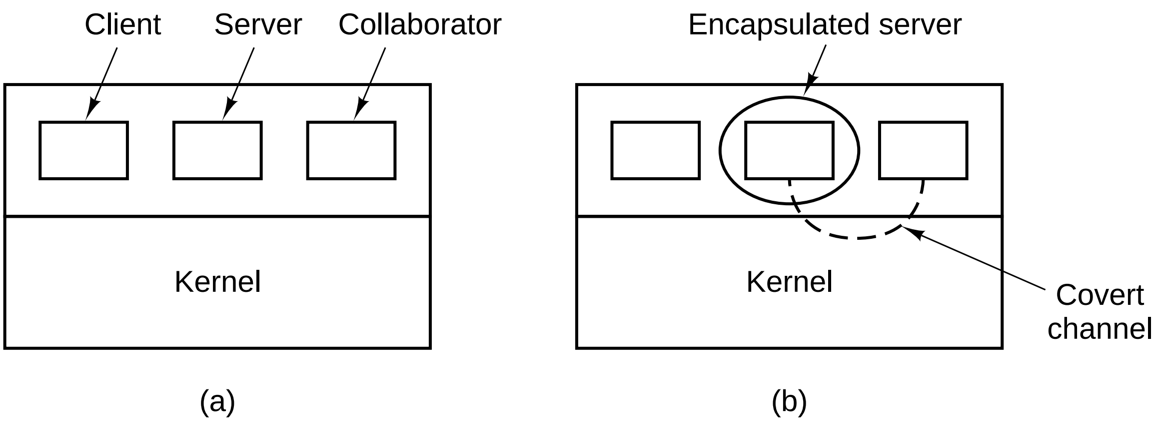


Figure 5-26. (a) The client, server, and collaborator processes. (b) The encapsulated server can still leak to the collaborator via covert channels.

Messages from users	Input parameters	Reply value
ACCESS	File name, access mode	Status
CHDIR	Name of new working directory	Status
CHMOD	File name, new mode	Status
CHOWN	File name, new owner, group	Status
CHROOT	Name of new root directory	Status
CLOSE	File descriptor of file to close	Status
CREAT	Name of file to be created, mode	File descriptor
DUP	File descriptor (for dup2, two fds)	New file descriptor
FCNTL	File descriptor, function code, arg	Depends on function
FSTAT	Name of file, buffer	Status
IOCTL	File descriptor, function code, arg	Status
LINK	Name of file to link to, name of link	Status
LSEEK	File descriptor, offset, whence	New position
MKDIR	File name, mode	Status
MKNOD	Name of dir or special, mode, address	Status
MOUNT	Special file, where to mount, ro flag	Status
OPEN	Name of file to open, r/w flag	File descriptor
PIPE	Pointer to 2 file descriptors (modified)	Status
READ	File descriptor, buffer, how many bytes	# Bytes read
RENAME	File name, file name	Status
RMDIR	File name	Status
STAT	File name, status buffer	Status
STIME	Pointer to current time	Status
SYNC	(None)	Always OK
TIME	Pointer to place where current time goes	Status
TIMES	Pointer to buffer for process and child times	Status
UMASK	Complement of mode mask	Always OK
UMOUNT	Name of special file to unmount	Status
UNLINK	Name of file to unlink	Status
UTIME	File name, file times	Always OK
WRITE	File descriptor, buffer, how many bytes	# Bytes written
Messages from MM	**Input parameters**	**Reply value**
EXEC	Pid	Status
EXIT	Pid	Status
FORK	Parent pid, child pid	Status
SETGID	Pid, real and effective gid	Status
SETSID	Pid	Status
SETUID	Pid, real and effective uid	Status
Other messages	**Input parameters**	**Reply value**
REVIVE	Process to revive	(No reply)
UNPAUSE	Process to check	(See text)

Figure 5-27. File system messages.

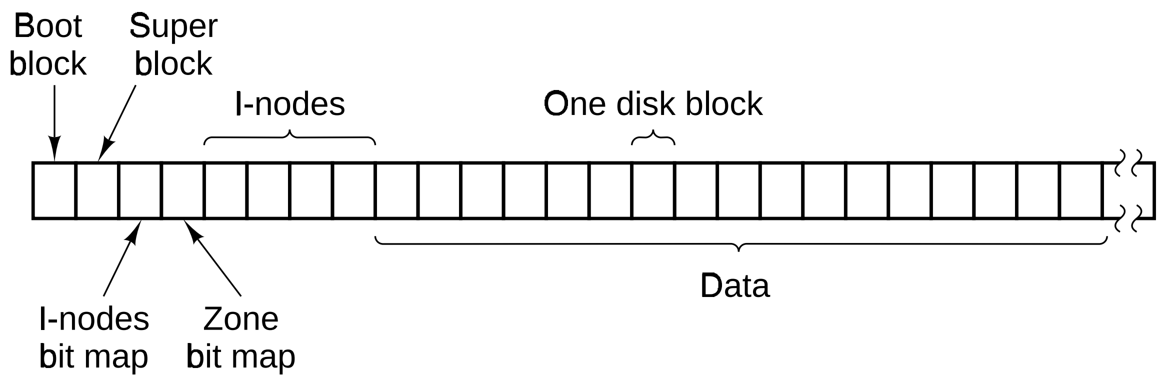


Figure 5-28. Disk layout for the simplest disk: a 360K floppy disk, with 128 i-nodes and a 1K block size (i.e., two consecutive 512-byte sectors are treated as a single block).

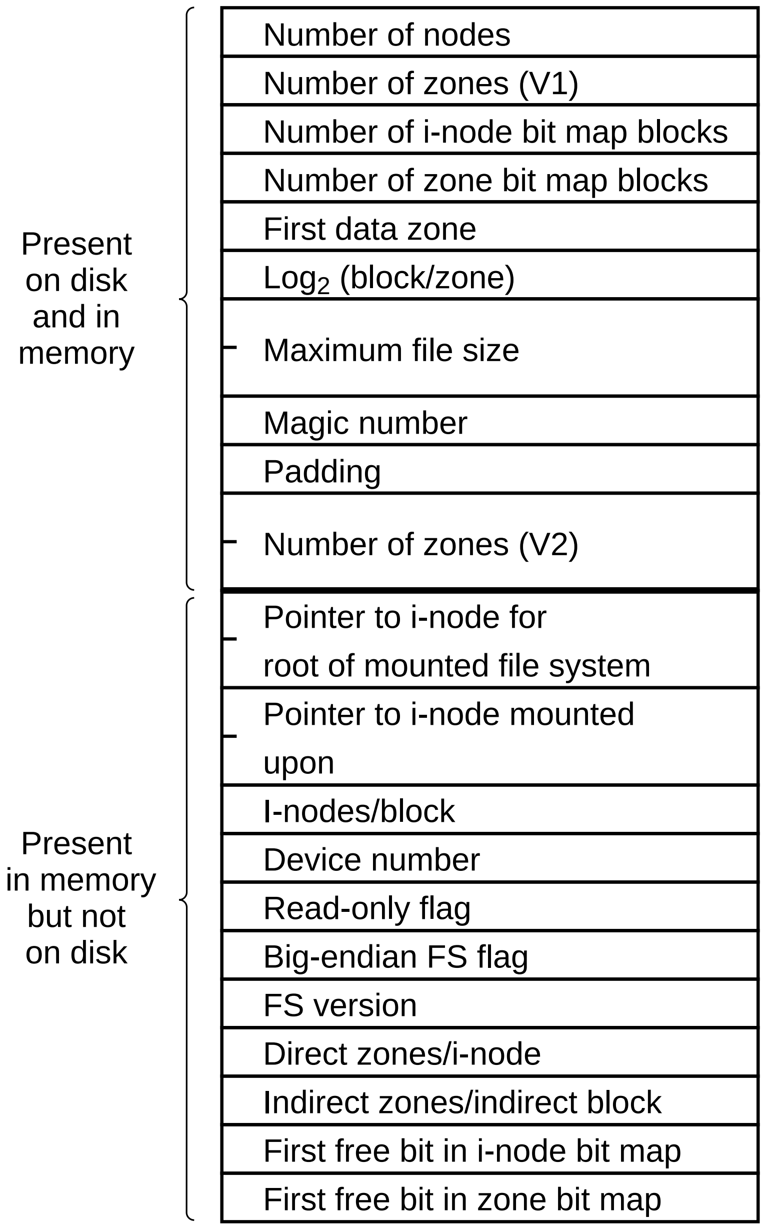


Figure 5-29. The MINIX super-block.

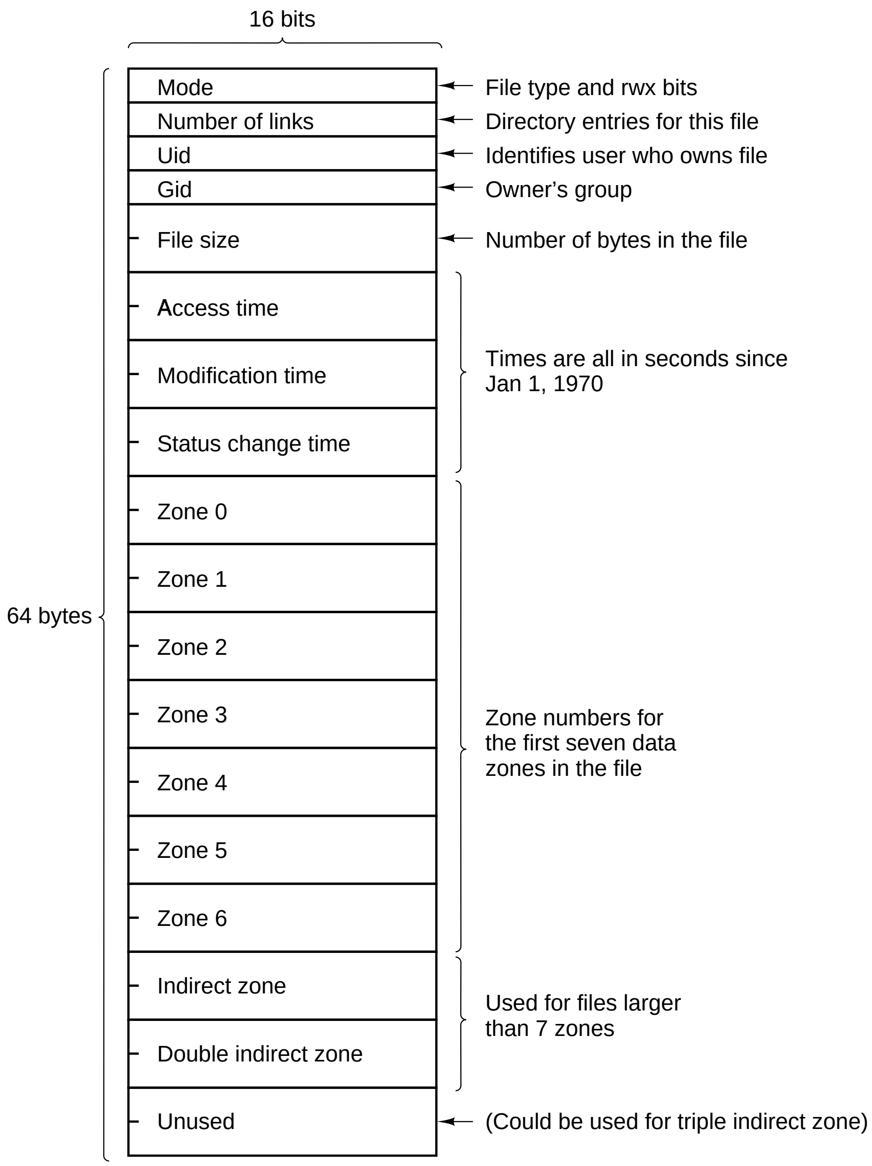


Figure 5-30. The MINIX i-node.

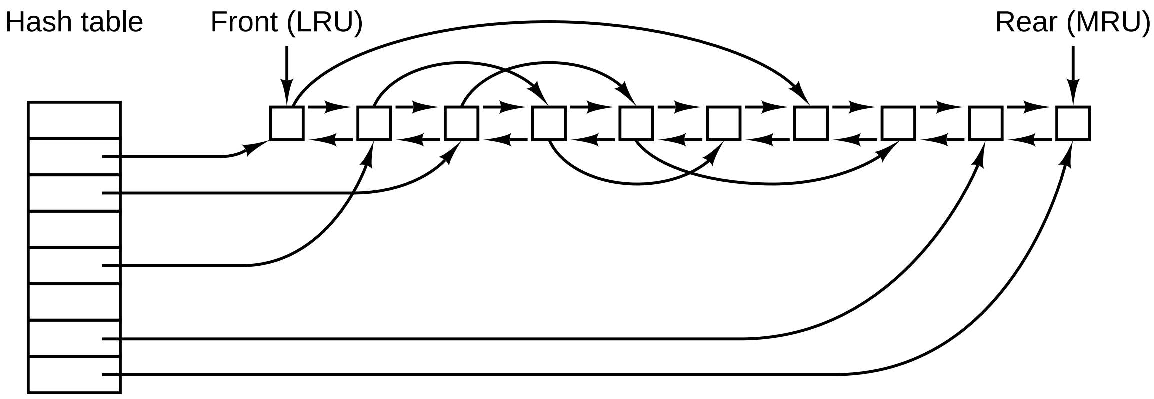


Figure 5-31. The linked lists used by the block cache.

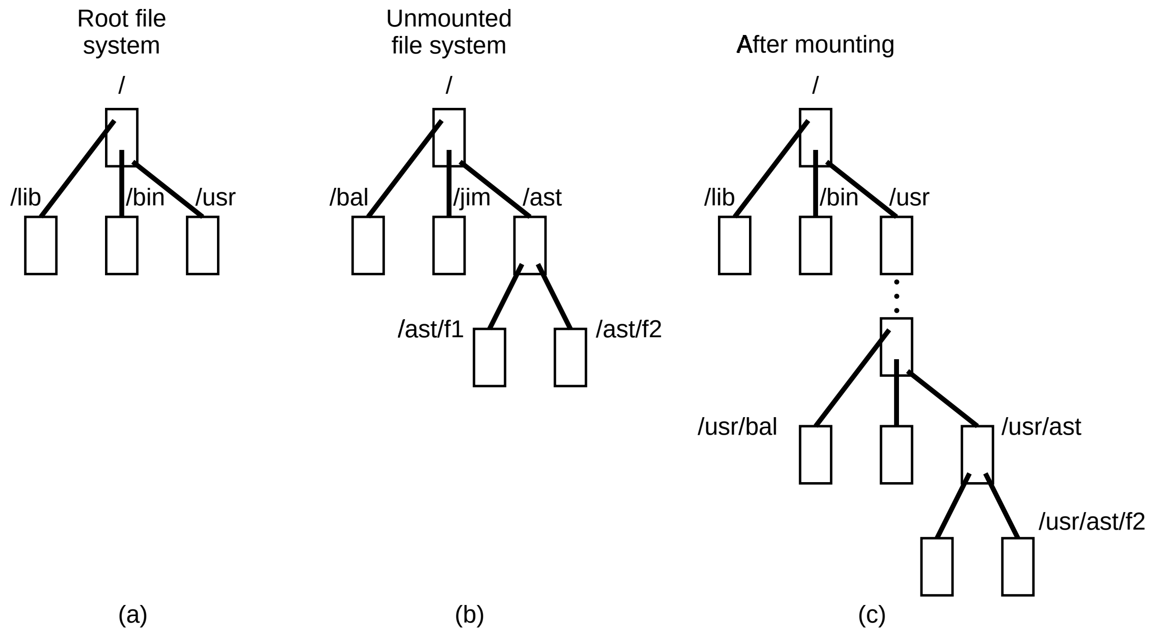


Figure 5-32. (a) Root file system. (b) An unmounted file system. (c) The result of mounting the file system of (b) on */usr*.

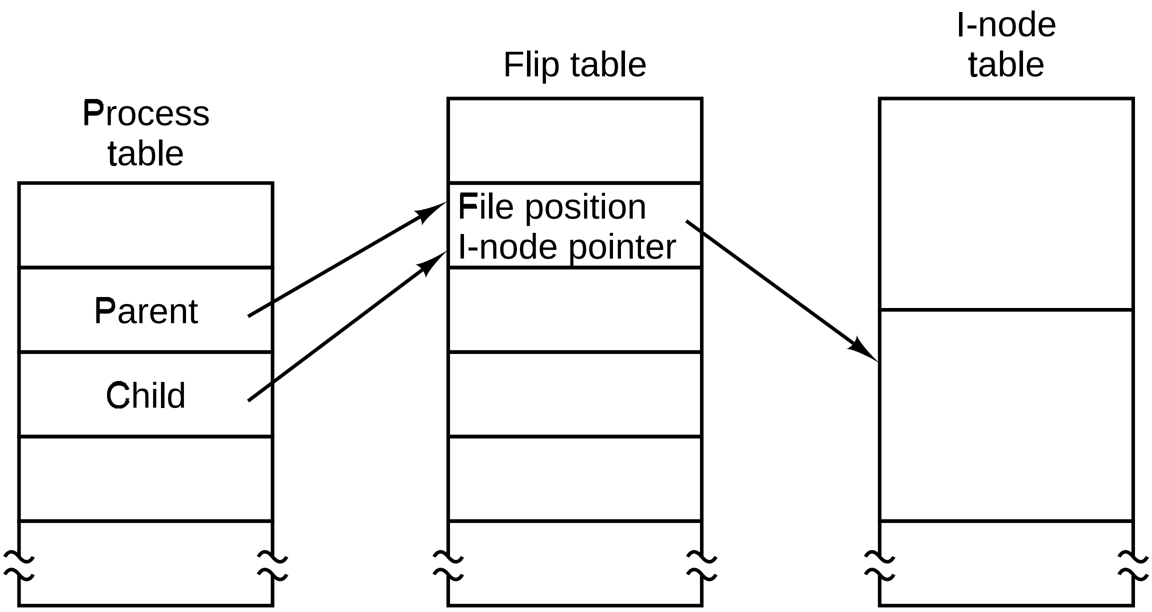


Figure 5-33. How file positions are shared between a parent and a child.

Procedure	Function
get_block	Fetch a block for reading or writing
put_block	Return a block previously requested with get_block
alloc_zone	Allocate a new zone (to make a file longer)
free_zone	Release a zone (when a file is removed)
rw_block	Transfer a block between disk and cache
invalidate	Purge all the cache blocks for some device
flushall	Flush all dirty blocks for one device
rw_scattered	Read or write scattered data from or to a device
rm_lru	Remove a block from its LRU chain

Figure 5-34. Procedures used for block management.

Procedure	Function
get_inode	Fetch an i-node into memory
put_inode	Return an i-node that is no longer needed
alloc_inode	Allocate a new i-node (for a new file)
wipe_inode	Clear some fields in an i-node
free_inode	Release an i-node (when a file is removed)
update_times	Update time fields in an i-node
rw_inode	Transfer an i-node between memory and disk
old_icopy	Convert i-node contents to write to V1 disk i-node
new_icopy	Convert data read from V1 file system disk i-node
dup_inode	Indicate that someone else is using an i-node

Figure 5-35. Procedures used for i-node management.

Procedure	Function
alloc_bit	Allocate a bit from the zone or i-node map
free_bit	Free a bit in the zone or i-node map
get_super	Search the super-block table for a device
mounted	Report whether given i-node is on a mounted (or root) f.s.
read_super	Read a super-block

Figure 5-36. Procedures used to manage the super-block and bit maps.

Operation	Meaning
F_SETLK	Lock region for both reading and writing
F_SETLKW	Lock region for writing
F_GETLK	Report if region is locked

Figure 5-37. The POSIX advisory record locking operations. These operations are requested by using an FCNTL system call.

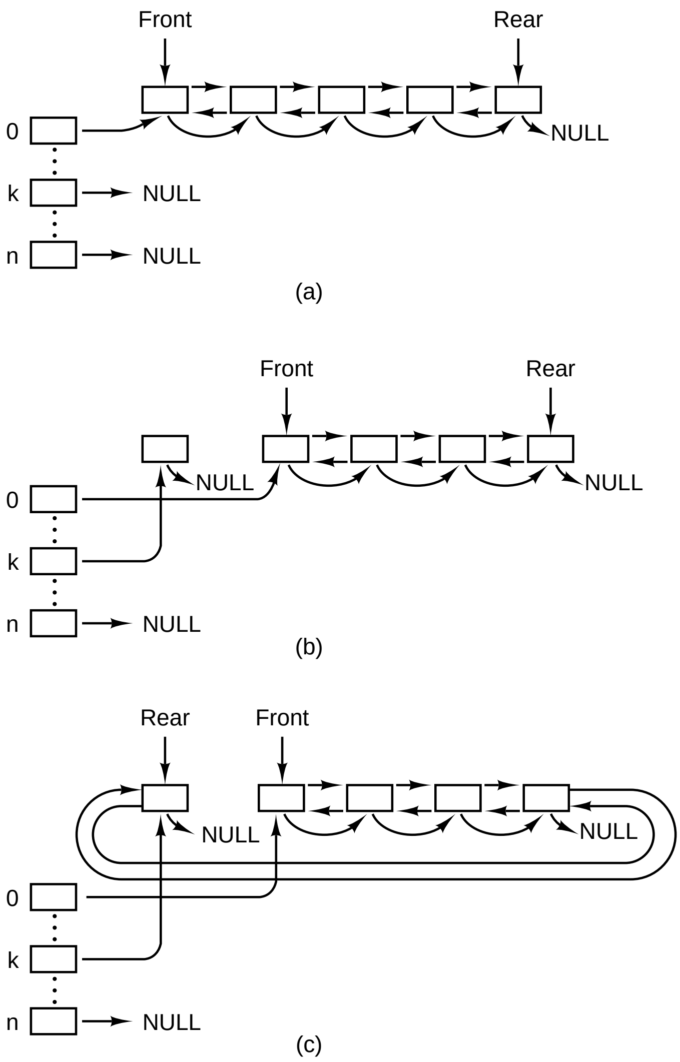


Figure 5-38. Block cache initialization. (a) Before any buffers have been used. (b) After one block has been requested. (c) After the block has been released.

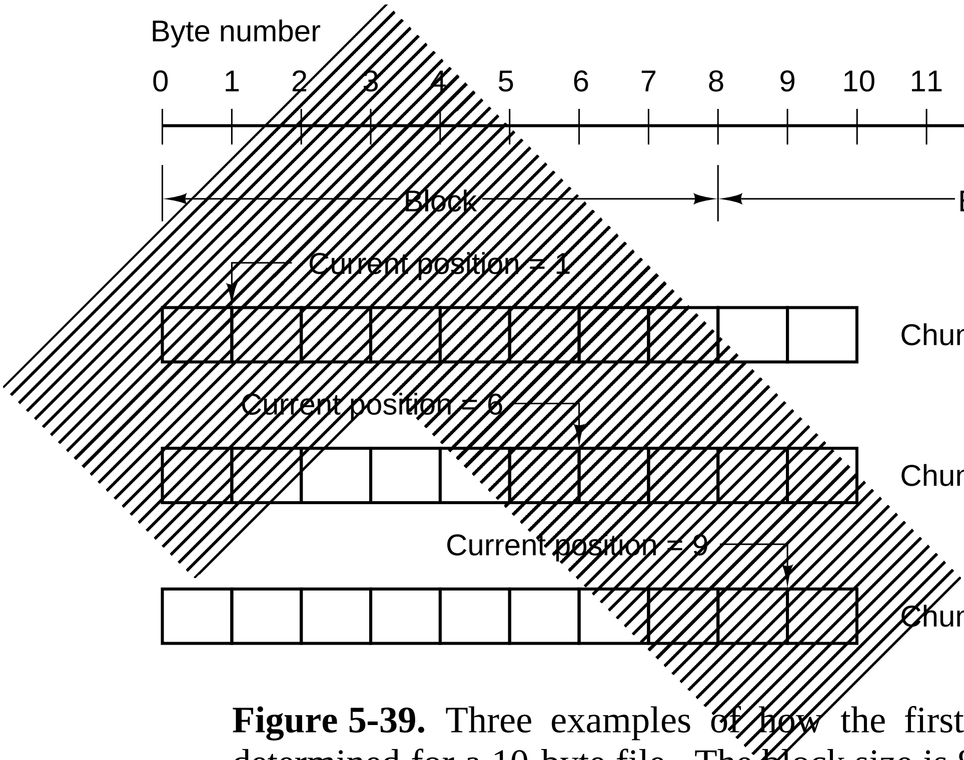


Figure 5-39. Three examples of how the first chunk size is determined for a 10-byte file. The block size is 8 bytes, and the number of bytes requested is 6. The chunk is shown shaded.

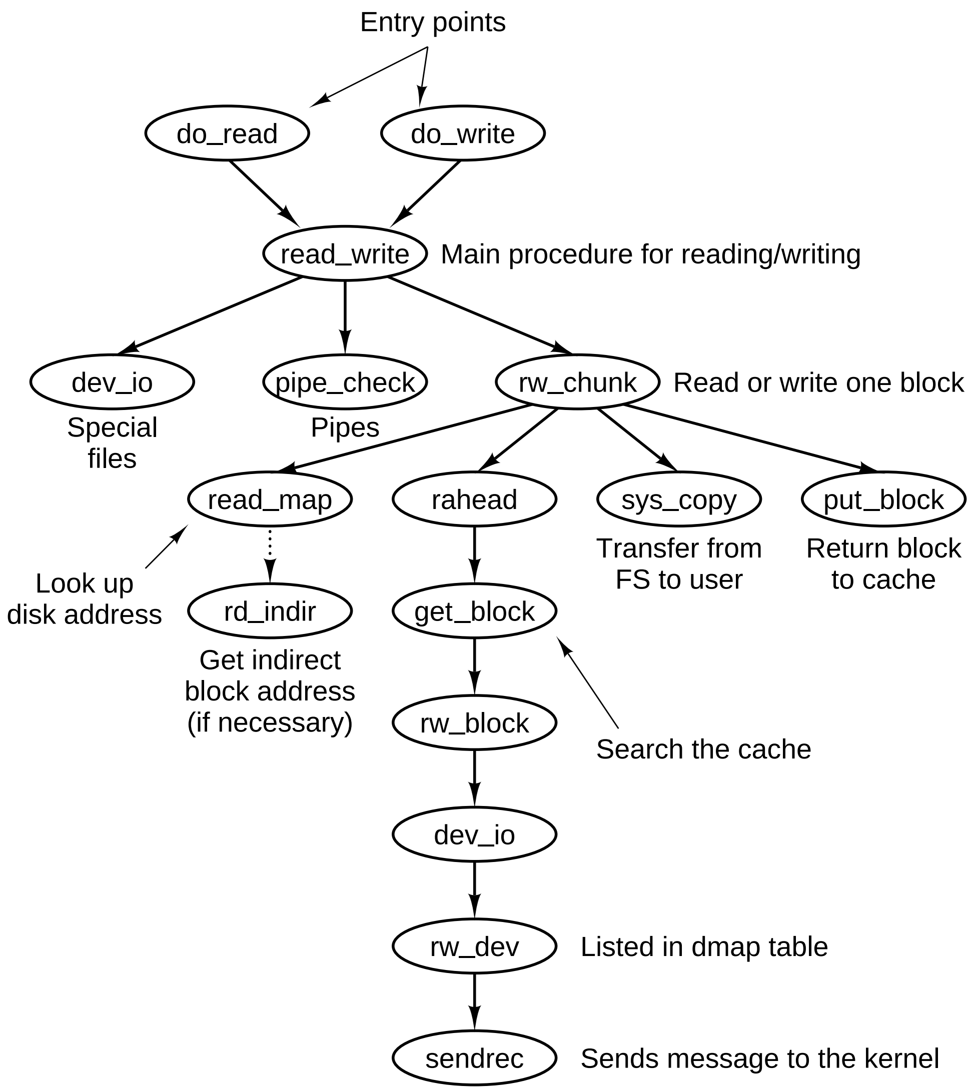


Figure 5-40. Some of the procedures involved in reading a file.

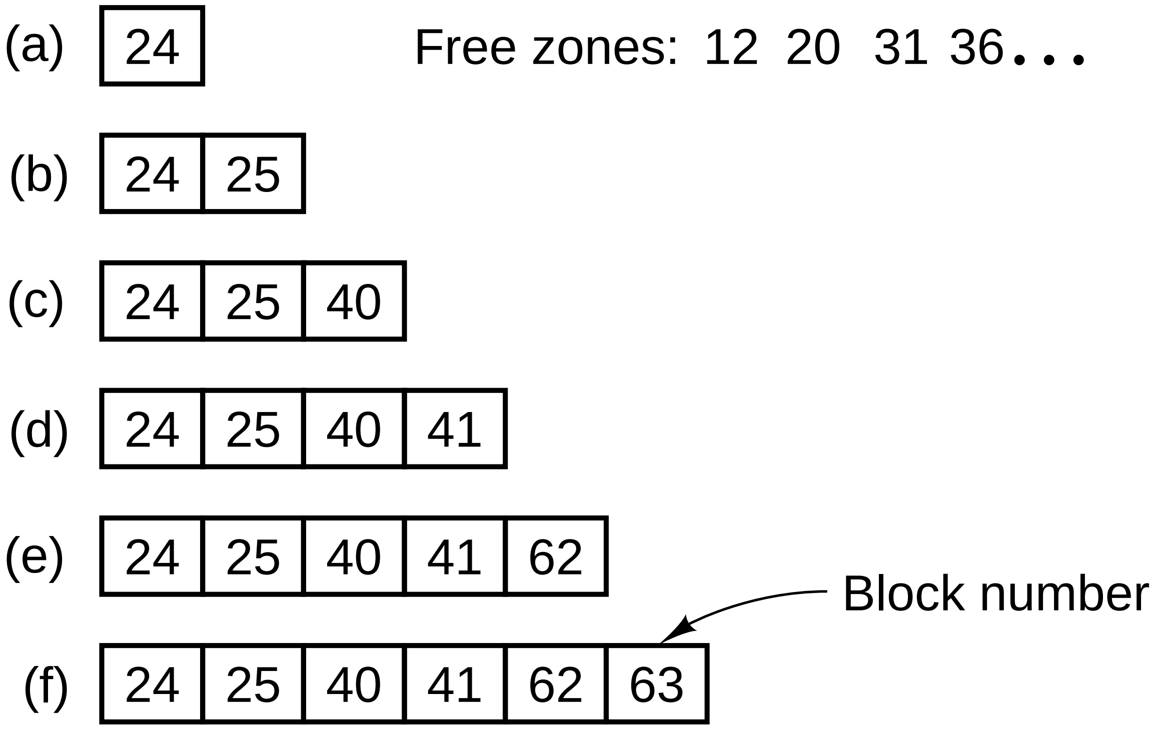


Figure 5-41. (a) - (f) The successive allocation of 1K blocks with a 2K zone.

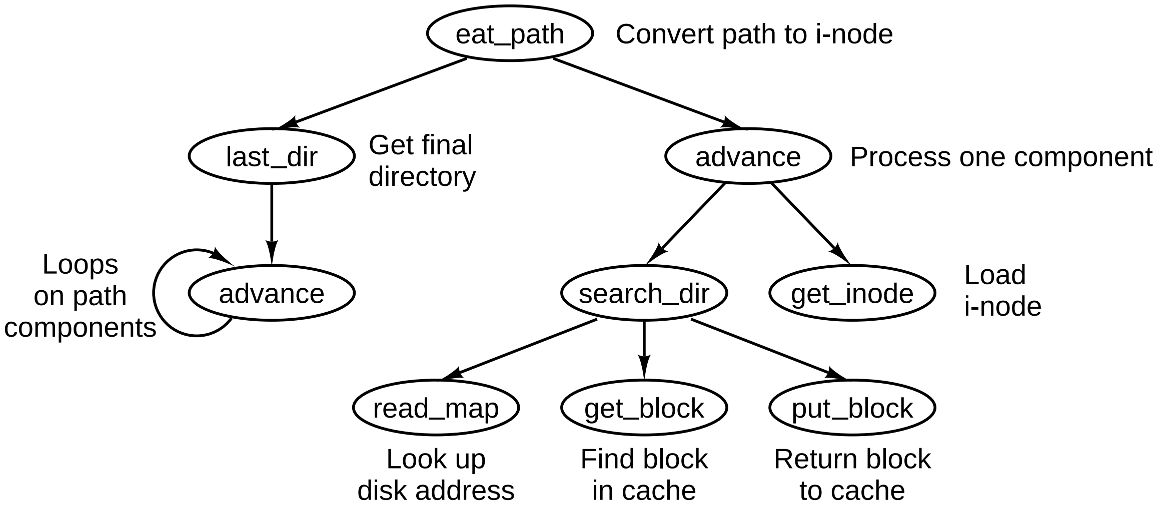


Figure 5-42. Some of the procedures used in looking up path names.

Call	Function
UTIME	Set a file's time of last modification
TIME	Set the current real time in seconds
STIME	Set the real time clock
TIMES	Get the process accounting times

Figure 5-43. The four system calls involving time.

Operation	Meaning
F_DUPFD	Duplicate a file descriptor
F_GETFD	Get the close-on-exec flag
F_SETFD	Set the close-on-exec flag
F_GETFL	Get file status flags
F_SETFL	Set file status flags
F_GETLK	Get lock status of a file
F_SETLK	Set read/write lock on a file
F_SETLKW	Set write lock on a file

Figure 5-44. The POSIX request parameters for the FCNTL system call.