```
algorithm bread     /* block read */
input:  file system block number
output: buffer containing data
{
        get buffer for block (algorithm getblk);
        if (buffer data valid)
                return buffer;
        initiate disk read;
        sleep(event disk read complete);
        return(buffer);
}
```

Figure 3.13. Algorithm for Reading a Disk Block

```
algorithm breada        /* block read and read ahead */
input:  (1) file system block number for immediate read
        (2) file system block number for asynchronous read
output: buffer containing data for immediate read
{
        if (first block not in cache)
        {
                get buffer for first block (algorithm getblk);
                if (buffer data not valid)
                        initiate disk read;
        }
        if (second block not in cache)
        {
                get buffer for second block (algorithm getblk);
                if (buffer data valid)
                        release buffer (algorithm brelse);
                else
                        initiate disk read;
        }
        if (first block was originally in cache)
        {
                read first block (algorithm bread);
                return buffer;
        }
        sleep(event first buffer contains valid data);
        return buffer;
}
```

Figure 3.14. Algorithm for Block Read Ahead

```
algorithm bwrite      /* block write */
input:  buffer
output: none
{
        initiate disk write;
        if (I/O synchronous)
        {
                sleep(event I/O complete);
                release buffer (algorithm brelse);
        }
        else if (buffer marked for delayed write)
                mark buffer to put at head of free list;
}
```

Figure 3.15. Algorithm for Writing a Disk Block

```
algorithm iget
input:  file system inode number
output: locked inode
{
        while (not done)
        {
                if (inode in inode cache)
                {
                        if (inode locked)
                        {
                                sleep (event inode becomes unlocked);
                                continue;       /* loop back to while */
                        }
                        /* special processing for mount points (Chapter 5) */
                        if (inode on inode free list)
                                remove from free list;
                        increment inode reference count;
                        return (inode);
                }

                /* inode not in inode cache */
                if (no inodes on free list)
                        return(error);
                remove new inode from free list;
                reset inode number and file system;
                remove inode from old hash queue, place on new one;
                read inode from disk (algorithm bread);
                initialize inode (e.g. reference count to 1);
                return(inode);
        }
}
```

Figure 4.3. Algorithm for Allocation of In-Core Inodes

```
algorithm iput                /* release (put) access to in-core inode */
input:  pointer to in-core inode
output: none
{
        lock inode if not already locked;
        decrement inode reference count;
        if (reference count == 0)
        {
                if (inode link count == 0)
                {
                        free disk blocks for file (algorithm free, section 4.7);
                        set file type to 0;
                        free inode (algorithm ifree, section 4.6);
                }
                if (file accessed or inode changed or file changed)
                        update disk inode;
                put inode on free list;
        }
        release inode lock;
}
```

Figure 4.4. Releasing an Inode

```
algorithm bmap      /* block map of logical file byte offset to file system block */
input:   (1) inode
         (2) byte offset
output:  (1) block number in file system
         (2) byte offset into block
         (3) bytes of I/O in block
         (4) read ahead block number
{
    calculate logical block number in file from byte offset;
    calculate start byte in block for I/O;                /* output 2 */
    calculate number of bytes to copy to user;            /* output 3 */
    check if read-ahead applicable, mark inode;           /* output 4 */
    determine level of indirection;
    while (not at necessary level of indirection)
    {
        calculate index into inode or indirect block from
                    logical block number in file;
        get disk block number from inode or indirect block;
        release buffer from previous disk read, if any (algorithm brelse);
        if (no more levels of indirection)
            return (block number);
        read indirect disk block (algorithm bread);
        adjust logical block number in file according to level of indirection;
    }
}
```

Figure 4.8. Conversion of Byte Offset to Block Number in File System

```
algorithm namei        /* convert path name to inode */
input:  path name
output: locked inode
{
        if (path name starts from root)
                working inode = root inode (algorithm iget);
        else
                working inode = current directory inode (algorithm iget):

        while (there is more path name)
        {
                read next path name component from input;
                verify that working inode is of directory, access permissions OK;
                if (working inode is of root and component is "..")
                        continue;       /* loop back to while */
                read directory (working inode) by repeated use of algorithms
                        bmap, bread and brelse;
                if (component matches an entry in directory (working inode))
                {
                        get inode number for matched component;
                        release working inode (algorithm iput);
                        working inode = inode of matched component (algorithm iget);
                }
                else        /* component not in directory */
                        return (no inode);
        }
        return (working inode);
}
```

Figure 4.11. Algorithm for Conversion of a Path Name to an Inode

```
algorithm ialloc        /* allocate inode */
input:   file system
output: locked inode
{
        while (not done)
        {
                if (super block locked)
                {
                        sleep (event super block becomes free);
                        continue;       /* while loop */
                }
                if (inode list in super block is empty)
                {
                        lock super block;
                        get remembered inode for free inode search;
                        search disk for free inodes until super block full,
                                or no more free inodes (algorithms bread and brelse);
                        unlock super block;
                        wake up (event super block becomes free);
                        if (no free inodes found on disk)
                                return (no inode);
                        set remembered inode for next free inode search;
                }
                /* there are inodes in super block inode list */
                get inode number from super block inode list;
                get inode (algorithm iget);
                if (inode not free after all)       /* !!! */
                {
                        write inode to disk;
                        release inode (algorithm iput);
                        continue;       /* while loop */
                }
                /* inode is free */
                initialize inode;
                write inode to disk;
                decrement file system free inode count;
                return (inode);
        }
}
```

Figure 4.12. Algorithm for Assigning New Inodes

```
algorithm ifree          /* inode free */
input:   file system inode number
output: none
{
        increment file system free inode count;
        if (super block locked)
                return;
        if (inode list full)
        {
                if (inode number less than remembered inode for search)
                        set remembered inode for search = input inode number;
        }
        else
                store inode number in inode list;
        return;
}
```

Figure 4.14. Algorithm for Freeing Inode

The algorithm for freeing an inode is much simpler. After incrementing the

```
algorithm alloc     /* file system block allocation */
input:  file system number
output: buffer for new block
{
        while (super block locked)
                sleep (event super block not locked);
        remove block from super block free list;
        if (removed last block from free list)
        {
                lock super block;
                read block just taken from free list (algorithm bread);
                copy block numbers in block into super block;
                release block buffer (algorithm brelse);
                unlock super block;
                wake up processes (event super block not locked);
        }
        get buffer for block removed from super block list (algorithm getblk);
        zero buffer contents;
        decrement total count of free blocks;
        mark super block modified;
        return buffer;
}
```

Figure 4.19. Algorithm for Allocating Disk Block

```
#include <fcntl.h>
main()
{
        int fd1, fd2;
        char buf1[512], buf2[512];

        fd1 = open("/etc/passwd", O_RDONLY);
        fd2 = open("/etc/passwd", O_RDONLY);
        read(fd1, buf1, sizeof(buf1));
        read(fd2, buf2, sizeof(buf2));
}
```

Figure 5.9. *Reading a File via Two File Descriptors*

```
algorithm open
inputs:  file name
         type of open
         file permissions (for creation type of open)
output: file descriptor
{
      convert file name to inode (algorithm namei);
      if (file does not exist or not permitted access)
            return(error);
      allocate file table entry for inode, initialize count, offset;
      allocate user file descriptor entry, set pointer to file table entry;
      if (type of open specifies truncate file)
            free all file blocks (algorithm free);
      unlock(inode);                /* locked above in namei */
      return(user file descriptor);
}
```

Figure 5.2. Algorithm for Opening a File

```
algorithm read
input:  user file descriptor
        address of buffer in user process
        number of bytes to read
output: count of bytes copied into user space
{
        get file table entry from user file descriptor;
        check file accessibility;
        set parameters in u area for user address, byte count, I/O to user;
        get inode from file table;
        lock inode;
        set byte offset in u area from file table offset;
        while (count not satisfied)
        {
                convert file offset to disk block (algorithm bmap);
                calculate offset into block, number of bytes to read;
                if (number of bytes to read is 0)
                        /* trying to read end of file */
                        break;          /* out of loop */
                read block (algorithm breada if with read ahead, algorithm
                                bread otherwise);
                copy data from system buffer to user address;
                update u area fields for file byte offset, read count,
                                address to write into user space;
                release buffer;         /* locked in bread */
        }
        unlock inode;
        update file table offset for next read;
        return(total number of bytes read);
}
```

Figure 5.5. Algorithm for Reading a File

```
#include <fcntl.h>
/* process A */
main()
{
        int fd;
        char buf[512];
        fd = open("/etc/passwd", O_RDONLY);
        read(fd, buf, sizeof(buf));          /* read1 */
        read(fd, buf, sizeof(buf));          /* read2 */
}

/* process B */
main()
{
        int fd, i;
        char buf[512];
        for (i = 0;  i < sizeof(buf);  i++)
                buf[i] = 'a';
        fd = open("/etc/passwd", O_WRONLY);
        write(fd, buf, sizeof(buf));         /* write1 */
        write(fd, buf, sizeof(buf));         /* write2 */
}
```

Figure 5.8. A Reader and a Writer Process

```
algorithm creat
input:   file name
         permission settings
output: file descriptor
{
        get inode for file name (algorithm namei);
        if (file already exists)
        {
                if (not permitted access)
                {
                        release inode (algorithm iput);
                        return(error);
                }
        }
        else        /* file does not exist yet */
        {
                assign free inode from file system (algorithm ialloc);
                create new directory entry in parent directory: include
                        new file name and newly assigned inode number;
        }
        allocate file table entry for inode, initialize count;
        if (file did exist at time of create)
                free all file blocks (algorithm free);
        unlock(inode);
        return(user file descriptor);
}
```

Figure 5.12. Algorithm for Creating a File

```
algorithm make new node
inputs:  node (file name)
         file type
         permissions
         major, minor device number (for block, character special files)
output: none
{
      if (new node not named pipe and user not super user)
            return(error);
      get inode of parent of new node (algorithm namei);
      if (new node already exists)
      {
            release parent inode (algorithm iput);
            return(error);
      }
      assign free inode from file system for new node (algorithm ialloc);
      create new directory entry in parent directory: include new node
            name and newly assigned inode number;
      release parent directory inode (algorithm iput);
      if (new node is block or character special file)
            write major, minor numbers into inode structure;
      release new node inode (algorithm iput);
}
```

Figure 5.13. Algorithm for Making New Node

```
algorithm change directory
input:   new directory name
output: none
{
        get inode for new directory name (algorithm namei);
        if (inode not that of directory or process not permitted access to file)
        {
                release inode (algorithm iput);
                return(error);
        }
        unlock inode;
        release "old" current directory inode (algorithm iput);
        place new inode into current directory slot in u area;
}
```

Figure 5.14. Algorithm for Changing Current Directory

```
algorithm pipe
input:   none
output: read file descriptor
        write file descriptor
{
        assign new inode from pipe device (algorithm ialloc);
        allocate file table entry for reading, another for writing;
        initialize file table entries to point to new inode;
        allocate user file descriptor for reading, another for writing,
                initialize to point to respective file table entries;
        set inode reference count to 2;
        initialize count of inode readers, writers to 1;
}
```

Figure 5.16. Algorithm for Creation of (Unnamed) Pipes

DUP

```
#include  <fcntl.h>
main()
{
        int i, j;
        char buf1[512], buf2[512];

        i = open("/etc/passwd", O_RDONLY);
        j = dup(i);
        read(i, buf1, sizeof(buf1));
        read(j, buf2, sizeof(buf2));
        close(i);
        read(j, buf2, sizeof(buf2));
}
```

Figure 5.21. C Program Illustrating Dup

```
algorithm link
input:  existing file name
        new file name
output: none
{
    get inode for existing file name (algorithm namei);
    if (too many links on file or linking directory without super user permission)
    {
        release inode (algorithm iput);
        return(error);
    }
    increment link count on inode;
    update disk copy of inode;
    unlock inode;
    get parent inode for directory to contain new file name (algorithm namei);
    if (new file name already exists or existing file, new file on
                                    different file systems)
    {
        undo update done above;
        return(error);
    }
    create new directory entry in parent directory of new file name:
        include new file name, inode number of existing file name;
    release parent directory inode (algorithm iput);
    release inode of existing file (algorithm iput);
}
```

Figure 5.29. Algorithm for Linking Files

```
algorithm unlink
input:   file name
output: none
{
        get parent inode of file to be unlinked (algorithm namei);
        /* if unlinking the current directory... */
        if (last component of file name is ".")
                increment inode reference count;
        else
                get inode of file to be unlinked (algorithm iget);
        if (file is directory but user is not super user)
        {
                release inodes (algorithm iput);
                return(error);
        }
        if (shared text file and link count currently 1)
                remove from region table;
        write parent directory:  zero inode number of unlinked file;
        release inode parent directory (algorithm iput);
        decrement file link count;
        release file inode (algorithm iput);
                /* iput checks if link count is 0:  if so,
                 * releases file blocks (algorithm free) and
                 * frees inode (algorithm ifree);
                 */
}
```

Figure 5.31. Algorithm for Unlinking a File

```
algorithm sleep
input:  (1) sleep address
        (2) priority
output: 1 if process awakened as a result of a signal that process catches,
        longjump algorithm if process awakened as a result of a signal
                        that it does not catch,
        0 otherwise;
{
        raise processor execution level to block all interrupts;
        set process state to sleep;
        put process on sleep hash queue, based on sleep address;
        save sleep address in process table slot;
        set process priority level to input priority;
        if (process sleep is NOT interruptible)
        {
                do context switch;
                /* process resumes execution here when it wakes up */
                reset processor priority level to allow interrupts as when
                        process went to sleep;
                return(0);
        }

        /* here, process sleep is interruptible by signals */
        if (no signal pending against process)
        {
                do context switch;
                /* process resumes execution here when it wakes up */
                if (no signal pending against process)
                {
                        reset processor priority level to what it was when
                                process went to sleep;
                        return(0);
                }
        }
        remove process from sleep hash queue, if still there;

        reset processor priority level to what it was when process went to sleep;
        if (process sleep priority set to catch signals)
                return(1)
        do longjmp algorithm;
}
```

Figure 6.31. Sleep Algorithm

```
algorithm wakeup                    /* wake up a sleeping process */
input:   sleep address
output: none
{
        raise processor execution level to block all interrupts;
        find sleep hash queue for sleep address;
        for (every process asleep on sleep address)
        {
                remove process from hash queue;
                mark process state "ready to run";
                put process on scheduler list of processes ready to run;
                clear field in process table entry for sleep address;
                if (process not loaded in memory)
                        wake up swapper process (0);
                else if (awakened process is more elligible to run than
                                currently running process)
                        set scheduler flag;
        }
        restore processor execution level to original level;
}
```

Figure 6.32. Algorithm for Wakeup

```
algorithm fork
input:   none
output: to parent process, child PID number
        to child process, 0

{
    check for available kernel resources;
    get free proc table slot, unique PID number;
    check that user not running too many processes;
    mark child state "being created;"
    copy data from parent proc table slot to new child slot;
    increment counts on current directory inode and changed root (if applicable);
    increment open file counts in file table;
    make copy of parent context (u area, text, data, stack) in memory;
    push dummy system level context layer onto child system level context;
                dummy context contains data allowing child process
                to recognize itself, and start running from here
                when scheduled;
    if (executing process is parent process)
    {
        change child state to "ready to run;"
        return(child ID);        /* from system to user */
    }
    else        /* executing process is the child process */
    {
        initialize u area timing fields;
        return(0);        /* to user */
    }
}
```

Figure 7.2. Algorithm for Fork

purpose administrative processes.

```
algorithm exit
input:   return code for parent process
output: none
{
        ignore all signals;
        if (process group leader with associated control terminal)
        {
                send hangup signal to all members of process group;
                reset process group for all members to 0;
        }
        close all open files (internal version of algorithm close);
        release current directory (algorithm iput);
        release current (changed) root, if exists (algorithm iput);
        free regions, memory associated with process (algorithm freereg);
        write accounting record;
        make process state zombie
        assign parent process ID of all child processes to be init process (1);
                if any children were zombie, send death of child signal to init;
        send death of child signal to parent process;
        context switch;
}
```

Figure 7.14. Algorithm for Exit

```
algorithm wait
input:  address of variable to store status of exiting process
output: child ID, child exit code
{
        if (waiting process has no child processes)
                return(error);

        for (;;)        /* loop until return from inside loop */
        {
                if (waiting process has zombie child)
                {
                        pick arbitrary zombie child;
                        add child CPU usage to parent;
                        free child process table entry;
                        return(child ID, child exit code);
                }
                if (process has no children)
                        return error;
                sleep at interruptible priority (event child process exits);
        }
}
```

Figure 7.16. Algorithm for Wait

```
algorithm exec
input:   (1) file name
         (2) parameter list
         (3) environment variables list
output: none
{
        get file inode (algorithm namei);
        verify file executable, user has permission to execute;
        read file headers, check that it is a load module;
        copy exec parameters from old address space to system space;
        for (every region attached to process)
                detach all old regions (algorithm detach);
        for (every region specified in load module)
        {
                allocate new regions (algorithm allocreg);
                attach the regions (algorithm attachreg);
                load region into memory if appropriate (algorithm loadreg);
        }
        copy exec parameters into new user stack region;
        special processing for setuid programs, tracing;
        initialize user register save area for return to user mode;
        release inode of file (algorithm iput);
}
```

Figure 7.19. Algorithm for Exec

```
algorithm start                /* system startup procedure */
input:  none
output: none
{
        initialize all kernel data structures;
        pseudo-mount of root;
        hand-craft environment of process 0;
        fork process 1:
        {
                /* process 1 in here */
                allocate region;
                attach region to init address space;
                grow region to accommodate code about to copy in;
                copy code from kernel space to init user space to exec init;
                change mode: return from kernel to user mode;
                /* init never gets here---as result of above change mode,
                 * init exec's /etc/init and becomes a "normal" user process
                 * with respect to invocation of system calls
                 */
        }
        /* proc 0 continues here */
        fork kernel processes;
        /* process 0 invokes the swapper to manage the allocation of
         * process address space to main memory and the swap devices.
         * This is an infinite loop;  process 0 usually sleeps in the
         * loop unless there is work for it to do.
         */
        execute code for swapper algorithm;
}
```

Figure 7.30. Algorithm for Booting the System

kernel, but that would

```
/* read command line until "end of file" */
while (read(stdin, buffer, numchars))
{
        /* parse command line */
        if (/* command line contains & */)
                amper = 1;
        else
                amper = 0;
        /* for commands not part of the shell command language */
        if (fork() == 0)
        {
                /* redirection of IO? */
                if (/* redirect output */)
                {
                        fd = creat(newfile, fmask);
                        close(stdout);
                        dup(fd);
                        close(fd);
                        /* stdout is now redirected */
                }
                if (/* piping */ )
                {
                        pipe(fildes);
```

Figure 7.28. Main Loop of the Shell

7.7

```
                if (fork() == 0)
                {
                        /* first component of command line */
                        close(stdout);
                        dup(fildes[1]);
                        close(fildes[1]);
                        close(fildes[0]);
                        /* stdout now goes to pipe */
                        /* child process does command */
                        execlp(command1, command1, 0);
                }
                /* 2nd command component of command line */
                close(stdin);
                dup(fildes[0]);
                close(fildes[0]);
                close(fildes[1]);
                /* standard input now comes from pipe */
        }
        execve(command2, command2, 0);
    }
    /* parent continues over here...
     * waits for child to exit if required
     */
    if (amper == 0)
        retid = wait(&status);
}
```

Figure 7.28. Main Loop of the Shell (continued)

```
algorithm init          /* init process, process 1 of the system */
input:  none
output: none
{
        fd = open("/etc/inittab", O_RDONLY);
        while (line_read(fd, buffer))
        {
                /* read every line of file */
                if (invoked state != buffer state)
                        continue;       /* loop back to while */
                /* state matched */
                if (fork() == 0)
                {
                        execl("process specified in buffer");
                        exit();
                }
                /* init process does not wait */
                /* loop back to while */
        }

        while ((id = wait((int *) 0)) != -1)
        {
                /* check here if a spawned child died;
                 * consider respawning it */
                /* otherwise, just continue */
        }
}
```

Figure 7.31. Algorithm for Init

```
algorithm schedule_process
input:  none
output: none
{
        while (no process picked to execute)
        {
                for (every process on run queue)
                        pick highest priority process that is loaded in memory;
                if (no process eligible to execute)
                        idle the machine;
                        /* interrupt takes machine out of idle state */
        }
        remove chosen process from run queue;
        switch context to that of chosen process, resume its execution;
}
```

Figure 8.1. Algorithm for Process Scheduling

```
algorithm clock
input:  none
output: none
{
        restart clock;              /* so that it will interrupt again */
        if (callout table not empty)
        {
                adjust callout times;
                schedule callout function if time elapsed;
        }
        if (kernel profiling on)
                note program counter at time of interrupt;
        if (user profiling on)
                note program counter at time of interrupt;
        gather system statistics;
        gather statistics per process;
        adjust measure of process CPU utilitization;
        if (1 second or more since last here and interrupt not in critical
                                                region of code)
        {
                for (all processes in the system)
                {
                        adjust alarm time if active;
                        adjust measure of CPU utilization;
                        if (process to execute in user mode)
                                adjust process priority;
                }
                wakeup swapper process is necessary;
        }
}
```

Figure 8.9. Algorithm for the Clock Handler