Operating Systems
with Linux

John O’Gorman
1 Introduction

1.1 Overview of a computer system
1.2 Interfaces to an operating system
1.3 Layered structure
1.4 Poorly designed system
1.5 Modular operating system
1.6 Microkernel with two personalities

2 Interfaces to an operating system

2.1 Memory layout for global variables
2.2 Checking error values
2.3 Indexing into the system call table
2.4 Calling a system service
2.5 Computer memory
2.6 Dual-ported memory
2.7 Keyboard interface with computer
2.8 Interface using interrupt line
2.9 Interrupt arbitration by controller
2.10 Direct memory access

3 Process manager

3.1 Program to illustrate change of state
3.2 States as program executes
3.3 Overlap of processor and devices
3.4 An underutilised system
3.5 High utilisation of CPU and disk drives
3.6 How the system keeps track of processes
3.7 Data structure representing a process
3.8 Program to illustrate fork()
3.9 Function call and thread creation
3.10 An entry in a wait queue
3.11 Three processes waiting for a resource
3.12 Process states and transitions
3.13 Running in user or kernel mode
3.14 The volatile environment of a process
3.15 Data saved by context switcher
3.16 Context switching
3.17 Processes on the ready and wait queues
3.18 Two different priority queues

4 Concurrency

4.1 Code to increment counter
4.2 Code to decrement counter
4.3 General structure of a cooperating program
4.4 First attempt at solution for two processes
4.5 Compiled version of entry protocol
4.6 Algorithms using turn
4.7 Algorithm using two flags
4.8 Setting the flag before testing
4.9 Algorithm for mutual exclusion
4.10 Entry protocols for both processes
4.11 Eight contending processes
4.12 Eisenberg and McGuire algorithm
4.13 Lamport’s algorithm

5 Low level IPC mechanisms

5.1 A selection of signal values
5.2 Changing the signal mask
5.3 Data structure tracking signal handlers
5.4 Entry protocol using test and set
5.5 Entry protocol using exchange
5.6 Semaphore for mutual exclusion
5.7 Successive states of a mutual exclusion semaphore
5.8 Semaphore for synchronisation
5.9 An alternative for synchronisation
5.10 Semaphore for resource allocation
5.11 Circular buffer with eight slots
5.12 Algorithm for producer
5.13 Algorithm for consumer
5.14 Algorithm for one of many producers
5.15 Algorithm for one of many consumers
5.16 Entry protocol for a reader
5.17 Exit protocol for a reader
5.18 Solution 2
5.19 Solution 3
5.20 Reader prologue code
5.21 Incorrect reader prologue
5.22 Reader epilogue code
5.23 Writer prologue code
5.24 Writer epilogue code
5.25 Solution 4
5.26 Reader epilogue code
5.27 Writer prologue code
5.28 Writer epilogue code
5.29 Timing relationship of locks and semaphores
5.30 Data structure representing a semaphore set
5.31 Data structure controlling access to a semaphore
5.32 Data structure representing an individual semaphore
5.33 System V semaphore layout
5.34 Data structure representing a waiting process
5.35 Processes waiting on the semaphore set
5.36 Eventcount for synchronisation
5.37 Eventcount and sequencer for mutual exclusion
5.38 Eventcounts with one producer, one consumer

6 Higher level mechanisms for IPC

6.1 Outline of message passing
6.2 A message queue
6.3 Data structure representing a message queue
6.4 Structure of an actual message
6.5 System V message queue layout
6.6 Data structure representing a message
6.7 Readers/writers using conditional critical regions
6.8 Implementation of CWAIT
6.9 Implementation of CSIGNAL
6.10 An illustration of a monitor
6.11 Monitor to allocate a single resource
6.12 Monitor to manage a bounded buffer
6.13 Monitor to implement reader/writer interlock

7 Deadlock

7.1 A resource allocation graph
7.2 A resource allocation graph for four processes
7.3 Safe, unsafe, and deadlocked system
7.4 A claims graph
7.5 One resource allocated
7.6 The banker’s algorithm
7.7 Safety algorithm
7.8 State of the system before request
7.9 Application of the safety algorithm
7.10 State of the system reflecting the request
7.11 Resource allocation graph
7.12 Wait-for graph
7.13 Algorithm to detect deadlock
7.14 A system which is not deadlocked
7.15 Application of the deadlock detection algorithm
7.16 A deadlocked system

8 Memory manager

8.1 Hardware memory
8.2 The memory pyramid
8.3 Program as loaded by relocating loader
8.4 Virtual memory
8.5 Address modification in CPU
8.6 Protection and address modification
8.7 Compaction
8.8 A logical address
8.9 Address translation with segment table
8.10 A segment table entry
8.11 Searching a segmentation cache
8.12 Segmentation algorithm
8.13 Data structure representing a physical page
8.14 Tracking free physical memory
8.15 Linux page table structure
8.16 Format of page table entry
8.17 Address translation with cache and page table
8.18 Program to initialise an array
8.19 The root of the memory management data structures
8.20 Data structure controlling a region
8.21 Memory management data structures for a process
8.22 Possible values for the flags field
8.23 Operations on a region
8.24 Pages actually in memory
8.25 Example mapping of three regions
8.26 Relationship between page frames and page faults
8.27 Algorithm for NRU replacement
8.28 Effect of thrashing on CPU utilisation
8.29 Effect of window size on working set
8.30 An address interpreted as segment, page, offset
8.31 Algorithm for paged segmentation
8.32 Address translation with paged segmentation

9 Input and output

9.1 Overview of I/O processing
9.2 A tree-structured directory
9.3 Structure of a directory entry
9.4 A directory with links
9.5 How a hard link is implemented
9.6 How a symbolic link is implemented
9.7 A directory structure with cycles
9.8 Inode for an open file
9.9 Inodes on LRU and hash lists
9.10 Mode bit values
9.11 Some functions of the inode interface
9.12 An element in the cache of recent lookups
9.13 Directory name lookup cache
9.14 A protection bitmap
9.15 Definition of file descriptor table
9.16 Descriptor and open file data structure
9.17 Structure representing an open stream
9.18 File opened by one user
9.19 Operations on open files
9.20 Algorithm for open
9.21 File opened by two users
9.22 Generic I/O algorithm
9.23 Generic algorithm for Unix I/O procedure
9.24 Data structure for polling a descriptor
9.25 Several processes waiting on a device
9.26 Structure representing a lock on a file
9.27 A lock request which blocks
9.28 How locks are recorded
9.29 A buffered input procedure
9.30 A buffer head
9.31 Bits representing the state of a buffer
9.32 Buffer cache

10 Regular file systems

10.1 An entry in the linked list of file systems
10.2 Layout of a disk
10.3 Contiguous allocation
10.4 Linked allocation
10.5 Example of a file map
10.6 Layout of a partition with Ext2
10.7 A block group on disk
10.8 An Ext2 super block
10.9 Structure of a block group descriptor
10.10 Structure of an Ext2 disk inode
10.11 Allocation information in an inode
10.12 Single indirect block
10.13 Double indirect block
10.14 Fragments of two files in a block
10.15 An Ext2 directory entry
10.16 File systems as they exist on disk
10.17 Directory structure after attaching file system B
10.18 Data structure representing a mounted file system
10.19 Operations on a file system
10.20 Superblock for an Ext2 file system
10.21 Data structures involved in mounting a file system
10.22 Ext2 file system inode data in memory

11 Special files

11.1 Calling a Unix style driver
11.2 Device driver as a process
11.3 The character and block device switches
11.4 An entry in the character device switch
11.5 Algorithm for reading from a character device
11.6 Request block for block devices
11.7 Algorithm for strategy routine
11.8 Algorithm for interrupt service routine
11.9 Algorithm for writing to a character device
11.10 A basic view of a stream
11.11 Data structures constituting a minimal stream
11.12 A many-to-one multiplexor
11.13 A one-to-many multiplexor
11.14 A many-to-many multiplexor
11.15 Networking multiplexor
11.16 Data structure controlling terminal characteristics
11.17 Logging on using a pseudo terminal

12 IPC files

12.1 Setting up a pipe for interprocess communication
12.2 Data structure representing a pipe
12.3 Representation of a pipe
12.4 Supported address families
12.5 Data structure representing a protocol family
12.6 Domains registered in the kernel
12.7 Data structure representing a socket
12.8 State values for a socket
12.9 Data structure representing a protocol
12.10 Data structures after socket is allocated
12.11 Generic format of a socket address
12.12 Message header
12.13 Domain-specific control block
12.14 Hash chain of Unix domain control blocks
12.15 A protocol specific structure
12.16 Links for a Unix specific socket
12.17 Structure containing address of a Unix domain socket
12.18 A pair of connected sockets
12.19 Buffer used for communication over sockets
12.20 Data part of an sk_buff

13 Distributed systems

13.1 Networked computers
13.2 Workstations connected on a LAN
13.3 Diskless workstations on a LAN
13.4 Processor pool model
13.5 Combined model
13.6 Part of the Internet name space
13.7 Using a name server
13.8 Translating an Internet name
13.9 Overview of CORBA
13.10 The distributed computing environment

14 Communication

14.1 Data structure describing a network interface
14.2 An Internet protocol control block
14.3 Internet socket address structure
14.4 Connecting two sockets
14.5 A physical frame
14.6 An ethernet header
14.7 Structure identifying a packet type
14.8 An IP header
14.9 A UDP header
14.10 A TCP header
14.11 Function call/return
14.12 Call/return with RPC
14.13 Format of RPC specification language
14.14 Interface definition for MATH program
14.15 Unix style credentials
14.16 Overview of RPC system
14.17 Structure of an RPC message
14.18 Body of a call message
14.19 Body of a reply message
15 Concurrency in distributed systems

15.1 Events in process X
15.2 Events in process Y
15.3 Events in process Z
15.4 Centralised control of mutual exclusion
15.5 Distributed control of mutual exclusion
15.6 Successful two phase commit
15.7 Aborted two phase commit
15.8 A cascaded abort
15.9 A lock table
15.10 Algorithm for lock
15.11 Algorithm for unlock
15.12 Local and global wait-for graphs

16 Distributed file systems

16.1 Situation before attaching remote files
16.2 Local file system after attaching remote files
16.3 A client and two servers
16.4 File system after mounting Dir4
16.5 File system after mounting Dir7
16.6 Mounting a remote file system
16.7 NFS specific super block information
16.8 Interaction between client and server
16.9 NFS specific inode information
16.10 Handling local and remote file systems

17 Fault tolerance and security

17.1 Example access matrix
17.2 Sequence of messages with secret keys
17.3 Digital signature with secret key
17.4 Encryption for both security and authentication
17.5 Three phases in the Kerberos system
17.6 Initial exchange of messages with authentication server
17.7 Authenticating a user to a file server
17.8 Authenticating a user to a file server
17.9 Requesting service of a file server
Figure 1.1: Overview of a computer system
Figure 1.2: Interfaces to an operating system
Figure 1.3: Layered structure
Figure 1.4: Poorly designed system
System service interface

File manager

I/O manager

Disk driver

Interface with hardware

Network manager

Terminal driver

Process manager

Memory manager

Network driver

Figure 1.5: Modular operating system
Figure 1.6: Microkernel with two personalities
Figure 2.1: Memory layout for global variables
#include <errno.h>

ret_val = sysservice();
if (ret_val == -1) switch(errno)
    {case EAGAIN: printf("one message");
        break;
        case EBADPARM: printf("another message");
    };

Figure 2.2: Checking error values
Figure 2.3: Indexing into the system call table
Figure 2.4: Calling a system service
Figure 2.5: Computer memory
Figure 2.6: Dual-ported memory
Figure 2.7: Keyboard interface with computer
Figure 2.8: Interface using interrupt line
Figure 2.9: Interrupt arbitration by controller
Figure 2.10: Direct memory access
1: MOV EAX, 7  ; load the value 7 into register EAX
2: INC EAX      ; increment register EAX
3: MOV total, EAX; store the value from register EAX to total

Figure 3.1: Program to illustrate change of state
<table>
<thead>
<tr>
<th>State</th>
<th>EIP</th>
<th>EAX</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial state</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>State after 1st instruction</td>
<td>2</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>State after 2nd instruction</td>
<td>3</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>State after 3rd instruction</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

**Figure 3.2:** States as program executes
Figure 3.3: Overlap of processor and devices
Figure 3.4: An underutilised system
Figure 3.5: High utilisation of CPU and disk drives
Figure 3.6: How the system keeps track of processes
struct task_struct{
    volatile long state;
    long counter, priority;
    struct task_struct *next_task, *prev_task;
    struct task_struct *next_run, *prev_run;
    int exit_code, exit_signal;
    int pid;
    struct task_struct *p_opptr, *p_cptr;
    struct wait_queue *wait_chldexit;
    struct task_struct *pidhash_next;
    unsigned long policy;
    struct tms times;
    unsigned long start_time;
    unsigned short uid, gid;
    struct thread_struct tss;
    struct files_struct *files;
    struct mm_struct *mm;
    struct signal_struct *sig;
    sigset_t signal, blocked;
};

Figure 3.7: Data structure representing a process
printf ("Before the fork\n");
fork();
printf ("After the fork\n");

Figure 3.8: Program to illustrate fork()
Figure 3.9: Function call and thread creation
struct wait_queue{
    struct task_struct *task;
    struct wait_queue *next;
};

Figure 3.10: An entry in a wait queue
Figure 3.11: Three processes waiting for a resource
Figure 3.12: Process states and transitions
Figure 3.13: Running in user or kernel mode
struct thread_struct{
    unsigned long cr3;
    unsigned long eip;
    unsigned long eflags;
    unsigned long eax, ebx, ecx, edx;
    unsigned long esp;
    unsigned long ebp;
    unsigned long esi;
    unsigned long edi;
    unsigned short ss;
    unsigned short cs;
    unsigned short ds;
    unsigned short es;
    unsigned short fs;
    unsigned short gs;
};

Figure 3.14: The volatile environment of a process
Figure 3.15: Data saved by context switcher
Figure 3.16: Context switching
Figure 3.17: Processes on the ready and wait queues
Figure 3.18: Two different priority queues
MOV EAX, items ; load from items into register EAX
INC EAX ; increment register EAX
MOV items, EAX ; store from register EAX back to items

Figure 4.1: Code to increment counter
MOV EAX, items ; load from items to register EAX
DEC EAX      ; decrement register EAX
MOV items, EAX ; store from register EAX back to items

Figure 4.2: Code to decrement counter
beginning section
Entry protocol for critical section
critical section
Exit protocol for critical section
remainder section

Figure 4.3: General structure of a cooperating program
beginning section

WHILE (guard == 1)
    DO nothing
ENDWHILE

beginning section

guard = 1

critical section

guard = 0

remainder section

Figure 4.4: First attempt at solution for two processes
test:  MOV EAX, guard
   CMP EAX, #1
   JE   test:
   MOV guard, #1

Figure 4.5: Compiled version of entry protocol
Process 0
beginning section

WHILE (turn != 0)
  DO nothing
ENDWHILE

critical section

turn = 1

remainder section

Process 1
beginning section

WHILE (turn != 1)
  DO nothing
ENDWHILE

critical section

turn = 0

remainder section

Figure 4.6: Algorithms using turn
beginning section

WHILE (flag[1] == 1)
    DO nothing
ENDWHILE
flag[0] = 1

critical section

flag[0] = 0

remainder section

Figure 4.7: Algorithm using two flags
beginning section

flag[0] = 1
WHILE (flag[1] == 1)
    DO nothing
ENDWHILE

critical section

flag[0] = 0

remainder section

Figure 4.8: Setting the flag before testing
beginning section

flag[0] = 1
turn = 1
WHILE ((flag[1] == 1) AND (turn == 1))
  DO nothing
ENDWHILE

critical section

flag[0] = 0

remainder section

Figure 4.9: Algorithm for mutual exclusion
Figure 4.10: Entry protocols for both processes
Figure 4.11: Eight contending processes
beginning section

REPEAT
  flag[4] = want-in
  p = turn
  WHILE (p ≠ 4)
    IF (flag[p] == idle) THEN
      p++ MOD 8
    ELSE
      p = turn
    ENDIF
  ENDWHILE

flag[4] = in-cs

j = 0
WHILE ((j < 8) AND ((j == 4) OR (flag[j] ≠ in-cs)))
  j ++
ENDWHILE

UNTIL ((j == 8) AND ((turn == 4) OR (flag[turn] == idle)))

turn = 4

critical section

flag[4] = idle

remainder section

Figure 4.12: Eisenberg and McGuire algorithm
beginning section

    choosing[i] = TRUE
    counter++
    number[i] = counter
    choosing[i] = FALSE

FOR j = 0 TO n - 1 DO
    WHILE (choosing[j] == TRUE)
        DO nothing
    ENDWHILE
    WHILE ((number[j] ≠ 0) AND (number[j] < number[i]))
        DO nothing
    ENDWHILE
ENDFOR

critical section

    number[i] = 0

remainder section

Figure 4.13: Lamport's algorithm
#define SIGINT 2    /* interrupt, generated from terminal */
#define SIGILL 4    /* illegal instruction */
#define SIGABRT 6    /* abort process */
#define SIGFPE 8     /* floating point exception */
#define SIGKILL 9    /* kill a process */
#define SIGUSR1 10   /* user defined signal 1 */
#define SIGSEGV 11   /* segmentation violation */
#define SIGUSR2 12   /* user defined signal 2 */
#define SIGALRM 14   /* alarm clock timeout */
#define SIGCHLD 17   /* sent to parent on child exit */
#define SIGXCPU 24   /* cpu time limit exceeded */

Figure 5.1: A selection of signal values
Figure 5.2: Changing the signal mask
struct signal_struct{
    atomic_t count;
    struct k_sigaction action[32];
    spinlock_t siglock;
};

Figure 5.3: Data structure tracking signal handlers
WHILE (Test_and_set (Key))
DO nothing
ENDWHILE

**Figure 5.4:** Entry protocol using test and set
REPEAT
  Local = 1
  Exchange(Local, Key)
UNTIL Local == 0

**Figure 5.5**: Entry protocol using exchange
WAIT(Guard)
critical section
SIGNAL(Guard)

beginning section
remainder section

Figure 5.6: Semaphore for mutual exclusion
Figure 5.7: Successive states of a mutual exclusion semaphore
Figure 5.8: Semaphore for synchronisation
Figure 5.9: An alternative for synchronisation
Figure 5.10: Semaphore for resource allocation
Figure 5.11: Circular buffer with eight slots
Produce an item
WAIT(SlotFree)
Put item in buffer at NextIn
NextIn++
SIGNAL(ItemAvailable)
WAIT(ItemAvailable)
Get item from buffer at NextOut
NextOut++
SIGNAL(SlotFree)
Consume the item

Figure 5.13: Algorithm for consumer
Produce an item
WAIT(SlotFree)
WAIT(Guard)
Put item in buffer at NextIn
NextIn++
SIGNAL(Guard)
SIGNAL(ItemAvailable)

Figure 5.14: Algorithm for one of many producers
WAIT(ItemAvailable)
WAIT(Guard)
Get item from buffer at NextOut
NextOut++
SIGNAL (Guard)
SIGNAL(SlotFree)
Consume the item

Figure 5.15: Algorithm for one of many consumers
WAIT(Guard)
readers++
IF (readers == 1) THEN
    WAIT(Writing)
ENDIF
SIGNAL(Guard)

**Figure 5.16**: Entry protocol for a reader
WAIT(Guard)
readers--
IF (readers == 0) THEN
    SIGNAL(Writing)
ENDIF
SIGNAL(Guard)

Figure 5.17: Exit protocol for a reader
Figure 5.18: Solution 2
Figure 5.19: Solution 3
BEGIN
WAIT(Guard)
wr++
IF (ww == 0) THEN
    ar++
    SIGNAL(R)
ENDIF
SIGNAL(Guard)
WAIT (R)
END

**Figure 5.20:** Reader prologue code
WAIT(Guard)
wr++
IF (ww > 0) THEN
   WAIT(R)
ELSE
   ar++
ENDIF
SIGNAL(Guard)

Figure 5.21: Incorrect reader prologue
WAIT (Guard)
ar--
wr--
IF (ar == 0) THEN
    IF (ww > 0) THEN
        SIGNAL(w)
    ENDIF
ENDIF
ENDIF
SIGNAL (Guard)

Figure 5.22: Reader epilogue code
\begin{verbatim}
WAIT(Guard)
ww++
IF (ar == 0) AND (ww == 1) THEN
    SIGNAL(W)
ENDIF
SIGNAL(Guard)
WAIT(W)
\end{verbatim}

\textbf{Figure 5.23}: Writer prologue code
Figure 5.24: Writer epilogue code

```
WAIT(Guard)
ww--
WHILE (ar < wr)
ar++
    SIGNAL(R)
ENDWHILE
IF (ar == 0) AND (ww > 0) THEN
    SIGNAL(W)
ENDIF
SIGNAL(Guard)
```
Figure 5.25: Solution 4
WAIT (Guard)
ar--
wr--
IF (ar == 0) THEN
    WHILE (pw < ww)
        pw++
        SIGNAL(W)
    ENDWHILE
ENDIF
SIGNAL (Guard)

Figure 5.26: Reader epilogue code
WAIT(Guard)
ww++
IF (ar == 0) THEN
  pw++
  SIGNAL(W)
ENDIF
SIGNAL(Guard)
WAIT(W)
WAIT(Writing)

Figure 5.27: Writer prologue code
SIGNAL(Writing)
WAIT(Guard)
pw--
ww--
IF (pw == 0) THEN
  WHILE (ar < wr)
    ar++
    SIGNAL(R)
  ENDWHILE
ENDIF
ENDWHILE
SIGNAL(Guard)

Figure 5.28: Writer epilogue code
Figure 5.29: Timing relationship of locks and semaphores
Figure 5.30: Data structure representing a semaphore set

```c
struct semid_ds{
    struct ipc_perm sem_perm;
    struct sem *sem_base;
    struct sem_queue *sem_pending;
};
```
struct ipc_perm{
    kernel_key_t key; /* user supplied key */
    kernel_uid_t uid; /* owner's user id */
    kernel_gid_t gid; /* owner's group id */
    kernel_mode_t mode; /* access modes */
};

Figure 5.31: Data structure controlling access to a semaphore
struct sem{
    int semval; /* current value */
    int sempid; /* process which last operated on sem */
};

Figure 5.32: Data structure representing an individual semaphore
Figure 5.33: System V semaphore layout
struct sem_queue{
    struct sem_queue  *next;
    struct wait_queue *sleeper;
    struct semid_ds   *sma;
    struct sembuf     *sops;
    int                nsops;
};

**Figure 5.34**: Data structure representing a waiting process
Figure 5.35: Processes waiting on the semaphore set
Figure 5.36: Eventcount for synchronisation

<table>
<thead>
<tr>
<th>Producer</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADVANCE(E)</td>
<td>AWAIT(E, i)</td>
</tr>
<tr>
<td></td>
<td>i++</td>
</tr>
</tbody>
</table>
beginning section

AWAIT(E, TICKET(S))

critical section

ADVANCE(E)

remainder section

**Figure 5.37**: Eventcount and sequencer for mutual exclusion
Figure 5.38: Event counts with one producer, one consumer
Figure 6.1: Outline of message passing
Operating Systems with Linux © John O'Gorman 2001

Figure 6.2: A message queue
msqid_ds{
    struct ipc_perm msg_perm; /* access permissions */
    struct msg *msg_first; /* first message on queue */
    struct msg *msg_last; /* last message on queue */
    struct wait_queue *wwait; /* blocked writing threads */
    struct wait_queue *rwait; /* blocked reading threads */
    unsigned short msg_qnum; /* number of messages on queue */
};

Figure 6.3: Data structure representing a message queue
struct msg{
    struct msg *msg_next; /* next message on queue */
    long msg_type; /* as specified by sender */
    char *msg_spot; /* message text address */
    time_t msg_stime; /* msgsnd time */
    short msg_ts; /* message text size */
};

Figure 6.4: Structure of an actual message
Figure 6.5: System V message queue layout
struct msgbuf{
    long mtype;    /* message type */
    char mtext[];  /* message contents */
};

Figure 6.6: Data structure representing a message
shared struct{
    int writers;
    int readers;
}shared_counts;

shared char writelock;

/* reader code */
critical region shared_counts{
    await (writers == 0);
    readers++;
}

    /* READ */
critical region shared_counts{
    readers--;
}

/* writer code */
critical region shared_counts{
    writers++;
    await (readers == 0);
}
critical region writelock{

    /* WRITE */
}
critical region shared_counts{
    writers--;
}

Figure 6.7: Readers/writers using conditional critical regions
CWAIT(conditionvar)
  LOCK
  Mark process unrunnable
  Queue the process on the condition variable
  Release mutual exclusion on the monitor
  UNLOCK
  Call context switcher

Figure 6.8: Implementation of CWAIT
CSIGNAL(conditionvar)
   LOCK
   IF there is a process waiting on conditionvar THEN
      Mark one runnable
      Mark current process unrunnable
      Transfer mutual exclusion on the monitor to selected process
      UNLOCK
      Call context switcher
   ELSE
      UNLOCK
      Return
   ENDIF

Figure 6.9: Implementation of CSIGNAL
Figure 6.10: An illustration of a monitor
MONITOR allocator{

boolean busy = FALSE;
condition free;

reserve()
{
    while (busy == TRUE)
    CWAIT(free);
    busy = TRUE;
}

release()
{
    busy = FALSE;
    CSIGNAL(free);
}
}

Figure 6.11: Monitor to allocate a single resource
MONITOR buffer{

    int count = 0;
    condition spaceavail;
    condition itemavail;

    producer(){
        while (count == MAX)
        CWAIT(spaceavail);
        /* Put item in buffer */
        count++;
        CSIGNAL(itemavail);
    }

    consumer(){
        while (count == 0)
        CWAIT(itemavail);
        /* Get item from buffer */
        count--;
        CSIGNAL(spaceavail);
    }
}

Figure 6.12: Monitor to manage a bounded buffer
MONITOR readers-writers{

int readers = 0, writers = 0;
boolean busy-writing = FALSE;
condition readers-waiting, writers-waiting;

StartRead()
{
    while (writers > 0)
        CWAIT(readers-waiting);
    readers++;
    CSIGNAL(readers-waiting)
}

EndRead()
{
    readers--;
    if (readers == 0)
        CSIGNAL(writers-waiting)
}

StartWrite()
{
    writers++;
    while ((busy-writing == TRUE) || (readers > 0))
        CWAIT(writers-waiting);
    busy-writing = TRUE
}

EndWrite()
{
    busy-writing = FALSE;
    writers--;
    if (writers > 0)
        CSIGNAL(writers-waiting);
    else
        CSIGNAL(readers-waiting);
}
}

Figure 6.13: Monitor to implement reader/writer interlock
Figure 7.1: A resource allocation graph
Figure 7.2: A resource allocation graph for four processes
Figure 7.3: Safe, unsafe, and deadlocked system
Figure 7.4: A claims graph
Figure 7.5: One resource allocated
IF (Request > Need[i]) THEN
    Error—illegal request
ENDIF
IF (Request > Available) THEN
    Wait
ENDIF

Available = Available - Request
Allocation[i] = Allocation[i] + Request
Need[i] = Need[i] - Request

Check if this is a safe state
IF Safe THEN
    Allocate resources
ELSE
    Restore state
    Wait
ENDIF

Figure 7.6: The banker's algorithm
Work[r] = Available[r]
Finish[p] = FALSE (all elements)

REPEAT
  Found = FALSE
  i = 0
  REPEAT
    IF (Finish[i] == FALSE) AND (Need[i] ≤ Work) THEN
      Finish[i] = TRUE
      Work = Work + Allocation[i]
      Found = TRUE
    ENDIF
    i++
  UNTIL (Found == TRUE) OR (i == p)
UNTIL Found == FALSE

Safe = TRUE
FOR i = 0 TO p - 1
  IF (Finish[i] == FALSE) THEN
    Safe = FALSE
  ENDIF
ENDFOR

Figure 7.7: Safety algorithm
### Figure 7.8: State of the system before request

<table>
<thead>
<tr>
<th></th>
<th>Max A</th>
<th>Max B</th>
<th>Allocation A</th>
<th>Allocation B</th>
<th>Need A</th>
<th>Need B</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀</td>
<td>7</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>P₁</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>P₂</td>
<td>9</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>P₃</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
Figure 7.9: Application of the safety algorithm

<table>
<thead>
<tr>
<th>OUTER LOOP</th>
<th>INNER LOOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Work = 3,5</td>
<td>i = 0 Need[0] &gt; Work</td>
</tr>
<tr>
<td>2 Work = 5,5</td>
<td>i = 0 Need[0] &gt; Work</td>
</tr>
<tr>
<td></td>
<td>i = 1 Finish[1] = TRUE</td>
</tr>
<tr>
<td>3 Work = 7,6</td>
<td>i = 0 Need[0] &lt; Work Finish[0] = TRUE</td>
</tr>
<tr>
<td>4 Work = 7,7</td>
<td>i = 0 Finish[0] = TRUE</td>
</tr>
<tr>
<td></td>
<td>i = 1 Finish[1] = TRUE</td>
</tr>
<tr>
<td>5 Work = 10,7</td>
<td>i = 0 Finish[0] = TRUE</td>
</tr>
<tr>
<td></td>
<td>i = 1 Finish[1] = TRUE</td>
</tr>
<tr>
<td></td>
<td>i = 2 Finish[2] = TRUE</td>
</tr>
<tr>
<td></td>
<td>i = 3 Finish[3] = TRUE</td>
</tr>
</tbody>
</table>
### Available

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Max</th>
<th>Allocation</th>
<th>Need</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>(P_0)</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>(P_1)</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>(P_2)</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>(P_3)</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

**Figure 7.10:** State of the system reflecting the request
Figure 7.11: Resource allocation graph
Figure 7.12: Wait-for graph
FOR $i = 0$ TO MaxProc − 1
   IF (Allocation[i] == 0) THEN
      Finish[i] = TRUE
   ELSE
      Finish[i] = FALSE
   ENDIF
ENDFOR

REPEAT
   Found = FALSE
   i = 0
   REPEAT
      IF (Finish[i] == FALSE) AND (Request[i] ≤ Work) THEN
         Work = Work + Allocation[i]
         Finish[i] = TRUE
         Found = TRUE
      ENDIF
      i++
   UNTIL (Found == TRUE) OR (i == p)
UNTIL (Found == FALSE)

Deadlocked = FALSE
FOR $i = 0$ TO MaxProc − 1
   IF (Finish[i] == FALSE) THEN
      Deadlocked = TRUE
   ENDIF
ENDFOR

Figure 7.13: Algorithm to detect deadlock
Figure 7.14: A system which is not deadlocked
Figure 7.15: Application of the deadlock detection algorithm
Figure 7.16: A deadlocked system

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀</td>
<td>A: 0, B: 1</td>
</tr>
<tr>
<td>P₁</td>
<td>A: 2, B: 0</td>
</tr>
<tr>
<td>P₂</td>
<td>A: 3, B: 0</td>
</tr>
<tr>
<td>P₃</td>
<td>A: 2, B: 1</td>
</tr>
</tbody>
</table>

Available

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
**Figure 8.1**: Hardware memory
Figure 8.2: The memory pyramid
Figure 8.3: Program as loaded by relocating loader
Figure 8.4: Virtual memory
Figure 8.5: Address modification in CPU
Figure 8.6: Protection and address modification
Figure 8.7: Compaction
<table>
<thead>
<tr>
<th>Segment (s)</th>
<th>Offset (o)</th>
</tr>
</thead>
</table>

**Figure 8.8**: A logical address
Figure 8.9: Address translation with segment table
| r | w | e | base address | length |

**Figure 8.10:** A segment table entry
Figure 8.11: Searching a segmentation cache
Present segment number to associative store
IF no match, THEN
  Use segment number as index into segment table
  IF not present, THEN
    Segment fault-fetch from secondary memory
    Update segment table
  ENDIF
  Update associative store
ENDIF
Add offset to base address

Figure 8.12: Segmentation algorithm
struct page{
    struct page  *next;
    struct inode *inode;
    unsigned long offset;
    struct page  *next_hash;
    atomic_t     count;
};

Figure 8.13: Data structure representing a physical page
Figure 8.14: Tracking free physical memory
Figure 8.15: Linux page table structure
| high order 20 bits of page frame address | page table bits |

**Figure 8.16:** Format of page table entry
Figure 8.17: Address translation with cache and page table
int array[128][128];

for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        array[i][j] = 0;

Figure 8.18: Program to initialise an array
struct mm_struct{
    struct vm_area_struct *mmap;
    pgd_t *pgd;
    int count;
    unsigned long start_code, end_code;
    unsigned long start_data, end_data;
    unsigned long start_stack;
};

**Figure 8.19:** The root of the memory management data structures
struct vm_area_struct{
    struct mm_struct *vm_mm;
    unsigned long vm_start;
    unsigned long vm_end;
    struct vm_area_struct *vm_next;
    unsigned short vm_flags;
    struct vm_operations_struct *vm_ops;
    unsigned long vm_offset;
    struct file *vm_file;
};

Figure 8.20: Data structure controlling a region
Figure 8.21: Memory management data structures for a process
Figure 8.22: Possible values for the flags field

#define VM_READ 0x0001
#define VM_WRITE 0x0002
#define VM_EXEC 0x0004
#define VM_SHARED 0x0008
#define VM_LOCKED 0x2000
struct vm_operations_struct{
    void    (*open)();
    void    (*close)();
    void    (*unmap)();
    int     (*swapout)();
    pte_t   (*swaipin)();
};
Figure 8.24: Pages actually in memory
Figure 8.25: Example mapping of three regions
Figure 8.26: Relationship between page frames and page faults
REPEAT
   Examine the reference bit in the page table entry for pagenumber
   IF (reference bit == 1) THEN
      reference bit = 0
   ELSE
      remove page
   ENDIF
   pagenumber = (pagenumber + 1) MOD size of page table
UNTIL (required number of pages removed)

Figure 8.27: Algorithm for NRU replacement
Figure 8.28: Effect of thrashing on CPU utilisation
Figure 8.29: Effect of window size on working set
Logical address

<table>
<thead>
<tr>
<th>Segment (s)</th>
<th>Page (p)</th>
<th>Offset (o)</th>
</tr>
</thead>
</table>

**Figure 8.30:** An address interpreted as segment, page, offset
Split the program address into s, p, o
Use s to index into the segment table
IF the presence bit in the segment descriptor is cleared
   (This means there are no pages for this segment in memory,
    nor is there a page table for this segment)
THEN create a new (empty) page table for this segment,
   enter its address in the segment table,
   and set the corresponding presence bit
ENDIF
Extract the address of the page table
Use p to index the page table
IF the presence bit in the corresponding entry
   in the page table is cleared (This means that
   the page is not in memory)
THEN fetch the page from backing store,
   enter its address in the pagetable,
   and set the corresponding presence bit
ENDIF
Extract the page frame address f
Add f to o to give the required location

Figure 8.31: Algorithm for paged segmentation
Figure 8.32: Address translation with paged segmentation
Figure 9.1: Overview of I/O processing
Figure 9.2: A tree-structured directory
struct dirent{
    long int d_ino;    /* file number of this entry */
    unsigned short int d_namlen; /* length of string in d_name */
    unsigned short int d_reclen; /* length of this record */
    char d_name[];        /* actual filename */
};

Figure 9.3: Structure of a directory entry
Figure 9.4: A directory with links
Figure 9.5: How a hard link is implemented
Figure 9.6: How a symbolic link is implemented
Figure 9.7: A directory structure with cycles
struct inode{
    struct list_head i_hash; /* link on hash list */
    struct list_head i_list; /* link on inode list */
    unsigned long i_ino; /* inode number */
    unsigned short i_count; /* count of users */
    kdev_t i_dev; /* device number */
    umode_t i_mode; /* type and permissions */
    nlink_t i_nlink; /* hard links */
    uid_t i_uid; /* owner */
    gid_t i_gid; /* owner's group */
    kdev_t i_rdev; /* if a device file */
    off_t i_size; /* in bytes */
    time_t i_atime; /* access */
    time_t i_mtime; /* change of contents */
    time_t i_ctime; /* inode modified */
    unsigned long i_blksize; /* size of block */
    unsigned long i_blocks; /* number of blocks */
    struct semaphore i_sem; /* mutex on inode */
    struct inode_operations *i_op; /* operations vector */
    struct super_block *i_sb; /* if a mount point */
    struct wait_queue *i_wait; /* wait queue */
    struct file_lock *i_flock; /* locks on this file */
    struct vm_area_struct *i_mmap; /* memory region */
    struct page *i_pages; /* pages in memory */
    union{
        struct pipe_inode_info pipe_i;
        struct ext2_inode_info ext2_i;
        struct msdos_inode_info msdos_i;
        struct nfs_inode_info nfs_i;
        struct socket socket_i;
    }u;
};

Figure 9.8: Inode for an open file
**Figure 9.9:** Inodes on LRU and hash lists
#define S_IFSOCK 0140000 /* socket */
#define S_IFLNK 0120000 /* symbolic link */
#define S_IFREG 0100000 /* regular */
#define S_IFBLK 0060000 /* block special */
#define S_IFDIR 0040000 /* directory */
#define S_IFCHR 0020000 /* character special */
#define S_IFIFO 0010000 /* fifo */
#define S_ISUID 0004000 /* set user id on execution */
#define S_ISGID 0002000 /* set group id on execution */
#define S_IRUSR 0000400 /* read permission: owner */
#define S_IWUSR 0000200 /* write permission: owner */
#define S_IXUSR 0000100 /* execute/search permission: owner */
#define S_IRGRP 0000040 /* read permission: group */
#define S_IWGRP 0000020 /* write permission: group */
#define S_IXGRP 0000010 /* execute/search permission: group */
#define S_IROTH 0000004 /* read permission: other */
#define S_IWOTH 0000002 /* write permission: other */
#define S_IXOTH 0000001 /* execute/search permission: other */

Figure 9.10: Mode bit values
struct inode_operations{
    struct file_operations *default_file_ops;
    int (*create)();
    struct dentry (*lookup)();
    int (*link)();
    int (*unlink)();
    int (*symlink)();
    int (*mkdir)();
    int (*rmdir)();
    int (*mknod)();
    int (*rename)();
    int (*readlink)();
    struct dentry (*followlink)();
    int (*readpage)();
    int (*writepage)();
    int (*bmap)();
};

Figure 9.11: Some functions of the inode interface
Figure 9.12: An element in the cache of recent lookups

```c
struct dentry{
    struct inode  *d_inode;
    struct list_head d_hash;
    struct list_head d_lru;
    unsigned char   d_iname[];
};
```
Figure 9.13: Directory name lookup cache
Figure 9.14: A protection bitmap
struct files_struct{
    atomic_t    count;
    fd_set     close_on_exec;
    fd_set     open_fds;
    struct file *fd_array[];
};

Figure 9.15: Definition of file descriptor table
**Figure 9.16**: Descriptor and open file data structure
struct file{
    struct file *f_next; /* next struct file */
    struct dentry *f_dentry; /* representing this stream */
    struct file_operations *f_op; /* operations on this stream */
    mode_t f_mode; /* open mode (read/write) */
    loff_t f_pos; /* current position in file */
    unsigned short f_count; /* number of threads sharing */
    int f_owner; /* where SIGIO should be sent */
};

Figure 9.17: Structure representing an open stream
Figure 9.18: File opened by one user
struct file_operations{
    loff_t     (*llseek) ();
    ssize_t    (*read) ();
    ssize_t    (*write) ();
    int        (*readdir) ();
    unsigned int (*poll) ();
    int        (*ioctl) ();
    int        (*mmap) ();
    int        (*open) ();
    void       (*release)();
};

Figure 9.19: Operations on open files
Set up file descriptor, link to struct file
Search the directory name cache
IF not found THEN
    lookup() (from directory i_op)
    IF not found THEN
        errno = EFAULT
        Return (-1)
    ENDIF
    Create appropriate entry in name cache
    Create inode in memory
    Link dentry to inode
ENDIF
Set up f_op field in struct file
open() (from f_op)
IF no permission THEN
    errno = EACCES
    Return (-1)
ENDIF
Link struct file to dentry
Increment i_count
Return (file descriptor)
Figure 9.21: File opened by two users
Map fileid onto stream
IF not open, THEN
    errno = errornumber
    Return (-1)
ENDIF
Check mode and length against stream characteristics
IF not compatible THEN
    errno = errornumber
    Return (-1)
ENDIF
Call stream specific function with appropriate parameters
IF not successful THEN
    errno = errornumber
    Return (-1)
ENDIF
Return (0)

Figure 9.22: Generic I/O algorithm
Use stream number to index into the file descriptor table
IF not assigned, THEN
   errno = EBADF
   Return (-1)
ENDIF
Follow the pointer from descriptor to struct file
Examine f_mode
IF request not compatible with open mode THEN
   errno = EBADF
   Return (-1)
ENDIF
IF request is for simple I/O THEN
   Call appropriate function in f_op
ELSE
   Follow f_dentry pointer to struct inode
   Call appropriate function in i_op
ENDIF
IF not successful THEN
   errno = EIO
   Return (-1)
ELSE
   Return (0)
ENDIF

Figure 9.23: Generic algorithm for Unix I/O procedure
struct pollfd{
    int fd;            /* the descriptor we are interested in */
    short int events; /* flag specifying events of interest */
    short int revents; /* flag specifying events which have occurred */
};

Figure 9.24: Data structure for polling a descriptor
**Figure 9.25:** Several processes waiting on a device
Figure 9.26: Structure representing a lock on a file
Figure 9.27: A lock request which blocks
Figure 9.28: How locks are recorded
Check parameters against device characteristics
Return if error
IF buffer_empty THEN
    Call device driver (for bufferfull)
    Process any errors
ENDIF
Transfer data from buffer
Return

Figure 9.29: A buffered input procedure
Figure 9.30: A buffer head

```c
struct buffer_head{
    unsigned long b_blocknr; /* block number */
    kdev_t b_dev; /* device */
    struct buffer_head *b_next; /* hash list */
    unsigned long b_state; /* state bitmap */
    struct buffer_head *b_next_free; /* free list */
    char *b_data; /* pointer to data block */
    struct wait_queue *b_wait; /* processes waiting */
};
```
Figure 9.31: Bits representing the state of a buffer

```c
#define BH_Uptodate 0 /* contains valid data */
#define BH_Dirty 1 /* is dirty */
#define BH_Lock 2 /* is locked */
#define BH_Req 3 /* has been invalidated */
```
Figure 9.32: Buffer cache
struct file_system_type{
    const char *name;
    struct super_block *(*read_super)();
    struct file_system_type *next;
};

Figure 10.1: An entry in the linked list of file systems
Figure 10.2: Layout of a disk
Figure 10.3: Contiguous allocation
Figure 10.4: Linked allocation
Figure 10.5: Example of a file map
Figure 10.6: Layout of a partition with Ext2
Figure 10.7: A block group on disk
struct ext2_super_block{
    u32 s_inodes_count;    /* Inodes count */
    u32 s_blocks_count;    /* Blocks count */
    u32 s_free_blocks_count; /* Free blocks count */
    u32 s_free_inodes_count; /* Free inodes count */
    u32 s_first_data_block; /* First data block */
    u32 s_log_block_size;   /* Block size */
    s32 s_log_frag_size;    /* Fragment size */
    u32 s_blocks_per_group; /* # Blocks per group */
    u32 s_inodes_per_group; /* # Inodes per group */
    u16 s_inode_size;       /* size of inode structure */
    u16 s_block_group_nr;   /* number of this block group */
};

Figure 10.8: An Ext2 super block
Figure 10.9: Structure of a block group descriptor

```c
struct ext2_group_desc{
    u32 bg_block_bitmap;    /* Blocks bitmap block */
    u32 bg_inode_bitmap;    /* Inodes bitmap block */
    u32 bg_inode_table;     /* Inodes table block */
    u16 bg_free_blocks_count; /* Free blocks count */
    u16 bg_free_inodes_count; /* Free inodes count */
    u16 bg_used_dirs_count; /* Directories count */
};
```
```
struct ext2_inode{
    u16 i_mode;    /* File mode */
    u16 i_uid;     /* Owner uid */
    u32 i_size;    /* Size in bytes */
    u32 i_atime;   /* Access time */
    u32 i_ctime;   /* Creation time */
    u32 i_mtime;   /* Modification time */
    u16 i_gid;     /* Group id */
    u16 i_links_count; /* Links count */
    u32 i_blocks;  /* Blocks count */
    u32 i_block[15]; /* Pointers to blocks */
    u32 i_faddr;   /* Fragment address */
    u8  l_i_frag;  /* Fragment number */
    u8  l_i_fsize; /* Fragment size */
};
```

**Figure 10.10**: Structure of an Ext2 disk inode
Figure 10.11: Allocation information in an inode
**Figure 10.12**: Single indirect block
Inode

Figure 10.13: Double indirect block
Figure 10.14: Fragments of two files in a block
struct ext2_dir_entry{
    u32    inode;   /* inode number of the entry */
    u16    rec_len; /* total length of entry */
    u16    name_len; /* length of name */
    char   name[]    /* file name */
};

Figure 10.15: An Ext2 directory entry
Figure 10.16: File systems as they exist on disk
File system A

/ users include

sys network I/O

ufs cdfs

Figure 10.17: Directory structure after attaching file system B
struct super_block{
    struct list_head s_list;
kdev_t s_dev;
    struct file_system_type *s_type;
    struct super_operations *s_op;
    struct dentry *s_root;
    union{
        struct ext2_sb_info ext2_sb;
        struct msdos_sb_info msdos_sb;
        struct nfs_sb_info nfs_sb;
    }u;
};

Figure 10.18: Data structure representing a mounted file system
struct super_operations{
    void (*read_inode)();
    void (*write_inode)();
    void (*put_inode)();
    void (*put_super)();
    void (*write_super)();
    int (*statfs)();
};

Figure 10.19: Operations on a file system
struct ext2_sb_info{
    struct ext2_super_block *s_es;
    struct buffer_head **s_group_desc;
    struct buffer_head *s_inode_bitmap[];
    struct buffer_head *s_block_bitmap[];
};

Figure 10.20: Superblock for an Ext2 file system
Figure 10.21: Data structures involved in mounting a file system
struct ext2_inode_info{
    u32 i_data[15];
    u32 i_faddr;
    u8  i_frag_no;
    u8  i_frag_size;
    u32 i_file_acl;
    u32 i_dir_acl;
    u32 i_block_group;
};

Figure 10.22: Ext2 file system inode data in memory
Figure 11.1: Calling a Unix style driver
Figure 11.2: Device driver as a process
struct device_struct{
    const char *name;
    struct file_operations *fops;
};

static struct device_struct chrdevs[];
static struct device_struct blkdevs[];

Figure 11.3: The character and block device switches
Figure 11.4: An entry in the character device switch
IF device busy THEN
    Sleep
ENDIF
Set up the I/O operation
Sleep
IF error THEN
    return value = errornumber
ELSE
    Transfer data to destination
    return value = success
ENDIF
Return (return value)

Figure 11.5: Algorithm for reading from a character device
Figure 11.6: Request block for block devices

```c
struct request{
    kdev_t    rq_dev;    /* device identifier */
    int       cmd;       /* READ or WRITE */
    unsigned long sector; /* start sector */
    unsigned long nr_sectors; /* number of sectors */
    char       *buffer;   /* buffer address */
    struct request *next; /* next IORB */
};
```
Put struct request on device queue
IF device idle, THEN
    Select struct request from queue
    IF (cmd == WRITE) THEN
        Transfer data from user space to device memory
    ENDIF
    Set up the device for specified operation
ENDIF
Return to caller.

Figure 11.7: Algorithm for strategy routine
Check for error
IF (cmd == READ) THEN
    Transfer data from device memory to user space
ENDIF
Move user process from wait queue to run queue
IF request queue not empty THEN
    Select struct request from queue
    IF (cmd == WRITE) THEN
        Transfer data from user space to device memory
        ENDIF
    Set up the device for specified operation
ENDIF
IF device busy THEN
    Sleep
ENDIF
Transfer data from user buffer to device memory
Set up the device for writing
Sleep
IF error THEN
    return value = erornumber
ELSE
    return value = success
ENDIF
Return (return value)
Figure 11.10: A basic view of a stream
Figure 11.11: Data structures constituting a minimal stream
Figure 11.12: A many-to-one multiplexor
Figure 11.13: A one-to-many multiplexor
Figure 11.14: A many-to-many multiplexor
Figure 11.15: Networking multiplexor
struct termios{
    tcflag_t c_iflag;
    tcflag_t c_oflag;
    tcflag_t c_cflag;
    tcflag_t c_lflag;
    cc_t c_cc[];
};

Figure 11.16: Data structure controlling terminal characteristics
Figure 11.17: Logging on using a pseudo terminal
Figure 12.1: Setting up a pipe for interprocess communication
Figure 12.2: Data structure representing a pipe

```
struct pipe_inode_info{
    struct wait_queue *wait /* for blocked threads */
    char *base /* buffer */
    unsigned int start /* next byte to read */
    unsigned int readers /* number of readers */
    unsigned int writers /* number of writers */
};
```
Figure 12.3: Representation of a pipe
#define AF_UNSPEC 0 /* Unspecified */
#define AF_UNIX 1 /* Unix domain sockets */
#define AF_INET 2 /* Internet IP Protocol */
#define AF_IPX 4 /* Novell IPX */
#define AF_APPLE TALK 5 /* Appletalk */
struct net_proto_family{
    int family;
    int (*create)();
};

Figure 12.5: Data structure representing a protocol family
Figure 12.6: Domains registered in the kernel
struct socket{
    socket_state state;  /* current state of connection */
    struct proto_ops *ops;  /* domain specific functions */
    struct sock *data;  /* domain specific data */
    struct wait_queue **wait;  /* waiting processes */
    short type;  /* SOCK_STREAM, etc. */
};

Figure 12.7: Data structure representing a socket
typedef enum{
    SS_FREE,       /* not allocated        */
    SS_UNCONNECTED, /* unconnected to any socket */
    SS_CONNECTING,  /* in process of connecting */
    SS_CONNECTED,   /* connected to socket */
    SS_DISCONNECTING /* in process of disconnecting */
} socket_state;

Figure 12.8: State values for a socket
struct proto_ops{
    int family;
    int (*bind)();
    int (*connect)();
    int (*socketpair)();
    int (*accept)();
    int (*listen)();
    int (*shutdown)();
    int (*setsockopt)();
    int (*getsockopt)();
    int (*sendmsg)();
    int (*recvmsg)();
};

Figure 12.9: Data structure representing a protocol
Figure 12.10: Data structures after socket is allocated
struct sockaddr{
    sa_family_t sa_family; /* address family, AF_xxx */
    char        sa_data[]; /* protocol address */
};

Figure 12.11: Generic format of a socket address
struct msghdr{
    void       *msg_name;  /* name of destination socket */
    int        msg_name_len;  /* length of name */
    struct iovec *msg_iov;  /* array of data buffers */
    int        msg_iovlen;  /* number of buffers */
};

Figure 12.12: Message header


Figure 12.13: Domain-specific control block

```c
struct sock{
    struct sock    *next;
    int            rcvbuf;
    struct sk_buff_head receive_queue;
    int            sndbuf;
    struct sk_buff_head write_queue;
    struct proto   *prot;
    struct unix_opt af_unix;
    struct socket  *socket;
};
```
Figure 12.14: Hash chain of Unix domain control blocks
struct proto{
    void (*close)();
    int (*connect)();
    struct sock* (*accept)();
    int (*poll)();
    void (*shutdown)();
    int (*setsockopt)();
    int (*getsockopt)();
    int (*sendmsg)();
    int (*recvmsg)();
    int (*bind)();
};

Figure 12.15: A protocol specific structure
struct unix_opt{
    struct unix_address *addr;
    struct sock *other;
};

Figure 12.16: Links for a Unix specific socket
Figure 12.17: Structure containing address of a Unix domain socket

```c
struct sockaddr_un{
    unsigned short sun_family; /* address family, AF_UNIX */
    char sun_path[104]; /* pathname */
};
```
Figure 12.18: A pair of connected sockets
struct sk_buff{
    struct sk_buff *next; /* Next buffer in list */
    struct sock *sk;    /* Socket we are owned by */
    unsigned long len;  /* Length of actual data */
    unsigned int truesize; /* Buffer size */
    unsigned char *head; /* Head of buffer */
    unsigned char *data /* Data head pointer */
    unsigned char *tail /* Tail pointer */
    unsigned char *end   /* End pointer */
};

Figure 12.19: Buffer used for communication over sockets
Figure 12.20: Data part of an sk_buff
Figure 13.1: Networked computers
Figure 13.2: Workstations connected on a LAN
Figure 13.3: Diskless workstations on a LAN
Figure 13.4: Processor pool model
Figure 13.5: Combined model
Figure 13.6: Part of the Internet name space
Figure 13.7: Using a name server
Figure 13.8: Translating an Internet name
Figure 13.9: Overview of CORBA
Figure 13.10: The distributed computing environment
Figure 14.1: Data structure describing a network interface

```c
struct device{
    char *name;
    struct device *next;
    int (*init)();
    unsigned short type;
    unsigned char dev_addr[];
    int (*hard_start_xmit)();
};
```
struct sock{
    struct options *opt;
    struct sock *next;
    struct sk_buff *partial;
    struct sk_buff_head receive_queue, write_queue;
    struct proto *prot;
    u32 daddr;
    u32 rcv_saddr;
    u16 dport;
    unsigned short num;
    u32 saddr;
    struct socket *socket;
    void (*data_ready)();
};

Figure 14.2: An Internet protocol control block
sockaddr_in{
    short int sin_family; /* Address family, AF_INET */
    unsigned short int sin_port; /* Port number */
    struct in_addr sin_addr; /* Internet address */
};

Figure 14.3: Internet socket address structure
Figure 14.4: Connecting two sockets
| ethhdr | iphdr | udphdr | Data |

**Figure 14.5**: A physical frame
struct ethhdr{
    unsigned char h_dest[];
    unsigned char h_source[];
    unsigned short h_proto;
};

Figure 14.6: An ethernet header
Figure 14.7: Structure identifying a packet type

```
struct packet_type{
    unsigned short type;
    int (*func)();
    struct packet_type *next;
};
```
struct iphdr{
    u8    protocol;
    u16   check;
    u32   saddr;
    u32   daddr;
};

Figure 14.8: An IP header
Figure 14.9: A UDP header

```c
struct udphdr{
    u16 source;
    u16 dest;
    u16 check;
};
```
struct tcphdr{
    u16 source;
    u16 dest;
    u32 seq;
    u16 check;
};

**Figure 14.10**: A TCP header
Figure 14.11: Function call/return
Figure 14.12: Call/return with RPC
program-definition:
   "program" program-name "{" 
     version-list 
   "}" "=" value ";" 

version-list:
   version ";" 
   | version ";" version list 

version:
   "version" version-name "{" 
     procedure-list 
   "}" "=" value ";" 

procedure-list:
   procedure ";" 
   | procedure ";" procedure-list 

procedure:
   type-ident procedure-name "(" type-ident ")" "=" value ";" 

Figure 14.13: Format of RPC specification language
program MATH{
    version MATHVERSION{
        int DOUBLE (int) = 1;
        } = 1;
    } = 0x20000001;

**Figure 14.14**: Interface definition for MATH program
struct authunix_parms{
    u_long aup_time;    /* time credentials were created */
    char  *aup_machname; /* host name of client's machine */
    int   aup_uid;      /* client's Unix user id */
    int   aup_gid;      /* client's current group id */
    int   *aup_gids;    /* array of client's groups */
};

Figure 14.15: Unix style credentials
Figure 14.16: Overview of RPC system
struct rpc_msg{
    u_long rm_xid;
    enum msg_type rm_direction;
    union{
        struct call_body RM_cmb;
        struct reply_body RM_rmb;
    }ru;
};

Figure 14.17: Structure of an RPC message
struct call_body {
    u_long cb_rpcvers; /* RPC version number */
    u_long cb_prog;   /* program number */
    u_long cb_vers;   /* version number */
    u_long cb_proc;   /* procedure number */
    struct opaque_auth cb_cred; /* authentication */
};

**Figure 14.18**: Body of a call message
struct reply_body{
    enum reply_stat rp_stat;
    union{
        struct accepted_reply RP_ar;
        struct rejected_reply RP_dr;
    }ru;
};

Figure 14.19: Body of a reply message
Figure 15.1: Events in process X
Figure 15.2: Events in process Y
Figure 15.3: Events in process Z
Figure 15.4: Centralised control of mutual exclusion
Figure 15.5: Distributed control of mutual exclusion
Figure 15.6: Successful two phase commit
Figure 15.7: Aborted two phase commit
Transaction 1

Lock A
Lock B
Lock C
Write A
Unlock A
Unlock B
Abort

Transaction 2

Lock A
Read A
Read A again

Figure 15.8: A cascaded abort
**Lock Table**

<table>
<thead>
<tr>
<th>data_id</th>
<th>cond</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_id</td>
<td>cond</td>
</tr>
<tr>
<td>data_id</td>
<td>cond</td>
</tr>
<tr>
<td>data_id</td>
<td>cond</td>
</tr>
</tbody>
</table>

**Figure 15.9:** A lock table
Lock (data-id, trans-id, lock-type)

MUTEX_LOCK(table)
IF (data-id already in table) THEN
    WHILE conflicting
        CWAIT(condvar-data-id)
    ENDWHILE
    Add trans-id to existing entry
ENDIF
Add new entry to table
MUTEX_UNLOCK(table)
Unlock(trans-id)

MUTEX_LOCK(table)
FOR (each entry in table belonging to trans-id) DO
Remove the entry
IF (trans-id was only holder) THEN
    CSIGNAL (condvar-data-id)
ENDIF
ENDFOR
MUTEX_UNLOCK(table)
Figure 15.12: Local and global wait-for graphs
Figure 16.1: Situation before attaching remote files
Figure 16.2: Local file system after attaching remote files
Figure 16.3: A client and two servers
Figure 16.4: File system after mounting Dir4
Figure 16.5: File system after mounting Dir7
read_super()

Client

request for mountd portnumber

portnumber

pathname

Server

rpcbind

mountd

filehandle

Figure 16.6: Mounting a remote file system
struct nfs_sb_info{
    struct nfs_server s_server;
    struct nfs_fh s_root;
};

Figure 16.7: NFS specific super block information
Figure 16.8: Interaction between client and server
nfs_inode_info{
    struct nfs_fh fhandle;
    unsigned long read_cache_jiffies;
    unsigned long read_cache_mtime;
    unsigned long attrtimeo;
};

Figure 16.9: NFS specific inode information
Figure 16.10: Handling local and remote file systems
<table>
<thead>
<tr>
<th></th>
<th>File1</th>
<th>File2</th>
<th>File3</th>
<th>CDdrive</th>
<th>Printer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain1</td>
<td>Read</td>
<td></td>
<td>Read</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Domain2</td>
<td></td>
<td></td>
<td></td>
<td>Read</td>
<td>Print</td>
</tr>
<tr>
<td>Domain3</td>
<td></td>
<td>Read</td>
<td>Execute</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Domain4</td>
<td>Read/Write</td>
<td></td>
<td>Read/Write</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 17.1**: Example access matrix
Figure 17.2: Sequence of messages with secret keys
Figure 17.3: Digital signature with secret key
Figure 17.4: Encryption for both security and authentication
Figure 17.5: Three phases in the Kerberos system
Figure 17.6: Initial exchange of messages with authentication server
Figure 17.7: Authenticating a user to a file server
Figure 17.8: Authenticating a user to a file server

| New session key | Encrypted ticket for file server | Original session key |
Figure 17.9: Requesting service of a file server