INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE CENTRATOM TRIESTE

The United Nations University

SMR/774 - 28

THIRD COLLEGE ON MICROPROCESSOR-BASED REAL-TIME CONTROL - PRINCIPLES AND APPLICATIONS IN PHYSICS
26 September - 21 October 1994

INTRODUCTION TO REAL TIME OPERATING SYSTEMS

Catharinus VERKERK
C.E.R.N.-European Organization
for Nuclear Research
Computing and Networks Division
CH-1211 Geneva
SWITZERLAND

Introduction.

- These lectures give a basic introduction to real-time operating systems.
- We will concentrate on **principles** and show various solutions to some of the problems.
- For concrete examples, we will - nearly always - use Linux
- The lectures will largely follow the presentation given in the book:
  A. Tanenbaum: "Operating Systems, Design and Implementation",
  Prentice-Hall Int.,1987,
  ISBN 0-13-637331-3.

Introduction.

- What is an operating system? Why do we need it? What can we do with it?
- A computer is useful only when it executes a program. Certain parts of this program must handle directly pieces of hardware. This is not simple and very often beyond the average programmer or user. Just try to think of disk I/O!

Introduction.

- Thus one view of an operating system is that it shall provide a **reasonable interface** to the user. In other words, it should provide to the programmer a set of **easily understandable commands**, for instance for doing I/O. All idiosynchracies of the CPU architecture and of interface chips should be hidden.

- Another view of an operating system is that it should provide **resource management**, resolving possibly conflicting user requests.

Introduction.

- One can distinguish different types of **interactive** operating systems:
  - simple, single user (MSDOS, Flex)
  - multitasking (Macintosh, GEM, OS2)
  - multi-user, time sharing (UNIX, VM, VMS, OS-9, LynXOS)
  - real-time Kernels (iRMK, AMX, VRTX, etc.)

- As you will have to start soon working with Linux we will look first at its outside, before delving into the internal workings of operating systems.

Levels of Abstraction.

- One can view a computer at different levels of abstraction:
  - at the bottom is **hardware**.
  - a microprogram (or hardwired logic) acting on the hardware defines a **machine language**.
  - the **operating system** provides a more convenient **interface** fo the user. It defines a **virtual machine**.
  - on top of the operating system we have **system utilities** such as : a command interpreter, compilers, editors, etc.

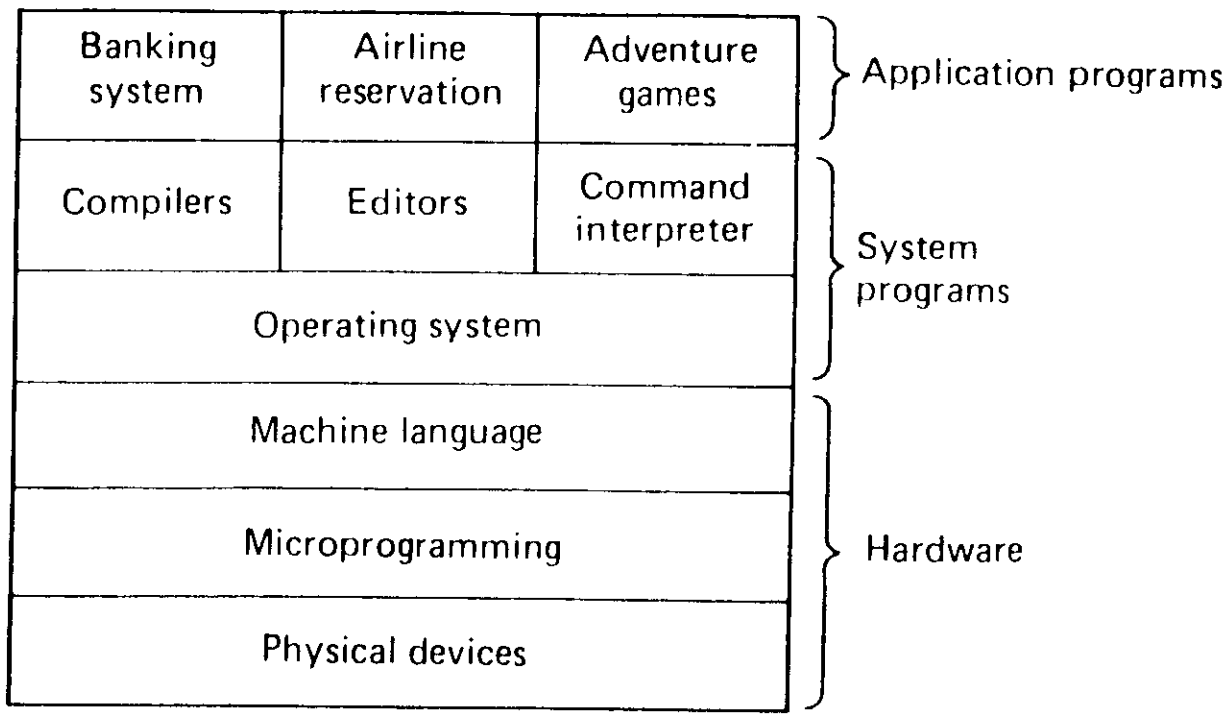


Fig. 1-1. A computer system consists of hardware, system programs and application programs.

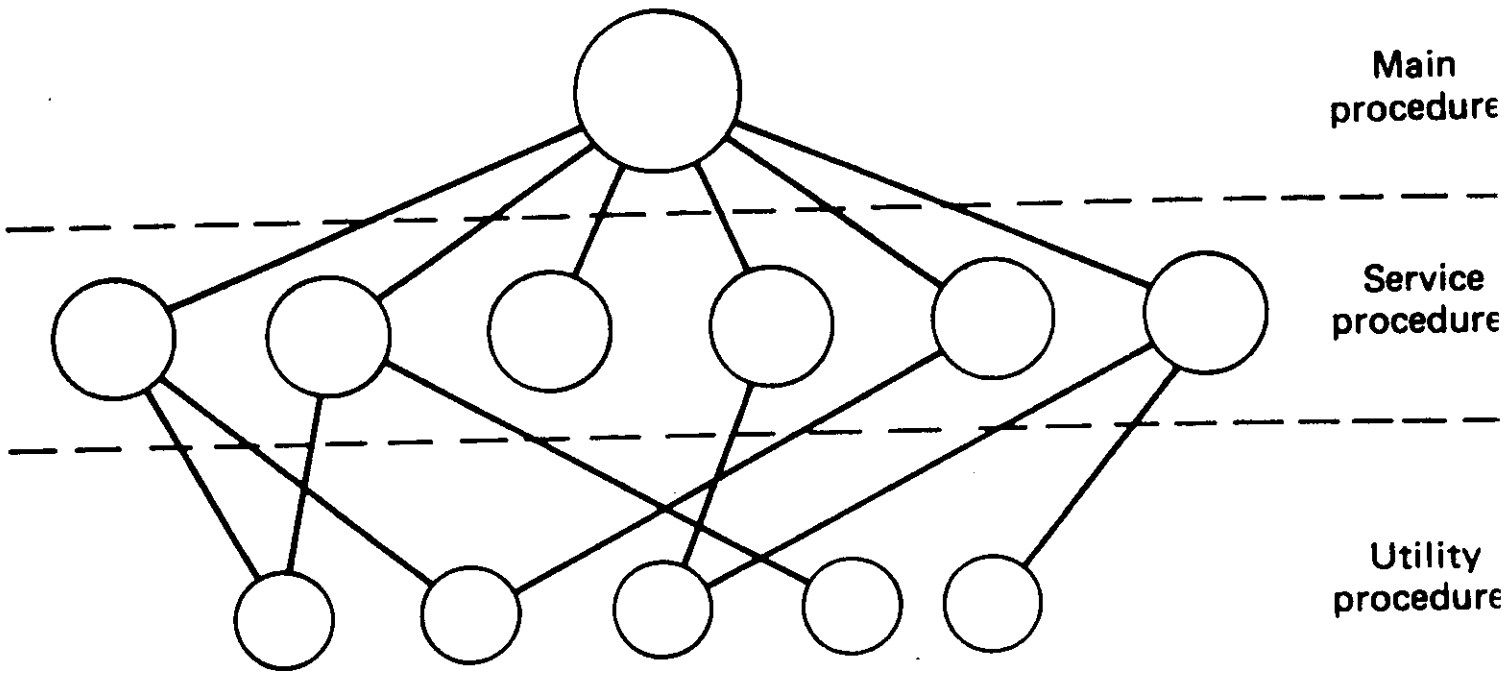


Fig. 1-19. A simple structuring model for a monolithic system.

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

Fig. 1-20. Structure of the THE operating system.

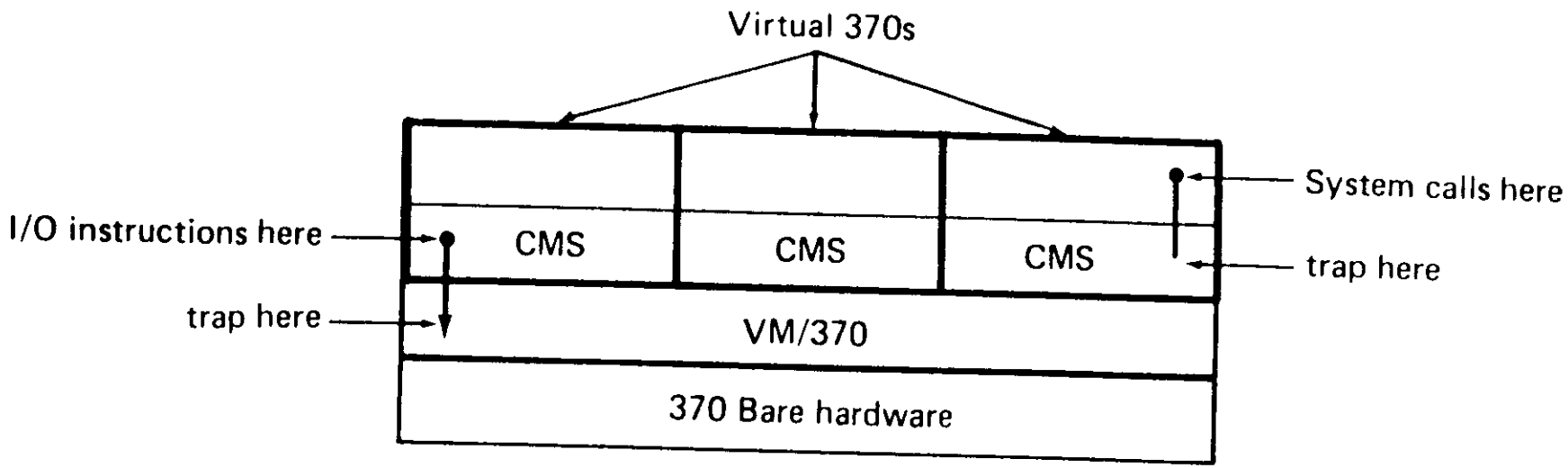


Fig. 1-21. The structure of VM/370 with CMS.

Levels of Abstraction.

- the top most layer is made up of the application programs.

- The operating system itself can (and usually is) also be **layered**.

- The operating system may in fact have one of four possible structures:
  - monolithic
  - layered
  - virtual machine
  - server-client.

- In the server-client concept some functions, traditionally part of the operating system, are pushed upward to higher layers (e.g. file-server).

Levels of Abstraction.

- The client-server model is well adapted to distributed systems.

- On most machines, the operating system runs in **kernel (or protected) mode**, whereas the layers above it run in **user mode**. Certain machine instructions cannot be executed in user mode.

- Evolution of operating systems: "plugboards", batch, multi-programming, time-sharing. With PCs and workstations evolution toward user-friendliness and network operating systems.

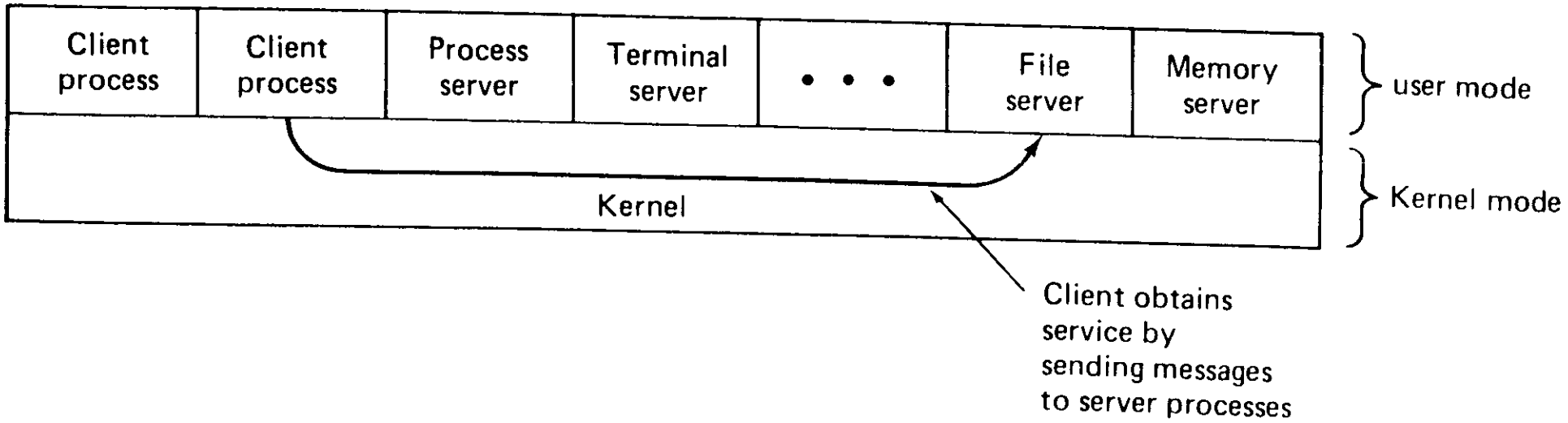
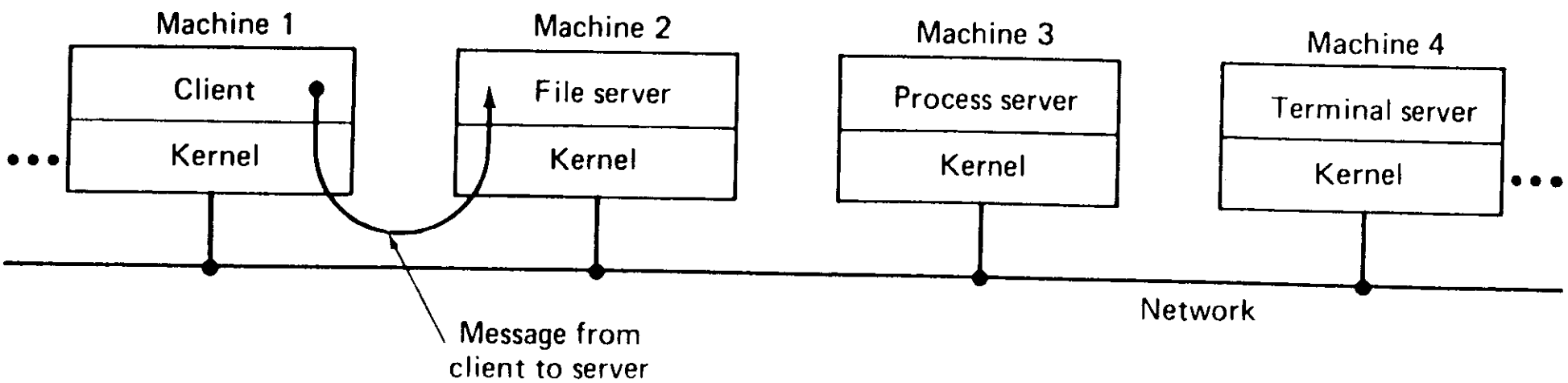


Fig. 1-22. The client-server model.



System Calls and Processes.

- Programs communicate with the operating system through **system calls** (or **service requests**).Implementation varies, but it usually works through a **software interrupt**.

- Key concept is the **process**. A process is a program in execution; it consists of the executable program, its data and stack, program counter, stack pointer and all other registers and other information needed to run the program.

System Calls and Processes.

- The existence of processes implies that there must exist system calls for **creating** and **terminating** processes. The **shell** or **command interpreter** will create a process when the user has typed in a line. For instance, it may create a process that will run the compiler. When that process has finished its job, it will execute a system call to terminate itself.

- The shell itself is also a process. In the example, the compiler is a **child** process of the shell. When it terminates it returns to the **parent**.

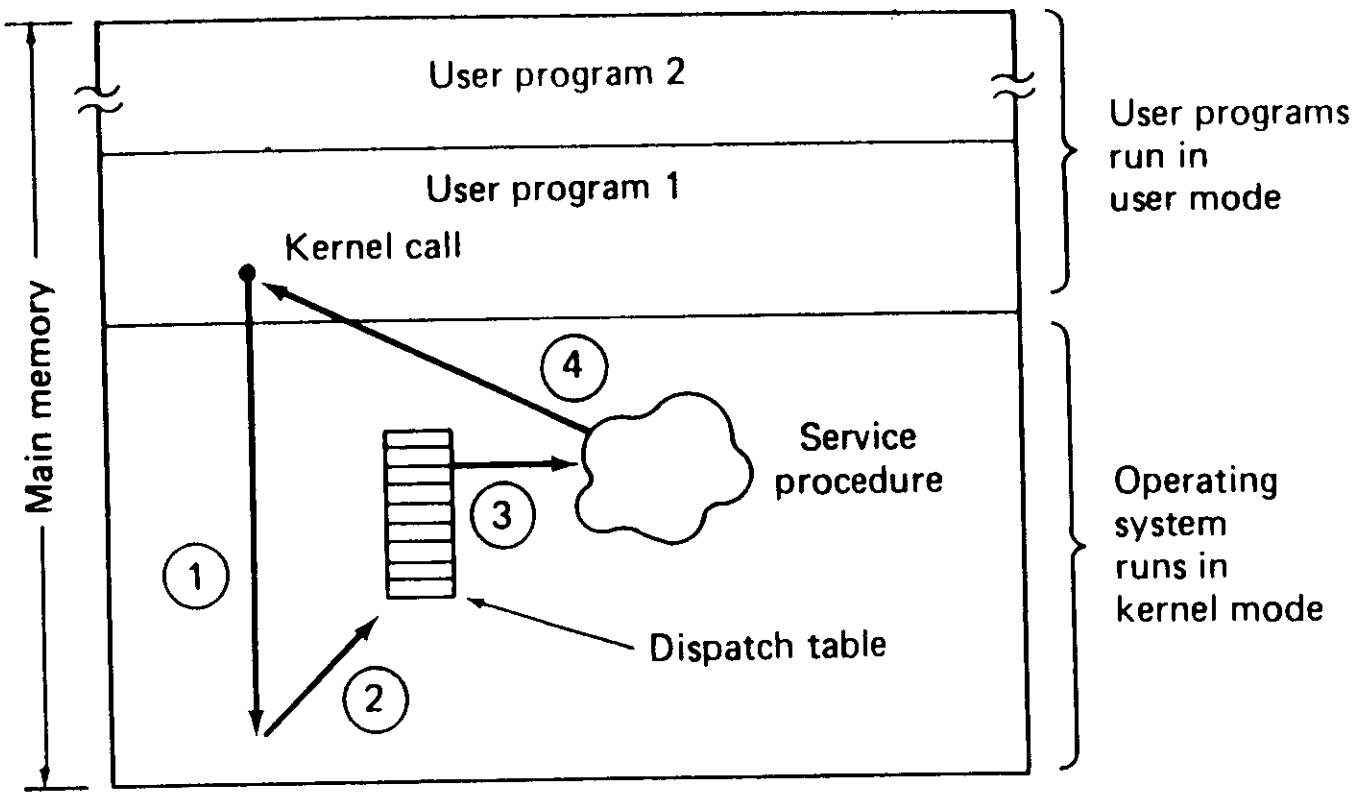
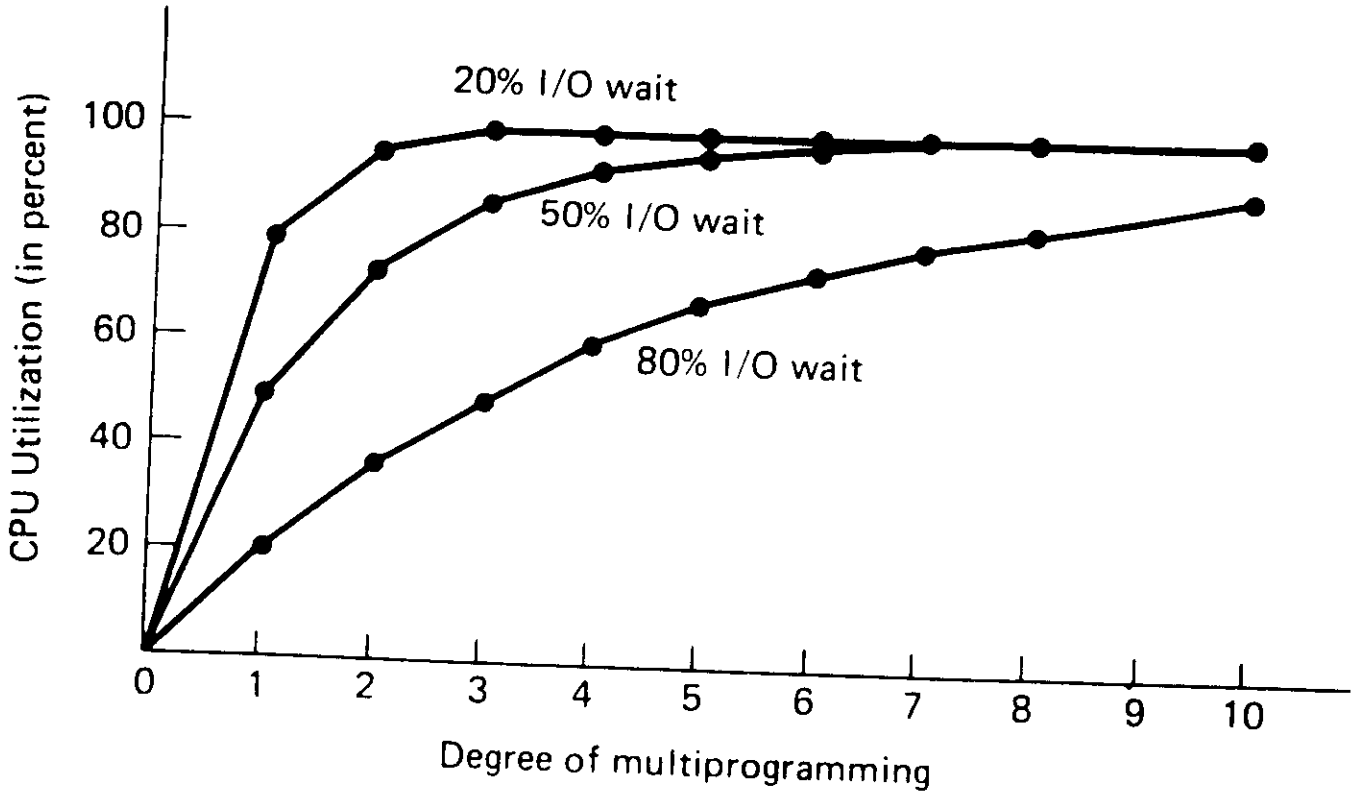
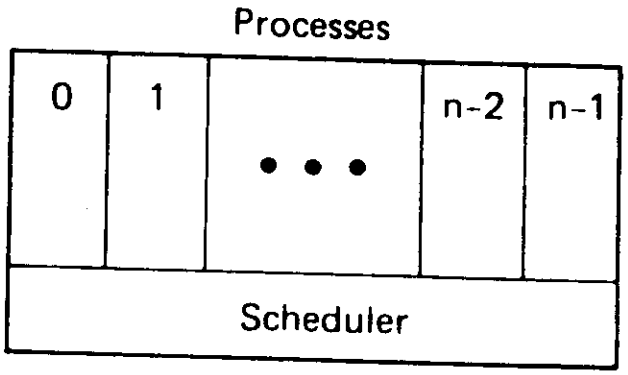


Fig. 1-18. How a system call can be made: (1) User program traps to the kernel. (2) Operating system determines service number required. (3) Operating system locates and calls service procedure. (4) Control is returned to user program.



. **4-2.** CPU utilization as a function of the number of processes in memory.



3. The lowest layer of a process-structured operating system handles in- ; and does scheduling. The rest of the system consists of sequential es.

System Calls and Processes.

- A process may be suspended and later resumed. All information must be saved before suspension!
- Processes will need memory to run in. When a process dies, the memory becomes free for another process to use.
- Processes will often create child processes to communicate with the outside world, e.g. perform input-output.

System Calls and Processes.

- It is now easy to see that an operating system has four main functions:
  - **process management**
  - **memory management**
  - **Input/Output management**
  - **Interrupt handling** and time management.
- All these functions have their specific system calls. They are implemented as a disjoint **set of programs**, which make use of **common utility routines** (for instance for adding or deleting items from a table).

Processes.

- A single CPU can only do one thing at a time. Rapid switching from one task to another creates the illusion that the CPU is doing many things in parallel.
- The **process model** helps to understand what happens and to keep track of things going on.
- All runnable software (including the OS) is organised in **sequential processes**.

Processes.

- A **process** is an *executing program,including its input and output and its state (e.g. the contents of machine registers and the values of the program's variables).*
- The operating system must be able to **create** a process when needed and to destroy it when finished. In UNIX,Minix andLinuxa process is created with the **FORK** system call.
- A process may issue one or more FORKs, and create **child process**(es). The child(s) may again create other childs, etc.

Processes.

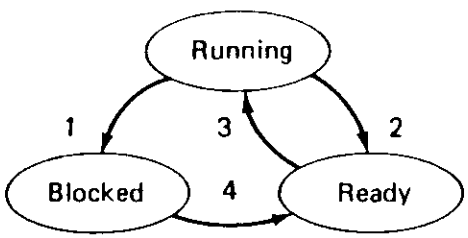
- When the system is booted, at some stage a process is forked, which will start things going. In a multiprogramming system, one process per terminal may be started; each will wait till someone logs in.
- In the model, processes are independent entities, but often they need to interact with each other (through a pipe or otherwise).

Processes.

- A common situation is that a process does not find input ready, when it needs it. It should then **block**, until input becomes available.
- IT must **not** do **busy-wait**! This occupies the CPU uselessly.
- Processes can be in one of **three states**:
  - **running** (e.g. using the CPU at this precise moment in time)
  - **blocked** (e.g. waiting for an external event or another process)
  - **ready** (e.g. ready to run, waiting to be scheduled for execution)

Processes.

- The diagram shows the possible transitions between the three states.



Processes.

- The process model allows us to think in terms of user processes, disk processes, terminal I/O processes, etc, without having to consider **interrupts** and interrupt handling. (Obviously running various **processes concurrently** needs **interrupts**, generated by **I/O devices** and by a **clock**).

- Inside the operating system **kernel** is the **scheduler**, which takes care of the **interrupt handling** and schedules processes for execution. The rest of the operating system can be nicely structured in processes.

Processes.

- The remaining part of the kernel only needs to contain initialization and utility routines used by other parts of the system (or the user).

- The scheduler uses a process table of some sort to manage the various processes. Every process which may run has an entry in this table. It contains **all the information** needed to restart the process exactly where it left off and with all states and conditions restored as they were at the moment the process was interrupted.

Processes.

- The **scheduling algorithm** tries to satisfy one or more of the following -contradictory- criteria:
  - **fairness**
  - **efficiency**
  - **response time**
  - **turn around**
  - **throughput**

Processes.

- For a real-time system controlling equipment, **response time** is generally the most important. If the system must respond within a given limit of time, we call it a **hard real-time** system.
- The scheduler cannot predict what a process is going to do, thus a **clock** must interrupt the system regularly, so that the scheduler may intervene.
- **Preemptive scheduling** allows temporary suspension of a running process, in contrast to **run to completion**.

Processes.

- Many scheduling algorithms exist:
  - round robin: all processes get a time slice in turn.
  - priority: each process has an initial priority assigned. At each clock the scheduler may decrement the priority of the running process.
  - multiple priority classes. Within a class priority scheduling is applied.
  - shortest job first: run times must be known in advance, or estimated from past behaviour.
  - policy driven: a goal is fixed and the scheduler tries to live up to it (fairness, response time).
  - Two-level: needed when part of the runnable processes are on disk.

Processes.

- Processes need often to communicate with each other. **Inter-process communication** is usually done with **messages**

- When a message is sent to a process, it will be delivered to it by the operating system. It is the responsability of the receiving process to interpret the message. Some messages are interpreted by the operating system itself (e.g. kill, wakeup).

Mutual Exclusion.

- In most **real-time systems various tasks run concurrently**, some of them may be **interrupt driven**.
- **Synchronisation** of the tasks is necessary.
- Problems arise if several tasks **access a variable** and can **modify its value**. (or claim a sharable resource).
- Example: real-time clock.
  - an interrupt driven routine maintains the time of the day in 3 bytes in memory: hh, mm, ss.

Mutual Exclusion.

- another, independent task may read these bytes and display them.
- Now suppose hh:mm:ss have the values 11:59:59 when the second task is reading the time.
- Suppose a clock interrupt occurs when the task has read hh, but before it was able to read mm.
- What will be displayed?

- The access to the shared variable(s) is a **critical section** of the program and the two processes must **mutually exclude** access to this critical section.

- Access to hardware control registers of devices is also critical.

Mutual Exclusion.

- Examples, where synchronisation and mutual exclusion are needed:
  - **car park** (of a supermarket) with several entrance gates and one or more exit gates, where barriers must be operated. Problems become serious when the car park is full.
  - **client-server** model (particularly relevant for real-time systems), where a server produces items and puts them into a buffer. The client takes items out of the buffer and consumes them. The critical sections are the updating of the buffer pointers. As long as the buffer is not full or empty no great harm is done. Serious problems when **buffer is full or empty**.

Mutual Exclusion.

- Easy solution which **always works**: *disable interrupts before* entering and *enable interrupts after* leaving the critical section.
- This is **inadmissible** in a general purpose multi-user system. It may be dangerous in a **hard real-time system**.
- A **flag**, which is set by one process and read by the other **does not work**. Why?

Mutual Exclusion.

CRITICAL REGION, USING NORMAL VARIABLE: FLAG

PROCESS 1	PROCESS 2
TEST1 *	TEST2 *
LDA FLAG	LDA FLAG
BEQ GOON1	BEQ GOON2
GO TO SLEEP	GO TO SLEEP
BRA TEST1	BRA TEST2
GOON1 INCA	GOON2 INCA
STA FLAG	STA FLAG
USE RESOURCE	USE RESOURCE
CLR FLAG	CLR FLAG

Mutual Exclusion.

- An **indivisible** (or **atomic**) **test-and-set** instruction is needed to guard the entrance to a critical section. TAS instruction will test a variable and if it is equal to zero, will set it to one. If it is already one it will leave it unchanged. The whole operation must be **uninterruptable**.

Mutual Exclusion.

CRITICAL REGION, USING LSR.

```
          PROCESS 1              PROCESS 2
TEST1     *              TEST2   *
          LSR FLAG               LSR FLAG
          BCS GOON1              BCS GOON2
          GO TO SLEEP            GO TO SLEEP
          BRA TEST1              BRA TEST2
GOON1     USE RESOURCE   GOON2   USE RESOURCE
          LDA #1                 LDA #1
          STA FLAG               STA FLAG
```

Mutual Exclusion.

- What does a process do when access to a critical section is denied to it? (Or when a server finds the buffer full?)
- Easiest and very **inefficient solution**: **busy-wait**.
- **Better solution:** The process which cannot get access should **go to sleep**, and be **woken-up** later, when access becomes free.

Mutual Exclusion.

- It is important to realize that the TAS (or equivalent) instruction is in itself **not enough** to handle all possible situations. It is useful help given by the hardware.
- Several mechanisms have been invented:
  - Dekkers' algorithm (very complicated)
  - Dijkstra's **semaphore**
  - **Event counters**
  - **Monitors** (a **language construct**)
  - **Message passing**
  - Ada rendez-vous (another language construct).

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

producer()
{
  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? */
  }
}


consumer()
{
  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 */
  }
}
```

```
#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 */

producer()
{
  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 */
  }
}


consumer()
{
  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 */
  }
}
```

Fig. 2-11. The producer-consumer problem using semaphores.

Mutual Exclusion.

- Note that these methods will require in most cases considerable effort by the programmer.
- Also note that **none of these methods** will automatically **prevent deadlock** or **starvation**.
- Simple example of deadlock:
  - process A holds resource X and needs Y
  - process B holds resource Y and needs X.

This is Info file ipc.info, produced by Makeinfo-1.47 from the input file ipc.texi.

This file documents the System V style inter process communication primitives available under linux.

File: ipc.info, Node: top, Next: Overview, Prev: Notes, Up: (dir)

System V IPC.
****

These facilities are provided to maintain compatibility with programs developed on system V unix systems and others that rely on these system V mechanisms to accomplish inter process communication (IPC).

The specifics described here are applicable to the Linux implementation. Other implementations may do things slightly differently.

* Menu:

File: ipc.info, Node: Overview, Next: example, Prev: top, Up: top

Overview
========

System V IPC consists of three mechanisms:

* Messages : exchange messages with any process or server.

* Semaphores : allow unrelated processes to synchronize execution.

* Shared memory : allow unrelated processes to share memory.

* Menu:

Access to all resources is permitted on the basis of permissions set up when the resource was created.

A resource here consists of message queue, a semaphore set (array) or a shared memory segment.

A resource must first be allocated by a creator before it is used. The creator can assign a different owner. After use the resource must be explicitly destroyed by the creator or owner.

A resource is identified by a numeric ID. Typically a creator

defines a KEY that may be used to access the resource. The user process may then use this KEY in the "get" system call to obtain the ID for the corresponding resource. This ID is then used for all further access. A library call "ftok" is provided to translate pathnames or strings to numeric keys.

There are system and implementation defined limits on the number and sizes of resources of any given type. Some of these are imposed by the implementation and others by the system administrator when configuring the kernel (*Note msglimits::, *Note semlimits::, *Note shmlimits::).

There is an `msqid_ds', `semid_ds' or `shmid_ds' struct associated with each message queue, semaphore array or shared segment. Each ipc resource has an associated `ipc_perm' struct which defines the creator, owner, access perms ..etc.., for the resource. These structures are detailed in the following sections.

example

Here is a code fragment with pointers on how to use shared memory. The same methods are applicable to other resources.

In a typical access sequence the creator allocates a new instance of the resource with the `get' system call using the IPC_CREAT flag.

creator process:

```
     #include <sys/shm.h>
     int id;
     key_t key;
     char proc_id = 'C';
     int size = 0x5000; /* 20 K */
     int flags = 0664 | IPC_CREAT;                    /* read-only for others */

     key = ftok ("creator/ipckey", proc_id);
     id = shmget (key, size, flags);
     exit (0);   /* quit leaving resource allocated */
```

Users then gain access to the resource using the same key.
Client process:

```
     #include <sys/shm.h>
     char *shmaddr;
     int id;
     key_t key;
     char proc_id = 'C';

     key = ftok ("creator/ipckey", proc_id);

     id = shmget (key, 0, 004);          /* default size   */
     if (id == -1)
           perror ("shmget ...");

     shmaddr = shmat (id, 0, SHM_RDONLY); /* attach segment for reading */
     if (shmaddr == (char *) -1)
           perror ("shmat ...");

     local_var = *(shmaddr + 3);         /* read segment etc. */

     shmdt (shmaddr);                 /* detach segment */
```

When the resource is no longer needed the creator should remove it.
Creator/owner process 2:

```
key = ftok ("~creator/ipckey", proc_id)
id = shmget (key, 0, 0);
shmctl (id, IPC_RMID, NULL);
```

File: ipc.info, Node: perms, Next: syscalls, Prev: example, Up: Overview

Permissions

Each resource has an associated 'ipc_perm' struct which defines the creator, owner and access perms for the resource.

```
struct ipc_perm
        key_t key;      /* set by creator */
        ushort uid;     /* owner euid and egid */
        ushort gid;
        ushort cuid;    /* creator euid and egid */
        ushort cgid;
        ushort mode;    /* access modes in lower 9 bits */
        ushort seq;     /* sequence number */
```

The creating process is the default owner. The owner can be reassigned by the creator and has creator perms. Only the owner, creator or super-user can delete the resource.

The lowest nine bits of the flags parameter supplied by the user to the system call are compared with the values stored in 'ipc_perms.mode' to determine if the requested access is allowed. In the case that the system call creates the resource, these bits are initialized from the user supplied value.

As for files, access permissions are specified as read, write and exec for user, group or other (though the exec perms are unused). For example 0624 grants read-write to owner, write-only to group and read-only access to others.

For shared memory, note that read-write access for segments is determined by a separate flag which is not stored in the 'mode' field. Shared memory segments attached with write access can be read.

The 'cuid', 'cgid', 'key' and 'seq' fields cannot be changed by the user.

File: ipc.info, Node: syscalls, Next: Messages, Prev: perms, Up: Overview

IPC system calls

This section provides an overview of the IPC system calls. See the specific sections on each type of resource for details.

Each type of mechanism provides a "get", "ctl" and one or more "op" system calls that allow the user to create or procure the resource (get), define its behaviour or destroy it (ctl) and manipulate the resources (op).

The "get" system calls

The 'get' call typically takes a KEY and returns a numeric ID that is used for further access. The ID is an index into the resource table. A sequence number is maintained and incremented when a resource is destroyed so that acceses using an obselete ID is likely to fail.

The user also specifies the permissions and other behaviour charecteristics for the current access. The flags are or-ed with the permissions when invoking system calls as in:

```
msgflg = IPC_CREAT | IPC_EXCL | 0666;
id = msgget (key, msgflg);
```

* 'key' : IPC_PRIVATE => new instance of resource is initialized.

* 'flags' :
  IPC_CREAT : resource created for KEY if it does not exist.

  IPC_CREAT | IPC_EXCL : fail if resource exists for KEY.

* returns : an identifier used for all further access to the resource.

Note that IPC_PRIVATE is not a flag but a special 'key' that ensures (when the call is successful) that a new resource is created.

Use of IPC_PRIVATE does not make the resource inaccessible to other users. For this you must set the access permissions appropriately.

There is currently no way for a process to ensure exclusive access to a resource. IPC_CREAT | IPC_EXCL only ensures (on success) that a new resource was initialized. It does not imply exclusive access.

See Also : *Note msgget::, *Note semget::, *Note shmget::.

The "ctl" system calls

Provides or alters the information stored in the structure that describes the resource indexed by ID.

```
#include <sys/msg.h>
struct msqid_ds buf;
err = msgctl (id, IPC_STAT, &buf);
if (err)
        !$#%*
else
        printf ("creator uid = %d\n", buf.msg_perm.cuid);
        ....
```

Commands supported by all 'ctl' calls:

* IPC_STAT : read info on resource specified by id into user allocated buffer. The user must have read access to the resource.

* IPC_SET : write info from buffer into resource data structure. The user must be owner creator or super-user.

* IPC_RMID : remove resource. The user must be the owner, creator or super-user.

The IPC_RMID command results in immediate removal of a message queue or semaphore array. Shared memory segments however, are only destroyed upon the last detach after IPC_RMID is executed.

The 'semctl' call provides a number of command options that allow the user to determine or set the values of the semaphores in an array.

See Also: *Note msgctl::, *Note semctl::, *Note shmctl::.

The "op" system calls

Used to send or receive messages, read or alter semaphore values, attach or detach shared memory segments. The IPC_NOWAIT flag will cause the operation to fail with error EAGAIN if the process has to wait on the call.

'flags' : IPC_NOWAIT => return with error if a wait is required.

See Also: *Note msgsnd::,*Note msgrcv::,*Note semop::,*Note shmat::, *Note shmdt::.

File: ipc.info, Node: Messages, Next: msgget, Prev: syscalls, Up: top

Messages
========

A message resource is described by a struct 'msqid_ds' which is allocated and initialized when the resource is created. Some fields in 'msqid_ds' can then be altered (if desired) by invoking 'msgctl'. The memory used by the resource is released when it is destroyed by a 'msgctl' call.

```
struct msqid_ds
    struct ipc_perm msg_perm;
    struct msg *msg_first;  /* first message on queue (internal) */
    struct msg *msg_last;   /* last message in queue (internal) */
    time_t msg_stime;       /* last msgsnd time */
    time_t msg_rtime;       /* last msgrcv time */
    time_t msg_ctime;       /* last change time */
    struct wait_queue *wwait; /* writers waiting (internal) */
    struct wait_queue *rwait; /* readers waiting (internal) */
    ushort msg_cbytes;      /* number of bytes used on queue */
    ushort msg_qnum;        /* number of messages in queue */
    ushort msg_qbytes;      /* max number of bytes on queue */
    ushort msg_lspid;       /* pid of last msgsnd */
    ushort msg_lrpid;       /* pid of last msgrcv */
```

To send or receive a message the user allocates a structure that looks like a 'msgbuf' but with an array 'mtext' of the required size. Messages have a type (positive integer) associated with them so that (for example) a listener can choose to receive only messages of a given type.

```
struct msgbuf
    long mtype;       type of message (*Note msgrcv::).
    char mtext[1];    message text .. why is this not a ptr?
```

The user must have write permissions to send and read permissions to receive messages on a queue.

When 'msgsnd' is invoked, the user's message is copied into an internal struct 'msg' and added to the queue. A 'msgrcv' will then read this message and free the associated struct 'msg'.

* Menu:

* msgget::
* msgsnd::
* msgrcv::
* msgctl::
* msglimits:: Implementation defined limits.

File: ipc.info, Node: msgget, Next: msgsnd, Prev: Messages, Up: Messages

msgget

A message queue is allocated by a msgget system call :

```
msqid = msgget (key_t key, int msgflg);
```

* 'key' : an integer usually got from 'ftok( )' or IPC_PRIVATE.

* 'msgflg' :
  IPC_CREAT : used to create a new resource if it does not already exist.

  IPC_EXCL | IPC_CREAT : used to ensure failure of the call if the resource already exists.

  rwxrwxrwx : access permissions.

* returns: msqid (an integer used for all further access) on success. -1 on failure.

A message queue is allocated if there is no resource corresponding to the given key. The access permissions specified are then copied into the 'msg_perm' struct and the fields in 'msqid_ds' initialized. The user must use the IPC_CREAT flag or key = IPC_PRIVATE, if a new instance is to be allocated. If a resource corresponding to KEY already exists, the access permissions are verified.

Errors:
EACCES : (procure) Do not have permission for requested access.
EEXIST : (allocate) IPC_CREAT | IPC_EXCL specified and resource exists.
EIDRM : (procure) The resource was removed.
ENOSPC : All id's are taken (max of MSGMNI id's system-wide).
ENOENT : Resource does not exist and IPC_CREAT not specified.
ENOMEM : A new 'msqid_ds' was to be created but ... nomem.

File: ipc.info, Node: msgsnd, Next: msgrcv, Prev: msgget, Up: Messages

msgsnd

```
int msgsnd (int msqid, struct msgbuf *msgp, int msgsz, int msgflg);
```

* 'msqid' : id obtained by a call to msgget.

* 'msgsz' : size of msg text ('mtext') in bytes.

* 'msgp' : message to be sent. (msgp->mtype must be positive).

* 'msgflg' : IPC_NOWAIT.

* returns : msgsz on success. -1 on error.

The message text and type are stored in the internal 'msg' structure. 'msg_cbytes', 'msg_qnum', 'msg_lspid', and 'msg_stime' fields are updated. Readers waiting on the queue are awakened.

Errors:
EACCES : Do not have write permission on queue.
EAGAIN : IPC_NOWAIT specified and queue is full.
EFAULT : msgp not accessible.
EIDRM : The message queue was removed.
EINTR : Full queue ... would have slept but ... was interrupted.

EINVAL : mtype < 1, msgsz > MSGMAX, msgsz < 0, msqid < 0 or unused.
ENOMEM : Could not allocate space for header and text.

File: ipc.info, Node: msgrcv, Next: msgctl, Prev: msgsnd, Up: Messages

msgrcv

```
int msgrcv (int msqid, struct msgbuf *msgp, int msgsz, long msgtyp,
                    int msgflg);
```

* msqid : id obtained by a call to msgget.

* msgsz : Maximum size of message to receive.

* msgp : Allocated by user to store the message in.

* msgtyp :
  0 => get first message on queue.

  > 0 => get first message of matching type.

  < 0 => get message with least type which is <= abs(msgtyp).

* msgflg :
  IPC_NOWAIT : Return immediately if message not found.

  MSG_NOERROR : The message is truncated if it is larger than msgsz.

  MSG_EXCEPT : Used with msgtyp > 0 to receive any msg except of specified type.

* returns : size of message if found. -1 on error.

The first message that meets the 'msgtyp' specification is identified. For msgtyp < 0, the entire queue is searched for the message with the smallest type.

If its length is smaller than msgsz or if the user specified the MSG_NOERROR flag, its text and type are copied to msgp->mtext and msgp->mtype, and it is taken off the queue.

The 'msg_cbytes', 'msg_qnum', 'msg_lrpid', and 'msg_rtime' fields are updated. Writers waiting on the queue are awakened.

Errors:
E2BIG : msg bigger than msgsz and MSG_NOERROR not specified.
EACCES : Do not have permission for reading the queue.
EFAULT : msgp not accessible.
EIDRM : msg queue was removed.
EINTR : msg not found ... would have slept but ... was interrupted.
EINVAL : msgsz > msgmax or msgsz < 0, msqid < 0 or unused.
ENOMSG : msg of requested type not found and IPC_NOWAIT specified.

File: ipc.info, Node: msgctl, Next: msglimits, Prev: msgrcv, Up: Messages

msgctl

```
int msgctl (int msqid, int cmd, struct msqid_ds *buf);
```

* msqid : id obtained by a call to msgget.

* buf : allocated by user for reading/writing info.

* cmd : IPC_STAT, IPC_SET, IPC_RMID (*Note syscalls::).

IPC_STAT results in the copy of the queue data structure into the user supplied buffer.

In the case of IPC_SET, the queue size ('msg_qbytes') and the 'uid', 'gid', 'mode' (low 9 bits) fields of the 'msg_perm' struct are set from the user supplied values. 'msg_ctime' is updated.

Note that only the super user may increase the limit on the size of a message queue beyond MSGMNB.

When the queue is destroyed (IPC_RMID), the sequence number is incremented and all waiting readers and writers are awakened. These processes will then return with 'errno' set to EIDRM.

Errors:

EPERM : Insufficient privilege to increase the size of the queue (IPC_SET) or remove it (IPC_RMID).
EACCES : Do not have permission for reading the queue (IPC_STAT).
EFAULT : buf not accessible (IPC_STAT, IPC_SET).
EIDRM : msg queue was removed.
EINVAL : invalid cmd, msqid < 0 or unused.

File: ipc.info, Node: msglimits, Next: Semaphores, Prev: msgctl, Up: Messages

Limis on Message Resources

Sizeof various structures:

```
     msqid_ds          52   /* 1 per message  queue .. dynamic */

     msg               16   /* 1 for each message in system .. dynamic */

     msgbuf             8   /* allocated by user */
```

Limits

* MSGMNI : number of message queue identifiers ... policy.

* MSGMAX : max size of message. Header and message space allocated on one page. MSGMAX = (PAGE_SIZE - sizeof(struct msg)). Implementation maximum MSGMAX = 4080.

* MSGMNB : default max size of a message queue ... policy. The super-user can increase the size of a queue beyond MSGMNB by a 'msgctl' call.

Unused or unimplemented:
MSGTQL max number of message headers system-wide.
MSGPOOL total size in bytes of msg pool.

File: ipc.info, Node: Semaphores, Next: semget, Prev: msglimits, Up: top

Semaphores
==========

Each semaphore has a value >= 0. An id provides access to an array of 'nsems' semaphores. Operations such as read, increment or decrement semaphores in a set are performed by the 'semop' call which processes

'nsops' operations at a time. Each operation is specified in a struct 'sembuf' described below. The operations are applied only if all of them succeed.

If you do not have a need for such arrays, you are probably better off using the 'test_bit', 'set_bit' and 'clear_bit' bit-operations defined in <asm/bitops.h>.

Semaphore operations may also be qualified by a SEM_UNDO flag which results in the operation being undone when the process exits.

If a decrement cannot go through, a process will be put to sleep on a queue waiting for the 'semval' to increase unless it specifies IPC_NOWAIT. A read operation can similarly result in a sleep on a queue waiting for 'semval' to become 0. (Actually there are two queues per semaphore array).

A semaphore array is described by:

```
struct semid_ds
  struct ipc_perm sem_perm;
  time_t          sem_otime;      /* last semop time */
  time_t          sem_ctime;      /* last change time */
  struct wait_queue *eventn;      /* wait for a semval to increase */
  struct wait_queue *eventz;      /* wait for a semval to become 0 */
  struct sem_undo  *undo;         /* undo entries */
  ushort          sem_nsems;      /* no. of semaphores in array */
```

Each semaphore is described internally by :

```
struct sem
  short   sempid;          /* pid of last semop() */
  ushort  semval;          /* current value */
  ushort  semncnt;         /* num procs awaiting increase in semval */
  ushort  semzcnt;         /* num procs awaiting semval = 0 */
```

* Menu:

* semget::
* semop::
* semctl::
* semlimits:: Limits imposed by this implementation.

File: ipc.info, Node: semget, Next: semop, Prev: Semaphores, Up: Semaphores

semget

A semaphore array is allocated by a semget system call:

```
semid = semget (key_t key, int nsems, int semflg);
```

* 'key' : an integer usually got from 'ftok' or IPC_PRIVATE

* 'nsems' :
  # of semaphores in array (0 <= nsems <= SEMMSL <= SEMMNS)

  0 => dont care can be used when not creating the resource. If successful you always get access to the entire array anyway.

* semflg :
  IPC_CREAT used to create a new resource

  IPC_EXCL used with IPC_CREAT to ensure failure if the resource exists.

rwxrwxrwx access permissions.

* returns : semid on success. -1 on failure.

An array of nsems semaphores is allocated if there is no resource corresponding to the given key. The access permissions specified are then copied into the 'sem_perm' struct for the array along with the user-id etc. The user must use the IPC_CREAT flag or key = IPC_PRIVATE if a new resource is to be created.

Errors:
EINVAL : nsems not in above range (allocate).
nsems greater than number in array (procure).
EEXIST : (allocate) IPC_CREAT : IPC_EXCL specified and resource exists.
EIDRM : (procure) the resource was removed.
ENOMEM : could not allocate space for semaphore array.
ENOSPC : No arrays available (SEMMNI), too few semaphores available (SEMMNS).
ENOENT : Resource does not exist and IPC_CREAT not specified.
EACCES : (procure) do not have permission for specified access.

File: ipc.info, Node: semop, Next: semctl, Prev: semget, Up: Semaphores

semop

Operations on semaphore arrays are performed by calling semop :

```
int semop (int semid, struct sembuf *sops, unsigned nsops);
```

* semid : id obtained by a call to semget.

* sops : array of semaphore operations.

* nsops : number of operations in array (0 < nsops < SEMOPM).

* returns : semval for last operation. -1 on failure.

Operations are described by a structure sembuf:

```
struct sembuf
    ushort  sem_num;        /* semaphore index in array */
    short   sem_op;         /* semaphore operation */
    short   sem_flg;        /* operation flags */
```

The value 'sem_op' is to be added (signed) to the current value semval of the semaphore with index sem_num (0 .. nsems -1) in the set. Flags recognized in sem_flg are IPC_NOWAIT and SEM_UNDO.

Two kinds of operations can result in wait:

1. If sem_op is 0 (read operation) and semval is non-zero, the process sleeps on a queue waiting for semval to become zero or returns with error EAGAIN if (IPC_NOWAIT | sem_flg) is true.

2. If (sem_op < 0) and (semval + sem_op < 0), the process either sleeps on a queue waiting for semval to increase or returns with error EAGAIN if (sem_flg & IPC_NOWAIT) is true.

The array sops is first read in and preliminary checks performed on the arguments. The operations are parsed to determine if any of them needs write permissions or requests an undo operation.

The operations are then tried and the process sleeps if any operation that does not specify IPC_NOWAIT cannot go through. If a process sleeps it repeats these checks on waking up. If any operation that requests

IPC_NOWAIT, cannot go through at any stage, the call returns with errno set to EAGAIN.

Finally, operations are committed when all go through without an intervening sleep. Processes waiting on the zero_queue or increment_queue are awakened if any of the semval's becomes zero or is incremented respectively.

Errors:
E2BIG : nsops > SEMOPM.
EACCES : Do not have permission for requested (read/alter) access.
EAGAIN : An operation with IPC_NOWAIT specified could not go through.
EFAULT : The array sops is not accessible.
EFBIG : An operation had semnum >= nsems.
EIDRM : The resource was removed.
EINTR : The process was interrupted on its way to a wait queue.
EINVAL : nsops is 0, semid < 0 or unused.
ENOMEM : SEM_UNDO requested. Could not allocate space for undo structure.
ERANGE : sem_op + semval > SEMVMX for some operation.

File: ipc.info, Node: semctl, Next: semlimits, Prev: semop, Up: Semaphores

semctl

```
int semctl (int semid, int semnum, int cmd, union semun arg);
```

* semid : id obtained by a call to semget.

* cmd :
  GETPID return pid for the process that executed the last semop.

  GETVAL return semval of semaphore with index semnum.

  GETNCNT return number of processes waiting for semval to increase.

  GETZCNT return number of processes waiting for semval to become 0

  SETVAL set semval = arg.val.

  GETALL read all semval's into arg.array.

  SETALL set all semval's with values given in arg.array.

* returns : 0 on success or as given above. -1 on failure.

The first 4 operate on the semaphore with index semnum in the set. The last two operate on all semaphores in the set.

'arg' is a union :

```
union semun
    int val;                value for SETVAL.
    struct semid_ds *buf;   buffer for IPC_STAT and IPC_SET.
    ushort *array;          array for GETALL and SETALL
```

* IPC_SET, SETVAL, SETALL : sem_ctime is updated.

* SETVAL, SETALL : Undo entries are cleared for altered semaphores in all processes. Processes sleeping on the wait queues are awakened if a semval becomes 0 or increases.

* IPC_SET : sem_perm.uid, sem_perm.gid, sem_perm.mode are updated from user supplied values.

Errors:

EACCES : do not have permission for specified access.
EFAULT : arg is not accessible.
EIDRM : The resource was removed.
EINVAL : semid < 0 or semnum < 0 or semnum >= nsems.
EPERM : IPC_RMID, IPC_SET ... not creator, owner or super-user.
ERANGE : arg.array[i].semval > SEMVMX or < 0 for some i.

File: ipc.info, Node: semlimits, Next: Shared Memory, Prev: semctl, Up: Semaphores

Limits on Semaphore Resources

Sizeof various structures:

```
     semid_ds     44   /* 1 per semaphore array .. dynamic */
     sem           8   /* 1 for each semaphore in system .. dynamic */
     sembuf        6   /* allocated by user */
     sem_undo     20   /* 1 for each undo request .. dynamic */
```

Limits :

* SEMVMX 32767 semaphore maximum value (short).

* SEMMNI number of semaphore identifiers (or arrays) system wide...policy.

* SEMMSL maximum number of semaphores per id. 1 semid_ds per array, 1 struct sem per semaphore => SEMMSL = (PAGE_SIZE - sizeof(semid_ds)) / sizeof(sem). Implementation maximum SEMMSL = 500.

* SEMMNS maximum number of semaphores system wide ... policy. Setting SEMMNS >= SEMMSL*SEMMNI makes it irrelevent.

* SEMOPM Maximum number of operations in one semop call...policy.

Unused or unimplemented:
SEMAEM adjust on exit max value.
SEMMNU number of undo structures system-wide.
SEMUME maximum number of undo entries per process.

File: ipc.info, Node: Shared Memory, Next: shmget, Prev: semlimits, Up: top

Shared Memory

Shared memory is distinct from the sharing of read-only code pages or the sharing of unaltered data pages that is available due to the copy-on-write mechanism. The essential difference is that the shared pages are dirty (in the case of Shared memory) and can be made to appear at a convenient location in the process' address space.

A shared segment is described by :

```
     struct shmid_ds
         struct  ipc_perm shm_perm;
         int     shm_segsz;              /* size of segment (bytes) */
         time_t  shm_atime;              /* last attach time */
```

```
        time_t  shm_dtime;              /* last detach time */
        time_t  shm_ctime;              /* last change time */
        ulong   *shm_pages;             /* internal page table */
        ushort  shm_cpid;               /* pid, creator */
        ushort  shm_lpid;               /* pid, last operation */
        short   shm_nattch;             /* no. of current attaches */
```

A shmget allocates a shmid_ds and an internal page table. A shmat maps the segment into the process' address space with pointers into the internal page table and the actual pages are faulted in as needed. The memory associated with the segment must be explicitly destroyed by calling shmctl with IPC_RMID.

* Menu:

* shmget::
* shmat::
* shmdt::
* shmctl::
* shmlimits:: Limits imposed by this implementation.

File: ipc.info, Node: shmget, Next: shmat, Prev: Shared Memory, Up: Shared Memory

shmget

A shared memory segment is allocated by a shmget system call:

```
  int shmget(key_t key, int size, int shmflg);
```

* key : an integer usually got from 'ftok' or IPC_PRIVATE

* size : size of the segment in bytes (SHMMIN <= size <= SHMMAX).

* shmflg :
  IPC_CREAT used to create a new resource

  IPC_EXCL used with IPC_CREAT to ensure failure if the resource exists.

  rwxrwxrwx access permissions.

* returns : shmid on success. -1 on failure.

A descriptor for a shared memory segment is allocated if there isn't one corresponding to the given key. The access permissions specified are then copied into the 'shm_perm' struct for the segment along with the user-id etc. The user must use the IPC_CREAT flag or key = IPC_PRIVATE to allocate a new segment.

If the segment already exists, the access permissions are verified, and a check is made to see that it is not marked for destruction.

'size' is effectively rounded up to a multiple of PAGE_SIZE as shared memory is allocated in pages.

Errors:
EINVAL : (allocate) Size not in range specified above.
(procure) Size greater than size of segment.
EEXIST : (allocate) IPC_CREAT | IPC_EXCL specified and resource exists.
EIDRM : (procure) The resource is marked destroyed or was removed.
ENOSPC : (allocate) All id's are taken (max of SHMMNI id's system-wide).
Allocating a segment of the requested size would exceed the system wide

limit on total shared memory (SHMALL).
ENOENT : (procure) Resource does not exist and IPC_CREAT not specified.
EACCES : (procure) Do not have permission for specified access.
ENOMEM : (allocate) Could not allocate memory for shmid_ds or pg_table.

File: ipc.info, Node: shmat, Next: snmat, Prev: shmget, Up: Shared Memory

shmat

Maps a shared segment into the process' address space.

```
char *virt_addr;
virt_addr = shmat (int shmid, char *shmaddr, int shmflg);
```

* shmid : id got from call to shmget.

* shmaddr : requested attach address.
  If shmaddr is 0 the system finds an unmapped region.
  If a non-zero value is indicated the value must be page
  aligned or the user must specify the SHM_RND flag.

* shmflg :
  SHM_RDONLY : request read-only attach.
  SHM_RND : attach address is rounded DOWN to a multiple of SHMLBA.

* returns: virtual address of attached segment. -1 on failure.

When shmaddr is 0, the attach address is determined by finding an unmapped region in the address range 1G to 1.5G, starting at 1.5G and coming down from there. The algorithm is very simple so you are encouraged to avoid non-specific attaches.

Algorithm:
Determine attach address as described above.
Check region (shmaddr, shmaddr + size) is not mapped and allocate page tables (undocumented SHM_REMAP flag!).
Map the region by setting up pointers into the internal page table.
Add a descriptor for the attach to the task struct for the process.
'shm_nattch', 'shm_lpid', 'shm_atime' are updated.

Notes:
The 'brk' value is not altered. The segment is automatically detached when the process exits. The same segment may be attached as read-only or read-write and more than once in the process' address space. A shmat can succeed on a segment marked for destruction. The request for a particular type of attach is made using the SHM_RDONLY flag. There is no notion of a write-only attach. The requested attach permissions must fall within those allowed by 'shm_perm.mode'.

Errors:
EACCES : Do not have permission for requested access.
EINVAL : shmid < 0 or unused, shmaddr not aligned, attach at brk failed.
EIDRM : resource was removed.
ENOMEM : Could not allocate memory for descriptor or page tables.

File: ipc.info, Node: shmdt, Next: shmctl, Prev: shmat, Up: Shared Memory

shmdt

```
int shmdt (char *shmaddr);
```

* shmaddr : attach address of segment (returned by shmat).

* returns : 0 on success. -1 on failure.

An attached segment is detached and 'shm_nattch' decremented. The occupied region in user space is unmapped. The segment is destroyed if it is marked for destruction and 'shm_nattch' is 0. 'shm_lpid' and 'shm_dtime' are updated.

Errors:
EINVAL : No valid memory segment attached at shmaddr.

shmctl

Destroys allocated segments. Reads/Writes the control structures.

```
int shmctl (int shmid, int cmd, struct shmid_ds *buf);
```

* shmid : id got from call to shmget.

* cmd : IPC_STAT, IPC_SET, IPC_RMID (*Note syscalls::).
  IPC_SET : Used to set the owner uid, gid, and shm_perms.mode field.

  IPC_RMID : The segment is marked destroyed. It is only destroyed on the last detach.

  IPC_STAT : The shmid_ds structure is copied into the user allocated buffer.

* buf : used to read (IPC_STAT) or write (IPC_SET) information.

* returns : 0 on success, -1 on failure.

The user must execute an IPC_RMID shmctl call to free the memory allocated by the shared segment. Otherwise all the pages faulted in will continue to live in memory or swap.

Errors:
EACCES : Do not have permission for requested access.
EFAULT : buf is not accessible.
EINVAL : shmid < 0 or unused.
EIDRM : identifier destroyed.
EPERM : not creator, owner or super-user (IPC_SET, IPC_RMID).

Limits on Shared Memory Resources

Limits:
* SHMMNI max num of shared segments system wide ... 4096.

* SHMMAX max shared memory segment size (bytes) ... 4M

* SHMMIN min shared memory segment size (bytes). 1 byte (though PAGE_SIZE is the effective minimum size).

* SHMALL max shared mem system wide (in pages) ... policy.

* SHMLBA segment low boundary address multiple. Must be page aligned. SHMLBA = PAGE_SIZE.

Unused or unimplemented:
SHMSEG : maximum number of shared segments per process.

File: ipc.info, Node: Notes, Next: top, Prev: Limits, Up: top

Miscellaneous Notes

The system calls are mapped into one ... 'sys_ipc' ... this should be transparent to the user.

Semaphore 'undo' requests

There is one sem_undo structure associated with a process for each semaphore which was altered (with an undo request) by the process. 'sem_undo' structures are freed only when the process exits.

One major cause for unhappiness with the undo mechanism is that it does not fit in with the notion of having an atomic set of operations on an array. The undo requests for an array and each semaphore therein may have been accumulated over many 'semop' calls. Thus use the undo mechanism with private semaphores only.

Should the process sleep in 'exit' or should all undo operations be applied with the IPC_NOWAIT flag in effect? Currently those undo operations which go through immediately are applied and those that require a wait are ignored silently.

Shared memory, 'malloc' and the 'brk'.

Note that ... this ... was written the implementation was changed so that non-specific attaches are done in the region 1G - 1.5G. However much of the following is still worth thinking about so I left it in.

On many systems, the shared memory is allocated in a special region of the address space ... way up somewhere. As mentioned earlier, this implementation attaches shared segments at the lowest possible address. Thus if you plan to use 'malloc', it is wise to malloc a large space and then proceed to attach the shared segments. This way malloc sets the brk sufficiently above the region it will use.

Alternatively you can use 'sbrk' to adjust the 'brk' value as you make shared memory attaches. The implementation is not very smart about selecting attach addresses. Using the system default addresses will result in fragmentation if detaches do not occur in the reverse sequence as attaches.

Taking control of the matter is probably best. The rule applied is that attaches are allowed in unmapped regions other than in the text space (see <a.out.h>). Also remember that attach addresses and segment sizes are multiples of PAGE_SIZE.

One more trap (I quote Bruno on this). If you use malloc() to get space for your shared memory (ie. to fix the 'brk'), you must ensure you get an unmapped address range. This means you must mallocate more memory than you had ever allocated before. Memory returned by malloc(), used,

then freed by free() and then again returned by malloc is no good. Neither is calloced memory.

Note that a shared memory region remains a shared memory region until you unmap it. Attaching a segment at the 'brk' and calling malloc after that will result in an overlap of what malloc thinks is its space with what is really a shared memory region. For example in the case of a read-only attach you will not be able to write to the overlapped portion.

Fork, exec and exit

On a fork, the child inherits attached shared memory segments but not the semaphore undo information.

In the case of an exec, the attached shared segments are detached. The sem undo information however remains intact.

Upon exit, all attached shared memory segments are detached. The adjust values in the undo structures are added to the relevant semvals if the operations are permitted. Disallowed operations are ignored.

Other Features

These features of the current implementation are likely to be modified in the future.

The SHM_LOCK and SHM_UNLOCK flag are available (super-user) for use with the 'shmctl' call to prevent swapping of a shared segment. The user must fault in any pages that are required to be present after locking is enabled.

The IPC_INFO, MSG_STAT, MSG_INFO, SHM_STAT, SHM_INFO, SEM_STAT, SEMINFO 'ctl' calls are used by the 'ipcs' program to provide information on allocated resources. These can be modified as needed or moved to a proc file system interface.

Thanks to Ove Ewerlid, Bruno Haible, Ulrich Pegelow and Linus Torvalds for ideas, tutorials, bug reports and fixes, and merriment. And more thanks to Bruno.

Input-Output.

- We usually distinguish between **block devices** and **character devices**.
- The hardware of the device is interfaced to the computer via a **device controller** or **adapter**. In most cases a specialized chip: disk controller, ACIA, PIA, etc.
- The device is controlled by writing **commands** into registers of the controller, by reading back **status** information. Data is transfered by reading/writing from/to the controller's **data register(s)**.

Input-Output.

- Reading and writing the registers of a controller is done with special I/O instructions (Intel), or using **memory-mapping**. In the latter case, normal load and store instructions are used (DEC, Motorola).
- **Direct Memory Access** is done entirely by the controller. The transfer is set up by software. Data do not transit through the CPU. The completion of a **block transfer** usually generates an interrupt.

I/O Software.

- I/O software should fulfill two goals:
  - **hide** the peculiarities of the **hardware**
  - present a nice, clean and **regular interface** to the user.
- This leads naturally to a layered structure:
  - **interrupt handlers**
  - **device drivers**
  - **device-independent operating system software**
  - **user-level software**.

I/O Software.

- Another responsibility of I/O software is to handle errors.
- I/O software should also take account of the type of device: **sharable** (disks) or **dedicated** (printers, terminals).
- **device independence** means that the user should see no difference between writing to a file on a hard disk or writing on a printer.
- The **device driver's task** is to **receive abstract orders** from the software above and then to see to it that the job gets done.

Uniform interfacing for the 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

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

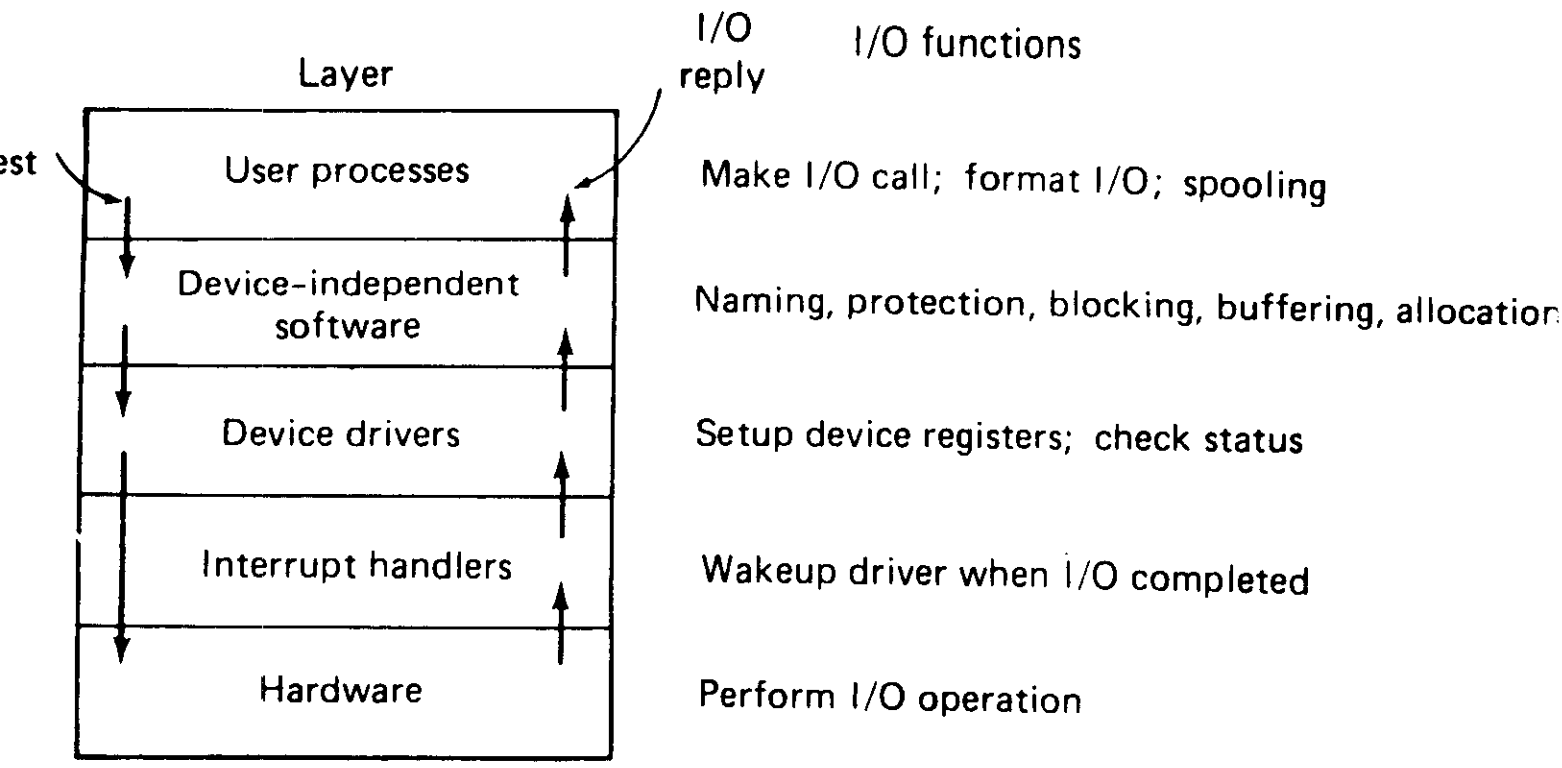


Fig. 3-6. Layers of the I/O system and the main functions of each layer.

I/O software.

- This means: translating the request into commands to the controller (start motor, move arm, set up DMA, etc., etc.) and issue them. If the execution of the command takes time, the driver must **block** (go to **sleep**), until an interrupt will cause it to be woken up. Finally errors are checked and status information passed back to the caller.

I/O software.

- The **device-independent I/O software** is responsible for:
  - uniform interfacing to the drivers
  - translating device names into selection of the appropriate driver
  - protection
  - buffering
  - storage allocation on disks
  - allocation and release of dedicated devices
  - error reporting.

I/O software.

- Library functions, which are linked into user-programs, do the remaining part of the input-output. Some library routines simply pass parameters on to a system call (e.g. *read, write*), others do more work (e.g. *printf, scanf*).
- A final part of I/O software is a **spooling system**. Files to be printed are put in the spooling directory. A printer **daemon** is the only process allowed to access the printer.

Deadlocks.

- In a multi-programming system, where non-sharable resources are allocated to processes, **deadlock** situations may occur.
- Deadlocks have been extensively studied, but the subject is not very important for real-time control or embedded systems, where dynamic allocation of non-sharable resources is rare.

Deadlocks.

- Different aspects of the problem are:
  - detection and recovery
  - prevention (by imposing rules on the processes)
  - avoidance (using an algorithm to make the right choice when resources must be allocated (the Banker's algorithm).

Memory management.

- The **aim of memory management** is to make **best use of available memory** and **to keep the CPU busy**.
- Two classes:
  - **without swapping** or **paging**: a process stays in memory until finished.
  - **with swapping or paging**: processes are moved between memory and disk, during "execution" of the process.
- The simple mono-programming case is of no interest.

Memory management.

- Multiprogramming is more complicated:
  - p = probability of process being idle (waiting for I/O). With n processes in memory p^n is probability that CPU is idle. For p=0.8 (not unusual at all!) , n must be 10 for idle time to be less than 10%.
- A multiprogramming system without swapping will need a large memory, which can be divided into **fixed size partitions** (not necessarily all of the same size) or **variable size partitions**.

Memory management.

- In all cases programs must be **relocated** as the memory address where it will run is not known at compile-time.
- Also, the partitions should be **protected**, to avoid that a bug in program A destroys program B in memory. Protection needs special hardware (**base** and **limit** registers for instance).

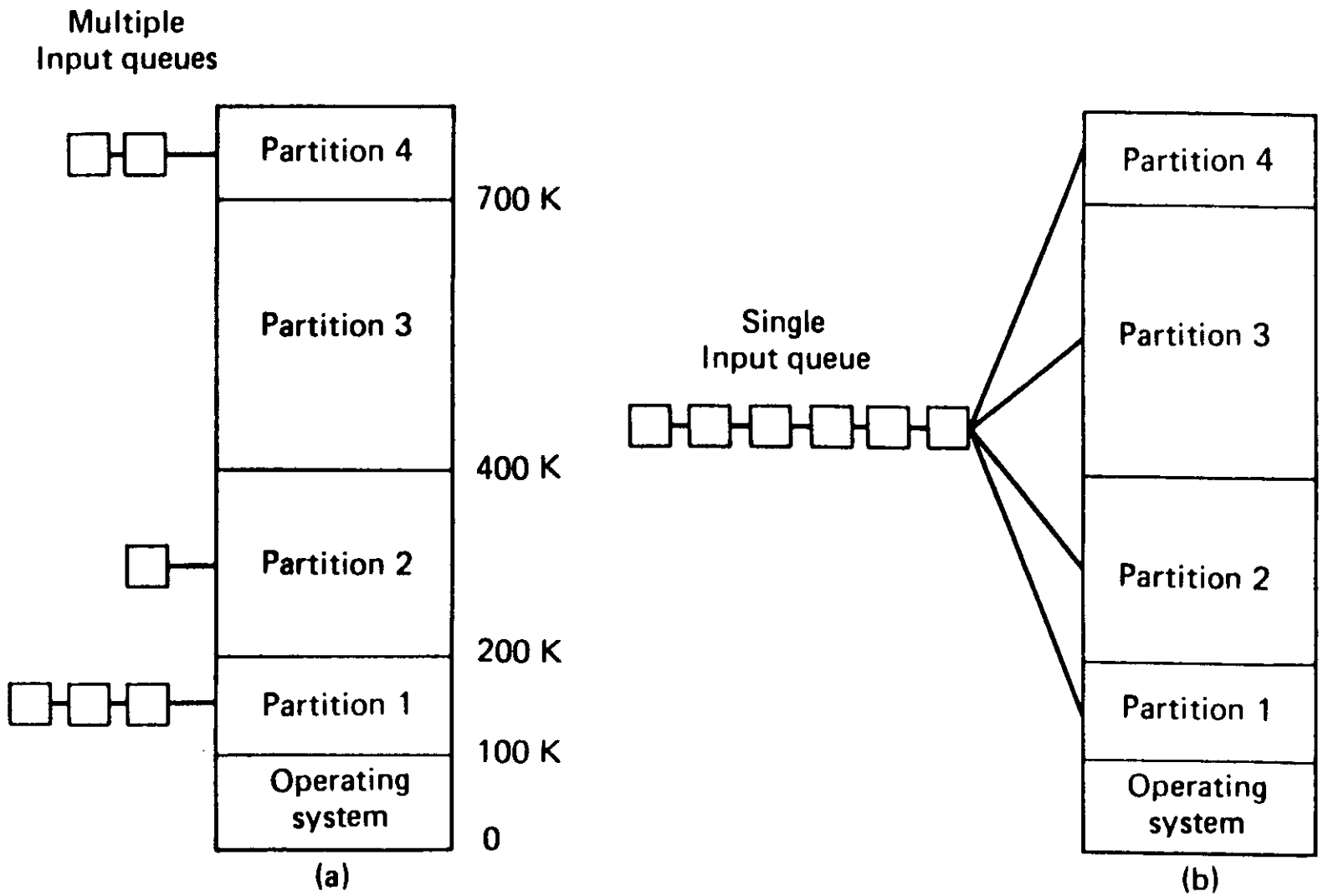


Fig. 4-4. (a) Fixed memory partitions with separate input queues for each partition. (b) Fixed memory partitions with a single input queue.

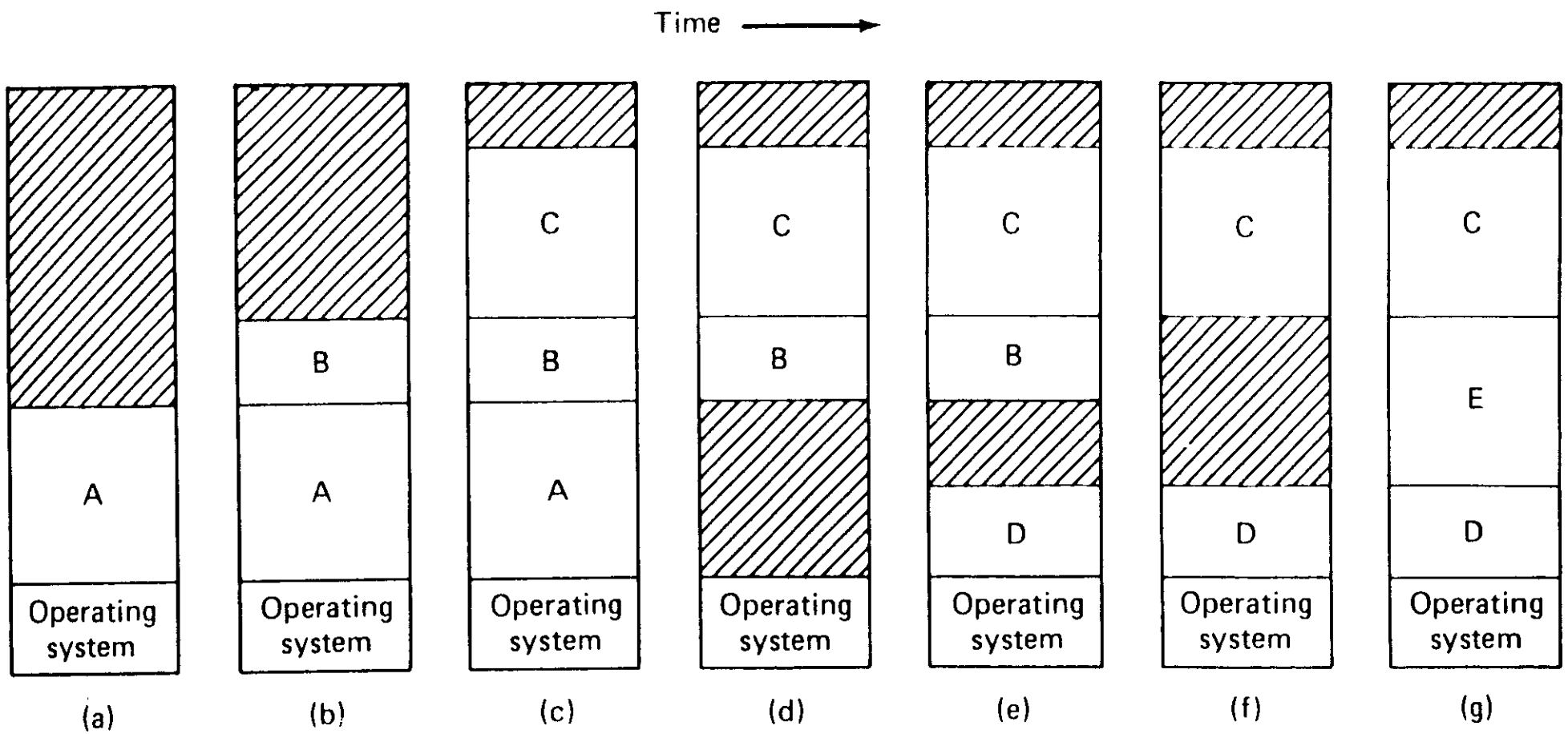


Fig. 4-5. Memory allocation changes as processes come into memory and leave it. The gray regions are unused memory.

Memory management.

- Fixed partition schemes may under-use memory, variable partition schemes will leave "holes" in memory when processes finish.
- The holes will be filled only partially by new processes. Memory **fragmentation** may occur, where all holes are too small to receive a reasonable program. Memory **compaction** combines all holes into one large hole.
- An extra complication is that data and stack areas may grow during execution (think of malloc()). So a process may grow out of its seams.

Swapping.

- Some of these problems may be aleviated if processes may be swapped from memory to disk (when they have to wait for I/O for instance) and brought back into memory later.
- Variable partitions may again be used. To keep track of where things are and of free memory space, different techniques are used:
  - bit maps. Each bit in the bit map represents a fixed size of memory.
  - linked lists. The list is sorted by address and links processes and holes.
  - buddy system.

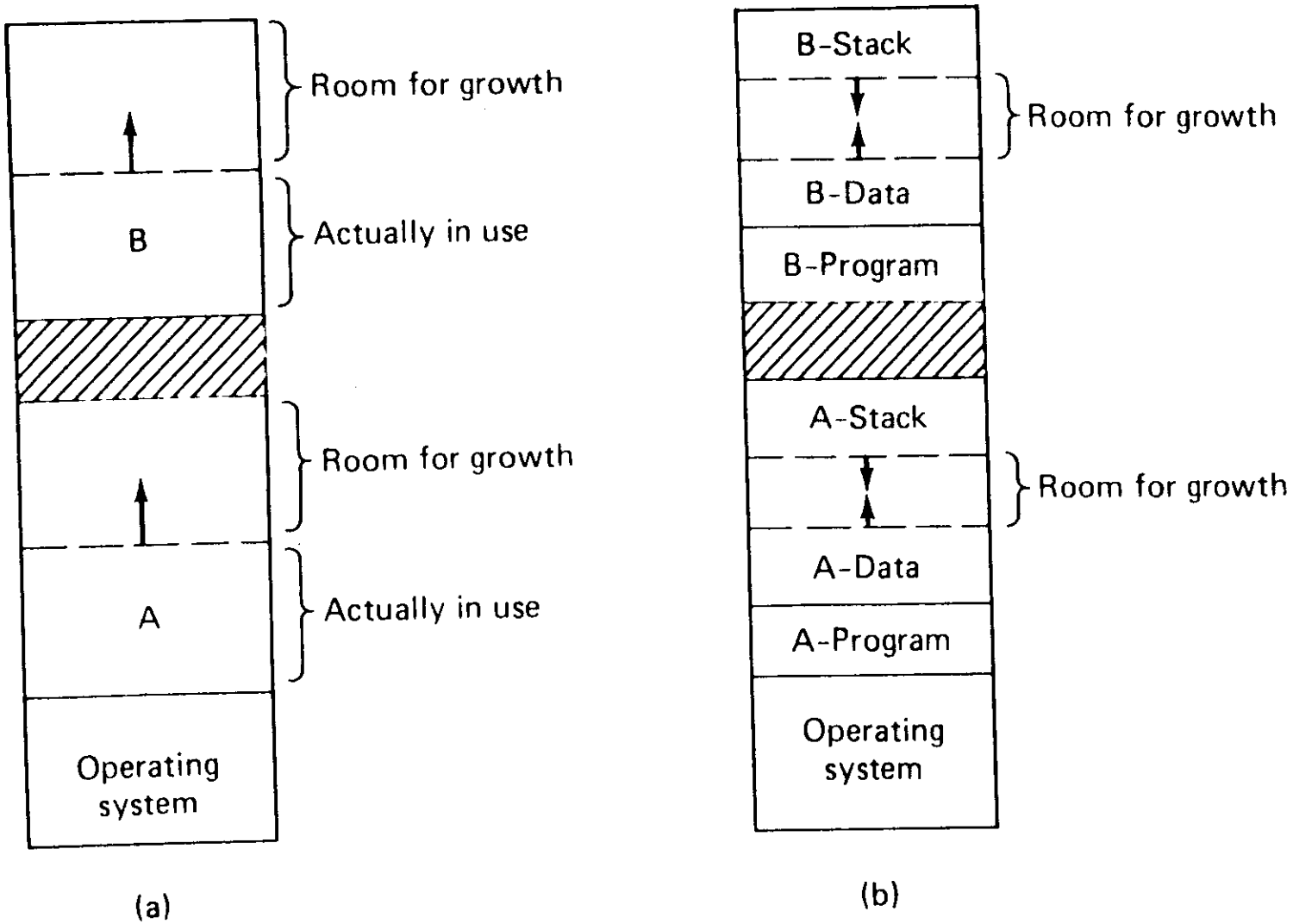


Fig. 4-6. (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

Swapping.

- When a process must be brought in'o memory, the **memory manager** must find a hole where to put it. Four algorithms:
  - first fit
  - next fit
  - best fit
  - worst fit.

Virtual Memory.

- Total size of program, data and stack may exceed size of memory. Keep those parts needed now in memory and the rest on disk. When a piece now on disk is needed, bring it into memory, throwing out (maybe) a piece no longer needed.

- When these things happen without the user being aware of it, we have a **virtual memory** system.

- **Virtual memory** and multiprogramming go very well together: when process A is swapped out, because it is waiting for I/O, another process may run.

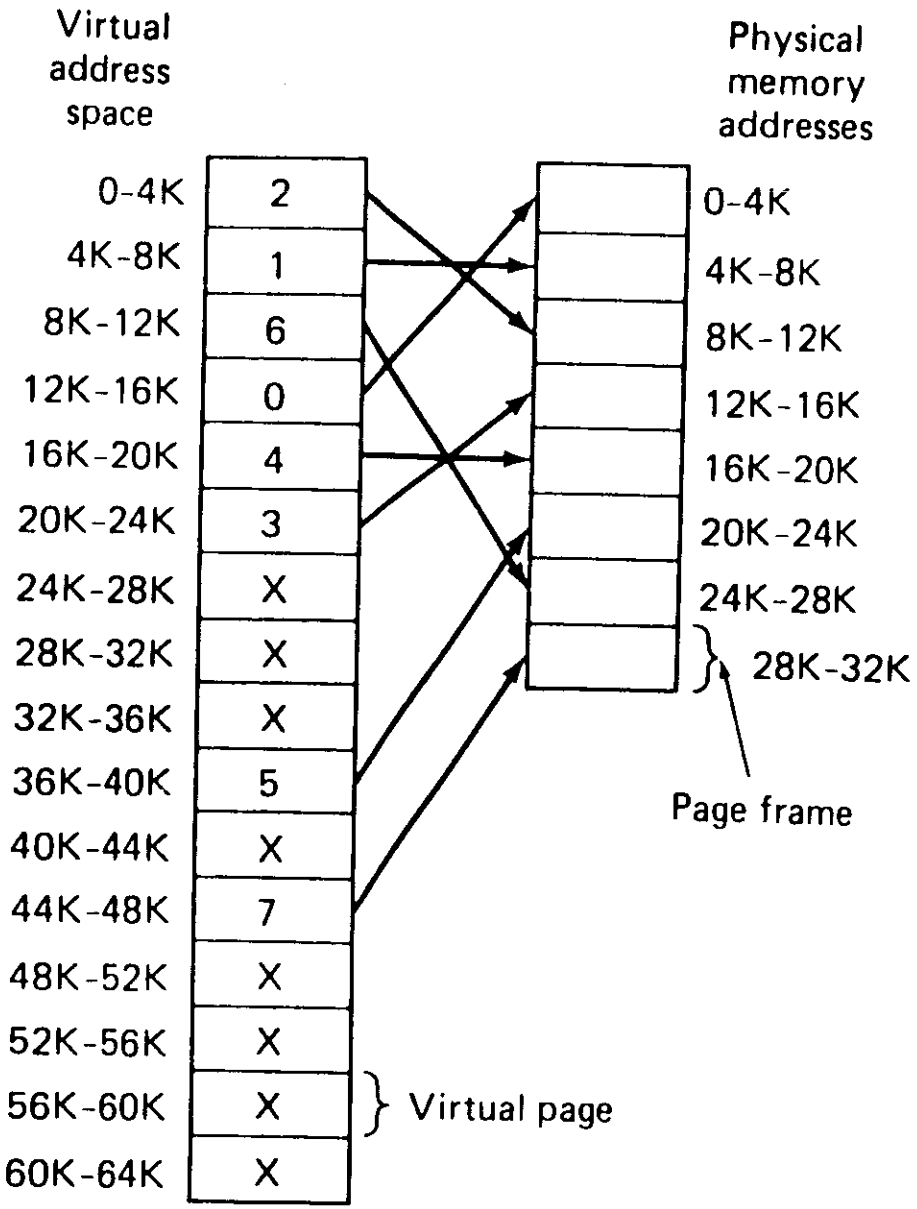


Fig. 4-11. The relation between virtual addresses and physical memory addresses is given by the page table.

Virtual Memory.

- Most **virtual memory** systems use **paging Virtual addresses** (the addresses the program uses) are translated into **physical addresses** by a **Memory Management Unit**.

- A **page fault** occurs when the program issues a virtual address in the range of an **unmapped** page (e.g. for which no physical address exists). This page is now brought into memory. If it is necessary to make room, another rarely used page is written to disk.

Virtual Memory.

- Several **page replacement algorithms** exist to choose the page to be thrown out.

- The ideal, but unrealisable, algorithm would throw out the page that will not be used before long in the future.

- Realisable algorithms are:
  - not-recently-used page replacement (NRU)
  - first-in first-out replacement (FIFO)
  - least recently used page replacement (LRU).