in the entry structure. The kernel increments the descriptor number so that the next instance of assigning the entry will return a different descriptor. Hence, system calls will fail if a process attempts to access an entry by an old descriptor. as explained earlier.

11.2.1 Messages

There are four system calls for messages: *msgget* returns (and possibly creates) a message descriptor that designates a message queue for use in other system calls, *msgctl* has options to set and return parameters associated with a message descriptor and an option to remove descriptors, *msgsnd* sends a message, and *msgrcv* receives a message.

The syntax of the *msgget* system call is

```
msgqid = msgget(key, flag);
```

where *msgqid* is the descriptor returned by the call, and *key* and *flag* have the semantics described above for the general "get" calls. The kernel stores messages on a linked list (queue) per descriptor, and it uses *msgqid* as an index into an array of message queue headers. In addition to the general IPC permissions field mentioned above, the queue structure contains the following fields:

- Pointers to the first and last messages on a linked list;
- The number of messages and the total number of data bytes on the linked list;
- The maximum number of bytes of data that can be on the linked list;
- The process IDs of the last processes to send and receive messages;
- Time stamps of the last *msgsnd*, *msgrcv*, and *msgctl* operations.

When a user calls *msgget* to create a new descriptor, the kernel searches the array of message queues to see if one exists with the given key. If there is no entry for the specified key, the kernel allocates a new queue structure, initializes it, and returns an identifier to the user. Otherwise, it checks permissions and returns.

A process uses the *msgsnd* system call to send a message:

```
msgsnd(msgqid, msg, count, flag);
```

where *msgqid* is the descriptor of a message queue typically returned by a *msgget* call, *msg* is a pointer to a structure consisting of a user-chosen integer type and a character array, *count* gives the size of the data array, and *flag* specifies the action the kernel should take if it runs out of internal buffer space.

The kernel checks (Figure 11.4) that the sending process has write permission for the message descriptor, that the message length does not exceed the system limit, that the message queue does not contain too many bytes, and that the message type is a positive integer. If all tests succeed, the kernel allocates space for the message from a message *map* (recall Section 9.1) and copies the data from user space. The kernel allocates a message header and puts it on the end of the linked list of message headers for the message queue. It records the message type and

```
algorithm msgsnd                /* send a message */
input:  (1) message queue descriptor
        (2) address of message structure
        (3) size of message
        (4) flags
output: number of bytes sent
{
      check legality of descriptor, permissions;
      while (not enough space to store message)
      {
            if (flags specify not to wait)
                  return;
            sleep(until event enough space is available);
      }
      get message header;
      read message text from user space to kernel;
      adjust data structures: enqueue message header,
                  message header points to data,
                  counts, time stamps, process ID;
      wakeup all processes waiting to read message from queue;
}
```

Figure 11.4. Algorithm for Msgsnd

size in the message header, sets the message header to point to the message data, and updates various statistics fields (number of messages and bytes on queue, time stamps and process ID of sender) in the queue header. The kernel then awakens processes that were asleep, waiting for messages to arrive on the queue. If the number of bytes on the queue exceeds the queue's limit, the process sleeps until other messages are removed from the queue. If the process specified not to wait (flag IPC_NOWAIT), however, it returns immediately with an error indication. Figure 11.5 depicts messages on a queue, showing queue headers, linked lists of message headers, and pointers from the message headers to a data area.

Consider the program in Figure 11.6: A process calls *msgget* to get a descriptor for *MSGKEY*. It sets up a message of length 256 bytes, although it uses only the first integer, copies its process ID into the message text, assigns the message type value 1, then calls *msgsnd* to send the message. We will return to this example later.

A process receives messages by

```
count = msgrcv(id, msg, maxcount, type, flag);
```

where *id* is the message descriptor, *msg* is the address of a user structure to contain the received message, *maxcount* is the size of the data array in *msg*, *type* specifies the message type the user wants to read, and *flag* specifies what the kernel should

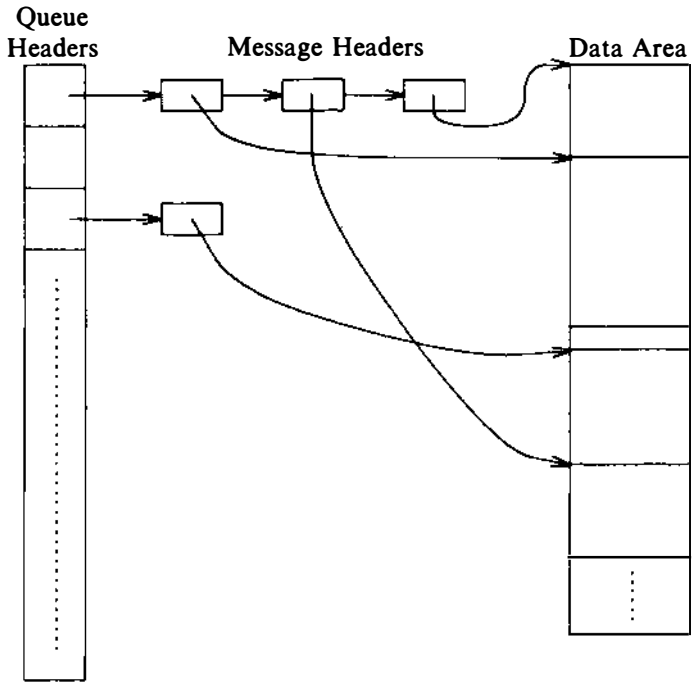

Figure 11.5. Data Structures for Messages

do if no messages are on the queue. The return value, *count*, is the number of bytes returned to the user.

The kernel checks (Figure 11.7) that the user has the necessary access rights to the message queue, as above. If the requested message *type* is 0, the kernel finds the first message on the linked list. If its size is less than or equal to the size requested by the user, the kernel copies the message data to the user data structure and adjusts its internal structures appropriately: It decrements the count of messages on the queue and the number of data bytes on the queue, sets the receive time and receiving process ID, adjusts the linked list, and frees the kernel space that had stored the message data. If processes were waiting to send messages because there was no room on the list, the kernel awakens them. If the message is bigger than *maxcount* specified by the user, the kernel returns an error for the system call and leaves the message on the queue. If the process ignores size constraints, however (bit *MSG_NOERROR* is set in *flag*), the kernel truncates the message, returns the requested number of bytes, and removes the entire message from the list.

```
#include      <sys/types.h>
#include      <sys/ipc.h>
#include      <sys/msg.h>

#define MSGKEY      75

struct msgform {
      long      mtype;
      char      mtext[256];
};

main()
{
      struct msgform msg;
      int msgid, pid, *pint;

      msgid = msgget(MSGKEY, 0777);

      pid = getpid();
      pint = (int *) msg.mtext;
      *pint = pid;      /* copy pid into message text */
      msg.mtype = 1;

      msgsnd(msgid, &msg, sizeof(int), 0);
      msgrcv(msgid, &msg, 256, pid, 0);      /* pid is used as the msg type */
      printf("client: receive from pid %d\n", *pint);
}
```

Figure 11.6. A Client Process

A process can receive messages of a particular type by setting the *type* parameter appropriately. If it is a positive integer, the kernel returns the first message of the given type. If it is negative, the kernel finds the lowest type of all messages on the queue, provided it is less than or equal to the absolute value of *type*, and returns the first message of that type. For example, if a queue contains three messages whose types are 3, 1, and 2, respectively, and a user requests a message with type −2, the kernel returns the message of type 1. In all cases, if no messages on the queue satisfy the receive request, the kernel puts the process to sleep, unless the process had specified to return immediately by setting the *IPC_NOWAIT* bit in *flag*.

Consider the programs in Figures 11.6 and 11.8. The program in Figure 11.8 shows the structure of a *server* that provides generic service to *client* processes. For instance, it may receive requests from client processes to provide information from a database; the server process is a single point of access to the database, making consistency and security easier. The server creates a message structure by setting

```
algorithm msgrcv                /* receive message */
input:  (1) message descriptor
        (2) address of data array for incoming message
        (3) size of data array
        (4) requested message type
        (5) flags
output: number of bytes in returned message
{
        check permissions;
    loop:
        check legality of message descriptor;
        /* find message to return to user */
        if (requested message type == 0)
                consider first message on queue;
        else if (requested message type > 0)
                consider first message on queue with given type;
        else    /* requested message type < 0 */
                consider first of the lowest typed messages on queue,
                        such that its type is <= absolute value of
                        requested type;
        if (there is a message)
        {
                adjust message size or return error if user size too small;
                copy message type, text from kernel space to user space;
                unlink message from queue;
                return;
        }
        /* no message */
        if (flags specify not to sleep)
                return with error;
        sleep (event message arrives on queue);
        goto loop;
}
```

Figure 11.7. Algorithm for Receiving a Message

the *IPC_CREAT* flag in the *msgget* call and receives all messages of type 1 — requests from client processes. It reads the message text, finds the process ID of the client process, and sets the return message type to the client process ID. In this example, it sends its process ID back to the client process in the message text, and the client process receives messages whose message type equals its process ID. Thus, the server process receives only messages sent to it by client processes, and client processes receive only messages sent to them by the server. The processes cooperate to set up multiple channels on one message queue.

```
#include      <sys/types.h>
#include      <sys/ipc.h>
#include      <sys/msg.h>

#define MSGKEY      75
struct msgform
{
      long mtype;
      char mtext[256];
} msg;
int msgid;

main()
{
      int i, pid, *pint;
      extern cleanup();

      for (i = 0;  i < 20;  i++)
            signal(i, cleanup);
      msgid = msgget(MSGKEY, 0777 | IPC_CREAT);

      for (;;)
      {
            msgrcv(msgid, &msg, 256, 1, 0);
            pint = (int *) msg.mtext;
            pid = *pint;
            printf("server: receive from pid %d\n", pid);
            msg.mtype = pid;
            *pint = getpid();
            msgsnd(msgid, &msg, sizeof(int), 0);
      }
}

cleanup()
{
      msgctl(msgid, IPC_RMID, 0);
      exit();
}
```

Figure 11.8. A Server Process

Messages are formatted as type-data pairs, whereas file data is a byte stream. The *type* prefix allows processes to select messages of a particular type, if desired, a feature not readily available in the file system. Processes can thus extract messages of particular types from the message queue in the order that they arrive, and the kernel maintains the proper order. Although it is possible to implement a message

passing scheme at user level with the file system, messages provide applications with a more efficient way to transfer data between processes.

A process can query the status of a message descriptor, set its status, and remove a message descriptor with the *msgctl* system call. The syntax of the call is

```
msgctl(id, cmd, mstatbuf)
```

where *id* identifies the message descriptor, *cmd* specifies the type of command, and *mstatbuf* is the address of a user data structure that will contain control parameters or the results of a query. The implementation of the system call is straightforward; the appendix specifies the parameters in detail.

Returning to the server example in Figure 11.8, the process catches signals and calls the function *cleanup* to remove the message queue from the system. If it did not catch signals or if it receives a *SIGKILL* signal (which cannot be caught), the message queue would remain in the system even though no processes refer to it. Subsequent attempts to create (exclusively) a new message queue for the given key would fail until it was removed.

11.2.2 Shared Memory

Processes can communicate directly with each other by sharing parts of their virtual address space and then reading and writing the data stored in the *shared memory*. The system calls for manipulating shared memory are similar to the system calls for messages. The *shmget* system call creates a new region of shared memory or returns an existing one, the *shmat* system call logically attaches a region to the virtual address space of a process, the *shmdt* system call detaches a region from the virtual address space of a process, and the *shmctl* system call manipulates various parameters associated with the shared memory. Processes read and write shared memory using the same machine instructions they use to read and write regular memory. After attaching shared memory, it becomes part of the virtual address space of a process, accessible in the same way other virtual addresses are; no system calls are needed to access data in shared memory.

The syntax of the *shmget* system call is

```
shmid = shmget(key, size, flag);
```

where *size* is the number of bytes in the region. The kernel searches the shared memory table for the given *key*: if it finds an entry and the permission modes are acceptable, it returns the descriptor for the entry. If it does not find an entry and the user had set the *IPC_CREAT* flag to create a new region, the kernel verifies that the size is between system-wide minimum and maximum values and then allocates a region data structure using algorithm *allocreg* (Section 6.5.2). The kernel saves the permission modes, size, and a pointer to the region table entry in the shared memory table (Figure 11.9) and sets a flag there to indicate that no memory is associated with the region. It allocates memory (page tables and so on) for the region only when a process attaches the region to its address space. The

kernel also sets a flag on the region table entry to indicate that the region should not be freed when the last process attached to it *exit*s. Thus, data in shared memory remains intact even though no processes include it as part of their virtual address space.

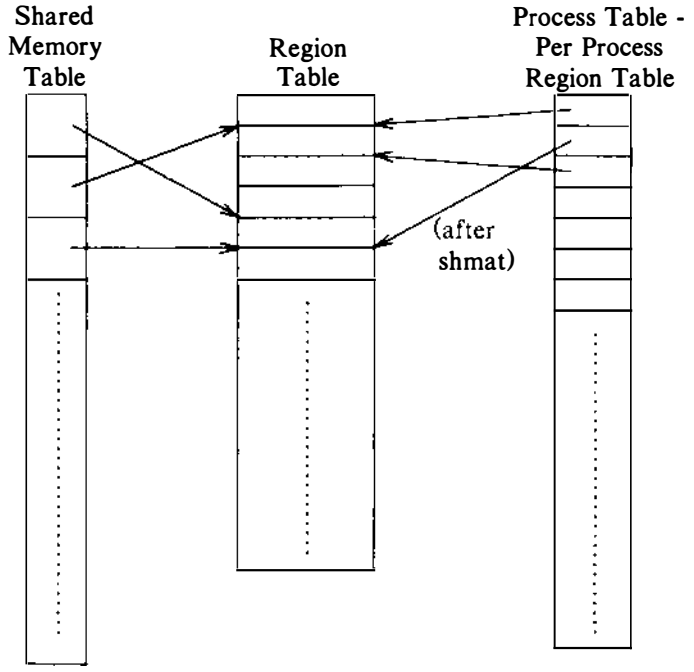

Figure 11.9. Data Structures for Shared Memory

A process attaches a shared memory region to its virtual address space with the *shmat* system call:

```
virtaddr = shmat(id, addr, flags);
```

Id, returned by a previous *shmget* system call, identifies the shared memory region, *addr* is the virtual address where the user wants to attach the shared memory, and *flags* specify whether the region is read-only and whether the kernel should round off the user-specified address. The return value, *virtaddr*, is the virtual address where the kernel attached the region, not necessarily the value requested by the process.

When executing the *shmat* system call, the kernel verifies that the process has the necessary permissions to access the region (Figure 11.10). It examines the address the user specifies: If 0, the kernel chooses a convenient virtual address.

```
algorithm shmat                 /* attach shared memory */
input:  (1) shared memory descriptor
        (2) virtual address to attach memory
        (3) flags
output: virtual address where memory was attached
{
        check validity of descriptor, permissions;
        if (user specified virtual address)
        {
                round off virtual address, as specified by flags;
                check legality of virtual address, size of region;
        }
        else        /* user wants kernel to find good address */
                kernel picks virtual address: error if none available;
        attach region to process address space (algorithm attachreg);
        if (region being attached for first time)
                allocate page tables, memory for region
                                        (algorithm growreg);
        return(virtual address where attached);
}
```

Figure 11.10. Algorithm for Attaching Shared Memory

The shared memory must not overlap other regions in the process virtual address space; hence it must be chosen judiciously so that other regions do not grow into the shared memory. For instance, a process can increase the size of its data region with the *brk* system call, and the new data region is virtually contiguous with the previous data region; therefore, the kernel should not attach a shared memory region close to the data region. Similarly, it should not place shared memory close to the top of the stack so that the stack will not grow into it. For example, if the stack grows towards higher addresses, the best place for shared memory is immediately before the start of the stack region.

The kernel checks that the shared memory region fits into the process address space and attaches the region, using algorithm *attachreg*. If the calling process is the first to attach the region, the kernel allocates the necessary tables, using algorithm *growreg*, adjusts the shared memory table entry field for "last time attached," and returns the virtual address at which it attached the region.

A process detaches a shared memory region from its virtual address space by

```
shmdt(addr)
```

where *addr* is the virtual address returned by a prior *shmat* call. Although it would seem more logical to pass an identifier, the virtual address of the shared memory is used so that a process can distinguish between several instances of a shared memory region that are attached to its address space, and because the

identifier may have been removed. The kernel searches for the process region attached at the indicated virtual address and detaches it from the process address space, using algorithm *detachreg* (Section 6.5.7). Because the region tables have no back pointers to the shared memory table, the kernel searches the shared memory table for the entry that points to the region and adjusts the field for the time the region was last detached.

Consider the program in Figure 11.11: A process creates a 128K-byte shared memory region and attaches it twice to its address space at different virtual addresses. It writes data in the "first" shared memory and reads it from the "second" shared memory. Figure 11.12 shows another process attaching the same region (it gets only 64K bytes, to show that each process can attach different amounts of a shared memory region); it waits until the first process writes a nonzero value in the first word of the shared memory region and then reads the shared memory. The first process *pause*s to give the second process a chance to execute; when the first process catches a signal, it removes the shared memory region.

A process uses the *shmctl* system call to query status and set parameters for the shared memory region:

```
shmctl(id, cmd, shmstatbuf);
```

Id identifies the shared memory table entry, *cmd* specifies the type of operation, and *shmstatbuf* is the address of a user-level data structure that contains the status information of the shared memory table entry when querying or setting its status. The kernel treats the commands for querying status and changing owner and permissions similar to the implementation for messages. When removing a shared memory region, the kernel frees the entry and looks at the region table entry: If no process has the region attached to its virtual address space, it frees the region table entry and all its resources, using algorithm *freereg* (Section 6.5.6). If the region is still attached to some processes (its reference count is greater than 0), the kernel just clears the flag that indicates the region should not be freed when the last process detaches the region. Processes that are using the shared memory may continue doing so, but no new processes can attach it. When all processes detach the region, the kernel frees the region. This is analogous to the case in the file system where a process can *open* a file and continue to access it after it is *unlink*ed.

11.2.3 Semaphores

The semaphore system calls allow processes to synchronize execution by doing a set of operations atomically on a set of semaphores. Before the implementation of semaphores, a process would create a lock file with the *creat* system call if it wanted to lock a resource: The *creat* fails if the file already exists, and the process would assume that another process had the resource locked. The major disadvantages of this approach are that the process does not know when to try again, and lock files may inadvertently be left behind when the system crashes or is