UNIT-VI

Semaphores, Shared Memory, and Message Queues

Semaphores

- When you write programs that use threads operating in multiuser systems, multiprocessing systems, or a combination of the two, you may often discover that you have *critical sections* of code,
- where you need to ensure that a single process (or a single thread of execution) has exclusive access to a resource.
- To prevent problems caused by more than one program simultaneously accessing a shared resource, you need a way of generating and using a token that grants access to only one thread of execution in a critical section at a time.

Continues…..

- It’s surprisingly difficult to write general-purpose code that ensures that one program has exclusive access to a particular resource, although there’s a solution known as Dekker’s Algorithm. Unfortunately, this algorithm relies on a “busy wait,” or “spin lock,” where a process runs continuously, waiting for a memory location to be changed. In a multitasking environment such as Linux, this is an undesirable waste of CPU resources.
- One possible solution that you’ve already seen is to create files using the O_EXCL flag with the open function, which provides atomic file creation. This allows a single process to succeed in obtaining a token: then newly created file. This method is fine for simple problems, but rather messy and very inefficient for more complex examples.

- A semaphore is a special variable that takes only whole positive numbers and upon which programs can only act atomically. A more formal definition of a semaphore is a special variable on which only two operations are allowed; these operations are officially termed *wait and signal*. Because "wait" and "signal" already have special meanings in Linux programming, we'll use the original notation:

❑ P(semaphore variable) for wait

❑ V(semaphore variable) for signal

- These letters come from the Dutch words for wait (*passeren*: to pass, as in a checkpoint before the critical section) and signal (*vrijgeven*: to give or release, as in giving up control of the critical section).
- The simplest semaphore is a variable that can take only the values 0 and 1, a *binary semaphore*. This is the most common form. Semaphores that can take many positive values are called *general semaphores*.

The definitions of P and V are surprisingly simple. Suppose you have a semaphore variable sv. The two operations are then defined as follows: P(sv) If sv is greater than zero, decrement sv. If sv is zero, suspend execution of this process.

- V(sv) If some other process has been suspended waiting for sv, make it resume execution. If no process is suspended waiting for sv, increment sv.
- The semaphore variable, sv, is true when the critical section is available, is decremented by P(sv) so it’s false when the critical section is busy, and is incremented by V(sv) when the critical section is again available. Be aware that simply having a normal variable that you decrement and increment is not good enough, because you can’t express in C, C++, C#, or almost any conventional programming language the need to make a single, atomic operation of the test to see whether the variable is true, and if so change the variable to make it false.

A Theoretical Example

- Suppose you have two processes proc1 and proc2, both of which need exclusive access to a database at some point in their execution. You define a single binary semaphore, sv, which starts with the value 1 and can be accessed by both processes. Both processes then need to perform the same processing to access the critical section of code; indeed, the two processes could simply be different invocations of the same program.
- The two processes share the sv semaphore variable. Once one process has executed P(sv), it has obtained the semaphore and can enter the critical section. The second process is prevented from entering the critical section because when it attempts to execute P(sv), it's made to wait until the first process has left the critical section and executed V(sv) to release the semaphore.

Continues…..

- The required pseudo code is identical for both processes:
- semaphore sv = 1;
- loop forever {
- P(sv);
- critical code section;
- V(sv);
- noncritical code section;
- }

Linux Semaphore Facilities

- All the Linux semaphore functions operate on arrays of general semaphores rather than a single binary semaphore. At first sight, this just seems to make things more complicated, but in complex cases where a process needs to lock multiple resources, the ability to operate on an array of semaphores is a big advantage.
- The semaphore function definitions are
- **#include <sys/sem.h>**
- **int semctl(int sem_id, int sem_num, int command, ...);**
- **int semget(key_t key, int num_sems, int sem_flags);**
- **int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops);**

Continues………….

- As you work through each function in turn, remember that these functions were designed to work for arrays of semaphore values, which makes their operation significantly more complex than would have been required for a single semaphore.
- Notice that key acts very much like a filename in that it represents a resource that programs may use and cooperate in using if they agree on a common name. Similarly, the identifier returned by semget and used by the other shared memory functions is very much like the FILE * file stream returned by fopen in that it’s a value used by the process to access the shared file. Just as with files different processes will have different semaphore identifiers, though they refer to the same semaphore. This use of a key and identifiers is common to all of the IPC facilities discussed here, although each facility uses independent keys and identifiers.

semget

- The semget function creates a new semaphore or obtains the semaphore key of an existing semaphore:
- **int semget(key_t key, int num_sems, int sem_flags);**
- The first parameter, key, is an integral value used to allow unrelated processes to access the same semaphore.All semaphores are accessed indirectly by the program supplying a key, for which the system then generates a semaphore identifier. The semaphore key is used only with semget. All other semaphore functions use the semaphore identifier returned from semget.
- There is a special semaphore key value, IPC_PRIVATE, that is intended to create a semaphore that only the creating process could access.

semget

- The num_sems parameter is the number of semaphores required. This is almost always 1.
- The sem_flags parameter is a set of flags, very much like the flags to the open function. The lower nine bits are the permissions for the semaphore, which behave like file permissions. In addition, these can be bitwise ORed with the value IPC_CREAT to create a new semaphore. It’s not an error to have the IPC_CREAT flag set and give the key of an existing semaphore.
- The semget function returns a positive (nonzero) value on success; this is the semaphore identifier used in the other semaphore functions. On error, it returns –1.

semop

- **The function semop is used for changing the value of the semaphore:**
- **int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops);**
- The first parameter, sem_id, is the semaphore identifier, as returned from semget. The second parameter, sem_ops, is a pointer to an array of structures, each of which will have at least the following members:
- **struct sembuf {**
- **short sem_num;**
- **short sem_op;**
- **short sem_flg;**
- **}**

semop

- The first member, sem_num, is the semaphore number, usually 0 unless you’re working with an array of semaphores. The sem_op member is the value by which the semaphore should be changed .
- In general, only two values are used, –1, which is your P operation to wait for a semaphore to become available, and +1, which is your V operation to signal that a semaphore is now available.
- The final member, sem_flg, is usually set to SEM_UNDO. This causes the operating system to track the changes made to the semaphore by the current process and, if the process terminates without releasing the semaphore, allows the operating system to automatically release the semaphore if it was held by this process.

semctl

- **The semctl function allows direct control of semaphore information:int semctl(int sem_id, int sem_num, int command, ...);**
- The first parameter, sem_id, is a semaphore identifier, obtained from semget. The sem_num parameter is the semaphore number. You use this when you’re working with arrays of semaphores. Usually, this is 0, the first and only semaphore. The command parameter is the action to take, and a fourth parameter, if present, is a union semun, which according to the X/OPEN specification must have at least the following members:
- **union semun {   int val;**
- **struct semid_ds *buf;**
- **unsigned short *array;**
- **}**

semctl

- Most versions of Linux have a definition of the semun union in a header file (usually sem.h), though X/Open does say that you have to declare your own.
- There are many different possible values of command allowed for semctl. Only the two that we describe here are commonly used
- ❑ SETVAL: Used for initializing a semaphore to a known value. The value required is passed as the val member of the union semun. This is required to set the semaphore up before it’s used for the first time.
- ❑ IPC_RMID: Used for deleting a semaphore identifier when it’s no longer required.
- The semctl function returns different values depending on the command parameter. For SETVAL and IPC_RMID it returns 0 for success and –1 on error.

semun structure

- union semun

 {

- int val; /* Value for SETVAL */ struct semid_ds *buf; /* Buffer for IPC_STAT, IPC_SET */ unsigned short *array; /* Array for GETALL, SETALL */ struct seminfo *__buf; /* Buffer for IPC_INFO (Linux-specific) */

};

Program using semaphore

- **1. After the system #includes, you include a file semun.h. This defines the union semun, as required by X/OPEN, if the system include sys/sem.h doesn’t already define it. Then come the function prototypes, and the global variable, before you come to the main function. There the semaphore is created with a call to semget, which returns the semaphore ID. If the program is the first to be called (that is, it’s called with a parameter and argc > 1), a call is made to set_semvalue to initialize the semaphore and op_char is set to X:**

- #include <unistd.h>
- #include <stdlib.h>
- #include <stdio.h>
- #include <sys/sem.h>
- #include “semun.h”
- static int set_semvalue(void);
- static void del_semvalue(void);
- static int semaphore_p(void);
- static int semaphore_v(void);
- static int sem_id;
- int main(int argc, char *argv[])
- {
- int i;
- int pause_time;
- char op_char = ‘O’;

Continues….

- srand((unsigned int)getpid());
- sem_id = semget((key_t)1234, 1, 0666 | IPC_CREAT);
- if (argc > 1) {
- if (!set_semvalue()) {
- fprintf(stderr, “Failed to initialize semaphore\n”);
- exit(EXIT_FAILURE);   }
- op_char = ‘X’;
- sleep(2);   }
- 2. **Then you have a loop that enters and leaves the critical section 10 times. There you first make a call to semaphore_p, which sets the semaphore to wait as this program is about to enter the critical section:**
- for(i = 0; i < 10; i++) {
- if (!semaphore_p()) exit(EXIT_FAILURE);
- printf(“%c”, op_char);fflush(stdout);
- pause_time = rand() % 3;
- sleep(pause_time);
- printf(“%c”,op_char);fflush(stdout);
- 3. **After the critical section, you call semaphore_v, setting the semaphore as available, before going through the for loop again after a random wait. After the loop, the call to del_semvalue is made to clean up the code:**

Continues…

- if (!semaphore_v()) exit(EXIT_FAILURE);
- pause_time = rand() % 2;
- sleep(pause_time);
- } printf("\n%d - finished\n", getpid());
- if (argc > 1) {
- sleep(10);
- del_semvalue();
- } exit(EXIT_SUCCESS); }
- 4. **The function set_semvalue initializes the semaphore using the SETVAL command in a semctl**

  call. You need to do this before

  you can use the semaphore:

- static int set_semvalue(void)
- {
- union semun sem_union;
- sem_union.val = 1;
- if (semctl(sem_id, 0, SETVAL, sem_union) == -1) return(0);
- return(1); }
- 5. **The del_semvalue function has almost the same form, except that the call to semctl uses the command IPC_RMID to remove the semaphore's ID:**
- static void del_semvalue(void) {
- union semun sem_union;
- if (semctl(sem_id, 0, IPC_RMID, sem_union) == -1)
- fprintf(stderr, "Failed to delete semaphore\n"); }

Continues.....

- **6. semaphore_p changes the semaphore by –1. This is the "wait" operation:**
- static int semaphore_p(void)
- {
- struct sembuf sem_b;
- sem_b.sem_num = 0;
- sem_b.sem_op = -1; /* P() */
- sem_b.sem_flg = SEM_UNDO;
- if (semop(sem_id, &sem_b, 1) == -1) { fprintf(stderr, "semaphore_p failed\n");
- return(0);
- } return(1); }

- **7. semaphore_v is similar except for setting the sem_op part of the sembuf structure to 1. This is the "release" operation, so that the semaphore becomes available:**
- static int semaphore_v(void)
- { struct sembuf sem_b;
- sem_b.sem_num = 0;
- sem_b.sem_op = 1; /* V() */
- sem_b.sem_flg = SEM_UNDO;
- if (semop(sem_id, &sem_b, 1) == -1) {
- fprintf(stderr, "semaphore_v failed\n");
- return(0); } return(1); }

Execution of the above code

- Here’s some sample output, with two invocations of the program.
- $ **cc sem1.c -o sem1**
- $ **./sem1 1 &**
- [1] 1082
- $ **./sem1**
- OOXXOOXXOOXXOOXXOOX XOOOOXXOOXXOOXXOOXX XX
- 1083 - finished
- 1082 - finished

- Remember that“O” represents the first invocation of the program, and “X” the second invocation of the program. Because each program prints a character as it enters and again as it leaves the critical section, each character should only appear as part of a pair. As you can see, the Os and Xs are indeed properly paired, indicating that the critical section is being correctly processed.

How above code Works

- The program starts by obtaining a semaphore identity from the (arbitrary) key that you’ve chosen using the semget function. The IPC_CREAT flag causes the semaphore to be created if one is required.
- If the program has a parameter, it’s responsible for initializing the semaphore, which it does with the function set_semvalue, a simplified interface to the more general semctl function. It also uses the presence of the parameter to determine which character it should print out. The sleep simply allows you some time to invoke other copies of the program before this copy gets to execute too many times around its loop. You use srand and rand to introduce some pseudo-random timing into the program.
- The program then loops 10 times, with pseudo-random waits in its critical and noncritical sections. The critical section is guarded by calls to your semaphore_p and semaphore_v functions,which are simplified interfaces to the more general semop function.
- Before it deletes the semaphore, the program that was invoked with a parameter then waits to allow other invocations to complete. If the semaphore isn’t deleted, it will continue to exist in the system even though no programs are using it. In real programs, it’s very important to ensure you don’t unintentionally leave semaphores around after execution.

Shared Memory

- Shared memory is the second of the three IPC facilities. It allows two unrelated processes to access the same logical memory. Shared memory is a very efficient way of transferring data between two running processes.
- Shared memory is a special range of addresses that is created by IPC for one process and appears in the address space of that process. Other processes can then “attach” the same shared memory segment into their own address space. All processes can access the memory locations just as if the memory had been allocated by malloc. If one process writes to the shared memory, the changes immediately become visible to any other process that has access to the same shared memory.

Continues…..

- Shared memory provides an efficient way of sharing and passing data between multiple processes. By itself, shared memory doesn’t provide any synchronization facilities. Because it provides no synchronization facilities, you usually need to use some other mechanism to synchronize access to the shared memory. Typically, you might use shared memory to provide efficient access to large areas of memory and pass small messages to synchronize access to that memory.

- There are no automatic facilities to prevent a second process from starting to read the shared memory before the first process has finished writing to it. It’s the responsibility of the programmer to synchronize access.

- The functions for shared memory resemble those for semaphores:

#include <sys/shm.h>

- void *shmat(int shm_id, const void *shm_addr, int shmflg);
- int shmctl(int shm_id, int cmd, struct shmid_ds *buf);
- int shmdt(const void *shm_addr);
- int shmget(key_t key, size_t size, int shmflg);

- As with semaphores, the include files sys/types.h and sys/ipc.h are normally automatically included by shm.h.

shmget

- You create shared memory using the shmget function:
- **int shmget(key_t key, size_t size, int shmflg);**
- As with semaphores, the program provides key, which effectively names the shared memory segment,and the shmget function returns a shared memory identifier that is used in subsequent shared memory functions. There's a special key value, IPC_PRIVATE, that creates shared memory private to the process.
- The second parameter, size, specifies the amount of memory required in bytes.
- The third parameter, shmflg, consists of nine permission flags that are used in the same way as the modeflags for creating files. A special bit defined by IPC_CREAT must be bitwise ORed with the permissions to create a new shared memory segment.

shmget

- The permission flags are very useful with shared memory because they allow a process to create shared memory that can be written by processes owned by the creator of the shared memory, but only read by processes that other users have created. You can use this to provide efficient read-only access to data by placing it in shared memory without the risk of its being changed by other users.
- If the shared memory is successfully created, shmget returns a nonnegative integer, the shared memory identifier. On failure, it returns –1.

shmat

- When you first create a shared memory segment, it’s not accessible by any process. To enable access to the shared memory, you must attach it to the address space of a process. You do this with the shmat function:
- **void *shmat(int shm_id, const void *shm_addr, int shmflg);**
- The first parameter, shm_id, is the shared memory identifier returned from shmget.
- The second parameter, shm_addr, is the address at which the shared memory is to be attached to the current process.
- The third parameter, shmflg, is a set of bitwise flags. The two possible values are SHM_RND, which, in conjunctionwith shm_addr, controls the address at which the shared memory is attached, and SHM_RDONLY, which makes the attached memory read-only.

shmat

- If the shmat call is successful, it returns a pointer to the first byte of shared memory. On failure –1 is returned.
- The shared memory will have read or write access depending on the owner (the creator of the shared memory), the permissions, and the owner of the current process. Permissions on shared memory are similar to the permissions on files.

Shmdt

- The shmdt function detaches the shared memory from the current process. It takes a pointer to the address returned by shmat. On success, it returns 0, on error –1.
- Note that detaching the shared memory doesn’t delete it; it just makes that memory unavailable to the current process.

shmctl

- The control functions for shared memory are (thankfully) somewhat simpler than the more complex ones for semaphores:
- **int shmctl(int shm_id, int command, struct shmid_ds *buf);**
- The shmid_ds structure has at least the following members:
- **struct shmid_ds { uid_t shm_perm.uid; uid_t shm_perm.gid;**
- **mode_t shm_perm.mode; }**
- The first parameter, shm_id, is the identifier returned from shmget.
- The second parameter, command, is the action to take. It can take three values.
- The third parameter, buf, is a pointer to the structure containing the modes and permissions for the shared memory. On success, it returns 0, on failure, –1.

Shared memory program

- Now that you’ve seen the shared memory functions, you can write some code to use them. In this Try It Out, you write a pair of programs, shm1.c and shm2.c.
- The first (the consumer) will create a shared memory segment and then display any data that is written into it. The second (the producer) will attach to an existing shared memory segment and allow you to enter data into that segment.
- 1. First create a common header file to describe the shared memory you want to pass around. Call this shm_com.h:

shm_com.h and shm1.c

- #define TEXT_SZ 2048
- struct shared_use_st {
- int written_by_you;
- char some_text[TEXT_SZ];
- };
- This defines a structure to use in both the consumer and producer programs. You use an int flag written_by_you to tell the consumer when data has been written to the rest of the structure and arbitrarily decide that you need to transfer up to 2k of text.
- 2. The first program, shm1.c, is the consumer. After the headers, the shared memory segment (the size of your shared memory structure) is created with a call to shmget, with the IPC_CREAT
- bit specified:
- #include <unistd.h>
- #include <stdlib.h>
- #include <stdio.h>
- #include <string.h>
- #include <sys/shm.h>

shm1.c Continues…………..

- #include “shm_com.h”
- int main()
- {
- int running = 1;
- void *shared_memory = (void *)0;
- struct shared_use_st *shared_stuff;
- int shmid;
- srand((unsigned int)getpid());
- shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);
- if (shmid == -1) {
- fprintf(stderr, “shmget failed\n”);
- exit(EXIT_FAILURE);
- }

- **3. You now make the shared memory accessible to the program:**
- shared_memory = shmat(shmid, (void *)0, 0);
- if (shared_memory == (void *)-1) {
- fprintf(stderr, “shmat failed\n”);
- exit(EXIT_FAILURE);
- }
- printf(“Memory attached at %X\n”, (int)shared_memory);
- 4. The next portion of the program assigns the shared_memory segment to shared_stuff, which then prints out any text in written_by_you. The loop continues until end is found in written_by_you. The call to sleep forces the consumer to sit in its critical section, which
- makes the producer wait:

Continues…………

- shared_stuff = (struct shared_use_st *)shared_memory;
- shared_stuff->written_by_you = 0;
- while(running) {
- if (shared_stuff->written_by_you) {
- printf(“You wrote: %s”, shared_stuff->some_text);
- sleep(rand() % 4); /* make the other process wait for us ! */
- shared_stuff->written_by_you = 0;
- if (strncmp(shared_stuff->some_text, “end”, 3) == 0) {
- running = 0;
- }
- }
- }
- **5. Finally, the shared memory is detached and then deleted:**
- if (shmdt(shared_memory) == -1) {
- fprintf(stderr, “shmdt failed\n”);
- exit(EXIT_FAILURE);
- }
- if (shmctl(shmid, IPC_RMID, 0) == -1) {
- fprintf(stderr, “shmctl(IPC_RMID) failed\n”);
- exit(EXIT_FAILURE);
- }
- exit(EXIT_SUCCESS);
- }

Continues…………..

- **6. The second program, shm2.c, is the producer; it allows you to enter data for consumers. It's very similar to shm1.c and looks like this:**
- #include <unistd.h>
- #include <stdlib.h>
- #include <stdio.h>
- #include <string.h>
- #include <sys/shm.h>
- #include "shm_com.h"
- int main()
- {
- int running = 1;
- void *shared_memory = (void *)0;
- struct shared_use_st *shared_stuff;
- char buffer[BUFSIZ];
- int shmid;
- shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);
- if (shmid == -1) {
- fprintf(stderr, "shmget failed\n");
- exit(EXIT_FAILURE);
- }
- shared_memory = shmat(shmid, (void *)0, 0);
- if (shared_memory == (void *)-1) {
- fprintf(stderr, "shmat failed\n");
- exit(EXIT_FAILURE);
- }
- printf("Memory attached at %X\n", (int)shared_memory);
- shared_stuff = (struct shared_use_st *)shared_memory;

Continues……….

- shared_stuff = (struct shared_use_st *)shared_memory;
- while(running) {
- while(shared_stuff->written_by_you == 1) {
- sleep(1);
- printf(“waiting for client...\n”);
- }
- printf(“Enter some text: “);
- fgets(buffer, BUFSIZ, stdin);
- strncpy(shared_stuff->some_text, buffer, TEXT_SZ);
- shared_stuff->written_by_you = 1;
- if (strncmp(buffer, “end”, 3) == 0) {
- running = 0;
- }
- }
- if (shmdt(shared_memory) == -1) {
- fprintf(stderr, “shmdt failed\n”);
- exit(EXIT_FAILURE);
- }
- exit(EXIT_SUCCESS);
- }

Output of the above code

- When you run these programs, you get sample output such as this:
- $ **./shm1 &**
- [1] 294
- Memory attached at 40017000
- $ **./shm2**
- Memory attached at 40017000
- Enter some text: **hello**
- You wrote: hello
- waiting for client...
- waiting for client...
- Enter some text: **Linux!**
- You wrote: Linux!
- waiting for client...
- waiting for client...
- waiting for client...
- Enter some text: **end**
- You wrote: end
- $

How above code Works

- The first program, shm1, creates the shared memory segment and then attaches it to its address space. You impose a structure, **shared_use_st** on the first part of the shared memory. This has **a flag,written_by_you,** which is set when data is available. When this flag is set, the program reads the text, prints it out, and clears the flag to show it has read the data. Use the special string, end, to allow a clean exit from the loop. The program then detaches the shared memory segment and deletes it.
- The second program, shm2, gets and attaches the same shared memory segment, because it uses the same key, 1234. It then prompts the user to enter some text. If the **flag written_by_you** is set, shm2 knows that the client process hasn’t yet read the previous data and waits for it. When the other process clears this flag, shm2 writes the new data and sets the flag. It also uses the magic string end to terminate and detach the shared memory segment.

Message Queues

- Message queues are like named pipes, but without the complexity associated with opening and closing the pipe.

Message queues provide a reasonably easy and efficient way of passing data between two unrelated processes. They have the advantage over named pipes that the message queue exists independently of both the sending and receiving processes, which removes some of the difficulties that occur in synchronizing the opening and closing of named pipes.

- Message queues provide a way of sending a block of data from one process to another. Additionally, each block of data is considered to have a type, and a receiving process may receive blocks of data having different type values independently. The good news is that you can almost totally avoid the synchronization and blocking problems of named pipes by sending messages. Even better, you can“look ahead” for messages that are urgent in some way. The bad news is that, just like pipes, there’s a maximum size limit imposed on each block of data and also a limit on the maximum total size of all blocks on all queues throughout the system.

Message Queue continues…..

- Linux does have two defines, MSGMAX and MSGMNB, which define the maximum size in bytes of an individual message and the maximum size of a queue, respectively.
- The message queue function definitions are:
- **#include <sys/msg.h>**
- **int msgctl(int msqid, int cmd, struct msqid_ds *buf);**
- **int msgget(key_t key, int msgflg);**
- **int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);**
- **int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);**
- As with semaphores and shared memory, the include files sys/types.h and sys/ipc.h are normally automatically included by msg.h.

msgget()

- You create and access a message queue using the msgget function:
- **int msgget(key_t key, int msgflg);**
- The program must provide a key value that, as with other IPC facilities, names a particular message queue. The special value IPC_PRIVATE creates a private queue, which in theory is accessible only by the current process.
- The second parameter, msgflg, consists of nine permission flags. A special bit defined by IPC_CREAT must be bitwise ORed with the permissions to create a new message queue
- The msgget function returns a positive number, the queue identifier, on success or –1 on failure.

msgsnd()

- The msgsnd function allows you to add a message to a message queue:

int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);

- The structure of the message is constrained in two ways. First, it must be smaller than the system limit, and second, it must start with a long int, which will be used as a message type in the receive function. When you're using messages, it's best to define your message structure something like this:

```
struct my_message {
long int message_type;        /* The data you wish to transfer */
 }
```

- The first parameter, msqid, is the message queue identifier returned from a msgget function.The second parameter, msg_ptr, is a pointer to the message to be sent, which must start with a long int type as described previously.The third parameter, msg_sz, is the size of the message pointed to by msg_ptr. This size must notinclude the long int message type.
- The fourth parameter, msgflg, controls what happens if either the current message queue is full or the system wide limit on queued messages has been reached. If msgflg has the IPC_NOWAIT flag set, the function will return immediately without sending the message and the return value will be –1.If the msgflg has the IPC_NOWAIT flag clear, the sending process will be suspended, waiting for space to become available in the queue.
- On success, the function returns 0, on failure –1. If the call is successful, a copy of the message data has been taken and placed on the message queue.

msgrcv()

- The msgrcv function retrieves messages from a message queue:

int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);

- The first parameter, msqid, is the message queue identifier returned from a msgget function
- The second parameter, msg_ptr, is a pointer to the message to be received, which must start with a long int type as described previously in the msgsnd function.
- The third parameter, msg_sz, is the size of the message pointed to by msg_ptr, not including the long int message type.
- On success, msgrcv returns the number of bytes placed in the receive buffer, the message is copied into the user-allocated buffer pointed to by msg_ptr, and the data is deleted from the message queue. It returns –1 on error.

- The fourth parameter, msgtype, is a long int, which allows a simple form of reception priority to be implemented. If msgtype has the value 0, the first available message in the queue is retrieved. If it's greater than zero, the first message with the same message type is retrieved. If it's less than zero, the first message that has a type the same as or less than the absolute value of msgtype is retrieved.

•This sounds more complicated than it actually is in practice. If you simply want to retrieve messages in the order in which they were sent, set msgtype to 0. If you want to retrieve only messages with a specific message type, set msgtype equal to that value. If you want to receive messages with a type of n or smaller, set msgtype to -n.

•The fifth parameter, msgflg, controls what happens when no message of the appropriate type is waiting to be received. If the IPC_NOWAIT flag in msgflg is set, the call will return immediately with a return value of –1. If the IPC_NOWAIT flag

msgctl()

- The final message queue function is msgctl, which is very similar to that of the control function for shared memory:

int msgctl(int msqid, int command, struct msqid_ds *buf);

- The msqid_ds structure has at least the following members:

```
struct msqid_ds {
uid_t  msg_perm.uid;
uid_t msg_perm.gid
mode_t msg_perm.mode;
}
```

- The first parameter, msqid, is the identifier returned from msgget.

- The second parameter, command, is the action to take. It can take three values, described in the following table:

Command	Description
IPC_STAT	Sets the data in the msqid_ds structure to reflect the values associated with the message queue.
IPC_SET	If the process has permission to do so, this sets the values associated with the message queue to those provided in the msqid_ds data structure.
IPC_RMID	Deletes the message queue.0 is returned on success, –1 on failure. If a message queue is deleted while a process is waiting in a msgsnd or msgrcv function, the send or receive function will fail.

Program using Message Queues

- **1. Here’s the receiver program, msg1.c:**
- #include <stdlib.h>
- #include <stdio.h>
- #include <string.h>
- #include <errno.h>
- #include <unistd.h>
- #include <sys/msg.h>
- struct my_msg_st {
- long int my_msg_type;
- char some_text[BUFSIZ];
- };
- int main()
- {
- int running = 1;
- int msgid;
- struct my_msg_st some_data;
- long int msg_to_receive = 0;
- **2. First, set up the message queue:**
- msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
- if (msgid == -1) {
- fprintf(stderr, “msgget failed with error: %d\n”, errno);
- exit(EXIT_FAILURE);
- }
- **3. Then the messages are retrieved from the queue until an end message is encountered. Finally,the message queue is deleted**
- while(running) {
- if (msgrcv(msgid, (void *)&some_data, BUFSIZ,
- msg_to_receive, 0) == -1) {
- fprintf(stderr, “msgrcv failed with error: %d\n”, errno);
- exit(EXIT_FAILURE);
- }
- printf(“You wrote: %s”, some_data.some_text);
- if (strncmp(some_data.some_text, “end”, 3) == 0) {
- running = 0;

Continues…………..

- } }
- if (msgctl(msgid, IPC_RMID, 0) == -1) {
- fprintf(stderr, “msgctl(IPC_RMID) failed\n”);
- exit(EXIT_FAILURE); }
- exit(EXIT_SUCCESS); }
- **4. The sender program, msg2.c, is very similar to msg1.c. In the main setup, delete the msg_to_receive declaration, and replace it with buffer[BUFSIZ]. Remove the message**

queue delete, and make the following changes to the running loop. You now have a call to msgsnd to send the entered text to the queue. The program msg2.c is shown here with the differences from msg1.c highlighted:

- #include <stdlib.h>
- #include <stdio.h>
- #include <string.h>
- #include <errno.h>
- #include <unistd.h>
- #include <sys/msg.h>
- #define MAX_TEXT 512
- struct my_msg_st {
- long int my_msg_type;
- char some_text[MAX_TEXT];
- };
- int main()
- {

Continues……

- int running = 1;
- struct my_msg_st some_data;
- int msgid;
- char buffer[BUFSIZ];
- msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
- if (msgid == -1) {
- fprintf(stderr, “msgget failed with error: %d\n”, errno);
- exit(EXIT_FAILURE);
- }
- while(running) {
- printf(“Enter some text: “);
- fgets(buffer, BUFSIZ, stdin);
- some_data.my_msg_type = 1;
- strcpy(some_data.some_text, buffer);
- if (msgsnd(msgid, (void *)&some_data, MAX_TEXT, 0) == -1) {
- fprintf(stderr, “msgsnd failed\n”);
- exit(EXIT_FAILURE);
- }
- if (strncmp(buffer, “end”, 3) == 0) {
- running = 0;
- }
- }
- exit(EXIT_SUCCESS);
- }
- Unlike in the pipes example, there’s no need for the processes to provide their own synchronization method. This is a significant advantage of messages over pipes.

Output of above code

- **Providing there's room in the message queue, the sender can create the queue, put some data into the queue, and then exit before the receiver even starts. You'll run the sender, msg2, first. Here's some**
- sample output:
- $ **./msg2**
- Enter some text: **hello**
- Enter some text: **How are you today?**
- Enter some text: **end**
- $ **./msg1**
- You wrote: hello
- You wrote: How are you today?
- You wrote: end

- **How It Works**
- **The sender program creates a message queue with msgget; then it adds messages to the queue with**

msgsnd. The receiver obtains the message queue identifier with msgget and then receives messages until the special text end is received. It then tidies up by deleting the message queue with msgctl.

IPC Status Commands

- Most Linux systems provide a set of commands that allow command-line access to IPC information, and to tidy up stray IPC facilities.
- One of the irritations of the IPC facilities is that a poorly written program, or a program that fails for some reason, can leave its IPC resources (such as data in a message queue) loitering on the system long after the program completes. This can cause a new invocation of the program to fail, because the program expects to start with a clean system, but actually finds some left over resource.
- The status (ipcs)and remove (ipcrm) commands provide a way of checking and tidying up IPC facilities.

Displaying Semaphore Status

- To examine the state of semaphores on the system, use the ipcs -s command. If any semaphores are present, the output will have this form:
- $ **./ipcs -s**
- **------ Semaphore Arrays --------**
- **Key   semid   owner   perms   nsems**
- **0x4d00df1a   768   rick   666   1**
- You can use the ipcrm command to remove any semaphores accidentally left by programs. To delete the preceding semaphore, the command (on Linux) is
- $ **./ipcrm -s 768**
- Some much older Linux systems used to use a slightly different syntax: $ **./ipcrm sem 768**

- Like semaphores, many systems provide command-line programs for accessing the details of shared memory. These are ipcs -m and ipcrm -m <id> (or ipcrm shm <id>).
- Here’s some sample output from ipcs -m:
- $ **ipcs -m**
- ------ Shared Memory Segments --------
- key shmid owner perms bytes nattch status
- 0x00000000 384 rick 666 4096 2 dest
- This shows a single shared memory segment of 4 KB attached by two processes.
- The ipcrm -m <id> command allows shared memory to be removed. This is sometimes useful when a program has failed to tidy up shared memory.

Displaying Message Queue Status

- For message queues the commands are ipcs -q and ipcrm -q <id> (or ipcrm msg <id>).
- Here’s some sample output from ipcs -q:
- $ **ipcs -q**
- **------ Message Queues --------**
- **key msqid owner perms used-bytes messages**
- **0x000004d2 3384 rick 666 2048 2**
- This shows two messages, with a total size of 2,048 bytes in a message queue.
- The ipcrm -q <id> command allows a message queue to be removed.

THE END.....