A

Dissertation on

SYSTEM LOG PROCESS FOR APPLICATION SOFTWARE IN LINUX

Submitted in partial fulfillment of the requirements for the award of the degree of

MASTER OF ENGINEERING
(Computer Technology & Applications)

Submitted By :
Dhirender Kumar
College Roll No. 03/CTA/07
Delhi University Roll No. 12202

Under the guidance of :
Mr. Manoj Kumar **(Asstt. Professor)**
Department Of Computer Engineering
Delhi College of Engineering, Delhi.

Department Of Computer Engineering
Delhi College of Engineering
Bawana Road, Delhi-110042
University of Delhi
(2008-2009)

CERTIFICATE

DELHI COLLEGE OF ENGINEERIG
(Govt. of national capital territory of Delhi)
BAWANA ROAD, DELHI- 110042.

It is to certify that the work that is being presented in this project entitled *“**SYSTEM LOG PROCESS FOR APPLICATION SOFTWARE IN LINUX**”*, in partial fulfillment of the requirement for the award of the degree of Master of Engineering in Computer Technology and Application submitted by ***Dhirender Kumar*** (03/CTA/07), is an authentic record of the student’s own work carried out under the supervision and guidance of ***Mr. Manoj Kumar*** , in the Department of Computer Engineering.

Mr. Manoj Kumar
Department Of Computer Engineering
Delhi College of Engineering

ABSTRACT

Operating system observability requires communications with the system log process by the application software. IPC stands for interprocess communication, which describe the different ways of message passing between different processes that are running on some operating systems. The histories of these messages or processes are used in the software testing process. This task is done by log file analyzer. Main objective of this thesis is to propose a methodology of the system log process which is used to make the log files of the different processes running for the different software. System log process contains the two components – message queue operation and process log.

Message queue consist of the method that perform on the message queue. Methods are defined for the creation of message queue, sending and receiving message to / from the message queue. Process log defines the method for processing the log message, it involves the checking of process_id of the receiving log message, buffering of log message and writing the contents of the buffer to the log files.

ACKNOWLEDGEMENT

It is a great pleasure to have the opportunity to extend my heartiest felt gratitude to everybody who helped me throughout the course of this project.
I would like to express my heartiest felt regards to ***Mr. Manoj kumar*** for the constant motivation and support during the duration of this project. It is my privilege and honor to have worked under his supervision. His invaluable guidance and helpful discussions in every stage of this project really helped me in materializing the project.

I would also like to take this opportunity to present my sincere regards to my teachers viz. Dr. Daya Gupta (HOD), Dr. Anita Goel, Mrs. Rajni Jindal, Dr S. K. Saxena, Mr. Manoj Sehti, Ms. Akshi Kumar for their support and encouragement.
I am thankful to my friends and classmates for their unconditional support and motivation during this project.

Dhirender Kumar

M.E. (Computer Technology & Applications)

College Roll No. 03/CTA/07

Delhi University Roll No. 12202

TABLE OF CONTENTS

LIST OF FIGURES & PROGRAMS

LIST OF TABLES

Chapter 1

Introduction

1.1 History

The first computers did not have operating systems. By the early 1960s, commercial computer vendors were supplying quite extensive tools for streamlining the development, scheduling, and execution of jobs on batch processing systems.

The operating systems originally deployed on mainframes, and, much later, the original microcomputer operating systems, only supported one program at a time, requiring only a very basic scheduler. Each program was in complete control of the machine while it was running. Multitasking (timesharing) first came to mainframes in the 1960s.

In 1969-70, UNIX first appeared on the PDP-7 and later the PDP-11. It soon became capable of providing cross-platform time sharing using preemptive multitasking, advanced memory management, memory protection, and a host of other advanced features. UNIX soon gained popularity as an operating system for mainframes and minicomputers alike.

MS-DOS provided many operating system like features, such as disk access. However, many DOS programs bypassed it entirely and ran directly on hardware. IBM's version, PC DOS, ran on IBM microcomputers, including the IBM PC and the IBM PC XT, and MS-DOS came into widespread use on clones of these machines.

IBM PC compatibles could also run Microsoft Xenix, a UNIX-like operating system from the early 1980s. Xenix was heavily marketed by Microsoft as a multi-user alternative to its single user MS-DOS operating system. The CPUs of these personal computers could not facilitate kernel memory protection or provide dual mode operation, so Xenix relied on cooperative multitasking and had no protected memory.

The 80286-based IBM PC AT was the first IBM compatible personal computer capable of using dual mode operation, and providing memory protection. However, the adoption of these features by software vendors was delayed due to numerous bugs in their implementation on the 286, and were only widely accepted with the release of the Intel 80386.

Classic Mac OS, and Microsoft Windows supported only cooperative multitasking (Windows 95, 98, & ME supported preemptive multitasking only when running 32-bit applications, but ran legacy 16-bit applications using cooperative multitasking), and were very limited in their abilities to take advantage of protected memory. Application programs running on these operating systems must yield CPU time to the scheduler when they are not using it, either by default, or by calling a function.

Windows NT's underlying operating system kernel which was a designed by essentially the same team as Digital Equipment Corporation's VMS, a UNIX-like operating system which provided protected mode operation for all user programs, kernel memory protection, preemptive multi-tasking, virtual file system support, and a host of other features.

1.2 Operating system

Operating system (commonly abbreviated to either OS or O/S) is an interface between hardware and user; it is responsible for the management and coordination of activities and the sharing of the resources of the computer. The operating system acts as a host for computing applications that are run on the machine. As a host, one of the purposes of an operating system is to handle the details of the operation of the hardware. This relieves application programs from having to manage these details and makes it easier to write applications. Almost all computers (including handheld computers, desktop computers, supercomputers, video game consoles) as well as some robots, domestic appliances (dishwashers, washing machines), and portable media players use an operating system of some type.

1.3 Log files

Log files are files that contain messages about the system, including the kernel, services, and applications running on it. There are different log files for different information. For example, there is a default system log file, a log file just for security messages, and a log file for cron tasks.

Log files can be very useful if you are trying to troubleshoot a problem with the system such as trying to load a kernel driver or if you are looking for unauthorized log in attempts to the system. This chapter discusses where to find log files, how to view log files, and what to look for in log files.

Some log files are controlled by a daemon called syslogd. A list of log messages maintained by syslogd can be found in the /etc/syslog.conf configuration file.

1.4 Proposed work

In every operation system the processes are control by the kernel, their history is stored in the log files which is formed by default in the operating system. This research was done to make the log files of the particular processes running in the different software packages at different terminals in a network. In this research we have some terminals over which the different software packages are running these terminals are connected through the server. Every terminal send the system calls to the server in the form of messages. These messages are collect in a queue called message queue. These messages are received from message queue and are segregated on the basis of their process_id. Then these messages are buffered in the individual buffers and when the buffer get full then the content of the buffer are write to the log files to maintain the record of the process running on the different software packages

1.5 Related work

Lot of work has been done for the log file analysis, i.e. to analyze the log files of the operating system to compare the test result checking in the process of the software testing. In the process of log file analysis the log files are analyzed line by line to find out the difference between the assumed result and actual result. Log files are analyzed to find out the debugging in the working of any software. Efficiency of any software is also determined by analyzing the log files through log file analyzer. But rarely work has been done in the field of the system log process through which we can make the log files of the process running on some particular software package. System log process is a very useful tool in the field of the software testing. Because by system log process we can make the log files of any particular software which can be used by log file analyzer to know the efficiency of any software to perform any particular task under the controlled way.

Chapter 2

Message communication in Linux

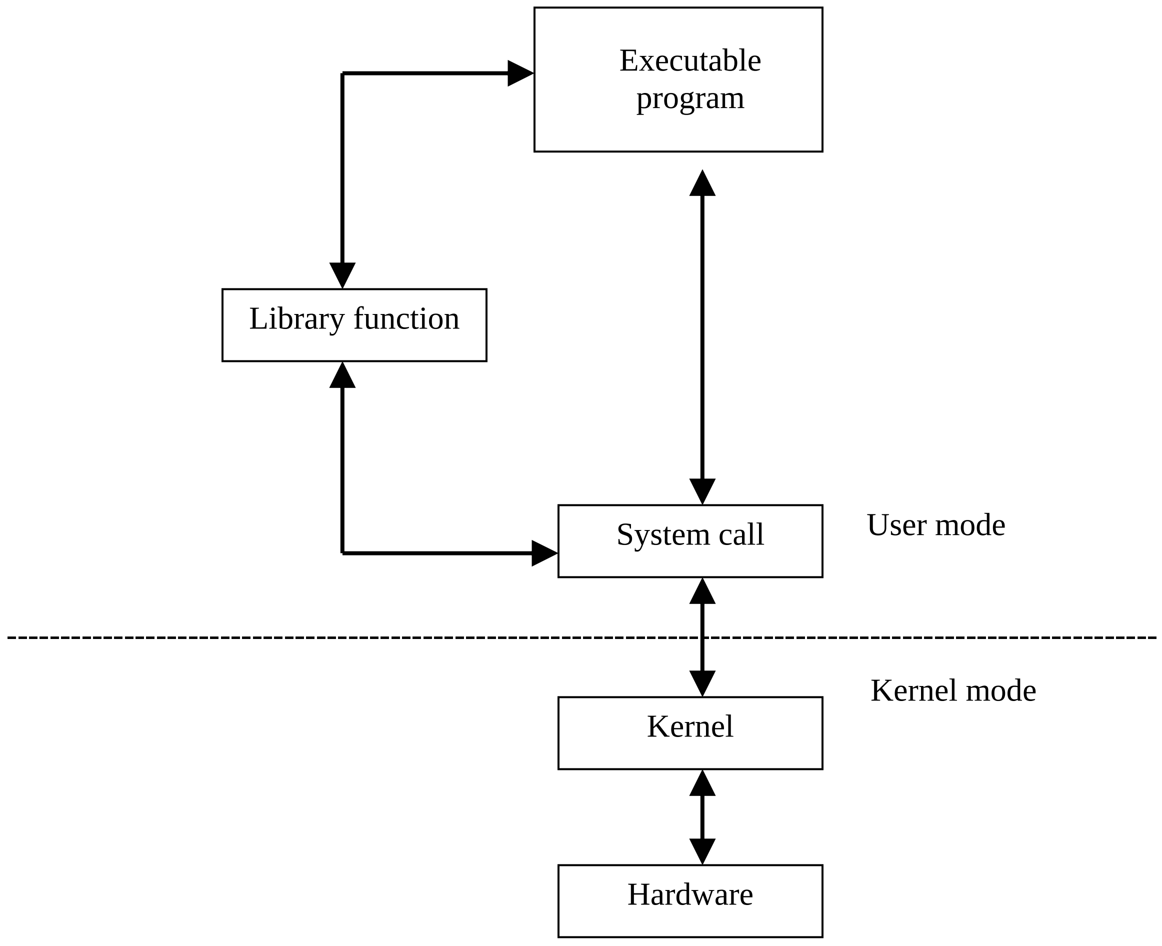
2.1 System Calls

Some previously defined functions used by programs are actually system calls. While resembling library functions in format, system calls request the operating system to directly perform some work on behalf of the invoking process. The code that is executed by the operating system lies within the kernel (the central controlling program that is normally maintained permanently in memory). The system call acts as a high/mid-level language interface to this code. To protect the integrity of the kernel, the process executing the system call must temporarily switch from user mode (with user privileges and access permissions) to system mode (with system/root privileges and access permissions). This switch in context carries with it a certain amount of overhead and may, in some cases, make a system call less efficient than a library function that performs the same task. Keep in mind many library functions (especially those dealing with input and output) are fully buffered and thus allow the system some control as to when specific tasks are actually executed.

Issuing an `apropos` command similar to the one previously discussed but using the value 2 in place of 3 will generate synopsis information on all the system calls . It is important to remember that some library functions have embedded system calls. For example, << and >>, the C++ insertion and extraction operators, make use of the underlying system calls `read` and `write`.

The relationship of library functions and system calls is shown in Figure 1.. The arrows in the diagram indicate possible paths of communication, and the dark circles indicate a context switch. As shown, executable programs may make use of system calls directly to request the kernel to perform a specific function. On the other hand, the executable programs may invoke a library function, which in turn may perform system calls.

2.2 Figure 1. Hardware and software layers of LINUX.



Chapter 3

Basic Format of Program

3.1 source program— A source program is a series of valid statements for a specific programming language (such as C or C++). The source program is stored in a plain ASCII text file. For purposes of our discussion we will consider a plain ASCII text file to be one that contains characters represented by the ASCII values in the range of 32–127. Such source files can be displayed to the screen or printed on a line printer. Under most conditions, the access permissions on the source file are set as nonexecutable. A sample C++ language source program is shown in Program 1

3.2 executable program— An executable program is a source program that, by way of a translating program such as a compiler, or an assembler, has been put into a special binary format that the operating system can execute (run). The executable program is not a plain ASCII text file and in most cases is not displayable on the terminal or printed by the user.

3.3. *Program 1 A source program in C .*

```
       /*
                Display Hello World 3 times
        */
       #include <stdio.h>
       #include <unistd.h>                      // needed for write
       #include <cstring>                       // needed for strcpy
       #include <cstdlib>                       // needed for exit
       using namespace std;
       char           *cptr = "Hello World\n";  // static by placement
      char            buffer1[25];
       int main( )
{
         void            showit(char *);        // function prototype
         int             i = 0;                 // automatic variable
         strcpy(buffer1, "A demonstration\n");  // library function
         write(1, buffer1, strlen(buffer1)+1);  // system call
         for ( ; i < 3; ++i)
           showit(cptr);                        // function call
         return 0;
       }
      void showit( char *p ){
         char           *buffer2;
         buffer2= new char[ strlen(p)+1 ];
         strcpy(buffer2, p);                    // copy the string
         printf(“buffer2”);                       // display string
         delete [] buffer2;                     // release location
}
```

3.4 Executable File Format

In a Linux environment, source files that have been compiled into an executable form to be run by the system are put into a special format called `ELF` (Executable and Linking Format). Files in `ELF` format contain a header entry (for specifying hardware/program characteristics), program text, data, relocation information, and symbol table and string table information. Files in `ELF` format are marked as executable by the operating system and may be run by entering their name on the command line. Older versions of UNIX stored executable files in `a.out` format (Assembler output Format). When C/C++ program files are compiled, the compiler, by default, places the executable file in a file called `a.out`.

3.5 System Memory

In UNIX, when an executable program is read into system memory by the kernel and executed, it becomes a process. We can consider system memory to be divided into two distinct regions or spaces. First is user space, which is where user processes run. The system manages individual user processes within this space and prevents them from interfering with one another. Processes in user space, termed user processes, are said to be in user mode. Second is a region called kernel space, which is where the kernel executes and provides its services. As noted previously, user processes can only access kernel space through system calls. When the user process runs a portion of the kernel code via a system call, the process is known temporarily as a kernel process and is said to be in kernel mode. While in kernel mode, the process will have special (root) privileges and access to key system data structures. This change in mode, from user to kernel, is called a context switch.

In UNIX environments, kernels are reentrant, and thus several processes can be in kernel mode at the same time. If the system has a single processor, then only one process will be making progress at any given time while the others are blocked.

Chapter 4
Process Management

4.1 Process

Fundamental to all operating systems is the concept of a process. A process is a dynamic entity scheduled and controlled by the operating system. While somewhat abstract, a process consists of an executing (running) program, its current values, state information, and the resources used by the operating system to manage the process. In a UNIX-based operating system, such as Linux, at any given point in time, multiple processes appear to be executing concurrently. From the viewpoint of each of the processes involved, it appears they have access to and control of all system resources as if they were in their own standalone setting. Both viewpoints are an illusion. The majority of operating systems run on platforms that have a single processing unit capable of supporting many active processes. However, at any point in time, only one process is actually being worked upon. By rapidly changing the process it is currently executing, the operating system gives the appearance of concurrent process execution. The ability of the operating system to multiplex its resources among multiple processes in various stages of execution is called multiprogramming (or multitasking). Systems with multiple processing units, which by definition can support true concurrent processing, are called multiprocessing.

As noted, part of a process consists of the execution of a program. A program is an inactive, static entity consisting of a set of instructions and associated data.If a program is invoked multiple times, it can generate multiple processes We can consider a program to be in one of two basic formats.

4.2. Process Memory

Each process runs in its own private address space. When residing in system memory, the user process, like Gaul, is divided into three segments or regions: text, data, and stack.

4.2.1 Text segment—

The text segment (sometimes called the instruction segment) contains the executable program code and constant data. The text segment is marked by the operating system as read-only and cannot be modified by the process. Multiple processes can share the same text segment. Processes share the text segment if a second copy of the program is to be executed concurrently. In this setting the system references the previously loaded text segment rather than reloading a duplicate. If needed, shared text, which is the default when using the C/C++ compiler, can be turned off by using the `-N` option on the compile line. In Program 1, the executable code for the functions `main` and `showit` would be found in the text segment.

4.2.2 Data segment—

The data segment, which is contiguous (in a virtual sense) with the text segment, can be subdivided into initialized data (e.g., in C/C++, variables that are declared as `static` or are static by virtue of their placement) and uninitialized data. In Program 1., the pointer variable `cptr` would be found in the initialized area and the variable `buffer1` in the uninitialized area. During its execution lifetime, a process may request additional data segment space. In Program 1. the call to the library routine `new` in the `showit` function is a request for additional data segment space. Library memory allocation routines (e.g., `new`, `malloc`, `calloc`, etc.) in turn make use of the system calls `brk` and `sbrk` to extend the size of the data segment. The newly allocated space is added to the end of the current uninitialized data area. This area of available memory is sometimes called the heap.

4.2.3 Some authors use the term BSS segment for the unitialized data segment.

4.2.4 **Stack segment—** The stack segment is used by the process for the storage of automatic identifiers, register variables, and function call information. The identifier `i` in the function `main`, `buffer2` in the function `showit`, and stack frame information stored when the `showit` function is called within the `for` loop would be found in the stack segment. As needed, the stack segment grows toward the uninitialized data segment. The area beyond the stack contains the command-line arguments and environment variables for the process. The actual physical location of the stack is system-dependent.

4.3. The u Area

In addition to the text, data, and stack segments, the operating system also maintains for each process a region called the `u` area (user area). The `u` area contains information specific to

the process (e.g., open files, current directory, signal actions, accounting information) and a system stack segment for process use. If the process makes a system call (e.g., the system call to `write` in the function `main` in Program 1.), the stack frame information for the system call is stored in the system stack segment. Again, this information is kept by the operating system in an area that the process does not normally have access to. Thus, if this information is needed, the process must use special system calls to access it. Like the process itself, the contents of the `u` area for the process are paged in and out by the operating system.

4.4. Creating a Process

It is apparent that there must be some mechanism by which the system can create a new process. With the exception of some special initial processes generated by the kernel during bootstrapping (e.g., `init`), all processes in a Linux environment are created by a `fork` system call, shown in Table 1. The initiating process is termed the parent, and the newly generated process, the child.

Table 1. Summary of the `fork` System Call.

Include File(s)	`<sys/types.h>`
	`<unistd.h>`
			Manual Section	2

Summary	`Pid_t fork ( void );`			
Return	Success	Failure	Sets `errno`	
	0 in child, child process ID in the parent	-1	Yes	

[*] The include file `<sys/types.h>` usually contains the definition of `pid_t`. However, in some environments the actual definition will reside in `<bits/types.h>`. Fortunately, in these environments the `<sys/types.h>` contains an include statement for the alternate definition location, and all remains transparent to the casual user. The include file `<unistd.h>` contains the declaration for the `fork` system call.

The `fork` system call does not take an argument. If the `fork` system call fails, it returns a -1 and sets the value in `errno` to indicate one of the error conditions shown in Table 2.

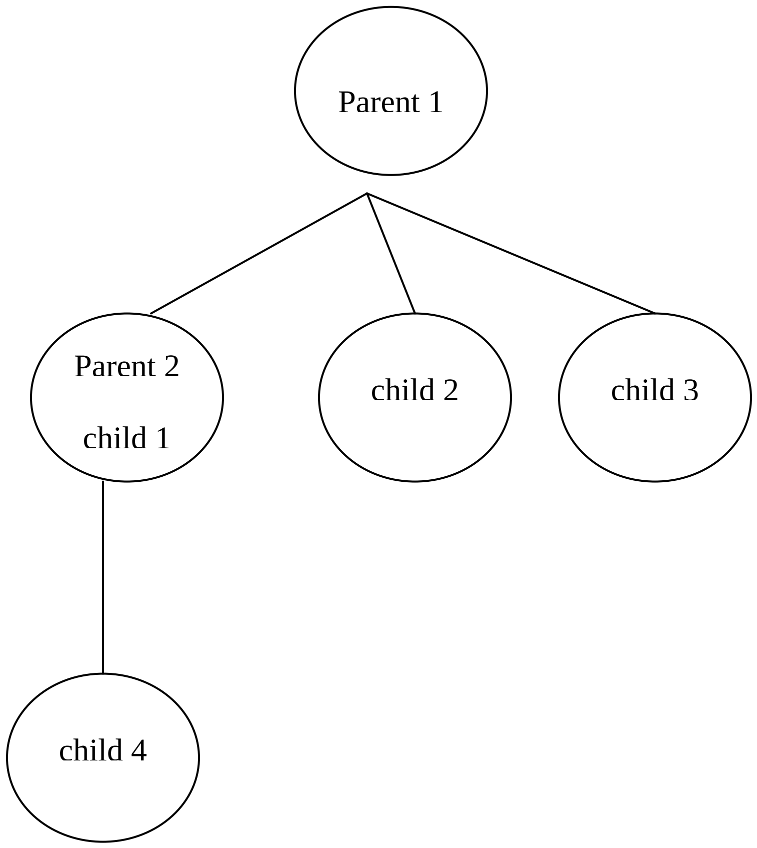
Table 2. `fork` Error Messages.

#	Constant	`perror` Message	Explanation
11	EAGAIN	Resource temporarily unavailable	The operating system was unable to allocate sufficient memory to copy the parent's page table information and allocate a task structure for the child.
12	ENOMEM	Cannot allocate memory	Insufficient swap space available to generate another process.

If the library function/system call sets `errno` and can fail in multiple ways, an error message table will follow the summary table. This table will contain the error number (#), the equivalent defined constant, the message generated by a call to `perror`, and a brief explanation of the message in the current context.

Otherwise, when successful, `fork` returns the process ID (a unique integer value) of the child process to the parent process, and it returns a 0 to the child process. By checking the return value from `fork`, a process can easily determine if it is a parent or child process. A parent process may generate multiple child processes, but each child process has only one parent. Figure 2 shows a typical parent/child process relationship.

(a). Figure 2. The parent/child process relationship.



As shown, process P1 gives rise to three child processes: C1, C2, and C3. Child process C1 in turn generates another child process (C4). As soon as a child process generates a child process of its own, it becomes a parent process.

(b)Generating a child process.

```
/*   First example of a fork system call (no error check)   */
       #include <iostream>
       #include <sys/types.h>
       #include <unistd.h>
       using namespace std;
       int       main( )
{
        cout << "Hello\n";
         fork( );
         cout << "bye\n";
         return 0;
       }
```

The output of the program is as follows:.

```
linux$ p1.5
Hello
bye
bye
```

Notice that the statement `cout << "bye\n";` only occurs once in the program at line 12, but the run of the program produces the word "`bye`" twice—once by the parent process and once by the child process. Once the `fork` system call at line 11 is executed there are two processes each of which executes the remaining program statements.

4.5. Process ID

Associated with each process is a unique positive integer identification number called a process ID (PID). As process IDs are allocated sequentially, when a system is booted, a few system processes, which are initiated only once, will always be assigned the same process ID. For example, on a Linux system process 0 (historically known as `swapper`) is created from scratch during the startup process. This process initializes kernel data structures and creates another process called `init`. The `init` process, PID 1, creates a number of special kernel threads to handle system management. These special threads typically have low PID numbers.

Other processes are assigned free PIDs of increasing value until the maximum system value for a PID is reached. The maximum value for PIDs can be found as the defined constant `PID_MAX` in the header file `<linux/threads.h>` (on older systems check `<linux/tasks.h>`). When the highest PID has been assigned, the system wraps around and begins to reuse lower PID numbers not currently in use.

The system call `getpid` can be used to obtain the PID . The `getpid` system call does not accept an argument. If it is successful, it will return the PID number. If the calling process does not have the proper access permissions, the `getpid` call will fail, returning a value of – 1 and setting `errno` to `EPERM` (1).

Table 3. Summary of the `getpid` System Call.

Include File(s)	`<sys/types.h>` `<unistd.h>`		Manual Section	2
Summary	`pid_t getpid( void );`			
Return	Success	Failure	Sets `errno`	
	The process ID	–1	Yes	

Chapter 5

Interprocess Communication

5.1. Interprocess communication

The designers of UNIX found the types of interprocess communications that could be implemented using signals and pipes to be restrictive. To increase the flexibility and range of interprocess communication, supplementary communication facilities were added. These facilities, added with the release of System V in the 1970s, are grouped under the heading IPC (Interprocess Communication). In brief, these facilities are

5.1.1 **Message queues**— Information to be communicated is placed in a predefined message structure. The process generating the message specifies its type and places the message in a system-maintained message queue. Processes accessing the message queue can use the message type to selectively read messages of specific types in a first in first out (FIFO) manner. Message queues provide the user with a means of asynchronously multiplexing data from multiple processes.

5.1.2 **Semaphores—** Semaphores are system-implemented data structures used to communicate small amounts of data between processes. Most often, semaphores are used for process synchronization.

5.1.3 **Shared memory—** Information is communicated by accessing shared process data space. This is the fastest method of interprocess communication. Shared memory allows participating processes to randomly access a shared memory segment. Semaphores are often used to synchronize the access to the shared memory segments.

All three of these facilities can be used by related and unrelated processes, but these processes must be on the same system (machine).

Like a file, an IPC resource must be generated before it can be used. Each IPC resource has a creator, owner, and access permissions. These attributes, established when the IPC is created, can be modified using the proper system calls. At a system level, information about the IPC facilities supported by the system can be obtained with the `ipcs` command. For example, on our system the `ipcs` command produces the following output shown in following Figure 3.

. *Figure 3 Some `ipcs` output.*

```
linux$ ipcs

------ Shared Memory Segments ------

Key         shmid      owner      perms    bytes      nattch   Status <-- 1
0x00000000  25198594   root       666      247264     3


------ Semaphore Arrays ------

key         semid      owner      perms   nsems      Status <-- 2
0x00000000  65537      root       666     4
0x00000000  98306      root       666     16
0x00000000  131075     root       666     16
0x00000000  163844     root       666     16
```

(1) One shared memory segment attached (shared) by three processes.

(2) Four sets of semaphores all owned by root.

(3) No message queues are currently allocated.

The `ipcs` utility supports a variety of options for specifying a specific resource and the format of its output. The meaning of each is shown in Table 3

Additionally, `-s`, `-q`, or `-m` can be used to indicate semaphore, message queue, or shared memory, and can be followed by `–i` and a valid decimal ID to display additional information about a specific IPC resource .

Table 4. `ipcs` Command Line Options.

Resource Specification		**Output Format**	
`–a`	All (default)	`–c`	Creator
`–m`	Shared memory	`–l`	Limits
`–q`	Message queues	`–p`	Process ID
`–s`	Semaphores	`–t`	Time
		`–u`	Summary

5.2. Creating a Message Queue

A message queue is created using the `msgget` system call (Table 5).

Table 5. Summary of the `msgget` System Call.

Include File(s)	`<sys/types.h>` `<sys/ipc.h>` `<sys/msg.h>`		Manual Section	2
Summary	`int msgget (key_t key,int msgflg);`			
Return	Success	Failure	Sets `errno`	
	Nonnegative message queue identifier associated with `key`	–1	Yes	

If the `msgget` system call is successful, a nonnegative integer is returned. This value is the message queue identifier and can be used in subsequent calls to reference the message queue. If the `msgget` system call fails, the value –1 is returned and the global variable `errno` is set appropriately to indicate the error (see Table 6). The value for the argument `key` can be specified directly by the user or generated using the `ftok` library function (as covered in the previous discussion). The value assigned to `key` is used by the operating system to produce a unique message queue identifier. The low-order bits of the `msgflg` argument are used to determine the access permissions for the message queue. Additional flags (e.g., IPC_CREAT, IPC_EXCL) may be `OR`ed with the permission value to indicate special creation conditions.

A new message queue is created if the defined constant IPC_PRIVATE is used as the `key` argument or if the IPC_CREAT flag is `OR`ed with the access permissions and no previously existing message queue is associated with the `key` value. If IPC_CREAT is specified (without IPC_EXCL) and the message queue already exists, `msgget` will not fail but will return the message queue identifier that is associated with the `key` value (Table 6.).

Table .6. `msgget` Error Messages.

#	Constant	Perror Message	Explanation
2	EOENT	No such file or directory	Message queue identifier does not exist for this `key` and IPC_CREAT was not set.
12	ENOMEM	Cannot allocate memory	Insufficient system memory to allocate the message queue.
13	EACCES	Permission denied	Message queue identifier exits for this `key`, but requested operation is not allowed by current access permissions.
17	EEXIST	File exists	Message queue identifier exists for this `key`, but the flags IPC_CREAT and IPC_EXCL are both set.
28	ENOSPC	No space left on device	System imposed limit (MSGMNI) for the number of message queues has been reached.
43	EIDRM	Identifier removed	Specified message queue is marked for removal.

Program 2 generates five message queues with read/write access, uses the `ipcs` command (via a pipe) to display message queue status, and then removes the message queues.

Program .3 Generating message queues.

```
 /*  Message queue generation */
 #define _GNU_SOURCE
 #include <cstdio>
 #include <unistd.h>
 #include <linux/limits.h>
 #include <sys/types.h>
 #include <sys/ipc.h>
 #include <sys/msg.h>
using namespace std;
 const int MAX=5;
 int                                         <-- 1
 main( ){
   FILE *fin;
   char  buffer[PIPE_BUF], proj = 'A';
   int   i, n, mid[MAX];
   key_t key;
   for (i = 0; i < MAX; ++i, ++proj) {
     key = ftok(".", proj);
    if ((mid[i] = msgget(key, IPC_CREAT | 0660)) == -1) {
       perror("Queue create");
       return 1;
     }
   }
   fin = popen("ipcs", "r");                 <-- 2
   while ((n = read(fileno(fin), buffer, PIPE_BUF)) > 0)
    write(fileno(stdout), buffer, n);
   pclose(fin);
   for (i = 0; i < MAX; ++i )                <-- 3
    msgctl(mid[i], IPC_RMID, (struct msqid_ds *) 0);
   return 0;
 }
```

(1) Create five message queues.

(2) Use a named pipe to execute the `ipcs` command.

(3) Remove the five message queues.

When run on our system, this program produces the output as follows indicating that five message queues have been generated.

Output of Program 3.

```
linux$ p3

------ Shared Memory Segments ------
key         shmid     owner     perms     bytes     nattch   status
0x00000000  25198594  root      666       247264    3

------ Semaphore Arrays ------
key         semid     owner     perms     nsems     status
0x00000000  65537     root      666       4
0x00000000  98306     root      666       16
0x00000000  131075    root      666       16
0x00000000  163844    root      666       16

------ Message Queues ------
key         msqid     owner     perms     used-bytes  messages
0x41153384  2260992   gray      660       0           0
0x42153384  2293761   gray      660       0           0
0x43153384  2326530   gray      660       0           0
0x44153384  2359299   gray      660       0           0
0x45153384  2392068   gray      660       0           0
```

5.3. Message Queue Operations

Message queues are used to send and receive messages. An actual message, from the system's standpoint, is defined by the `msgbuf` structure found in the header file `<sys/msg.h>` as

```
struct msgbuf  {
    long int mtype;          /* type of received/sent message */
    char mtext[1];           /* text of the message */
  };
```

This structure is used as a template for the messages to be sent to and received from the message queue.

The first member of the `msgbuf` structure is the message type. The message type, `mtype`, is a long integer value and is normally greater than 0. The message type, generated by the process that originates the message, is used to indicate the kind (category) of the message. The type value is used by the `msgrcv` system call to selectively retrieve messages falling within certain boundary conditions. Messages are placed in the message queue in the order they are sent and not grouped by their message type.

Following `mtype` is the reference to the body of the message. As shown, this is defined as a character array with one element: `mtext[1]`. In actuality, any valid structure member(s), character arrays or otherwise, that make up a message can be placed after the requisite `mtype` entry. The system assumes a valid message always consists of a long integer followed by a series of 0 or more bytes (the organization of the data bytes is the programmer's prerogative). It is the address of the first structure member after `mtype` that the system uses as its reference when manipulating the `msg` structure . Therefore, users can generate their own message structures to be placed in the message queue so long as the first member (on most systems this is the first four bytes) is occupied by a long integer.

Messages are placed in the message queue (sent) using the system call `msgsnd` (Table 7).

Table 7. Summary of the `msgsnd` System Call.

Include File(s)	`<sys/types.h>` `<sys/ipc.h>` `<sys/msg.h>`		Manual Section	2
Summary	`int msgsnd (int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);`			
Return	Success	Failure	Sets `errno`	
	0	-1	Yes	

The `msgsnd` system call requires four arguments. The first argument, `msqid`, is a valid message queue identifier returned from a prior `msgget` system call. The second argument, `msgp`, is a pointer to the message to be sent. As noted, the message is a structure with the first member being of the type long integer. The message structure must be allocated (and hopefully initialized) prior to its being sent. The third argument, `msgsz`, is the size (number of bytes) of the message to be sent. The size of the message is the amount of storage allocated for the message structure minus the storage used for the message type (stored as a long integer). The message size can be from 0 to the system-imposed limit. The fourth argument to `msgsnd`, `msgflg`, is used to indicate what action should be taken if system limits for the message queue (e.g., the limit for the number of bytes in a message queue) have been reached. The `msgflg` can be set to IPC_NOWAIT or to 0. If set to IPC_NOWAIT and a system limit has been reached, `msgsnd` will not send the message and will return to the calling process immediately with `errno` set to EAGAIN. If `msgflg` is set to 0, `msgsnd` will block until the limit is no longer at system maximum (at which time the message is sent), the message queue is removed, or the calling process catches a signal. The system uses the `msgsz` argument to `msgsnd` as its `msg.msg_ts` value, the `msgbuf.mtype` value as its `msg.msg_type`, and the `msgbuf.mtext` reference as `msg.msg_spot`.

If `msgsnd` is successful, it returns a value of 0; otherwise, it returns a value of -1 and sets `errno` to indicate the nature of the error. See Table 8.

Table 8. `msgsnd` Error Messages.

#	Constant	`perror` Message	Explanation
4	EINTR	Interrupted system call	When sleeping on a full message queue, the process received an interrupt.
11	EAGAIN	Resource temporarily unavailable	Message cannot be sent (`msg_qbyte` limit exceeded) and IPC_NOWAIT was specified.
12	ENOMEM	Cannot allocate memory	Insufficient system memory to copy message.
13	EACCES	Permission denied	Calling process lacks write access for the message queue.
14	EFAULT	Bad address	`msgp` references a bad address.
22	EINVAL	Invalid argument	• Message queue identifier is invalid. • `mtype` is nonpositive. • `msgsz` is less than 0 or greater than system limit.
43	EIDRM	Identifier removed	Message queue has been removed.

Messages are retrieved from the message queue using the system call `msgrcv`, summarized in Table 9

Table 9. Summary of the `msgrcv` System Call.				

Include File(s)	`<sys/types.h>`
	`<sys/ipc.h>`
	`<sys/msg.h>`
			Manual Section	2

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

Return	Success	Failure	Sets `errno`	
	Number of bytes actually received	-1	Yes	

The `msgrcv` system call takes five arguments. The first, as for the `msgsnd` system call, is the message queue identifier. The second, `msgp`, is a pointer to the location (structure) where the received message will be placed. The receiving location should have as its first field a long integer to accommodate the message type information. The third argument, `msgsz`, is the maximum size of the message in bytes. This value should be equal to the longest message to be received. Truncation of the message will occur if the size value is incorrectly specified, and depending upon the value for `msgflg` (see following section), an error may be generated. The fourth argument, `msgtyp`, is the type of the message to be retrieved. The message type information is interpreted by the `msgrcv` system call, as shown in Table 10.

Table 10. Actions for `msgrcv` as Indicated by `msgtyp` Values.

When `msgtyp` value is	**`msgrcv` takes this action**
0	Retrieve the first message of any `msgtyp`.
> 0	Retrieve the first message equal to `msgtyp` if MSG_EXCEPT is not specified. If MSG_EXCEPT is specified, the first message that is not equal to the `msgtyp`.
< 0	Retrieve the first message of the lowest type less than or equal to absolutevalue of `msgtyp`.

Using the type argument judiciously, a user can, with minimal effort, implement a priority-based messaging arrangement whereby the message type indicates its priority.

The fifth and final argument, `msgflg`, is used to indicate what actions should be taken if a given message type is not in the message queue, or if the message to be retrieved is larger in size than the number of bytes indicated by `msgsz`. There are three predefined values that `msgflg` can take. IPC_NOWAIT is used to indicate to `msgrcv` that it should not block if the requested message type is not in the message queue. If MSG_EXCEPT is specified and the `msgtyp` value is greater than 0, `msgrcv` returns the first message not equal to `msgtyp`. MSG_NOERROR directs `msgrcv` to silently truncate messages to `msgsz` bytes if they are found to be too long. If MSG_NOERROR is not specified and `msgrcv` receives a message that is too long, it returns a -1 and sets the value in `errno` to E2BIG to indicate the error. In don't-care situations, the value for `msgflg` can be set to 0. When `msgrcv` is successful, it returns the number of bytes actually retrieved. See Table 11

Table 11. `msgrcv` Error Messages.

Constant	`Perror` Message	Explanation
EINTR	Interrupted system call	When sleeping on a full message queue, the process received an interrupt.
E2BIG	Argument list too long	`mtext` is greater than `msgsz` and MSG_NOERROR is not specified.
EACCES	Permission denied	Attempt made to read a message, but the calling process does not have permission.
EFAULT	Bad address	`msgp` references a bad address.
EINVAL	Invalid argument	• Message queue identifier is invalid. • `msgsz` is less than 0 or greater than the system limit.
ENOMSG	No message of desired type	Message queue does not have a message of type `msgtyp`, and IPC_NOWAIT is set.
EIDRM	Identifier removed	Message queue has been removed.

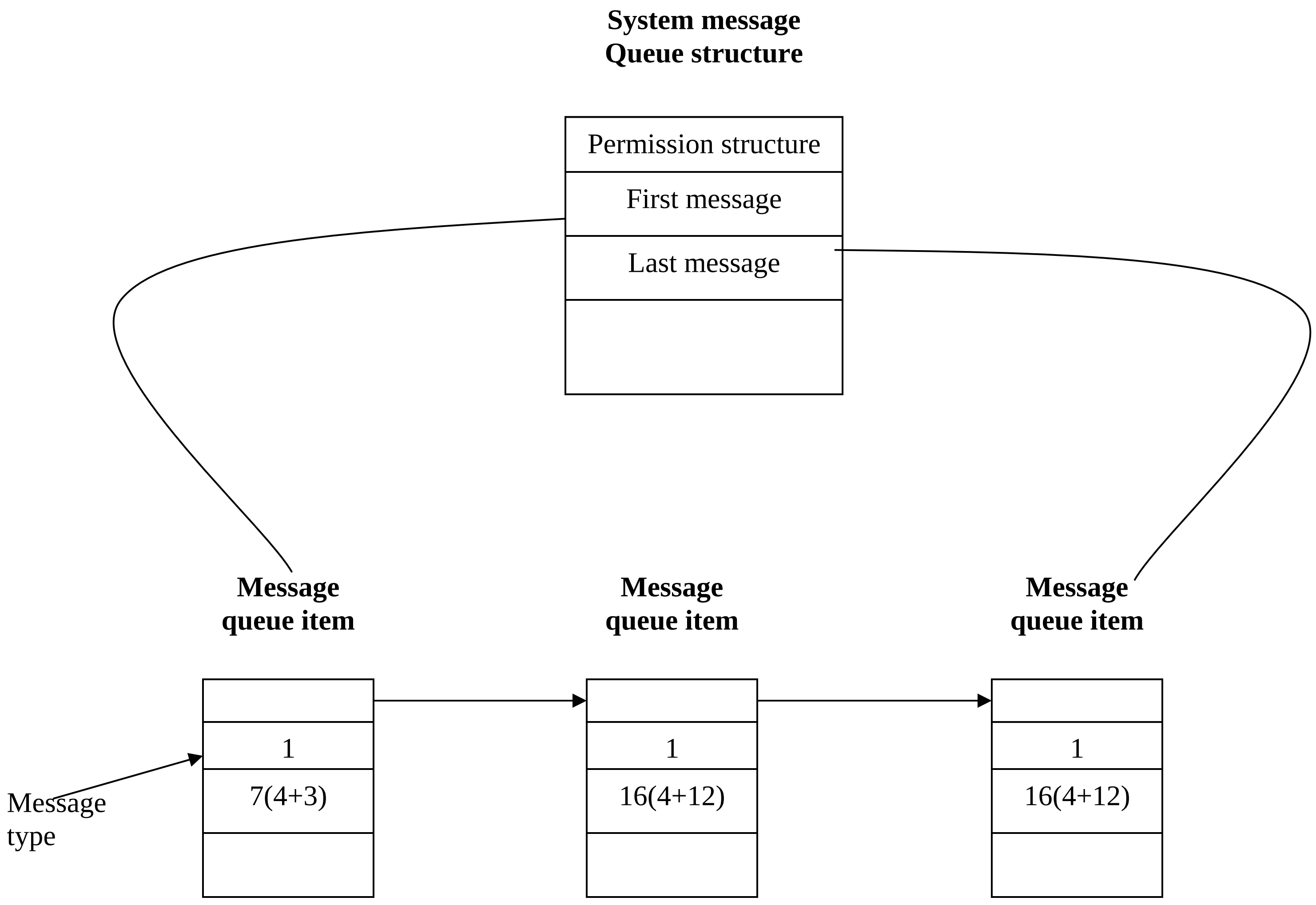
5.4. A Client–Server Message Queue

At this point we can use what we have learned about message queues to write a pair of programs that establish a client–server relationship and use message queues for bidirectional interprocess communication. The client process obtains input from the keyboard and sends it via a message queue to the server. The server reads the message from the queue, manipulates the message by converting all alphabetic text in the message to uppercase, and places the message back in the queue for the client to read. By mutual agreement, the client process identifies messages designated for the server by placing the value 1 in the message type member of the message structure. In addition, the client includes its process ID (PID) number in the message. The server uses the PID number of the client to identify messages it has processed and placed back in the queue. Labeling the processed messages in this manner allows the server to handle messages from multiple clients.

This works nicely, as in multiple client situations, because not every client has initial access to the PID of the server.

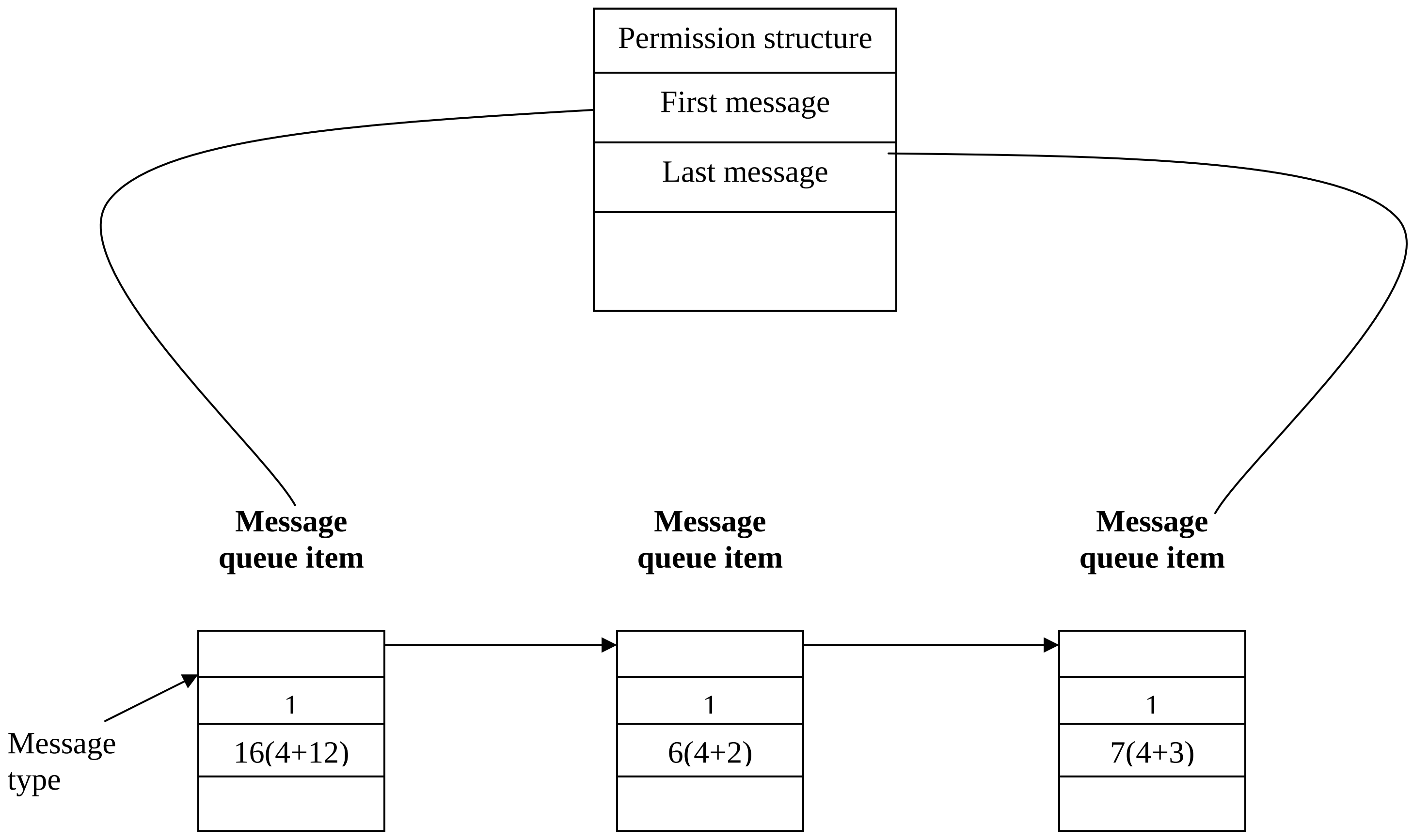
For example, if the client process with a PID of 17 placed each word in the statement `"The anticipation is greater than the realization."` into separate messages, the current state of the message queue would be as depicted in Figure A. As shown, the messages placed in the queue by the client (PID 17) are labeled as a message type of 1 (for the server).

Figure A. Conceptual view of message queue after the client has sent all seven messages



When the server reads the queue, it obtains the first message of type 1. In our example this is the message containing the word `The`. The server processes the message, changes the message type to that of the client, and puts the message back on the queue. This leaves the message queue in the state shown in Figure 4.

Figure B. Conceptual view of message queue after the first client message has been processed.



To accomplish this task, both the client and server programs need to access common include files and data structures. These items are placed in a local header file called `local.h`, An examination of this file reveals that the messages placed in the queue consist of a structure with three members. The first member (which must be of type `long` if things are to work correctly) acts as the message type (`mtype`) member. Here we call this member `msg_to`, since it contains a value that indicates the process to whom we are addressing the message. We use the value of 1 to designate a message for the server process, and other positive PID values to indicate a message for a client. The second member of the message structure, called `msg_fm` (which is also a long integer), contains the ID of the process that is sending the message. In the program example, if the message is sent by a client, this value will be the client PID. If the message is sent by the server, this value will be set to 1. The third member of the message structure is an array of a fixed size that will contain the text of the actual message.

Chapter 6

System log process

6.1 Proposed Algorithm For System Log Process

This algorithm consist of the two parts namely message queue operation and process log whose steps are as follows:

ALGORITHM :

Message queue operation

Step 1:- Define the three methods **getQ()**, **sendQ(log_msg)**, and **rcvQ()** to create the message queue and sending and receiving the message to/from the message queue.

Step 2:- Send the messages to the message queue through **sendQ(log_msg)** method.

Step 3:- Receive the message from the message queue through **rcvQ()** method.

Process log

Step 4:- Define the method **msg segregate(msg, type, process_id)** to segregate the message on the basis of their process_id.

Step 5:- Buffering the messages on the basis of their process-id.

Step 6:- If the buffer is full then write the buffer to the log files.

6.2 SYSTEM LOG PROCESS

System log process is a process through which we make the log files on the server, for all the processes running on the client nodes. Every process running on the node has a unique identification number called process_id. It means that the every process coming to the server from a particular client will have the same process_id. And the process coming to the server from any other client or node will have some other process_id. Thus first of all we receive the message from the different nodes which contains process_id, message_id, and message size and message text. These messages are received in the form of the system calls sending to the server and are collected in the message queue through the getQ() function.

Message queue is initialize through getQ() function. It uses the msgget() system call to create the message queue whose syntax is :

int msgget (key_t key, int msgflg)

getQ() uses this system call and return a unique message queue identifier, msg_id, on successful compilation. This identifier is used for interacting with the queue for sending and receiving messages. The msgget() return -1, if the system wide limit on the number of message queue has exceeded.

sendQ() function is used to send the message to the message queue. This method is used by the application software to send the message to the message queue, which is read by the system log process. The syntax of the msgsnd() system call is as follows-

int msgsnd(int msqid, struct msgbuf*msg, int size, int flag);

The msqid is the message queue identifier returned by msgget() in the method getQ(). Using this identifier ensures that the messages are sent to the queue created by our mechanism.msg has the two parts mtype, and mtext. The mtext defines contain the message and mtype defines the type of the message. size is the size of message.

The method rcvQ() is used to receive the message from the message queue and to get the process_id of the sending process. rcvQ() uses msgrcv() system call to get the message from the message queue. The syntax of Msgrcv() is as follows-

int msgrcv(int msqid, struct msgbuf*msg, int size, long type, int flag):

The msqid, size and msg are as defined above for msgsnd() system call . The type specifies which message on the queue is desired. Flag is specified as MSG_NOERROR which means that any message bigger than size bytes will be shortened to size bytes and no error will be returned.

rcvQ() uses the system call msgctl() to get the process_id of the sending process for the received message. The syntax for msgctl() is as follows-

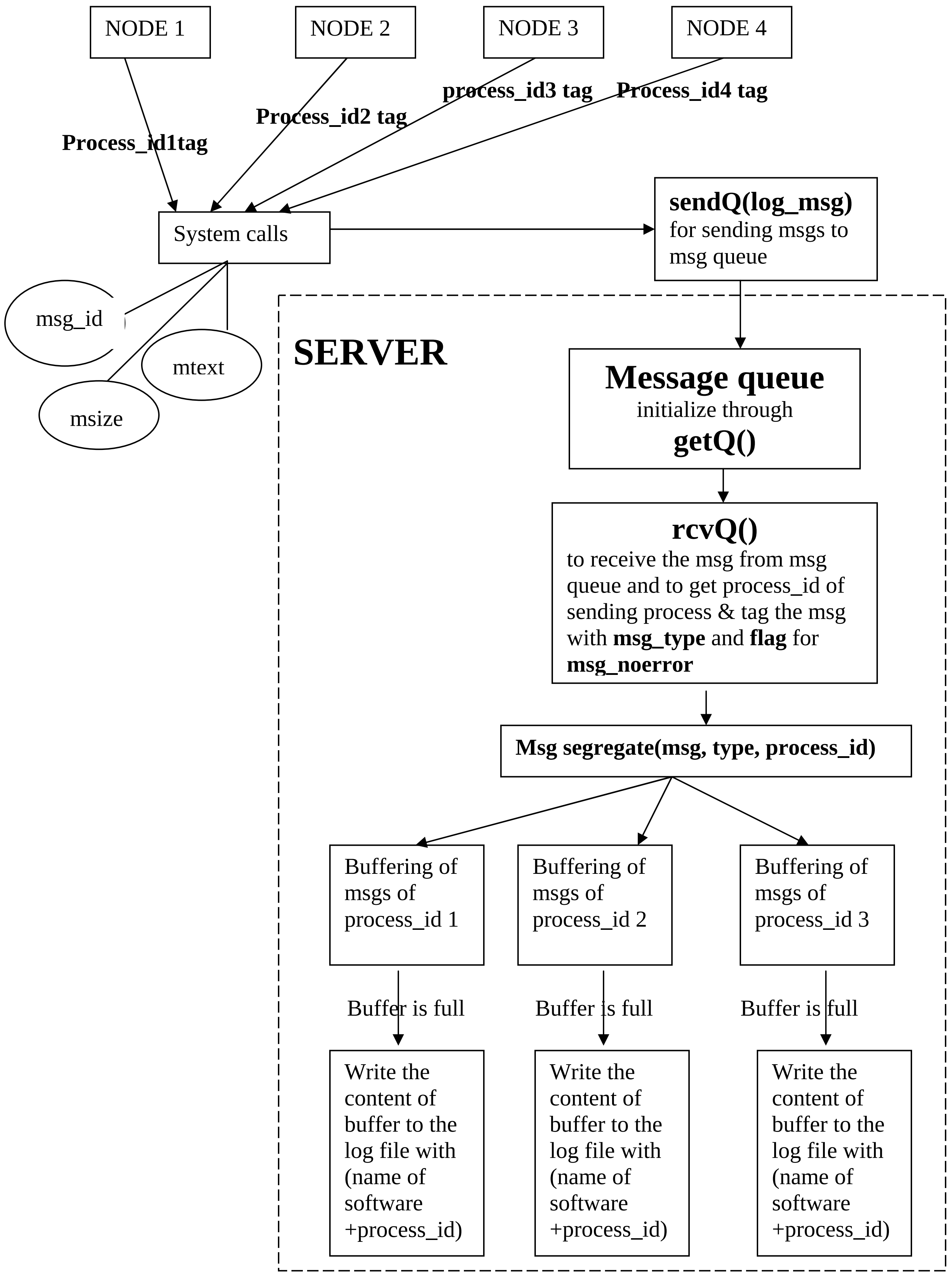
int msgctl(int msqid, int cmd, struct msqid_ds *buf)

here struct msqid_ds is maintained by the kernel of the operating system for every message queue.

Each program in execution has a unique process_id. Msgsegregate() function segregate the message received from different application software on the basis of their Process_id. It maintain a table indexed on process_id i.e. to buffer all the process of the same node or same process_id. On receiving the message from message queue it is stored in its buffer if buffer is not full. If the buffer is full then the content of the buffer is sent to log files , and buffer being empty to store the other messages in it. Then a new log file with the name (name of software + process_id) will create, if it does not exist and the content will write to it. The name of software is same for each run of the application software, but the process_id is different. Thus log message for each run of the application software will store in a new log file.

Through this method the processes of any software can be stored in the form of a log file. Which can be used in the future to check the processes of a particular software to perform the particular task.

6.3 FLOW DIAGRAM OF SYSTEM LOG PROCESS


NODE 1
NODE 2
NODE 3
NODE 4
Process_id1tag
Process_id2 tag
process_id3 tag
Process_id4 tag
System calls
msg_id
mtext
msize
sendQ(log_msg)
for sending msgs to msg queue
SERVER
Message queue
initialize through
getQ()
rcvQ()
to receive the msg from msg queue and to get process_id of sending process & tag the msg with msg_type and flag for msg_noerror
Msg segregate(msg, type, process_id)
Buffering of msgs of process_id 1
Buffering of msgs of process_id 2
Buffering of msgs of process_id 3
Buffer is full
Buffer is full
Buffer is full
Write the content of buffer to the log file with (name of software +process_id)
Write the content of buffer to the log file with (name of software +process_id)
Write the content of buffer to the log file with (name of software +process_id)

6.4 Source Code For System Log Process

```
*/      Local header files used in the System Log Process     */

      #define _GNU_SOURCE
      #include <stdio.h>
      #include <unistd.h>
      #include <linux/limits.h>
      #include <sys/types.h>
      #include <sys/ipc.h>
      #include <sys/msg.h>

      const int MAX=5;

      int  main( )
        {
         FILE *fin;
         char  buffer[PIPE_BUF], proj = 'A';
         int   i, n, mid[MAX];
         key_t key;
         for (i = 0; i < MAX; ++i, ++proj)
           {
               key = ftok(".", proj);
               if ((mid[i] = msgget(key, IPC_CREAT | 0660)) == -1)
             {
               perror("Queue create");
               return 1;
             }
           }
        fin = popen("ipcs", "r");
        while ((n = read(fileno(fin), buffer, PIPE_BUF)) > 0)
        write(fileno(stdout), buffer, n);
        pclose(fin);
        for (i = 0; i < MAX; ++i )
        msgctl(mid[i], IPC_RMID, (struct msqid_ds *) 0);
        return 0;
       }
```

This is the program to send the messages of client to the server so that the server can respond accordingly. These messages are the system calls of all the processes running on the client terminal.

```
/*         program for client to send messages to the server        */


        #include "local.h"
       #include <cstdio.h>
       using namespace std;

        int main( )
          {
           key_t       key;
           pid_t       cli_pid;
           int         mid, n;
           MESSAGE     msg;
           static char m_key[10];
           cli_pid = getpid( );
           if ((key = ftok(".", SEED)) == -1)
                {
                  perror("Client: key generation");
                  return 1;
                }
           if ((mid=msgget(key, 0 )) == -1 )
                {
                  mid = msgget(key,IPC_CREAT | 0660);
                  switch (fork()) {
                  case -1:
                  perror("Client: fork");
                  return 2;
                  case 0:
                  sprintf(m_key, "%d", mid);
                  execlp("./server", "server", m_key, "&", 0);
                  perror("Client: exec");
                  return 3;
                } }          }
           while (1)
                {
```

```
       msg.msg_to = SERVER;
       msg.msg_fm = cli_pid;
       write(fileno(stdout), "cmd> ", 6);
       memset(msg.buffer, 0x0, BUFSIZ);
 if ( (n=read(fileno(stdin), msg.buffer, BUFSIZ)) == 0 )
       break;
       n += sizeof(msg.msg_fm);
 if (msgsnd(mid, &msg, n, 0) == -1 )
     {
       perror("Client: msgsend");
       return 4;
     }
 If( (n=msgrcv(mid, &msg, BUFSIZ, cli_pid, 0)) != -1 )
      write(fileno(stdout), msg.buffer, n);
    }
      msgsnd(mid, &msg, 0, 0);
      return 0;
}
```

This is the program to receive the messages from client to the server so that the server can store the system calls of the client software application. These messages are the system calls of all the processes running on the client terminal.

```
/*      program for server to receive the messages from clients     */


      #include "local.h"
      #include <stdio.h>
      #include <ctype.h>
      #include <stdlib.h>

      int  main(int argc, char *argv[ ])
       {
         int     mid, n;
         MESSAGE msg;
         void    process_msg(char *, int);
         if (argc != 3)
           {
        //   cerr << "Usage: " << argv[0] << " msq_id &" << endl;
             return 1;
           }
         mid = atoi(argv[1]);
         while (1)
           {
             memset( msg.buffer, 0x0, BUFSIZ );
             if ((n=msgrcv(mid, &msg, BUFSIZ, SERVER, 0)) == -1 )
               {
                 perror("Server: msgrcv");
                 return 2;
               }
             else if (n == 0) break;
             process_msg(msg.buffer, strlen(msg.buffer));
             msg.msg_to = msg.msg_fm;
             msg.msg_fm = SERVER;
             n += sizeof(msg.msg_fm);
             if (msgsnd(mid, &msg, n, 0) == -1 )
               {
```

```
         perror("Server: msgsnd");
         return 3;
        }
    }
  msgctl(mid, IPC_RMID, (struct msqid_ds *) 0 );
  exit(0);
}
```

This is the program to make a message queue on the server so that all the system calls coming to the server can stored in the message queue. Through this program we can send and receive the messages to the message queue.

/*   a program to make a message queue. Receive and send the message to the message queue   */

```
#include <stdio.h>
#include <errno.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int main()
 {
   printf("Hi there, here is msgget!\n") ;
   /* Parameters */
   key_t key = 0 ;
   int msgflg = 03600 ;
   /* Return value */
   int msqid ;
   /* Create the message queue */
   msqid = msgget( key, msgflg ) ;
   printf( "msqid: %i\n", msqid) ;
   if ( msqid == -1 )
     {
       printf("msgget: initializing message queue failed!\n") ;
       perror("errno:") ;
       return -1 ;
```

```
          }
        else
          {
            printf("msgget: msgget succeeded (msqid: %i)\n", msqid);
           }

        printf("Hi there, here is msgsnd!\n") ;
        /* Parameters */
        size_t msgsz = 5 ;
        struct message
            {
              long mtype ;
              const char *mpointer ;
             } ;
        struct message msg ;
        msg.mtype = 0 ;
        msg.mpointer = "hello" ;
        if ( msgsnd( msqid, msg.mpointer, msgsz, msgflg ) == 0 )
          {
            printf("msgsnd: message was successfully placed into
             messagequeue %i.\n", msqid) ;
          }
        else
          { printf("msgsnd: sending message to message queue %i
            failed.\n", msqid) ;
            return -1 ;
    }
       printf("Hi there, here is msgrcv!\n") ;
       /* Parameters */
       int msgtyp = 0 ;
       struct message_rcv
         {
           long mtype ;
           char messtxt[msgsz] ;
         } *msg_rcv ;
       msg_rcv = malloc( msgsz*sizeof(char) + sizeof(long) ) ;
       /* Receive the message */
       printf("msgflg: %5o\n", msgflg) ;
```

```
     printf("msgtyp: %d\n", msgtyp) ;
     long msgactsz ;
     // Actual number of bytes placed in the structure
     if ( msgactsz = msgrcv( msqid, msg_rcv, msgsz, msgtyp,
          IPC_NOWAIT ) != -1 )
       {
         printf("msgrcv: message successfully read from message
         queue%i.\n", msqid) ;
       }
    else
       {
         printf("msgrcv: receiving message from message queue %i
         failed.\n", msqid) ;
         fprintf( stderr, "errno: %i\n", errno ) ;
    return -1 ;
  }
    printf("We finally made it past the invocation ( msgactsz
    =%d,mtext: %s )!\n", msgactsz, msg_rcv->messtxt ) ;
    return 0 ;
}
```

Chapter 7

Conclusion and Future Work

7.1 CONCLUSION

System log file is a process through which we can make the log record of all the processes running on any particular node in any software to check its efficiency and checking its error. System log process is a part of the software testing, which can be used to test the error and efficiency of the software under testing. By using the log file analyzer we can recognize that which process is responsible for the error in the software so that we can remove that very easily. The name of log files made in system log process will contain the process_id i.e. we can easily calculate the number of processes executed by the software.

7.2 FUTURE WORK

Every event of the software process can be stored in the log files and by using the log file analyzer we can analyze the log files to find out any error or any kind of failure of the software to perform any task. We can use the system log process to test any kind of software very effectively. We can compare the two software on the basis of their performance to do any particular task. We can use the system log process to find out the efficient software to do any task. Because the efficiency of the software is depending on the time and the number of processes to complete any work. Through system log process we can compare the most efficient software running on the different systems at the same time by comparing their number of processes. Log analyzer uses the system log files to find out the number of processes executed by any software to perform any given task.

REFERNCES

1. A. James H, Z. Yingjun: *General Test Result Checking with Log File Analysis*, IEEE Trans. On Software Engineering, vol 29, No. 7, pp. 634-648 (2003)
2. A. Goel, S.C. Gupta, S.K. Wasan: Probe Mechanism for Object-Oriented Software Testing. In Mauro Pezze, editor, In Proceedings of Fundamental Approaches to Software Engineering (FASE 2003), Lecture Notes in Computer Science, LNCS 2621, pp. 310-324, Warsaw, Springer, Poland, (2003)
3. *A.Silberschatz, P.B. Galvin*: Operating System Concept, Fifth Edition ,John Wiley & Sons (2000).
4. J.H. Andrews, "*Testing Using Log File Analysis: Tools, Methods and Issues*", Proc. Int'l Conf. Automated Software Eng. (ASE '98), pp. 157-166, Oct. 1998.
5. J.S. Gray,"Interprocess Communications in Linux", Prentice Hall PTR, (2003).
6. R.J. Moore: A universal dynamic trace for Linux and other operating systems. In Proceedings of FREENIX Track (2001)
7. W.R. Stevens, "Advanced Programming in the UNIX Environment", Addison - Wesley Longman, Singapore Pte. Ltd., (2001).
8. W.R. Stevens: UNIX Network Programming Volume 2: Interprocess Communications. 2nd Edition, Pearson Education (1995)