Synchronization in Minix

CS4089 Project

End Semester Report

Palivela Rohit (B120931CS)
Chandra-sekhar Guntupalli (B120618CS)
Guided By: Dr. Muralikrishnan K

November 17, 2015

Abstract

This report presents a method of solving Synchronization problems in MINIX.

1 Introduction

Process synchronization refers to the idea that multiple processes are to join up or handshake at a certain point, in order to reach an agreement or commit to a certain sequence of actions. If proper synchronization techniques are not applied, it may cause a race condition where, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes or threads.This project is inclined to solving the synchronization problems in Minix operating system. This is achieved by the addition of a new Minix service, Semaphore. Given the semaphore service, the users can invoke it to solve synchronization issues through system calls.

1.1 The Internal Structure of MINIX 3

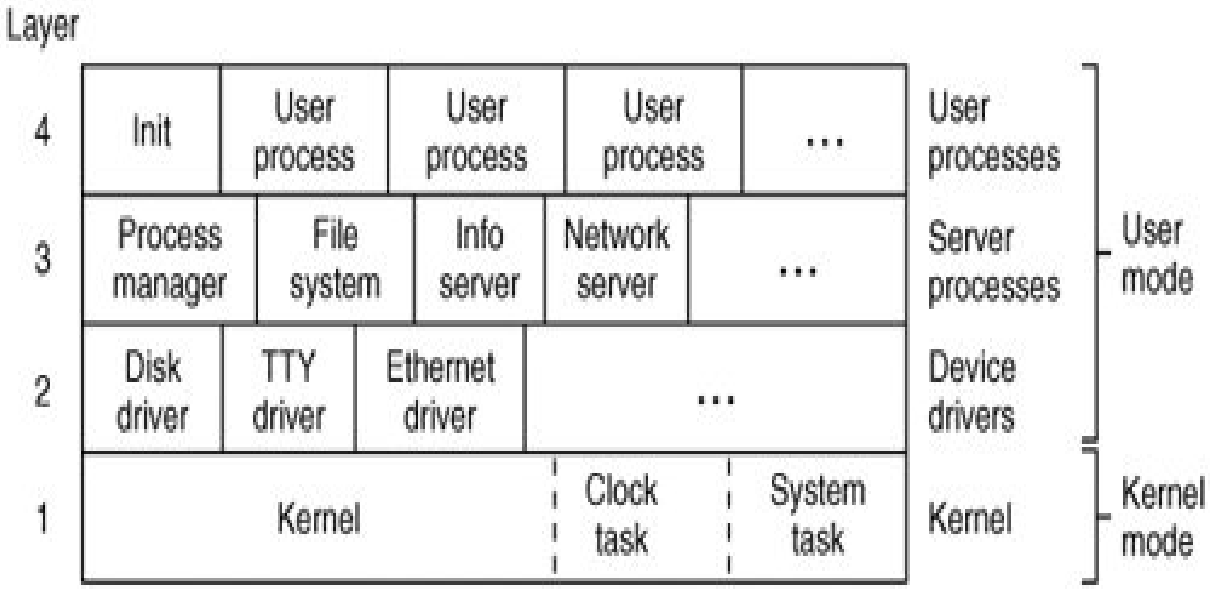
MINIX 3 is structured in four layers with each layer performing a well defined function. The four different layers are illustrated in Fig.1-1

The bottom layer contains kernel which schedules processes and manages the transitions between the ready, running and blocked states. The kernel also handles all the messages between processes. One of the main functions of the layer1 is to provide a set of privileged kernel calls to the drivers and servers above it like reading and writing I/O ports, copying data between addresses.

The three layers above the bottom layer could be considered to be a single layer because the kernel treats them the same way. Each one is limited to user mode instructions, and each is scheduled to be run by the kernel. However processes have special privileges(such as the ability to make kernel calls). The processes in layer 2 have the most privileges, those in layer 3 have fewer whereas processes in layer4 have none. This is the real difference between these layers.

The second layer consists of device drivers which are allowed to request that the system task read data from or write data to I/O ports. The third layer contains servers, processes that provide useful services to the user processes. Most important processes in the layer3 are the PROCESS MANAGER(PM) which carries the system calls related to process execution and the FILE SYSTEM(FS) which carries out file system calls such as read, mount, and chdir. Finally, the layer4 contains all the user process shells, editors, compilers and user-written a.out programs

The aim of this project is to achieve synchronization among the processes in


Layer
4
3
2
1
Init
User process
User process
User process
...
Process manager
File system
Info server
Network server
...
Disk driver
TTY driver
Ethernet driver
...
Kernel
Clock task
System task
User processes
Server processes
Device drivers
Kernel
User mode
Kernel mode

the layer4 which are the user programs.

1.2 Interprocess Communication in MINIX 3

The different processes present in MINIX communicate through primitive message passing. Three primitives are provided for sending and receiving messages. They are called by the C library procedures.

- **send(dest, &message);** to send a message to process dest
- **receive(source, &message);** to receive a message from process source(or ANY)
- **sendrec(src_dst, &message);** to send a message and wait for a reply from the same process. This is peer level.

Each task, driver or server process is allowed to exchange messages only with certain other processes. The usual flow of messages is downward in the layers and messages can be between processes in the same layer and processes between adjacent layers. User processes cannot exchange messages between each other. When a process sends a message to another process that is not currently waiting for a message, the sender blocks until destination does a receive thereby eliminating the need for buffer management.

There exists another important message passing primitive. It is called by the C library procedure

- **notify(dest);** is used when a process has to make another process aware that something important has happened. It is nonblocking, which means the sender continues to send whether or not the recipient is waiting. These are meant to be used by system processes.

The above are the most important message passing primitives. So, for processes to achieve synchronization, the processes have to pass messages between themselves and the kernel to avoid race conditions.

2 Problem Statement

The problem is to solve the synchronization problems and avoid race conditions (two or more processes trying to simultaneously read or write some shared data). This involves implementing one of the interprocess communication primitives.

3 Literature Survey

The Internal structure, Interprocess Communication and Implementation of processes in Minix 3 are presented by Andrew S. Tanenbaum and Albert S.Woodhull [1].

4 Work Done

A semaphore is a variable that is used for controlling access, by multiple processes, to a common resource in a concurrent system such as a multiprogramming operating system. A trivial semaphore is a plain variable that is changed (for example, incremented, decremented, toggled) and is then used as a condition to control access to some system resource. Semaphores which allow an arbitrary resource count are called counting semaphores, while semaphores which are restricted to the values 0 or 1 are called binary semaphores. Semaphores are a useful tool in preventing race conditions.

Semaphores are accessed only through two standard atomic operations:

P(S):
while S<=0
block;
S:=S-1;
V(S):
S:=S+1;

Using semaphore service, we can prevent the system from going into race conditions and the processes can work in a synchronized fashion.

4.1 Synchronization in MINIX

To provide synchronization among user processes in MINIX 3, we implement a new Semaphore service for MINIX. This service needs to understand four different messages : sem_init, sem_release, sem_up, sem_down. The semantics of these four messages should be:

- **int sem_init (int start_value):** This function creates a SEM_INIT message that is sent to the semaphore service. This parameter start_value specifies the initial value for the semaphore. It can be an arbitrary integer value. For a binary semaphore, this value would be +1. Once a semaphore has been established, it is considered active. The semaphore service supports unlimited semaphores, that is until memory is exhausted. On success, the function returns the next available semaphore number starting with 0. All the errors return the appropriate error codes.

- **int sem_down (int sem_number):** This function creates a SEM_DOWN message that is sent to the semaphore service. This call can only be invoked for an active semaphore. Calling it on an uninitialized semaphore returns an error. This function implements the standard semantics of semaphore.p( ). That is, the call decrements the counter of the corresponding semaphore by one. When the counter(after decrement) has a value of <0, then the calling process is put to sleep(waiting in the queue that corresponds to that semaphore). On success, the semaphore returns 0.

- **int sem_up (int sem_number):** This function creates a SEM_UP message that is sent to the semaphore service. This call can only be invoked for an active semaphore. Calling it on an uninitialized semaphore returns an error. This function implements the standard semantics of semaphore.v( ). That is, the call increments the counter of the corresponding semaphore by one. When there is at least one process waiting(sleeping) in the queue, the first process sleeping in the queue is woken up. On success, this returns 0.

- **int sem_realease (int sem):** This function creates a SEM_RELEASE message that is sent to the semaphore service. This call can only be invoked for an active semaphore. Calling it on an uninitialized semaphore returns an error. The purpose of this function is to release an active semaphore and put it back into inactive state. When there are processes waiting for this semaphore, this function should fail returning an error.

4.2 Implementing a new system call in MINIX 3

In MINIX3, the servers handle system calls. Adding a new system call for our Semaphore service involves three steps:

- Writing a system call handler
- Writing a user library
- Building a server to receive these new system calls and handle them

Creating a system call handler

The source codes for all servers are located at /usr/src/servers. Each server has a separate directory. Let us name the server used for our semaphore service as sema. So the source code for our server will be available at /usr/src/servers/sema. Each of the

server source directories consists of two files: table.c and proto.h. table.c contains definitions for the call_vec table. The call_vec table is an array of function pointers that is indexed by the system call number. In each line, the address of a system call handler function is assigned to one entry in the table and the index of the entry is the system call number.

Let us see the table.c file for FileSystem(FS) Server

```
PUBLIC _PROTOTYPE (int (*call_vec[]), (void) ) = {
        no_sys,         /*  0 = unused    */
        do_exit,        /*  1 = exit      */
        do_fork,        /*  2 = fork      */
        do_read,        /*  3 = read      */
        do_write,       /*  4 = write     */
        do_open,        /*  5 = open      */
        do_close,       /*  6 = close     */
        no_sys,         /*  7 = wait      */
        do_creat,       /*  8 = creat     */
```

The above figure contains a few entries from /usr/src/servers/fs/table.c. The second line in the figure assigns the address of function do_exit to the second entry in the table. The index of the second entry, which is number 2, is the system call number for calling the handler do_exit. The entries in our sema server will be similar to the above figure

```
        do_unpause,     /* 65 = UNPAUSE*/
        no_sys,         /* 66 = unused  */
        do_revive,      /* 67 = REVIVE    */
        no_sys,         /* 68 = TASK_REPLY      */
        no_sys,         /* 69 = unused */
        no_sys,         /* 70 = unused */
        no_sys,         /* 71 = si */
        no_sys,         /* 72 = sigsuspend */
        no_sys,         /* 73 = sigpending */
        no_sys,         /* 74 = sigprocmask */
```

There are a few unused entries. no_sys represents an unused entry. For adding a new system call we need to identify one unused entry. For example index 69 contains an unused entry. We need four system calls for our four requirements SEM_INIT, SEM_UP, SEM_DOWN, SEM_RELEASE. So we need to find four unused entries. For example we could use slot number 69 for our SEM_INIT system call handler. To use entry 69 we replace no_sys with SEM_INIT(int start_val). We do the same for other system call handlers.

The next step is to declare a prototype of the system call handler in file /usr/src/servers/sema/proto.h. This file contains the prototypes of all system call handler functions. The below figure contains the prototype declarations from /usr/src/servers/fs/proto.h. We should add the prototypes for our system call handlers to the proto.h.

_PROTOTYPE(int SEM_INIT,(int));

```
/* open.c */
_PROTOTYPE( int do_close, (void)                );
_PROTOTYPE( int do_creat, (void)                );
_PROTOTYPE( int do_lseek, (void)                );
_PROTOTYPE( int do_mknod, (void)                );
_PROTOTYPE( int do_mkdir, (void)                );
_PROTOTYPE( int do_open, (void)                 );
```

A few files like misc.c, stadir.c, write.c and read.c contain the definitions for the system call handler functions. We could either add our system call handlers to these files or have it in a separate file. It is more logical in our approach to add these to a new file. When we add them to a new file we need to make changes to the /usr/src/services/sema/MakeFile accordingly. After implementing the system call handlers, we can compile the sema server to ensure that our new system call handlers do not contain any errors.

What different system call handlers should do?

- **SEM_INIT:** This routine is called upon receiving SEM_INIT message from the user processes. Whenever this routine is called it increments the number of semaphores count by one and returns value of the first free entry found in the field of semaphore id in semaphore table(which is the semaphore assigned) to the user process.
  Semaphore table contains four fields (i)Semaphore id, (ii)Process id, (iii)Number of processes, (iv) Pointer to the head of the queue.

- **Semaphore id:** This field shows

the semaphore id of corresponding semaphores. It is unique for each semaphore.

- **Process id:** This field points to the list of processes currently holding the Semaphore. If the Semaphore is free i.e, no process is currently holding the Semaphore, it points to NULL.
- **Count:** This field shows the count of available semaphores which can be acquired by processes.
- **Pointer to head:** This field points to the head of the queue of the processes waiting to get the corresponding semaphore.

If this routine does not find an empty entry in the semaphore table, it creates a new row in the table. It takes the count of semaphores as parameter and sets it accordingly.

- **SEM_DOWN:** This routine is called upon receiving SEM_DOWN message from the user processes. It takes the semaphore id as an argument. It checks for the semaphore id field of the corresponding row of the semaphore. If no row is found, we return an error message to the process saying that semaphore is not initialized. If a row is found, we check the count field of the row to see if it is free or not(count>0). If the semaphore is free(count>0), we add the process id of the user process which sent the message in the process id field. If it is not free, we put the process to sleep and add the process id to the queue of processes waiting and increment the number of processes field of the row.
- **SEM_UP:** This routine is called upon receiving SEM_UP message from the user processes. It takes the semaphore id as an argument. It checks for the semaphore id field of the corresponding row of the semaphore. If no row is found, we return an error message to the process saying that semaphore is not initialized. If a row is found, we check if the process id of the user process is present in the list of processes field or not. If it is not present, we return an error message saying that the process has never acquired the semaphore. If it is present, we remove this process id from the list and increment the count field by one. If there is at least one process waiting in the queue to acquire the Semaphore, we wake that process, add its process id to the list of processes and decrement the count field by one.
- **SEM_RELEASE:** This routine is called upon receiving SEM_RELEASE message from the user processes. It takes the Semaphore id as an argument. It checks for the Semaphore id field of the corresponding row of the semaphore. If no row is found, we return an error message to the process saying that semaphore is not initialized. If a row is found, we check if the processes list is empty or not. If the process list is not empty, we return an error saying that there are process which currently acquired the Semaphore. If not, we check for the queue of processes waiting for the Semaphore. If there are any processes in the queue we return an error.

4.3 Creating a User Library Function:

A user library function would package the parameters for the system call handler in the message structure and would call the handler function. First, we should use #define to map the system call number of the handler function to an identifier in the

file /usr/src/include/minix/callnr.h and usr/include/minix/callnr.h. We implement the library function for the SEMINIT system call in a separate file called _seminit.c. This file should be placed in the directory /usr/src/lib/posix/.

Steps to compile the new library

- Go to /usr/src/lib/posix/
- Add the name of the file in the usr/src/lib/posix/Makefile.in.
- Issue the command make Makefile.
- Go to the directory /usr/src/
- Issue the command make libraries.
- All these steps will complete and install the updated posix library.

4.4 Building a Server for Semaphore Service:

The semaphore service will require two things, making a service that is started on boot up, and putting the processes to sleep and waking them up at the appropriate time. We can build our sema server by copying a server like system scheduler server. Most servers have the same basic structure: a dispatch loop that checks for messages and executes system calls and some routines to initialize and exit the server. The source code for the scheduler service can be found at /usr/src/servers/sched. We can copy the code using the below command:

cp r /usr/src/servers/sched /usr/src/servers/sema

- We need to modify the Makefile in /usr/src/servers/sema folder to update the name of the binary
- We need to modify the Makefile in /usr/src/servers as well
- We need to remove all the irrelevant code from our service

Now we need to add our sema server to boot image. Since MINIX uses a microkernel architecture there are several places where the boot image has to be updated. The order that services are listed in tables are important as it determines the order they startup.

- /usr/src/tools/Makefile contains the Makefile that compiles to the boot image. We add path to our server in the programs variable.
- /usr/src/include/minix/com.h lists constants for all of the boot image processes. We can see entries like #define DS_PROC_NR for each process. We create our own constant for our process and update the constants for others to keep the order consistent.
- /usr/src/kernel/table.c has a table of the boot image processes that the kernel uses which needs to be updated.
- /usr/src/servers/rs/table.c has three tables which need to be filled.
- /usr/src/servers/rs/managers.c has a list of if statements around line 1645 which need to be updated for our process.

We recompile the kernel using make fresh install from the /usr/src/tools directory.

5 Future Work and Conclusions

We will be implementing the above design in the coming semester.

References

[1] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems-Design and Implementation (The MINIX book)*, Pearson Education, 2006.