# Process Synchronization in Minix

## CS4089 Project

## Midterm Evaluation

G Chandra Sekhar, P Rohit
Guided By: Dr. Murali Krishnan

November 17, 2015

# Outline

# Introduction

- Critical Sections:
  - the part of the program where shared memory is accessed
- Interprocess communication primitives:
  - semaphores, monitors, messages-these primitives are used to ensure that no two processes are ever in their critical sections at the same time.
- Process Synchronization:
  - refers to the idea that multiple processes communicate with each other, in order to reach an agreement.

# Problem Statement

- to solve synchronization problems and avoid race conditions.
  - involves implementing Semaphore service for MINIX - one of the interprocess communication primitives

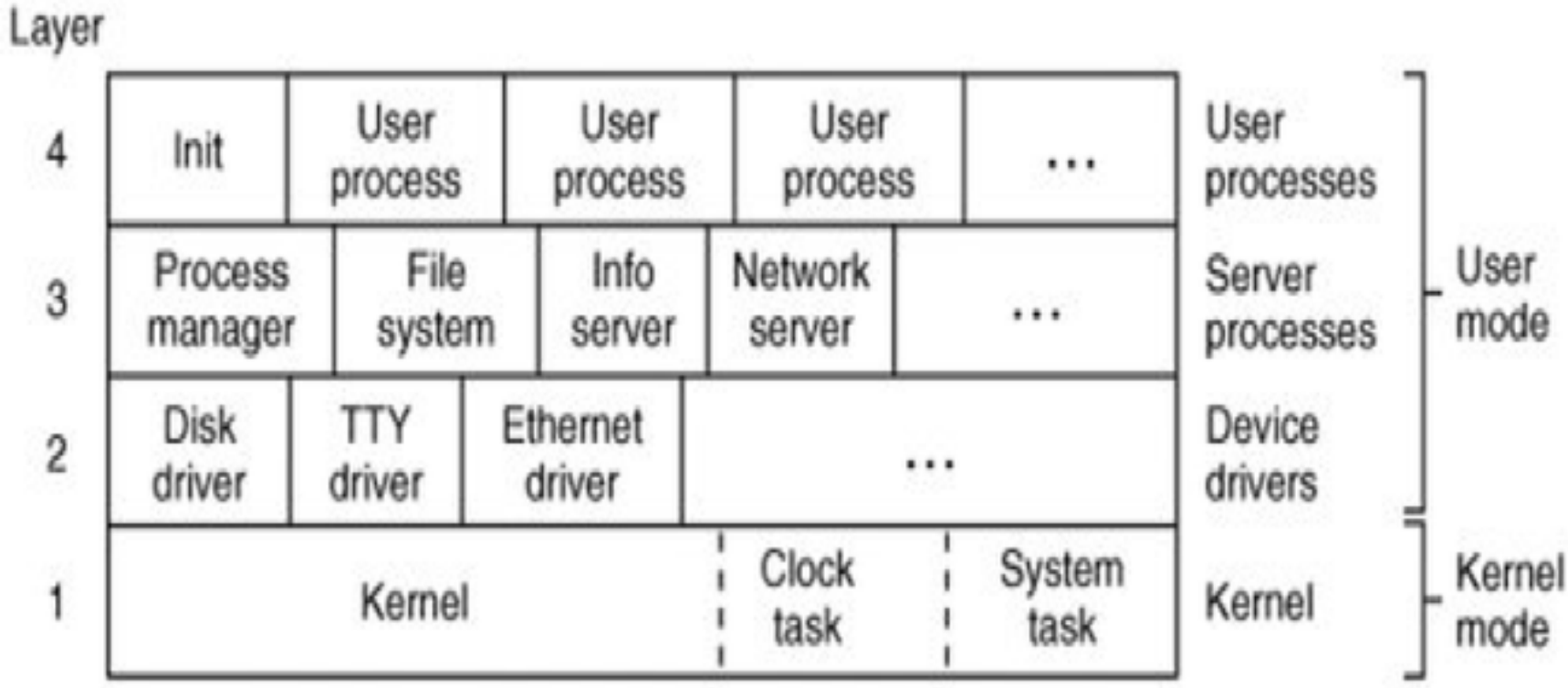
# Literature Survey

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

# Work Done

- Completed the planned literature survey
  - studied the 4-layered structure of MINIX 3, namely, Kernel, Device drivers, Server processes and User processes
  - also studied about important message passing primitives - send(dest,&message),receive(source,&message),sendrec(src_dst ,&message),notify(dest).
- The semaphore service needs to understand four different messages: sem_init, sem_release, sem_up, sem_down

# 4-layered structure of MINIX



# Important message passing primitives in MINIX 3

- 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
- 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.

# Messages to be understood by semaphore service

- int sem_init(int start_value)
  - creates a SEM_INIT message that is sent to semaphore service
  - on success, returns next available semaphore number and all errors return the appropriate error codes
- int sem_down(int semaphore_number)
  - this function creates a SEM_DOWN message that is sent to the semaphore service.
  - this call decrements the counter of the corresponding semaphore by one
  - calling it on uninitialized semaphore returns error

- int sem_up(int semaphore_number)
  - this function creates a SEM_UP message that is sent to the semaphore service.
  - this call increments the counter of the corresponding semaphore by one
  - calling it on uninitialized semaphore returns error
- int sem_release(int semaphore)
  - this function creates a SEM_RELEASE message that is sent to the semaphore service.
  - releases an active semaphore and put it back into inactive state
  - calling it on uninitialized semaphore returns error

# Implementing a new system call in MINIX 3

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

# Creating a system call handler

- 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.
- no_sys represents an unused entry. For adding a new system calls we need to identify unused entries.
- We should add the prototypes for our system call handlers to the proto.h.

# Semaphore Table

- 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.

# Creating a user library

- A user library function would package the parameters for the system call handler in the message structure and would call the handler function.
- 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.
- For example, 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/.
- compile the new library

# Building a Server for semaphore service

- 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 command cp -r /usr/src/servers/sched /usr/src/servers/sema
- we need to modify the the Makefile in /usr/src/servers/sema and /usr/src/servers
- add our sema server to boot image
- modify the required Makefiles and recompile the kernel using make fresh install from the /usr/src/tools directory.

# Future Work

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

# References I

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