PROJECT 4 - Semaphore

Purpose :

The purpose of this project is to add a new synchronization service to MINIX. This service provides synchronization is implemented as a system service.

This is achieved in three steps as mentioned:

1. Implemented a system call in MINIX

2. Added the semaphore as system service to MINIX

3. Using the semaphores to solve a simple synchronization problem. Two programs named prof.c and student.c are part of this.

The source codes for all servers are located at /usr/src/servers. Each server has a separate directory. I have named the server used for semaphore service as sema. I followed the steps below:

usr/src/servers/sema/table.c / - this contains definitions for the call vec tablemapping the system call numbers onto the routines

/usr/src/servers/sema/proto.h - declare a prototype of the system call handler in file .Added the prototypes for semaphore system call handlers to the proto.h.

/usr/src/servers/sema/ semaphore.c - System call handlers are added in this file. Initialized 10 semaphores as per the requirement. This file also contains queues for storing the waiting processes. Adding waiting processes to the queue and deleting the processes from the queue once they get the resource.

usr/src/servers/sema/semaphore.h – includes the header files for semaphore.

Different system call handlers:

SEM INIT (do_sem_init)- This function creates a SEMA_INIT message that is sent to the semaphore service. The parameter semaphore_number is an integer that specifies the index of the semaphore that should be initialized. Your service is supposed to support 10 different semaphores, thus, this argument must be a number between 0 and 9. The parameter start_value specifies the initial value for the semaphore. It can be an arbitrary integer value. Typically, for a mutex, this value would be +1. Once a semaphore has been initialized, it is considered active. Calling sema_init on an active semaphore is an error. You can only initialize inactive semaphores. Make sure to return an error code to the calling function in case an active semaphore is initialized, or in case on an invalid argument. Of course, initially, all semaphores are inactive.

SEM DOWN (do_sem_down) - This function creates a SEMA_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 leads to an error. This function implements the standard semantics of semaphore.down(). That is, the call decrements the counter of the corresponding semaphore by one. When the counter (after the decrement) has a value of < 0, then the calling process should be put to sleep (waiting in the queue that corresponds to the semaphore). The value of the parameter semaphore_number must be between 0 and 9.

SEM UP (do_sem_up) - This function creates a SEMA_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 leads to an error. This function implements the standard semantics of semaphore.up(). That is, the call increments the counter of the corresponding semaphore by one. When there is at least one processes waiting (sleeping) in the queue, the first process that was put to sleep should be woken up (i.e., sleep and wake up is FIFO). The value of the parameter semaphore_number must be between 0 and 9.

SEM RELEASE (do_sem_release) - This function creates a SEMA_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 leads to an error. The purpose of this function is to release an active semaphore and put it back into the inactive state. The value of the parameter semaphore_number must be between 0 and 9. This function fails when there are currently processes waiting in the queue of the semaphore that should be released.

To compile the servers: Go to directory /usr/src/servers/ and Issue make image and then Issue make install.

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.

Copied ds to sema in the usr/src/servers

Modified the Makefile in /usr/src/servers/sema folder to update the name

Modified the Makefile in /usr/src/servers as well

Removed all the irrelevant code from the semaphore service

Now we need to add our sema server to boot image. The order that services are listed in tables are important as it determines the order they startup.

/usr/src/releasetools/Makefile - contains the Makefile that compiles to the boot image Specify the semaphore to be a part of the system image. Programs are in the order they should be loaded by boot

/usr/src/include/minix/com.h - lists constants for all of the boot image processes.

/usr/src/include/minix/callnr.h - map the system call number of the handler function to an identifier

/usr/include/semaphoresyscall.h – include the header for the semaphore.

/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 – including the semaphore in the boot image table. The order is very important in this file. Definition of the boot image in the sys table.

/usr/src/servers/rs/managers.c has a list of if statements which need to be updated for the semaphore

Now in /usr/src/releasetools run make clean install and then issue the hdboot command.

To compile a User Library Function : Go to the directory /usr/src/lib/posix/, Add the name of the file in the /usr/src/lib/posix/Makefile.in, Issue the command make Makefile. (This command will generate a new, makefile with the rules for the new file included.), Go to directory /usr/src/, Issue command make libraries, All these steps will compile and install the updated posix library.

Design:

There is a professor process and several student processes. prof.c implements the actions of the professor and student.c implements the actions of the student. One prof process and multiple student processes running at the same time. The prof process is started first. The semaphore initialization is done in the prof.c . It checks if the students are available to ask questions and once the students are answred it does a sem_up and waits for the next students question . In the student.c processes are forked and when one process is asking the question or when sem_down is called by a process, the other processes are in the waiting queue until the prof does the sem_up.

Design of each function

askQuestion by student in student.c

sem_down – it is the students turn to ask the question to the professor

Prints out that the student asks question

Answering the question in prof.c

sem_init – initialize the semaphore

void askQuestion(int studentNbr) // function for the student asking a question to the prof

```
{
        sem_down(1);
        printf("student %d: ask question\n", studentNbr);
        sleep(4);
        printf("student %d: leave room\n", studentNbr);
}
int main(int argc, char **argv)  // main for the student program
{
        pid_t pids[100];  //array to store pids
        int i;
        int n = atoi(argv[1]);;          //no.of students input
        for (i = 1; i <= n; ++i) {       //forking new processes for the number of students
                                           given as input at the console.
          if ((pids[i] = fork()) < 0) {
                perror("fork");
                abort();
          } else if (pids[i] == 0) {
                sleep(2);
                askQuestion(i);
                exit(0);
          }
        }

        int status;              //waiting
        pid_t pid;
        while (n > 0) {
          pid = wait(&status);
          --n;            // Remove pid from the pids array.
        }
        sem_release(1);         //semaphore release
}
```

Sleep is introduced for the professor to wait until the student asks the question

The professor calls sem_up once the prof answers the students question and this student leaves once hos/her question is answered. This gives the next student the chance to ask his/her question.

The while loop facilitates the professor to answer the questions to the number of students given at the console during runtime.

The professor process runs first and then the student processes run.

```
int main(int argc, char **argv)                    //main for the prof program
{
        int i=1, noOfStudents = atoi(argv[1]);
        sem_init(1,0);                             //initializing semaphore
        sleep(5);
```

```
        while( i <= noOfStudents)               //answering question loop for the given no.of
                                                  students
        {
                sem_up(1);
                sleep(3);
                printf("prof: answer question\n");
                sleep(1);
                i++;
        }
        return 0;
}
```

There is one professor process running and if there are three student processes that are started simultaneously. The output to the console should be:

Student1: ask question

Prof: answer question

Student1: leave room

Student2: ask question

Prof: answer question

Student2: leave room

Student3: ask question

Prof: answer question

Student3: leave room

Testing:

The test strategy for this design will be to try this out with varying student numbers to check if the sequence of Student: ask question Professor: answer question and Student: leave room remain the same. This has been tested and it works fine for varying input.