Indian Institute of Information Technology Guwahati
Operating Systems Lab (CS232)

# CG1 / Assignment 3

Dated 3rd February, 2020

- **Submission instructions:** Upload Roll_Server.c (cpp) and Roll_Client.c (cpp) using the link http://172.16.1.102:8080/ within the stated deadline
- Your submission will be checked through plagiarism checker. Copy cases will be credited with ‘0’ mark for this assignment.
- **Deadline: 10th Feb, 2020, 5pm.**

## Problem 1

In this assignment you will learn the following:

- How the kernel schedules different processes through multiprogramming.
- Whether the kernel favors specific processes
- How much time a process spends in different queues of the OS.
- How other processes running in the system affects the performance of a given process.
- The CPU utilization of the system

Write a collection of  sufficient no. of processes which carry out the following different types of tasks independently:

1. Only computation
2. Only printfs
3. Low computation, heavy console output (printfs)
4. Heavy File I/O

Tune each of the above process to run for 30 seconds when it runs standalone. Then run them simultaneously with different combinations.
Gather per process and system statistics e.g. required cpu time, system call time, turnaround time, wait time, penalty ratio, system cpu utilization in each test combination.

Report your numbers in a file with an explanation about your conclusions/observations.

Some Pointers: You can use *procinfo*, *getrusage*, *getrlimit* etc for gathering process statistics.

## Problem 2

In this assignment you will develop a system for multiple terminals which will be interacting with each other using message queues. The commands on a given terminal (shell), will be mirrored in the coupled terminal along with output.

Server: The system will have a server, which will start a message queue. It will also maintain the list of client terminals which joins the coupling. The server will be started first which will maintain the message queue and the coupling list. After receiving a message from a client for the group, it sends the message to all the members of the group.

Client: The client program will have an interface similar to that of the shell assignments. The programs will share a message queue, through which you can communicate among with the other terminals (via the server). There is a joining protocol to the terminal mirroring. This protocol will add the members to the list of terminals sharing their screen (like Team Viewer for GUI). After a terminal joins the group all the commands and the relevant outputs will be broadcasted to all the members which has joined the group.

Joining Protocol:

1. When a client types couple in the terminal, it is added to the broadcast group and an ID is sent to it from the server.
2. For leaving the group a client has to type uncouple, which removes it from the broadcast group.

Broadcasting Protocol:

While broadcasting, when a member of the group types a command, it is executed in the terminal. The command along with the relevant output of the command is then forwarded to the server for broadcasting.

Interface

Server: The server will be started first and operate in the background and maintain the necessary components. It will print the updated list of clients on the screen.

Client:

1. It will have an interface similar to the shell. It will run in an infinite loop and exit when “exit” command is given.
2. Upon joining the coupling group the server will send an ID to the client, and it will be displayed in the shell, to tell the user it’s ID in the group.
3. Any command entered at a client is executed and the command along with the relevant output is sent to all the members of the group. The commands can be executed using a bash in forked subprocess. Only single line commands need to be supported.
4. The command entered by another terminal in the group will be displayed in the terminal of all the members (to be sent by server).
5. In addition to standard shell commands, “couple” and “uncouple” should be supported. Couple should join the server and mirror all other coupled clients as well as the current client to the other clients. Uncouple should “unjoin” from the server. Initial state when starting the client can be “uncoupled”.
6. The ID of the origin terminal is also displayed, to the member terminal. Below example shows sample behaviour.

Example: Let’s say we have two coupled terminals (command typed by the terminal are shown in bold)

| Terminal 1 | Terminal 2 |
|---|---|
| /usr/home: **ls**<br>File1.c<br>File2.txt<br>Terminal 2: mkdir test<br>Terminal 2 : cd test<br>Terminal 2: ls<br>/usr/home: **uncouple**<br>/usr/home: **cd ..**<br>/usr:<br>/usr: **couple**<br>ID : 1<br>Terminal 2 : rmdir test | Terminal 1 : ls<br>File1.c<br>File2.txt<br>/usr/home: **mkdir test**<br>/usr/home : **cd test**<br>/usr/home/test: **ls**<br>Terminal 1 : uncouple<br>/usr/home/test : **ifconfig**<br>/* Output of the command */<br>/usr/home/test: **cd /usr/home**<br>Terminal 1 : couple<br>/usr/home : **rmdir test** |

Deliverable:

Server.c (cpp), which will create the message queue and maintain the list of clients.

Client.c (cpp), which will act as the shell and run in each terminal and execute commands.

Upload the files in the portal appending 'Roll_' to the file name, for an example Roll_Server.c and Roll_Client.c

Strategy:

- The client sends the message to the server which sends the message to the members.
- For executing a command in a terminal bash in a subprocess can be used.
- Only one message queue can be used by all the programs using priority feature.

Message Queue:

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by **msgget()**. New messages are added to the end of a queue by **msgsnd()**. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd() when the message is added to a queue. Messages are fetched from a queue by **msgrcv()**. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

All processes can exchange information through access to a common system message queue. The sending process places a message (via some (OS) message-passing module) onto a queue which can be read by another process. Each message is given an identification or type so that processes can select the appropriate message. Process must share a common key in order to gain access to the queue in the first place.

POSIX Messages: <mqueue.h>

The POSIX message queue functions are:

`mq_open() -- Connects to, and optionally creates, a named message queue.`

`mq_close() -- Ends the connection to an open message queue.`

`mq_unlink() -- Ends the connection to an open message queue and causes the queue to be removed when the last process closes it.`

`mq_send() -- Places a message in the queue.`

`mq_receive() -- Receives (removes) the oldest, highest priority message from the queue.`

`mq_notify() -- Notifies a process or thread that a message is available in the queue.`

`mq_setattr() -- Set or get message queue attributes.`

`The basic operation of these functions is as described above. For full function prototypes and further information see the UNIX man pages`

```
// C Program for Message Queue (Writer Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mesg_buffer {
      long mesg_type;
      char mesg_text[100];
} message;
int main()
{
      key_t key;
      int msgid;
      // ftok to generate unique key
      key = ftok("progfile", 65);
```

```c
      // msgget creates a message queue and returns identifier
      msgid = msgget(key, 0666 | IPC_CREAT);
      message.mesg_type = 1;
      printf("Write Data : ");
      gets(message.mesg_text);

      // msgsnd to send message
      msgsnd(msgid, &message, sizeof(message), 0);

      // display the message
      printf("Data send is : %s \n", message.mesg_text);

      return 0;
}

// C Program for Message Queue (Reader Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mesg_buffer {
      long mesg_type;
      char mesg_text[100];
} message;

int main()
{
      key_t key;
      int msgid;
      // ftok to generate unique key
      key = ftok("progfile", 65);
```

```
    // msgget creates a message queue and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);

    // msgrcv to receive message
    msgrcv(msgid, &message, sizeof(message), 1, 0);

    // display the message
    printf("Data Received is : %s \n", message.mesg_text);

    // to destroy the message queue
    msgctl(msgid, IPC_RMID, NULL);

    return 0;
}
```

Read the following link for more details:

IPC:Message Queues:<sys/msg.h>