BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
IS ZC462 : Network Programming
I Semester 2015-16
Assignment
Weightage: 15% (45M) Due Date of Submission: 25-OCT-2015

==

Important to Note:

1. **Don't use temporary files and system() function.**
2. **Only working programs will be evaluated. If there are compilation errors, it will not be evaluated.**
3. *Provide makefile for each problem.*
4. Upload instructions are given at the end.
5. For any clarifications please contact me (khari@pilani.bits-pilani.ac.in).

Plagiarism will be thoroughly penalized.

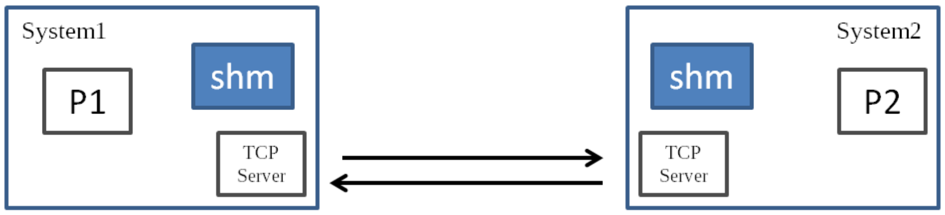
==

P1. In this problem you will implement a command line interpreter or shell**.** The shell takes in a command from user at its prompt and executes it and then prompts for more input from user. **[10M]**

- shell should execute a command using fork()and execve() calls. It should not use temporary files, popen(), or system() calls. It should not use sh or bash shells to execute a command. Once the command is completed it should print its pid and status.
- shell should support <, >, and >> redirection operators.
- shell should support pipelining any number of commands. E.g.: `ls|wc|wc, cat x| grep pat| uniq| sort`
- shell should support two new pipeline operators "||" and "|||". E.g.: `ls -l || grep ^-, grep ^d`. It means that output of ls -l command is passed as input to two other commands. Similarly "|||" means, output of one command is passed as input to three other commands separated by ",".

Write a program called *shell.c* that implements the above requirements.

P2. In this problem, goal is to facilitate two processes on two different systems use shared memory for IPC as if they are on the same system as shown in the diagram below. **[16M]**



- P1 and P2 will use shared memory for inter process communication. But they use the following API instead of usual System V shared memory API.

```
int rshmget(key_t key, size_t size);
void rshmat(int rshmid, void* addr);
int rshmdt(int rshmid, void* addr);
int rshmctl(int rshmid, int cmd);
void rshmChanged(int rshmid);
```

API	**TCP Server**	**Remote TCP Servers**
`rshmget`(): it prepares a message with appropriate contents and send to local TCP server.	It checks the key. If it already exists it returns the rshmid otherwise, it creates shared memory locally and stores the shmid. Generates the rshmid, a random number, and sends a message to other nodes about the new shared memory creation. Returns the rshmid. This node becomes the owner of the shared memory.	Create the shared memory locally and map it with the rshmid in the structure.
`rshmat()`:it prepares a message with appropriate contents and send to local TCP server.	It calls shmat() and returns the return address of shmat(). Sends a message to other nodes to increase ref count.	Increase ref count for the corresponding shm segment.
`rshmdt()`:it prepares a message with appropriate contents and send to local TCP server.	It calls the shmdt() and sends message to other nodes to decrease ref count.	Decrease ref count for the corresponding shm segment.
`rshmctl()` : only command applicable is IPC_RMID. it prepares a message with appropriate contents and send to local TCP server.	It calls the shmctl() and informs the same to other nodes using multicast message	Remove the shared memory corresponding to rshmid that is received from the owner.
After completing the write operation, the application calls `rshmChanged()`.it prepares a message with appropriate contents and send to local TCP server.	The server reads the data from shared memory and sends to all other nodes to update their local shared memory segments.	Update the local shared memory corresponding to rshmid received.
	When the server starts up, it sends hello message to the group. It rears the replies and updates its internal state tables.	Reply with the local state table

- TCP server stores the following information for every shared memory segment request it receives.

```
struct rshminfo{
      int rshmid;
                  /*unique id across all systems. created by the
                  first system*/
      key_t key;        /*key used to create shm segment*/
      int shmid;        /*shmid returned by the local system*/
      void *addr;       /*address returned by the local system*/
      int ref_count;    /*no of processes attached to*/
      struct sockaddr_in *remote_addrs; /* list of remote end
      points*/
};
```

- TCP server maintains connections to all remote TCP servers. It uses message queues for communicating with local system processes and TCP connections to communicate with remote systems.

  Implement client.c (run as P1 or P2), rshmAPI.c (implementation of API) and rshmServer.c (run as TCP server) for the above specifications. client.c uses rshmAPI.c.

P3. Write a program webserver.c for concurrent web server supporting HTTP GET requests. It supports concurrency through a event-driven model, using epoll() edge-triggered notifications.

- server maintains a message queue to queue the events.
- main thread waits on epoll_wait() to get IO notifications from kernel. For each IO notification, it adds an event to the message queue.
- "process" thread waits on message queue msgrcv() call. It gets a message and responds according to its state. While processing the message it may add new events to the queue.
- each client request goes through states: READING_REQUEST, HEADER_PARSING, READING_DISKFILE, WRITING_HEADER, WRITING_BODY, DONE. Next state is reached only after the previous one is complete.
- All IO (read and write) operations are non-blocking. This necessitates buffer management.
- data pertaining to client's request is stored in a central data structure such as hash table. This is to avoid unnecessary copying of data which may happen if we keep client data in the message queue itself.
- When a request reaches DONE state, connection remains open for another request. Client connection is closed only when EOF is received from client. That may happen during any state.
- web server testing tools such apache ab or httpperf can be used as clients.

[10M]

P4. Write a C program *multicast.c* for IP Multicast that does the following.

- It takes multicast group ip and port on command line. It joins the group and waits for messages.
- It sends a "hello-" + time() every 15 seconds to count how many members are present in the group.
- Any member who receives "hello-"+time(), simply echoes the same.
- The member which has sent hello, counts the replies received within 5 seconds and displays count on the screen.
- Same program is run multiple times to create multiple members.
- Each member prints the sender ip and message received every time a message is received.
- member exits only when Ctrl-c is pressed. Before it exits, it sends "bye-" +time() to all in the group.

[9M]

How to upload?

- Make a directory for each problem like P1, P2 etc and copy your source files into these directories.
- Tar all of them including group.txt into idno.tar
- Upload on taxila (http://taxila).

===End of assignment===