Message Queue

- Used to transfer packets of data from one process to another. It is bi-directional IPC mechanism.
- Internally OS maintains list of messages called as "message queue" or "mailbox".
- The info about msg que is stored in a object. It contains unique KEY, permissions, message list, number of messages in list, processes waiting for a message to receive (waiting queue).

Message Queue Syscalls

msgget()

- Create message queue object.
- mqid = msgget(mq_key, flags);
  - arg1: unique key
  - arg2: IPC_CREAT | 0600 to create new message queue.
  - returns message queue id on success

msgctl()

- Get info about message queue or destroy message queue
- msgctl(mqid, IPC_RMID, NULL) -- to destroy message queue
  - arg1: message queue id
  - arg2: commad = IPC_RMID to destroy message queue
  - arg3: NULL (not required while destroying message queue)

msgsnd() - send message into que

- Send message in the message queue.
- ret = msgsnd(mqid, msg_addr, msg_size, flags)
  - arg1: message queue id
  - arg2: base address of message object
  - arg3: message body size
  - arg4: flags (=0 for default behaviour)
  - returns 0 on success.

msgrcv() - receive message from que

- Receive message from the message queue of given type.
- ret = msgrcv(mqid, msg_addr, msg_size, msg_type, flags)
  - arg1: message queue id
  - arg2: base address of message object to collect message (out param)
  - arg3: message body size
  - arg4: type of message to be received
  - arg5: flags (=0 for default behaviour)
  - returns number of bytes (body) received on success.

IPC commands

- ipcs -- to show IPC (Shared memory, Semaphore, Message queue) status
- ipcrm -- to delete IPC (Shared memory, Semaphore, Message queue) object

IO Redirection

- Shell provides redirection feature, which which input, output and/or error of the process can be redirected into the file instead of terminal.
- terminal> command < in.txt
- terminal> command > out.txt
- terminal> command 2> err.txt
- redirection can be done programmatically using dup() system call.

dup() syscall

- Copies given file descriptor on lowest numbered available file descriptor.
- dup(fd)
  - arg1: Copies file descriptor on lowest numbered available file descriptor.

dup2() syscall

- Copies given old file descriptor on given new file descriptor.
- dup2(old_fd, new_fd)
  - arg1: Old file descriptor to be copied.
  - arg2: New file descriptor on which old fd is to be copied. If new_fd is open, it will be first closed and old_fd will be copied on it.

Pipe

- Pipe is used to communicate between two processes.
- It is stream based uni-directional communication.
- Pipe is internally implemented as a kernel buffer, in which data can be written/read.
- If pipe (buffer) is empty, reading process will be blocked.
- If pipe (buffer) is full, writing process will be blocked.
- If writer process is terminated, reader process will read the data from pipe buffer and then will get EOF.
- If reading process is terminated, writing process will receive SIGPIPE signal.
- There are two types of pipe:
  - Unnamed Pipe
  - Named Pipe

Unnamed Pipe

- Used to communicate between related (parent-child) processes.
- terminal> who | wc
- Internally pipe is created using pipe() syscall.

pipe() syscall

- To create unnamed pipe.
- ret = pipe(fd); // int fd[2];
  - arg1: array of two ints to collect fd (out param).
  - returns 0 on success.
  - fd[] gets two ends of pipe.

- fd[0] -- read end (file descriptor) of pipe
- fd[1] -- write end (file descriptor) of the file

Named Pipe

- Is also called as FIFO
- Used to communicate between unrelated processes.
- Named pipe is created using mkfifo command, which internally calls mkfifo() syscall.
- Fifo is special file. In its inode, type=p and no data blocks are created.

mkfifo() syscall

- To create named pipe.
- ret = mkfifo("fifo path", mode);
  - arg1: path of fifo to be created
  - arg2: fifo mode (permissions)
  - returns 0 on success.

Assignment

1. The child process send two numbers to the parent process via message queue. The parent process calculate the sum and return via same message queue. The child process print the result and exit. The parent process wait for completion of the child and then exit.
2. The child process send two numbers to the parent process via pipe. The parent process calculate the sum and return via another pipe. The child process print the result and exit. The parent process wait for completion of the child and then exit.
3. Write a program to find the size of the pipe buffer.
4. Write a program that will launch two programs (e.g. who and wc). The output of first program (e.g. who) should be given as input to second program (e.g. wc).