IPC

Class Notes

Inter Process Communication

- Linux kernel supports inter process communication mechanisms by providing three different kernel managed resources.
    1) Message queues
    2) Shared memory
    3) Semaphores
- BSD and POSIX standards have different specifications for allocations and management of the above resources. Linux kernel supports both the standards and implementations.
- Posix's implementation of IPC resources is file system oriented. System V and BSD implementations are specific system calls for allocation and access of these resources.

Message queues (msgq)

Message queues are kernel managed list of buffers that can be used to exchange messages between n processes.

Reasons for using msgq than files

- Persistent data files can be used as messaging resources between applications/processes. But persistent files are sourced from a storage device and read/write operations on them include disk I/O.
- Data return to persistent file is retained in the file even after it is read by the intended receiver.
- Special logical files called pipes can be used as a messaging resource but pipes do not provide message boundaries and are not recommended to be used in one-to-many communication scenarios.

System V message queues

- Each message is identified with message id.
- Reading process can read a message using message id.
- Message can be read in any order.
- Supports one-to-many communication

Posix message queues

- These are managed and implemented under **mqueue** file system.
- Applications are provided with file API's to initiate message queue operations.
- Each message must be assigned with priorities.
- Reader can read only highest priority message from queue.
- Reader can register for message arrival notifications.

- Allocating message queues

```
#define NAME "/my_mq"

mqd_t mqd;
 struct mq_attr attr;

 attr.mq_maxmsg = 50;
 attr.mq_msgsize = 2048;

 mqd = mq_open(NAME,O_RDWR | O_CREAT, 666, &attr);
```

  - To mount message queue

    mount –tmqueue none /media

  - To view message queue

    cd /media

    cat my_mq

- Writing messages on message queue

```
mqd_t mqd;
unsigned int prio = 50;

flags = O_WRONLY ;

 mqd = mq_open(NAME , flags);
mq_send(mqd,"writing message to queue", 25 , prio);
```

- Reading from message queue

```
struct mq_attr attr;
flags = O_RDONLY;
 prio = 50;
mq_getattr(mqd, &attr)
buffer = malloc(attr.mq_msgsize);
mq_receive(mqd, buffer, attr.mq_msgsize, &prio);
```

- Posix message queue interface provides message arrival notification facility. Receiver process can register for notifications to deliver on arrival of fresh message.

Sample code:

```
struct sigevent sev;
sev.sigev_notify = SIGEV_SIGNAL;
sev.sigev_signo = NOTIFY_SIG;

mq_notify(mqd, &sev);
```

- Notification type can also be set for a thread to run on arrival of thread.

Sample code:

```
static void notifySetup(mqd_t *mqdp);
struct sigevent sev;

 sev.sigev_notify = SIGEV_THREAD;        /* Notify via thread */
 sev.sigev_notify_function = threadFunc;
 sev.sigev_notify_attributes = NULL;
/* Could be pointer to pthread_attr_t structure */

 sev.sigev_value.sival_ptr = mqdp;

 mq_notify(*mqdp, &sev) ;
```

- Posix message queues can be deallocated using **mq_unlink().**

Shared memory

- Message queues cannot be used for sharing persistent data between concurrent processes or user level threads.
- Shared memory allows n number of processes to map and share a common buffer with shared data.
- Accessing data in shared memory does not involve user-kernel mode transition, since shared memory is directly mapped to the requesting process's address space.

Posix shared memory

- Posix IPC standard provides shared memory through a file system "shmfs".
- Each shared memory block is mapped to an inode of "shmfs".

Allocating shared memory

- Allocate a new shm_fs inode and assign buffer

```
fd=shm_open("/myshm" , O_CREAT | O_RDWR , 666);
ftruncate(fd , size);
```

- Map shared memory to the process address space

```
map(NULL , size ,PROT_READ|PROT_WRITE, MAP_SHARED, fd ,0);
```

- Shared memories are mounted automatically on **/dev/shm**.
- Process accessing shared memory need to use **mmap** with shared memory file descriptor.
- Process sharing common data using shared memory must ensure validity