a Raju Alluri presentation

Advanced Unix Programming

Module 07

Raju Alluri
askraju @ spurthi.com

Advanced Unix Programming: Module 7

- Inter-Process Communication (IPC)
  - How two processes on the same system communicate with each other
  - Several concepts have evolved
    - Pipes
    - Named Pipes
    - Message Queues
    - Semaphores
    - Shared Memory

Techniques used in IPC

- Pipes
  - Have two (related) processes communicate with each other using file descriptors (that are common)
- Named Pipes (FIFOs)
  - Have (unrelated) processes communicate with filesystem like storage called named pipes
- Message queues
  - Have (unrelated) processes communicate with messages stored in kernel as queues

Pipes

- The `pipe`() system call creates a pipe
  - A pipe is a set of two descriptors, one for output and the other for input, both of them connected together
  - Then the process forks to get a child. The parent and child share the pipe now.
  - Parent closes the input descriptor. Child closes the output descriptor
  - Now parent can write to the output descriptor and it can be read by child using the input descriptor

Pipes

- Synopsis of the `pipe()` call

```
int pipe(int filedes[2]);
```

  - Create a pipe with `filedes[0]` as the input channel and `filedes[1]` as the output channel
  - Now the process does a `fork()`
  - For a pipe "from" parent "to" child
    - The parent does `close(filedes[0])`
    - The child does `close(filedes[1])`
  - For a pipe "from" child "to" parent
    - The parent does `close(filedes[1])`
    - The child does `close(filedes[0])`

The popen() and pclose() calls

- The `popen()` call returns a file pointer (either input or outout) for a process executed with a given command

```
FILE *popen(const char *cmd, const char
  *type);
```

  - The argument `cmd` gives the command to be executed
  - The argument `type` gives the direction of the pipe
    - "w" indicates the returned file pointer is for writing
    - "r" indicates the returned file pointer is for reading
- Closing the file created with `popen()`

```
int pclose(FILE *fp);
```

  - Closes the pipe that was opened with `popen()`

Named Pipes (FIFOs)

- The `mkfifo()` call makes a named pipe (just like a file on the filesystem)

```
int mkfifo(const char *path, mode_t mode);
```

  - Once the named pipe is created, use regular File I/O calls to operate on the pipe
    - e.g. Open, read, write, close etc.
  - Typical Uses
    - Shell commands pass data from one shell pipeline to another
    - FIFOs are used in client server application to pass data

Named Pipes (FIFOs)

- The `mkfifo()` call makes a named pipe (just like a file on the filesystem)

```
int mkfifo(const char *path, mode_t mode);
```

  - Once the named pipe is created, use regular File I/O calls to operate on the pipe
    - e.g. Open, read, write, close etc.
  - Typical Uses
    - Shell commands pass data from one shell pipeline to another
    - FIFOs are used in client server application to pass data

Message Queues

- Message Queues are linked lists of messages stored within the kernel
  - Each queue has a Queue ID.
  - A queue can be created/opened by using `msgget()`
  - New messages are appended to the queue using msgsnd
    - Each message has type, length, data etc.
  - Messages are fetched from the queue using msgrcv
    - By using type, messages can be fetched in any order

Message Queue Calls

- Creating/Opening a message queue

```
int msgget(key_t key, int flag);
```

  - Return the Queue ID, which is used by other functions

Message Queue Calls

- Placing a message onto a queue

```
int msgsnd(int qid, const void *ptr, size_t
  nbytes, int flag);
```

  - The `qid` is ID of the queue as returned by msgget()
  - The `ptr` points to a memory area with the first field being the message type, the second field being the message data

```
struct mymessage{
    long msgtype;
    char mtext[256];
};
```

  - The `nbytes` gives the number of bytes in the message

Message Queue Calls

- Placing a message onto a queue

```
int msgrcv(int qid, struct msgbuf *ptr, int
  length, long msgtype, int flag);
```

  - The `qid` is ID of the queue as returned by msgget()
  - The `ptr` points to a memory area where the returned message is stored
  - The `length` is the max length of the message received
  - The `msgtype` is used to selectively fetch messages
    - `msgtype = 0`: Fetch first message in the queue
    - `msgtype > 0`: Fetch first message where type equal to msgtype
    - `msgtype < 0`: Fetch first message with the lowest type <= msgtype

Message Queue Calls

- Operating on a queue

```
int msgctl(int qid, int cmd, struct
  msqid_ds *buf);
```

  - The `qid` is the ID of the queue as returned by msgget()
  - The `cmd` is one of
    - IPC_STAT: Get the msgid_ds from the queue
    - IPC_SET: set several fields (related to permissions etc.) from buf to the queue
      - See man page for more info
    - IPC_RMID: remove the message queue from the system

Semaphores

- Semaphores are the way to give access to data objects
- Typical algorithm
  - If the semaphore value is >0, decrement the value by 1 and start using the resource
    - Once done with using the resource, increment the semaphore value by 1
  - If the semaphore value is 0, sleep for sometime and check the value again
- The test and increment/decrement should be atomic

System V Semaphores

- Complexity in System V semaphores
  - Semaphores are implemented as a set of values instead of a single value
  - The creation of a semaphore and initialization are not atomic. So care should be taken while using them
  - Programs terminating without releasing the sempahores allocated for them is an issue.

Semaphore structure members

- At semaphore set level
  - Permissions
  - Number of semaphores
  - Last semop() time
  - Last change time
- At semaphore level
  - Semaphore value
  - PID for last operation
  - Number of processes awaiting semval>currval
  - Number of processes awaiting semval = 0

Semaphore Calls

- Creating a semaphore

```
int semget(key_t key, int nsems, int flag);
```

  - Returns the semaphore ID that is used in other calls

- Controlling a semaphore

```
int semctl(int semid, int semnum, int cmd,
  union semun arg);
```

  - The `semun` union is defined as

```
union semun {
   int val;
   struct semid_ds *buf;
   ushort *array;
};
```

Semaphore Calls

- Controlling a semaphore
  - The `cmd` is one of
    - IPC_STAT: Get the semid_ds from the semaphore set
    - IPC_SET: set several fields (related to permissions etc.) from arg.buf to the semaphore set
    - IPC_RMID: remove the semaphore set from the system
    - GETVAL: Return the value of semval for semnum
    - SETVAL: Set the value of semval for semnum, using arg.val
    - GETPID: return the value of sempid for the member semnum
    - GETNCNT: Return the value of semncnt for the member semnum
    - GETZCNT: Return the value of semzcnt for the member semnum
    - GETALL: Fetch all the semaphore values arg.array
    - SETALL: Set all the semaphore values using arg.array

Semaphore Calls

- Operating on a semaphore

```
int semop(int semid, struct sembuf
  semoparray[], size_t nops);
```

  - Struct `sembuf` is defined as

```
struct sembuf {
    ushort sem_num;
    short sem_op;
    short sem_flag; /* IPC_NOWAIT, SEM_UNDO */
};
```

    - If sem_op > 0, return of resources by process
    - If sem_op < 0, want to get resource that the semaphore controls
    - If sem_op = 0, want to wait until the semaphore value becomes 0

Shared Memory

- This form of IPC is very efficient in terms of kernel interactions
- Two or more processes share a memory object
  - The memory object is used just like any other block of memory
- Two distinct forms of shared memory
  - System V shared memory
  - Posix shared memory
    - Not discussed here in this class

Shared Memory Calls

- Creating a shared memory

```
int shmget(key_t key, int size, int flag);
```

  - Returns the shared memory ID that is used in other calls

- Controlling shared memory

```
int shmctl(int shmid, int cmd, struct
  shmid_ds *buf);
```

  - IPC_STAT
  - IPC_SET
  - IPC_RMID
  - SHM_LOCK (only by superuser)
  - SHM_UNLOCK (only by superuser)

Shared Memory Calls

- Attaching shared memory at a particular place

```
void *shmat(int shmid, const void *shmaddr,
  int flag);
```

  - Attach the shared memory to the current process space and return the address
  - If `shmaddr` is
    - NULL: The kernel decides at which location to attach the shared memory (recommended)
    - Non-NULL:
      - If `flag` contains SHM_RND, then attach the shared memory at `shmaddr` rounded down to SHMLBA
      - Attach the shared memory at `shmaddr` otherwise

Shared Memory Calls

- Detaching shared memory

```
void *shmdt(const void *shmaddr);
```

  - Detach the shared memory pointed by `shmaddr`
    - The shared memory is not deleted though.