Principles of Operating Systems

Abhishek Dubey

Inter Process Communication

Spring 2016

Inter process communication(IPC)

- IPC refers to a set of services/methods provided by the operating system used for exchanging information/messages between processes.
- IPC mechanisms can be roughly divided into
  - Synchronization (semaphores/monitors)
  - Message passing (can be used to exchange information as well as for synchronization)
  - Shared memory
  - signals
  - Shared files.

Sidebar: File Descriptors

- A *File Descriptor* is a handle provided by the OS when interacting with files
  - Typically an index into a data structure in kernel memory
  - Used for all interactions with that file
- In UNIX, almost *everything* is treated like a file – files, pipes, sockets, you name it
- Typical ways to interact with a file descriptor:
  - *open* – creates a new file descriptor that points to a resource
  - *write* – Write a stream of bytes to that resource
  - *Read* – Read a stream of bytes from that resource
  - *Close* – Indicate you no longer need access to the resource

Sidebar: File Descriptors

- All processes have at least three open file descriptors:
  - 0 – standard input
  - 1 – standard output
  - 2 – standard error
- A particular file descriptor can be *reopened* and pointed at a different resource

Message Passing

- The actual function is normally provided in the form of a pair of primitives:

  send (destination, message)

  » Blocking/non blocking

  receive (source, message)

  » Blocking/non blocking

» Blocking send and receive can be used for synchronization as well as mutual exclusion

» Non-blocking mode is often referred to as asynchronous communication

» A process sends information in the form of a *message* to another process designated by a *destination*

» A process receives information by executing the `receive` primitive, indicating the *source* and the *message*

Signal API - recap

- Sender side
  - **int kill(pid_t** *pid*, **int** *sig***);**
- Recepient side
  - Register a signal handler. Note that this is possible only if the signal can be trapped
    - sighandler_t signal(int *signum*, sighandler_t *handler*);
  - Where sighandler_t is
    - typedef void (*sighandler_t)(int);
  - int pause(void);
    - Wait until a signal is delievered that either terminates the process or causes the invocation of a signal handler.

Pipes

- buffers allowing two processes to communicate on the producer-consumer model
    - first-in-first-out queue, written by one process and read by another

Two types:

- Named
- Unnamed

IPC: Pipes & FIFOs (1/2)

- Pipes & FIFOs (Named Pipes) provide unidirectional communication between a *read end* and a *write end*
- Only different in *how they are created* and *who can access them*
  - Pipes are created with the *popen* or the pipe library routine, and access is limited to *related processes*
  - FIFOs are *named objects* in the file system created by *mkfifo* and may be opened by any process

IPC: Pipes & FIFOs (2/2)

- Communication over pipes is a *stream of bytes*; applications must
  - Use a common message format
  - Indicate/detect message boundaries
- Pipes have *limited capacity*
  - Writing to a full/reading from an empty pipe will block
  - There are, of course, some exceptions

Named Pipe Creation

```
int mkfifo (const char *pathname, mode_t mode)
```

- Creates a *special file* at the given *pathname*
  - Any process that has access to the file may open it
  - Access permissions can be controlled with the *mode* parameter
- This file may then be opened in *read* mode or *write* mode
  - Both sides (read and write) must be opened by at least one process
  - *Open* will typically block until the other side has opened

Creating a FIFO File

- The easiest way to create a FIFO file is to use the *mkfifo* command.
- You can also use the API shown in previous slide
- You may also use the *mknod* command to accomplish the same thing.
- To learn more about these commands check out the man pages on them
- The following example shows one way to use the mkfifo command:
  - *mkfifo /tmp/myFIFO*

Using a FIFO File

- Just like a file. But the read and write will follow pipe semantics.
  - Writing to a full/reading from an empty pipe will block
- *int open(const char *pathname, int flags);*
- *int read(int fd, void *buf, size_t count);*
- *int write(int fd, const void *buf, size_t count);*
- *int close(fd);*

Pipe Creation

```
int pipe (int pipefd[2]);
int pipe2 (int pipefd[2], int flags)
```

- Creates a *read* and *write* pipe, placing the *file descriptors* in *pipefd*
  - pipefd[0] is the *read end*
  - pipefd[1] is the *write end*
- *Pipe2* has flags that control two properties:
  - O_NONBLOCK – full/empty pipes do not cause read/write to block
  - O_CLOEXEC The pipes are closed on exec.
- man 2 pipe

A bit more about pipes

- Pipes are half duplex
  - That is the data flows typically from a writer to a reader in only one direction.
- The pipes can only be used between processes with common ancestor
  - Parent-child
  - Child-child
- Reading from a pipe whose write end has been closed returns 0 (end of file)
- Writing to a pipe whose read end has been closed generates SIGPIPE

Pipe Capacity

A pipe has a limited capacity. If the pipe is full, then a write(2)

will block or fail, depending on whether the **O_NONBLOCK** flag is set (see below).

In Linux versions before 2.6.11, the capacity of a pipe was thesame as the system page size (e.g., 4096 bytes on i386). Since Linux 2.6.11, the pipe capacity is 65536 bytes.

Since Linux 2.6.35, the default pipe capacity is 65536 bytes, but the capacity can be queried and set using the fcntl(2) **F_GETPIPE_SZ** and **F_SETPIPE_SZ** operations. See fcntl(2) for more information.

A bit more about pipes

- The pipe on child side can be associated with the STDIN using dup2 system call.
  - int dup2(int oldfd, int newfd);

```
int fd[2];
pid_t pid;
pipe(fd); //fildes[0] is open for reading and
//fildes[1] is open for writing
pid = fork();
if(pid == 0) {
 dup2(fd[0], STDIN_FILENO);
 exec(<your program>); }
```

- Anything written by parent on the pipe will be read by the child on stdin.

dup2

- Duplicate a file descriptor
- Used to redirect an existing file description to a file pointed by the new file descriptor

- **`int dup2(int oldfd, int newfd);`**
  - Copies descriptor table entry **`oldfd`** to descriptor table entry **`newfd`**

Pipe to a subprocess

- A common use of pipes is to send data to or receive data from a program being run as subprocess.
- One way of doing this is by using a combination of pipe (to create the pipe), fork (to create the subprocess), dup2 (to force the subprocess to use the pipe as its standard input or output channel), and exec (to execute the new program).
- Or, you can use popen and pclose.

Popen/pclose

- Function: FILE * **popen** *(const char *command, const char *mode)*
  - The popen function is closely related to the system function;
  - It executes the shell command *command* as a subprocess.
  - However, instead of waiting for the command to complete, it creates a pipe to the subprocess and returns a stream that corresponds to that pipe.

- Function: int **pclose** *(FILE *stream)*

Popen example

```
#include <stdio.h>
#include <stdlib.h>

void
write_data (FILE * stream)
{
  int i;
  for (i = 0; i < 100; i++)
    fprintf (stream, "%d\n", i);
  if (ferror (stream))
    {
      fprintf (stderr, "Output to stream failed.\n");
      exit (EXIT_FAILURE);
    }
}
```

```
int main (void)
 {
 FILE *output;
output = popen ("more", "w");
if (!output) {
  fprintf (stderr, "Could not run more.\n");
  return EXIT_FAILURE;
 }
write_data (output);
pclose (output); return EXIT_SUCCESS; }
```

Pipe Summary

- Signal events (one pipe)
  - Wait for a message
- Synchronize (one or two pipes)
  - Wait for a message or set of messages
  - You send me a message when you are ready, then I’ll send you a message when I am ready
- Communicate (one or two pipes)
  - Send messages back and forth

Message Queues

- Linked list of messages stored in the kernel
  - A message is a block of bytes.
- A message queue is like a mailbox (with maximum size)
  - Senders can add messages
  - Receivers can take message
- Often the operating system guarantees that the action of taking 1 message and adding 1 message is mutually exclusive.
- Senders have an option of blocking forever or wait until timeout if queue is full.
- Receivers block or wait for a specified time if the queue is empty.

POSIX Message Queues: Creation

```
mqd_t mq_open (const char *name, int oflag);
mqd_t mq_open (const char *name, int oflag, mode_t mode, struct mq_attr *attr
```

- Creates (or accesses an existing) message queue identified by a name
- Oflag works similarly to POSIX semaphores (O_CREAT, O_EXCL)
- Can be created in *nonblocking* form (O_NONBLOCK
- Oflag also specifies the *direction* of the message queue (O_RDONLY, O_WRONLY, O_RDWR)
- Second form allows you to set advanced options (access restrictions, size of queue, size of message)
- Like semaphores, MQs exist until they are *unlinked* with mq_unlink

POSIX Message Queues: Sending

```
int mq_send(mqd_t mqdes, const char *msg_ptr,
            size_t msg_len, unsigned msg_prio);
int mq_timedsend(mqd_t mqdes, const char *msg_ptr,
            size_t msg_len, unsigned msg_prio,
            const struct timespec *abs_timeout);
```

- Sends a message (msg_ptr) of a specified length (msg_len) to a message queue (mqdes)
- A priority is specified (msg_prio). Messages are delivered in *priority FIFO* order.
- Both forms will block (if the MQ is not nonblocking) if the queue is full. The timed form accepts an absolute timeout.

POSIX Message Queues: Receiving

```
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
                   size_t msg_len, unsigned *msg_prio);
ssize_t mq_timedreceive(mqd_t mqdes, char *msg_ptr,
                   size_t msg_len, unsigned *msg_prio,
                   const struct timespec *abs_timeout);
```

- Receives a message into a buffer (msg_ptr) or specified length (msg_len) from a specific queue (mqdes).
- Buffer *must be at least* the maximum size of a message.
- Will block (if the MQ is not nonblocking) if the queue is empty.

Shared Memory

- Fastest form of interprocess communication
- Common block of virtual memory shared by multiple processes
- Permission is read-only or read-write for a process
- Mutual exclusion constraints are not part of the shared-memory facility but must be provided by the processes using the shared memory

Typical Sequence of Activity

- Open shared memory segment
  - shm_open system call
- Set the size of the shared memory segment
  - ftruncate system call
- Map the shared memory segment in the virtual address space of the calling process
  - mmap
  - Typically the output of mmap is stored into a pointer
- Use the pointer associated with the memory segment
- Unlink the shared memory segment.

Shared Memory: Initialization

int shm_open(const char *<u>name</u>, int <u>oflag</u>, mode_t <u>mode</u>);

- Creates a new or opens an existing named shared memory segment
- The *oflag* parameter allows the calling process to specify access and creation restrictions:
  - O_RDWR/O_RDONLY: Read/write access or read-only access
  - O_CREAT/O_EXCL: Controls behavior of the segment does/does not exist
  - O_TRUNCATE: If a segment exists, erase its contents
- The *mode* parameter provides access control: it is possible to restrict access to other users/groups
- Returns a *file descriptor* that identifies the shared memory segment

shm_open example

```
// Open the shared memory segment
int shm_fd = shm_open ("/SHM_example",
            O_CREAT | O_EXCL | O_RDWR,
            S_IRUSR | S_IWUSR | S_IRGRP |
S_IROTH);
```

ftruncate

- **#include <unistd.h>**
  #include <sys/types.h>
- **int ftruncate(int** *fd*, **off_t** *length***);**
- Truncates a file pointed by file descriptor fd to the specified length.
- Necessary to set the size of the shared memory.

Ftruncate example

```
if (ftruncate (shm_fd, sizeof (mymessage)) != 0)
{
  perror ("ftruncate");
ret = -1;
 goto exit;
  }
```

Mmap

- **mmap**() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in *addr*. The *length* argument specifies the length of the mapping.
- Used to map the contents of a *file descriptor* into a *memory segment* located in the virtual address space of calling process.

Mmap

- #include <sys/mman.h>
- **void *mmap(void ***_start_**, size_t** _length_**, int** _prot_**, int** _flags_**, int** _fd_**, off_t** _offset_**);**
- Flags
  - **MAP_FIXED** Do not select a different address than the one specified.
  - **MAP_SHARED** Share this mapping with all other processes that map this object.
  - **MAP_PRIVATE** Create a private copy-on-write mapping. Stores to the region do not affect the original file.
  - MAP_PRIVATE and **MAP_SHARED** cannot be used together.

mmap

- #include <sys/mman.h>
- **void *mmap(void ****start*, **size_t** *length*, **int** *prot*, **int** *flags*, **int** *fd*, **off_t** *offset***);**
- prot
  - **PROT_EXEC** Pages may be executed.
  - **PROT_READ** Pages may be read.
  - **PROT_WRITE** Pages may be written.
  - **PROT_NONE** Pages may not be accessed.
- Fd: file to map starting at the specified offset.

mmap

- #include <sys/mman.h>
- **void *mmap(void ****start*, **size_t** *length*, **int** *prot*, **int** *flags*, **int** *fd*, **off_t** *offset***);**
- If start is 0, it returns the best location where the memory segment map has been created.
  - Do not pass MAP_FIXED in that case
- Return MAP_FAILED on error.
  - Exact error code is available with errno
- Example
  - shm_segment =(mymessage *) mmap (0, sizeof (mymessage), (PROT_READ | PROT_WRITE),
      MAP_SHARED, shm_fd, 0))

unlink

- Similar to pipes/semaphores/message queues
- Deletes the shared memory segment.