# CS415 Module 5 Part B - IPC, Shared Memory, and Named Pipes

Athens State University

**Outline**

## Contents



## 1 Shared Memory

### System V vs. POSIX Shared Memory

- There are two different shared memory interfaces provided in Linux
  - The older UNIX System V interface
  - The more recent POSIX standard interface
- There are reasons why one might use one or the other interface
  - We will start with the System V interface

### The Six Steps Required to Use SysV Shared Memory

- Call the `shmget()` system call create a new segment or connect to an existing segment
- Use `shmat()` to *attach* the shared memory segment to a memory segment in the calling process's address space
- Use the memory location as if it was in the local address space
- Call the `shmdt()` system call to detach the shared memory from the process's address space
- Call `shmctl()` to delete the shared memory segment. The OS will not actually destroy the segment until all users are done with it

### The Critical System Calls for SysV Shared Memory

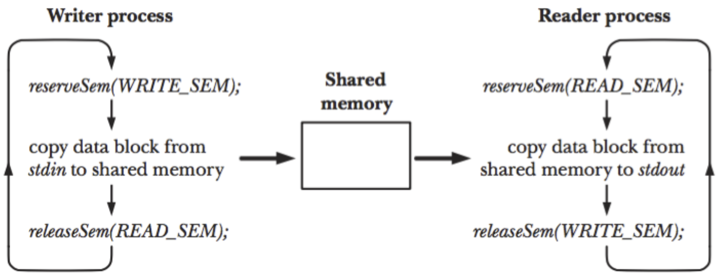
```
#include <sys/types.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmfig);

void *shmat(int shmid, void *shmaddr, int shmfig);

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

### Shared memory and synchronization




- Reading and writing shared memory is typically a critical section
- Thus, we need to use sync. primitives to control access

## 1.1 Case Study: Data Transfer Between Processes

### Case Study: Data Transfer Between Processes

- Two applications: a *writer* process and a `reader` process
- The writer process reads data from the keyboard (standard input) and places into the shared memory segment
- And the reader process reads the data from the shared memory segment and dumps it to the display (standard output)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ     27
int main(int argc, char *argv[])
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;
    // We'll name our shared memory segment
    // "5678".
    key = 5678;
    // Create the segment.
```

```c
    if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
        exit(1);
    }
    // Now we attach the segment to our data space.
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        exit(1);
    }
    //
    s = shm;
    for (c = 'a'; c <= 'z'; c++)
        *s++ = c;
    *s = NULL;
    //
    // Finally, we wait until the other process
    // changes the first character of our memory
    // to '*', indicating that it has read what
    // we put there.
    while (*shm != '*')   sleep(1);
    exit(0);
}
```

**Shared Memory Example: Reader**

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ     27
int main(int argc, char *argv[]) {
    int shmid;
    key_t key;
    char *shm, *s;
    // We need to get the segment named
    // "5678", created by the server.
    key = 5678;
    // Locate the segment.
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        exit(1);
    }
    // Now we attach the segment to our data space.
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        exit(1);
    }
    // Now read what the server put in the memory.
    for (s = shm; *s != NULL; s++)
        putchar(*s);
    putchar('\n');
    // Finally, change the first character of the
    // segment to '*', indicating we have read
    // the segment.
    *shm = '*';
    exit(0);
}
```

## 1.2 POSIX Shared Memory

**System V Shared Memory Design Issues**

- The SysV shared memory model uses keys and identifies to manage shared memory segments
  - This is inconsistent with the standard UNIX I/O model that uses filenames and file descriptors
- Use of shared file mappings for IPC requires the use of a disk file, even if the application doesn't require persistent backing store

### The POSIX Shared Memory Model

- The POSIX model is a variation of "memory mapping" model
  - Pages of main memory are mapped into a virtual memory space
  - One may map a file (or other things) into this space and access it as if it was memory

### The `mmap()` System Call

```c
#include <sys/mman.h>

void *mmap(void *addr,
             size_t length, int prot,
             int flags, int fd, off_t offset);
```

- The `addr` parameter indicates where in the virtual address space the mapped page should be place.
  - If `NULL`, the OS chooses a suitable address in the memory map, this is preferred for portability issues

### Example: Using memory mapping with files

```c
#include <sys/mman.h>
#include <fcntrl.h>
#define MEMSIZE 10
int main(int argc, char *argv[]) {
  char *addr;
  int fd;
  fd = open(argv[1], OW_RDWR);
  if (fd == -1) exit(-1);
  addr = mmap(NULL,
                  MEM_SIZE,
                  PROT_READ | PROT_WRITE,
                  MAP_SHARED,
                  fd,
                  0);
  if (addr == MAP_FAILED) exit(-1);
  if (close(fd) == -1) /* No longer need 'fd' */
      exit(-1);
  printf("Current string=%.*s\n", MEM_SIZE, addr);
  /* Secure practice: output at most MEM_SIZE bytes */
  memset(addr, 0, MEM_SIZE); /* Zero out region */
  strncpy(addr, argv[2], MEM_SIZE - 1);
  if (msync(addr, MEM_SIZE, MS_SYNC) == -1)
       exit(-1);
   printf("Copied \"%s\" to shared memory\n", argv[2]); }
  exit(EXIT_SUCCESS);
}
```

### Case Study: Message capture to log - Server

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <semaphore.h>
#include <sys/mman.h>
// Buffer data structures
#define MAX_BUFFERS 10
#define LOGFILE "/tmp/example.log"
#define SEM_MUTEX_NAME "/sem-mutex"
#define SEM_BUFFER_COUNT_NAME "/sem-buffer-count"
#define SEM_SPOOL_SIGNAL_NAME "/sem-spool-signal"
#define SHARED_MEM_NAME "/posix-shared-mem-example"

struct shared_memory {
    char buf [MAX_BUFFERS] [256];
    int buffer_index;
    int buffer_print_index;
};

void error (char *msg);
// Print system error and exit
void error (char *msg)
{
    perror (msg);
    exit (1);
}
```

### Case Study: Message capture to log - Server

```
int main (int argc, char **argv)
{
    struct shared_memory *shared_mem_ptr;
    sem_t *mutex_sem, *buffer_count_sem, *spool_signal_sem;
    int fd_shm, fd_log;
    char mybuf [256];
    // Open log file
    if ((fd_log = open (LOGFILE, O_CREAT | O_WRONLY | O_APPEND | O_SYNC, 0666)) == -1)
        error ("fopen");
    //  mutual exclusion semaphore, mutex_sem with an initial value 0.
    mutex_sem = sem_open (SEM_MUTEX_NAME, O_CREAT, 0660, 0);
    if (mutex_sem == SEM_FAILED)
        error ("sem_open");
    // Get shared memory
    fd_shm = shm_open (SHARED_MEM_NAME, O_RDWR | O_CREAT, 0660)
    if (fd_shm == -1)
        error ("shm_open");
    ftruncate (fd_shm, sizeof (struct shared_memory)
    if (fd_shm == -1)
       error ("ftruncate");
```

### Case Study: Message capture to log - Server

```
    // Continuing from previous slide..
    shared_mem_ptr = mmap (NULL, sizeof (struct shared_memory), PROT_READ | PROT_WRITE,
        MAP_SHARED,
            fd_shm, 0);
    if (shared_mem_ptr == MAP_FAILED)
       error ("mmap");
    // Initialize the shared memory
    shared_mem_ptr -> buffer_index = shared_mem_ptr -> buffer_print_index = NULL;

    // counting semaphore, indicating the number of available
    // buffers. Initial value = MAX_BUFFERS
    buffer_count_sem = sem_open (SEM_BUFFER_COUNT_NAME, O_CREAT, 0660, MAX_BUFFERS);
    if (buffer_count_sem == SEM_FAILED)
        error ("sem_open");
    // counting semaphore, indicating the number of strings to be
    // printed. Initial value = 0
    spool_signal_sem = sem_open (SEM_SPOOL_SIGNAL_NAME, O_CREAT, 0660, 0);
    if (spool_signal_sem == SEM_FAILED)
        error ("sem_open");
```

### Case Study: Message capture to log - Server

```
    // Initialization complete; now we can set mutex semaphore as 1 to
    // indicate shared memory segment is available
    if (sem_post (mutex_sem) == -1)
        error ("sem_post: mutex_sem");

    while (true) {  // forever
        // Is there a string to print? P (spool_signal_sem);
        if (sem_wait (spool_signal_sem) == -1)
            error ("sem_wait: spool_signal_sem");

        strcpy(mybuf, shared_mem_ptr -> buf [shared_mem_ptr -> buffer_print_index]);

        /* Since there is only one process (the logger) using the
           buffer_print_index, mutex semaphore is not necessary */
        (shared_mem_ptr -> buffer_print_index)++;
        if (shared_mem_ptr -> buffer_print_index == MAX_BUFFERS)
           shared_mem_ptr -> buffer_print_index = 0;

        /* Contents of one buffer has been printed.
           One more buffer is available for use by producers.
           Release buffer: V (buffer_count_sem);  */
        if (sem_post(buffer_count_sem) == -1)
            error ("sem_post: buffer_count_sem");

        // write the string to file
        if (write (fd_log, mybuf, strlen (mybuf)) != strlen (mybuf))
            error ("write: logfile");
    }
}
```

### Case Study: Message capture to log - Client

```
int main (int argc, char **argv)
{
    struct shared_memory *shared_mem_ptr;
    sem_t *mutex_sem, *buffer_count_sem, *spool_signal_sem;
```

```
int fd_shm;
char mybuf [256];
struct shared_memory *shared_mem_ptr;
sem_t *mutex_sem, *buffer_count_sem, *spool_signal_sem;
int fd_shm, fd_log;
char mybuf [256];
//  mutual exclusion semaphore, mutex_sem
mutex_sem = sem_open (SEM_MUTEX_NAME, 0, 0, 0);
if (mutex_sem == SEM_FAILED)
    error ("sem_open");
// Get shared memory
fd_shm = shm_open (SHARED_MEM_NAME, O_RDWR, 0);
if (fd_shm == -1)
    error ("shm_open");
shared_mem_ptr = mmap (NULL,
                       sizeof (struct shared_memory),
                       PROT_READ | PROT_WRITE, MAP_SHARED,
                       fd_shm,
                       0);
if (shared_mem_ptr == MAP_FAILED)
   error ("mmap");
```

**Case Study: Message capture to log - Client**

```
// counting semaphore, indicating the number of available buffers.
buffer_count_sem = sem_open (SEM_BUFFER_COUNT_NAME, 0, 0, 0);
if (buffer_count_sem == SEM_FAILED)
    error ("sem_open");
// counting semaphore, indicating the number of strings to be
// printed. Initial value = 0
spool_signal_sem = sem_open (SEM_SPOOL_SIGNAL_NAME, 0, 0, 0);
if (spool_signal_sem == SEM_FAILED)
    error ("sem_open");
```

**Case Study: Message capture to log - Client**

```
char buf [200], *cp;
printf ("Please type a message: ");
while (fgets (buf, 198, stdin)) {
    // remove newline from string
    int length = strlen (buf);
    if (buf [length - 1] == '\n')
       buf [length - 1] = '\0';
    // get a buffer: P (buffer_count_sem);
    if (sem_wait (buffer_count_sem) == -1)
        error ("sem_wait: buffer_count_sem");
    /* There might be multiple producers. We must ensure that
        only one producer uses buffer_index at a time.  */
    // P (mutex_sem);
    if (sem_wait (mutex_sem) == -1)
        error ("sem_wait: mutex_sem");
```

**Case Study: Message capture to log - Client**

```
        // Critical section
        time_t now = time (NULL);
        cp = ctime (&now);
        int len = strlen (cp);
        if (*(cp + len -1) == '\n')
                *(cp + len -1) = '\0';
        sprintf (shared_mem_ptr -> buf [shared_mem_ptr -> buffer_index],
                 "%d: %s %s\n",
                 getpid (),
                 cp, buf);
        (shared_mem_ptr -> buffer_index)++;
        if (shared_mem_ptr -> buffer_index == MAX_BUFFERS)
                shared_mem_ptr -> buffer_index = 0;
        // Release mutex sem: V (mutex_sem)
        if (sem_post (mutex_sem) == -1)
            error ("sem_post: mutex_sem");
  // Tell spooler that there is a string to print: V (spool_signal_sem);
        if (sem_post (spool_signal_sem) == -1)
            error ("sem_post: (spool_signal_sem");
        printf ("Please type a message: ");
    }
    if (munmap (shared_mem_ptr, sizeof (struct shared_memory)) == -1)
        error ("munmap");
    exit (0);
}
```

## 2 Pipes, Named and Otherwise

### Pipes?

- Pipes are the oldest method of IPC on UNIX systems, having been defined in Classical UNIX in the early 1970s
- Solution to the question of "How can the shell allow the output produced by one process be used as input to a related process?"
- Named pipes (or FIFOs) are a variation on this idea that allows for communication between *any* processes

### Pipes and the shell

- Command-line shells in UNIX-like OSes allow one to send the output from a command to the input of a second command:

```
ls | wc -l
find . -name "*.doc?" -print0 | xargs grep -i -n -e "test grade*"
```

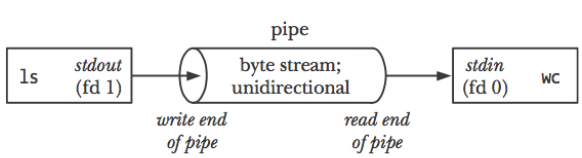



**Figure 44-1:** Using a pipe to connect two processes

### Characteristics of a pipe

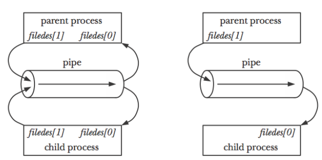
- A pipe is a byte stream
- Reads from an empty pipe will block
- Reads from a closed write-end of a pipe will return EOF
- Pipes are unidirectional
- Writes of sizes up to the pipe capacity are guaranteed to be atomic
- Pipes have a limited capacity and writes will block until a reader pulls data from the pipe

### `pipe()`: Opening a pipe

```c
#include unistd.h>
int pipe(int filedescriptors[2]);
```

- A success call to `pipe()` returns two open file descriptors: [0] for the read-end of the pipe, [1] for the write end.
- Once opened, we treat the pipe as if it was any other type of file

### Pipes, `fork()`, and `exec()`




a) After *fork()* b) After closing unused descriptors

- So, with multiple processes, one of the first things one does is reconfigure the ends of the pipe for the parent and child
- Parent is the writer, and so will close the read-end of the pipe
- Child the reader, and so will close the write-end of the pipe

### Making things work in your code

```c
int filedes[2];
if (pipe(filedes) == -1)
    error("pipe");
switch (fork()) {
case -1:
    error("fork");
case 0:  /* Child */
    if (close(filedes[1]) == -1)
        error("close");
    /* Child now reads from pipe */
    break;
default: /* Parent */
    if (close(filedes[0]) == -1)
        error("close");
    /* Parent now writes to pipe */
    break;
}
```

**Remapping standard input or standard output**

```c
int pfd[2];
pipe(pfd);
// Code from previous slide goes here to fork()
close(STDOUT_FILENO);
dup(pfd[1]);
```

## 2.1 Named Pipes

### A Named Pipe is a Pipe

- Also known as a FIFO (think about it)
- Main difference is that a named pipe has a name within the file system is opened in the same way as a regular file
- Named pipes are created using the `mkfifo` shell command
  - And a system call with the same name can be used to create a FIFO

Consider the following shell script:

```sh
mkfifo myfifo
wc -l < myfifo &
ls -l | tee myfifo | sort -k5n
```