System and Device Programming

Advanced UNIX IPC

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Pipes

- Original pipes (or **unnamed** pipes) are a common form of UNIX System IPC but they
  - Can be used only between two processes that have a **common ancestor**
    - Last only as long as the process last
  - Are **half duplex**
    - Data flows only in one direction
  - Can be seen as an **unstructured** sequential files
    - The communication channel is a FIFO queue
    - Structured data transfers require a communication protocol
  - Are limited to transfer only **limited quantity** of memory

Communication channels

- This section extends pipes in the following directions
  - Pipes extensions
    - FIFOs
    - Message queues
  - Shared memory

To avoid a common ancestor

To allow structured data

To transfer a lot on information

Memory mapped files have been introduced in the filesystem unit

FIFOs

- FIFOs are an extension of traditional pipes and are sometimes called named pipes
  - They allow a communication among **unrelated** processes
  - They can last as long as the system does
    - They can be deleted if no longer used
- A FIFO is a type of file
  - Creating a FIFO is similar to creating a file
    - Indeed, a FIFO corresponds to a file in the local storage
      - They have a **pathname** in the filesystem
    - Once a FIFO has been created, processes can open it and perform R/W operations on it

FIFOs

- Process logic
  - A FIFO special file is entered into the filesystem by calling **mkfifo**
    - **Subsequent** calls to (the "same") mkfifo **have no effect**
  - Once we have created a FIFO special file, any process can open it, using the **open** system call
  - Once a FIFO has been opened, it can be used for reading or writing
    - Use **read** and **write**, as for ordinary files
  - Notice that a FIFO has to be open at **both** ends before you can proceed to do any input or output operations on it

FIFOs

```
#include <sys/stat.h>

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

- Function **mkfifo** creates a FIFO
  - The parameters **path** and **mode** are similar to the corresponding ones specified for function **open**
    - Please refer to **open** for any further explanation on the **mode** parameter
      - Use constant S_I[RWX]USR or an octal representation to specify user and group ownership

FIFOs

- Return value
  - The value 0, on success
  - The value -1, on error
- Once the FIFO is in the system, we can use normal file I/O functions to operate on it
  - Please refer to system calls open, read, write, and close for further details

It creates a FIFO with pathname, e.g.,
prw-rw-r-- 1 quer quer 0 apr 5 16:14 **path**

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

FIFOs

```
#include <sys/stat.h>

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

- Function **mkfifoat** is similar to mkfifo but
  - Parameter **fd** indicates a directory path to use to open the FIFO file
    - If **path** specifies an absolute pathname, **fd** is ignored (and mkfifoat is equivalent to mkfifo)
    - If **path** specifies a relative pathname and **fd** is a descriptor for an open directory, the pathname is evaluated starting from this directory
    - If **path** specifies a relative pathname and **fd** is AT_FDCWD the pathname is evaluated starting from the current working directory

FIFOs

- Caveats
  - As with a pipe, if we write to a FIFO that no process has opened for reading, the signal SIGPIPE is generated
  - When the last writer for a FIFO closes the FIFO, an end of file is generated for the reader of the FIFO
  - It is common to have multiple writers for a given FIFO
    - We have to worry about atomic writes if we don't want the outputs from multiple processes to be interleaved

Example

1 Reader + 1 Writer
(client server communication)

Run this process on **a** shell windows

(W) P_1 ⇒ (R) P_2

The Writer

```
int main() {
  int fd; char str[80];
  char *myfifo = "/tmp/myfifo";
  mkfifo (myfifo, 0666);
  fd = open (myfifo, O_WRONLY);
  while (1) {
    printf ("Send to reader: ");
    fgets (str, 80, stdin);
    write (fd, str, strlen (str)+1);
    if (strncmp (str, "end", 3)==0) {
      break;
    }
  }
  close (fd);
  return 0;
}
```

Read from stdin

Write to FIFO

Stop the process when "end" is introduced

Example

Run this process on **another** shell windows

(W) P_1 ⇒ (R) P_2

The Reader

```
int main() {
  int fd;
  char str[80];
  char *myfifo = "/tmp/myfifo";
  mkfifo (myfifo, 0666);
  fd1 = open (myfifo, O_RDONLY);
  while (1) {
    read (fd, str, 80);
    printf ("Received from writer: %s", str);
    if (strncmp (str, "end", 3)==0) {
      break;
    }
  }
  close(fd);
  return 0;
}
```

Read from the FIFO

Stop the process when "end" is received

Blocking versus Non-blocking

- The open operation on a FIFO can be blocking or non-blocking

```
fd = open (myfifo, ... | O_NONBLOCK);
```

- Without the O_NONBLOCK flag
  - On **open** in read-only (write-only) mode is **blocking** until some other process open the FIFO in write-only (read-only)
- With the O_NONBLOCK flag
  - An open in read-only mode return immediately
  - An open in write-only mode returns -1 (and errno set to ENXIO)

Error Checking

- Many UNIX system functions (such as mkfifo and mkfifoat) returns -1 on error
- Once this happens, the strategy to check the origin of the error is the following one
  - Include header <errno.h>
    - This header defines error codes and error manipulation functions
  - Define the global integer variable **errno**
    - This variable is automatically set to the proper error
  - Use functions **perror** and **strerror** to display an error message

Error Checking

When **mkfifo** generates an error, the following error codes are set in **errno**

```
int mkfifo (path, S_IRWXU | S_IRWXG | S_IRWXO);
```

Error Code	Error Meaning
EACCES	One of the directories in **path** did not allow search/execute permission
EDQUOT	The user's quota on the filesystem has been exhausted
EEXIST	**path** already exists This includes the case where path is a symbolic link, dangling or not
ENAMETOOLONG	Either the total length of **path** is greater than **PATH_MAX**, or an individual filename component has a length greater than **NAME_MAX**
ENOENT	A directory component in **path** does not exist or is a dangling symbolic link
ENOSPC	The directory or filesystem has no room for the new file
ENOTDIR	A component used as a directory in **path** is not, in fact, a directory
EROFS	**path** refers to a read-only filesystem

Example

To the Reader and the Writer analyzed before **add**

Define the **errno** header

Define the **errno** variable (automatically set by the system in case of an error, for many system calls)

Grab error code (-1) from mkfifo

Manually/Explicitly check for error

Use **perror** to display error condition

```
#include <errno.h>

extern int errno;

int ret;
ret = mkfifo (myfifo, 0666);

if (errno == EEXIST)
  fprintf (stderr, "FIFO exists.\n");


sprintf (str, "Reader (return value=%d)", ret);
perror (str);
```

Function **perror** displays the string str followed by ":", an error message, and "\n"

Example

- Possibile error messages

```
sprintf (str, "... (return value=%d)", ret);
perror (str);
```

Writer (return value=0): Success

Reader (return value=-1): File exists

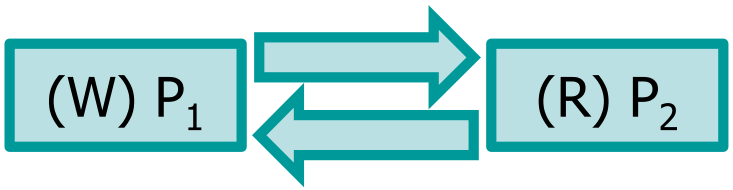
- FIFO may be eventually removed

```
sprintf (str, "rm -rf %s", myfifo);
system (str);
```

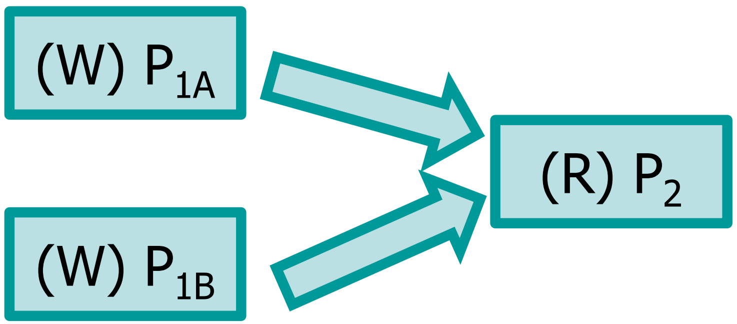
To remove "/tmp/myfifo", i.e.,
prw-rw-r-- 1 quer quer 0 apr 5 16:14 myfifo

Example: Extensions

- Alternate Write and Read operations
  - Use same FIFO in both directions



- Coordinate more Writers with one Reader
  - Use more FIFOs

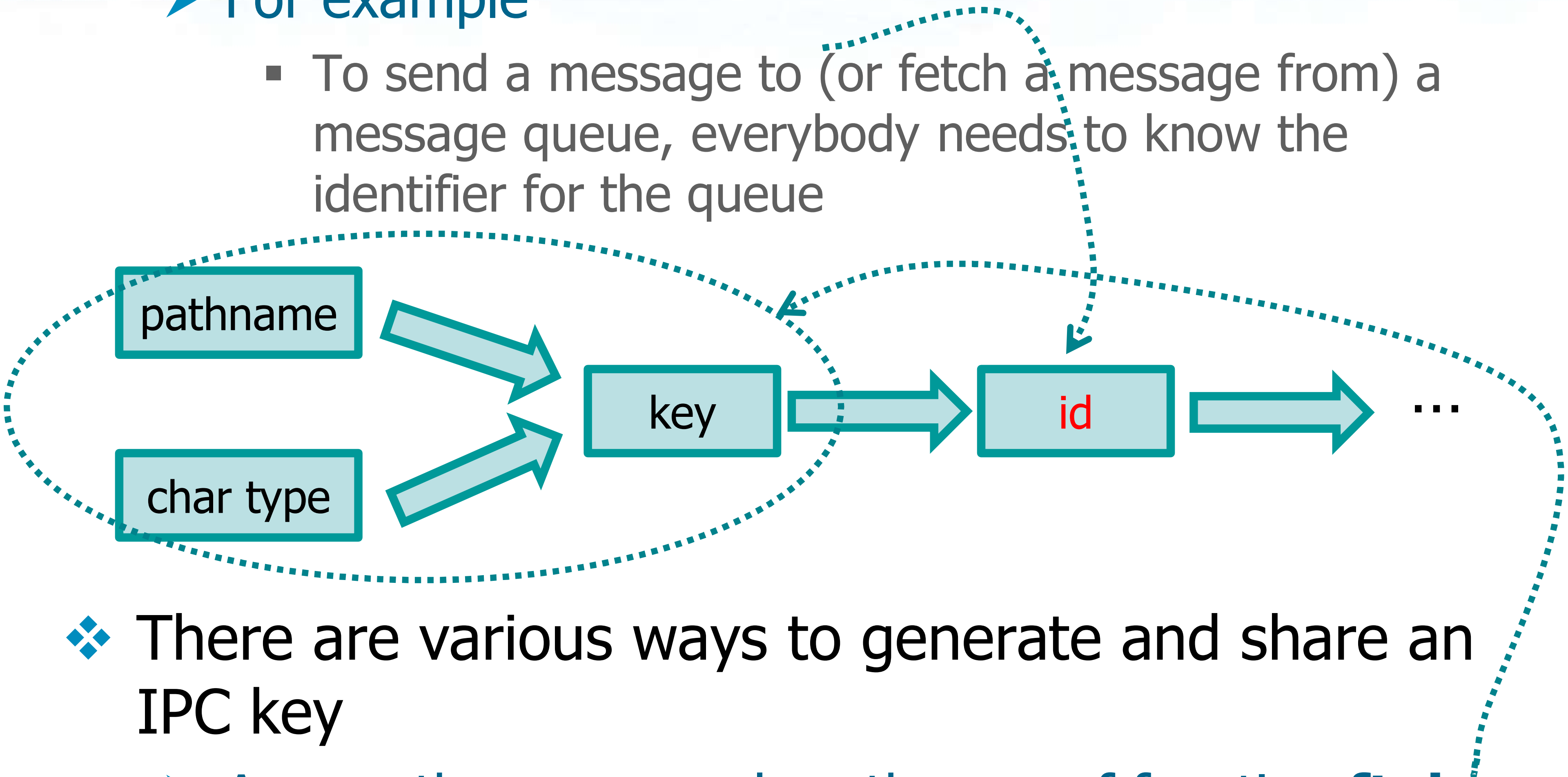


Identifiers and keys

- Message queues and shared memories are related to IPC structures
  - Whenever an IPC structure is being created a key must be specified
    - The data type of this key is the primitive system datatype **key_t**
    - **key_t** is often defined as a long integer in the header <sys/types.h>
  - The kernel converts this key into an an identifier
    - Each IPC structure is referred in the kernel by a non-negative integer identifier
    - This identifier is like an internal reference to the object

Identifiers and keys

- For example
  - To send a message to (or fetch a message from) a message queue, everybody needs to know the identifier for the queue



- There are various ways to generate and share an IPC key
  - Among them, we analyze the use of function **ftok**

Identifiers and keys

```
#include <sys/msg.h>

key_t ftok (const char *path, int id);
```

- To share a key, the different processes (e.g., the clients and the server) must agree on a
  - Standard (file) pathname
    - The file **must** exist
  - Project ID
    - A character value between 0 and 255
- Function **ftok** converts these two parameters into a key (of type **key_t**)

Identifiers and keys

- This key is then used during the communication phase
  - Notice that the only service provided by **ftok** is a way of generating a key from a pathname and a project ID
  - Function **ftok** does not perform any sort of communication
- Return value
  - Message key, if success
  - The value −1, on error

```
key_t ftok(const char *path, int id);
```

Message queues

- FIFOs are used to pass streams of anonymous bytes
  - Applications using FIFOS have to manage their own data chunking
    - The have to agree on data delimiters, such as end-of-field, end-of-record, etc.
- To pass structured data chunks it is necessary to use message queues
  - A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier

Message queues

What is a queue?

- A message queue
  - Is created or an existing queue opened by **msgget**
  - The queue may be controlled using **msgctl**
  - New messages are added to the end of a queue by **msgsnd**
  - Messages are fetched from a queue by **msgrcv**
    - Messages do not have to be fetched in a first-in, first-out order
    - Messages can be fetched based on their type field

All right ...
A message queue manipulates messages.
But what is a message?

Message queues

What is a message?

- Each message manipulated by msgsnd
  - Is composed of
    - A positive long integer type field
    - A non-negative length (N_BYTES)
    - The actual data bytes (mtext) of size nbytes
  - Has to be defined by the user as a C data structure including
    - The message type mtype
    - The data field mtext of size N_BYTES

```
struct mymesg {
  long int mtype;
  char mtext[N_BYTES];
};
```

  - Messages are always placed at the end of the queue

Message get

```
#include <sys/msg.h>

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

- Function **msgget** either open an existing queue or create a new queue
  - Key is the values generated with **ftok**
  - Flag is used to define the mode permission field to a data structure associated to the message queue
- Return value
  - Message queue identifier (**msqid**), if success
  - The value −1, on error

Message control

```
#include <sys/msg.h>

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

- Function **msgctl** performs various operations on a queue
  - The queue is specified by its identifier (msqid)
    - The parameter msqid is the value returned by msgget

Message control

- The cmd argument specifies the command to be performed on the queue
  - IPC_STAT
    - Fetch the msqid_ds structure for this queue, storing it in the structure pointed to by buf
  - IPC_SET
    - Copy the following fields from the structure pointed to by buf to the msqid_ds structure associated with this queue: msg_perm.uid, msg_perm.gid, msg_perm.mode, and msg_qbytes

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

Message control

- IPC_RMID
  - Remove the message queue from the system and any data still on the queue
  - The removal is immediate
  - Any other process still using the message queue will get an error of EIDRM on its next attempted operation on the queue
- These three constants are also provided for semaphores and shared memory

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

Message send

```
#include <sys/msg.h>

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

- Data is placed onto a message queue by calling **msgsnd**
  - The identifier **msqid** specifies the queue on which to send a message
  - The **ptr** argument points to the specific uder-defined message data structure **mymsg**
  - **nbytes** specify the size of the data array in **mymsg**

Message send

- The **flag** value is 0 or IPC_NOWAIT
  - If the message queue is full and we specify IPC_NOWAIT, then msgsnd returns with the error **EAGAIN**

- Return value
  - The value 0, if success
  - The value –1, on error

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

Message receive

```
#include <sys/msg.h>

ssize_t msgrcv (
  int msqid, void *ptr, size_t nbytes, long type,
  int flag
);
```

- Messages are retrieved from a queue by **msgrcv**
  - Parameters follow the same logic described for msgsnd
  - As with msgsnd, the ptr argument points to the user-defined message structure **mymsg**
    - If the returned message is larger than nbytes and the MSG_NOERROR bit in flag is set, the message is truncated

Message receive

- The type argument lets us specify which message we want
  - type=0: The first message on the queue is returned
  - type>0: The first message on the queue whose message type equals type is returned
  - type<0: The first message on the queue whose message type is the lowest value less than or equal to the absolute value of type is returned

A message queue can implement several FIFOs

- Return value
  - Size of data portion of message, if success
  - The value, −1 on error

```
ssize_t msgrcv (int msqid, void *ptr,
  size_t nbytes, long type, int flag);
```

Example

The Writer

1 Reader + 1 Writer

(W) P_1 ⇒ (R) P_2

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define L 512

struct mesg_buffer {
  long mesg_type;
  char mesg_text[L];
} message;
```

Example

The Writer

(W) P_1 → (R) P_2

```
int main() {
  key_t key;
  int msgid;

  key = ftok ("progfile", 65);
  msgid = msgget (key, 0666 | IPC_CREAT);
  message.mesg_type = 1;

  printf ("Read data: ");
  fgets (message.mesg_text, L, stdin);

  msgsnd (msgid, &message, L*sizeof(char), 0);
  printf ("Data send: %s\n", message.mesg_text);
  return 0;
}
```

Get key

Get the id

Read message from stdin

Send message

Example


```c
struct mesg_buffer {
  long mesg_type;
  char mesg_text[L];
} message;

int main() {
  key_t key;
  int msgid;
  key = ftok ("progfile", 65);
  msgid = msgget (key, 0666 | IPC_CREAT);
  msgrcv (msgid, &message, L*sizeof(char), 1, 0);
  printf ("Data received: %s\n", message.mesg_text);
  msgctl (msgid, IPC_RMID, NULL);
  return 0;
}
```

Get key

Get the id

Receive message

Remove the queue from the system

Shared Memory

- Shared memory allows two or more processes to share a given region of memory
  - It is the fastest form of IPC, because the data does not need to be copied between the client and the server
    - With pipes and message queues, the information has to go through the kernel
    - With shared memory, all processes can access the common memory and changes made by one process can be viewed by another process

Shared Memory

- Shared memory requires synchronization accesses to a given region among multiple processes
  - If the server is placing data into a shared memory region, the client shouldn't try to access the data until the server is done
  - Often, semaphores are used to synchronize shared memory access

Shared Memory

- Process logic
  - **ftok** is use to generate a unique key to manage the entire process
  - **shmget** returns an identifier for the shared memory segment
  - **shmat** attach the user to the shared memory segment
  - **shmdt** detach the process with with the shared memory segment at the end of the sharing phase
  - **shmctl** destroy the shared memory buffer once the process has been detached

Shared memory get

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

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

- Function shmget is used to obtain a shared memory identifier given the key of the IPC object
  - The parameter **size** is the size of the shared memory segment in bytes
  - The parameter **flag** set the mode field of the IPC structure
    - See te example for further details

Shared memory get

- Return value
  - Shared memory ID, if success
  - The value −1, on error

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

Shared memory control

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

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

- Function **shmctl** performs various operations on a shared memory
  - The queue is specified by its identifier (msqid)
- the **cmd** argument specifies a command to be performed on the segment
  - As with function **msgctl**, it is possibile to specify
    - PC_STAT
    - IPC_SET
    - IPC_RMID

Shared memory control

- Or, when the process is running in super-user mode
  - SHM_LOCK to lock the shared memory segment in memory
  - SHM_UNLOCK to unlock the shared memory segment

- Return value
  - 0, if OK
  - −1, on error

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

Shared memory attach

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

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

- Once a shared memory segment has been created, a process attaches it to its address space by calling shmat
- The address in the calling process at which the segment is attached depends on
  - The **addr** argument
  - Whether the SHM_RND bit is specified in **flag** argument

Shared memory attach

Most common case

- If **addr** is 0, the segment is attached at the first available address selected by the kernel
- If **addr** is nonzero and SHM_RND is not specified, the segment is attached at the address given by addr
- If **addr** is nonzero and SHM_RND is specified, the segment is attached at the address given by (addr – (addr modulus SHMLBA))

❖ Return value

- ➢ Pointer to the shared memory segment, if success
- ➢ The value –1, on error

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

Shared memory delete

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

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

- When we are done with a shared memory segment, we call shmdt to detach it
  - Note that this does not remove the identifier and its associated data structure from the system
  - The identifier must be removed by calling shmctl with a command of IPC_RMID
- Return value
  - The value 0, if success
  - The value –1, on error

Example

1 Reader + 1 Writer

(W) P_1 $\Rightarrow$ (R) P_2

The same process works as
- a reader (no parameter)
- a writer (writing the parameter on the shared memory)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024


int main (int argc, char *argv[]) {
  key_t key;
  int shmid;
  char *data;
```

Make it a 1K shared memory segment

Example

Make the key

Here the file must exist

Create the segment

```
if ((key = ftok ("hello.txt", 5)) == -1) {
  perror ("ftok");
  exit (1);
}
if ((shmid = shmget (key, SHM_SIZE,
             0644 | IPC_CREAT)) == -1) {
  perror ("shmget");
  exit (1);
}
data = shmat (shmid, NULL, 0);
if (data == (char *)(-1)) {
  perror ("shmat");
  exit (1);
}
```

Attach the segment to the local pointer **data**

Example

Modify the segment, based on the command line

Read the segment

Detach from the segment

```
  if (argc == 2) {
    printf ("Writing to segment: \"%s\"\n", argv[1]);
    strncpy (data, argv[1], SHM_SIZE);
  }
  else
  {
    printf("segment contains: \"%s\"\n", data);
  }
  if (shmdt(data) == -1) {
    perror ("shmdt");
    exit (1);
  }
  return 0;
}
```