CS240 Operating Systems, Communications and Concurrency – Dermot Kelly
Practical 5 - Unix Interprocess Communication Mechanisms
Named Pipes and Message Queues

The pipe communication mechanism we have seen in the last practical is a means by which *related* processes can communicate, that is, processes within the same creation hierarchy all having a common parent. As each child in the hierarchy receives a copy of its parent's descriptor table when it is forked, it is possible for it to access any pipes created by its hierarchical ancestors. Pipes can be implemented as memory-based circular buffers by the operating system. These pipes are known as unnamed or anonymous pipes and are transient. An unnamed pipe ceases to exist when all processes having a descriptor for it terminate or close their descriptors.

Named Pipes

A **named pipe** is similar in behaviour to an unnamed pipe except that it has a name in the file system like a regular file. Therefore it is a persistent communication channel and remains permanently, even when not in use, until explicitly removed.

Processes open **named pipes** in the same way as regular files, so **unrelated** processes can communicate by opening the pipe and either reading or writing it. A process can open a pipe in read or write mode. A process opening the pipe for reading will block if there is no process that has it open for writing. Similarly, a process opening the pipe for writing will block if there is no process that has it open for reading.

Multiple readers and writers must coordinate their activity by some means if meaningful message exchange is to take place.

The first thing we need to do is to create a named pipe. A named pipe can be created by our program as follows:-

```
mknod("my_pipe", 010600, 0);
/* Create a file called "my_pipe" in current directory
   using mknod. The file type is a fifo special pipe
   indicated by the value 010.
   Access permissions to the pipe allow read, write
   for the owner, but none for group and world with
   value 600 (110000000 in binary).
   The last parameter to mknod is only relevant to
   block or char special files and is set to 0 here*/
```

If the `mknod` system call is successful in creating a named pipe, it will return a code of 0, and -1 if it is not successful. What if a pipe of that name already exists for example? You can check if a file exists in the filesystem using the `access()` system call.

`access(path, mode)` checks whether the calling process can access the specified file pathname using the given mode of access. It returns 0 if it can, -1 if it can't. The mode flag F_OK tests for the existence of the file only. The other flags R_OK, W_OK, and X_OK test whether the file exists and allows read, write, and execute permissions, respectively. All the flags can be ORed together into a single mode expression if required.

So the following piece of code ensures that our named pipe "my_pipe" will exist before we use it in our program for communicating. First we check to see if it exists and if not, then the code below creates it.

```
/* Check for existence of pipe and create it if it
doesn't exist */
if (access ("my_pipe", F_OK) == -1) /*doesn't exist */
  if (mknod("my_pipe", 010600, 0) == 0)
     printf("Named Pipe created successfully \n");
  else {
     printf("Failed to create Named Pipe \n");
     exit(0); /* Program terminates */
  }
else
   printf("Using existing named pipe  \n");
```

We are going to write a program that demonstrates the use of named pipes. We are going to use the same program to write to the pipe or to read from the pipe depending on what parameter is supplied on the command line when the program is executed.

If our finished program is called `namedpipe.c` , and we compile it using
`cc namedpipe.c –o namedpipe`
then to read from the pipe we will execute
`./namedpipe  reader &`
and to write to the pipe we will execute
`./namedpipe  writer &`

So you see before our program uses the pipe for any reading or writing, we are going to have to check that the correct parameters were supplied on the command line and if so, decide on whether the program should read or write to the pipe.

The declaration of the `main()` function in a C program contains parameters which may be used to access any command line arguments passed to the program when it is executed.

```
int main(int argc, char *argv[])
```

The first parameter represents the number of arguments, and the second parameter is an array of strings, with each command line parameter stored as a separate string in the array.
The first parameter `argv[0]` by convention is the string for the command name you executed (the name of your executable program) and so without any additional parameters on the command line the value of `argc` would usually be equal to 1 by default. We would like our program to accept one additional command line argument having the value of either "`reader`" or "`writer`" depending on what we would like our program to do.

So when the program runs, first we must check that the value of `argc` is precisely 2, and then look at the argument in `argv[1]` and make sure it is equal to either "`reader`" or "`writer`". If any of this is not the case, the program should terminate telling the user that they did not run it correctly. Sample code for achieving this is given below.

```
   /* Check command line parameter count, it should be  2.
      Check 2nd parameter is either "reader" or "writer" and open
      the pipe accordingly */
   if (argc == 2) {
      if (strcmp (argv[1], "reader")==0)
         namedpipe = open("my_pipe", O_RDONLY);
      else if (strcmp(argv[1], "writer")==0)
         namedpipe = open("my_pipe", O_WRONLY);
      else
         paramerror = true;
      }
   else
      paramerror = true;

   if (paramerror)
      printf("Incorrect usage: Use namedpipe <reader | writer>
\n");
   else if (namedpipe<0)
      printf("Couldn't open named pipe\n");
```

Ok, so if we got this far, we have now successfully created and opened a named pipe called "`my_pipe`" in the desired mode of access. So now all that remains is to either write to it or read from it depending on the value of the command line parameter in `argv[1]`.

Let's say that if we want the program to act as a writer it will write 10 messages of arbitrary length into the pipe and end. And if it is a reader, let's say it will try to read 10 times from the pipe into a buffer reading a desired maximum number of characters each time and printing them out for us to see.

The program should write data to the pipe in the form "n nessage text" where n is the number of the nth message out of ten.

The reader is free to read as many characters as it wants from the pipe, but only executes ten read operations in its loop before terminating.

Sample code for this behaviour is given overleaf:-

```
{ /* If we get as far as here, the pipe is open and ready */
  /* Now do reader or writer actions next */
   int i;
   if (strcmp(argv[1], "writer")==0)
      /*Writer outputs 10 messages to pipe */
      for (i=1; i<=10; i++) {
         /* convert i to string as part of message */
         sprintf(message, "%d", i);
         strcat(message, " message text");
         write(namedpipe, message, strlen(message));
         printf("writer: Sent message <%s> to named
pipe\n",message);
         }
   else for (i=1; i<=10; i++) {
      /* Reader will read 10 times from pipe */
      int n = read(namedpipe, buffer, max_size);
      buffer[n]=0;
      printf("reader: Read message <%s> %d characters \n",buffer,
n);
   }
   close(namedpipe);
}
```

OK, so now we are ready to put the whole program together. Here is a template for it.

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <sys/stat.h>

int main(int argc, char *argv[]) {
   int namedpipe;
   int max_size = 20;
   char message[max_size+1];
   char buffer[max_size+1];
   bool paramerror= false;

/* Check for existence of named pipe and create it if it
   doesn't exist */

/* Check command line parameter count, it should be  2.
   Check 2nd parameter is either "reader" or "writer" and open
   the pipe accordingly */

/* If there are no command line errors or pipe opening errors then
   do the reader writer actions */

}
```

Exercise: Complete the program on the previous page and save the program as `namedpipe.c`
Compile using **`cc namedpipe.c -o namedpipe`**
To test communication using the named pipe run the program twice as two separate processes as follows:-

```
./namedpipe writer &
./namedpipe reader &
```

You notice the writer blocks until the reader opens the pipe for reading and then all the output comes at once. Notice that the writer will create the pipe, but the reader will discover that it exists already and use the existing pipe.

When both processes are complete just run the writer on its own as follows:-
`./namedpipe writer &`

The writer blocks because there is no reader.
We are going to use the `gzip` compression utility to compress the output from the writer and put it in a compressed file.

So while the writer is running and blocked above enter the following at the shell prompt:

```
gzip <my_pipe >pipeout.gz
```

`gzip` will act as a reader and use `my_pipe` as its input and put its output into `pipeout.gz` (a compressed file). See how two unrelated processes can use a named pipe.

Print the compressed file to the screen using:
`cat pipeout.gz`

Notice that it is not a text file, it is a binary encoded file, and so not all byte codes in it are printable.

Now uncompress the file using:

```
gunzip pipeout.gz
```

This produces the uncompressed file called `pipeout` in your directory.

Now print this file to the screen using
`cat pipeout`

You should see the ten messages of text data that your original writer process wrote to `my_pipe.`

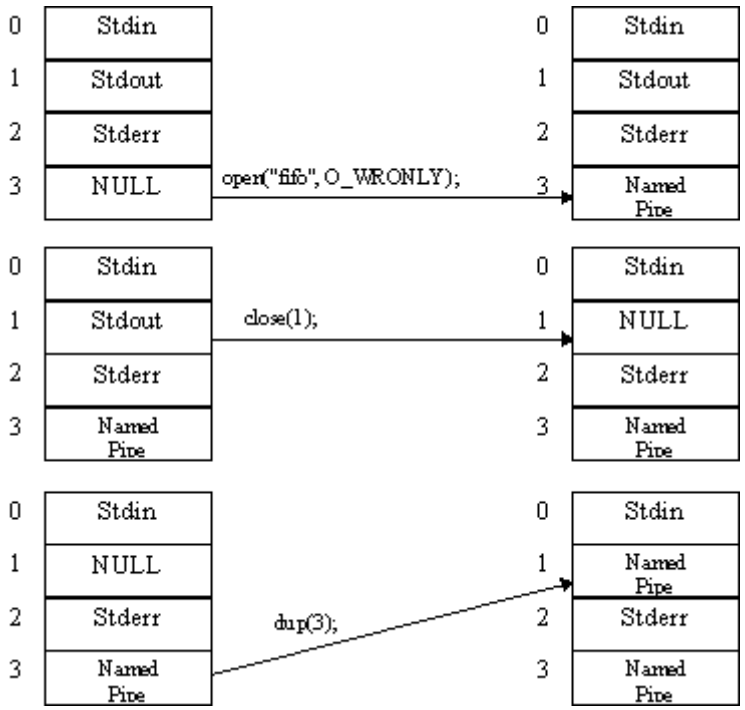
Managing Descriptor Table Entries

The dup system call copies a specified file descriptor into the first free slot of the process's descriptor table and returns the index value of that slot. The format of dup is as follows:-

```
newfd = dup(fd);
```

The diagram below demonstrates the operation of dup where a process first opens a pipe called "fifo" in write only mode, it then closes its standard output device and redirects its standard output to the named pipe called "`fifo`".

The three diagrams on the left indicate the state of the descriptor table before the system call indicated on the transition arrow. The corresponding diagrams on the right indicate the state of the descriptor table after each system call.



Any subsequent write operations to device 1 by this process will now go to the named pipe instead of going to the terminal.

Exercise

Write a program which prints your name to standard output using `printf`. Modify this program so that before the output is displayed, a named pipe called "fifo" is created (using mknod as seen earlier) and the standard output device is redirected to point to the named pipe **using the three statement program sequence shown in the picture above** before your program reaches its `printf`.

Write a separate reader program which can open this named pipe in RD_ONLY mode, then reads your name and displays this on the terminal window. Compile and run both of these programs together as background processes (using &) on the command line.

Message Queues

Message queues allow processes to exchange data in the form of whole messages asynchronously. Communicating parties do not have to be active at the same time. Messages are sent to the queue and subsequently requested from the queue by other processes. The operating system maintains the message queue until it is unlinked by a user process. Messages have an associated priority and are queued and delivered in priority order.

The main functions in the message queue API are the following:

`mq_open()`	function creates a new message queue or opens an existing queue, returning a message queue descriptor for use in later calls.
`mq_send()`	function writes a message to a queue.
`mq_receive()`	function reads a message from a queue.
`mq_close()`	function closes a message queue that the process previously opened.
`mq_unlink()`	function removes a message queue name and marks the queue for deletion when all processes have closed it.

Each message queue has an associated set of attributes, which are set when it is created. The set of attributes can be queried using:- `mq_getattr()`

The message queue attribute structure has four fields defined as follows:-

```
struct mq_attr {
     long mq_flags;  /*flags 0 or O_NONBLOCK*/
     long mq_maxmsg; /*Max number of messages on queue*/
     long mq_msgsize; /*Max message size (in bytes)*/
     long mq_curmsgs; /*Num messages currently in queue*/
};
```

The use of message queues in C programs requires the following include files:-

```
     #include <fcntl.h>        /* for using the the O_ mode constants */
     #include <sys/stat.h>     /* for using other mode constants */
     #include <mqueue.h>       /* Defines the mqueue functions */
```

Creating a message queue

```
char qname [20];
int max_size =512;
int modeflags = O_CREAT | O_RDWR; /*Create if not exist*/
mode_t permissions = 0600; /*Read/Write for owner only */
struct mq_attr attr;
mqd_t mq;

strcpy(qname, "/test_queue");

attr.mq_flags = 0; /* Blocking allowed mode */
attr.mq_maxmsg = 10;
attr.mq_msgsize = max_size;

mq = mq_open(qname, modeflags, permissions, &attr);
```

If the message queue already exists it can be opened with

```
mq = mq_open(qname, modeflags);
```

Message queues are created and opened using mq_open(). This function returns a *message queue descriptor* (*mqd_t*), which is used to refer to the open message queue in later calls.

Each message queue is identified by a name of the form */somename*; that is, a null-terminated string of up to **NAME_MAX** (i.e., 255) characters consisting of an initial slash, followed by one or more characters, none of which are slashes. Two processes can operate on the same queue by passing the same text name to `mq_open()`.

POSIX message queues have kernel persistence: if not removed by `mq_unlink()`, a message queue will exist until the system is shut down.

Sending a message to an opened message queue mq

```
char buffer[max_size];
int priority = 0;

memset(buffer,0, max_size);
strcpy(buffer,"Hello There");
mq_send(mq, buffer, max_size, priority);
```

Reading a message from an opened message queue mq

```
int numRead = mq_receive(mq, buffer, max_size,
                                        &priority);
```

In circumstances where send or receive would block, the attributes flags setting determines whether or not blocking occurs. If the flag is set to 0 (as it is in this example), then blocking occurs, if the flag is set to O_NONBLOCK then these functions would return an error without blocking.

To close access to the message queue for the current process

```
mq_close(mq);
```

To request destruction of the message queue by the operating system

```
mq_unlink(qname);
```

If the requesting process has permission to do this, then the queue is not destroyed until all open descriptors in other processes that are using the queue have been closed.

Exercise: Save the program below as `mqueuewriter.c`
Compile using `cc mqueuewriter.c -o mqueuewriter -lrt`
Run using `./mqueuewriter`
The program creates and then writes 10 messages of different priorities into a message queue. Note that the message queue still exists in the operating system after the program terminates, so the messages are not lost.

```
#include <fcntl.h> /* for using the the O_mode constants */
#include <sys/stat.h> /* for using other mode constants */
#include <mqueue.h> /* Defines the mqueue functions */
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
   int modeflags = O_CREAT | O_WRONLY; /*Create if it doesn't
exist*/
   mode_t permissions = 0600; /* Read/Write for owner only */
   struct mq_attr attr;
   mqd_t mq;

   char qname [20];
   int max_size =512;
   char buffer[max_size];
   int priority;

   strcpy(qname, "/test_queue");
   attr.mq_flags = 0; /* Blocking allowed mode */
   attr.mq_maxmsg = 10;
   attr.mq_msgsize = max_size;

   mq = mq_open(qname, modeflags, permissions, &attr);
   if (mq < 0) {
      printf("Couldn't create queue %s\n",qname);
      exit(-1);
   }
   printf("Opened a message queue called %s and got descriptor %d
\n",qname,mq);

   for (priority = 1; priority<=10; priority++) {
     memset(buffer,0, max_size); /* Clear all buffer locations to
0 */
     sprintf(buffer, "%d", priority); /* Convert int priority to
string */
     strcat(buffer," message text"); /* The message we want to
send */
        printf("Sending message : '%s' to message
queue\n",buffer);
        mq_send(mq, buffer, strlen(buffer), priority);
    }
   if (!mq_close(mq))
      printf("Closed the queue\n");
}
```

Exercise: Save the program below as `mqueuereader.c`
Compile using `cc mqueuereader.c -o mqueuereader -lrt`
Run using `./mqueuereader`
The program opens the message queue, checks to see how many messages are present and then receives them all in order of priority. Notice it receives them and prints them out in reverse order to which they were sent to the queue due to the priority delivery. The program then destroys the queue, so you will need to run the writer again if you want to create it.

```
#include <fcntl.h> /* for using the the opening mode constants */
#include <sys/stat.h> /*for using the permission mode constants */
#include <mqueue.h> /* Defines the mqueue functions */
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {

   mqd_t mq;
   struct mq_attr qattr;

   char qname [20];
   int max_size =512;
   char msgbuffer[max_size];
   int i, n, rc, priority, num_messages;

   strcpy(qname, "/test_queue");
   mq = mq_open(qname, O_RDONLY);
   if (mq<0) {
      printf("Couldn't open queue %s\n", qname);
      exit(-1);
   }
   printf("Opened a message queue called %s and got descriptor %d
\n",qname,mq);

   /* Copy status information from kernel data structures
      associated with the mq into the data structure qattr */
   rc = mq_getattr(mq, &qattr);
   num_messages = qattr.mq_curmsgs; /* num msgs in the queue */

   printf("Number of messages in the queue %d\n",num_messages);
   for (i=1; i<=num_messages; i++) {
      memset(msgbuffer,0,max_size);
      n = mq_receive(mq,msgbuffer, max_size, &priority);
      printf("Received message : %s with priority %d numchars
%d\n",msgbuffer,priority,n);
   }

   if (!mq_close(mq))
      printf("Closed the queue\n");
   if (!mq_unlink(qname))
      printf("Destroyed the queue\n");
}
```