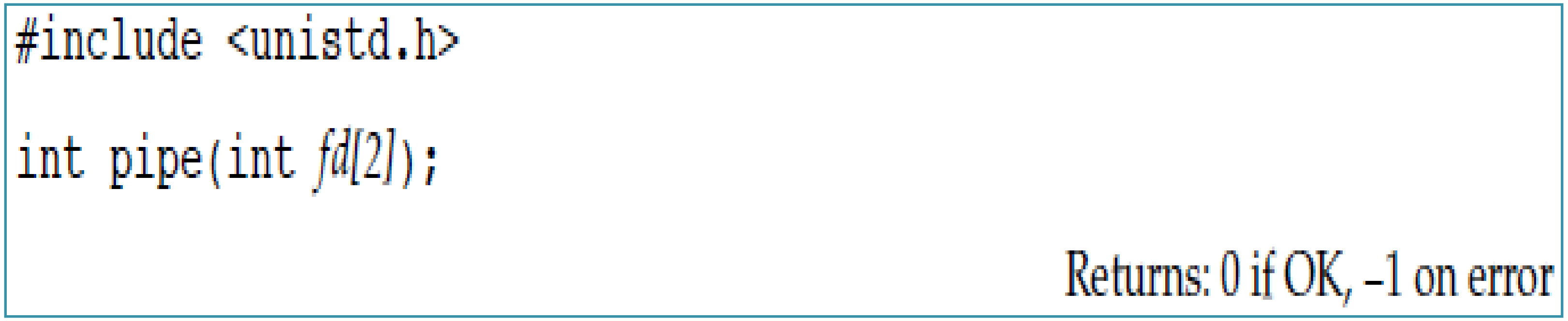
Inter-process Communication

Pipes

- Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems.
- Pipes have two limitations.
  1. Historically, they have been half duplex (i.e., data flows in only one direction).
  2. Pipes can be used only between processes that have a common ancestor.
  - Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

Pipes

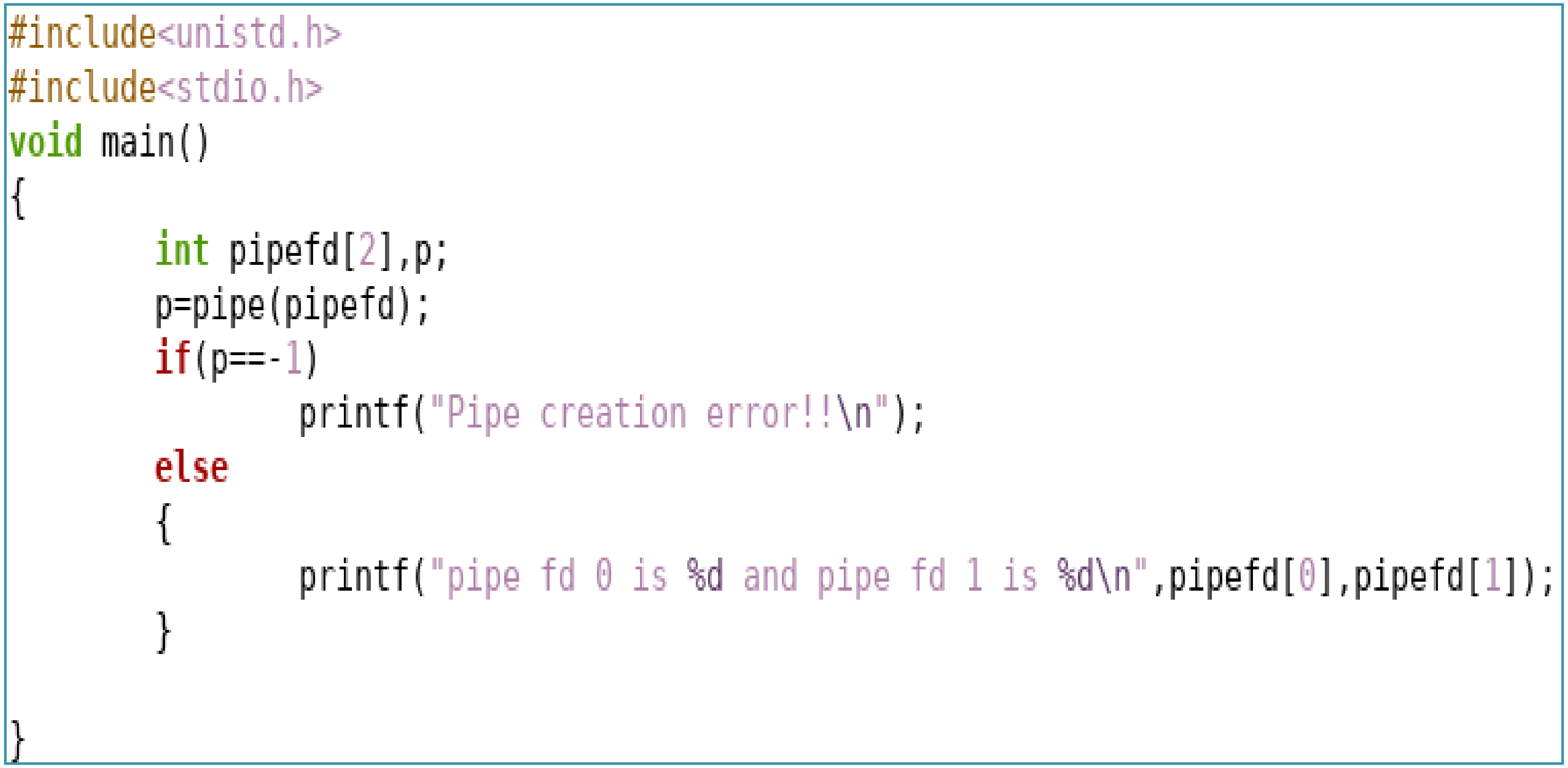
- A pipe is created by calling the pipe function.
- Two file descriptors are returned through the fd argument:
  - fd[0] is open for reading, and
  - fd[1] is open for writing.



```
#include <unistd.h>

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

Returns: 0 if OK, –1 on error

Pipes



```
#include<unistd.h>
#include<stdio.h>
void main()
{
        int pipefd[2],p;
        p=pipe(pipefd);
        if(p==-1)
                printf("Pipe creation error!!\n");
        else
        {
                printf("pipe fd 0 is %d and pipe fd 1 is %d\n",pipefd[0],pipefd[1]);
        }


}
```

Representation of Pipe

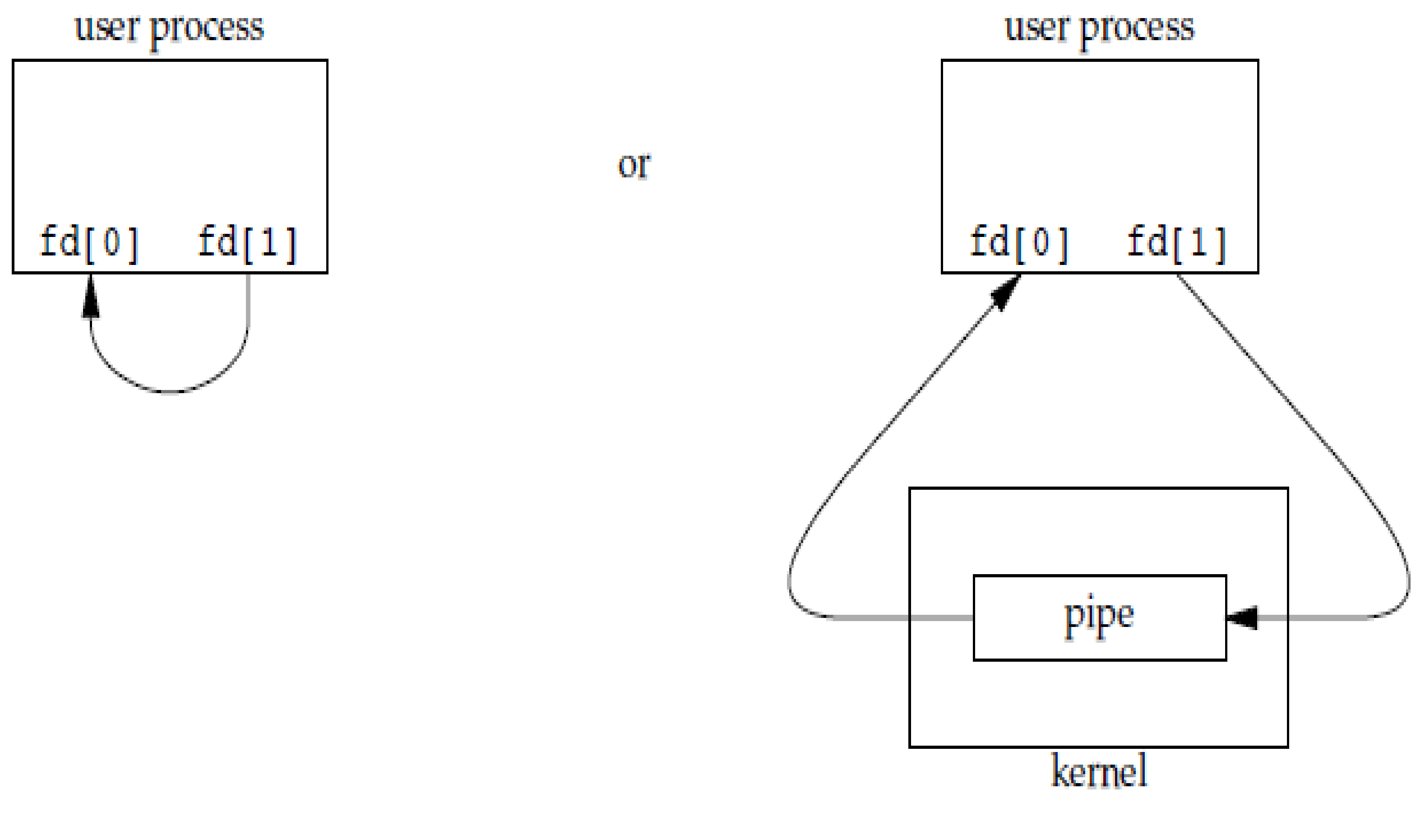


Figure 15.2 Two ways to view a half-duplex pipe

Representation of Pipe with fork

- Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child, or vice versa.

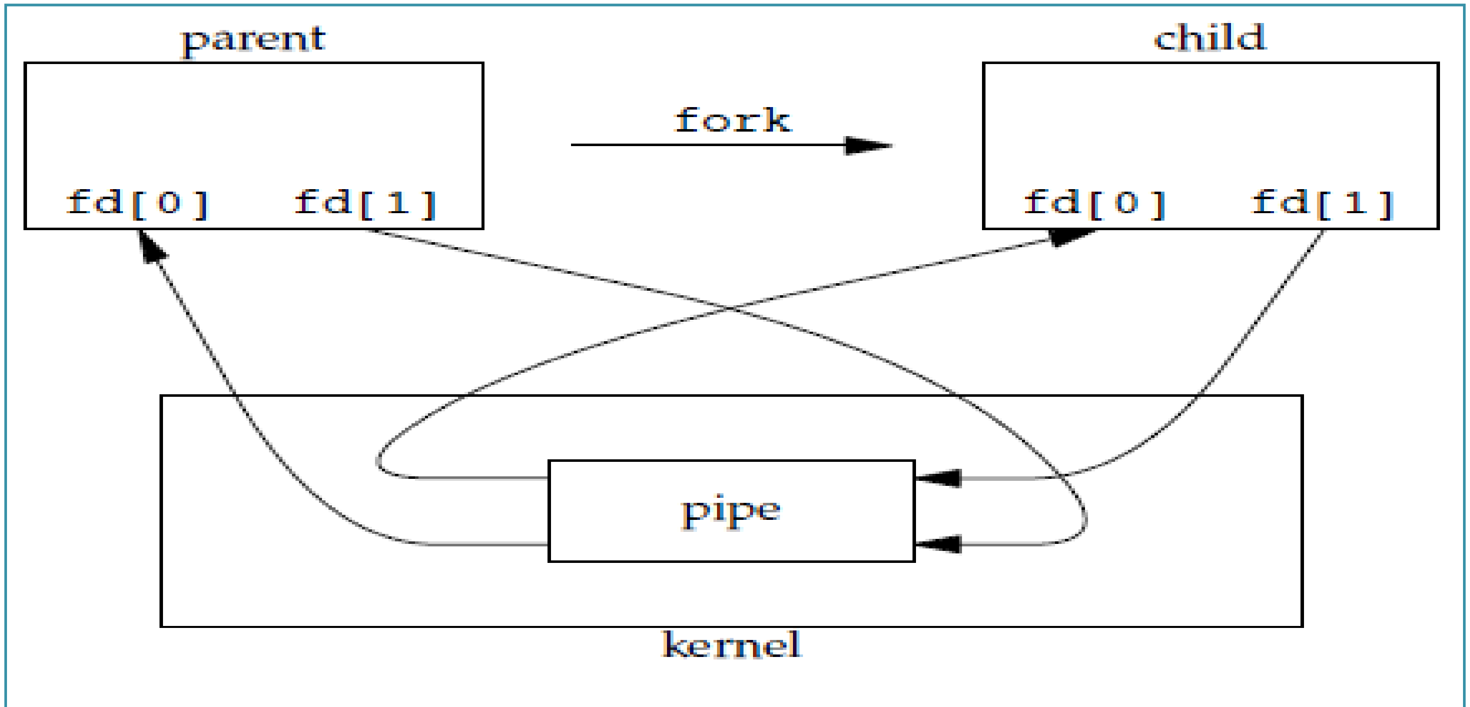


Figure 15.3 Half-duplex pipe after a fork

Pipe from Parent to Child

- For a pipe from the parent to the child, the parent closes the read end of the pipe (fd[0]), and the child closes the write end (fd[1]).

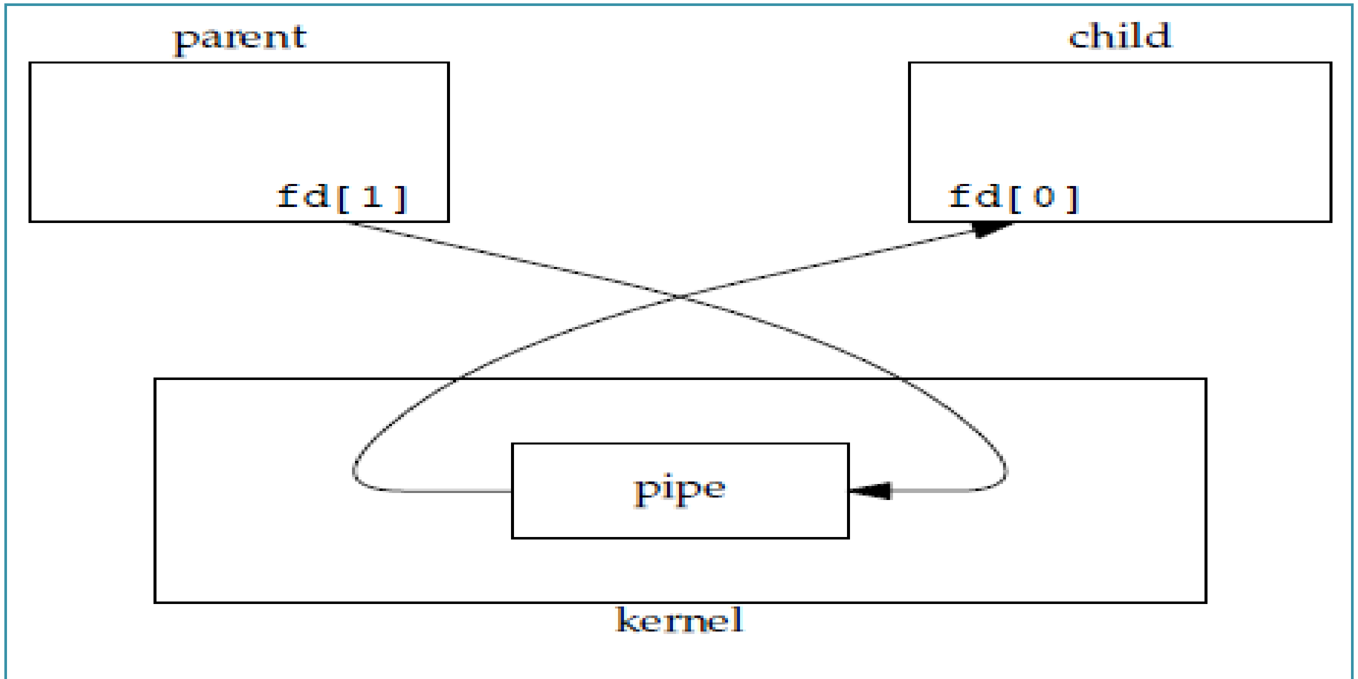
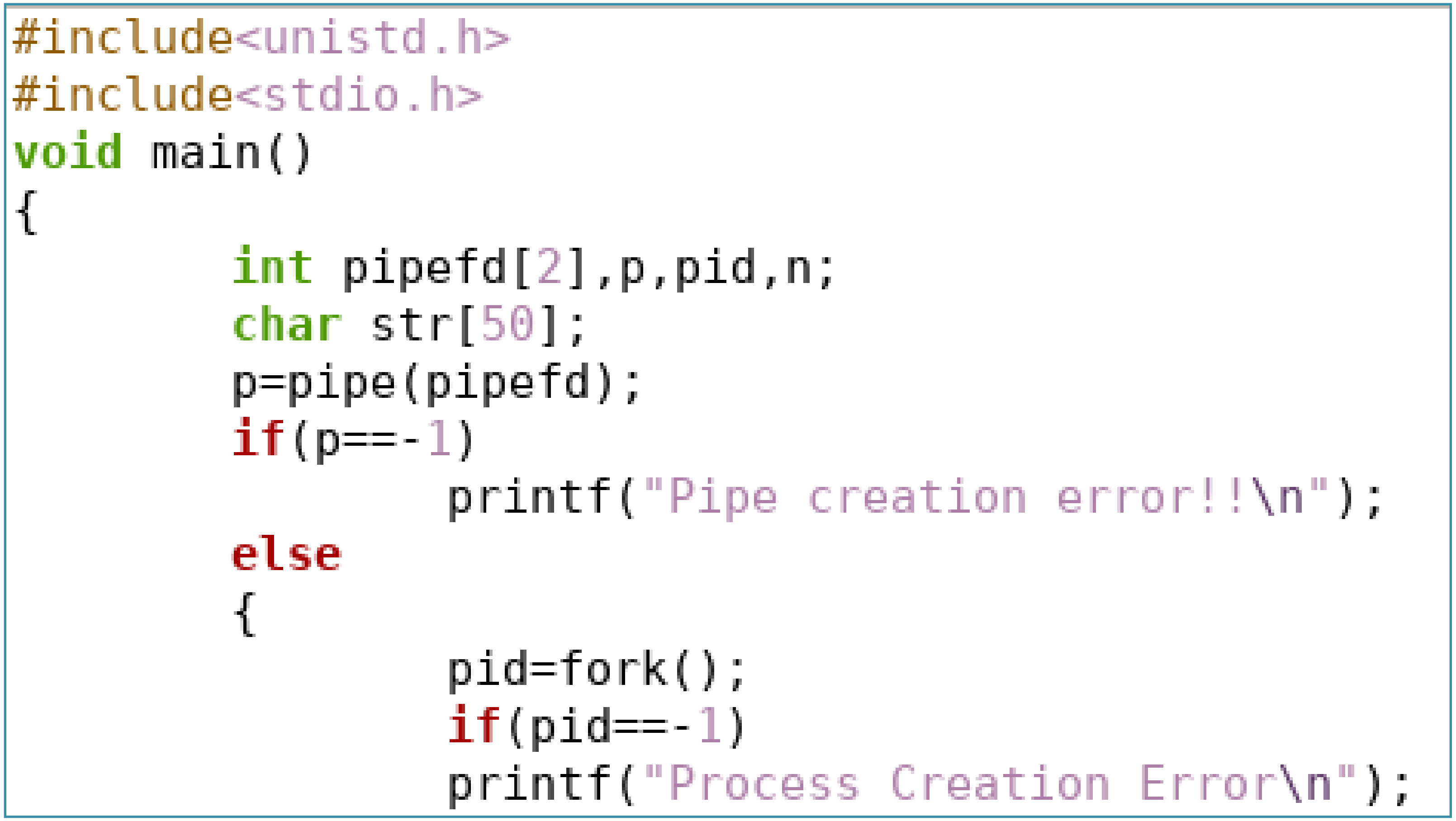


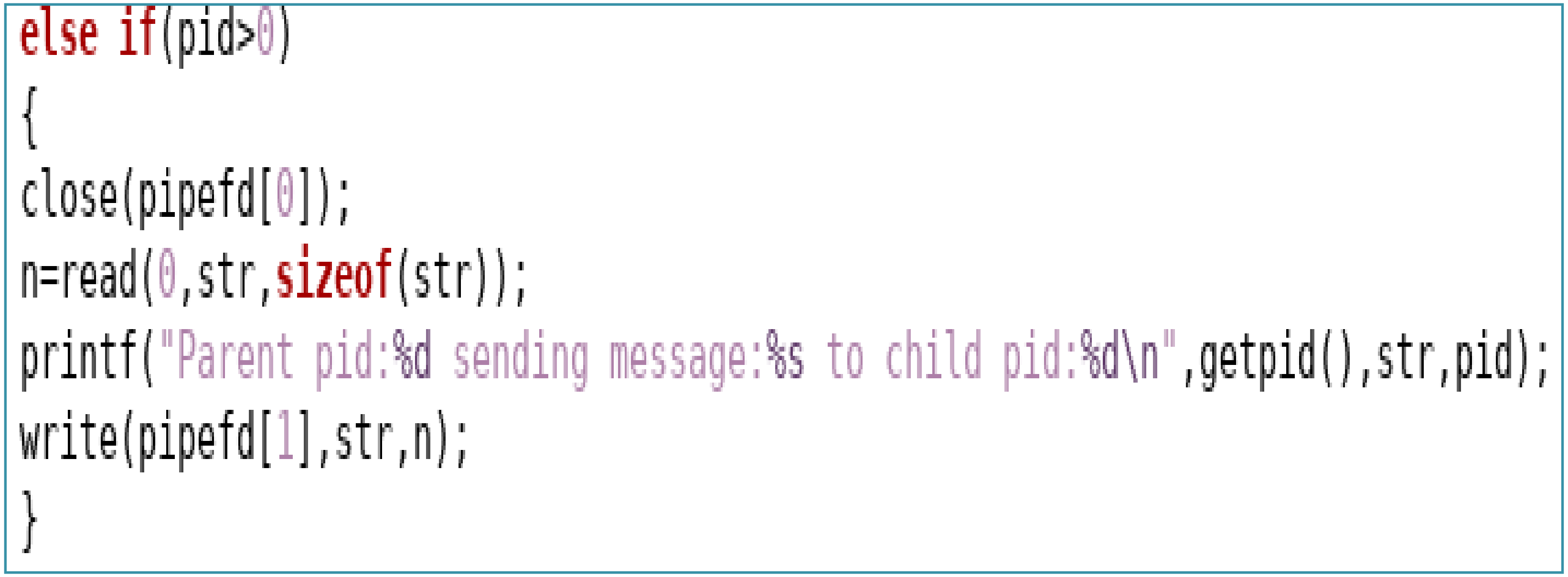
Figure 15.4 Pipe from parent to child

Program-1



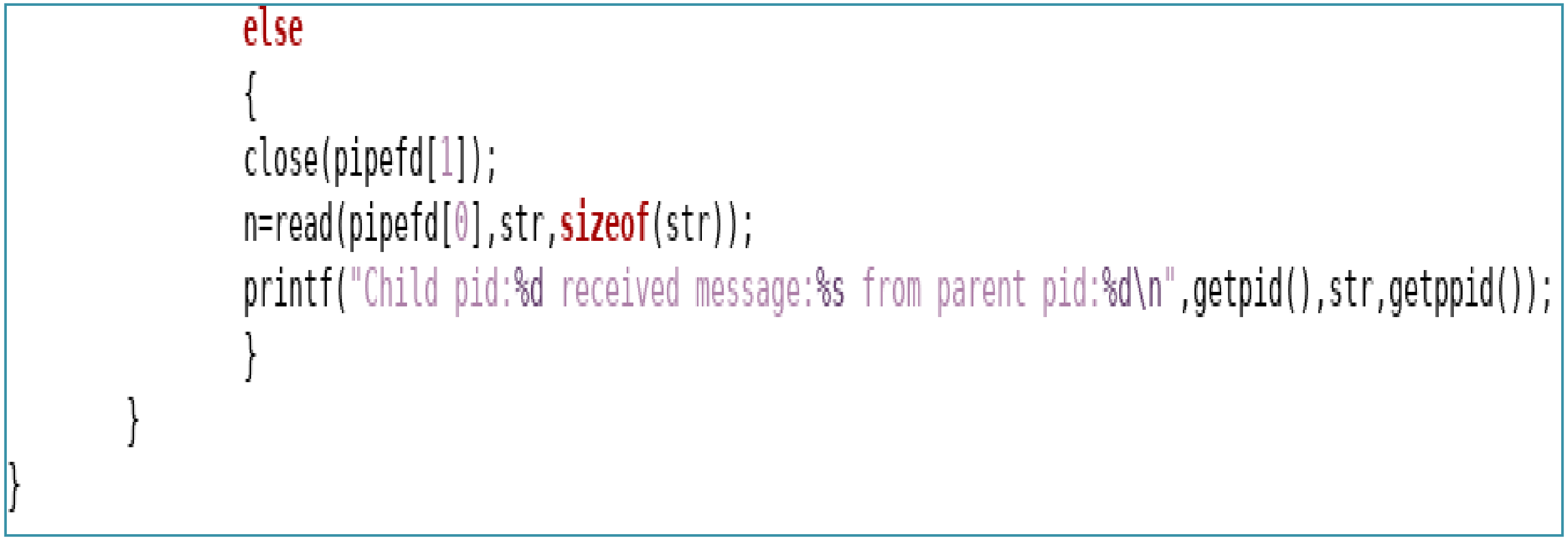
```
#include<unistd.h>
#include<stdio.h>
void main()
{
        int pipefd[2],p,pid,n;
        char str[50];
        p=pipe(pipefd);
        if(p==-1)
                printf("Pipe creation error!!\n");
        else
        {
                pid=fork();
                if(pid==-1)
                printf("Process Creation Error\n");
```

Program-1



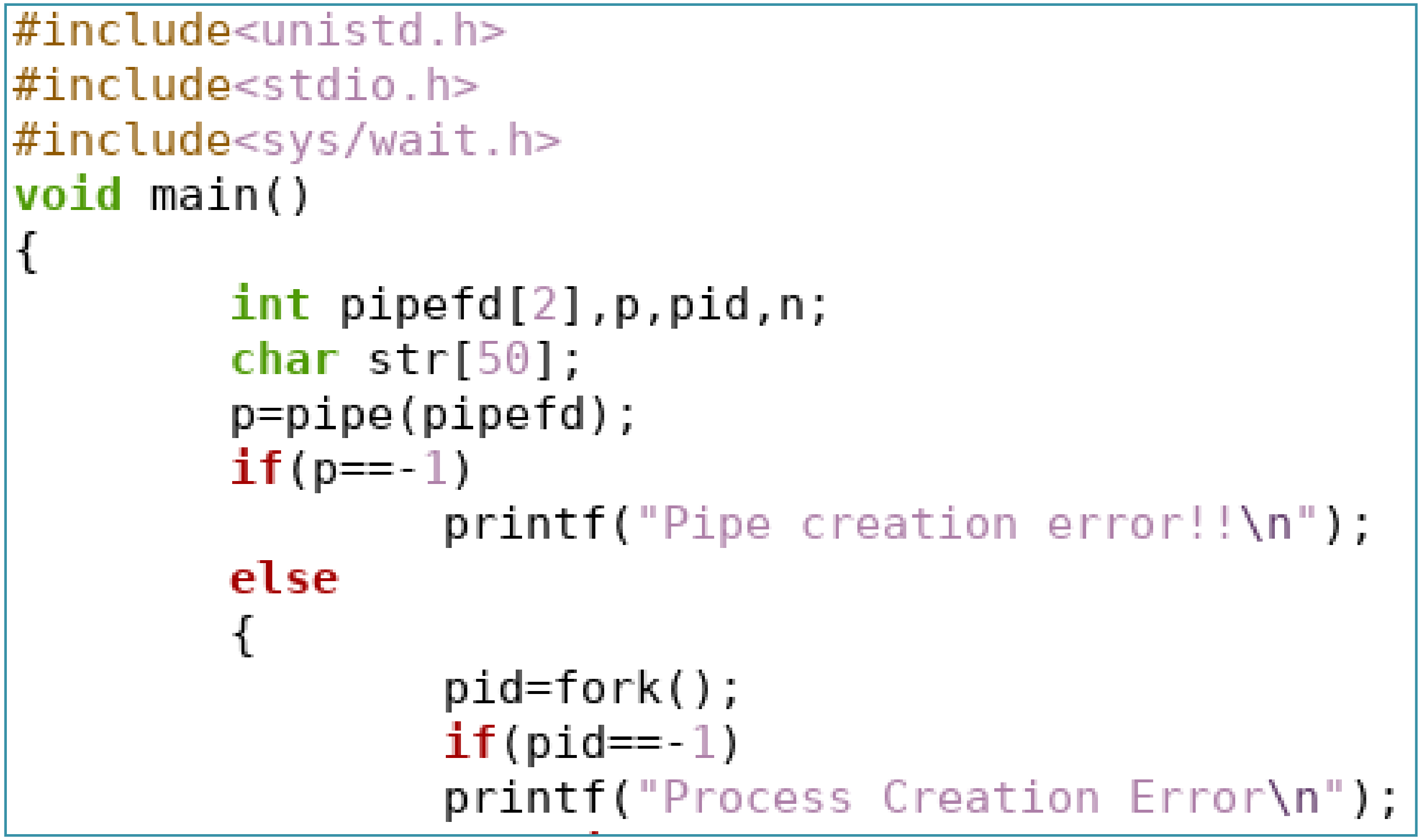
```
else if(pid>0)
{
close(pipefd[0]);
n=read(0,str,sizeof(str));
printf("Parent pid:%d sending message:%s to child pid:%d\n",getpid(),str,pid);
write(pipefd[1],str,n);
}
```

Program-1



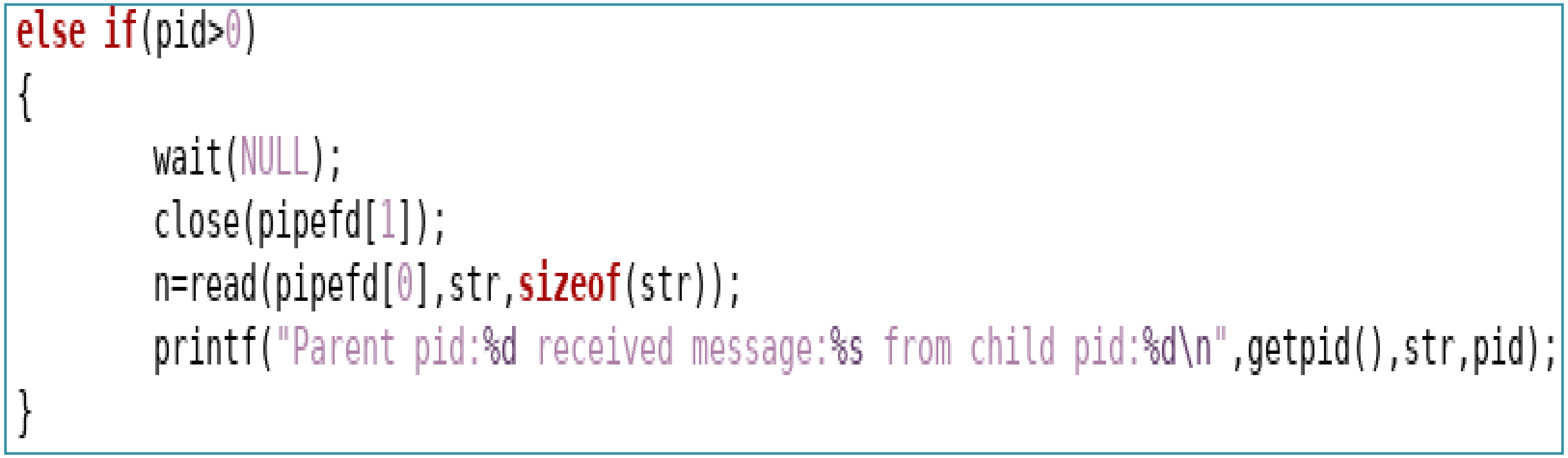
```
                else
                {
                close(pipefd[1]);
                n=read(pipefd[0],str,sizeof(str));
                printf("Child pid:%d received message:%s from parent pid:%d\n",getpid(),str,getppid());
                }
        }
}
```

Program-2



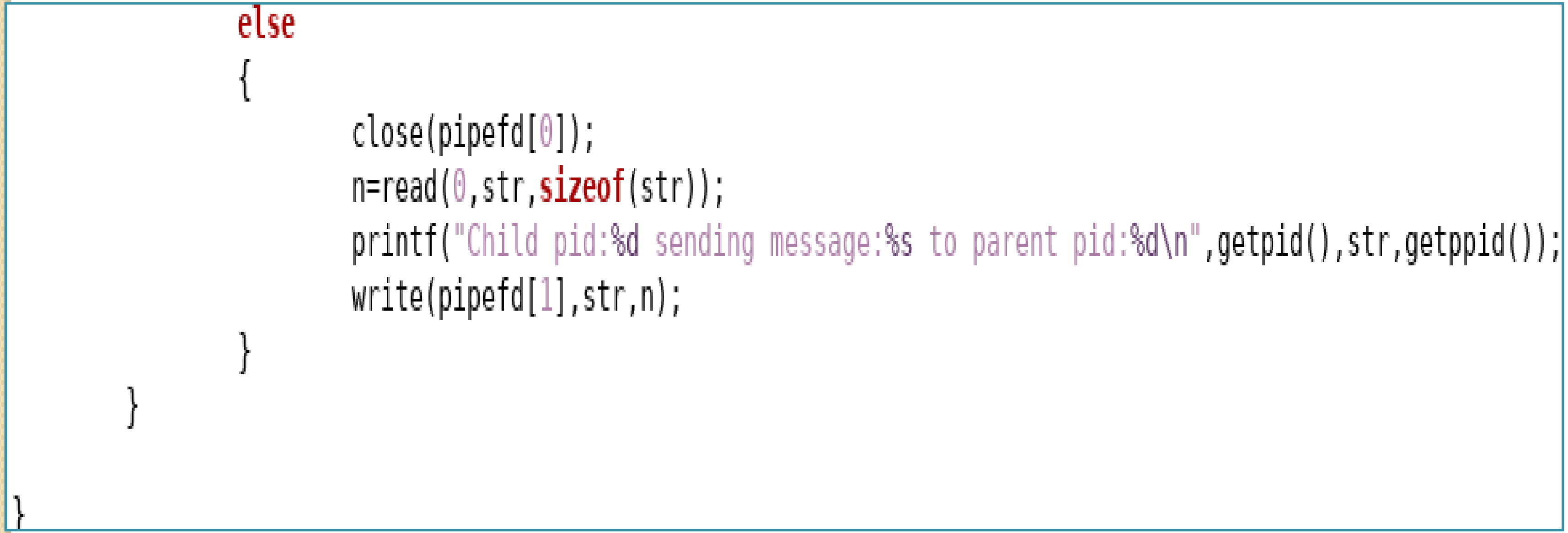
```
#include<unistd.h>
#include<stdio.h>
#include<sys/wait.h>
void main()
{
        int pipefd[2],p,pid,n;
        char str[50];
        p=pipe(pipefd);
        if(p==-1)
                printf("Pipe creation error!!\n");
        else
        {
                pid=fork();
                if(pid==-1)
                printf("Process Creation Error\n");
```

Program-2



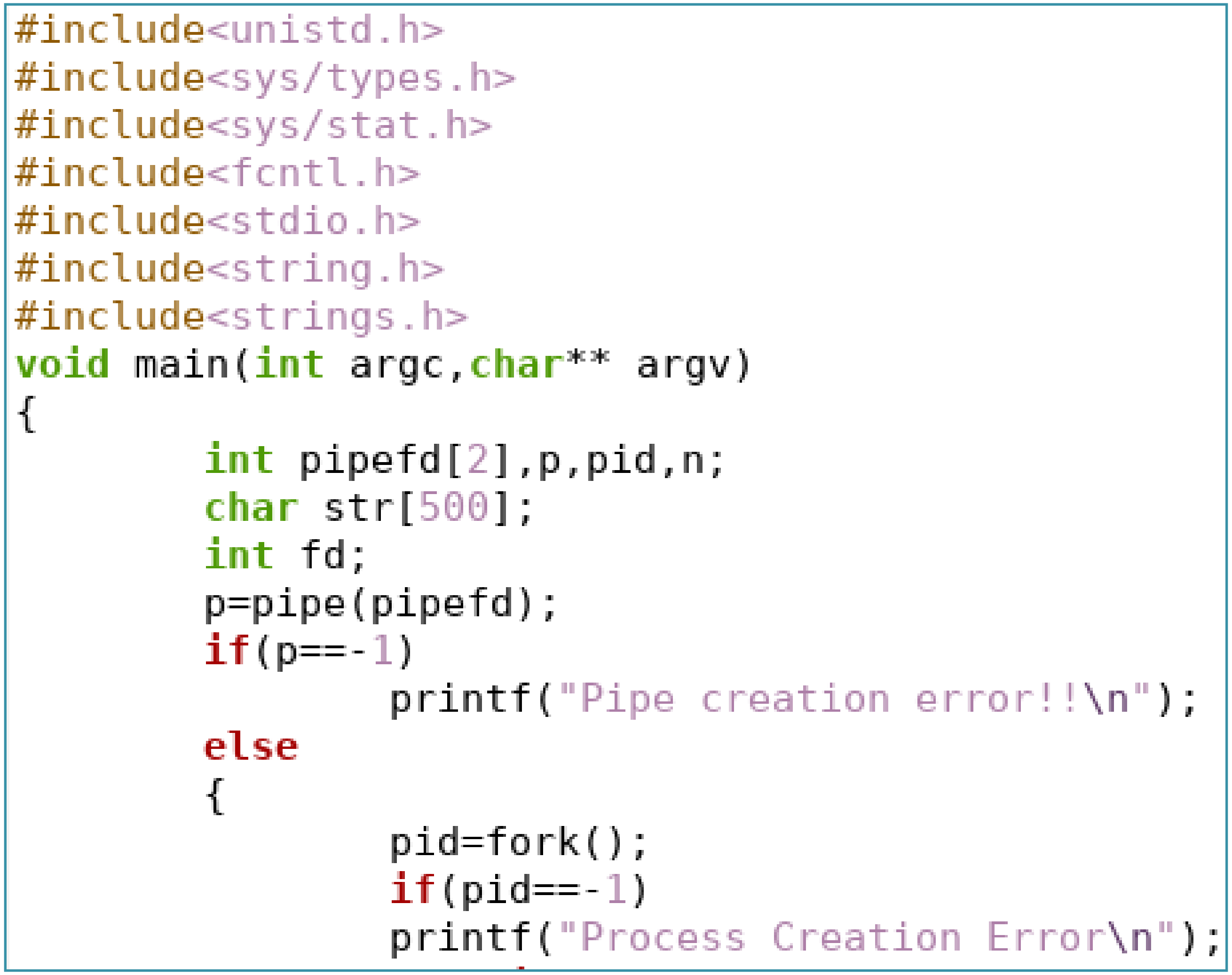
```
else if(pid>0)
{
        wait(NULL);
        close(pipefd[1]);
        n=read(pipefd[0],str,sizeof(str));
        printf("Parent pid:%d received message:%s from child pid:%d\n",getpid(),str,pid);
}
```

Program-2



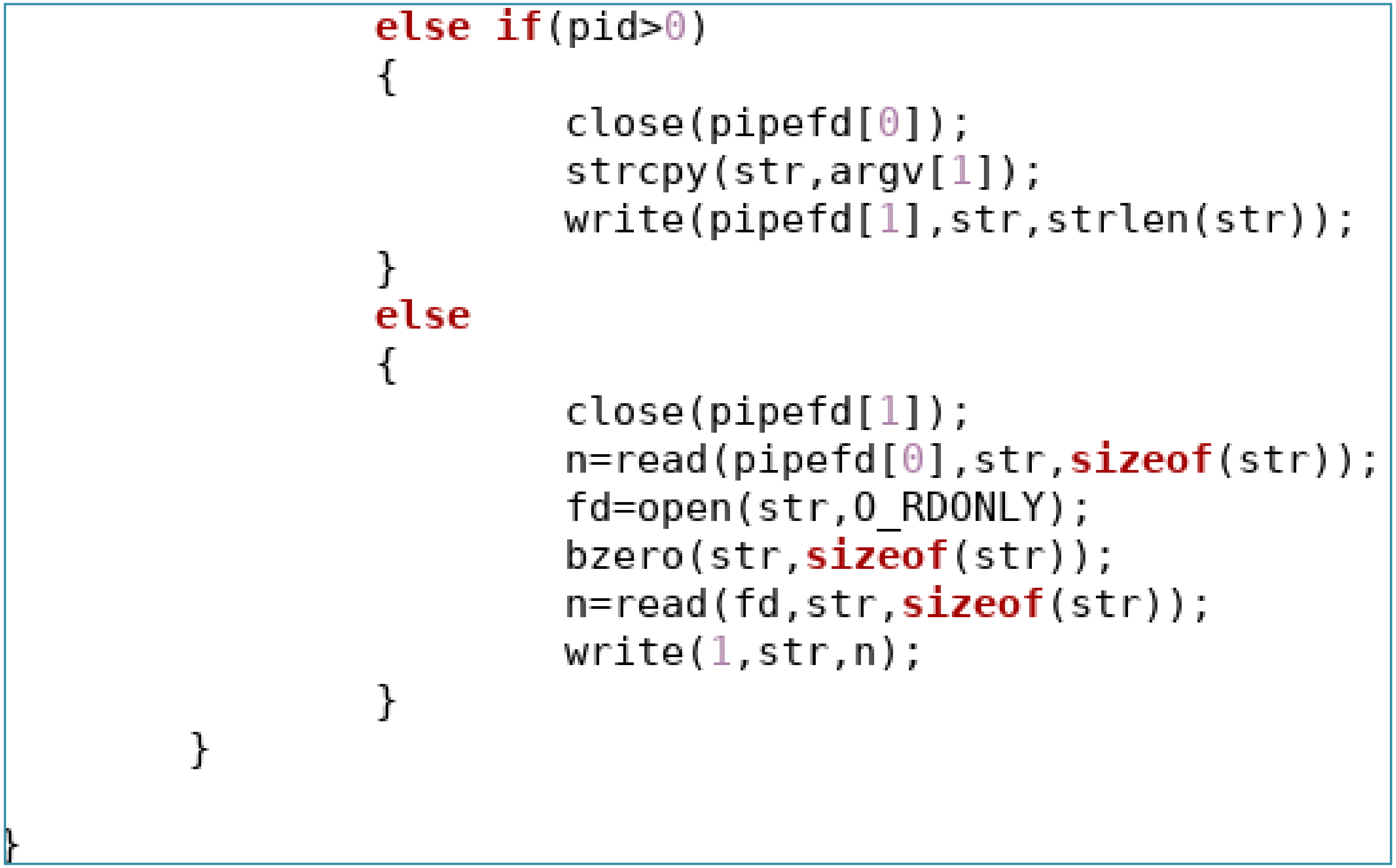
```
                else
                {
                        close(pipefd[0]);
                        n=read(0,str,sizeof(str));
                        printf("Child pid:%d sending message:%s to parent pid:%d\n",getpid(),str,getppid());
                        write(pipefd[1],str,n);
                }
        }

}
```

Program-3



```c
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<stdio.h>
#include<string.h>
#include<strings.h>
void main(int argc,char** argv)
{
        int pipefd[2],p,pid,n;
        char str[500];
        int fd;
        p=pipe(pipefd);
        if(p==-1)
                printf("Pipe creation error!!\n");
        else
        {
                pid=fork();
                if(pid==-1)
                printf("Process Creation Error\n");
```

Program-3



```
                else if(pid>0)
                {
                        close(pipefd[0]);
                        strcpy(str,argv[1]);
                        write(pipefd[1],str,strlen(str));
                }
                else
                {
                        close(pipefd[1]);
                        n=read(pipefd[0],str,sizeof(str));
                        fd=open(str,O_RDONLY);
                        bzero(str,sizeof(str));
                        n=read(fd,str,sizeof(str));
                        write(1,str,n);
                }
        }

}
```

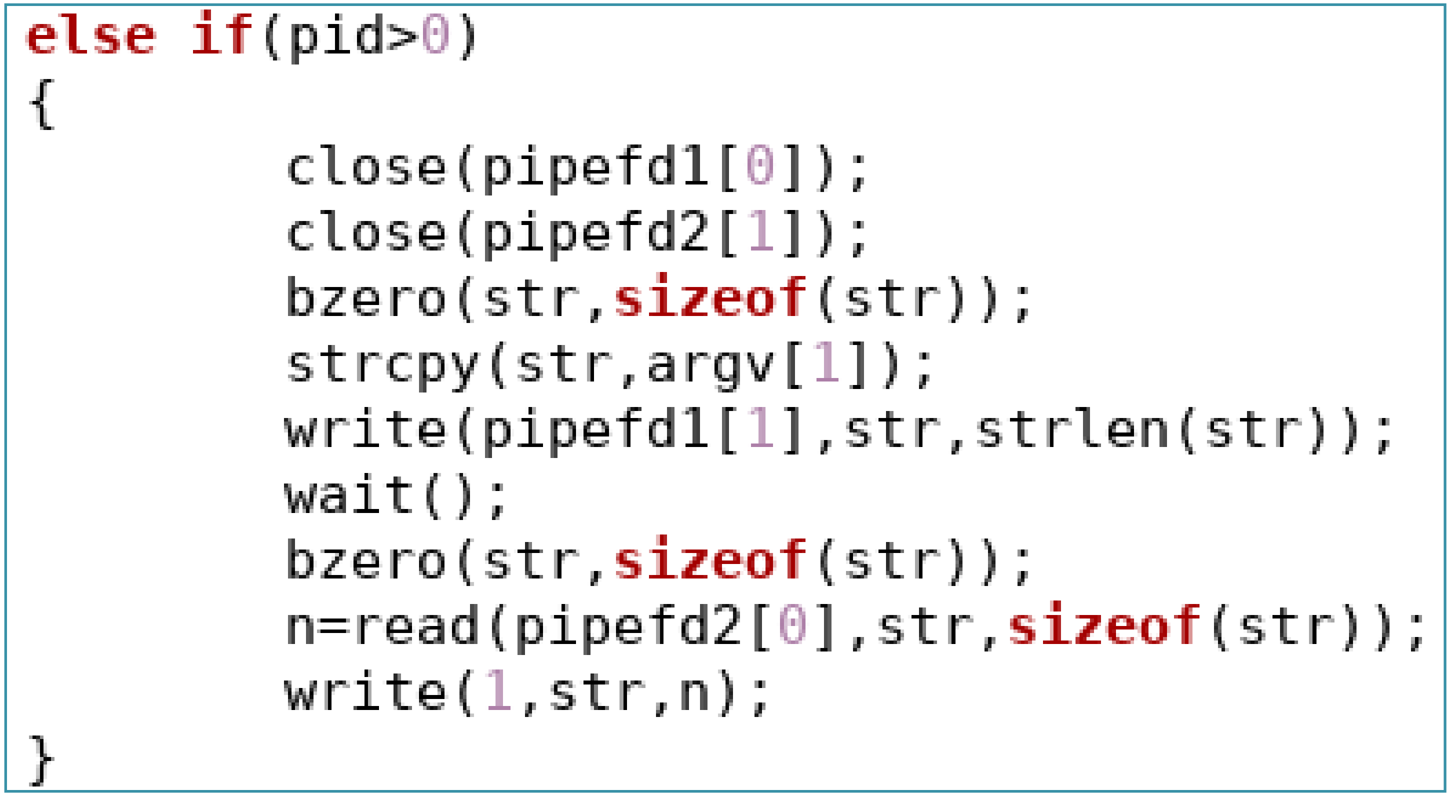
Program-4

- Parent Process writes file name in Pipe-1
- Child Process reads file name from Pipe-1
- Child Process Opens file, Reads Contents, Writes Contents in Pipe-2
- Parent Process reads Contents from Pipe-2 and Writes on Terminal

Program-4

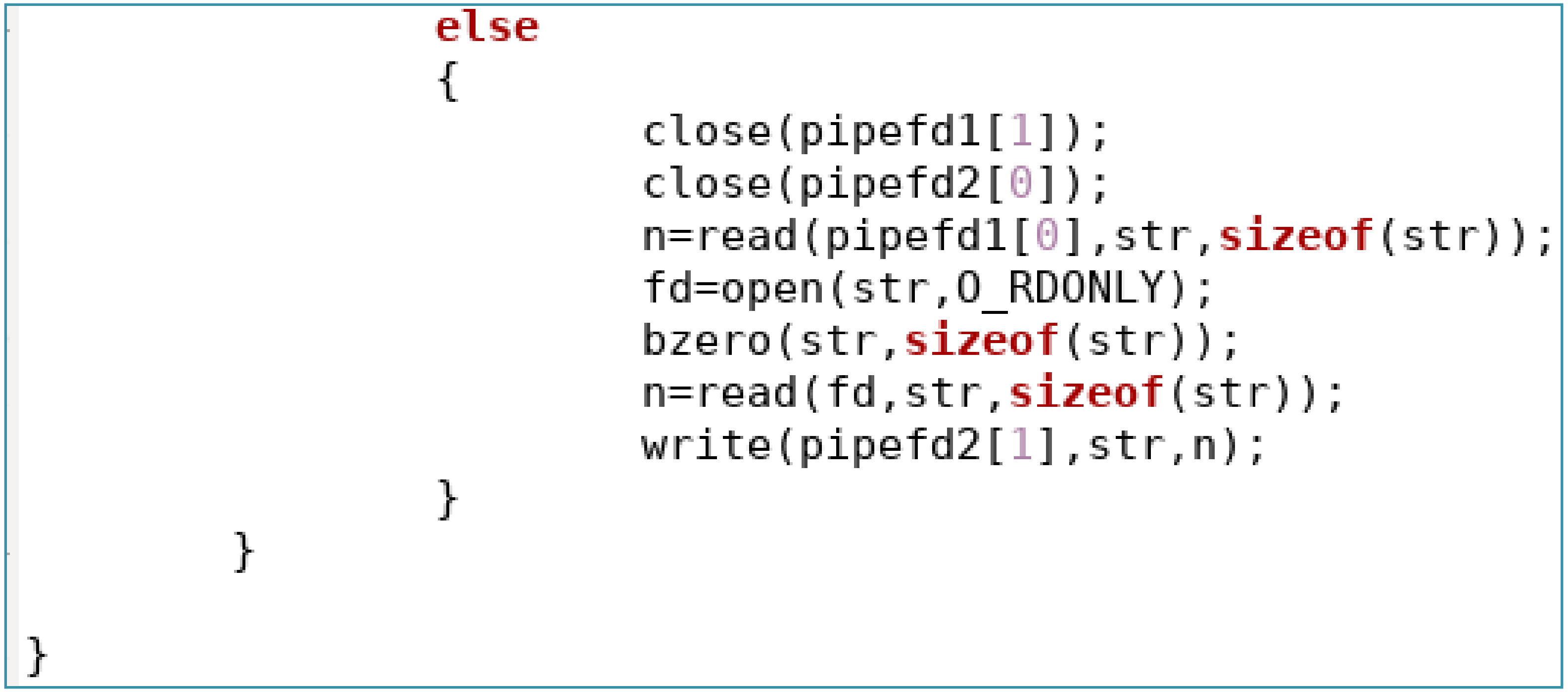
```c
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<stdio.h>
#include<string.h>
#include<strings.h>
void main(int argc,char** argv)
{
        int pipefd1[2],pipefd2[2],p1,p2,pid,n;
        char str[500];
        int fd;
        p1=pipe(pipefd1);
        p2=pipe(pipefd2);
        if(p1==-1 && p2==-1)
                printf("Pipe creation error!!\n");
        else
        {
                pid=fork();
                if(pid==-1)
                printf("Process Creation Error\n");
```

Program-4



```c
else if(pid>0)
{
        close(pipefd1[0]);
        close(pipefd2[1]);
        bzero(str,sizeof(str));
        strcpy(str,argv[1]);
        write(pipefd1[1],str,strlen(str));
        wait();
        bzero(str,sizeof(str));
        n=read(pipefd2[0],str,sizeof(str));
        write(1,str,n);
}
```

Program-4



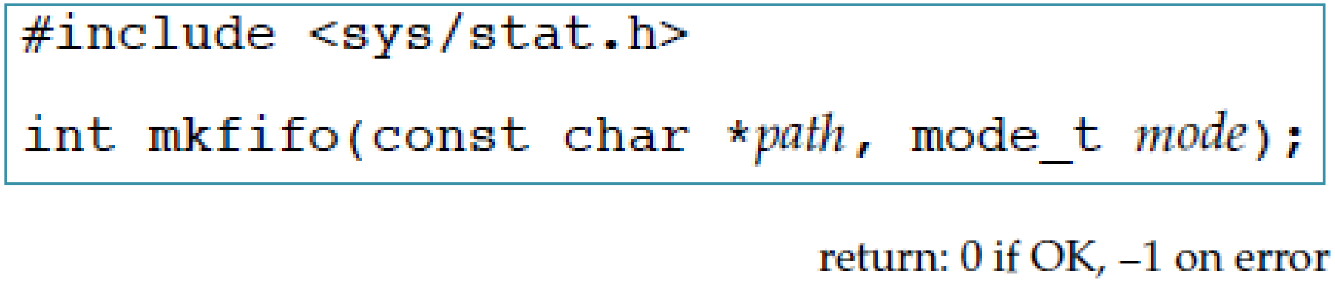
```
                else
                {
                        close(pipefd1[1]);
                        close(pipefd2[0]);
                        n=read(pipefd1[0],str,sizeof(str));
                        fd=open(str,O_RDONLY);
                        bzero(str,sizeof(str));
                        n=read(fd,str,sizeof(str));
                        write(pipefd2[1],str,n);
                }
        }

}
```

popen and pclose

- The standard I/O library has provided the popen and pclose functions.
- These two functions handle following:
  - Creating a pipe,
  - Forking a child,
  - Closing the unused ends of the pipe,
  - Executing a shell to run the command, and
  - Waiting for the command to terminate.

FIFO

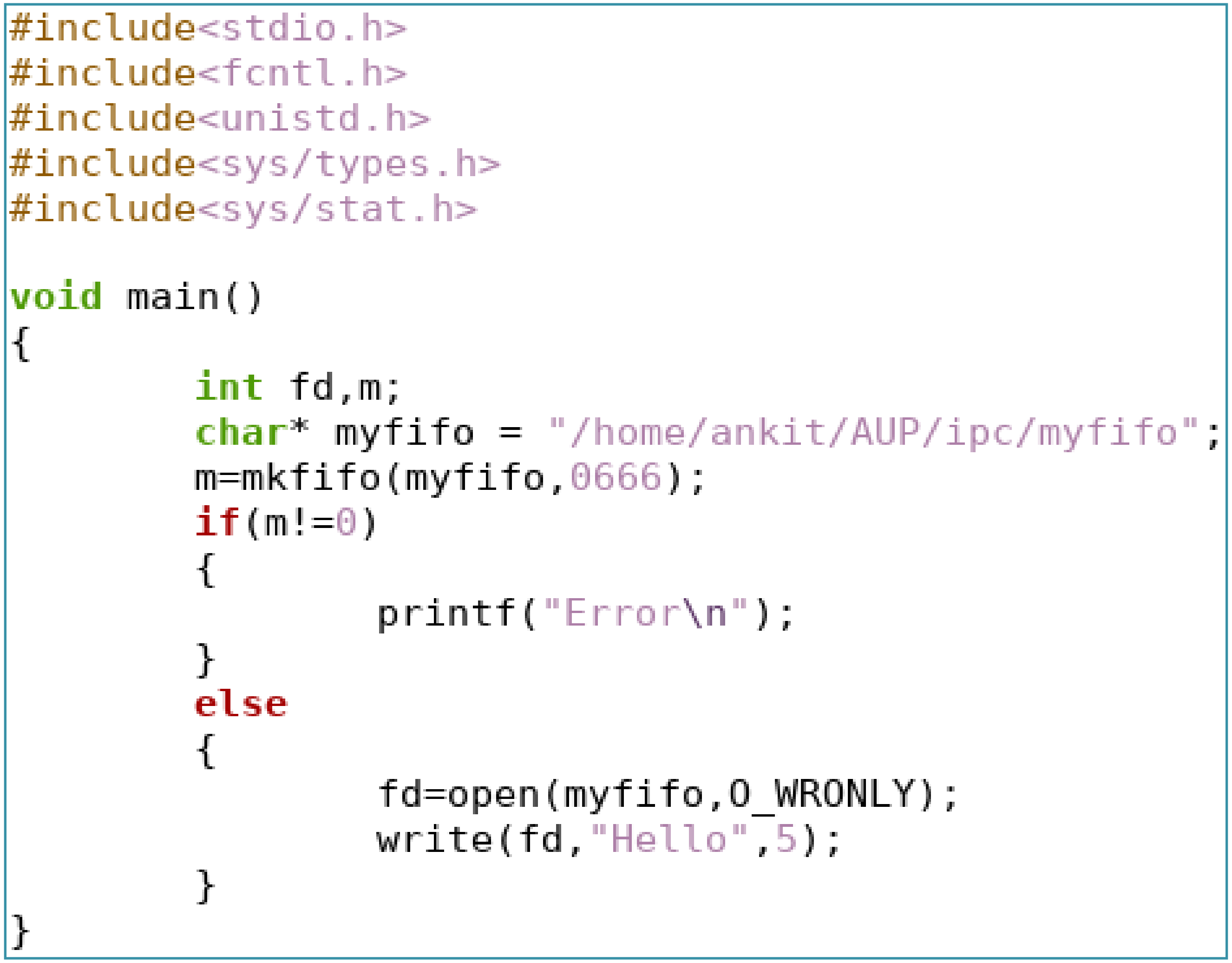
- FIFOs are also called named pipes.
- Unnamed pipes can be used only between related processes when a common ancestor has created the pipe.
- With FIFOs, unrelated processes can exchange data.



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

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

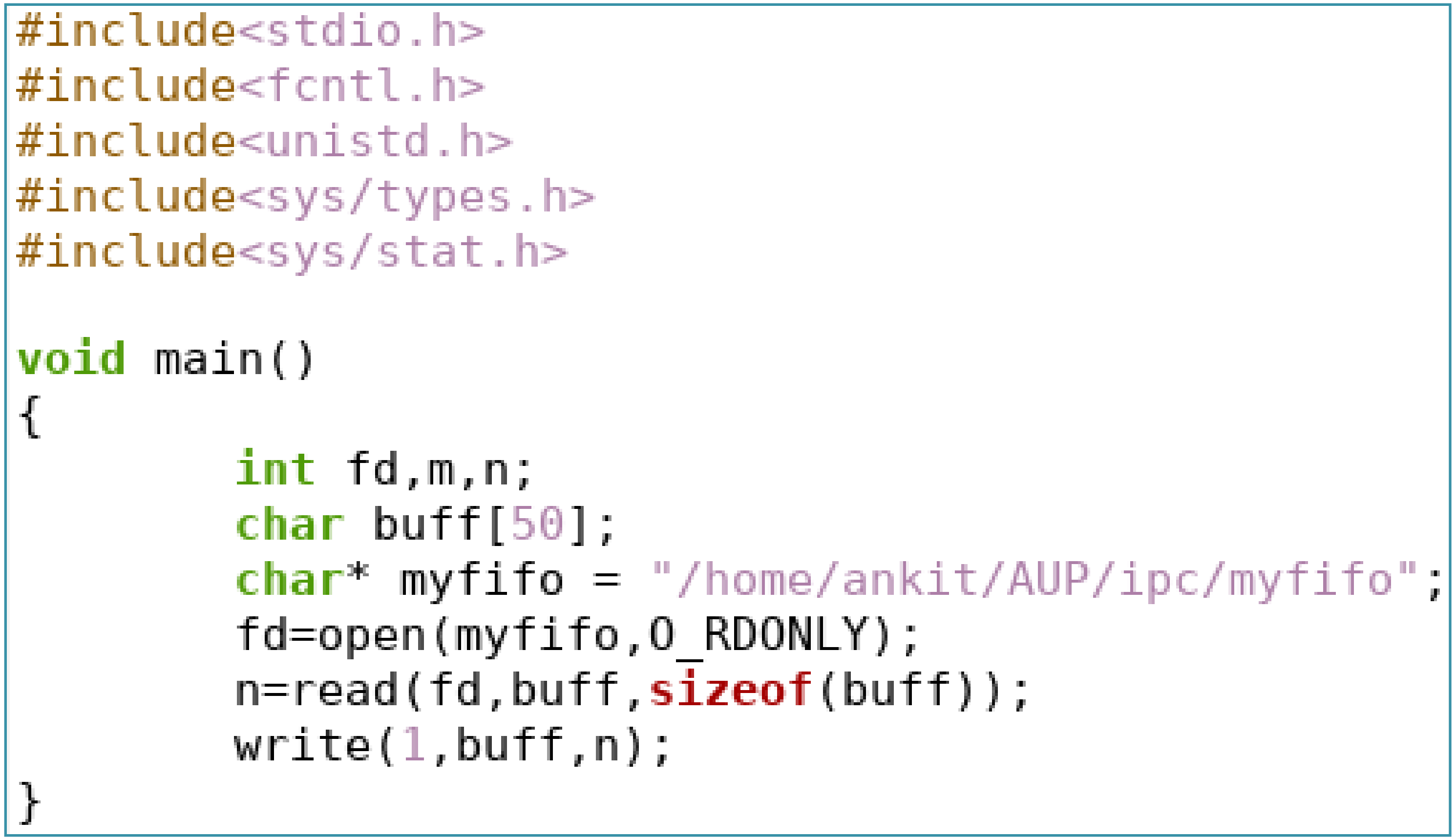
return: 0 if OK, –1 on error

FIFO - Write



```
#include<stdio.h>
#include<fcntl.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>

void main()
{
        int fd,m;
        char* myfifo = "/home/ankit/AUP/ipc/myfifo";
        m=mkfifo(myfifo,0666);
        if(m!=0)
        {
                printf("Error\n");
        }
        else
        {
                fd=open(myfifo,O_WRONLY);
                write(fd,"Hello",5);
        }
}
```

FIFO-Read



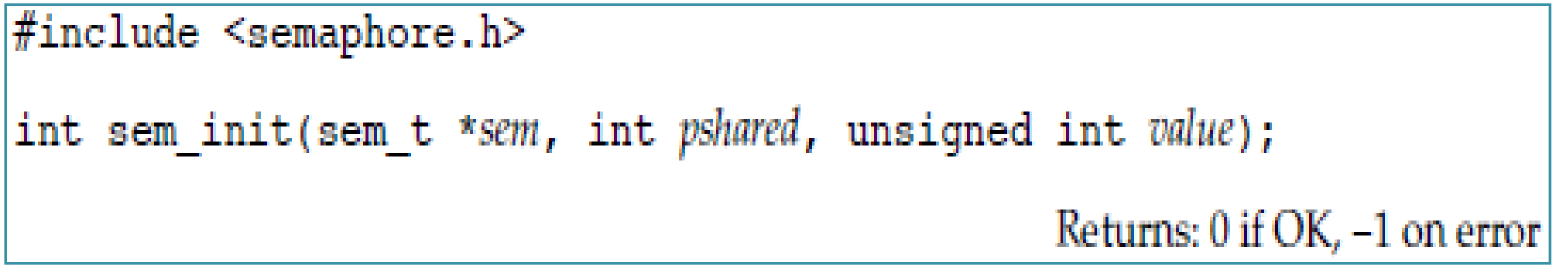
```c
#include<stdio.h>
#include<fcntl.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>

void main()
{
        int fd,m,n;
        char buff[50];
        char* myfifo = "/home/ankit/AUP/ipc/myfifo";
        fd=open(myfifo,O_RDONLY);
        n=read(fd,buff,sizeof(buff));
        write(1,buff,n);
}
```

Semaphores

- A semaphore is a counter used to provide access to a shared data object for multiple processes.
- To obtain a shared resource, a process needs to do the following:
  1. Test the semaphore that controls the resource.
  2. If the value of the semaphore is positive, the process can use the resource. In this case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource.
  3. Otherwise, if the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1.

sem_init

- To create a semaphore, sem_init is used.



```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Returns: 0 if OK, −1 on error

- The pshared argument indicates that the semaphore is used with multiple processes or not.
- If yes, then pshared is set to a nonzero value.
- The value argument specifies the initial value of the semaphore.

sem_wait

- To decrement the value of a semaphore, we can use the sem_wait function.

```
#include <semaphore.h>
```

```
int sem_wait(sem_t *sem);
```

return: 0 if OK, –1 on error

sem_post

- To increment the value of a semaphore, we call the sem_post function.
- This is analogous to unlocking a binary semaphore or releasing a resource associated with a counting semaphore.

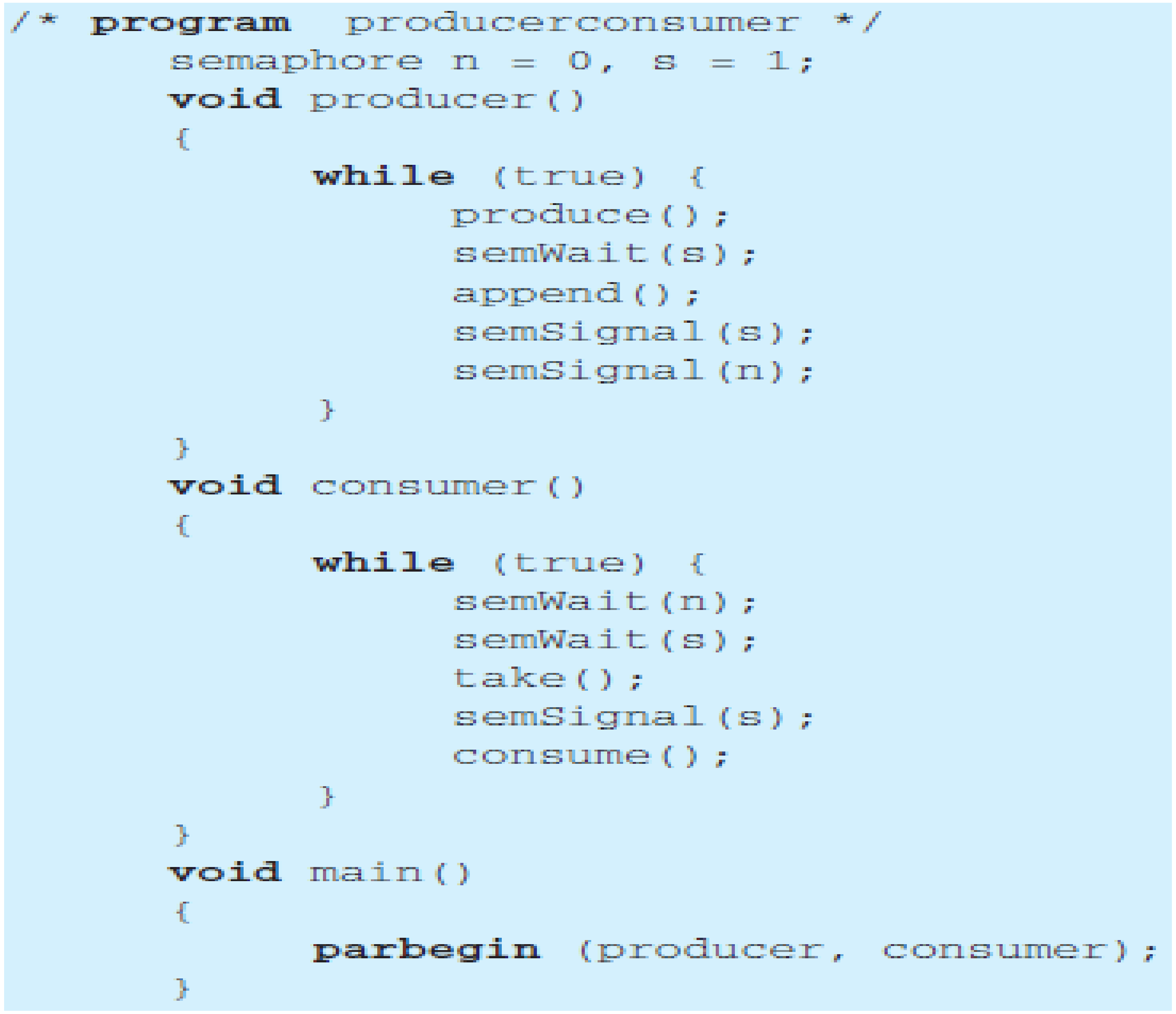
```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

Returns: 0 if OK, –1 on error

Definitions of functions

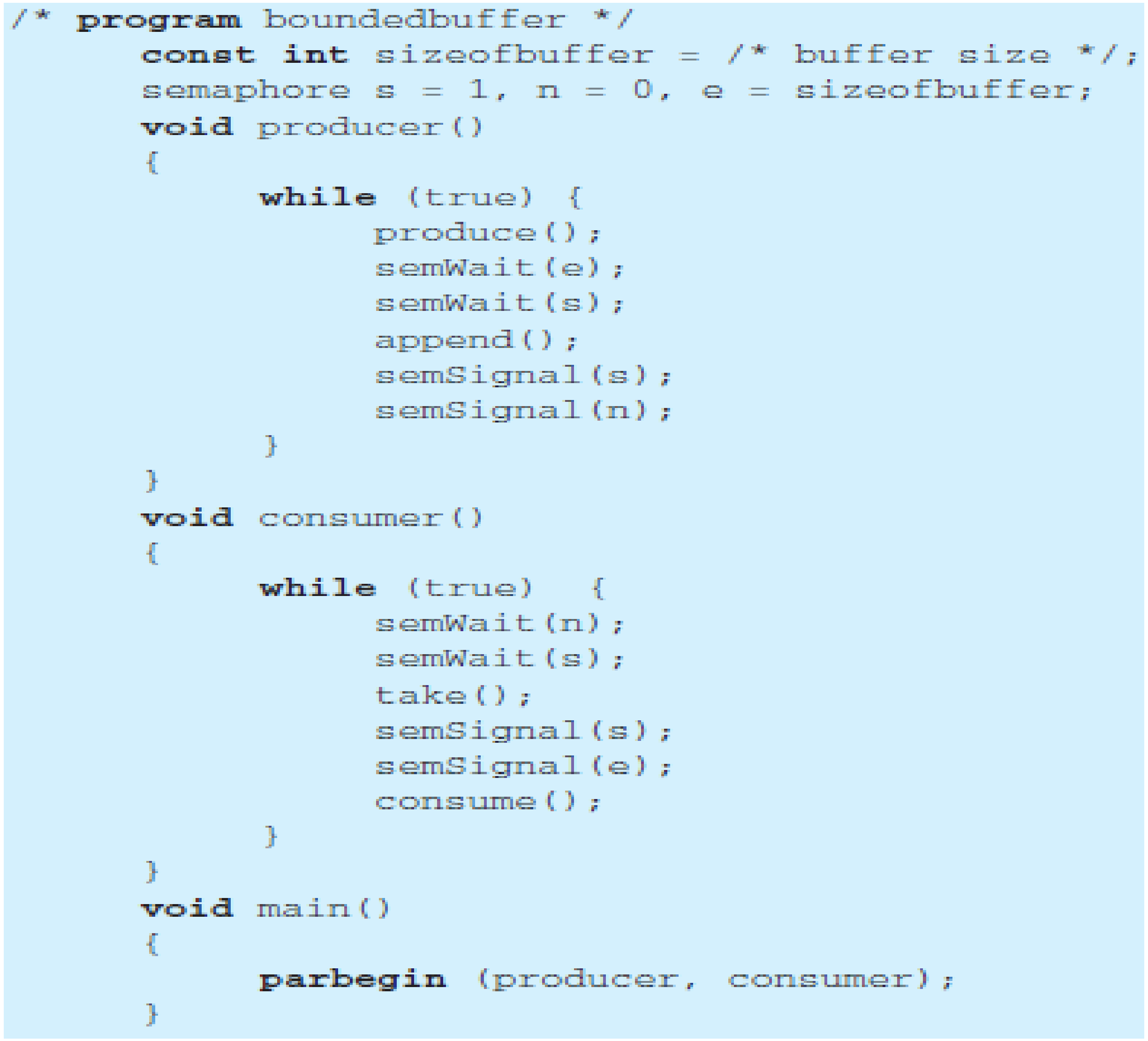
```
struct semaphore {
        int count;
        queueType queue;
};
void semWait(semaphore s)
{
        s.count--;
        if (s.count < 0) {
             /* place this process in s.queue */;
             /* block this process */;
        }
}
void semSignal(semaphore s)
{
        s.count++;
        if (s.count <= 0) {
             /* remove a process P from s.queue */;
             /* place process P on ready list */;
        }
}
```

Producer – Consumer Problem



```
/* program  producerconsumer */
     semaphore n = 0, s = 1;
     void producer()
     {
          while (true) {
               produce();
               semWait(s);
               append();
               semSignal(s);
               semSignal(n);
          }
     }
     void consumer()
     {
          while (true) {
               semWait(n);
               semWait(s);
               take();
               semSignal(s);
               consume();
          }
     }
     void main()
     {
          parbegin (producer, consumer);
     }
```

Producer – Consumer Problem



```
/* program boundedbuffer */
    const int sizeofbuffer = /* buffer size */;
    semaphore s = 1, n = 0, e = sizeofbuffer;
    void producer()
    {
         while (true) {
              produce();
              semWait(e);
              semWait(s);
              append();
              semSignal(s);
              semSignal(n);
         }
    }
    void consumer()
    {
         while (true)  {
              semWait(n);
              semWait(s);
              take();
              semSignal(s);
              semSignal(e);
              consume();
         }
    }
    void main()
    {
         parbegin (producer, consumer);
    }
```

Shared Memory

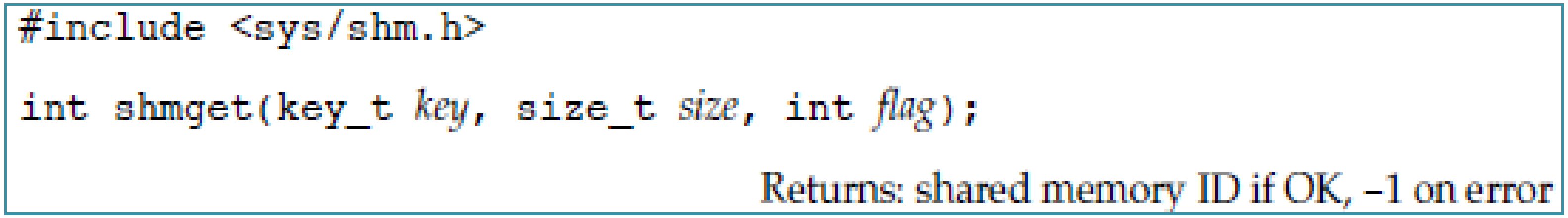
- If Client – Server communication is to be implemented, the problem with pipes, fifo and message queue is following:
  - For two process to exchange information, the information has to go through the kernel.
- Shared memory provides a way by letting two or more processes share a memory segment.
- With Shared Memory the data is directly copied from input file into shared memory and from shared memory to the output file.

Shared Memory

- Shared memory allows two or more processes to share a given region of memory.
- This is the fastest form of IPC.

shmget

- The first function called is usually shmget, to obtain a shared memory identifier.



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

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

Returns: shared memory ID if OK, –1 on error

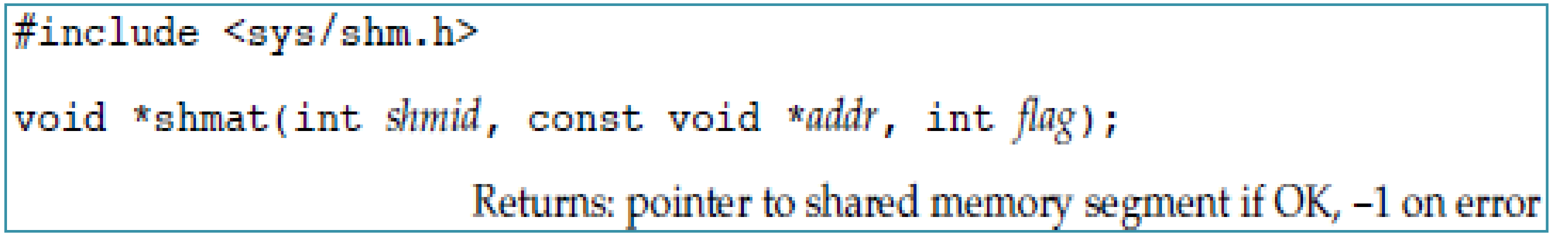
- The **first argument, key,** recognizes the shared memory segment. The key can be either an arbitrary value or one that can be derived from the library function ftok().
- The **second argument, size,** is the size of the shared memory segment rounded to multiple of PAGE_SIZE.

shmget

- The **third argument, flag,** specifies the required shared memory flag/s such as IPC_CREAT (creating new segment) or IPC_EXCL (Used with IPC_CREAT to create new segment and the call fails, if the segment already exists). Need to pass the permissions as well.

shmat

- Once a shared memory segment has been created, a process attaches it to its address space by calling shmat.



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

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

Returns: pointer to shared memory segment if OK, –1 on error

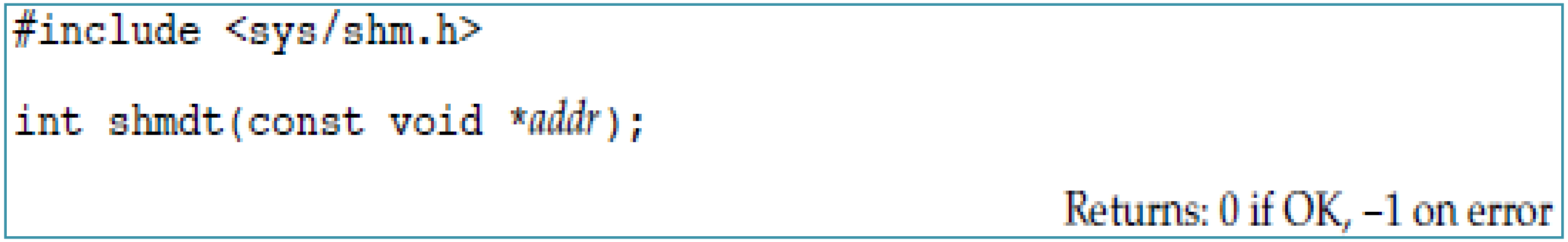
- **The first argument, shmid,** is the identifier of the shared memory segment. This id is the shared memory identifier, which is the return value of shmget() system call.

shmat

- **The second argument, addr,** is to specify the attaching address. If shmaddr is NULL, the system by default chooses the suitable address to attach the segment.
- **The third argument, flag,** specifies the required shared memory flag.

shmdt

- When we’re done with a shared memory segment, we call shmdt to detach it.



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

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

Returns: 0 if OK, –1 on error

- The addr argument is the value that was returned by a previous call to shmat.