Module XXVI

A Pipe Mechanism
(Bounded Buffer)

Bounded Buffer

- A basic idea
- Fixed-size buffer used to transfer data between processes
  - One process sends bytes of data in the buffer
  - Another process extracts the data
- The buffer mechanism includes process synchronization
  - The sender is blocked if the buffer is full
  - The receiver is blocked if the buffer is empty
- Note: this module explains how an operating system can offer a bounded buffer mechanism, and the next module shows how a shell uses such a mechanism

The Unix Pipe Mechanism

- A *pipe* consists of a bounded buffer integrated into the I/O subsystem
- Operations
  - *Create* a pipe and receive two file descriptors: one for input and one for output
  - *Write* data into the descriptor used for input
  - *Read* data from the descriptor used for output
  - *Close* the input descriptor to send *end-of-file* and once all data has been read, *close* the output descriptor to deallocate the pipe

The Xinu Pipe Mechanism

- Also integrates pipes into the I/O subsystem
- Only uses one device descriptor per pipe
- Both the sending and receiving processes must each close the descriptor before the pipe is deallocated

Unlike Unix, the Xinu pipe mechanism only uses one device descriptor for each pipe. The system requires the descriptor to be closed twice, once after the sending process finishes sending data, and a second time after the receiving process reaches end-of-file.

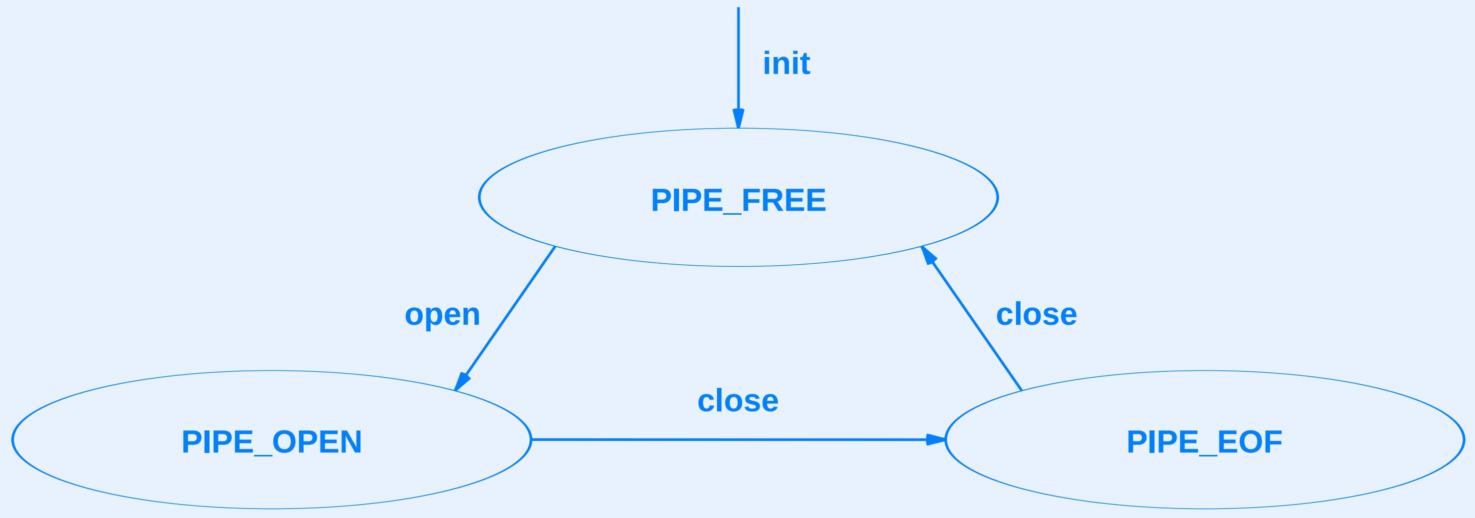
Pipe Devices In Xinu

- A single primary pipe device named *PIPE*
- A set of pipe pseudo-devices (PIPE0, PIPE1, PIPE2,...) with one of the pseudo-devices allocated whenever a pipe is created

Pipe Definitions

```
/* pipe.h */

/* Definitions for a pipe device */

#ifndef PIPE_BUF_SIZE
#define PIPE_BUF_SIZE   20      /* Default size of a pipe buffer        */
#endif

/* state constants for a pipe pseudo device */

#define PIPE_FREE       0       /* Entry is not currently used          */
#define PIPE_OPEN       1       /* Entry is open for reading and writing*/
#define PIPE_EOF        2       /* Entry has been closed for writes, but*/
                                /*  remains open for reading until all  */
                                /*  chars have been read and EOF has    */
                                /*  returned                            */
struct  pipecblk {
        int32   pstate;         /* State of this pipe device            */
        byte    pbuf[PIPE_BUF_SIZE]; /* Buffer for the pipe             */
        int32   phead;          /* Index of next byte in pbuf to read   */
        int32   ptail;          /* Index of next byte in pbuf to write  */
        sid32   ppsem;          /* Producer semaphore for the pipe      */
        sid32   pcsem;          /* Consumer semaphore for the pipe      */
        did32   pdevid;         /* Device ID of this pseudo device      */
        int32   pavail;         /* Available characters during drain    */
};

extern  struct  pipecblk pipetab[];
```

State Transitions For A Pipe Pseudo-Device



- The *PIPE_EOF* state indicates that one process has closed the pipe
- When a second *close* occurs, the pseudo-device returns to the *PIPE+FREE* state, meaning it has been deallocated

Synchronizing Access To The Pipe

- A Xinu pipe appears to be a classic producer-consumer situation where access can be controlled by two semaphores (i.e., producer and consumer)
- However, two special cases arise
  - Case 1: the normal case in which the producer process calls *close* before all data has been read
  - Case 2: an abnormal case there the receiving process is terminated before the producer process finishes sending data (if the buffer is full, the producer may be blocked waiting for space in the buffer)

Handling Special Cases

- Each function that deposits or extracts data from a pipe checks the state of the pipe when it begins
- The pipe functions cannot use the count of a semaphore to specify the number of bytes in the buffer because the receiving process may make the count –1 if it blocks on an empty buffer
- So, instead of using semaphore count, the control block contains an integer, *pavail*, that counts available bytes

Keeping a count of data bytes separate from the semaphore count allows the pipe code to signal a semaphore on end-of-file without changing the count of data bytes.

Initializing A Pipe Pseudo-Device

```
/* pipe_init.c  -  pipe_init */

#include <xinu.h>

struct  pipecblk pipetab[Npip];

/*------------------------------------------------------------------------
 * pipe_init  -  initialize a pipe pseudo device
 *------------------------------------------------------------------------
 */
devcall pipe_init (
          struct dentry *devptr         /* Entry in device switch table */
        )
{
        struct  pipecblk *piptr;        /* Pointer to pipe control block*/

        piptr = &pipetab[devptr->dvminor];

        piptr->pstate = PIPE_FREE;
        piptr->pdevid = devptr->dvnum;
        return OK;
}
```

Opening The PIPE Device To Create A Pipe (Part 1)

```
/* pipeopen.c - pipeopen */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  pipeopen  -  open one of the pipe pseudo devices
 *------------------------------------------------------------------------
 */
devcall pipe_open(
          struct dentry *devptr,        /* Entry in device switch table */
          char  *buff,                  /* Not used                     */
          int32 count                   /* Not used                     */
        )
{
        struct  pipecblk *piptr;        /* Pointer to pipe control block*/
        int     i;                      /* Walks through control blocks */
```

Opening The PIPE Device To Create A Pipe (Part 2)

```
        /* Find a pipe pseudo device that is available */

        for (i=0; i<Npip; i++) {
                piptr = &pipetab[i];

                if (piptr->pstate == PIPE_FREE) {
                        break;
                }
        }

        if (i >= Npip) {
                return SYSERR;
        }

        piptr->pstate = PIPE_OPEN;
        piptr->pcsem = semcreate(0);
        piptr->ppsem = semcreate(PIPE_BUF_SIZE);
        piptr->phead = piptr->ptail = 0;
        piptr->pavail= 0;
        return piptr->pdevid;
}
```

Extracting Data From A Pipe

- The special cases make extracting data surprisingly complex
- As an example, consider *pipe_getc* that must use the state of the pipe to decide how to extract the next byte
- If the pipe is in state PIPE_EOF, the code checks *pavail* to determine whether bytes remain in the buffer
- A separate section of code handles the case where the pipe is open

Extracting A Byte From A Pipe (pipe_getc.c Part 1)

```
/* pipe_getc.c - pipe_getc */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  pipe_getc  -  read one character from a pipe device
 *------------------------------------------------------------------------
 */
devcall pipe_getc(
          struct dentry *devptr         /* Entry in device switch table */
        )
{
        char    ch;                     /* Byte of data from the buffer */
        struct  pipecblk *piptr;        /* Pointer to pipe control block*/

        /* Get a pointer to the control block for this pipe */

        piptr = &pipetab[devptr->dvminor];

        /* Check if pipe is not in use or at EOF */

        if (piptr->pstate == PIPE_FREE) {
                /* The pipe is not available */
                return SYSERR;
        }
```

Extracting A Byte From A Pipe (pipe_getc.c Part 2)

```
if (piptr->pstate == PIPE_EOF) {

        /* The writer closed the pipe, so return bytes while    */
        /*  any remain in the buffer.                           */

        if (piptr->pavail > 0) {
                ch = piptr->pbuf[piptr->phead++];
                if (piptr->phead >= PIPE_BUF_SIZE) {
                        piptr->phead = 0;
                }
                piptr->pavail--;
                return 0xff & ch;
        }
        return EOF;
}

/* State is OPEN -- Wait for a byte to be available or a close  */

wait(piptr->pcsem);
```

Extracting A Byte From A Pipe (pipe_getc.c Part 3)

```
        /* If the state changed while we were blocked, the producer must*/
        /*  have called close, possibly after writing bytes to the pipe.*/

        if (piptr->pstate == PIPE_EOF) {
                if (piptr->pavail <= 0) {
                        /* The buffer is empty */
                        return EOF;
                }
        }
        /* A byte is available to be read -- pick up and return the byte*/

        ch = piptr->pbuf[piptr->phead++];
        if (piptr->phead >= PIPE_BUF_SIZE) {
                piptr->phead = 0;
        }
        piptr->pavail--;

        /* Signal the producer and return the byte */

        signal(piptr->ppsem);
        return 0xff & ch;
}
```

Depositing A Byte In A Pipe

- As usual, a sending process must wait on the producer semaphore, deposit a byte, and then signal the consumer semaphore
- A special case arises if the receiving process closes the pipe while the sending process is waiting
- Look at the code in *pipe_putc* to see how it checks the state

Depositing A Byte From A Pipe (pipe_putc.c Part 1)

```
/* pipeputc.c - pipeputc */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  pipeputc  -  write one character to a pipe device
 *------------------------------------------------------------------------
 */
devcall pipe_putc(
          struct dentry *devptr,        /* Entry in device switch table */
          char  ch                      /* Byte to write                */
        )
{
        struct  pipecblk *piptr;        /* Pointer to pipe control block*/

        /* Get a pointer to the control block for this pipe */

        piptr = &pipetab[devptr->dvminor];

        /* Check that the pipe is available for writing */

        if (piptr->pstate != PIPE_OPEN) {
                return SYSERR;
        }
```

Depositing A Byte From A Pipe (pipe_putc.c Part 2)

```
        wait(piptr->ppsem);

        /* See if pipe was closed or set to EOF while we were blocked   */

        if (piptr->pstate != PIPE_OPEN) {
                return SYSERR;
        }

        /* Deposit a byte in next buffer position */

        piptr->pbuf[piptr->ptail++] = ch;
        if (piptr->ptail >= PIPE_BUF_SIZE) {
                piptr->ptail = 0;
        }
        piptr->pavail++;

        /* Signal the consumer semaphore and return */

        signal(piptr->pcsem);
        return OK;
}
```

Closing A Pipe

- Recall that *close* will be called twice
- The code uses the state to determine the appropriate action
- During the first close, the code resets the producer semaphore
- To ensure it finishes resetting semaphores before any context switch occurs, the code defers rescheduling while making changes

Closing A Pipe (pipe_close.c Part 1)

```
/* pipe_close.c  -  pipe_close */
#include <xinu.h>
/*------------------------------------------------------------------------
 * pipe_close  -  Close a pipe
 *------------------------------------------------------------------------
*/
devcall pipe_close (
          struct dentry *devptr         /* Entry in device switch table */
        )
{
        struct  pipecblk *piptr;        /* Pointer to pipe control block*/

        /* Note: because both a writing process and reading process use */
        /*  a given pipe, both will close the pipe.  Conceptually, the  */
        /*  first call moves the pipe to a read-only state and marks the*/
        /*  end-of-ile. The second call deallocates the pipe device,    */
        /*  making it available for reuse.                              */

        piptr = &pipetab[devptr->dvminor];

        /* If pipe is completely closed, return SYSERR */

        if (piptr->pstate == PIPE_FREE) {
                return SYSERR;
        }
```

Closing A Pipe (pipe_close.c Part 2)

```
/* First call to close -- move to EOF state */

if (piptr->pstate == PIPE_OPEN) {
        piptr->pstate = PIPE_EOF;
        resched_cntl(DEFER_START);
        if (semcount(piptr->pcsem) < 0) {
                /* Pipe is empty and consumer is blocked, so    */
                /* Allow the consumer to run                    */
                semreset(piptr->pcsem, 0);
        }

        /* Allow a blocked producer to proceed, if any */

        if (semcount(piptr->ppsem) < 0) {
                semreset(piptr->ppsem, 0);
        }
        resched_cntl(DEFER_STOP);
        return OK;
}
```

Closing A Pipe (pipe_close.c Part 3)

```
        /* Second call to close - deallocate the pipe device */

        piptr->pstate = PIPE_FREE;
        semdelete(piptr->ppsem);
        semdelete(piptr->pcsem);
        piptr->pavail = 0;
        return OK;
}
```

Questions?