**Cheat Sheet:**

Exam 1 Part 2

**Fork()**

# fork() in C:

Fork system call use for creates a new process, which is called ***child process***, which runs concurrently with process (which process called system call fork) and this process is called ***parent process***. After a new child process created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.
It takes no parameters and returns an integer value. Below are different values returned by fork().

***Negative Value***: creation of a child process was unsuccessful.
***Zero***: Returned to the newly created child process.
***Positive value***: Returned to parent or caller. The value contains process ID of newly created child process

## Pipe()

When one end of a pipe is closed, two rules apply:

1. If we read from a pipe whose write end has been closed, read returns 0 to indicate an end of file after all the data has been read.
2. If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated. If we either ignore the signal or catch it and return from the signal handler, write returns −1 with errno set to EPIPE.

**Example**

Figure 15.5 shows the code to create a pipe between a parent and its child and to send data down the pipe.

```
#include "apue.h"

int
main(void)
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {       /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {                    /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

**Figure 15.5** Send data from parent to child over a pipe

In the previous example, we called read and write directly on the pipe descriptors. What is more interesting is to duplicate the pipe descriptors onto standard input or standard output. Often, the child then runs some other program, and that program can either read from its standard input (the pipe that we created) or write to its standard output (the pipe). In the previous example, we called read and write directly on the pipe descriptors. What is more interesting is to duplicate the pipe descriptors onto standard input or standard output. Often, the child then runs some other program, and that program can either read from its standard input (the pipe that we created) or write to its standard output (the pipe).

Consider a program that displays some output that it has created, one page at a time. Rather than reinvent the pagination done by several UNIX system utilities, we want to invoke the user's favorite pager. To avoid writing all the data to a temporary file and calling system to display that file, we want to pipe the output directly to the pager. To do this, we create a pipe, fork a child process, set up the child's standard input to be the read end of the pipe, and exec the user's pager program. Figure 15.6 shows how to do this

Before calling fork, we create a pipe. After the fork, the parent closes its read end, and the child closes its write end. The child then calls dup2 to have its standard input be the read end of the pipe. When the pager program is executed, its standard input will be the read end of the pipe. When we duplicate one descriptor onto another (fd[0] onto standard input in the child), we have to be careful that the descriptor doesn't already have the desired value. If the descriptor already had the desired value and we called dup2 and close, the single copy of the descriptor would be closed. (Recall the operation of dup2 when its two arguments are equal, discussed in Section 3.12.) In this program, if standard input had not been opened by the shell, the fopen at the beginning of the program should have used descriptor 0, the lowest unused descriptor, so fd[0] should never equal standard input. Nevertheless, whenever we call dup2 and close to duplicate one descriptor onto another, we'll always compare the descriptors first, as a defensive programming measure. Note how we try to use the environment variable PAGER to obtain the name of the user 's pager program. If this doesn't work, we use a default. This is a common usage of environment variables

```
#include "apue.h"
#include <sys/wait.h>

#define DEF_PAGER   "/bin/more"     /* default pager program */

int
main(int argc, char *argv[])
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    *pager, *argv0;
    char    line[MAXLINE];
    FILE    *fp;

    if (argc != 2)
        err_quit("usage: a.out <pathname>");

    if ((fp = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);
    if (pipe(fd) < 0)
        err_sys("pipe error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {                                   /* parent */
        close(fd[0]);       /* close read end */

        /* parent copies argv[1] to pipe */
        while (fgets(line, MAXLINE, fp) != NULL) {
            n = strlen(line);
            if (write(fd[1], line, n) != n)
                err_sys("write error to pipe");
        }
        if (ferror(fp))
            err_sys("fgets error");

        close(fd[1]);   /* close write end of pipe for reader */

        if (waitpid(pid, NULL, 0) < 0)
            err_sys("waitpid error");
```

Fig 15.6



```
        exit(0);
    } else {                                            /* child */
        close(fd[1]);   /* close write end */
        if (fd[0] != STDIN_FILENO) {
            if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
            close(fd[0]);   /* don't need this after dup2 */
        }

        /* get arguments for execl() */
        if ((pager = getenv("PAGER")) == NULL)
            pager = DEF_PAGER;
        if ((argv0 = strrchr(pager, '/')) != NULL)
            argv0++;        /* step past rightmost slash */
        else
            argv0 = pager;  /* no slash in pager */

        if (execl(pager, argv0, (char *)0) < 0)
            err_sys("execl error for %s", pager);
    }
    exit(0);
```

To use IPC mechanism of pipe, you declare a pair of integers as two "file descriptors." You then call pipe() on these two integers to create a pipe (which resides inside the kernel) and to bind the two descriptord to the pipe.

A pipe within a single process (Fig. 15.2) is of little value. So, often time, pipe() is used together with fork() to establish inter-process communication between two (parent and child) processes.

What happens after the fork() depends on which direction of data flow we want. 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]).

One very important feature of reading from pipe is that it is a blocking read. That is if there is no message in the pipe, calling read() on a pipe will block until a message becomes available.

Check out the following program adapted from apue.3e/ipc1/pipe1.c (Fig. 15.5). The parent waits for 15 seconds to write to the child, which keeps waiting. Iff you could use "ps -ef | grep <your login name" to validate the PIDs of the two processes.

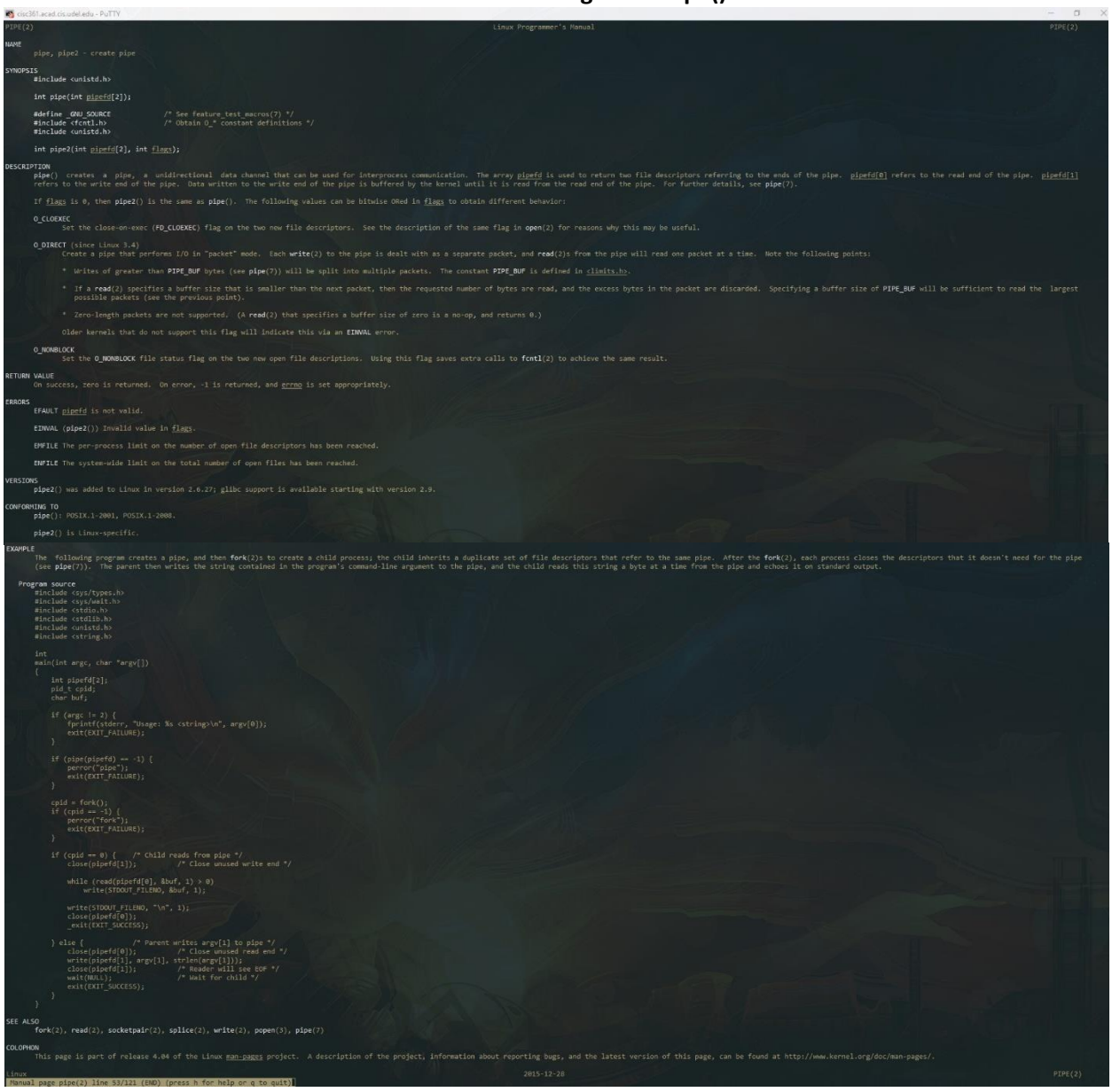
```
// mypipe.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#define MAXLINE 128

void main()
{
  int     n;
  int     fd[2];
  pid_t   pid;
  char    line[MAXLINE];

  if (pipe(fd) < 0) {
    printf("pipe error\n");
    exit(1);
  }
  if ((pid = fork()) < 0) {
    printf("fork error\n");
    exit(1);
  }
  else
    if (pid > 0) {           /* parent */
      close(fd[0]);
      printf("Parent PID (%d)...\n", getpid());
      sleep(15);             /* wait for 15 sec to write */
      write(fd[1], "hello world\n", 12);
    } else {                 /* child */
        close(fd[1]);
        printf("Child (%d) is ready to read...\n", getpid());
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
}
```

# Man Pages For Pipe()



```
cisc361.acad.cis.udel.edu - PuTTY

PIPE(2)                                  Linux Programmer's Manual                                  PIPE(2)

NAME
       pipe, pipe2 - create pipe

SYNOPSIS
       #include <unistd.h>

       int pipe(int pipefd[2]);

       #define _GNU_SOURCE             /* See feature_test_macros(7) */
       #include <fcntl.h>              /* Obtain O_* constant definitions */
       #include <unistd.h>

       int pipe2(int pipefd[2], int flags);

DESCRIPTION
       pipe()  creates  a  pipe,  a  unidirectional  data channel that can be used for interprocess communication.  The array pipefd is used to return two file descriptors referring to the ends of the pipe.  pipefd[0] refers to the read end of the pipe.  pipefd[1]
       refers to the write end of the pipe.  Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.  For further details, see pipe(7).

       If flags is 0, then pipe2() is the same as pipe().  The following values can be bitwise ORed in flags to obtain different behavior:

       O_CLOEXEC
              Set the close-on-exec (FD_CLOEXEC) flag on the two new file descriptors.  See the description of the same flag in open(2) for reasons why this may be useful.

       O_DIRECT (since Linux 3.4)
              Create a pipe that performs I/O in "packet" mode.  Each write(2) to the pipe is dealt with as a separate packet, and read(2)s from the pipe will read one packet at a time.  Note the following points:

              *  Writes of greater than PIPE_BUF bytes (see pipe(7)) will be split into multiple packets.  The constant PIPE_BUF is defined in limits.h.

              *  If a read(2) specifies a buffer size that is smaller than the next packet, then the requested number of bytes are read, and the excess bytes in the packet are discarded.  Specifying a buffer size of PIPE_BUF will be sufficient to read the  largest
                 possible packets (see the previous point).

              *  Zero-length packets are not supported.  (A read(2) that specifies a buffer size of zero is a no-op, and returns 0.)

              Older kernels that do not support this flag will indicate this via an EINVAL error.

       O_NONBLOCK
              Set the O_NONBLOCK file status flag on the two new open file descriptions.  Using this flag saves extra calls to fcntl(2) to achieve the same result.

RETURN VALUE
       On success, zero is returned.  On error, -1 is returned, and errno is set appropriately.

ERRORS
       EFAULT pipefd is not valid.

       EINVAL (pipe2()) Invalid value in flags.

       EMFILE The per-process limit on the number of open file descriptors has been reached.

       ENFILE The system-wide limit on the total number of open files has been reached.

VERSIONS
       pipe2() was added to Linux in version 2.6.27; glibc support is available starting with version 2.9.

CONFORMING TO
       pipe(): POSIX.1-2001, POSIX.1-2008.

       pipe2() is Linux-specific.

EXAMPLE
       The  following program creates a pipe, and then fork(2)s to create a child process; the child inherits a duplicate set of file descriptors that refer to the same pipe.  After the fork(2), each process closes the descriptors that it doesn't need for the pipe
       (see pipe(7)).  The parent then writes the string contained in the program's command-line argument to the pipe, and the child reads this string a byte at a time from the pipe and echoes it on standard output.

   Program source
       #include <sys/types.h>
       #include <sys/wait.h>
       #include <stdio.h>
       #include <stdlib.h>
       #include <unistd.h>
       #include <string.h>

       int
       main(int argc, char *argv[])
       {
           int pipefd[2];
           pid_t cpid;
           char buf;

           if (argc != 2) {
               fprintf(stderr, "Usage: %s <string>\n", argv[0]);
               exit(EXIT_FAILURE);
           }

           if (pipe(pipefd) == -1) {
               perror("pipe");
               exit(EXIT_FAILURE);
           }

           cpid = fork();
           if (cpid == -1) {
               perror("fork");
               exit(EXIT_FAILURE);
           }

           if (cpid == 0) {    /* Child reads from pipe */
               close(pipefd[1]);          /* Close unused write end */

               while (read(pipefd[0], &buf, 1) > 0)
                   write(STDOUT_FILENO, &buf, 1);

               write(STDOUT_FILENO, "\n", 1);
               close(pipefd[0]);
               _exit(EXIT_SUCCESS);

           } else {            /* Parent writes argv[1] to pipe */
               close(pipefd[0]);          /* Close unused read end */
               write(pipefd[1], argv[1], strlen(argv[1]));
               close(pipefd[1]);          /* Reader will see EOF */
               wait(NULL);                /* Wait for child */
               exit(EXIT_SUCCESS);
           }
       }

SEE ALSO
       fork(2), read(2), socketpair(2), splice(2), write(2), popen(3), pipe(7)

COLOPHON
       This page is part of release 4.04 of the Linux man-pages project.  A description of the project, information about reporting bugs, and the latest version of this page, can be found at http://www.kernel.org/doc/man-pages/.

Linux                                      2015-12-28                                      PIPE(2)
(Manual page pipe(2) line 53/121 (END) (press h for help or q to quit)
```