Augustana Campus, University of Alberta
AUCSC 380 Operating System Concepts
Winter Term, 2019

Programming Assignment 3
Pipes, Processes, and Paging

Due date: Friday, 2019 March 29, by midnight.

Objectives:

- To become familiar with the use of UNIX pipes for inter-process communication.
- To implement a simulation using multiple communicating processes.
- To understand the implementation of paging and page replacement algorithms.

Assignment:

Write a C program that simulates a simple paged-memory system.

The program should create three child processes (using `fork`), each of which simulates a different user process. The parent process simulates the paging system of an operating system. The child processes communicate with the parent process through pipes (use `pipe` and `fcntl`). Each child reads its memory reference string from a file and requests page translation by the parent process. The parent process polls the child processes in circular order (123123123...), responding to any request for page translation that is available or going on to the next child if the pipe from the current child is empty. The parent keeps track of which page is in each frame, and returns to the child the frame number that corresponds to the page number sent by the child. If the page requested is not currently in memory, the parent process services the page fault by finding an empty frame or choosing a victim frame and allocating it to the page requested, returning to the child the number of the frame chosen.

The system to be simulated is very simple. There are only 12 frames of physical memory, four of which are allocated to each of the three child processes. Rather than using a page table for each process (which would require knowledge as to the length of each process's logical address space), the frames allocated to the requesting process may be scanned to see if the requested page is currently in any of them. If so, the number of the frame containing that page is returned to the child process through the pipe connecting it with the parent process, and the page reference is counted. If the desired page is not present, the victim page is chosen by a page replacement algorithm (see below) and the number of the requested page replaces the number of the page which currently resides in the chosen frame. The page fault is counted and the frame number is returned to the child process.

The program takes three parameters, specifying the names of the files containing the memory reference strings (sequences of page numbers) for the three child processes. The filename will either specify a file in the current directory or be a complete pathname. If a file cannot be opened for reading, an error message should be written to the standard error file and the program should exit with the EXIT_FAILURE status. Each file will consist of a series of newline-delimited non-negative integers (representing page numbers). For simplicity, we will assume that no illegal page references are generated (i.e., that all integers in the file represent valid pages).

The page replacement algorithm should be either FIFO or LRU. Clearly indicate which algorithm you selected, both in the source code and in the output (see below). You should be sure to design the program in such a way that you can easily swap another page replacement algorithm for the one selected. If you wish, you may include both algorithms and select between them with a command-line option ('`-f`' for FIFO, '`-l`' for LRU). The parent process should print a start-up message indicating which page replacement algorithm it is using.

When a child process has read (and requested corresponding frame numbers for) all the page numbers in the memory reference string file, it prints a message indicating that it has terminated execution. The parent process should then print the filename of the memory reference string file for the terminating child, the total number of page references handled, the number of page faults, the page fault rate (in faults per references), and the numbers of the (up to) four pages currently in (simulated) physical memory (in order by frame number).

When all child processes have terminated, the parent process should print the total number of page references handled, the total number of page faults, and the average page fault rate for all three processes.

Your program should include appropriate error-checking (i.e., check the return value from all system calls, etc.) Error messages due to system errors should include the system-generated error message using the `perror` subroutine. Be sure to check that each system call was successful, and use `perror` to display a useful error message prior to terminating if a system call failed.

Hints and References:

See the discussion of `pipe` in *Beginning Linux Programming*, 4e, pp. 525–526, 531–534, and of `fcntl`, p. 132.

Normally, an attempt to read from an empty pipe will *block* until data becomes available. In order that the parent process can poll the child processes in circular order, the end of the pipe that the parent process will read has to be specified as non-blocking. Subsequent calls to `read` on the read end of an empty pipe will then return a value of -1 immediately and set `errno` to `EAGAIN`. (A return value of -1 with a different error number indicates that the read failed; a return value of 0 means the write end of the pipe has been closed and the pipe is empty.)

The `fcntl` system call can turn on the `O_NONBLOCK` file status flag for a descriptor that is already open (e.g., due to a call to `pipe`). However, one can't just call `fcntl` with the `F_SETFL` command with `O_NONBLOCK` as the flag argument, because this might turn off flag bits that were previously set. Instead, the flags for the relevant file descriptor must first be retrieved with `fcntl`'s `F_GETFL` command argument. Then `O_NONBLOCK` can be added to the existing flags with a binary OR and `fcntl` can be called again with the `F_SETFL` command and the augmented flags as the flag argument. The following program segment illustrates this technique. (See the section "Enumerated Types" in *C in a Nutshell* [p. 29 in the first edition] for an explanation of the use of enum to define integer constants. See Programming Assignment 2 regarding the function `fatal_sys`.)

```
enum { READ = 0, WRITE = 1 };
int pfd[2];
int flags;
    :
```

Call `pipe` *with* `pfd` *as argument.*

```
    :
if ((flags = fcntl(pfd[READ], F_GETFL)) < 0)
    fatal_sys("Error with fcntl get flags call");

flags |= O_NONBLOCK;

if (fcntl(pfd[READ], F_SETFL, flags) < 0)
    fatal_sys("Error with fcntl set flags call");
```

Grading:

The weightings of the various factors for grading will be approximately as follows:

Correctness	60%
Design and Efficiency	20%
Style and Documentation	20%

Submission of Solution:

Submit the C source code for your program (or for multiple programs, if your main program uses `exec`), along with any related header file(s), via *eClass* by midnight on the due date specified. Assignments will be accepted after the due date and time, but late submissions will be assessed a penalty of **1% per hour** or portion of an hour.