CS39002: Operating Systems Lab
Spring 2012

Assignment 3
Due: February 02, 2012, 1 pm

Part 1: A Simple Producer-Consumer Application

In this program, you will write a program to solve the *m*-producer *n*-consumer problem for the specific values of $m = 1$ and $n = 2$. You have a shared circular buffer that can hold 20 integers. The producer process stores the numbers 1 to 50 in the buffer one by one (in a for loop with 50 iterations) and then exits. Each of the consumer processes reads the numbers from the buffer and adds them to a shared variable `SUM` (initialized to 0). Any consumer process can read any of the numbers in the buffer. The only constrain is that every number written by some producer should be read exactly once by exactly one of the consumers. Of course, a producer should not write when the buffer is full, and a consumer should not read when the buffer is empty.

Write a program that first creates the shared circular buffer and the shared variable `SUM` using the `shm*()` calls in Linux. You can create any other shared variable that you think you may need. The program then forks the producers and the two consumers. The producer and consumer codes can be written as functions that are called by the child processes. After the producer and both the consumers have finished (the consumers exit after all the data produced by all the producers have been read. How does a consumer know this?), the parent process prints the value of `SUM`. Note that the value of `SUM` should be $25 \times 51 = 1275$ if your program is correct.

Name your program `ProdCons.c`. Submit only the C file.

Part 2: A Multi-process Image-processing Application

Write a program to interactively carry out some simple processing of images. The program (the parent process) first creates two segments of shared memory for storing the input and the output images, and also a third shared-memory segment for communication with its child processes. Let us call these segments the input, the output and the control segments. The parent process also creates a set of semaphores for synchronization. The parent process then forks four child processes and enters a loop described below.

In each iteration of the loop, the user enters an input file name, an output filename (both with extension `.ppm` or `.pnm`), and an integer value in the range [–100, 100] indicating the requested percentage of change in the brightness of the input image. The parent process then reads the input file from the disk and stores the image data in the input segment. Meanwhile, the child processes wait on appropriate semaphores. When writing the data and associated instructions in the control segment is completed by the parent, it wakes up the child processes, each of which works on one-fourth of the input image and writes the modified portion of the image in the output segment (how does a child process know which quarter of the image it has to handle?). When the child processes work on the image, the parent process waits on a semaphore. After a child process finishes, it signals the parent process and again goes to a wait state. When the parent receives signals from all the four child processes, it wakes up and writes the output image in the output file whose name was provided by the user. This completes one iteration of the loop.

In the next iteration, another image is processed similarly by the *same* child processes. The user enters `exit` to indicate that no further image needs to be processed. The parent then passes a termination instruction to the child processes via the control segment. When all of the four child processes exit, the parent deletes the shared memory segments and the semaphores created at the beginning, and exits itself.

Note that all interactions between the parent processes and the child processes must use the control segment. No other types of inter-process communication (like pipes) are allowed.

Changing the contrast of an image can be equally easily carried out. As an *optional* part in this assignment, you may write a similar multi-process code for changing the contrast of the input image. This means that the user may now supply a third type of request (in addition to brightness change and exit) for contrast changing. Such a request is handled by the four child processes exactly in the same way in which brightness changes are handled.

Assume a maximum image size that the program can handle (like 1024 × 768). The parent process neglects wider or taller images. A configuration file (usually `/proc/sys/kernel/shmmax`) stores an integer indicating the maximum allowed size in bytes of a shared-memory segment. In my machine, this size is $2^{25} = 33554432$. Three two-dimensional integer arrays each of size 1024 × 768 comfortably fit in this space. Since pixel components are in the range [0, 255], you can store each pixel in three `unsigned char` variables, reducing the storage requirement further. Still, if you are unable to allocate large shared-memory segments, work with smaller image sizes.

Name your program `ImageProc.c`. Submit only the C file.

SUPPLEMENTARY MATERIAL

A tutorial on pnm images

A popular format to represent images is by a two-dimensional array of *pixel* values. Each pixel is an instance of color (a dot on the screen). A black/white pixel is represented by the value 0 (black) or 1 (white). A gray-scale pixel is typically an integer value between 0 and 255 standing for different shades of gray. Finally, a colored pixel is a triple (r, g, b) of three integer values r, g and b each in the range 0 to 255. In this assignment, we plan to work with color images.

Image files are stored in several formats (like gif, jpeg, bmp). These formats differ in various aspects including the maximum number of allowed colors, the type of compression used, and so on. Here, we work with the somewhat uncommon format known as *pnm* (portable any map). It uses no compression and stores the pixel values in a human-readable (ASCII) form.

A pnm file starts with one of the signatures `P1` through `P6`. `P1` means black-and-white images, `P2` means gray-scale images, and `P3` means color images. In this assignment, we will not deal with pnm files with signatures `P4`, `P5` and `P6`. After this signature, come the dimensions of the image: first the horizontal dimension (width) and then the vertical dimension (height). The next item is a maximum level which you can assume to be 255 for this assignment. Then follows a listing of the pixel values of the image in the row-major order.

In a color image, each pixel is a triple (*r*, *g*, *b*) of integers with each component in the range 0 to 255. For example, `0,0,0` means black, `255,255,255` means white, `255,0,0` means red, `0,255,255` means cyan, `165,42,42` means brown, and so on. A color pnm file can be stored with the extension `.pnm` or `.ppm`.

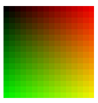
A line starting with a # indicates a comment in a pnm file. For simplicity, assume that the input file does not contain any comment. If you prepare your input file from some image-processing utility, manually delete all the comments in the images produced.

A 16 x 16 color image in the pnm format is shown below.

```
P3
16 16
255
0 0 0 16 0 0 32 0 0 48 0 0 64 0 0 80 0 0 96 0 0 112 0 0 128 0 0 144 0
0 160 0 0 176 0 0 192 0 0 208 0 0 224 0 0 240 0 0
0 16 0 16 16 0 32 16 0 48 16 0 64 16 0 80 16 0 96 16 0 112 16 0 128 16
0 144 16 0 160 16 0 176 16 0 192 16 0 208 16 0 224 16 0 240 16 0
0 32 0 16 32 0 32 32 0 48 32 0 64 32 0 80 32 0 96 32 0 112 32 0 128 32
0 144 32 0 160 32 0 176 32 0 192 32 0 208 32 0 224 32 0 240 32 0
0 48 0 16 48 0 32 48 0 48 48 0 64 48 0 80 48 0 96 48 0 112 48 0 128 48
0 144 48 0 160 48 0 176 48 0 192 48 0 208 48 0 224 48 0 240 48 0
0 64 0 16 64 0 32 64 0 48 64 0 64 64 0 80 64 0 96 64 0 112 64 0 128 64
0 144 64 0 160 64 0 176 64 0 192 64 0 208 64 0 224 64 0 240 64 0
0 80 0 16 80 0 32 80 0 48 80 0 64 80 0 80 80 0 96 80 0 112 80 0 128 80
0 144 80 0 160 80 0 176 80 0 192 80 0 208 80 0 224 80 0 240 80 0
0 96 0 16 96 0 32 96 0 48 96 0 64 96 0 80 96 0 96 96 0 112 96 0 128 96
0 144 96 0 160 96 0 176 96 0 192 96 0 208 96 0 224 96 0 240 96 0
0 112 0 16 112 0 32 112 0 48 112 0 64 112 0 80 112 0 96 112 0 112 112
0 128 112 0 144 112 0 160 112 0 176 112 0 192 112 0 208 112 0 224 112
0 240 112 0
0 128 0 16 128 0 32 128 0 48 128 0 64 128 0 80 128 0 96 128 0 112 128
0 128 128 0 144 128 0 160 128 0 176 128 0 192 128 0 208 128 0 224 128
0 240 128 0
0 144 0 16 144 0 32 144 0 48 144 0 64 144 0 80 144 0 96 144 0 112 144
0 128 144 0 144 144 0 160 144 0 176 144 0 192 144 0 208 144 0 224 144
0 240 144 0
0 160 0 16 160 0 32 160 0 48 160 0 64 160 0 80 160 0 96 160 0 112 160
0 128 160 0 144 160 0 160 160 0 176 160 0 192 160 0 208 160 0 224 160
0 240 160 0
0 176 0 16 176 0 32 176 0 48 176 0 64 176 0 80 176 0 96 176 0 112 176
0 128 176 0 144 176 0 160 176 0 176 176 0 192 176 0 208 176 0 224 176
0 240 176 0
0 192 0 16 192 0 32 192 0 48 192 0 64 192 0 80 192 0 96 192 0 112 192
0 128 192 0 144 192 0 160 192 0 176 192 0 192 192 0 208 192 0 224 192
0 240 192 0
0 208 0 16 208 0 32 208 0 48 208 0 64 208 0 80 208 0 96 208 0 112 208
0 128 208 0 144 208 0 160 208 0 176 208 0 192 208 0 208 208 0 224 208
0 240 208 0
0 224 0 16 224 0 32 224 0 48 224 0 64 224 0 80 224 0 96 224 0 112 224
```

```
0 128 224 0 144 224 0 160 224 0 176 224 0 192 224 0 208 224 0 224 224
0 240 224 0
0 240 0 16 240 0 32 240 0 48 240 0 64 240 0 80 240 0 96 240 0 112 240
0 128 240 0 144 240 0 160 240 0 176 240 0 192 240 0 208 240 0 224 240
0 240 240 0
```

Store the above lines in a file (like `colorimage.pnm`), and view that file using an image viewer. You should see an image like this.



A simple algorithm to change the brightness of an image

Darkening a pixel (r, g, b) by p% (assume that $p > 0$) means replacing this pixel by the new pixel

$$(r - [0.01pr], g - [0.01pg], b - [0.01pb]) ,$$

where [ ] stands for the round-to-the-nearest-integer function. Likewise, brightening a pixel (r, g, b) by p% replaces the pixel by the new pixel

$$(r + [0.01p(255 - r)], g + [0.01p(255 - g)], b + [0.01p(255 - b)]) .$$

A simple algorithm to change the contrast of an image

Increasing the contrast may be implemented by choosing a reference pixel like (127, 127, 127) and shifting pixel values away from this reference. The larger is the distance of a pixel from the reference, the higher is the amount of shift. A p% increase in contrast replaces the pixel (r, g, b) by

$$(r + [0.01p(r - 127)], g + [0.01p(g - 127)], b + [0.01p(b - 127)]) ,$$

whereas decreasing the contrast of the pixel (r, g, b) by p% indicates the new pixel value

$$(r - [0.01p(r - 127)], g - [0.01p(g - 127)], b - [0.01p(b - 127)]) .$$

Note: Using these formulas and/or floating-point calculations may lead to pixel components outside the allowed range [0, 255]. Any component larger than 255 is to be treated as 255. Likewise, any negative component is to be treated as 0.