Operating Systems

Homework Assignment #2

Due 15.12.2016, 23:55

In this assignment, we learn how two programs can communicate, i.e., how one process (writer) transfers a block of data to another process (reader). Two methods were mentioned in class – memory mapped files and named pipes. The objective is to apply these methods to implement basic data exchange.

In addition, you are required to implement a time measuring benchmark comparing the two methods of inter-process data transfer (IPC), via a memory mapped file and via a named pipe. Two versions of reader-writer pairs shall be implemented and measured.

Time Measurement

For the benchmarks, we will measure the running time of certain parts in the code. The measurements will be in microseconds, and we will use it to compare the two benchmarks.

See Appendix A for a code example of time measurement in Linux.

Submission:

- Follow all submission guidelines!
- Make sure filenames are correct, case-sensitive, zip structure is correct (no folders and extra files at all, including MAC hidden folders and Windows hidden files)
- Submit an actual zip file created using the *zip* command **only!**
- **Submit 4 files:** `mmap_reader.c`, `mmap_writer.c`, `fifo_reader.c`, `fifo_writer.c`

Memory Mapped File

In this section, you will implement two programs that communicate via a memory mapped file.

Hint: study the memory mapped file and signal handler examples in Moodle. Check Appendix B for invocation examples and printout formats.

mmap_writer

The writer part of the memory mapped file benchmark should be implemented as follows.

It receives 2 command-line arguments:

1. NUM - Number of bytes to transfer (assume a positive integer).
2. RPID - Process ID of an *already running* `mmap_reader` (assume a positive integer).

The implementation should follow these steps:

1. Create a file under the `/tmp` directory named `mmapped.bin` (use constants and not hard-coded values!)
2. Set the file permissions to *0600* (*man 2 chmod*)
3. Create a memory map for the file
4. Start the time measurement
5. Fill the array with NUM-1 sequential `'a'` bytes and then `NULL` (i.e., `'\0'`)
6. Send a signal (SIGUSR1) to the reader process (*man 2 kill*)
7. Print the measurement result together with the number of bytes written
8. Cleanup. Exit gracefully

mmap_reader

The reader part of the memory mapped file benchmark should be implemented as follows.

It receives **no** command-line arguments.

The implementation should follow these steps:

1. Register a signal handler for SIGUSR1 (*man 2 sigaction*)
2. Enter an infinite loop, sleeping 2 seconds in each iteration

Upon receiving a SIGUSR1 signal:

1. Open the file `/tmp/mmapped.bin`
2. Determine the file size (*man 2 lseek*, *man 2 stat*)
3. Start the time measurement
4. Create a memory map for the file
5. Count the number of `'a'` bytes in the array until the first `NULL` (`'\0'`)
6. Finish the time measurement
7. Print the measurement result along with the number of bytes counted
8. Remove the file from the disk (*man 2 unlink*)
9. Cleanup. Exit gracefully (*man 3 exit*)

Named Pipes

In this section, you will implement two programs that communicate through a named pipe.

Check Appendix B for invocation examples and printout formats.

fifo_writer

Command-line arguments:

1. NUM – number of bytes to transfer (assume a positive integer)

The flow:

1. Create a named pipe file (*man 3 mkfifo*) under `/tmp/osfifo`, with 0600 file permissions
2. Open the created file for writing
3. Start the time measurement
4. Write NUM 'a' bytes to this named pipe file
5. Finish the time measurement
6. Print the measurement result along with the number of bytes written
7. Remove the file from the disk (*man 2 unlink*)
8. Cleanup. Exit gracefully

fifo_reader

It receives **no** command-line arguments.

The flow:

1. Open `/tmp/osfifo` for reading
2. Start the time measurement
3. Read data and count the number of 'a' bytes read
4. Finish the time measurement
5. Print the measurement result along with the number of bytes read
6. Cleanup. Exit gracefully

Signal Handling

In addition to the instructions above, you will use signal handlers to handle certain edge cases that might arise during the operation of your implementations.

Can you create a situation in which the *FIFO writer* performs a `write()` without a *FIFO reader* holding the FIFO file open? What happens in this situation?

Recreate this situation. Then, read about `SIGPIPE` (*man 7 pipe*) and handle the issue.

You should cleanup and exit gracefully in this case.

The reader and writer processes can terminate in the middle of the operation, if a SIGINT or SIGTERM signal is received.
For the `mmap` implementations, ignore the `SIGTERM` signal during the entire operation.
For the `FIFO` implementations, ignore the `SIGINT` signal during the entire operation.

In all implementations, when finished (i.e., during cleanup), you should *restore* the signal handlers to whatever they were before ignoring them.

Appendix A

Time measurement (in microseconds) is demonstrated in the code below.

```c
#include <sys/time.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
  // Time measurement structures
  struct timeval t1, t2;

  double elapsed_microsec;

  // Start time measuring (CAN FAIL! CHECK RETURN VALUE!)
  gettimeofday(&t1, NULL);

  // Some hard processing
  printf("Simulating some work\n");
  sleep(2);

  // Finish time measuring (CAN FAIL! CHECK RETURN VALUE!)
  gettimeofday(&t2, NULL);

  // Counting time elapsed
  elapsed_microsec = (t2.tv_sec - t1.tv_sec) * 1000.0;
  elapsed_microsec += (t2.tv_usec - t1.tv_usec) / 1000.0;

  // Final report
  printf("%f microseconds passed\n", elapsed_microsec);

  // Exit gracefully
  return 0;
}
```

An output example. Colors: typed commands – red; printouts – blue; command prompt – **$**.

```
$./time_measure
Simulating some work
2000.654000 microseconds passed
```

Appendix B

This is an example session of invocations. The order of readers and writers is important. Colors: typed commands – red; printouts – blue; command prompt – **$**.

```
$./fifo_reader & ./fifo_writer 40960000 & fg
[1] 27438
[2] 27439
./fifo_writer 40960000
40960000 were written in 462.874000 microseconds through FIFO
$40960000 were read in 465.311000 microseconds through FIFO

[1]+  Done                    ./fifo_reader
$./mmap_reader &
[1] 27461
$./mmap_writer 40960000 27461
40960000 were written in 476.369000 microseconds through MMAP
$40960000 were read in 293.401000 microseconds through MMAP

[1]+  Done                    ./mmap_reader
```