OSM Exam - Buenos

Christian Enevoldsen 020691 - MRB852

March 27, 2015

Contents

1 PRACTICAL PART **3**

1.1 P1 . . . 3

1.2 P2 . . . 4

1.3 P3 . . . 4

1.4 P4 . . . 6

2 THEORETICAL **8**

2.1 T1 . . . 8

2.1.1 Correctness . . . 9

2.2 T3 . . . 9

2.2.1 Printing . . . 9

2.2.2 Users . . . 10

A Test Programs **11**

B Test Output **18**

C Added And Modified Source to Buenos **21**

C.1 Changes To Header Files . . . 29

1 PRACTICAL PART

1.1 P1

The queue's are tested using 2 to 4 threads; That means at least 1 thread pr consumer and at least one thread pr producer. The test queries 500000 items (int pointers) to test the difference, however other tests had also been made. For instance with up to 500 threads and only 100 items the coarse grained solution was a bit faster.

The following tables shows the results of running with a lot of threads and few items

Queue	**Threads**	**Items**	**Time (seconds)**
FG	1	100	0.000290
FG	2	100	0.000646
FG	4	100	0.004228
FG	8	100	0.010164
FG	16	100	0.020787
FG	32	100	0.046518
FG	64	100	0.062837
FG	128	100	0.299638
FG	256	100	0.800857
FG	512	100	2.928489

Queue	**Threads**	**Items**	**Time (seconds)**
CG	1	100	0.000413
CG	2	100	0.001808
CG	4	100	0.003369
CG	8	100	0.007035
CG	16	100	0.014763
CG	32	100	0.036597
CG	64	100	0.081186
CG	128	100	0.184895
CG	256	100	0.599984
CG	512	100	2.189591

The tables shows that the coarse grained queue wins with many threads and few items. This is most likely the result of a overhead since the Fine grained has a lot more house keeping with the locks.

If we try with more items and fewer threads the fine grained wins big. However whenever it runs with more than 1 thread pr producer and consumer it tends to be slower than the coarse grained. That might be the locks again that doesn't work well with more than 2 threads.

Queue	**Threads**	**Items**	**Time (seconds)**
FG	1	5000000	2.397133
FG	2	5000000	103.727394

Queue	Threads	Items	Time (seconds)
CG	1	5000000	29.838345
CG	2	5000000	69.666885

To conclude, it's about trade offs, trial and errors that depends on the result as well as how many cores you have. This is tested with 4 threads on 2 cores. Let's assume it's tested on a 8 core. Then it would most likely win again with 8 threads.

To run the test yourself open the terminal and cd into the benchmark folder. In there write: make run

If your want to run with a different number of items replace the MAX definition with the new value and for threads replace 'q' in the main function. Make sure that q is a power of 2. Ie: 2, 4, 8...

The test source code can be found in Appendix A, listing 7: queue_test.c

1.2 P2

For supporting random number generation syscall_rand is implemented. This function is pretty straight forward since it is just a wrapper function that calls another function.

For testing run tests/rand.c.

Changed source code can be found in Appendix C, and test output can be found in the appendix B. To run the test copy the rand.c to a fyams harddisk and set initprogram to it.

1.3 P3

Implementing pipes in buenos has affected multiple files. The full list and code can be seen in appendix C. A brief summarisation goes here together with the header for a new file main.c, syscall.c+h, io.c, lib.c+h, (new: pipe.c+h)

Listing 1: pipe.h

```
#ifndef BUENOS_PIPE_H
#define BUENOS_PIPE_H

#include "lib/types.h"

#define PIPE_BUFFER_LENGTH 256
#define MAX_PIPES 20

typedef uint32_t pipe_id_t;

typedef struct {
  char buffer[PIPE_BUFFER_LENGTH];
  uint32_t length;
  int read_end;
```

```
  int write_end;
  int write_id;
  int read_id;
  int free;
  pipe_id_t id;
} pipe_t;

void pipe_init();
pipe_t* pipe_get_pipe(int fd);
void pipe_write(char* buffer, int length, pipe_t* pipe);
void pipe_read(void* buffer, int length, pipe_t* pipe);
int pipe_pipe(int fds[2]);
int pipe_dup(int oldfd, int newfd);

#endif
```

As you can see most of the piping is done in the pipe module. This design choice is based on keeping the buenos source as clean as possible and to avoid cluster code. The pipe module must be initialised before any piping works. This is done in the main.c of buenos

Listing 2: main.c

```
void init(void) {
  ...

  kwrite("Initializing pipe system\n");
  pipe_init();
  ..
}
```

pipe_init() sets up an internal table of all available pipes and assigns default write and read ends towards them aswell as read id's and write id's. The id's are used to query the pipes for finding the right one (for the file descriptors) and the ends are used as the final file descriptor endpoint, which can be changed by calling pipe_dup. A pipe has an empty buffer that a process can read from and write to. This is done by first calling syscall_pipe which will set up the pipe for the process, which it can write to by calling syscall_write and read from by calling syscalll_read with the respective pipe handle fds[0] or fds[1] for read and write respectfully.

The existing system calls, read and write is modified to support piping. This is not done directly in the syscalls but rather in io.c which you can see in the Appendix C listing 17

The io.c read/write code is changed such that it will check if a file is related to a pipe (only stdout, stdin, stderr are not piped). if it is then write or read to or from the pipe and continue writing or reading from or to a file.

The tests does not work, as there is a bug in the read/write function in pipe.c. It seems that the logic needs more attention, however the general idea is there.

1.4 P4

For forcefully and gracefully terminating a process the kernel is extended with a syscall.

Listing 3: syscall.c

```
int syscall_kill(int32_t pid, int retval) {
  return process_kill(pid, retval);
}
```

The process of killing a process is delegated to the process module since it makes the most sense that it should be handled there.

Listing 4: process.c

```
/**
 * Forcefully kills a process
 * @param  pid     id of process
 * @param  retval killing process's return value
 * @return         0 on success -1 on error
 */
int process_kill(process_id_t pid, int retval) {

  // Only kill valid processes
  if (pid < 0 || pid >= PROCESS_MAX_PROCESSES) { return -1; }

  spinlock_acquire(&process_table_slock);
  // Finish the thread
  thread_finish_pid(pid, retval);
  spinlock_release(&process_table_slock);
  return 0;
}
```

First of all we check if the process id is valid. If not we return -1. Then we lock the process_table and call thread_finish_pid which then handles the rest. Ie making sure the process terminates. The reason for this is because it makes the most sense to make the thread subsystem handle thread termination and we need to make sure that the thread_table is locked before making changes to it.

Listing 5: thread.c

```
void thread_finish_pid(int pid, int retval) {
  ...
  t->user_context->cpu_regs[3] = SYSCALL_EXIT;
  t->user_context->cpu_regs[4] = retval;
  t->user_context->pc = 0x00001014;
  ...
}
```

The most important lines here are the once affecting the user_context in the thread. Here we force the thread to make a syscall to SYSCALL_EXIT, set the return value and lastly the program counter is moved towards the fixed syscall address as hinted in the handout.

It was tempting at first to just copy the contents of process_finish() however that would be doing the same thing twice since the last line calls thread_finish() (which by the way only would make the calling thread finish) and actually would do the same thing. It would probably also make the system crash since multiple contexts would be freed twice. Instead we force the the thread to exit it self by a pid as described before.

Testing the solution is done by creating a test program kill.c and two other programs that kill.c will call and kill before they finish them selves. The first test program (exec_child1.c) runs infinitely and doesn't do anything. The second test program (exec_child2.c) runs infinitely and prints out a message.

exec_child1.c and exec_child2.c is completely modified from the buenos handout. Don't rely on them working for other tests that might use them.

"kill.c" executes exec_child1.c first and counts down from 10 to 0 and then kills it. Next exec_child2.c is executed and the counter is set to 5 "time" before it is killed. To avoid the screen from being filled with prints and to be able to count down "slowly" I've implemented a pseudo timer. This timer just does a lot of empty executions before returning. When you run the test you can change the wait_clock in exec_child2.c and kill.c to suit your computers speed. On i5 dual core I've set it to 1000 which is almost half a second. Other than that make sure that kill, exec_child1 and exec_child2 is on disk and their names match exactly if you intend to test the program.

To be able to make the syscalls tests/lib.c got a wrapper function that calls SYSCALL_KILL in the kernel.

See appendix A for test source, Appendix B, for output and Appendix C for source code

2 THEORETICAL

2.1 T1

Listing 6: Bicycle Factory

```
sem wheel_sem = sem_create(1);
sem frame_sem = sem_create(1);
mutex bicycles_mutex = mutex_create();

void frame() {
  while (true) {
    wait(frame_sem);
    acquire(bicycles_mutex);

    if (bicycles_left <= 0) {
      release(bicycles_mutex);
      signal(frame_sem);
      exit_thread(NULL);
    }

    b = bicycles[BICYCLES_TO_MAKE - bicycles_left];
    b.frame = 1;

    if (b.frame == 1 && b.wheels == 2) {
      bicycles_left--;
    }

    signal(frame_sem);
    release(bicycles_mutex);
  }

}

void wheel() {
  while (true) {

    wait(wheel_sem);
    acquire(bicycles_mutex);

    if (bicycles_left <= 0) {
      release(bicycles_mutex);
      signal(wheel_sem);
      exit_thread(NULL);
    }
```

```
      b = bicycles[BICYCLES_TO_MAKE - bicycles_left];
      b.wheels = 2;

      if (b.frame == 1 && b.wheels == 2) {
        bicycles_left--;
      }

      signal(wheel_sem);
      release(bicycles_mutex);

    }
}
```

2.1.1 Correctness

The pseudo code uses mutex and semaphores to synchronize and protect the bicycles. The semaphores is used to make sure that the bicycle only has 2 wheels and one frame and the mutex lock is used to make sure that two threads won't interfere with each other.

2.2 T3

In this theoretical task the focus is going to be printing files and users.

I have chosen printing since it's a very common task and personally I think it's necessary of any OS to support.

Users was chosen since I think it's fun what kinds of solutions you can come up with regarding their responsibility and the protection methods. I will later cover just one of the possible ways to structure users.

2.2.1 Printing

This protocol has the responsibility for printing files. It works closely with the networking protocols and the file handle protocols. In fact it would (almost) rely on those protocols since without files there is nothing to print and without networking it would be impossible to send messages to the printer. The latter is a subject to discuss. It would be possible if it's wired, but a sub network is still needed.

One other thing that is needed too is a driver for the printer, to abstract the signal processing since a printer might not have the same architecture as the OS. If we assume a generic printer driver is provided and networking is working it would be trivial to support file printing.

First of all the kernel must be extended for printing file descriptors. This would most likely be done with system calls like syscall_print(file f) where file is the filehandle (the file id). This system call would then call print_process_print(file f) which would add the print job to

it's internal print job queue. That is, each print_process must maintain all it's own print jobs such that they upon termination can clean up after themselves. This list is only for internal housekeeping, such that terminating a print_process will reveal in the recurring jobs to be cancelled.

2.2.2 Users

Having different users is almost a prerequisite for any secure OS. A user can either be administrative or not. An administrator has usually access to change protected files and sectors (either user protected or system protected) of the OS where as the non administrator (user) only has user privileges, such that they can only change what they own.

Authenticating is done through passwords upon system startup and depending on who's logged in different access is granted.

An Access Control List (ACL) must be implemented to tell what different kind of users can do. This list can for instance provide a score to each kind. Let's assume we have three kinds of ACL levels. root(acl score 3): access to everything. admin(acl score 2): access to everything except for critical sections such as the kernel. user(acl score 1): access to user created content.

We would need a new subsystem called user. The user has a table of every user. the user is defined in a struct user_t which holds information about the user such as the user id, acl level and such. The user subsystem would also have at least a method to get the current user role. A user is able to act as another type if granted. This would be granted through a system call such as syscall_role(acl_t new_role, const char* password)

Now every syscall would need to be redefined to check for the current user's role to see if the ACL level is high enough to grant access.

Filehandle structures would also have a user id as there respective owner.

A Test Programs

Listing 7: queue_test.c

```
#include "fg_queue.h"
#include "cg_queue.h"
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <pthread.h>
#include <assert.h>

// The number of items to producer/consume
#define MAX 5000000

/**
 * producer function for the fine grained queue
 * @param  queue the fg queue
 * @return       null
 */
void *fg_producer(void* queue) {

  // Insert MAX items into queue
  for (long i = 0; i < MAX; ++i)
    fg_queue_put(queue, (void*)1);

  pthread_exit(NULL);
}

/**
 * consumer function for the fine grained queue
 * @param  queue the queue
 * @return       null
 */
void *fg_consumer(void* queue) {

  // Pop all items from the queue
  for(int i = 1; i < MAX;)
    if ((fg_queue_get(queue) != NULL)) ++i;

  pthread_exit(NULL);
}

/**
 * producer function for the coarse grained queue
 * @param  queue the cg queue
 * @return       null
```

```
 */
void *cg_producer(void* queue) {

  // Insert MAX items into queue
  for (long i = 0; i < MAX; ++i)
    cg_queue_put(queue, (void*)1);

  pthread_exit(NULL);
}

/**
 * consumer function for the coarse grained queue
 * @param  queue the queue
 * @return       null
 */
void *cg_consumer(void* queue) {

  // Pop all items from the queue
  for(int i = 1; i < MAX;)
    if ((cg_queue_get(queue) != NULL)) ++i;

  pthread_exit(NULL);
}

/**
 * Helper function for benchmarking. Creates threads and measures
    performance
 * @param t number of threads
 * @param producer the producer function pointer
 * @param consumer the consumer function pointer
 * @param arg the function argument
 * returns the benchmarking duration in seconds
 */
float benchmark_start(int t, void *(*producer) (void *), void *(*
   consumer) (void *), void* arg) {

  // Setup timers
  clock_t start_t, end_t;
  float total;

  // Setup array of threads
  pthread_t producer_threads[t];
  pthread_t consumer_threads[t];
  int s, s0;

  // Start timer
  start_t = clock();
```

```
  // Create the threads and start benchmarking
  for (int i = 0; i < t; ++i) {
    s = pthread_create(&producer_threads[i], NULL, producer, arg);
    s0 = pthread_create(&consumer_threads[i], NULL, consumer, arg)
       ;
    if (s != 0 || s0 != 0) {
      printf("Error creating thread.");
      exit(-1);
    }
  }

  // Synchronize threads
  for (int i = 0; i < t; i++) {
    pthread_join(producer_threads[i], 0);
    pthread_join(consumer_threads[i], 0);
  }

  // Calculate the benchmark duration in seconds
  end_t = clock();
  total = (double)(end_t - start_t) / CLOCKS_PER_SEC;

  return total;
}

/**
 * Performs a benchmark on fine grained queue
 * @param t number of threads
 */
void fg_benchmark(int t) {
  fg_queue_t queue;
  fg_queue_init(&queue);
  float dur = benchmark_start(t, &fg_producer, &fg_consumer, &
     queue);
  fg_queue_destroy(&queue);
  printf("%-10s%-10d%-10d%-10f\n", "FG", t, MAX, dur);
}

/**
 * Performs a benchmark on coarse grained queue
 * @param t number of threads
 */
void cg_benchmark(int t) {
  cg_queue_t queue;
  cg_queue_init(&queue);
  float dur = benchmark_start(t, &cg_producer, &cg_consumer, &
     queue);
```

```
  cg_queue_destroy(&queue);
  printf("%-10s%-10d%-10d%-10f\n", "CG", t, MAX, dur);
}

int main(int argc, char const *argv[]) {

  // Pretty print for results
  printf("%-10s%-10s%-10s%-10s\n", "Queue", "Threads", "Items", "
     Time (seconds)");

  int q = 2; // The number of threads

  // Start benchmarking fine grained
  for (int i = 1; i <= q; i *= 2) {
    fg_benchmark(i);
  }

  printf("\n\n
     ----------------------------------------------------------------------\
     n\n");

  // Benchmark coarse grained
  for (int i = 1; i <= q; i *= 2) {
    cg_benchmark(i);
  }
  return 0;
}
```

Listing 8: tests/rand.c

```
/**
 * This program tests the functionallity of the pseudo random
 * syscall implementation.
 */

#include "tests/lib.h"

int main() {

  // Gets random numbers with range from 0 to 40
  int i, r = 0;
  for (; i < 40; ++i) {
    r = syscall_rand(i);
    printf("syscall_rand(%d): %d\n", i, r);
    if (r >= i) {
      printf("ERROR: random is over range.");
    }
```

```
  }

  return 0;
}
```

Listing 9: exec_child1.c

```
/*
 * Userland exec test; child 1.
 */

#include "tests/lib.h"

int main() {
  printf("Running executable 1...\n\n");
  while(1);
  return 0;
}
```

Listing 10: exec_child1.c

```
/*
 * Userland exec test; child 1.
 */

#include "tests/lib.h"

int main() {
  printf("Running executable 1...\n\n");
  while(1);
  return 0;
}
```

Listing 11: exec_child2.c

```
#include "tests/lib.h"

#define wait_clock 1000

void wait(int time) {

  int ct = 0; int i = 0;
  while(ct < time) {
    i = 0;
    while(i++ < wait_clock); {
```

```
      ct++;
    }
  }
}

int main() {
  syscall_write(1, "Running executable 2\n\n", 36);
  while(1) {
    wait(100);
    syscall_write(1, "Myaaaaaahhhhhh\n\n", 36);
  };
  return 0;
}
```

Listing 12: kill.c

```
/*
 * Userland exec test.
 */

#include "tests/lib.h"

// No timer so just do something wait
#define wait_clock 100000

void read() {
  char c;
  syscall_read(stdin, &c, 1);
}

void wait(int time) {

  int ct = 0; int i = 0;
  while(ct < time) {
    i = 0;
    while(i++ < wait_clock); {
      ct++;
      printf("killing process in %d time...\n", time - ct);
    }
  }
}

void run_program(char *program, int time) {
  uint32_t pid = syscall_exec(program);
  printf("\nkilling process in %d time...\n\n", time);
  wait(time);
```

```
  syscall_kill(pid, 0);
  printf("Press any key to continue...\n\n");
  read();
}

int main() {
  run_program("[disk]exec_child1", 10);
  run_program("[disk]exec_child2", 5);
  return 0;
}
```

B Test Output

Listing 13: rand.c-output

```
Starting initial program '[disk]rand'
syscall_rand(0): 0
ERROR: random is over range.syscall_rand(1): 0
syscall_rand(2): 1
syscall_rand(3): 0
syscall_rand(4): 2
syscall_rand(5): 4
syscall_rand(6): 2
syscall_rand(7): 3
syscall_rand(8): 1
syscall_rand(9): 8
syscall_rand(10): 0
syscall_rand(11): 3
syscall_rand(12): 5
syscall_rand(13): 11
syscall_rand(14): 2
syscall_rand(15): 2
syscall_rand(16): 6
syscall_rand(17): 7
syscall_rand(18): 10
syscall_rand(19): 3
syscall_rand(20): 5
syscall_rand(21): 11
syscall_rand(22): 19
syscall_rand(23): 3
syscall_rand(24): 8
syscall_rand(25): 11
syscall_rand(26): 15
syscall_rand(27): 7
syscall_rand(28): 3
syscall_rand(29): 4
syscall_rand(30): 1
syscall_rand(31): 21
syscall_rand(32): 19
syscall_rand(33): 7
syscall_rand(34): 18
syscall_rand(35): 24
syscall_rand(36): 28
syscall_rand(37): 5
syscall_rand(38): 17
syscall_rand(39): 5
```

Listing 14: tests/kill.c

```
Starting initial program '[disk]kill'

killing process in 10 time...

Running executable 1...
killing process in 9 time...
killing process in 8 time...
killing process in 7 time...
killing process in 6 time...
killing process in 5 time...
killing process in 4 time...
killing process in 3 time...
killing process in 2 time...
killing process in 1 time...
killing process in 0 time...
killed process: 1

Press any key to continue...

killing process in 5 time...

Running executable 2

MyaaaaaahhhhkilMyaaaaaahhhhhhling processMyaaaaaahhhhhh

 in 4 time...
Myaaaaaahhhhhh

killing process in 3 time...
Myaaaaaahhhhhh

killing process in 2 time...
Myaaaaaahhhhhh

killing process in 1 time...
Myaaaaaahhhhhh

Myaaaaaahhhhhh

killing process in 0 time...
killed process: 2

Press any key to continue...
killed process: 0

Run out of initprog.  Return value: 0
Kernel: System shutdown started...
```

```
VFS: Entering forceful unmount of all filesystems.
VFS: Forcefully unmounting volume [disk]
Kernel: System shutdown complete, powering off

>>> Remote disconnect
>>> Waiting for incoming connection on socket '/home/arkimedes/.
    osm/fyams.socket' ...
```

C Added And Modified Source to Buenos

This appendix will contain all the source that is either modified or added to buenos

Listing 15: main.c

```
void init(void) {
  ...

  kwrite("Initializing pipe system\n");
  pipe_init();
  ..
}
```

Listing 16: libc.c

```
/**
 * Appends a string to another string
 * @param  dest the string that will get appended
 * @param  src  the string to append
 */
void str_cat(char* dest, const char* src) {

  // Find the end of the dest string
  if(*dest != '\0')
    while(*++dest != '\0');

  // Copy every character from the src
  while(*src != '\0') {
    *dest++ = *src++;
  }

  // Null terminate the string
  *dest = '\0';
}

/**
 * pops the first n characters from a string and puts it
 * into a buffer
 * @param src    the string to pop from
 * @param buffer the destination for the popped characters
 * @param length the amount of characters to pop
 */
void str_read(char* src, char* buffer, int length) {
  int i = 0;
  for (; i < length; ++i) {

    // append the first character from the src to the buffer
```

```
    buffer[i] = src[0];

    // move every character one step back (the popping)
    for (char *ps = src; *ps != '\0'; ps++)
      *ps = *(ps+1);
  }

  // null terminate
  buffer[i] = '\0';
}
```

Listing 17: io.c

```
int io_read(openfile_t file, void* buffer, int length)
{
  ...

  // Don't pipe if reading from std
  if (file > 2) {

    // Get the pipe for the file
    pipe_t *pipe = pipe_get_pipe(file);

    // Don't read to empty pipes
    if (pipe == NULL) {
      KERNEL_PANIC("Cannot read from empty pipe\n");
    }

    // Get the read end and update the file
    file = pipe->read_end;

    // Pipe read to the buffer
    pipe_read(buffer, length, pipe);
  }
  ...
}

int io_write(int file, void* buffer, int length)
{
  ...

  // Don't pipe if writing to std
  if (file > 2) {

    // Get the pipe for the file
    pipe_t *pipe = pipe_get_pipe(file);
```

```
    // Don't read to empty pipes
    if (pipe == NULL) {
      KERNEL_PANIC("Cannot write from empty pipe\n");
    }

    // Get the read end and update the file
    file = pipe->write_end;

    // Write to the pipe buffer
    pipe_write(buffer, length, pipe);
  }
  ...
}
```

Listing 18: pipe.c

```
#include "drivers/gcd.h"
#include "fs/vfs.h"
#include "kernel/assert.h"
#include "proc/syscall.h"
#include "proc/pipe.h"
#include "proc/io.h"
#include "kernel/interrupt.h"

// A table of pipes for reference
pipe_t pipe_table[MAX_PIPES];

// Spinlock to protect the pipe table
spinlock_t pipe_table_slock;

/**
 * Resets a pipe
 * @param id the id for the pipe to reset
 */
void pipe_reset(pipe_id_t id) {

  pipe_table[id].free = 1;
  pipe_table[id].id = id;
  pipe_table[id].length = 0;

  // Set the read, write-id and end to id with an offset of 3
  pipe_table[id].read_id = 3 + (id * 2);
  pipe_table[id].write_id = 4 + (id * 2);
  pipe_table[id].read_end = pipe_table[id].read_id;
  pipe_table[id].write_end = pipe_table[id].write_id;
```

```
  // initialize the buffer
  pipe_table[id].buffer[0] = '\0';
}

/**
 * initializes the pipe subsystem
 */
void pipe_init() {
  spinlock_release(&pipe_table_slock);
  int i;
  for (i = 0; i < MAX_PIPES; i++) {
    pipe_reset(i);
  }
}

/**
 * Sets up a pipe with a filehandle for read end and write end
 * @param fds file descriptors for read end and write end. 0 1
    respectively
 */
int pipe_pipe(int fds[2]) {
  // Find a free pipe
  int i;
  for(i = 0; i < MAX_PIPES; i++) {
    if(pipe_table[i].free) {
      pipe_table[i].free = 0;

      // Set the file descriptors
      fds[0] = pipe_table[i].read_id;
      fds[1] = pipe_table[i].write_id;
      return 0;
    }
  }
  kwrite("No available PIPES -1\n");
  return -1;
}

/**
 * Writes to a pipe
 * @param buffer the content to write
 * @param length the length of the buffer
 * @param pipe   the pipe to write to
 */
void pipe_write(char* buffer, int length, pipe_t* pipe) {

  // Protection
  interrupt_status_t intr_status = _interrupt_disable();
```

```
  spinlock_acquire(&pipe_table_slock);

  // Block while the buffer is full or are about to get full
  int data_to_write = (int)pipe->length + length;
  while(data_to_write >= PIPE_BUFFER_LENGTH) {
    spinlock_release(&pipe_table_slock);
    thread_switch();
    spinlock_acquire(&pipe_table_slock);
    data_to_write = (int)pipe->length + length;
    _interrupt_set_state(intr_status);
  }

  // update the pipe and write to it
  pipe->length += length;
  str_cat(pipe->buffer, buffer);

  spinlock_release(&pipe_table_slock);
  _interrupt_set_state(intr_status);
}

/**
 * Reads from the pipe
 * @param buffer the buffer to read into
 * @param length the amount to read
 * @param pipe   the pipe
 */
void pipe_read(void* buffer, int length, pipe_t* pipe) {
  // Protection
  interrupt_status_t intr_status = _interrupt_disable();
  spinlock_acquire(&pipe_table_slock);

  // Make sure we are not reading an empty pipe, and block if so
  int data_to_read = (int)pipe->length - length;
  while(data_to_read <= 0) {
    spinlock_release(&pipe_table_slock);
    thread_switch();
    spinlock_acquire(&pipe_table_slock);
    data_to_read = (int)pipe->length - length;
    _interrupt_set_state(intr_status);
  }

  // update the pipe and read from it
  pipe->length -= length;
  str_read(pipe->buffer, buffer, length);

  spinlock_release(&pipe_table_slock);
  _interrupt_set_state(intr_status);
```

```
}

/**
 * Gets a pipe by searching for a match on read_id or write_id
 * @param  fd the read or write id
 * @return    pipe
 */
pipe_t* pipe_get_pipe(int fd) {

  pipe_id_t i;
  for(i = 0; i < MAX_PIPES; i++) {
    pipe_t* pipe = &pipe_table[i];

    // If the pipe's write or read id matches the file descriptor
    // we have a match.
    if(pipe->write_id == fd || pipe->read_id == fd ) {
      return pipe;
    }
  }

  // No pipe found. Return null
  return NULL;
}

/**
 * Copies a pipe's file descriptor
 * @param  oldfd the old fd
 * @param  newfd the new fd
 * @return       0 on success, -1 on error
 */
int pipe_dup(int oldfd, int newfd) {

  pipe_t* pipe = pipe_get_pipe(oldfd);
  if (pipe) {
    if (pipe->write_id == oldfd) {
      pipe->write_end = newfd;
    } else if (pipe->read_id == oldfd) {
      pipe->read_end = newfd;
    }
    return 0;
  }
  return -1;
}
```

Listing 19: process.c

```
/**
```

```
 * Forcefully kills a process
 * @param  pid    id of process
 * @param  retval killing process's return value
 * @return        0 on success -1 on error
 */
int process_kill(process_id_t pid, int retval) {

  // Only kill valid processes
  if (pid < 0 || pid >= PROCESS_MAX_PROCESSES) { return -1; }

  spinlock_acquire(&process_table_slock);
  // Finish the thread
  thread_finish_pid(pid, retval);
  spinlock_release(&process_table_slock);
  return 0;
}
```

Listing 20: syscall.c

```
int syscall_kill(int32_t pid, int retval) {
  return process_kill(pid, retval);
}

/**
 * Generates a pseudo random number between 0 and range - 1
 * @param  range the maximum output + 1
 * @return a pseudo generated random number between 0 and range -
    1
 */
uint32_t syscall_rand(uint32_t range) {
  return _get_rand(range);
}

...

case SYSCALL_PIPE:
  V0 = pipe_pipe((int *)A1);
break;

case SYSCALL_DUP:
  V0 = pipe_dup((int)A1, (int)A2);
break;

...

case SYSCALL_RAND:
```

```
    V0 = syscall_rand((uint32_t)A1);
break;

...
```

C.1 Changes To Header Files

Listing 21: pipe.h

```
#ifndef BUENOS_PIPE_H
#define BUENOS_PIPE_H

#include "lib/types.h"

#define PIPE_BUFFER_LENGTH 256
#define MAX_PIPES 20

typedef uint32_t pipe_id_t;

typedef struct {
  char buffer[PIPE_BUFFER_LENGTH];
  uint32_t length;
  int read_end;
  int write_end;
  int write_id;
  int read_id;
  int free;
  pipe_id_t id;
} pipe_t;

void pipe_init();
pipe_t* pipe_get_pipe(int fd);
void pipe_write(char* buffer, int length, pipe_t* pipe);
void pipe_read(void* buffer, int length, pipe_t* pipe);
int pipe_pipe(int fds[2]);
int pipe_dup(int oldfd, int newfd);

#endif
```

Listing 22: libc.h

```
void str_cat(char* dest, const char* src);
void str_read(char* src, char* buffer, int length);
```

Listing 23: process.h

```
...

/**
 * Kills the process with the given process id
 * @param  pid    the process id
```

```
 * @param  retval the return value of the process
 * @param kill_child_processes  if true, process_kill is called
    recursively on child processes
 * @return        0 upon success, a negative value on error.
 */
int process_kill(process_id_t pid, int retval);

...
```

Listing 24: syscall.h

```
...
#define SYSCALL_RAND       0x107
#define SYSCALL_KILL       0x108
...
#define SYSCALL_PIPE     0x209
#define SYSCALL_DUP      0x210
...
```

Listing 25: thread.h

```
...

void thread_finish_pid(int pid, int retval);

...
```

Listing 26: lib.h

```
...

uint32_t syscall_rand(uint32_t);
int syscall_pipe(int fds[2]);
int syscall_dup(int oldfd, int newfd);
int syscall_kill(pid_t pid, int retval);

...
```