

SOPE - solved exams

Diogo Miguel Ferreira Rodrigues
dmfrodrigues2000@gmail.com

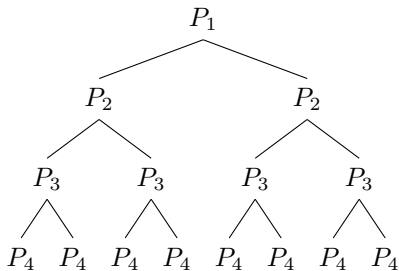
2019/2020, 2nd semester

Table of contents

19 Exam 2018/19	2
19.1 Exercise 1	2
19.2 Exercise 2	3
19.3 Exercise 3	4
19.4 Exercise 4	5
19.5 Exercise 5	5
19.6 Exercise 6	6
19.7 Exercise 7	7

19 Exam 2018/19

19.1 Exercise 1

- (a) True **Circular wait** is one of the four conditions necessary for a deadlock, so removing circular waits is usually the best way to solve the deadlock, assuming all the other three conditions (mutual exclusion, hold and wait and no preemption) are unavoidable, which usually are in a scenario where sharing resources is absolutely necessary.
- (b) True Let P_n be a process, where n is the value of i after incrementing and before evaluating the **for-loop** conditional. At each **fork**, a new process is split into a parent and a child. P_4 are the processes that will print **Hello**, since those will break the loop. Since there are eight P_4 processes, **Hello** will be printed eight times.
- 
- ```

graph TD
 P1 --> P2L[P2]
 P1 --> P2R[P2]
 P2L --> P3L1[P3]
 P2L --> P3L2[P3]
 P2R --> P3R1[P3]
 P2R --> P3R2[P3]
 P3L1 --> P4L11[P4]
 P3L1 --> P4L12[P4]
 P3L2 --> P4L21[P4]
 P3L2 --> P4L22[P4]
 P3R1 --> P4R11[P4]
 P3R1 --> P4R12[P4]
 P3R2 --> P4R21[P4]
 P3R2 --> P4R22[P4]

```
- (c) False **exec** calls do not create new processes, they load another program's code into the current process and execute it, so the process code is only replaced. The calls corresponding to the problem statement are **popen** and **system**.
- (d) True It only makes sense to talk about *quantum* when we are using round-robin (RR) scheduling, so we assume there are at least two **Multilevel Feedback Queue (MLQ)** queues which internally implement RR. Processes with higher priority are expected to finish execution faster, so they are assigned a smaller *quantum*; if the assumption is not true then they have the wrong priority, and are moved to less-priority queues.
- (e) True **Preemption** usually uses a timer to keep track of time since it was last dispatched. Preemption usually also includes a system to interrupt a lower-priority process if a new process with higher priority arrives.
- (f) False A **race condition** is a situation where two processes use the same data almost simultaneously, meaning the outcome may be highly unpredictable since it depends on the order of execution of each processes' instructions. It might lead to corrupted data if data is being simultaneously edited and read, or read by both processes. A **critical section** is a section of code that must be executed exclusively by one process at a time, so the fact two processes can't execute a critical section simultaneously is good, and usually achieved with a mutex.
- (g) True Two different processes can have the same **logical address** mapped to different **physical addresses**. Actually, it is also very likely that the virtual addresses of two processes sharing memory are different although they map to the same physical address.
- (h) False The **principle of locality of reference** states that there is a tendency that a program's instructions and data are repetitively accessed over a short period of time. This principle is not a base for virtual memory management, since its fundamental application would mean a process's code and data would be completely loaded into physical principal memory, and virtual memory is a way of seeming to have more virtual memory than physical memory, meaning instructions and data must be constantly swapped in and out of physical principal memory.
- (i) False **Multiprogramming** means there can be several processes loaded in principal memory simultaneously, while **multiprocessing** means several processes can run simultaneously. There can be multiprogramming without multiprocessing if it is possible to have multiple processes loaded in principal memory but only one process can be run at the same time (e.g., a virtual machine assigned with one running thread and 4GB of physical principal memory can have several loaded processes but only one can run at a time).

- |     |       |                                                                                                                                                                                                                                                                                                                                                    |
|-----|-------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| (j) | False | After reading the last <b>directory entry</b> , <code>readdir</code> signals there are no more entries by returning <code>NULL</code> .                                                                                                                                                                                                            |
| (k) | False | <code>exec</code> calls do not create processes, they replace the current process running code.                                                                                                                                                                                                                                                    |
| (l) | False | <code>ps</code> reports <b>current processes</b> , so it has nothing to do with signals.                                                                                                                                                                                                                                                           |
| (m) | False | <b>i-nodes</b> do not contain file names, only directories contain (filename, inode) pairs. That is why several hard links to the same file can have different names.                                                                                                                                                                              |
| (n) | True  | When a process is forked, all open file entries are duplicated automatically so the child process can use the same file without having to open it again (file pointers are also duplicated as a consequence, so file reads will not be synchronized between the parent and child processes).                                                       |
| (o) | True  | The <b>Process Control Block</b> holds essential information that must be available for retrieval by its parent, so the PCB may only be destroyed once that information has been retrieved by the parent.                                                                                                                                          |
| (p) | True  | A system with 32-bit logical addresses can represent $2^{32}$ addresses; since each address maps to a byte, the total memory is $2^{32}\text{B} = 4\text{GB}$ . Pages have size $16\text{kB} = 2^{14}\text{B}$ . This means <b>the page table has <math>2^{32}/2^{14} = 2^{18}</math> elements</b> .                                               |
| (q) | True  | There are little guarantees to several similar signals being all received. In practice, we determined only two consecutive similar signals can be received, so the third signal sent while the first signal is being processed does not even arrive.                                                                                               |
| (r) | True  | In Linux, a thread function must have a prototype <code>void* f(void*)</code> .                                                                                                                                                                                                                                                                    |
| (s) | False | Using <code>sem_open</code> , <code>sem_close</code> and <code>sem_unlink</code> , processes can create <b>named mutexes</b> , allowing two different processes to use the same, shared mutex. Besides, if an unnamed mutex is allocated in a memory region shared among several processes it might also be accessible.                            |
| (t) | False | A piece of code that must be executed rapidly and without interruptions is an <i>atomic</i> section. A critical section is one that must only be executed by one process/thread at a time, but there is no limitation as for interruptions, since the CPU may interrupt a process inside a critical section to run a completely unrelated process. |

## 19.2 Exercise 2

### 19.2a Item a

It makes no sense to initialize a semaphore with a negative value, since a semaphore aims at describing an amount of available resources that can be simultaneously consumed. If the initial semaphore value is negative, not only it breaks the logic by stating that initially there are already more resources being used than they should, but also by breaking the runtime logic that threads must `sem_wait` to use a resource and `sem_post` to unlock the resource; any thread aiming at using one of the resources controlled by that semaphore will be stuck in an eternal `sem_wait` call, since no thread should call `sem_post` to unlock a resource it does not own.

### 19.2b Item b

Semaphores `sem1` and `sem2` keep the number of times `doWork1` and `doWork2` may run. Only `doWork1` may run initially (`init(sem1, 1)`, `init(sem2, 0)`); each time a process is about to run its `doWork` function, it asks if it can still run by waiting for the respective semaphore; once it is done, it has exhausted its possibility to run but informs the other process that it has gained an opportunity to run (P1 signals `sem2` and vice-versa).

---

```
1 /// P0
2 init(sem1, 1);
3 init(sem2, 0);
```

---

|                                                                                                      |                                                                                                      |
|------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| <pre> 1  /// P1 2  while(goOn){ 3      wait(sem1); 4      doWork1(); 5      signal(sem2); 6  }</pre> | <pre> 1  /// P2 2  while(goOn){ 3      wait(sem2); 4      doWork2(); 5      signal(sem1); 6  }</pre> |
|------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|

### 19.3 Exercise 3

#### 19.3a Item a

For simplicity and without loss of generality, let time be measured in integers, starting in 0 and increasing by 1 for each page being referred. In the second column, the contents of the frames are represented, where the first number is the page, and the second number is the last time it was referenced (so 1(4) means page 1 is in memory and was last referenced in time  $t = 4$ ).

| Processed | In memory               | Comments                                          |
|-----------|-------------------------|---------------------------------------------------|
| 1 2 3 4   | 1(0) 2(1) 3(2) 4(3)     | No pages were loaded, pages 1, 2, 3, 4 swapped in |
| 1         | 1(4) 2(1) 3(2) 4(3)     | No page faults                                    |
| 5         | 1(4) 5(5) 3(2) 4(3)     | Page fault, page 2 swapped out, page 5 swapped in |
| 4 3       | 1(4) 5(5) 3(7) 4(6)     | No page faults                                    |
| 6         | 6(8) 5(5) 3(7) 4(6)     | Page fault, page 1 swapped out, page 6 swapped in |
| 5         | 6(8) 5(9) 3(7) 4(6)     | No page faults                                    |
| 1         | 6(8) 5(9) 3(7) 1(10)    | Page fault, page 4 swapped out, page 1 swapped in |
| 3         | 6(8) 5(9) 3(11) 1(10)   | No page faults                                    |
| 2         | 2(12) 5(9) 3(11) 1(10)  | Page fault, page 6 swapped out, page 2 swapped in |
| 4         | 2(12) 4(13) 3(11) 1(10) | Page fault, page 5 swapped out, page 4 swapped in |
| 6         | 2(12) 4(13) 3(11) 6(14) | Page fault, page 1 swapped out, page 6 swapped in |
| 1         | 2(12) 4(13) 1(15) 6(14) | Page fault, page 3 swapped out, page 1 swapped in |
| 5         | 5(16) 4(13) 1(15) 6(14) | Page fault, page 2 swapped out, page 5 swapped in |
| 2         | 5(16) 2(17) 1(15) 6(14) | Page fault, page 4 swapped out, page 2 swapped in |
| 4         | 5(16) 2(17) 1(15) 4(18) | Page fault, page 6 swapped out, page 4 swapped in |

There were 14 page faults.

#### 19.3b Item b

- (1) One can find a process started thrashing when it finds almost as many page faults as references.
- (2) Thrashing happened in the sequence 2 4 6 1 5 2 4.

#### 19.3c Item c

- (1) The process started thrashing because the number of actively used pages (5 pages: 1, 2, 4, 5, 6) is larger than the number of allocated frames (4), and because the LRU strategy is not optimal.
- (2) One could use the page fault rate strategy, which starts by making an assumption as to what is an acceptable range of page fault rates, and after monitoring the page fault rate of each process during a certain time interval it performs frame allocations/deallocations to processes:

- Allocates an additional frame if the process has more page faults than acceptable.
- Deallocates one of its frames if the process has less page faults than acceptable (these frames can then serve other processes).

## 19.4 Exercise 4

(1) Unix supports files of various sizes. This is achieved using indexed file allocation, which consists of keeping, for each file, a special block containing its metadata and a table of all blocks allocated to that file. This strategy has many advantages, namely allowing fast modification and reduced external fragmentation (compared to contiguous allocation), as well as relatively fast direct access and added reliability (compared to linked allocation). To allow fast file expansion, Linux file systems uses direct indexes combined with multilevel indexes, where smaller indexes point directly to file blocks, and higher indexes point to other index blocks of growing depth. This allows fast access to small files (since their blocks are directly indexed), while also allowing decent performance when using large files (since they are indexed in a sort of logarithmic way).

(2) Say an index can be interpreted as a reference. On accessing very large files, even on accessing small portions of the end of a large file, there is a need to dereference all the numerous indexes from index block to index block before finding the block we are looking for, since the lowest indexes require less dereferencing (memory blocks are more directly indexed) than higher indexes.

## 19.5 Exercise 5

---

```
1 #include <errno.h>
2 #include <fcntl.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <sys/stat.h>
6 #include <sys/types.h>
7 #include <unistd.h>
8 #include <string.h>
9
10 #define ERROR(s) { perror(s); return 1; }
11
12 #define LINE_SIZE 1024
13
14 int main(int argc, char *argv[]) {
15 if(argc != 2) {
16 fprintf(stderr, "Usage: ./2019N-05 FILE");
17 return 1;
18 }
19 char *output_filepath = argv[1];
20 int fd_out = open(output_filepath, O_CREAT|O_WRONLY|O_APPEND);
21 if(fd_out == -1) ERROR("open output_filepath");
22 if(dup2(fd_out, STDOUT_FILENO) == -1) ERROR("dup2 fd_out");
23
24 int to[2]; if(pipe(to)) ERROR("Could not create pipe to");
25 int fr[2]; if(pipe(fr)) ERROR("Could not create pipe fr");
26
27 pid_t pid = fork();
28 if(pid == 0) /** Child */ {
29 if(close(to[1])) ERROR("close to[1]"); // close write end
30 if(dup2(to[0], STDIN_FILENO) == -1) ERROR("dup2 to[0]"); // stdin point to to[0]
31 if(close(fr[0])) ERROR("close fr[0]"); // close read end
32 if(dup2(fr[1], STDOUT_FILENO) == -1) ERROR("dup2 fr[1]"); // stdout point to fr[1]
33 execlp("bc", "bc", "-qi", NULL);
34 perror("Call to exec failed");
35 return 1;
36 }
37
38 if(close(to[0])) ERROR("close to[0]");
39 FILE *fd_to_bc = fdopen(to[1], "w"); if(fd_to_bc == NULL) ERROR("fdopen to[1]");
40 if(close(fr[1])) ERROR("close fr[1]");
41 FILE *fd_fr_bc = fdopen(fr[0], "r"); if(fd_fr_bc == NULL) ERROR("fdopen fr[0]");
42
43 char *line = NULL;
44 size_t n = 0;
45 while(getline(&line, &n, stdin) != -1) {
```

```
46 fprintf(fd_to_bc, "%s", line);
47 fflush(fd_to_bc);
48
49 getline(&line, &n, fd_fr_bc);
50 line[strlen(line)-1] = '\0';
51 printf("%s = ", line);
52 getline(&line, &n, fd_fr_bc);
53 printf("%s", line);
54 }
55 free(line);
56
57 return 0;
58 }
```

---

## 19.6 Exercise 6

### 19.6a Item a

The FIFO must be created by the reader process, since it acts as the server and provides services to other processes that might know how to communicate with it. Since the FIFO serves the purpose of receiving messages for the reader to process, it only makes sense for the FIFO to exist as long as the reader is running.

### 19.6b Item b

---

```
1 mkfifo("/users/tmp/srvfifo", 0000);
```

---

### 19.6c Item c

Writes of up to PIPE\_BUF bytes are guaranteed to be atomic, so it is impossible for two messages to overlap on writing.

As for reads, there is no atomicity guarantee, so there is a chance reads are interleaved. However, common sense tells there is no chance to read the same message twice, since once a character is read once it is removed from the FIFO and cannot be read again by another process. So in the terms of the problem statement it is impossible to read the same message twice.

If interleaved reading is considered a problem, it can be solved using a named mutex to enforce mutually exclusive reads.

As an alternative, a message queue could be used.

### 19.6d Item d

(1)

---

```
1 /// Reader
2 int main(){
3 int mem = shm_open("/tmp/is_fifo_open", O_CREAT|ORDWR, 0600);
4 ftruncate(mem, sizeof(int));
5 int *is_fifo_open = mmap(NULL, sizeof(int), PROT_READ|PROT_WRITE,
6 MAP_SHARED, mem, 0);
7 *is_fifo_open = 1;
8 char message[8192];
9 while(readfifo_nonblock(message) == 0){
10 process(message);
11 if(should_close_fifo()){ *is_fifo_open = 0; break; }
12 }
13 while(readfifo_nonblock(message) == 0) process(message);
```

```
14 munmap(mem, sizeof(int)); shm_unlink("/tmp/is_fifo_open");
15 return 0;
16 }
```

---

```
1 /// Writer
2 int main(){
3 int mem = shm_open("/tmp/is_fifo_open", ORDONLY, 0600);
4 int *is_fifo_open = mmap(NULL, sizeof(int), PROT_READ,
5 MAP_SHARED, mem, 0);
6 char message[8192];
7 while(has_messages_to_write()){
8 create_message(message);
9 if(*is_fifo_open) writefifo(message);
10 else break;
11 }
12 munmap(mem, sizeof(int));
13 return 0;
14 }
```

---

(2)

---

```
1 /// Reader
2 int fifo_filedes;
3 int openfifo_nonblock(){
4 mkfifo("/tmp/fifo", 0600);
5 fifo_filedes = open("/tmp/fifo", ORDONLY|O_NONBLOCK, 0600);
6 }
7 int readfifo_nonblock(char *message){
8 int r = read(fifo_filedes, message, 8192);
9 if(r != 8192) return 1;
10 return 0;
11 }
```

---

(3) A writer could know it cannot write to the FIFO any longer by checking the value of the integer in shared memory `/tmp/is_fifo_open`.

## 19.7 Exercise 7

### 19.7a Item a

The usage of `arg[]` is justified because each thread must receive its own individual index, where each index is in a different memory address, to make it persistent and allow different indexes for different threads. Otherwise, if we used the cycle variable, not only would it sooner or latter become out-of-scope (after the cycle ends) and cause undefined behaviour in the threads, but even if it was defined in the heap we would be providing all threads the same pointer, meaning all threads would think they had the same `thrIndex`.

### 19.7b Item b

If `main()` ended with `exit(0)`, the whole process would be terminated, including all threads. With `pthread_exit(NULL)`, only the main thread is terminated, and the threads that were just launched continue running inside the original process.

**19.7c Item c**

It is made with blocking, since a thread waits by locking `mutex`, which is a blocking call. However, for each `numThreads` times a thread acquires the lock, it only performs useful work once. This can in some cases be considered busy-wait, particularly if the number of threads is large.

**19.7d Item d**

---

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4 #include <pthread.h>
5
6 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
7 int numThreads;
8 int turn = 0; // The first thread to run must have thrIndex=turn=0
9 pthread_cond_t *conds = NULL; // <=====
10
11 void * thr(void *arg){
12 int thrIndex = *(int*)arg; // The effective indexes are 0,1,2,...
13 while (1){
14 pthread_mutex_lock(&mutex);
15 while(thrIndex != turn) // <=====
16 pthread_cond_wait(&conds[thrIndex], &mutex); // <=====
17 printf("%d ", thrIndex + 1); // <=====
18 turn = (turn + 1) % numThreads; // <=====
19 pthread_cond_signal(&conds[turn]); // <=====
20 pthread_mutex_unlock(&mutex);
21 }
22 return NULL;
23 }
24
25 int main(){
26 printf("Number of threads ? "); scanf("%d", &numThreads);
27 int *arg = (int *) malloc(sizeof(int)*numThreads);
28 conds = calloc(numThreads, sizeof(pthread_cond_t)); // <=====
29 pthread_cond_signal(&conds[turn]); // <=====
30 pthread_t *tid = (pthread_t *) malloc(sizeof(pthread_t)*numThreads);
31 for (int i = 0; i < numThreads; i++){
32 arg[i] = i;
33 pthread_create(&tid[i], NULL, thr, (void*)&arg[i]);
34 }
35 pthread_exit(NULL);
36 }
```

---