UNIVERSITY OF COPENHAGEN Version 1.0, 22 June 2015
Department of Computer Science

Resit exam: "Operating Systems and Concurrent Programming", 2015

Exam period: Monday, 22 June 2015 at 9:00 – Friday, 26 June 2015 at 14:00

Handing in: via Absalon

This document lays down the rules and the questions for the take-home resit of the course "Operating Systems and Concurrent Programming". There are three theoretical exercises (T1, T2, and T3) and four practical tasks: one task in programming with Pthreads (P1) and three `BUENOS`-specific tasks (P2, P3, and P4).

Your solutions to the `BUENOS` tasks must be based upon the specific version of `BUENOS` published alongside this document. This version supports user processes, semaphores, TLB handling, and file I/O as described in [Roadmap to the `BUENOS` System], and as implemented in the G assignments. For the Pthreads task, your solution should be based upon the sequential implementation of "awesome heaps"(see T1 and P1), also published alongside this document.

Grading

The theoretical exercises are weighted in total 40%, and practical tasks in all 60% of your grade. Your answer will be assessed using the 7-step grading scale with an external examiner. The deadline for our grading is 17 July 2015 and the results will be announced soon after that. As to grading, any news will be announced on the course homepage.

Emergency line

If, for some reason or another, you have to or want to contact the course team—during the exam or after the exam, the course coordinator is your point of contact: Jyrki Katajainen; e-mail: `<jyrki@di.ku.dk>`; telephone: (+45) 35335680.

Rules

The exam is ***individual***.

The intention is, like in any other written exam, that you complete the answers *100% individually.* This is an open-book exam and you are welcome to make use of the course textbooks and other reading material from the course. Any other sources must be cited appropriately.

Any errors or ambiguities in the problem formulations, or the handed-out source code, are considered to be part of the exam. Just state clearly the assumptions and/or corrections you had to make in order to solve the issue(s).

To be more specific, you are not permitted

- to discuss or share any part of this exam, including but not limited to solutions, with any other student. Both helping and receiving help are expressly forbidden.
- to copy an answer from the Internet without giving the source.
- to post any questions or answers related to the exam on our Discussion Forum in Absalon, or any other fora, before the exam is over.

This examination format is based on trust. Do not disappoint us (and betray yourself)! *Breaches of the above-mentioned rules will be handled in accordance with our disciplinary procedures* and the exam result will be declared **`void`**.

Files to be handed in

Please submit your solution electronically via Absalon in two files:

1. a *report* (in `pdf` format) with your name and KU user ID on the front page;
2. an *archive* (in either `zip` or `tar.gz` format) that contains a directory `heap/`, containing your solution to P1, and a directory `buenos/`, containing a working `BUENOS` source tree, as your solution to tasks P2, P3, and P4.

Your report should consist of two parts: a theoretical part and a practical part, *separated by a page break.* The assessment will be based on your *report.* Your source code will only be used to *verify* what you state in your report. If you do any testing (as you should), explain how we can reproduce your results. NB You must include *all* modified and added code in the appendices of your report for documentation purposes. Remember to comment your programs so they are easy to understand. In particular, it is important to document the changes you made to the handed-out version of `BUENOS`.

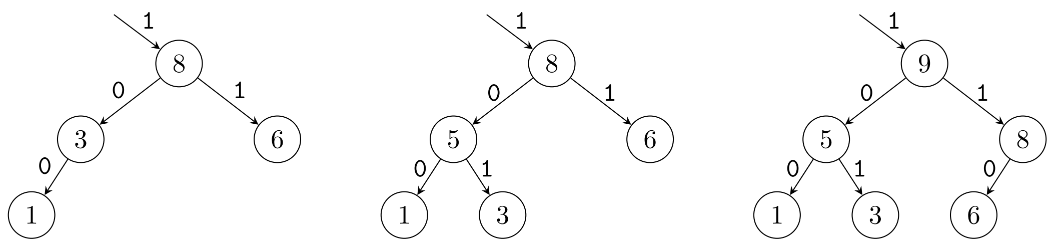
High-resolution scanned versions of handwritten solutions are acceptable for the theoretical exercises; please leave reasonable margins for printing and marking.

The practical part of your report should document which observations you made, and which assumptions your solution depends upon (either those which you chose yourself or those which were made by the designers of `BUENOS`). Document and justify the design decisions you have made, and reflect on your solution to each task.

Theoretical Part

T1 Concurrency: Awesome heaps (15%)

Let n denote the number of nodes in a nearly complete binary tree [Algorithms Book, Section B.5.3]. An awesome feature of such trees is the connection between the binary representation of n and the path from the root to the last node in the tree: The most significant bit denotes the root and each of the following bits leads to the left child if it is `0` and to the right child if it is `1`. Figures 1a, 1b, and 1c illustrate the cases where $n = 4 = \texttt{100}_2$, $n = 5 = \texttt{101}_2$, and $n = 6 = \texttt{110}_2$, respectively.



(a) The path to 1 is `100`. (b) The path to 3 is `101`. (c) The path to 6 is `110`.

Figure 1: Inserting 5 and 9 after having 3, 6, 8, and 1 inserted into an awesome heap.

Following this labelling scheme, we can insert a node into a nearly complete binary tree by following the path to the new last leaf using the binary representation of $n+1$, n being the number of nodes before the insertion. This ensures that the new node will be placed appropriately, guaranteeing that the height of the tree is logarithmic.

What remains, is to maintain the heap property. This can be done by following the chosen path downward until an adequate parent to the element being inserted is found, and "nudging" the rest of the path downwards. See Figure 1 for an example, and Figure 2 for the pseudo-code.

For lack of a better name, we call heaps that use this insertion procedure ***awesome heaps***. In this task you should study how well awesome heaps are suited for a concurrent implementation.

a) Critical sections (5%)

The first step from a sequential to a concurrent implementation is to identify the critical sections of the sequential implementation. Identify the critical sections in the pseudo-code in Figure 2. For each critical section, write down the line where it starts and stops, and argue why it forms a critical section.

b) Concurrency control (5%)

Solve the critical section problem for each of the critical sections identified above. Use notation from our textbook [Dinosaur Book]. For your convenience, we have published the LaTeX source code of the sequential pseudo-code in Figure 2 alongside this document.

```
INSERT($H$, $value$)
1  $N = $ NEW-NODE($value$, NIL, NIL) // no left or right child (yet)
2  $H.n = H.n + 1$
3  INSERT-ON-PATH($H, N, H.n$)

INSERT-ON-PATH($H, N, path$)
 1  $parent$ = NIL
 2  $current = H.root$
 3  while $current \neq$ NIL and $current.value \geq N.value$
 4      $path$ = ADVANCE-ONE-BIT($path$)
 5      $parent = current$
        // advance to left or right child, depending on the bit on the path
 6      $current$ = GET-NEXT-CHILD($current, path$)
    // $current$ points to first node on path such that $current.value < N.value$, if anything
 7  if $parent$ = NIL
 8      $H.root = N$
 9  else
10      SET-NEXT-CHILD($parent, path, N$)
    // subtree starting at $current$ is now outside $H$, so merge it back in!
11  MERGE-ON-PATH($current, N, path$)

MERGE-ON-PATH($out, in, path$)
 1  while $out \neq$ NIL
 2      $path$ = ADVANCE-ONE-BIT($path$)
 3      if NEXT-IS-LEFT($path$)
            // copy the right child, advance on left
 4          $in.right = out.right$
 5          $in.left = out$
 6          $in = out$
 7          $out = out.left$
 8      else
            // copy the left child, advance on right
 9          $in.left = out.left$
10          $in.right = out$
11          $in = out$
12          $out = out.right$
13  $in.left$ = NIL
14  $in.right$ = NIL
```

Figure 2: Pseudo-code for an insertion into an awesome heap.

c) Deleting elements (5%)

Heaps are not *that* awesome if we cannot extract the maximum element from the heap and maintain the heap property. Consider how you might do this for awesome heaps. Can deletion from an awesome heap happen concurrently with other deletion or insertion operations? Argue why, or why not.

Hint: You may want to sketch the deletion algorithm in pseudo-code. If in doubt, consult [Algorithms Book, Section 6.5].

T2 Online algorithms: Memory allocation and freeing (15%)

In this task we go deep in the implementation of a memory-management algorithm that can be used for servicing memory-allocation (`malloc`) and memory-freeing (`free`) requests. In the C standard library, these two functions are specified as follows:

void∗ `malloc(size_t n)`}

Effects: Allocate the amount of bytes specified by `n` from the heap. The allocated memory is uninitialized.

Returns: A pointer to the first byte of the allocated memory; or `NULL`, if not enough memory is available (i.e. `malloc` ***fails***).

void `free`(**void**∗ `ptr`)

Effects: Free the memory space pointed to by `ptr`, which must have been returned by a previous call to `malloc`. Otherwise, or if `free` has already been called with the same pointer value, undefined behavior occurs. If `ptr` is `NULL`, nothing is done.

Returns: Nothing.

Assume that we are given a contiguous memory segment, call it the ***arena***, for servicing memory-allocation and memory-freeing requests. The ***quick-fit scheme*** [Weinstock and Wulf 1988] divides the arena into two parts: (1) that which is currently allocated or previously allocated in the past, and (2) that which has never been allocated. The former part is called ***working storage*** and the latter ***pristine storage***. Initially, the pristine storage comprises the whole arena. Let the *minimum size* of an allocated block be $2^{\mathtt{k}}$ bytes and the *maximum size* $2^{\mathtt{m}}$ bytes. Each allocated block is of size 2^{ℓ}, for some integer $\mathtt{k} \le \ell \le \mathtt{m}$, and we say that the block is at ***level*** ℓ.

In the working storage, each block stores its level and a free block also stores a pointer to the next free block at the same level. That is, we maintain the list pointers in the free blocks themselves. In addition to these pointers, we need an array of pointers to the ***heads*** of the lists of free blocks, one for each level $\mathtt{k} \le \ell \le \mathtt{m}$. For the pristine storage, we maintain a pointer to its beginning.

Below is a partial implementation of the quick-fit scheme in C-like pseudo-code for your guidance:

```
unsigned char const k = ...;
unsigned char const m = ...;
unsigned long const  N = ...;

char arena[N];

struct free_block {
  unsigned char level;
  struct free_block* next;
};

struct free_block* heads[m + 1]; // heads[i] unused if i < k

char* pristine_storage;
```

The basic scheme for allocation is:

- Assume that the request under consideration is for a block of size `n`, and let δ be some implementation-dependent constant.
  - If $\mathtt{n} + \delta \leq 2^{\mathtt{k}}$, allocate a block of size $2^{\mathtt{k}}$.
  - If $\mathtt{n} + \delta > 2^{\mathtt{m}}$, `malloc` fails and returns `NULL`.
  - Otherwise, allocate a block of size $\max\{\mathtt{n} + \delta, 2^{\ell}\}$, where 2^{ℓ} is the smallest power of 2 larger than or equal to $\mathtt{n} + \delta$.
- If the free list of the required size is not empty, the request is serviced from that list. Otherwise, a block of the appropriate size is allocated from the pristine storage. If the pristine storage cannot service the request, `malloc` fails.

The scheme for freeing is also simple:

- If the block to be freed is at level ℓ, link it onto the free list for level ℓ.

a) Quick questions (5%)

Answer briefly to the following questions.

(i) What can you say about the parameter δ?

(ii) How to compute the smallest power of 2 larger than or equal to the given integer in constant worst-case time?

(iii) Do you have any proposals what to do with requests larger than $2^{\mathtt{m}} - \delta$ bytes?

(iv) How to know if an allocation from the pristine storage fails?

(v) Do you have any proposals what to do if an allocation from the pristine storage fails?

b) Implementation (5%)

Write the functions `malloc` and `free` in C-like pseudo-code; both functions should run in $O(1)$ worst-case time.

c) Fragmentation analysis (5%)

Assume that, at any given time, the total amount of memory allocated never exceeds $N \geq 2^{\mathtt{m}}$, and that all allocation requests are between 1 and $2^{\mathtt{m}} - \delta$ bytes. What is the smallest possible size of the arena so that the quick-fit scheme can work forever without `malloc` failing. This number is a function of $2^{\mathtt{k}}$, $2^{\mathtt{m}}$, and N, call it $\mathtt{A}(2^{\mathtt{k}}, 2^{\mathtt{m}}, N)$. Give as tight lower bound (`malloc` fails at some point for some request sequence) and upper bound (scheme works forever for any request sequence) for $\mathtt{A}(2^{\mathtt{k}}, 2^{\mathtt{m}}, N)$ as possible.

T3 Back-of-the-envelope calculations: Hard-disk drives (10%)

When doing the calculations, state clearly the specifications used and any additional assumptions you have made. Observe also that in your answer the calculations themselves are more important than the final results. But, if you can draw any interesting conclusions of the results, please, tell us.

a) Disk characteristics (4%)

For a disk, an important parameter is the ***data transfer rate***, also called ***throughput***, which is the amount of data transferred from (to) disk to (from) main memory in a particular period of time. The ***effective data transfer rate*** is the average of this, whereas the maximum data transfer rate is the maximum amount of data that can be transferred in a given time in ideal circumstances.

Consider a disk of type Seagate Barracuda with the following specifications:

Capacity: 3 TB
Cache size: 64 MB
Connects via: SATA 6 Gb/s
Spindle speed: 7 200 RPM
Maximum data transfer rate: 210 MB/s
Maximum seek time: 3 ms (read), 3.6 ms (write)
Sector size: 4 KB
Platters: 3
Heads: 6

Typically, this disk is used in two ways: (1) in ***sequential use***, chunks of size 2 MB are accessed sequentially and (2) in ***random use***, chunks of size 4 KB are accessed in different random locations.

On the basis of these data, calculate the following parameters for this type of disk:

(i) average seek time

(ii) average rotational latency

(iii) effective data transfer rate in sequential use

(iv) effective data transfer rate in random use.

b) RAID-system characteristics (6%)

For a redundant array of independent disks (RAID), an important parameter is ***steady-state throughput*** which is the data transfer rate when serving many independent requests, meaning that some of the requests can be served in parallel. Other relevant parameters are ***storage capacity***, i.e. the maximum amount of user data that can be stored, and ***error tolerance***, i.e. how many disks can fail before some data may be lost.

Consider a redundant storage system built according to the level-4 RAID scheme. Assume that each individual disk is of type Seagate Barracuda specified above and that the array consists of $N + 1$ such disks. Furthermore, assume that the typical use cases of the system are the same as above. For the purpose of the analysis, assume that, for a single disk, the transfer rate is S MB/s in sequential use and R MB/s in random use.

Under these assumptions and assumptions you may add, analyse the following parameters for this storage system:

(i) storage capacity

(ii) error tolerance

(iii) steady-state throughput for repeated sequential reads

(iv) steady-state throughput for repeated sequential writes

(v) steady-state throughput for repeated random reads

(vi) steady-state throughput for repeated random writes.

Practical Part

P1 Concurrent programming: Concurrent heaps (10%)

This task is related to T1 (please, read it too) where we define awesome heaps and give the pseudo-code for the insertion algorithm. In T1 you were to consider the theoretical properties of awesome heaps, and in this task you should write and test a concurrent implementation. Alongside this document, we have provided a sequential—i.e. non-thread-safe—implementation of awesome heaps written in C99.

a) A thread-safe implementation (5%)

We have mirrored the `sequential-heap.[hc]` implementation in `concurrent-heap.[hc]`. Add the relevant Pthreads mechanisms to `concurrent-heap.[hc]` to make the concurrent implementation thread-safe. Both the `init` and `clear` operations are suppossed to be executed by one thread, so only the `insert` operation should be thread-safe.

b) Testing and mental benchmarking (5%)

Assure that your implementation is thread-safe by writing a multi-threaded program which uses your implementation. Discuss in which kind of use scenarios your implementation may have performance benefits over the sequential implementation.

P2 System overload: Operating-system workload (5%)

In this task, you should extend an operating-system kernel with a new service and provide a simple test of your implementation. More specifically, add a new system call to `BUENOS` which enables a userland process to obtain the current workload. The ***workload*** is defined as

$$w := \frac{r}{r+s},$$

where w is the workload, r is the number of threads running or ready to run and s is the number of processes waiting for a resource to become available (i.e. sleeping threads). Since the YAMS CPU has no support for floating-point numbers, you shall return w as a percentage.

`int` `syscall_workload()`

Returns: The number $\lfloor 100 \cdot w \rfloor$ as an integer, with w defined as above.

P3 Write once, debug everywhere: Process tracing (25%)

In this task, you should extend an operating-system kernel with a process-tracing service which makes it possible for one process to trace and modify another. The developed service could be used to write a user-level debugger.

a) Debugger and debugee (10%)

Implement the necessary infrastructure to set up a debugger-debugee relationship between two or more processes. A process opts in to be debugged, thus becoming a ***debugee***, by executing the system call `syscall_traceme`. This will make the parent of this process the ***debugger***.

If a debugee executes the system calls `syscall_fork` or `syscall_exec`, then the child process shall inherit the debugger of its parent, thereby itself becoming a debugee.

The debugger monitors events in its debugees using the system call `syscall_wait_for_debugee`. This will not return until some event happens in one of the debugees. When a debugee stops due to an event (see below), its PID is returned. If a debugee exits, the negative of its PID is returned.

You may freely decide which events stop a debugee, although you must document and argue for your design choices in the written report. As a bare minimum, two events must stop the debugee:

1. Call `syscall_traceme`

   To synchronize debugger and debugee, the debugee shall be stopped when executing this system call. See Listings 1–3.

2. Breakpoint exception

   When the CPU executes the `BREAK` instruction, it generates a breakpoint exception. Currently this results in a kernel panic. You must instead stop the process.

The specifications of the involved system calls are as follows:

int `syscall_traceme`()

Effects: Register the calling process as a debugee of its parent, which becomes the debugger.

Returns: `0` upon success and `-1` otherwise.

int `syscall_wait_for_debugee`()

Effects: Wait for an event in one of the debugees. If an event has already happened, the call shall return immediately.

Returns: If the debugee stopped, its PID is returned. If it exited, the negative of its PID is returned.

Example: In Listing 1 a process forks and the child opts in to being a debugee using the system call `syscall_traceme`. The parent waits for the child to do this using the system call `syscall_wait_for_debugee`.

Listing 1: `tests/debug1.c`

```c
int main() {
  int pid;
  /* fork and trace the child. */
  printf("Forking... ");
  pid = syscall_fork();
  switch (pid) {
  case -1:
    printf("fork() failed\n");
    return -1;
  case 0: /* child */
    syscall_traceme();
    break;
  default: /* parent */
    printf("OK (PID = %d)\n", pid);
    printf("Waiting for child to call `traceme`... ");
    pid = syscall_wait_for_debugee();
    printf("OK (PID = %d)\n", pid);
  }
  return 0;
}
```

Hints:

- Look out for race conditions; If an event happens in a debugee *before* the debugger calls `syscall_wait_for_debugee`, the debugger should still see the event.
- Breakpoint exceptions are handled in `/proc/exception.c`.
- Maybe each debugger process should keep a list of its debugees.

b) Continuing the debugee (5%)

Extend the process-tracing service by making it possible for a debugger to continue execution of a stopped debugee. This is done using the system call `syscall_continue_debugee`. It is an error for a process to try to continue a process which is not its debugee. The specification of this system call is the following:

int `syscall_continue_debugee`(**int** `pid`)

Effects: Continue execution of the process identified by pid, if it is a debugee of the calling process.

Returns: 0 upon success and −1 otherwise.

Example: In Listing 2, the previous example in Listing 1 is extended with the new system call, and a software breakpoint is inserted. Notice how the debugger distinguishes between a stop and an exit.

Listing 2: `tests/debug2.c`

```c
int main() {
  int pid;
  int i;
  /* fork and trace the child. */
  printf("Forking... ");
  pid = syscall_fork();
  switch (pid) {
  case -1:
    printf("fork() failed\n");
    return -1;
  case 0: /* child */
    syscall_traceme();
    for (i = 0; i < 10; i++) {
      printf("[debugee] Before breakpoint (i =%d)\n", i);
      __asm__("BREAK");
      printf("[debugee] After breakpoint (i =%d)\n", i);
    }
    break;
  default: /* parent */
    printf("OK (PID = %d)\n", pid);
    printf("Waiting for child to call `traceme`... ");
    pid = syscall_wait_for_debugee();
    printf("OK (PID = %d)\n", pid);
    printf("[debugger] Continuing debugee (PID = %d)\n", pid);
    syscall_continue_debugee(pid);
    for (;;) {
      pid = syscall_wait_for_debugee();
      if (pid > 0) {
        printf("[debugger] Hit a breakpoint (PID = %d)\n", pid);
        syscall_continue_debugee(pid);
      }
      else {
        printf("[debugger] Debugee exited (PID = %d)\n", -pid);
        break;
      }
    }
  }
  return 0;
}
```

c) Inspecting and modifying the debugees (10%)

Extend the process-tracing service to make it possible for the debugger to inspect and modify its debugees. This task is divided into two: one for memory and one for registers.

c).1 Memory (5%)

When a debugee is stopped, the debugger shall be able to read and write its memory using the system calls `syscall_peek_byte` and `syscall_poke_byte`. It is an error to inspect or modify a running debugee. Likewise it is an error to try to access a process which is not a debugee of the calling process.

int `syscall_peek_byte`(**int** `pid`, **void**∗ `addr`, **unsigned char**∗ `byte`)

Effects: Read one byte from the virtual memory space of the target debugee.

Returns: 0 upon success and −1 otherwise.

int `syscall_poke_byte`(**int** `pid`, **void**∗ `addr`, **unsigned char** `byte`)

Effects: Write one byte to the virtual memory space of the target debugee.

Returns: 0 upon success and −1 otherwise.

Hint: To access the virtual memory space of the target process, you need to look it up in the page table of that process. You can do this manually or rely on the TLB. The former is suggested. Refer to `/vm/pagetable.h` and `/vm/tlb.h` for the relevant structures.

c).2 Registers (5%)

When a debugee is stopped, the debugger shall be able to read and write its CPU registers using the system calls `syscall_peek_register` and `syscall_poke_register`. It is an error to inspect or modify a running debugee. Likewise it is an error to try to access a process which is not a debugee of the calling process.

Registers are numbered as in `/kernel/cswitch.h`, with the addition of the `PC` register which is given the number 29.

int `syscall_peek_register`(**int** `pid`, **int** `reg`, `uint32_t`∗ `val`)

Effects: Read the value of a register in the target process.

Returns: 0 upon success and −1 otherwise.

int `syscall_poke_register`(**int** `pid`, **int** `reg`, `uint32_t` `val`)

Effects: Set the value of a register in the target process.

Returns: 0 upon success and −1 otherwise.

Hint: The CPU context(s) of the target process is stored on its kernel stack. Only the user context should be inspected or modified. Refer to `/kernel/thread.h` and `/kernel/cswitch.h` for the relevant structures.

Example: In Listing 3 the debugger updates the `PC` register of its debugee to skip a call to `printf`.

Listing 3: `tests/debug3.c`

```c
int main() {
  int pid;
  uint32_t pc;
  /* fork and trace the child. */
  printf("Forking... ");
  pid = syscall_fork();
  switch (pid) {
  case -1:
    printf("fork() failed\n");
    return -1;
  case 0: /* child */
    syscall_traceme();
    __asm__("BREAK");
    printf("[debugee] Hello world!\n\n");
    break;
  default: /* parent */
    printf("OK (PID = %d)\n", pid);
    printf("Waiting for child to call `traceme`... ");
    pid = syscall_wait_for_debugee();
    printf("OK (PID = %d)\n", pid);
    printf("[debugger] Continuing debugee (PID = %d)\n", pid);
    syscall_continue_debugee(pid);
    /* run until breakpoint */
    pid = syscall_wait_for_debugee();
    printf("[debugger] Hit breakpoint (PID = %d)\n", pid);
    /* Skip the call to `printf` */
    syscall_peek_register(pid, MIPS_REGISTER_PC, &pc);
    /* The call takes three instructions (found by disassembling) */
    syscall_poke_register(pid, MIPS_REGISTER_PC, pc + 4 * 3);
    printf("[debugger] Setting PC to 0x%x (was 0x%x)\n", pc, pc + 4 * 3);
    syscall_continue_debugee(pid);
    /* run until exit */
    pid = syscall_wait_for_debugee();
    printf("[debugger] Debugee exited (PID = %d)\n", -pid);
  }
  return 0;
}
```

P4 File-system permissions (15%)

File-system permissions are a primitive operating-system security mechanism. The idea is to protect files from unintended use by decorating the files with information about how they may be used, and by whom. The typical approach to file-system permissions is to exploit the concept of users and have users act as owners of files. Owners are distinguished from other users on the system by having permission to specify how their files may be used by others.

Unfortunately, `BUENOS` has no concept of users. On the other hand, a process can be considered as a primitive notion of a "user". For instance, processes, unlike threads, do not share memory. The idea is to leave it up to the process to decide whether to share files with others or not. (Are there any problems with this security model due to how PIDs are assigned in `BUENOS`?)

a) Process-based file ownership (5%)

Add an integer field to both `openfile_entry_t` in `fs/vfs.c`, and `tfs_inode_t` in `fs/tfs.h` to retain and persist the PID of the process that created the file.

Implement the necessary changes to the virtual and trivial file system layers.

Hint: You need to modify the definition of `TFS_BLOCKS_MAX`.

b) Read/write/execute flags—`syscall_create` (5%)

Typically, a user (process) is permitted to do some of the three possible actions with a file: read it, write it, or execute it (as in `syscall_exec`). The usual way to specify file permissions is to designate some bits in an integer variable for permission flags.

We have defined the file-system permission flags in `fs/perm.h`. For instance, a file that can be read and written by the owner, but only read by others, has the following permission flags set: `FS_PERM_NONE | FS_PERM_OWNER_RW | FS_PERM_OTHER_R`.

Extend `syscall_create` to fulfil the following specification:

int `syscall_create`(**char const**∗ `pathname`, **int** `size`, **int** `flags`)

Effects: Create a new file specified by `pathname` to be of the given `size` (in bytes) and to have the given permission `flags` set. The path name must include the mount point (full name would be `[root]shell`, for example). The file must not already exist.

Returns: Return `VFS_OK` on success, or an adequate VFS error on error.

Add an integer field to both `openfile_entry_t` in `fs/vfs.c`, and `tfs_inode_t` in `fs/tfs.h` to retain and persist the permission flags with which the file was created.

c) Read/write/execute flags—`syscall_open` (5%)

Extend `syscall_open` to fulfil the following specification:

int `syscall_open` (**char**∗ `pathname`, **int** `flags`)

Effects: Open the file specified by `pathname`, for operations given by `flags`. The flags are the same as for `syscall_create`. The path name must include the mount point.

Returns: Return an open file identifier. Open file identifiers are non-negative integers. On error, negative value is returned. In particular, `VFS_PERM` (defined in `fs/vfs.h`) if file could not be opened due to a insufficient permissions.

Add a field to `openfile_entry_t` in `fs/vfs.c` to retain the intent for which the file was opened. Trying to use a file in an unintended manner should result in a `VFS_PERM` error (defined in `fs/vfs.h`). For instance, a file opened with the flags `FS_PERM_READ` should yield `VFS_PERM` on an otherwise valid `syscall_write`.

Implement the necessary changes to the virtual and trivial file-system layers, as well as the process implementation.

References

[`BUENOS` Roadmap] Juha Aatrokoski, Timo Lilja, Leena Salmela, Teemu J. Takanen, and Aleksi Virtanen. *Roadmap to the `BUENOS` System*, Version 1.1.2, Aalto University School of Science (2012).

[Algorithms Book] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, 3rd ed., The MIT Press (2009).

[Dinosaur Book] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*, 9th ed., Wiley (2014).

[Weinstock and Wulf 1988] Charles B. Weinstock and William A. Wulf, Quick fit: An efficient algorithm for heap storage allocation. *SIGPLAN Notices* **23**(10) (1988), 141–148.