Grenoble Universities (GrenobleINP & Université Joseph Fourier)
M1 MoSIG
Academic year 2013-2014

GINF41E0 — Principles of operating systems — Final exam December 2013

Duration : 3 hours
Documents allowed. Electronic devices forbidden.
The number of points per exercise is only provided for indicative purposes.
The grade will take the quality of the presentation into account.

This exam is made of two parts: part I (exercises 1—2) and part II (exercise 3).
Use distinct answer sheets for each part.

1 Processes and threads (3 points)

1.1 Describe three different circumstances in which a mode switch from user mode to kernel mode may occur. Expected answer length : about 5 lines.

1.2 In the lecture on threads, we have seen that threads (i) can be implemented either as user-level or kernel-level constructs and (ii) can be based either on a cooperative scheduling model or on a preemptive scheduling model. This leads to four possible designs (user-level cooperative, user-level preemptive, kernel-level cooperative, kernel-level preemptive). Among them, only three are actually used in practice. Explain which one is generally avoided and why. Expected answer length: about 10 lines.

1.3 We consider the following C program, whose only purpose is to determine how many threads can be created within a given process.

```
1   /* #include directives omitted for simplification */
2
3   void* behavior(void *p) {
4       while(1) {
5           sleep(1000); /* in seconds */
6       }
7   }
8
9   int main (int argc, char **argv) {
10      int i, j, r;
11      pthread_t t;
12      void *b;
13
14      if (argc != 2) {
15         printf("usage: %s heap_increment\n", argv[0]); exit(-1);
16      }
17      i = atoi(argv[1]);
18      b = sbrk(i); /* increment the heap size by i bytes */
19      if (b == (void*)-1) {
20          printf("failed to increase heap size\n"); exit(-1);
21      }
22      j = 0;
23      while (1) { /* create threads until pthread_create returns an error */
24          r = pthread_create(&t, NULL, behavior, NULL);
25          if (r != 0) {
```

```
26                  break;
27              }
28              j++;
29          }
30          printf("Heap size increment: %d bytes. ", i);
31          printf("Nb of successfully created threads: %d\n", j);
32          exit(0); /* exit process (all threads will be destroyed) */
33      }
```

First, we compile and run the program on a 32-bit Linux system. We observe that, when the size of the heap increases, the maximum number of threads that can be created decreases, as shown with the following traces.

```
shell$ ./test 400000000
Heap size increment: 400000000 bytes. Nb of successfully created threads: 462
shell$ ./test 800000000
Heap size increment: 800000000 bytes. Nb of successfully created threads: 414
shell$ ./test 1200000000
Heap size increment: 1200000000 bytes. Nb of successfully created threads: 367
```

Second, we perform the same experiment on a 64-bit Linux system (with the same amount of physical RAM as the first system). In contrast to the previous experiment, we observe that the size of the heap does not impact the maximum number of threads that can be created (in any case, we manage to create the same number of threads, which equals a maximum bound defined by the operating system).

```
shell$ ./test 400000000
Heap size increment: 400000000 bytes. Nb of successfully created threads: 7620
shell$ ./test 800000000
Heap size increment: 800000000 bytes. Nb of successfully created threads: 7620
shell$ ./test 1200000000
Heap size increment: 1200000000 bytes. Nb of successfully created threads: 7620
```

Explain what is the limiting factor in the first experiment and why it is not problematic in the second experiment. Expected answer length: about 10 lines.

2 Virtual memory management (7 points)

Advice:

- Read the whole exercise several times before preparing and writing your answers.
- Question 1 is independent from the rest of the exercise and can be answered without studying the complete specifications.

In this problem, we consider a simplified processor architecture (called *MyProc*) inspired from the (64-bit) Intel x86-64 architecture. The *MyProc* architecture is a 64-bit architecture, in the sense that it features 64-bit registers and a 64-bit memory data bus. However, the virtual addresses are stored on 48 bits only (the 16 most significant bits of a 64-bit pointer to virtual memory are simply ignored) and the physical addresses are stored on 52 bits only (the 12 most significant bits of a 64-bit pointer to physical memory are simply ignored).

Upon a TLB miss, the processor automatically attempts to perform a lookup in the current paging structure (whose root is stored in the `CR3` register) in order to determine the mapping between the requested virtual address and its corresponding physical address. The format of the paging structure for a process address space (with a page size of 4 kiloBytes) is depicted in Figure 1.
Note that there are four different levels: a "*Page Map Level 4*" (PML4), *Page-Directory-Pointer Tables* (PDPT), *Page Directories* (PD) and *Page Tables* (PT). Each instance of these structures is stored on a

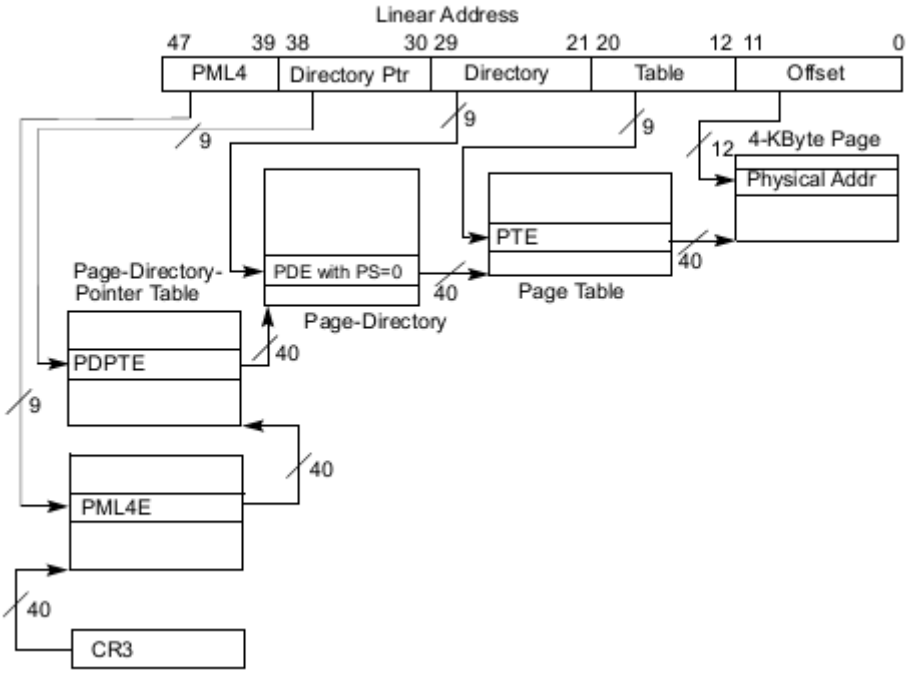

Figure 1: *MyProc* architecture — Virtual address translation with a 4kB page

4kB page, and stores 2^9 entries of 8 bytes each. Each entry contains several fields (for simplicity, we omit some of them): a *present* bit (P), a *read/write* bit (R/W), a *user/supervisor* bit (U/S) and a *physical page number* PPN (pointing to the requested physical page in the case of a PT entry, or pointing to the next level structure, in the case of a PML4 entry, a PDPT entry and a PD entry).

When a page fault occurs, the processor triggers a trap to supervisor (i.e., kernel) mode and jumps to the starting address of the corresponding exception handler (registered by the OS kernel). In addition, the processor sets up two registers with details on the circumstances of the page fault:

- The `ErrorCode` register contains the following bits:
  - *Pstate*: set to 0 if the fault was caused by a non-present page; set to 1 if the fault was caused by a protection violation
  - *RWstate*: set to 0 if the access causing the fault was a read; set to 1 if the access was a write
  - *USstate*: set to 0 if the processor was executing in user mode; set to 1 if it was running in supervisor mode
  - *Istate*: set to 1 if the fault was caused by an instruction fetch; set to 0 otherwise
- The `CR2` register contains the virtual address whose (virtual-to-physical) translation caused the page fault.

2.1 Propose extensions/modifications to the format of the hardware paging structure allowing to use three different page sizes (for different address ranges in the same address space): 4 kB (as in the original design), 2 MegaBytes and 1 GigaByte. In particular, describe the following points:

- The data structures that must be introduced or removed in each configuration.
- The fields that must be added in each entry of the various structures (PML4, PDPT, etc.).
- How each structure is indexed.
- The number of bits used for the offset within the page.

It may be helpful to illustrate/summarize you design with one or several figures.

2.2 We want to design a simple OS kernel for the *MyProc* architecture, with support for paging in/out individual pages from/to disk.

For simplicity, we make the following assumptions:

- The machine has a single processor.
- We only consider single-threaded processes.
- We only consider three regions within the user-level part of a process address space: the program code, the heap and the stack.
- Applications do not create/modify/delete memory mappings in their address spaces (e.g., using system calls like `mmap()`).
- The execution of the kernel code never triggers a page fault.
- A physical memory page is never shared between multiple virtual memory pages (in the same virtual memory address space or in different memory address spaces). Also, the intermediate levels of a paging structure (e.g., PDPT, PD, etc.) are never shared (for instance, there is never more than one reference pointing to a given PDPT). As a consequence, the kernel does not implement features such as copy-on-write upon fork, sharing of code pages, shared memory segments, etc.
- The size of each region within a process address space (user code, user heap, user stack, etc.) is a multiple of the smallest page size (4 kB).
- Processes are identified with an integer (Pid) and the kernel maintains a global variable named `current` to store the Pid of the currently running process.

Describe the different circumstances that may trigger a page fault. Then, describe the data structure(s) that the kernel must define and manage in order to make the distinction between these different circumstances.

2.3 Describe the data structure(s) that the kernel must define and manage in order to implement support for paging in/out individual pages from/to a disk swap partition. In particular, consider the following aspects:

- Keeping track of the available free space in physical RAM and on the swap partition.
- Determining where a given (swapped out) page can be found in the swap partition.

When designing your solution, take into account the fact that there are three different possible page sizes.
Note: You are not required to discuss the data structures and implementation details related to page replacement decisions.

For simplicity, we make the following assumptions:

- The swap partition has a fixed and known size, which is a multiple of the smallest page size.
- All the bytes of a (swapped-out) page are stored contiguously in the swap partition (for any page size).
- The kernel has the following interface to access the swap partition on disk:
  - The disk uses blocks of 4kB each and numbered between 0 and `MAXBLOCK`.
  - `void read_blocks_from_swap(void *buf, int b, int nb_blocks)`: read `nb_blocks` contiguous blocks from the swap partition, starting from block number `b` and put the read bytes in a contiguous buffer starting at physical address `buf`.

- `void write_blocks_to_swap(void *buf, int b, int nb_blocks)`: read `nb_blocks` blocks in a contiguous buffer starting from physical address `buf` and write them in contiguous blocks on the swap partition, starting from block number `b`.
- The above functions may block the requesting process P while waiting for the disk's reply. In this case, the kernel will automatically switch to another process and resume P when the disk request has completed. Thus, you are not required to discuss details related to context switches in your answers.

2.4 Give the pseudo-code of the kernel's page fault handler.

For simplicity, we make the following assumptions:

- A memory protection violation must simply result in the destruction of the current process, via a call to `destroy_current_process()`. This function handles all the necessary steps to elect another process and to run it on the CPU.
- The pages that contain intermediate paging structures (PDPT, PD, PT) are "pinned" in physical memory, i.e., they are never swapped out to disk.
- The `void *allocate_physical_frame(int nb)` function returns the starting address of `nb` contiguous (physical) page frames (of size 4kB each). If there is not enough free space in physical memory, this function internally handles the necessary actions (page replacement decision and eviction). Thus, you are not required to describe the details of the kernel's page replacement module.
- The `void read_pages_from_regular_file(char *file_path, int offset, int nb, void *start)` reads `nb` blocks (of size 4kB each) from the regular file on disk (e.g., an executable file) whose path is `file_path`, starting at offset bytes after the start of the file and store the read bytes in a contiguous memory buffer starting from physical address `start` (we assume that `offset` is a multiple of 4kB).

3 Christmas Workshop (10 points)

Times are changing, this year, in his workshop, Santa Claus has replaced elves with elvish robots driven by threads running on a supercomputer. Still, toys keep on being manufactured in the workshop using consumable materials such as wood, screws or paint and shared tools such as screwdriver, saw or brush. In the general case, we assume that there are m different kinds of consumable materials and that the stock is denoted $C = C_1, \ldots, C_m$ where each C_y is the number of available unit of the according material kind. Similarly, there are n different kinds of shared tools and the stock of tools is denoted $S = S_1, \ldots, S_n$. Toys are constructed from raw materials by following blueprint instructions which are made of a sequence of pieces manufacturing and assembly steps. There are k different toys and the construction of a given toy x requires an elfish robot to perform $t = time(x)$ successive operations where the y^{th} operation consumes $materials(x, y, i)$ units of materials of type i and uses $tools(x, y, j)$ copies of tool of type j.

Consumable materials are collected from external facilities by actual elves using elvish trucks and stocks modifications are injected concurrently in the system by using various terminals available in the storage area. We assume that the production of consumable materials is infinite, it just takes time for the elves to bring them in the storage area. Santa Claus has made some effort by learning multithreaded programming and writing by himself the code executed by the driving threads. Nevertheless, as he is not confident with synchronization he asked you to write the synchronization code for resources management. The code already written by Santa Claus, uses the following functions :

- `void *init(int m, int *C_init, int n, int *S_init, int k, int *time, int ***materials, int ***tools)` is called when starting the program that creates threads. It returns a pointer that we will call *resources handle* by the following. All the parameter are initial values for the workshop. `C_init`, `S_init` and `time` are stored as one dimensional arrays and `materials` as well as `tools` are stored as

three dimensional arrays. Santa Claus plans for expansion of its workshop include the concurrent management of several production sites as well as transportation facilities between sites. Thus, this function should not store anything in any global variable as it will hinder these evolutions of the system.

- `void refill(void *rh, int *C_refill)`
  is called by elves to add consumable materials units given by `C_refill` to the stock, `rh` is the resources handle returned by `init`.

- `void use(void *rh, int *C_use)`
  is called by elvish robots to gather consumable materials given by `C_use` required by a toys manufacturing step. This function should block until resources are available and given to the calling thread. Notice that a group of materials added to the stock using `refill` can be used one at a time using this function, this means that the parameter `C_use` in this function is not necessarily the same as one of the parameters `C_refill` given to the `refill` function.

- `void hold(void *rh, int *S_hold)`
  is called by elvish robots to grab shared tools given by `S_hold` required by a toy manufacturing step. This function should block until resources are available and given to the calling thread.

- `void release(void *rh, int *S_release)`
  is called by elvish robots to release shared tools given by `S_release` and previously grabbed using `hold`. Notice that a group of tools grabbed using `hold` can be released one at a time using this function, this means that the parameter `S_release` in this function is not necessarily the same as one of the parameters `S_hold` given to the `hold` function.

At the end of the day, the elvish robots place each completed toy in a shared delivery queue and elves pick toys in this queue to fill Santa Claus sleighs. This queue, initialized by the main program, is a solution to the producer/consumer problem : a synchronized FIFO queue supporting two operations :

- `void produce(void *queue, toy t)`

- `toy consume(void *queue)`

To be as efficient as possible, Santa Claus has a large number of sleighs, sufficiently large to enable elves to keep on filling them while he continuously travels to deliver christmas gifts. Each sleigh has a fixed capacity of Max pounds and each toy x weighs a number of pounds $weight(x)$ that depends on the materials it is made of. After having placed some toys in the sleigh, the remaining capacity can be completed by coal destined to the naughty kids. In this process, we assume that each toy eventually gets placed in one of the sleigh and that the time it takes to fill a sleigh is bounded.

In the following, at your convenience and when this applies, you can answer each question about implementation separately or give a single implementation that answers all the questions. You are required to write your implementation in correct C using POSIX threads. You are allowed to use the following functions :

- `int compare(int *v1, int *v2, int n)`
  `v1` and `v2` are two arrays of size `n` that represent vectors. Returns -1, 0 or 1 if, respectively, `v1` is lower than `v2` in all its coordinates, `v1` is not comparable to `v2` or `v1` is greater than `v2` in all its coordinates.

- `void add(int *v1, int *v2, int *v3, int n)`
  `v1`, `v2` and `v3` are three arrays of size `n` that represent vectors. Stores `v1`+`v2` into `v3`. `v3` can be the same address as `v1` or `v2`.

- `void sub(int *v1, int *v2, int *v3, int n)`
  `v1`, `v2` and `v3` are three arrays of size `n` that represent vectors. Stores `v1`-`v2` into `v3`. `v3` can be the same address as `v1` or `v2`.

Questions :

1. Give an implementation of the `init`, `refill`, `use`, `hold` and `release` functions.
2. Sometimes, the production can be stalled for some time, which means that all the threads are blocked for a bounded duration before being able to pursue their task. Give an exemple of such a situation.
3. Sometimes, the production can deadlock.
   (a) Give an exemple of such a situation.
   (b) In this situation, do you think it is better to handle deadlocks by detecting and correcting them or by preventing them? Explain why.
   (c) Give the modifications required in your implementation of the `init`, `refill`, `use`, `hold` and `release` functions and in the threads management in the workshop to enforce the deadlock handling strategy you proposed.
4. Santa Claus is thinking about replacing elves at the end of the delivery queue by new models of elvish robots driven by threads. We assume that each of these robots is in charge of filling a distinct sleigh. Explain what data and which shared structures should be given as arguments to these threads function and give the implementation of these threads.
5. Do you think it is possible to choose toys placement so that each filled sleigh only contains toys? If so explain how, otherwise, explain why it is not possible and propose a way to place toys that tries to minimize the quantity of coal contained in each sleigh.
6. Can your placement policy lead to starvation? If so explain why and explain how it can be modified to avoid starvation, otherwise just explain why.
7. Actually, the proportion of nice kids to naughty kids P is known and each nice kid deserves a toy while each naughty kid deserves a pound of coal. The elves in charge of filling the sleighs try to match the proportion of toys to coal with the proportion of nice to naughty kids. Is it always possible to completely fill the sleigh while matching asymptotically the desired proportion P of toys to coal? Explain your answer.