Wednesday, October 27, 2021,
17:00 – 18:30 written part
19:30 – 21:00 practical part

Practice Questions

Instructions for the written part of the exam:

- This exam is closed book. You may bring one A4 page of handwritten notes.
- No electronic tools whatsoever. Cheating, in whatever form, leads to exclusion of the exam, grade F, and a notification to the university.
- Write your answers in the space provided below the problem. Try not to make a mess. Clearly indicate your final answer. If you run out of space, ask the TA for additional sheets.
- You can write your answers in English or Korean. Your handwriting must be legible.
- Write down the logical steps of your computation, not just the final answer.
- This written part of the exam contains *q* questions with a maximum score of *p* points.
- The problems are of varying difficulty. The point value of each problem is indicated. Solve the easy problems quickly and then go back to the harder problems.
- If you finish before the time is up, stay seated and raise your hand. The TA will collect your exam and then let you go.

Good luck!

Problem	Points
1	
2	
3	
...	
Total	

Practice questions for the written part

1. When approximately did the first timesharing operating systems appear?
2. What are the three fundamental abstractions provided by an operating system and how do they relate to each other?
3. What illusions does the process abstraction provide to a process?
4. UNIX was born out of the failure of which operating system project?
5. What is POSIX?
6. List two principles of Unix philosophy.
7. What is the difference between the system call interface and the C standard library?
8. What system calls are used for (Unix) input/output?
9. By default, how do system calls report failure? Is there a way to know the exact cause of a failure?
10. Explain the benefits of representing devices and kernel information as files.
11. What types of files do you know?
12. What is a file descriptor?
13. Where is the current file position stored?
14. Describe the data structures used to manage open files in a system and show how they relate to teach other. Also indicate which are private to a process and which are global.
15. Is the file position shared between two processes who independently open the same file?
16. Is the file position shared between a parent and child process if the parent opened the file?
17. What is a short count? Give two examples for which short counts can occur.
18. What is the difference between Unix I/O and Standard I/O?
19. What are API functions do you need to know to list the contents of a directory?

20. Explain to which memory regions the variables of the following program are allocated. Illustrate the memory contents and variables similar to how we learned in ‘Variables and Memory Recap’, p. 25-30.

```
int global = 5;
int foo(int p1, int p2) {
  int local, *lp, **lpp;
  lp = &global; lpp = &lp;
  local = **lpp;
  return local+1;
}
```

21. How is it possible to execute two 32-bit processes with 4 GiB of virtual address space each on a computer with only 2 GiB of physical memory?

22. What four main functions are provided by the dynamic memory allocator in C?

23. What is internal fragmentation? External fragmentation?

24. `free(void *ptr)` only provides a pointer of the block to free but not its size. How do we know how much memory to free?

25. Explain how a heap with an implicit free list operates.

26. Explain coalescing for an implicit list with boundary tags.

27. Do explicit free lists increase internal fragmentation?

28. Why is the additional overhead of explicit free lists justified?

29. Compare the garbage collection techniques mark-and-sweep and reference counting.

30. Explain the difference between a program and a process.

31. Which system calls are used for process management?

32. A process that is waiting for a system call is in which process state?

33. Why is a process that is ‘ready’ not currently executing?

34. What happens in a context switch?

35. What is the PCB and how is it involved in context switching?

36. What are *all* possible outputs of the following program?

```
void main(void) {
  if (fork() == 0) {
    printf("1\n"); sleep(1); printf("2\n");
  } else {
    printf("3\n"); wait(NULL); printf("4\n");
  }
}
```

37. What is the output of the following program? Explain.

```
void main(void) {
  if (fork() == 0) {
    for (int i=0; i<2; i++) {
      printf("%d\n", getppid()); sleep(2);
    }
  }
  sleep(1);
}
```

38. Why do we need exceptional control flow in a computer system?

39. What different kinds of synchronous exceptions do you know, and how do they differ?

40. What are asynchronous exceptions? Give two examples of an asynchronous exception.

41. What function does the interrupt vector table have? How is it initialized (by whom)?

42. Explain the difference between exceptions and signals.

43. What is a pending signal?

44. Are signals of the same type queued?

Practice questions for the practical part

Questions for the practical part will be delivered electronic form. Most questions come with skeleton code that you have to fill in. During the exam, we "sell" hints for a point deduction. You will work with a locked-down version (no network access) of the CSAP VM.

1. To make or not to make

Add the following targets to the Makefile in directory 1/

- fibonacci:
  Compiles main.c, fib.c into fibonacci whenever main.c, fib.c, or fib.h have changed.
- clean:
  Removes the binary fibonacci
- all:
  The default target. Builds the 'fibonacci' target

If implemented correctly, you will see the following output for the sequence of commands:

```
$ ls
fib.c  fib.h  main.c  Makefile
$ make
gcc -std=c99 -g -o fibonacci main.c fib.c
$ make
make: Nothing to be done for 'all'.
$ ./fibonacci 10
fibonnaci(10) = 55
$ touch fib.c
$ make
gcc -std=c99 -g -o fibonacci main.c fib.c
$ make clean
rm -f fibonacci
$ ls
fib.c  fib.h  main.c  Makefile
$
```

2. Size Matters

Write a program that recursively traverses a directory tree and prints out the total size of all regular files in bytes.

3. Controlling your Flock

Given is a child program that expects an index on the command line and then sleeps a little.
$ ./child 3
[7375] Hi, this is child 3.
$ ./child 77
[7378] Hi, this is child 77.
$

Write a program that

- reads *nproc*, the number of children to create, from the command line (provided)
- executes *nproc* child processes with indices 1...*nproc*
- waits for all nproc children and prints
  - whether they have terminated normally or abnormally
  - if they terminated normally what their exit code was

If implemented correctly, you will see output similar to the following:

```
$ ./control 4
[64455] Hi, this is child 2.
[64454] Hi, this is child 1.
[64456] Hi, this is child 3.
[64457] Hi, this is child 4.
Child 64457 terminated normally with exit code 4.
Child 64456 terminated normally with exit code 3.
Child 64455 terminated normally with exit code 2.
Child 64454 terminated normally with exit code 1.
$
```