CS-202 OS Exercises: L1 - L3

26.02.2025

This exercise set covers the concepts discussed in the first two weeks of the course (*Introduction*, *All about processes*, *Sharing the CPU*). We advise that you work through it sequentially, referring back to lecture slides or videos as necessary. If anything is unclear, or if you could benefit from discussing a particular concept in depth, please seek an assistant's help.

Exercise 1: Process, program, thread

Fill in the blanks using the words *process*, *program*, and *thread*.

- A compiler takes a C _____ as input and outputs an assembly or executable _____.
- A file may contain a _____. Double-clicking it (or typing it into the command line) creates a _____.
- A _____ may be stored on disk or in memory. A _____ consists of one or many _____s that execute an in-memory copy of the _____.
- A _____ associates a unique ID with the memory image and CPU context required to execute a _____.

Exercise 2: The memory image of a process

Fill in the blanks using the words *data*, *text*, *stack*, and *heap.*

- The _____ and _____ segments have a fixed size, which is known at compile time.
- The size of _____ changes with function calls / returns.
- The _____ and the _____ grow in opposite directions.
- The data in the _____ disappears when the function that created it returns. The data in the _____ does not.

Exercise 3: CPU state and main memory during execution

Remember that the CPU contains a set of *registers*, which are small data-holding places that are close to computational circuits. The CPU loads data from the main memory into registers, executes operations on in-register data, and writes back from registers to main memory. There are also special registers that keep track of the current thread's state. One such register is the instruction pointer (IP), which stores the address of the next instruction to be executed.

The tables below describe the memory and CPU states for a thread. Draw the tables corresponding to the next four steps of execution, assuming that no interrupt will be raised.

Memory (address, value):

data:	
`0×000000f0cacc1a00`:	`0×0000000000000002`
`0×000000f0cacc1a10`:	`0×0000000000000005`
`0×000000f0cacc1a20`:	`0×00000000000000ff5`
text:	
`0×00000000c0ffee00`:	Load `0×000000f0cacc1a00` into r0.
`0×00000000c0ffee10`:	Add 3 to the value in r0, put the result in r1.
`0×00000000c0ffee20`:	If the value in r1 is less than 7, jump to `0×00000000c0ffee50`.
`0×00000000c0ffee40`:	Jump to `0×00000000c0ffee00`.
`0×00000000c0ffee50`:	Store the value in r1 at `0×000000f0cacc1a20`.

CPU (register, value):

r0:	0
r1:	0
IP:	`0×00000000c0ffee00`

Exercise 4: Variables, stack frames.

Answer the following questions about the C program below.

Clarification: When function f1 calls function f2, f1 must communicate to f2 the value(s) of the argument(s). In class, we did not discuss where f1 stores the arguments. In principle, they may be stored in registers or in the stack (though in modern systems, they are mostly stored in registers). You may assume either approach. Assuming that they are stored in registers makes the solution simpler, because you do not need to show them in the stack.

```
int g = 0;
int main() {
    g = 1;
    foo(2);
}
int foo(int a) {
    int l = 5;
    if (a < 3)
       return bar(l);
    return 0;
}
int bar(int l) {
    return g + l;
}
```

1) Draw the memory image of the program at the start of main(). Which variables are present?

2) Draw the stack just before the first line of foo() is executed. Show how many stack frames there are, and clearly indicate where each variable is located.

3) Draw the stack just before the first line of bar() is executed. Show how many stack frames there are, and clearly indicate where each variable is located.

Exercise 5: Time-sharing the CPU

Below, you are given two scenarios consisting of the memory image and the CPU context. For each scenario, identify what code will run next, and what the CPU's privilege level will be after the current instruction is executed.

Scenario 1:

heap:	
0×8badf00d00000000:	0×0000000000000014
stack:	
0×000decafbad00000:	0×0000000000000150
…	
data:	
0×000000f0cacc1a00:	0×0000000000000002
0×000000f0cacc1a10:	0×0000000000000005
0×000000f0cacc1a20:	0×0000000000000ff5
text:	
0×00000000c0ffee00:	Syscall.

r0:	0
r1:	0
IP:	0×00000000c0ffee00

What code will run next?

What will the CPU's privilege level be?

Scenario 2:

heap:	
0×8badf00d00000000:	0×0000000000000014
stack:	
0×000decafbad00000:	0×0000000000000150
…	
data:	
0×000000f0cacc1a00:	0×0000000000000002
0×000000f0cacc1a10:	0×0000000000000005
0×000000f0cacc1a20:	0×0000000000000ff5
text:	
0×00000000c0ffee00:	Divide the value in r0 by the value in r1, put the result in r1.

r0:	5
r1:	0
IP:	0×00000000c0ffee00

What code will run next?

What will the CPU's privilege level be?

Exercise 6: Scheduling states

Consider the following initial state.
There are three single-thread processes. Thread T0 belongs to process P0, which is executing prog0, T1 belongs to P1 executing prog1, and T2 belongs to P2 executing prog2.

After each event, list all threads along with their scheduling status (running, ready, blocked). Also indicate which program is running (e.g., the kernel - scheduler, the kernel - read syscall handler, prog0,).

Initial state: *T0: running, T1: ready, T2: ready - the CPU is running prog0.*

- *T0 makes a read syscall:*
- *The kernel starts a disk read for T0 and schedules T1 in the meantime:*
- *The kernel receives an interrupt indicating that the disk read is done:*
- *The kernel schedules T0:*
- *The CPU receives a timer interrupt set by the scheduler:*
- *The kernel schedules T2:*
- *T2 makes a fork syscall:*
- *The kernel handles the syscall, and schedules the fresh thread T3 belonging to the child process:*
- *T3 calls execvp("ls"):*
- *The kernel handles the syscall and returns control back to T3:*

Exercise 7: fork,exit,wait

Answer the following questions about the two C programs.

```
int main() {
    int f = fork();
    if (f == 0) {
        foo();
    }
    wait(f);
    bar();
}
void foo() {
    print("Hello ");
    exit();
}
void bar() {
    print("World!");
}
```

1) What does this program print?

2) What could the program print if the wait() call was omitted?

3) What do you think might happen if the exit() call was omitted?

(Advanced)

```
int g = 3;
int main() {
    int f = fork();
    if (f == 0) {
        foo();
    }
    wait(f);
    print("World");
    exit();
}
void foo() {
    // g--; // (1)
    if (g > 0) {
        print("Hello ");
        int f = fork();
        if (f == 0) {
            foo();
        }
        // g--; // (2)
        wait(f);
        exit();
    }
}
```

1) This program does not terminate. Explain why and identify what the program will print.

2) If we uncommented the line marked with // (1), would the program terminate? If so, what would it print?

3) If we uncommented the line marked with // (2), would the program terminate? If so, what would it print?

Exercise 8 (Advanced): Guessing how threads work

In the C program below, `spawn_thread(foo, 4)` creates a thread within the same process, which starts executing foo with argument 4. We have mentioned threads in class but not said exactly how they work. For this exercise, we would like you to make an informed guess. You are welcome and encouraged to discuss your guess — and your doubts — with the TAs.

```
int g = 0;
int main() {
    g = 1;
    spawn_thread(foo, 2);
    foo(4);
}
int foo(int a) {
    int l = 5;
    if (a < 3)
       return bar(l);
    return 0;
}
int bar(int l) {
    return g + l;
}
```

1) Draw the memory image of the program just after the call to spawn_thread. How many copies of '`l`' are there? How many copies of '`g`'?

2) Assume that the original thread returns from foo() at the same time the new thread returns from bar(). Draw the memory image of the process just before they both return.