New York University

CSCI-UA.0202-003: Operating Systems (Undergrad): Fall 2025

Quiz 1

- Write your full name on both:
  - the bubble sheet in the “Name” field
  - the quiz booklet
- Write your NYU NetID on the quiz booklet and the bubble sheet in the “ID” field
- Use a #2 pencil to fill in your answers on the bubble sheet (preferred, but you can also use a pen)
- This quiz contains 6 questions only. Each question has choices from A to D
- Fill the bubbles completely by darkening the entire circle, as shown in the example
- Only mark answers for questions 1-6. Do not mark any bubbles beyond question 6
- Choose only one answer per question
- Submit your bubble sheet **together with your quiz booklet**

Name:

NetId:

1. What is a key difference in how registers are managed between a standard function call (using the `call` instruction) and a system call (using the `syscall` instruction) on x86-64?

   (A) In a standard function call, the caller must save all registers, whereas in a system call, the kernel saves all registers.
   (B) Standard function calls use a mix of caller-save and callee-save registers, while for a system call, the kernel is responsible for preserving all registers except for the return value in `%rax` .
   (C) The return value of a standard function is placed in `%rax` , while the return value of a system call is placed on the stack.
   (D) The `syscall` instruction is just a more privileged version of the `call` instruction, but they use identical register-saving conventions.

2. In last Tuesday's lecture, we showed how a shell implements output redirection (e.g., `command > foo`). Which sequence of system calls in the child process correctly redirects its standard output to a file before calling `exec()` ?

   (A) `open("/tmp/foo", ...)` to get a new file descriptor, then use that descriptor in a special argument to `exec()` .
   (B) `wait(0)` to ensure the parent is paused, then `open("/tmp/foo", ...)` to get file descriptor 1.
   (C) `open("/tmp/foo", ...)` first, then `close(1)` to detach the terminal from the process.
   (D) `close(1)` to free the standard output file descriptor, then `open("/tmp/foo", ...)` so the new file is assigned the newly-freed file descriptor 1.

3. Analyze the following C code. What will be printed to the console when it is executed?

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    int x = 100;
    pid_t pid = fork();

    if (pid == 0) { // Child process
        x = 200;
        printf("Child x = %d\n", x);
    } else { // Parent process
        wait(NULL); // Wait for child to finish
        printf("Parent x = %d\n", x);
    }
    return 0;
}
```

(A) `Child x = 200` followed by `Parent x = 200`.
(B) `Child x = 200` followed by `Parent x = 100`.
(C) `Parent x = 100` followed by `Child x = 200`.
(D) The output is non-deterministic because the processes share the variable `x`.

4. Consider the following C code. In which memory segment would the string `"Hello"` and the memory block pointed to by `greeting` be located after line 6 executes?

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5      char *greeting = malloc(6);
6      strcpy(greeting, "Hello");
7      // ...
8  }
```

(A) `"Hello"` is in the Stack; the `greeting` buffer is in the Heap.
(B) `"Hello"` is in the Text section; the `greeting` buffer is in the Heap.
(C) Both `"Hello"` and the `greeting` buffer are in the Heap.
(D) Both `"Hello"` and the `greeting` buffer are in the Stack.

5. How does the operating system create the illusion that every process has its own private set of CPU registers?

   (A) Each process is assigned to a different physical CPU core that has its own dedicated registers.
   (B) The C compiler generates code to save all registers to the stack before any system call.
   (C) Before switching context from one process to another, the OS kernel saves the current CPU registers to the outgoing process's Process Control Block (PCB) and loads the registers from the incoming process's PCB.
   (D) Processes don't use the physical CPU registers directly; they operate on a virtual set of registers simulated in memory by the OS.

6. A process with PID 500 has a local variable `int count = 10;` in its `main` function. It then successfully calls an `exec()` variant to run a new program. Which of the following statements is true about the process state *after* the `exec()` call succeeds?

   (A) The PID is changed to a new value, and the memory for the `count` variable is gone.
   (B) The PID remains 500, but the memory space, including the stack frame containing the `count` variable, is discarded and replaced.
   (C) The PID remains 500, and the new program can access the `count` variable with its value of 10.
   (D) The `exec()` call will fail because local variables on the stack cannot exist when a new program starts.