Multiprocessing: processes; fork

COSC 208, Introduction to Computer Systems, 2021-11-08

Announcements

- Project 2 Part B due tomorrow at 11pm

Outline

- Warm-up
- System calls
- Creating processes: actual code & `fork`

Warm-up:

Q1: *True or False*

1. Code stored on secondary storage (e.g., a solid state drive) is called a process -- false; it is a program
2. Each process has its own code, heap, stack, and register values -- true
3. The CPU is in user mode when executing application code, and kernel mode when executing OS code -- true
4. A process can directly execute instructions on the CPU -- true; user mode uses Limited Restricted Execution (OS isn't an intermediary until privileged instructions)
5. A process can directly access input and output ports -- false; indirect, need a system call

System calls

- Invoked via a special assembly instruction: trap (generic) or `svc` (on ARM)
    - Example program

```c
#include <stdio.h>
#include <unistd.h>
int user() {
    int uid = getuid();
    return uid;
}
int main() {
    int u = user();
    printf("User %d is running this process\n", u);
}
```

    - Assembly code

```
0000000004006ac <user>:
    4006ac: d10083ff    sub sp, sp, #0x20
    4006b0: f9000bfe    str x30, [sp, #16]
    4006b4: 94007713    bl  41e300 <__getuid>
    4006b8: b9000fe0    str w0, [sp, #12]
    4006bc: b9400fe0    ldr w0, [sp, #12]
    4006c0: f9400bfe    ldr x30, [sp, #16]
    4006c4: 910083ff    add sp, sp, #0x20
    4006c8: d65f03c0    ret
000000000041e300 <__getuid>:
    41e300: d28015c8    mov x8, #0xae
    41e304: d4000001    svc #0x0
    41e308: d65f03c0    ret
```

```
efourquet@ulna:~$ cd demo/
efourquet@ulna:~/demo$ clang -o process.o -static process.c
efourquet@ulna:~/demo$ objdump -d process.o >process.s
efourquet@ulna:~/demo$ strace -i ./process
echo $UID
```

SVC

- Functions in the C standard library that involve a privileged operation (e.g., `printf`) put the system call number in a register and invoke a trap instruction — programmer doesn't have to worry about these details; they can just call the appropriate function in the C standard library
- `svc` makes system calls; 0xae 174 call number; go to kernel, save registers
- When `svc` is executed
    1. CPU saves registers to the kernel stack — kernel stack is at a fixed location in memory
        - *Why do we need to save the registers?* — so we can return to `user` when `__getuid` is done
    2. CPU switches to kernel mode
    3. CPU uses system call number to index into table of trap handlers
        - At boot, initialize table of trap handlers with pointers into OS code for handling each type of syscall
    4. Branch and link to trap handler code
    5. CPU restores registers from the kernel stack
    6. CPU switches to user mode
    7. Resume execution after `svc`
- *What should we do if a process tries to perform a privileged operation without making a system call?*
    - Let the code keep running — code may assume privileged operation was successful
    - Kill the process

Creating processes

- `int fork()`
  - Creates an exact copy of the running process, except for the return value from `fork` — return `0` to child (i.e., new) process; return child's process ID to parent process (i.e., process that called fork)
  - Both child and parent resume execution from place where `fork` was called
- Q2: *What does the following code output?*

```
int main(int argc, char **argv) {
    printf("Before fork\n");
    int pid = fork();
    printf("After fork\n");
    return 0;
}
```

```
Before fork
After fork
After fork
```

- Q3: *What does the following code output (assuming the new process has PID 1819)?*

```
int main(int argc, char **argv) {
    printf("Before fork");
    int pid = fork();
    if (pid == 0) {
        printf("Child gets %d\n", pid);
    } else {
        printf("Parent gets %d\n", pid);
    }
    return 0;
}
```

```
Before fork
Child gets 0
Parent gets 1819
```

OR

```
Before fork
Parent gets 1819
Child gets 0
```

- Creating an exponentially increasing number of processes (known as a *fork bomb*)

```
int main() {
    while(1) {
        fork();
    }
```

```
}
```