Operating system concepts

Introduction to Processes
Slides Set #4

By Prof K R Chowdhary

JNV University

2023

Process State Transitions

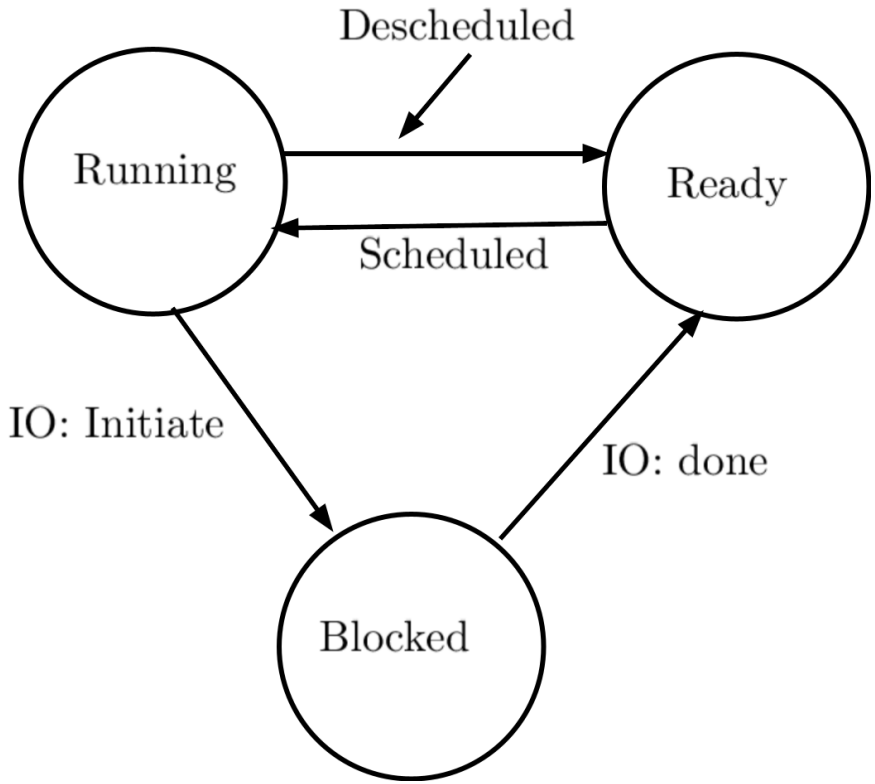


Figure 1: Process: State transition diagram

Processes: The Process Abstraction

OS provides process abstraction

- When you run an .exe file in windows (or a.out in Linux), the OS creates a process, i.e., a running program
- The operating system is using a unique id for every process to keep track of all processes.
- OS timeshares CPU across multiple processes: virtualizes CPU
- OS has a CPU scheduler that picks one of the many active processes to execute on a CPU
  - Policy: which process to run?
  - Mechanism: how to "context switch" between processes?

What constitutes a process?

Each process comprises following:

- Code & data (static)
- A unique identifier (PID) and PPID. (Linux command is: $ ps)
- Memory image
- Stack and heap (dynamic)
- CPU context: registers
  - Program counter
  - Current operands
  - Stack pointer
- File descriptors
  - Pointers to open files and devices

How does OS create a process? 'fork()' command.

- Allocates memory and creates memory image
  - Loads code, data from disk exe
  - Creates runtime stack, heap
- Opens basic files
  - STD IN, OUT, ERR
- Initializes CPU registers
  - PC points to first instruction

States of a process

- *Running*: currently executing on CPU
- *Ready*: waiting to be scheduled
- *Blocked*: suspended, not ready to run
  - Why? Waiting for some event, e.g., process issues a read from disk
  - When is it unblocked? Disk issues an interrupt when data is ready
- New process: being created, yet to run
- Dead process: terminated

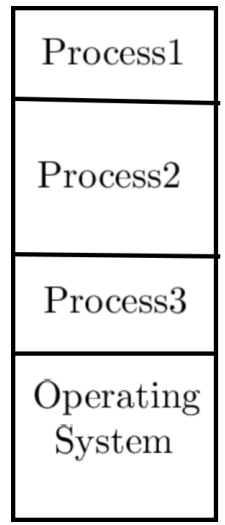


Figure 2: More than one processes in RAM

OS data structures

- OS maintains a data structure (e.g., list) of all active processes
- This information about each process is stored in a process control block (PCB)
  - Process identifier (pid)
  - Process state
  - Pointers to other related processes (i.e., parent process: ppid)
  - CPU context of the process (saved when the process is suspended)
  - Pointers to memory locations
  - Pointers to open files

Example: Process States

Table 1: Tracing process state: CPU and I/O

Time	Process *A*	Process *B*	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process *A* initiates I/O
4	Blocked	Running	Process *A* blocked,
5	Blocked	Running	so Process *B* runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process *B* now done
9	Running	exited	
10	Running	exited	Process *A* now done
11	exited	exited	

Process context switching (PCB=Process Control Block)

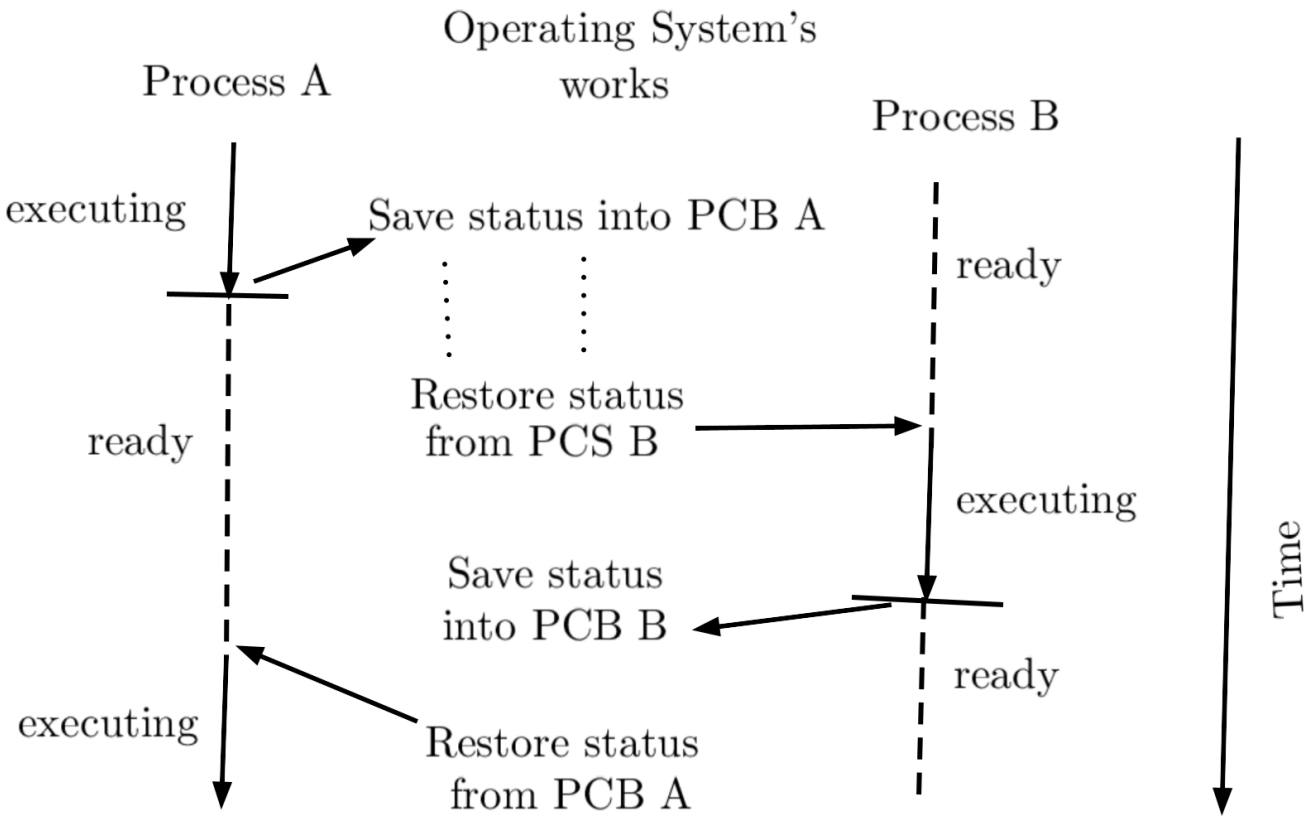


Figure 3: Process context switching

Process Concept

- From a user's point of view, the operating system is there to execute programs:
  - on batch system, refer to *jobs*
  - on interactive system, refer to *processes*
  - (we will use both terms fairly interchangeably)
- Process $\neq$ Program:
  - a program is *static*, while a process is *dynamic*
  - in fact, a process is "a program in execution"
- (Note: "program" here is pretty low level, i.e. native machine code or executable)
- Process includes:
  1. *code section*
  2. *program counter*
  3. *stack*
  4. *data section*
- Processes execute on *virtual processors*

Process Concept

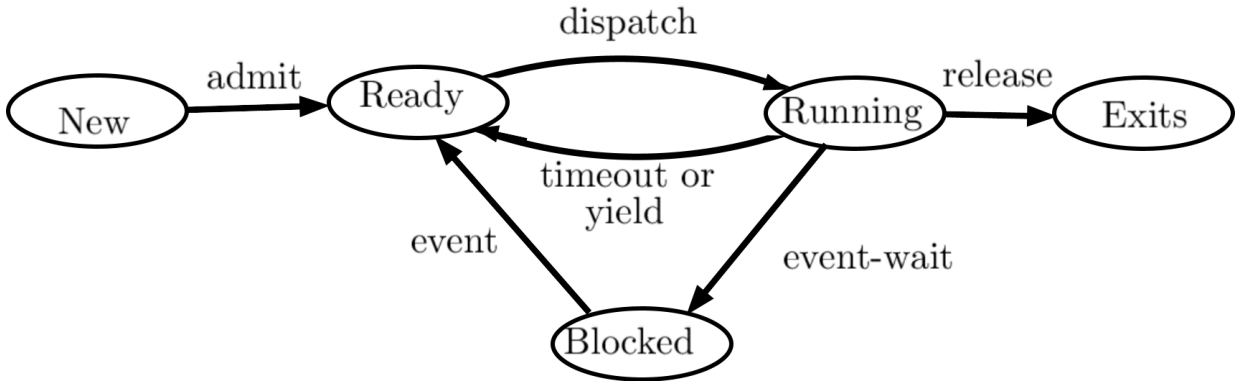


Figure 4: Process states

- As a process executes, it changes state:
  - *New*: the process is being created
  - *Running:* instructions are being executed
  - *Ready*: the process is waiting for the CPU (and is prepared to run at any time)
  - *Blocked*: the process is waiting for some event to occur (and cannot run until it does)
  - Exit: the process has finished execution.
- The operating system is responsible for maintaining the state of each process.

Process Concept

A program of infinite loop, compiling (gcc), running program (in foreground), running in background (by &), process status (ps), kill process (kill command)

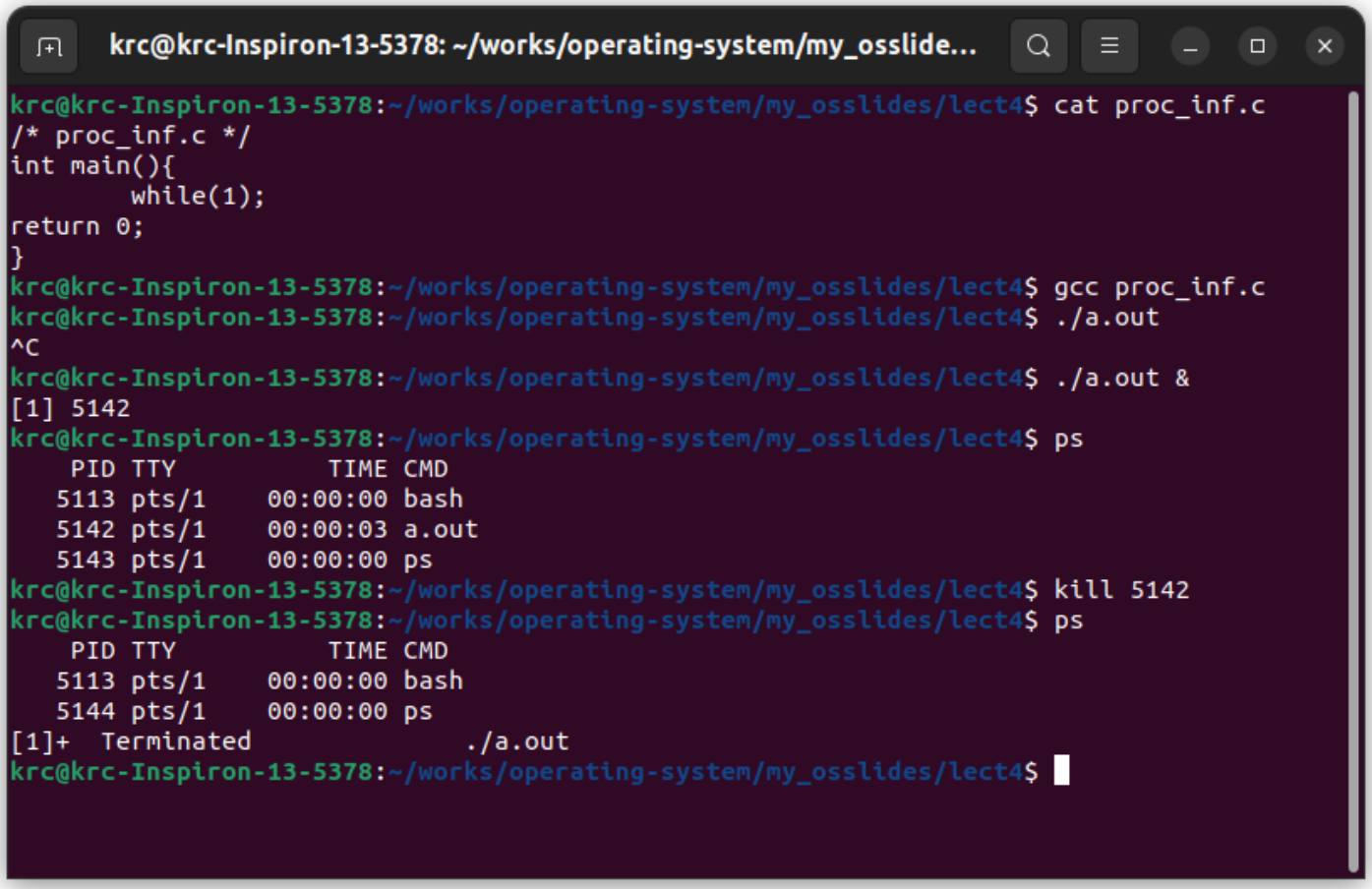


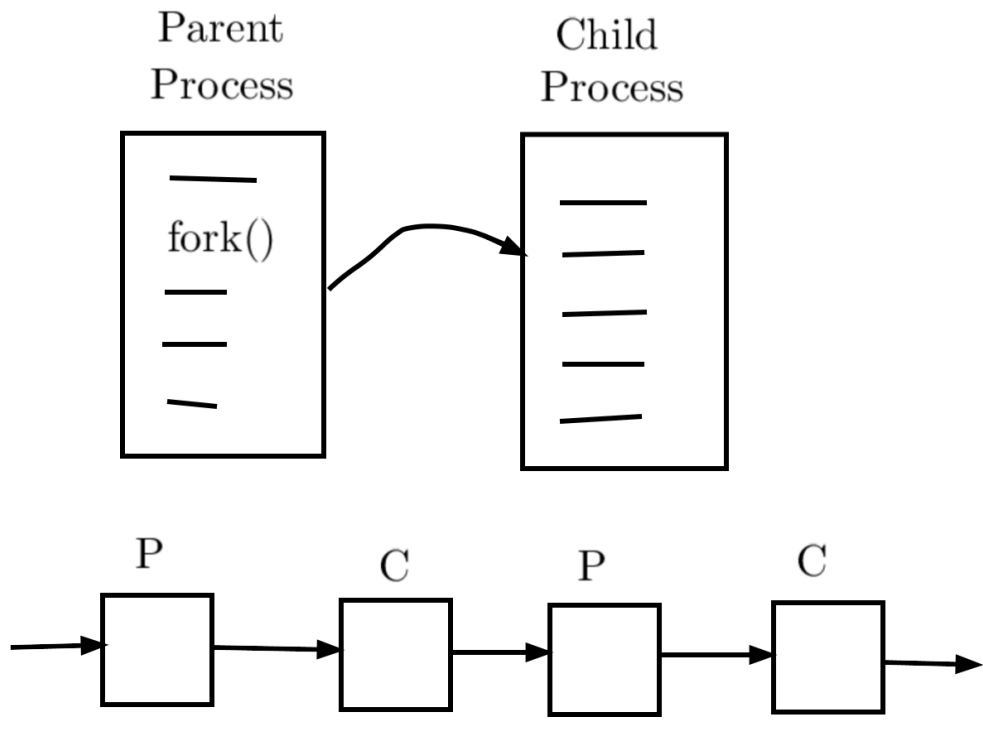
Figure 5: Process creation, run, kill

Process related system calls (in Unix)

- *fork()* creates a new child process
  - All processes are created by forking from a parent
  - The *init* process is ancestor of all processes
- *exec()* makes a process execute a given executable
- *exit()* terminates a process
- *wait()* causes a parent to block until child terminates
- Many variants exist of the above system calls with different arguments

What happens during a fork()?

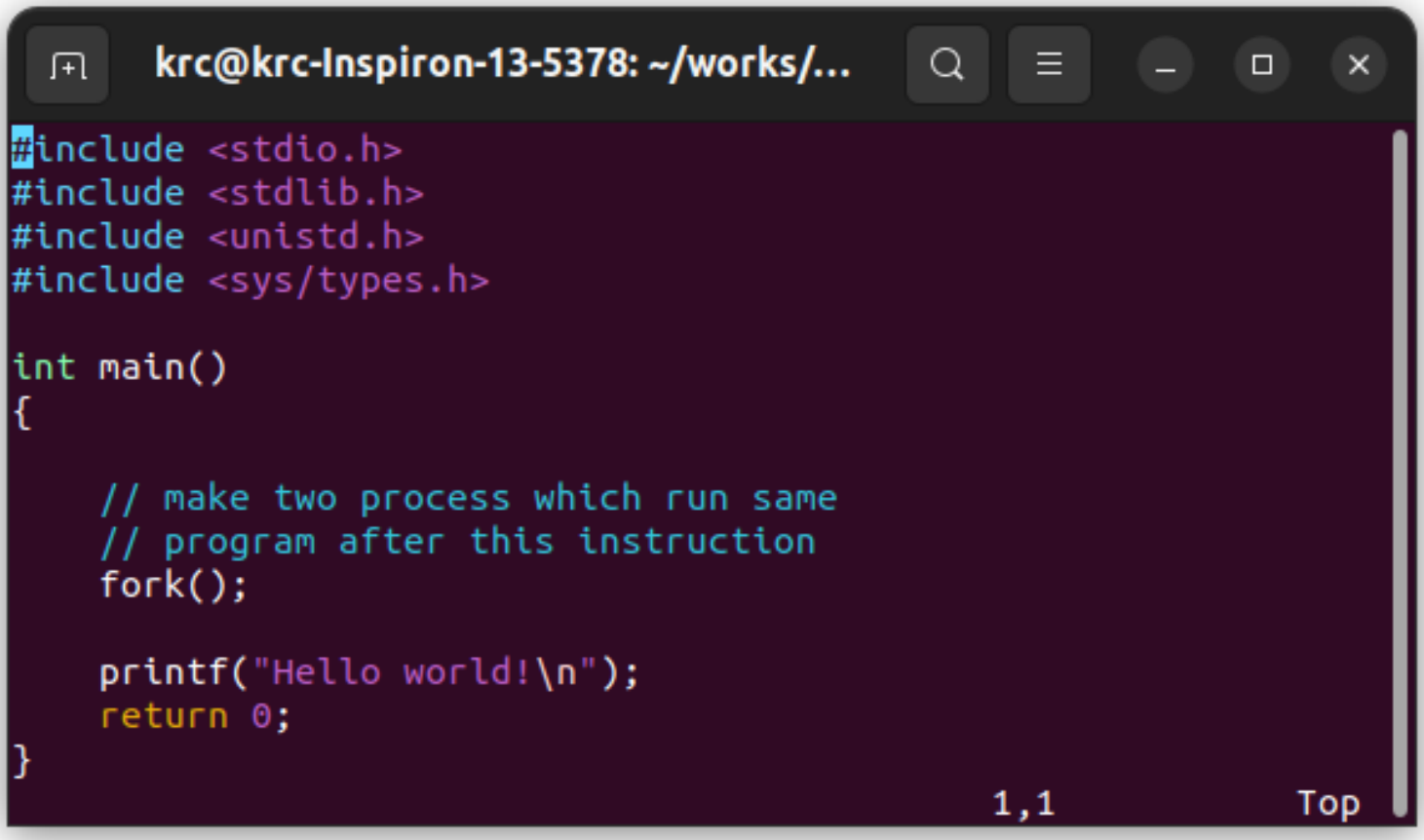
- A new process is created by making a copy of parent's memory image
- The new process is added to the OS process list and scheduled
- Parent and child start execution just after fork (with different return values)



What happens during a fork()?

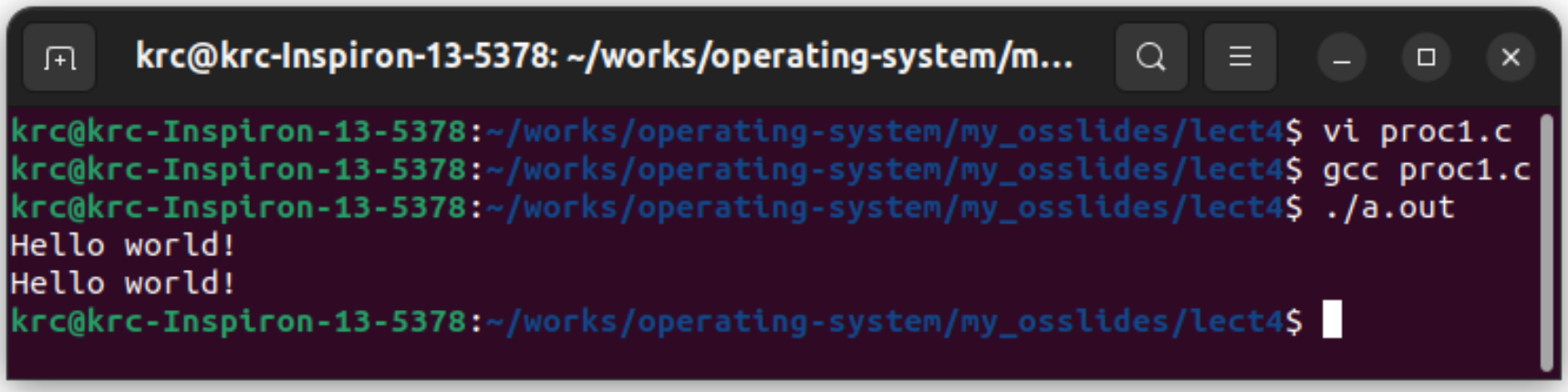
fork() return an int value as follows:

- *Zero*: if it is the child process (the process created).
- *Positive value*: if it is the parent process.
- *Negative value*: if an error occurred.



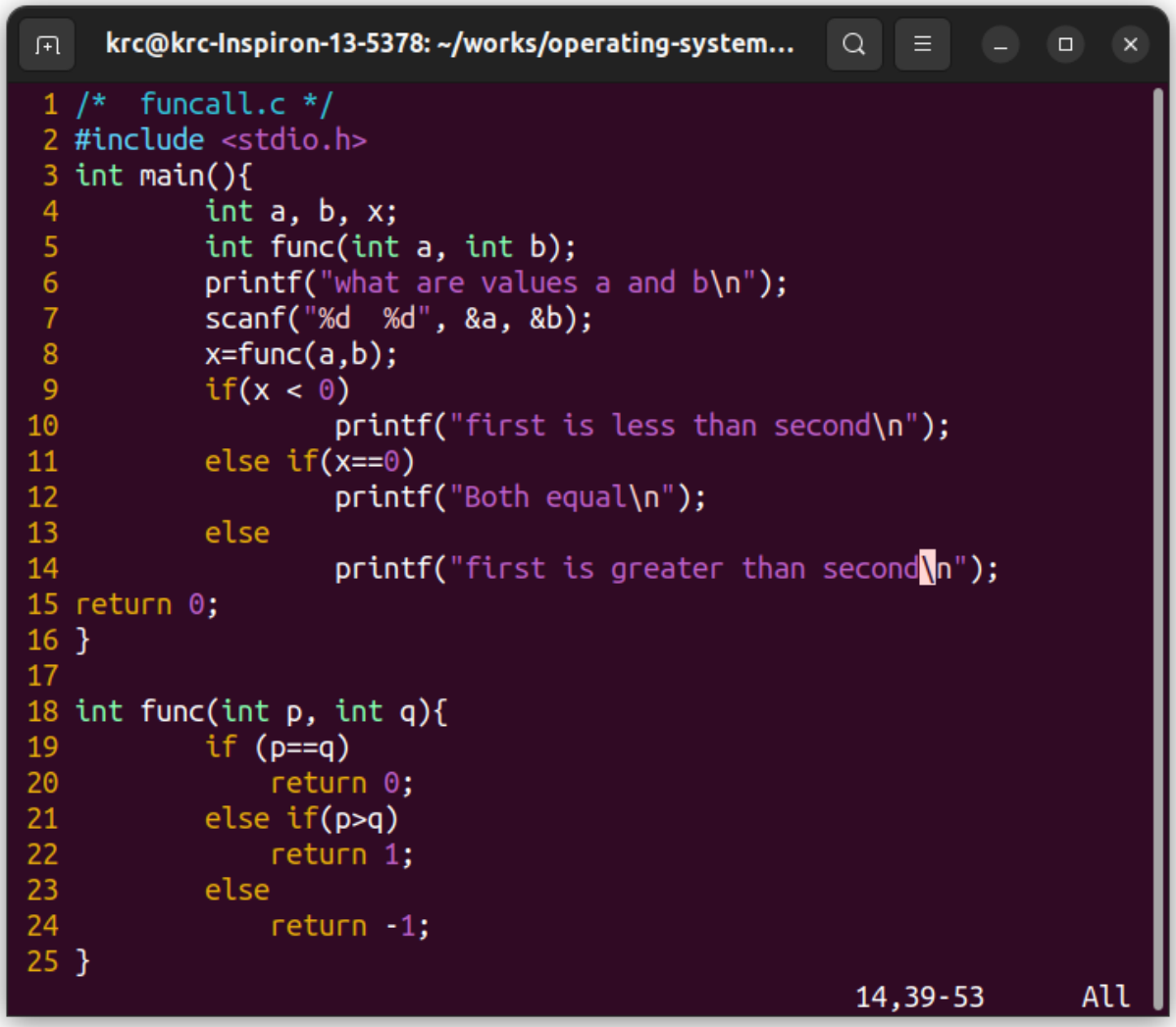
What happens during a fork()?...

- In the example above (proc1.c), the fork() function is used is once.
- The process will be forked in the form of 2^n processes. (n is number of fork() system calls)
- Below are steps for compilation and running of proc1.c



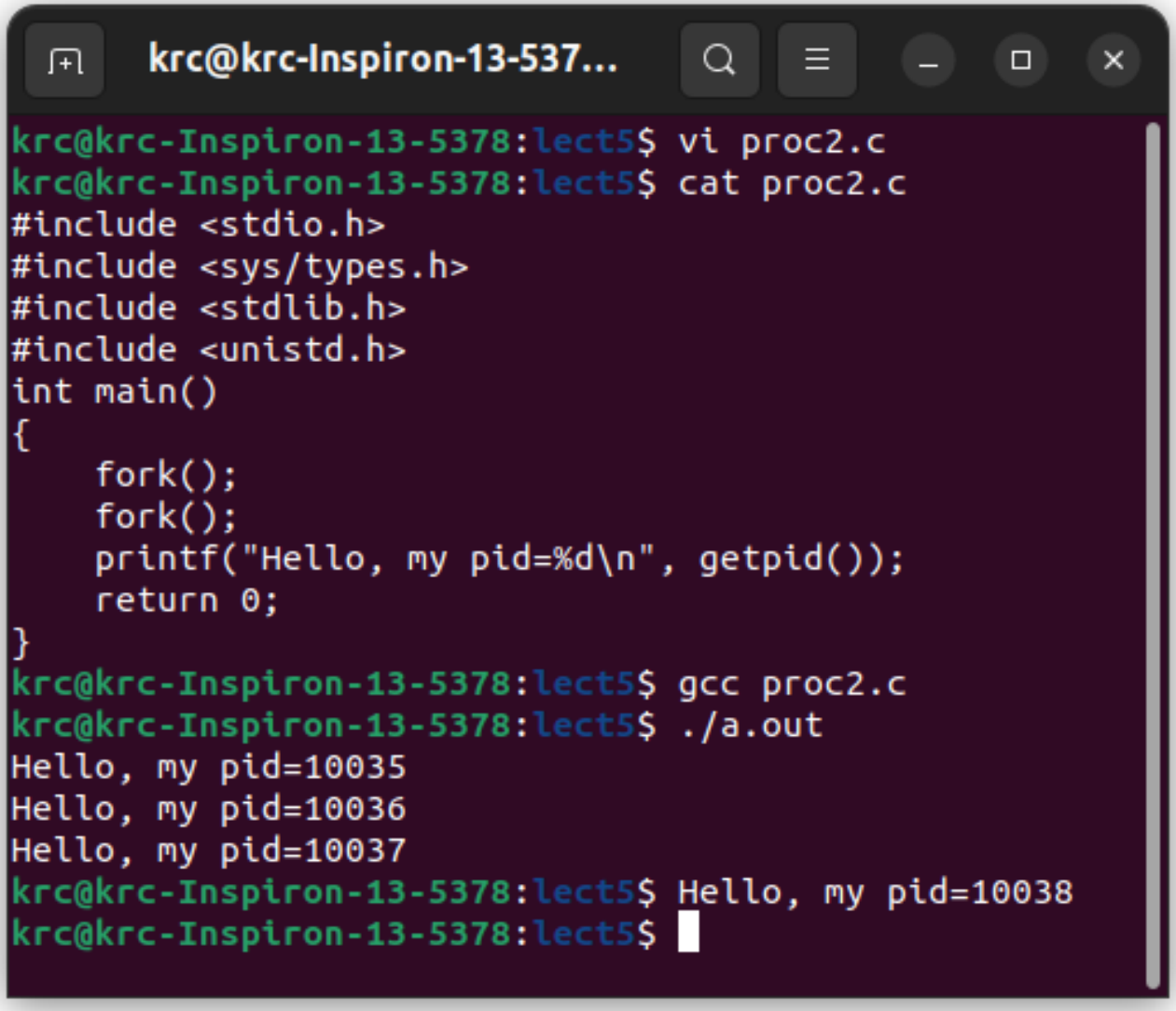
How a function call in C works?

A function call saves (pushes) the contexts (registers, PC in stack), loads PC by address of function program. Before return from function, pops stack and reloads the registers and PC



What happens during a fork()?...

fork() is like a function call, but very different!!



What happens during a fork()?...

Actually, instead of putting the fork() commands in sequence, it is called with conditions.

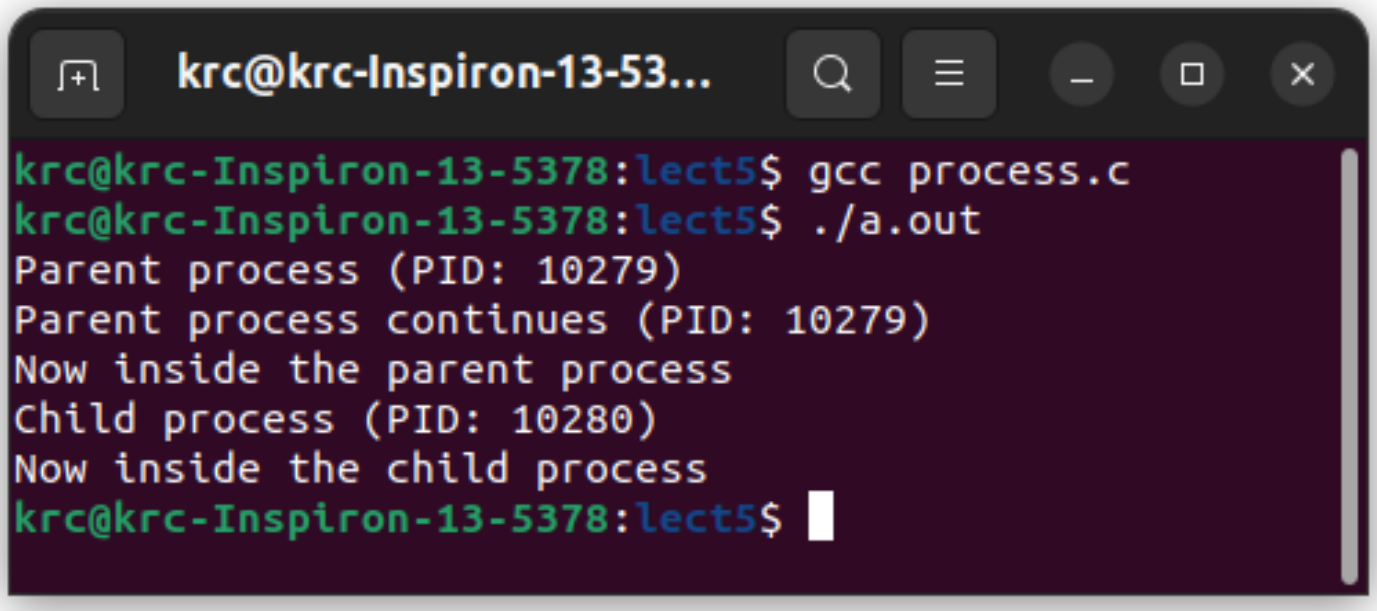
krc@krc-Inspiron-13-5378: ~/works/operating-system/my_osslides/lect5

```
/*Creating multiple processes in C can be achieved using the `fork()` system call. The `fork()` call creates
 a new process, which is a copy of the existing process. Both the parent and child processes continue execut
ing from the point of the `fork()` call, but they have different process IDs (PIDs).

Here's a simple C program that demonstrates multiple processes:

*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t child_pid;

    printf("Parent process (PID: %d)\n", getpid());

    child_pid = fork();

    if (child_pid < 0) {
        // Fork failed
        fprintf(stderr, "Fork failed.\n");
        exit(1);
    } else if (child_pid == 0) {
        // Child process
        printf("Child process (PID: %d)\n", getpid());
        printf("Now inside the child process\n");
    } else {
        // Parent process
        printf("Parent process continues (PID: %d)\n", getpid());
        printf("Now inside the parent process\n");
    }

    return 0;
}
```

5,1 Top

What happens during a fork()?...

- The "printf{Parent..}" is printed by parent process before fork() is executed. The $x = fork()$ execution returns a value 0 to x. So, "child_pid < 0" is false.
- if fork() fails, it returns -1.
- The last lines in program, which are due to parent, may execute before even the child is executed,



Create Process using fork()

Terminal

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/wait.h>
4 #include <unistd.h>
5 int main()
6 {
7 pid_t pid;
8 /* fork a child process */
9 pid = fork();
10 if (pid < 0) { /* error occurred */
11    fprintf(stderr, "Fork Failed");
12    return 1;
13 }
14 else if (pid == 0) { /* child process */
15    execl("/bin/ls","ls",NULL);
16      }
17    else { /* parent process */
18      /* parent will wait for the child to complete */
19            wait(NULL);
20            printf("Child Complete. Now in parent\n");
21     }
22 return 0;
23 }
:set nu                                          19,22         All
```