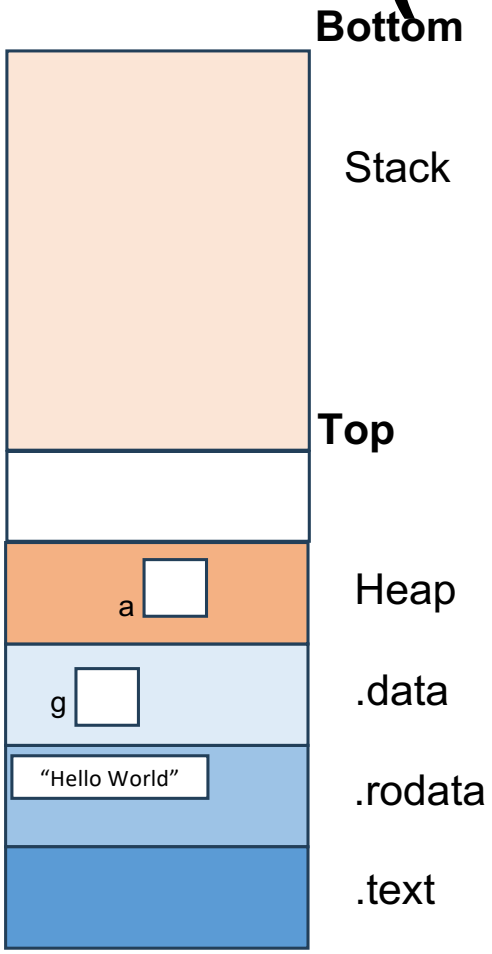
Lecture 05: The Process Abstraction

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

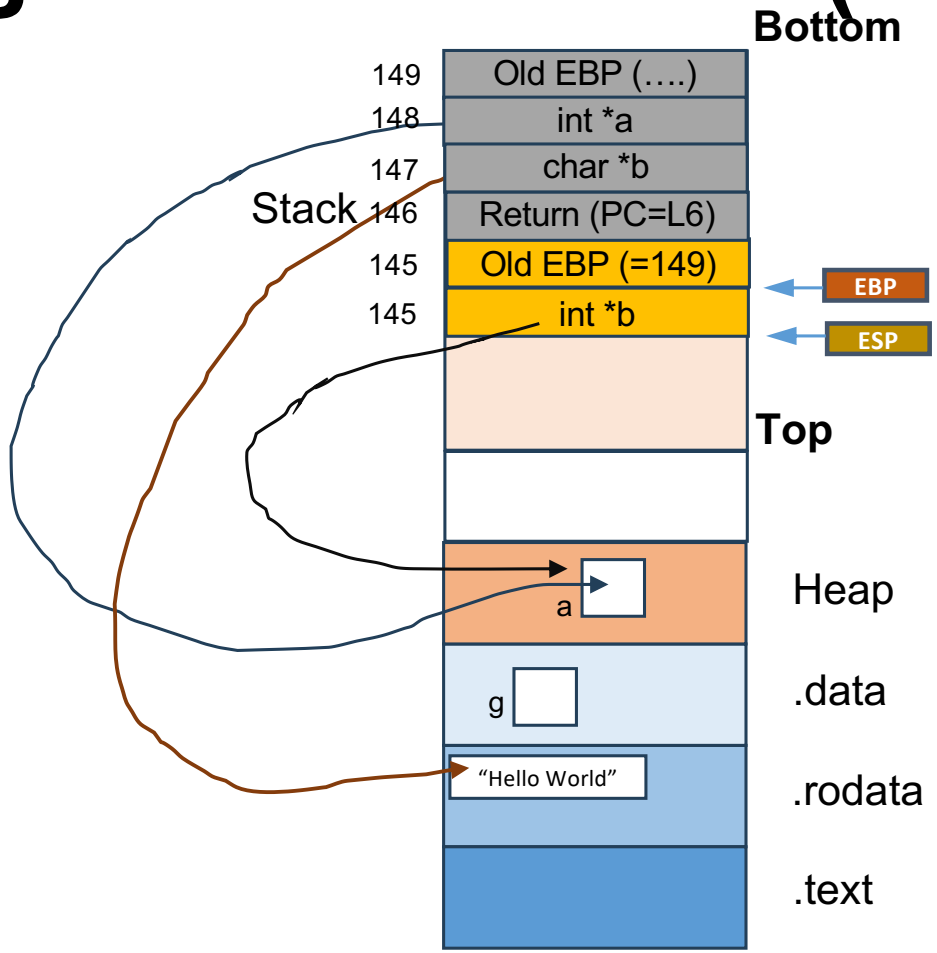
Last Lecture: Program Execution (1/2)

```
L1: int g=0;

L2: void main() {
L3:     int *a = (int*) malloc(4);
L4:     char *b = “Hello World”;
L5:     foo(a);
L6:     g=*a;
L7: }

L8: void foo(int* b) {
L9:     *b = 20;
L10:}
```



Last Lecture: Program Execution (2/2)

```
L1: int g=0;

L2: void main() {
L3:     int *a = (int*) malloc(4);
L4:     char *b = “Hello World”;
L5:     foo(a);
L6:     g=*a;
L7: }

L8: void foo(int* b) {
L9:     *b = 20;
L10:}
```



Today's Class

- The process abstraction
- Quiz-1 (Lectures 02-04)

All Executables are ELF

```
iiitd@possum:~$ file /bin/cat
/bin/cat: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamicall
y linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, Buil
dID[sha1]=747e524bc20d33ce25ed4aea108e3025e5c3b78f, stripped
iiitd@possum:~$ file /bin/ls
/bin/ls: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
 linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, Build
ID[sha1]=9567f9a28e66f4d7ec4baf31cfbf68d0410f0ae6, stripped
iiitd@possum:~$ file /bin/mkdir
/bin/mkdir: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamica
lly linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, Bu
ildID[sha1]=6c825e92c5eae304845d070ed34749495d67c566, stripped
```

The Process

```
iiitd@possum:~$ vi fib.c
iiitd@possum:~$ gcc fib.c
iiitd@possum:~$ ./a.out
Fib(40) = 102334155
```

```
  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 2034 iiitd      20   0  4384   820   756 R 100.  0.0  0:17.48 ./a.out
```

- Whenever we type and enter a command on shell, the command prompt is returned back only after that command has completed its execution
  - How this execution happens?
- A program under execution is called a process
  - It is again all about abstraction. First we discussed about ELF to have an abstraction and presenting a unified view of all object files for the loader. Now, we are going one level above where the OS needs to have an abstract representation of any kind of program under execution
  - Process is a collection of resources

There are Several Processes

```
iiitd@possum:~$ for i in {1..20}
> do
> ./a.out &
> done
[1] 2062
[2] 2063
[3] 2064
[4] 2065
[5] 2066
[6] 2067
[7] 2068
[8] 2069
[9] 2070
[10] 2071
[11] 2072
[12] 2073
[13] 2074
[14] 2075
[15] 2076
[16] 2077
[17] 2078
[18] 2079
[19] 2080
[20] 2081
```

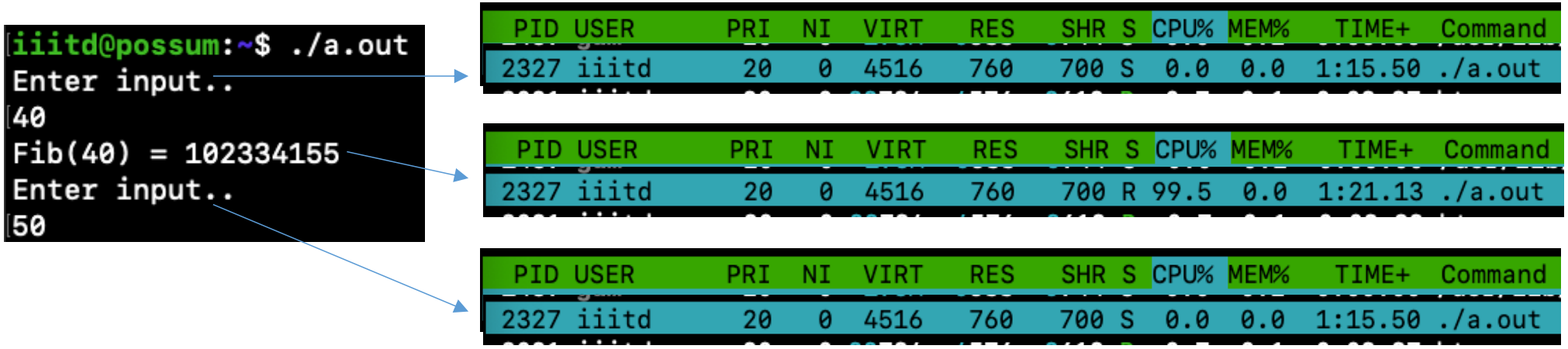
```
  1  |||||||||||||||||||||||||||||||100.0%    Tasks: 104, 202 thr; 4
  2  |||||||||||||||||||||||||||||||100.0%    Load average:      2.36
  3  |||||||||||||||||||||||||||||||100.0%    Uptime: 00:09:59
  4  |||||||||||||||||||||||||||||||100.0%
Mem  ||||||||||||                    470M/3.75G
Swp                                    0K/2.00G

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 2074 iiitd      20   0  4384   804   740 R 20.5  0.0  0:06.97 ./a.out
 2076 iiitd      20   0  4384   768   704 R 20.5  0.0  0:06.96 ./a.out
 2080 iiitd      20   0  4384   792   728 R 20.5  0.0  0:06.96 ./a.out
 2068 iiitd      20   0  4384   712   648 R 20.5  0.0  0:06.96 ./a.out
 2066 iiitd      20   0  4384   820   756 R 19.9  0.0  0:06.97 ./a.out
 2064 iiitd      20   0  4384   708   648 R 19.9  0.0  0:06.97 ./a.out
 2072 iiitd      20   0  4384   756   692 R 19.9  0.0  0:06.96 ./a.out
 2063 iiitd      20   0  4384   800   740 R 19.9  0.0  0:06.96 ./a.out
 2069 iiitd      20   0  4384   720   656 R 19.9  0.0  0:06.96 ./a.out
 2078 iiitd      20   0  4384   864   800 R 19.9  0.0  0:06.98 ./a.out
 2070 iiitd      20   0  4384   804   740 R 19.9  0.0  0:06.98 ./a.out
 2075 iiitd      20   0  4384   752   692 R 19.9  0.0  0:06.96 ./a.out
 2073 iiitd      20   0  4384   764   700 R 19.9  0.0  0:06.98 ./a.out
 2065 iiitd      20   0  4384   804   740 R 19.9  0.0  0:06.98 ./a.out
 2077 iiitd      20   0  4384   712   648 R 19.9  0.0  0:06.93 ./a.out
 2071 iiitd      20   0  4384   752   692 R 19.9  0.0  0:06.94 ./a.out
 2081 iiitd      20   0  4384   756   692 R 19.9  0.0  0:06.93 ./a.out
 2079 iiitd      20   0  4384   720   656 R 19.9  0.0  0:06.94 ./a.out
 2067 iiitd      20   0  4384   716   656 R 19.9  0.0  0:06.93 ./a.out
 2062 iiitd      20   0  4384   800   740 R 19.9  0.0  0:06.94 ./a.out
 2021 iiitd      20   0 33784  4536  3612 R  0.0  0.1  0:00.68 htop
```

```
iiitd@possum:~$ cat /proc/sys/kernel/pid_max
32768
```

```
iiitd@possum:~$ lscpu
Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
CPU(s):                  4
On-line CPU(s) list:     0-3
Thread(s) per core:      1
Core(s) per socket:      4
```

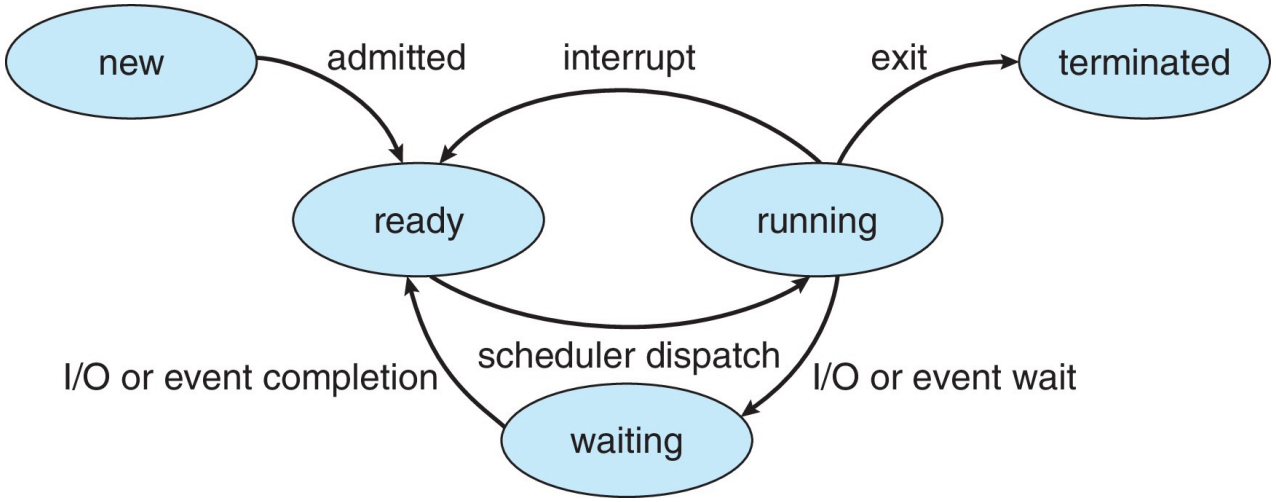
- Note the “R”(running) state of the program in “S”(state) column in htop output
- There are only four CPUs in this system, but how is the OS able to run 20 a.out simultaneously?
- Virtualization of the CPU!
  - Any limit?

Process Keep Changing its State



- There are two distinct phases in the above program
  - Asking the user for the input
    - The process representing the a.out is moved into “S” state
      - It got kicked out of the CPU
  - Calculating the Fibonacci number of that input
    - The process representing the a.out is moved into “R” state
      - It got its CPU back!
  - Why not simply leave the process occupying the CPU?

Process States



- As a process executes, it changes state
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process has finished execution

Process State: Example

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

Figure 4.4: **Tracing Process State: CPU and I/O**

What Constitutes a Process

- There are so many processes at any given time
  - Each process has its state
  - A running process will be using some CPU registers
    - We push and pop the registers on call stack during the course of execution
  - Process requires memory for execution (stack and heap)
  - Process have several open file handles
- For process management, the OS uses a data-structure called as “process descriptor” or “process control block” to keep track of every process’s progress and usage of the computer’s available resources

Process Descriptor (a.k.a PCB)

- Kernel maintains detail on each process inside a structure of type **task_struct**
- You can see it yourself
  - https://raw.githubusercontent.com/torvalds/linux/master/include/linux/sched.h

```
struct task_struct {
    unsigned int                    __state;
     void                           *stack;
     int                            prio;
     int                            nr_cpus_allowed;
     struct mm_struct               *mm;
     struct mm_struct               *active_mm;
     pid_t                          pid;
     struct list_head               children;
     struct list_head               sibling;
     struct task_struct             *group_leader;
     /*
      * executable name, excluding path.
      *
      * - normally initialized setup_new_exec()
      * - access it with [gs]et_task_comm()
      * - lock it with task_lock()
      */
     char                           comm[TASK_COMM_LEN];

     struct nameidata               *nameidata;
     /* Filesystem information: */
     struct fs_struct               *fs;

     /* Open file information: */
     struct files_struct            *files;
```

- Extracts from the struct task_struct
  - There are many more members!
- Registers are saved on stack
- Linked list of PCB to dynamically add a new process information

```
iiitd@possum:~$ ps -u iiitd --forest
  PID TTY          TIME CMD
23005 ?        00:00:00 sshd
23006 pts/1    00:00:00  \_ bash
23062 pts/1    00:00:01      \_ a.out
22906 ?        00:00:00 sshd
22908 pts/0    00:00:00  \_ bash
23063 pts/0    00:00:00      \_ ps
22807 ?        00:00:00 systemd
22808 ?        00:00:00  \_ (sd-pam)
```

Next Lecture

- System calls