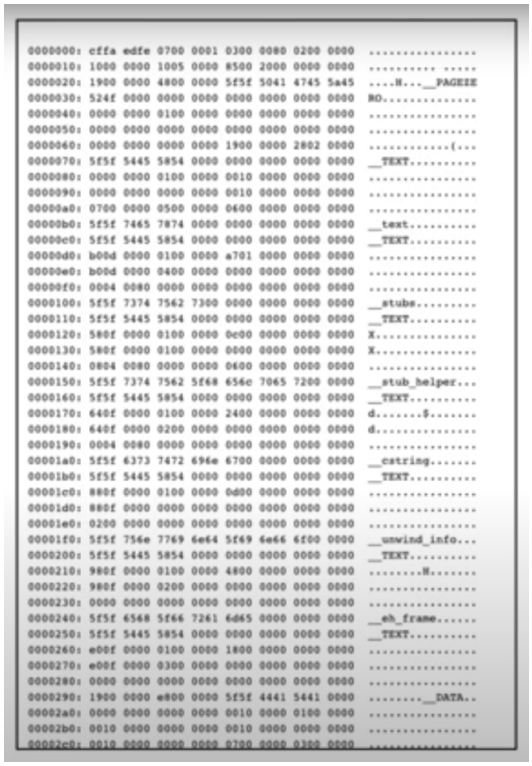
CSC209 Week 8 Notes

Hyungmo Gu

May 17, 2020

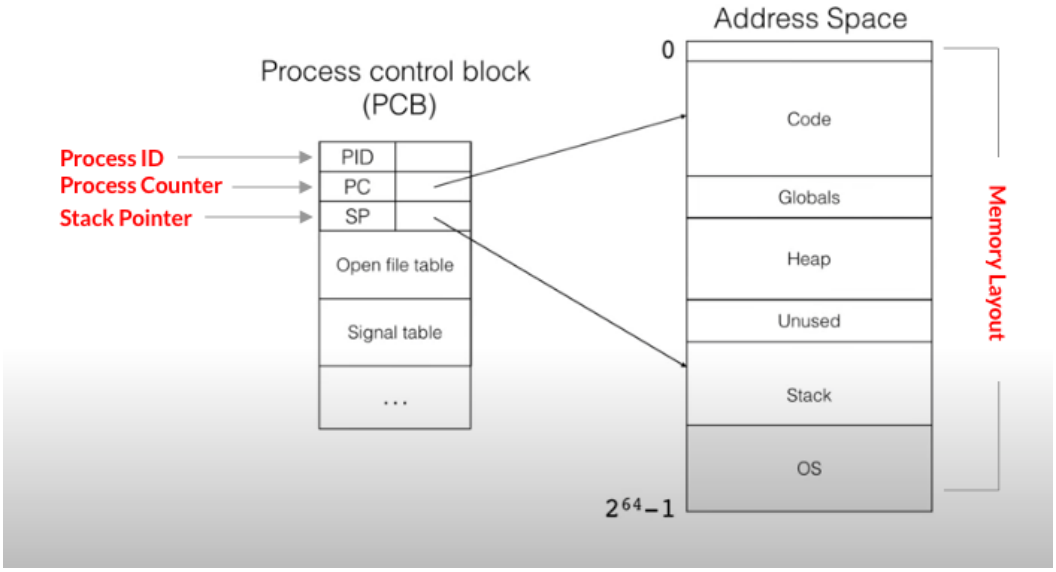
Processes 1 of 8

- Process Models
  - Program
    - The executable instructions of a program
    - Source code
    - Compiled machine code



  - Process
    - Running instance of program
  - Process Control Block (PCB)

* is a data structure used by computer operating system to store all information about process.
* Code, Global, Heap and Stack together are called the **State of Program**



  - **Code:** is program code
  - **Stack:** tells which function is being executed, and hold values of local variables
  - **Heap and Globals:** holds current value for other variables in the program
* is also known as **process descriptor**
  1. When a process is created (initialized or installed), the operating system creates a corresponding process control block.
  2. Information in a process lbock is updated during the transition of process states
  3. When the process terminates, its PCB is returned to the pool from which new PCBs are drawn
  4. Each process has a single PCB

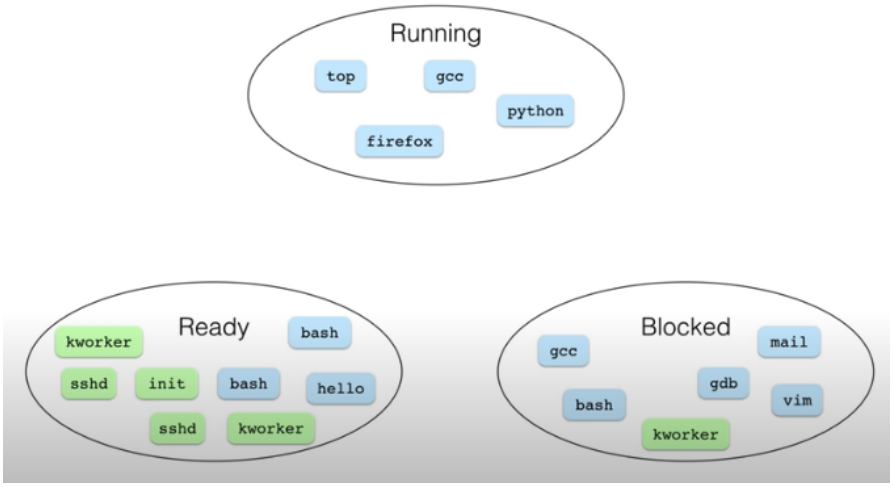
– Top

* is a basic unix command useful for observing current state of Unix System.

```
top - 14:23:18 up 11 days,  4:50, 13 users,  load average: 0.00, 0.05, 0.09
Tasks: 666 total,   1 running, 662 sleeping,   3 stopped,   0 zombie
Cpu(s):  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  65954776k total, 46558744k used, 19396032k free, 33357016k buffers
Swap: 102399996k total,        0k used, 102399996k free,  9139880k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
    8 root      20   0     0    0    0 S    0  0.0  12:03.39 rcu_sched
 2196 root      39  19     0    0    0 S    0  0.0  47:40.80 kipmi0
13365 reid      20   0 17732 1768  964 R    0  0.0   0:00.54 top
23412 root      20   0     0    0    0 S    0  0.0   0:21.41 kworker/2:0
51571 c4user    24   4 30668 5960 1228 S    0  0.0   8:47.88 tmux
    1 root      20   0 24452 2372 1336 S    0  0.0   0:09.76 init
```

- Three-state Process Management Model



  * Works with PCB
  * Works simultaneously on multiple processors
  * A program is in either one of three states:
    - **Running:** The process that is currently being executed
    - **Ready:** A process that is queuing and prepare to execute when given the opportunity
    - **Blocked:** A process that cannot execute until some event occurs, i.e. waking up from sleep

Processes 2 of 8

- Creating Processes with Fork
  - fork()
    * is a part of *unistd.h* library.
    * creates a new process, which is called child process, which runs
    * starts executing after fork() is called

```c
#include <stdio.h>
#include <unistd.h> // <- fork imported here

int main() {
    int i;
    pid_t result;

    i = 5;
    printf("%d\n", i);
```

```
10
11          result = fork(); // <- Here is the fork :)
12
13          if (result > 0) {
14              i = i + 2;
15          } else if (result == 0) {
16              i = i - 2;
17          } else {
18              perror("fork");
19          }
20
21          printf("%d\n", i);
22          return 0;
23      }
24
```

Listing 1: process_example_1.c

Processes 3 of 8

- Process Relationship and Termination (Part 1)
  - Parent and child processes don't occur in order
  - Child processes not in order since they are running concurrently
  - Key Queston: How to make the parent to wait before child finishes?

```
1     // Output of process_example_2.c
2     // Run by typing gcc -Wall process_example_2.c
3
4     [20480] Original process (my parent is 15738)
5     [20481] Child 0 0
6     [20481] Child 0 1
7     [20482] Child 1 0 //<- notice 1 starts before 0 ends
8     [20481] Child 0 2
9     [20482] Child 1 1
10    [20481] Child 0 3
11    [20483] Child 2 0
12    [20482] Child 1 2
13    [20481] Child 0 4
14    [20480] Parent about to terminate // <- notice parent ends
    before children
15    [20482] Child 1 3
16    [20483] Child 2 1
17    [20484] Child 3 0 //<- notice 3 starts before 1 and 2 ends
18    [20482] Child 1 4
19    [20483] Child 2 2
20    [20484] Child 3 1
21    [20485] Child 4 0
22    [20483] Child 2 3
```

```
    [20484] Child 3 2
    [20483] Child 2 4
    [20485] Child 4 1
    [20484] Child 3 3
    [20485] Child 4 2
    [20484] Child 3 4
    [20485] Child 4 3
    [20485] Child 4 4

```

Processes 4 of 8

- wait
  - **Syntax:** pid_t wait(int *wstatus)
  - Is a part of *sys/wait.h* library.
  - Blocks the calling process until one of its child processes exists or a signal is received.
  - Parent continues its execution after wait call
- WIFEXITED
  - **Syntax:** int WIFEXITED (int status)
  - Checks if child exited normally
  - Returns non-zero if true
- WIFEXITSTATUS(status)
  - **Syntax:** int WIFEXITSTATUS (int status)
  - Returns code when child exits
- WTERMSIG
  - **Syntax:** int WTERMSIG (int status)
  - Returns non-zero value if the child process terminated due to a signal, i.e. abort()

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

int main() {
    ...

    for(i = 0; i < 5; i++) {
```

```
10              pid_t pid;
11              int status;
12              if ((pid = wait(&status)) == -1) { // <- wait called here
13                  perror("wait");
14              } else {
15                  if (WIFEXITED(status)) { // <- processes ended
    normally
16                      printf("Child %d terminated with %d\n", pid,
    WEXITSTATUS(status));
17                  } else if (WIFSIGNALED(status)) { // <- child exited
    due to signal
18                      printf("Child %d terminated with signal %d\n", pid
    , WTERMSIG(status));
19                  } else {
20                      printf("Shouldn't get here");
21                  }
22              }
23          }
24          printf("[%d] Parent about to terminate\n", getpid());
25          return 0;
26      }
27
```

Listing 2: process_example_3.c

Processes 6 of 8

- Running Different Programs
  - execl
    - **Syntax:** int execl(const char *filepath, [const char *arg1, const char *arg2], NULL)
    - is a part of *unistd.h* library
    - Requires *NULL* as final arguement
    - Code doesn't return to main if excel is successful
    - Code returns to main if excel is unsuccessful

```
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main() {
5       printf("About to call excel. My PID is %d\n", getpid());
6       execl("./hello.out", NULL); // <- searches for .out
7       perror("exec");
8       return 1;
9   }
10
```

Listing 3: process_example_4.c