PROCESS IN PRACTICE

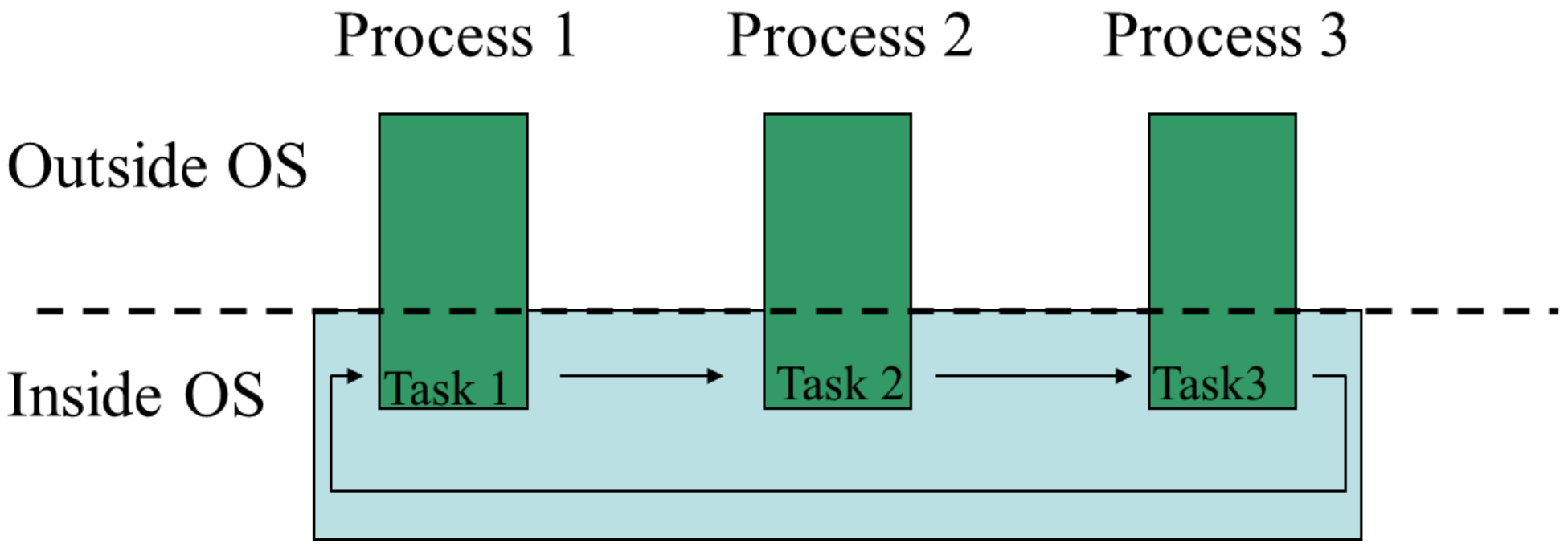
I GDE DHARMA NUGRAHA

OVERVIEW OF PROCESS

- Processes carry out tasks within the operating system.
- A program is a set of machine code instructions and data stored in an executable image on disk and is, as such, a passive entity; a process can be thought of as a computer program in action.
- To overview how the process works, we will explore the process management in Linux.

LINUX PROCESSES VS TASKS

- **Linux views all work as a series of tasks imposed on the OS. Processes and threads are essentially viewed in the same way, except that resources allocated are different**



LINUX PROCESS

- So that Linux can manage the processes in the system, each process is represented by a ***task_struct*** data structure (task and process are terms that Linux uses interchangeably).
  - The task vector is an array of pointers to every ***task_struct*** data structure in the system.
- The maximum number of processes in the system is limited by the size of the task vector; by default it has 512 entries.
  - As processes are created, a new task_struct is allocated from system memory and added into the task vector.
- Detail information of *task_struct* is showed in
  - /usr/src/linux-headers-[version]/include/linux/sched.h

LINUX PROCESS CONTROL BLOCKS (PCBS)

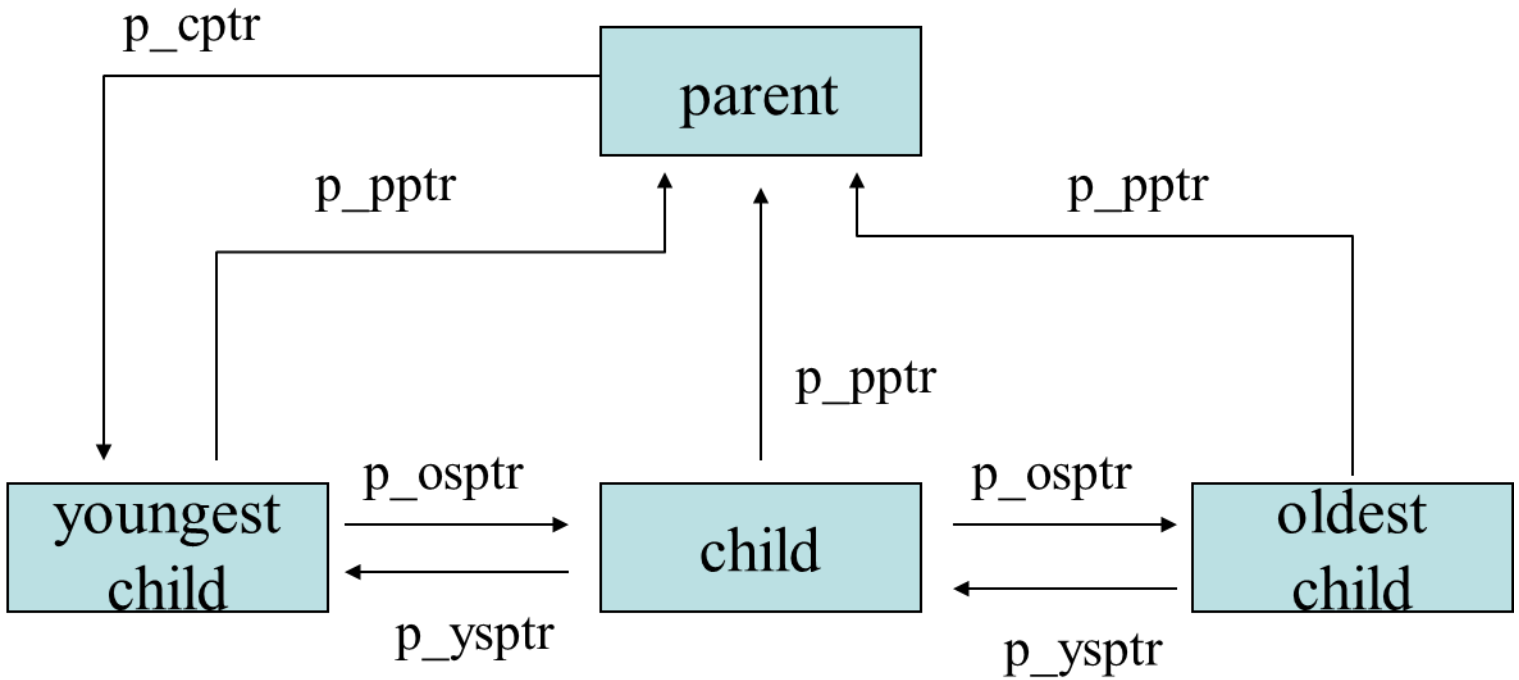
- PCBs are the blocks where the OS can find all the information it needs to know about a process.
  - Memory
  - Open streams/file
  - Devices
  - Link
  - Processor registers
  - Process identification
  - Process state
  - Priority
  - Owner
  - Which processor (SMP)
  - Link to others (Parent, Children)
  - Process group
  - Resource Limit
  - Access right
  - Process Result
- Doesn't have to be kept together
  - Different information is required at different times
- UNIX for example has two separate places in memory with this information. One of them is in the kernel the other is in user space

LINUX PROCESS CONTROL BLOCKS (PCBS)

- The implementation:

```
/* memory management info */
     struct mm_struct *mm;
/* open file information */
     struct files_struct *files;
/* tss for this task */
     struct thread_struct tss;
  int pid;
  volatile long state; /* -1 unrunnable, 0
   runnable, >0 stopped */
  long priority;
  unsigned short uid,euid,suid,fsuid;
#ifdef __SMP__
     int processor;
   #endif
struct task_struct *p_opptr, *p_pptr,
   *p_cptr, *p_ysptr, *p_osptr;
/* limits */
     struct rlimit rlim[RLIM_NLIMITS];
     long utime, stime, cutime, cstime,
   start_time;
```

PROCESS RELATIONSHIP



pid_t getpid(void);
pid_t getppid(void);

THE PROCESS LIST

- Tasks managed through a doubly linked list. Each task_struct includes:
  - struct task_struct *next_task;
  - struct task_struct *prev_task;
  - List terminated at global variable init_task

PROCESS CREATION

- Process Creation
  - fork( )
  - execve( )
  - clone( )

CREATING PROCESS – FORK()

- **pid_t fork(void);**

- Creates a child process that differs from the process only in its PID and PPID, and in the fact that resource utilizations are set to 0. Linux uses a copy-on-write procedure.
  - Child process returns PID = 0;
  - Parent process returns child PID

CREATING PROCESS – FORK()

- General Syntax

```
pid = fork( );                          /* if the fork succeeds, pid > 0 in the parent */
if (pid < 0) {
        handle_error( );                /* fork failed (e.g., memory or some table is full) */
} else if (pid > 0) {
                                        /* parent code goes here. /*/
} else {
                                        /* child code goes here. /*/
}
```

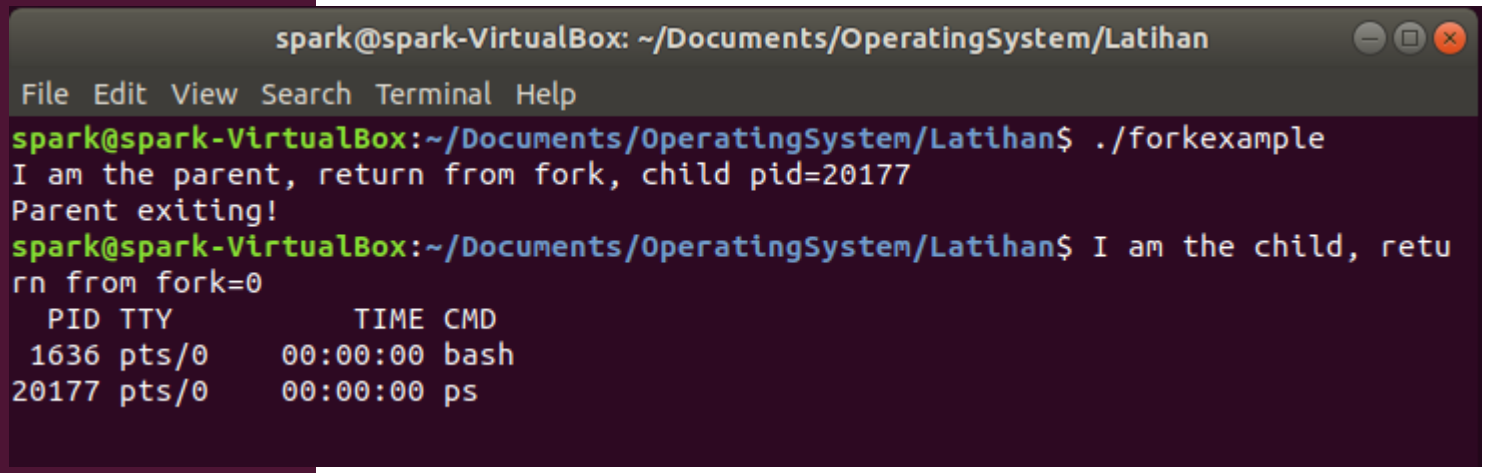
CREATING PROCESS - FORK()

- Example
  - forkexample.cpp

```
/* Example of use of fork system call */
#include <stdio.h>
#include <iostream>
#include <string>
// Required by for routine
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

main()
{
  int pid;

  pid = fork();

  if (pid < 0) { // error occurred
          fprintf(stderr, "Fork failed!\n");
          exit(-1);
  }else if (pid == 0) { // child process
          printf("I am the child, return from fork=%d\n", pid);
          execlp("/bin/ps", "ps", NULL);
  }else { // parent process
          printf("I am the parent, return from fork, child pid=%d\n", pid);
          printf("Parent exiting!\n");
          exit(0);
  }
}
```

CREATING PROCESS - FORK()

- Example



CREATING NEW PROCESS – EXEC.CPP

```
#include <iostream>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <errno.h>

extern char **environ;
extern int errno;
pid_t pid;
using namespace std;

int main (int argc, char* argv[]){
        FILE *fin;
        char lineBuf[128];
        char *progname = argv[1];
        char myname[15] = "Bob";
        char mynum[5] = "5";
        static int result;
        char *args[4];
        if (argc == 2)
                strcpy (progname, argv[1]);
        else{
                cout <<"Usage:"<<argv[0] <<" progname" << endl;
                exit(1);
        }
        pid = fork();
        if (pid == 0){
                args[0] = progname;
                args[1] = (char *) &myname;
                args[2] = (char *) &mynum;
                args[3] = (char *) 0;
                if (execv (progname, args) == -1){
                        cout << "ERROR IN EXECVE" << endl;
                        perror("execve");
                }
        }
        else
        {
                cout << "Created new process with process ID " <<pid<<endl;
                sleep(1);
        }
        return 0;
}
```

CREATING NEW PROCESS – HELLO.CPP

```
#include <iostream>
using namespace std;

int main(int argc, char * argv[])
{
        int i, mynum;
        if (argc != 3) {
                cout << "Usage: " << argv[0] << " Name Number" << endl;
        return 0;
        }
        cout << "Hello, World!" << endl;
        cout << "Passed in " << argc << " arguments" << endl;
        cout << "They are: " << endl;
        mynum = atoi(argv[2]);
        cout << "Program name is " << argv[0] << endl;
        cout << "Name argument is " << argv[1] << endl;
        cout << "The number is " << mynum << endl;
        return 0;
}
```

CREATING NEW PROCESS – OUTPUT

```
spark@spark-VirtualBox:~/Documents/OperatingSystem/Latihan$ g++ -o exec exec.cpp
spark@spark-VirtualBox:~/Documents/OperatingSystem/Latihan$ g++ -o hello hello.c
pp
spark@spark-VirtualBox:~/Documents/OperatingSystem/Latihan$ ./exec hello
Created new process with process ID 21254
Hello, World!
Passed in 3 arguments
They are:
Program name is hello
Name argument is Bob
The number is 5
```

PROCESS MANAGEMENT SYSTEM CALLS

- Linux system calls for process management:
  - Getting process ID
    - pidof [progname]
    - pgrep (progname]
    - echo (echo $$ & echo $PPID)
  - Interchange background – foreground jobs
    - fg $PID
    - Bg
    - halt process (ctrl+Z)
  - View active process
    - ps
    - Top
  - Stop process
    - kill
    - pkill [progname]
  - Changing process priority
    - renice [newpriority] [PID]
  - Show groups of the process
    - pstree

PROCESS MANAGEMENT SYSTEM CALLS

- KILL has many type of signals:

```
$ kill -l
```

```
[root@tecmint ~]# kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
[root@tecmint ~]#
```

PROCESS MANAGEMENT IN LINUX

- Process in Linux is treated as a directory with a set of files corresponds to the process.
  - Stored in /proc
- Various information regarding the process can be found in this folder.
  - Address space of the process
- Uses cat command to see the information contained in the file inside the folder.
  - Example: cat /proc/[PID]/status

EXERCISE

- Rerun and analyze the work of syntax in forkexample.cpp, exec.cpp and hello.cpp
- Modify hello.cpp to sum two numbers which come from exec.cpp.
- Shows the address space of the process inside your machine (minimum 3 process).
- Try to modify the type of process in Linux. (Stop, Foreground, Background).
- Analyze the variation of ps command to monitor the process in Linux.