**TAKE HOME ASSIGNMENT**

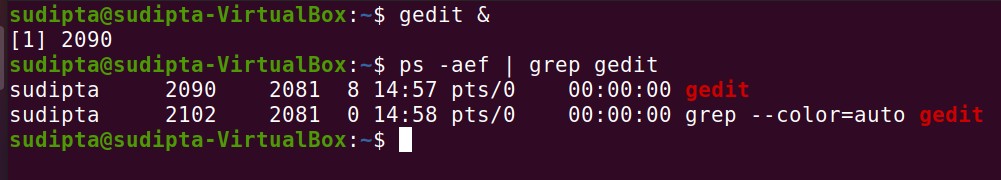
**Name: Sudipta Halder**

**Roll: 2021202011**

**SET - 2**

1. Clearly differentiate b/w Process, Lightweight process, and Threads. What are process descriptors and different process states?

* **Difference between Process, Lightweight Process and Threads**
  + **Process**
    - A process is a program in execution. For example, when we write a program in C or C++ and compile it, the compiler creates binary code. The original code and binary code are both programs. When we actually run the binary code, it becomes a process.[[1]](#endnote-1)
    - Processes are priority based. Based on this priority, kernel switches context between these processes. A process can be pre-empted if a process with higher priority is ready to be executed. For example, let’s assume a process is waiting for a system resource like some text file which is residing on disk. At the same time, a process with higher priority is also waiting for the same resource. Then the kernel would give the resource to the higher priority process and get the back to the waiting process when the other process completes its execution and resource is available. In this way, the resources are shared among tasks and user feels that tasks are being run in parallel manner. Processes can talk to other processes using inter process communication (IPC) and can share data using techniques like shared memory.
    - In Linux, fork() is used to create new processes. When a process is created, it is almost identical to its parent. It receives a (logical) copy of the parent's address space and executes the same code as the parent, beginning at the next instruction following the process creation system call. At the beginning of the program, each child process shares all segments like stack, heap, text etc. until child process tries to make any change to stack or heap. In case of any alteration, a separate copy of stack and heap segments are prepared for child so that changes remain child specific. The text segment is read-only so both parent and child share the same text segment.
    - I ran gedit on my machine in background. I ran ps command which gives us the status of gedit process.



* + **Linux Threads vs Light Weight Processes**
    - Threads are the core element of a multi-tasking programming environment. By definition, a thread is an execution context in a process. Hence, every process has at least one thread. Multi-threading implies the existence of multiple, concurrent (on multi-processor systems), and often synchronised execution contexts in a process.
    - For a non multi-threaded process there is one execution flow that is the main execution flow and hence it is also known as single threaded process.
    - For Linux kernel, there is no concept of thread. Each thread is viewed as a separate process by kernel and these processes are obviously somewhat different from normal processes.
    - An Light Weight Process(LWP) is a process created to facilitate a user-space thread. Each user-thread has a 1×1 mapping to an LWP. The creation of LWPs is different from an ordinary process; for a user process “P”, its set of LWPs share the same group ID. Grouping them allows the kernel to enable resource sharing among them (resources include the address space, physical memory pages (VM), signal handlers and files). This further enables the kernel to avoid context switches among these processes. Extensive resource sharing is the reason these processes are called light-weight processes.
    - Often we mix these two terms ‘Threads’ and ‘Light Weight Processes(LWP)’. The reason dates back to those times when Linux supported threads at user level only. For kernel there was no concept called Thread. This means that even a multi-threaded application was viewed by kernel as a single process only. This posed big challenges for the library that managed these user level threads because it had to take care of cases that a thread execution did not hinder if any other thread issued a blocking call.
    - Later, the implementation changed and processes were attached to each thread so that kernel can take care of them. But as discussed earlier, Linux kernel does not see them as threads, each thread is viewed as a process inside kernel. These processes are known as Light Weight Processes.
    - The main difference between a light weight process (LWP) and a normal process is that LWPs share same address space and other resources like open files etc. As some resources are shared so these processes are considered to be light weight as compared to other normal processes and hence the name light weight processes.
    - So, effectively we can say that Threads and Light Weight Processes(LWP) are the same. Thread is a term used at user level perspective and LWP is a term used at kernel level perspective.
    - From implementation point of view, threads are created using functions exposed by POSIX compliant pthread library in Linux. Internally, the clone() function is used to create a normal as well as alight weight process. This means that to create a normal process fork() is used that further calls clone() with appropriate arguments while to create a thread or LWP, a function from pthread library calls clone() with relevant flags. So, the main difference is generated by using different flags that can be passed to clone() function.
    - clone() is similar to fork(), but more generic. Actually, fork() itself is a manifestation of clone(), which allows programmers to choose the resources to share between processes. The clone() call creates a process, but the child process shares its execution context with the parent, including the memory, file descriptors and signal handlers. The pthread library too uses clone() to implement threads.
    - Below is a sample c program for implementing thread(user level perspective):

***#include <stdio.h>***

***#include <syscall.h>***

***#include <pthread.h>***

***int main()***

***{***

***pthread\_t tid = pthread\_self();***

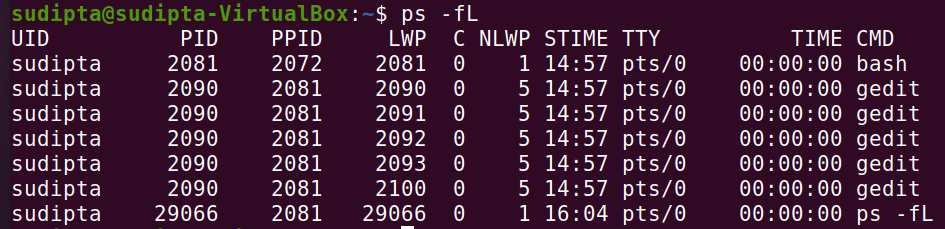
***int sid = syscall(SYS\_gettid);***

***printf("LWP id is %dn", sid);***

***printf("POSIX thread id is %dn", tid);***

***return 0;***

***}***

* + - If I run the ps -fL command, it will list processes and their LWP/ threads information: 
    - Below is a c program which creates LWP using clone().

***#include <malloc.h>***

***#include <sys/types.h>***

***#include <sys/wait.h>***

***#include <signal.h>***

***#include <sched.h>***

***#include <stdio.h>***

***#include <fcntl.h>***

***// 64kB stack***

***#define STACK 1024\*64***

***// The child thread will execute this function***

***int threadFunction( void\* argument ) {***

***printf( "child thread entering\n" );***

***close((int\*)argument);***

***printf( "child thread exiting\n" );***

***return 0;***

***}***

***int main() {***

***void\* stack;***

***pid\_t pid;***

***int fd;***

***fd = open("/dev/null", O\_RDWR);***

***if (fd < 0) {***

***perror("/dev/null");***

***exit(1);***

***}***

***// Allocate the stack***

***stack = malloc(STACK);***

***if (stack == 0) {***

***perror("malloc: could not allocate stack");***

***exit(1);***

***}***

***printf("Creating child thread\n");***

***// Call the clone system call to create the child thread***

***pid = clone(&threadFunction,***

***(char\*) stack + STACK,***

***SIGCHLD | CLONE\_FS | CLONE\_FILES |\***

***CLONE\_SIGHAND | CLONE\_VM,***

***(void\*)fd);***

***if (pid == -1) {***

***perror("clone");***

***exit(2);***

***}***

***// Wait for the child thread to exit***

***pid = waitpid(pid, 0, 0);***

***if (pid == -1) {***

***perror("waitpid");***

***exit(3);***

***}***

***// Attempt to write to file should fail, since our thread has***

***// closed the file.***

***if (write(fd, "c", 1) < 0) {***

***printf("Parent:\t child closed our file descriptor\n");***

***}***

***// Free the stack***

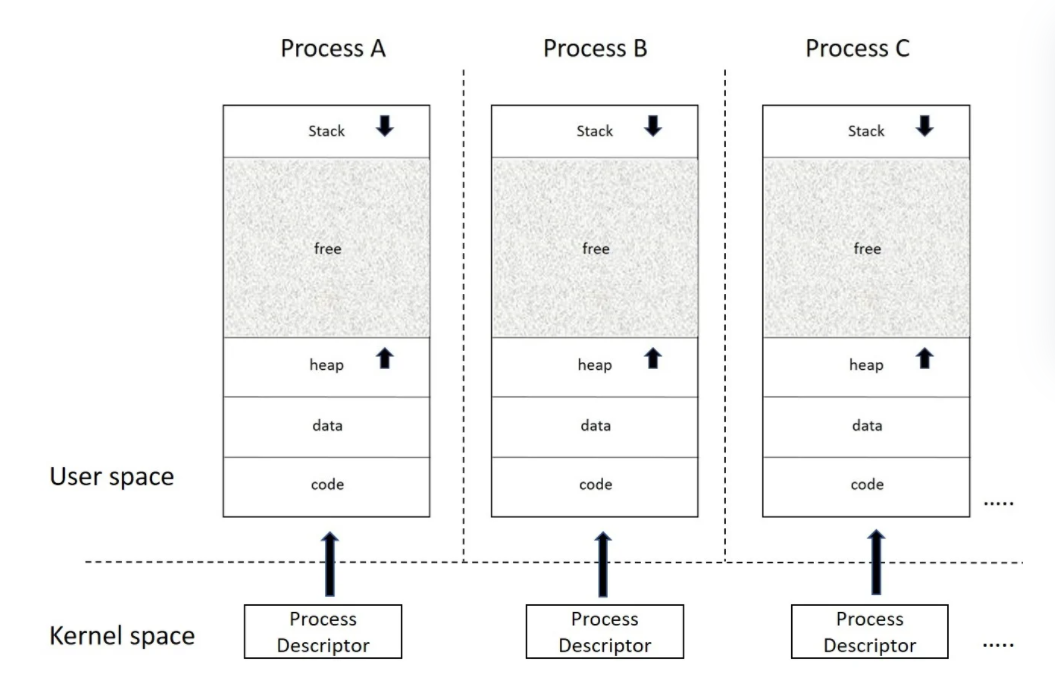
***free(stack);***

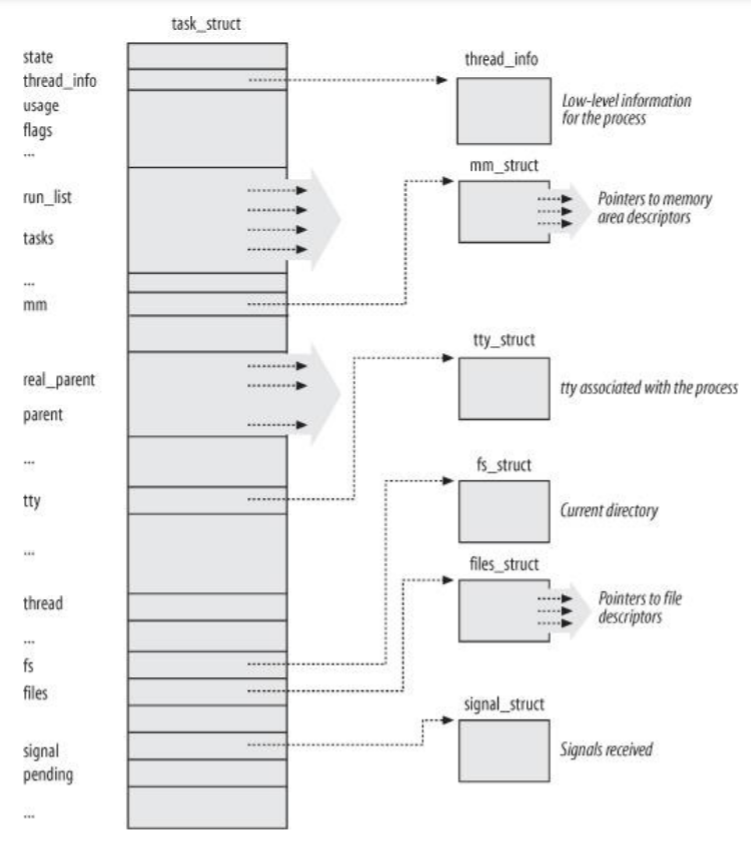
***return 0;***

***}[[2]](#endnote-2)***

* **Process Descriptors**
  + Right from the time a process is born until it exits, the kernel’s process management subsystem carries out various operations, ranging from process creation, allocating CPU time, and event notifications to destruction of the process upon termination.

Apart from the address space, a process in memory is also assigned a data structure called the process descriptor, which the kernel uses to identify, manage and schedule the process. The following figure depicts process address spaces with their respective process descriptors in the kernel:

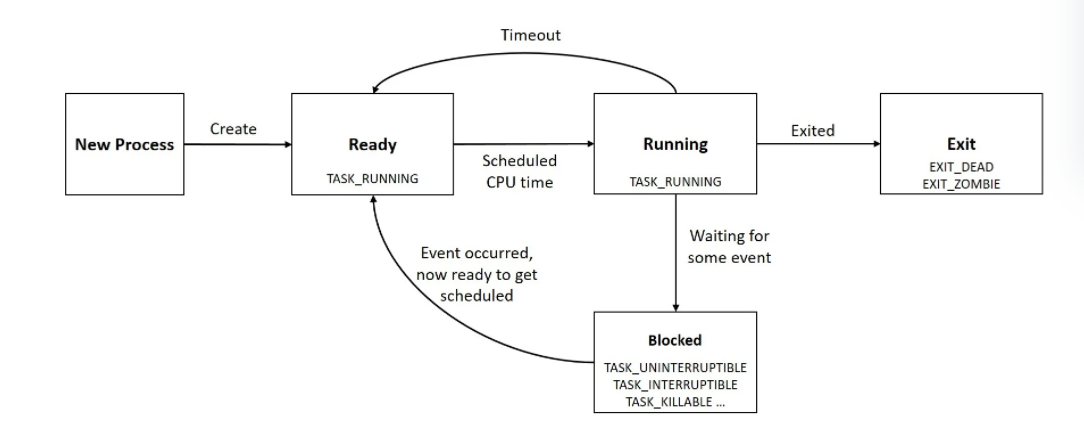




* + In Linux, a process descriptor is an instance of type struct task\_struct defined in <linux/sched.h>, it is one of the central data structures, and contains all the attributes, identification details, and resource allocation entries that a process holds. Looking at struct task\_struct is like a peek into the window of what the kernel sees or works with to manage and schedule a process.[[3]](#endnote-3)
* **Process States**

A process right from the time it is spawned until it exits may exist in various states, referred to as process states. The state field of the process descriptor describes what is currently happening to the process. It consists of an array of flags, each of which describes a possible process state. In the current Linux version, these states are mutually exclusive, and hence exactly one flag of state always is set; the remaining flags are cleared. The following are the possible process states:

* + **TASK\_RUNNING (0):** The task is either executing in CPU or contending for CPU in the scheduler run-queue.
  + **TASK\_INTERRUPTIBLE (1):** The process is suspended (sleeping) until some condition becomes true. Raising a hardware interrupt, releasing a system resource the process is waiting for, or delivering a signal are examples of conditions that might wake up the process (put its state back to TASK\_RUNNING).
  + **TASK\_KILLABLE:** This is similar to TASK\_INTERRUPTIBLE, with the exception that interruptions can only occur on fatal signals, which makes it a better alternative to TASK\_INTERRUPTIBLE.
  + **TASK\_UNINTERRUTPIBLE (2):** The task is in uninterruptible wait state similar to TASK\_INTERRUPTIBLE, except that generated signals to the sleeping process do not cause wake-up. When the event occurs for which it is waiting, the process transitions to TASK\_RUNNING. For instance, this state may be used when a process opens a device file and the corresponding device driver starts probing for a corresponding hardware device. The device driver must not be interrupted until the probing is complete, or the hardware device could be left in an unpredictable state. This process state is rarely used.
  + **TASK\_STOPPED(4):** Process execution has been stopped; the process enters this state after receiving a SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal. The process will be back to running on receiving the continue signal (SIGCONT).
  + **TASK\_TRACED (8):** A process is said to be in traced state when it is being combed, probably by a debugger. When a process is being monitored by another (such as when a debugger executes a ptrace( ) system call to monitor a test program), each signal may put the process in the TASK\_TRACED state.
  + **EXIT\_ZOMBIE (32):** The process is terminated, but its resources are not yet reclaimed. This happens when process execution is terminated, but the parent process has not yet issued a wait4( ) or waitpid( ) system call to return information about the dead process. Before the wait( )-like call is issued, the kernel cannot discard the data contained in the dead process descriptor because the parent might need it.
  + **EXIT\_DEAD (16):** The child is terminated and all the resources held by it freed, after the parent collects the exit status of the child using wait. This happens when the process is being removed by the system because the parent process has just issued a wait4( ) or waitpid( ) system call for it. Changing its state from EXIT\_ZOMBIE to EXIT\_DEAD avoids race conditions due to other threads of execution that execute wait( )-like calls on the same process.[[4]](#endnote-4)
  + A picture is given below which depicts the different process states.[[5]](#endnote-5)



1. <https://www.geeksforgeeks.org/introduction-of-process-management/> [↑](#endnote-ref-1)
2. <https://www.opensourceforu.com/2011/08/light-weight-processes-dissecting-linux-threads/?amp> [↑](#endnote-ref-2)
3. <https://subscription.packtpub.com/book/application-development/9781785883057/1/ch01lvl1sec9/process-descriptors> [↑](#endnote-ref-3)
4. Understanding the Linux Kernel, 3rd Edition [↑](#endnote-ref-4)
5. <https://subscription.packtpub.com/book/application-development/9781785883057/1/ch01lvl1sec9/process-descriptors> [↑](#endnote-ref-5)