Part B A2

It would be better to implement process tagging completely in userspace because we can achieve the same task in userspace and programming in userspace is much safer as any small mistake in kernelspace could cause lead to big problems on the machine while in userspace you will probably only get a compiler error. I don't get the job because the bank needs secure program and kernel level programming is more secure than user level and based on my reply it would seem like I am not comfortable with kernel-space.

Printk() is a kernel level function, which has the ability to print out to different loglevels as defined in <linux/kernel.h> . The major difference between printk() and printf() is the capability of the former to specify a loglevel. The kernel uses the loglevel to decide whether to print the message to the console.

The malloc() can be called in both user-space and kernel-space, and it allocates a physically fragmented memory area. While kmalloc() can be called only in kernel-space, and it allocates physically contiguous memory chunk. There exist a userspace and a kernel space so it only makes sense if there are different libraries that seperates the privledged functions with the restricted ones.This is necessary to prevent user applications from interferring with the hard drive. Thus, the kernel cannot access some userspace API. System libraries (such as glibc, libreadline, libproplist) that are typically available to userspace programmers are unavailable to kernel programmers. The loader will automatically load any dependent libraries into the address space of the process when being loaded.

The concept of running a process as root is a unix/linux term that means you're logged in as the administrator of the system. Any process you run, whether as root or a normal user, generally runs in both user mode and kernel mode. The system is continually switching between user mode (where the application code runs) and kernel mode (where the kernel code runs). Some programs, like many device drivers, always run in kernel mode, meaning they have full access to the hardware. A normal application running with root privileges still exists in user mode and only switches to kernel mode when a kernel system call is made and then switches right back to user mode.

Linux does not reserve empty space in the system call table like many modern operating systems (Solaris, FreeBSD, etc.). If its possible to use it then we wont have to compile the kernel each time which saves time but the reason to restrict it is that we might replace an existing system call and you mess up the program making it unable to boot into it.

**User space threads Advantages**: Can be implemented on an opetating system that does not support threads. Each process has their own customized scheduling algorithm. They scale better doesnot require table space or stack space in kernel which can be a problem if there are a very large number of threads.

**User space threads disadvantages:** User-space threads have a problem that a single thread can monopolize the timeslice thus starving the other threads within the task. Also, it has no way of taking advantage of the Symmetric MultiProcessor system features, e.g. dual-/quad-Pentiums. System calls are blocked so cannot do I/O or allocate memory. System calls will stop all the threads.

**Kernel space threads Advantages:** There is a table for each process with one entry per thread. So a task is less likely to hog the timeslice from the other threads within the task. No systtem run time is needed. The process automatically can take advantage of SMPs and will run incrementally faster with each added CPU**.** No blocking needed.

**Kernel space threads disadvantages:** System calls are costly. If there are a lot of thread creation, deletion, syncronization then it will be a bad choice. The buffer in one thread might get immiediately replaced by another because of clock interrupt. In order for this to be fixed the entire library have to most likely be over written.

6)

A thread would do that in order to avoid deadlocks or so one thread wouldn't replace the buffer in another thread right away. It could be necessary for a thread to yield itself in order for the process to continue successfully. We also don't want a thread hogging all the process memory for itself. This can create problems if a thread is waiting for a resource to become available. We want a process to give up control once it is at a stopping point.

7)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Processes | Burst Times(ms) | Priority | Round Robin with 2ms quantum (ms) | Priority Scheduling (ms) | First-come, First-served (ms) | Shortest job first (ms) |
| A | 10 | 2 | 30 | 24 | 10 | 30 |
| B | 6 | 0 | 22 | 6 | 16 | 12 |
| C | 2 | 3 | 6 | 26 | 18 | 2 |
| D | 4 | 4 | 16 | 30 | 22 | 6 |
| E | 8 | 1 | 28 | 14 | 30 | 20 |

8)

Switching from one process to another requires a certain amount of time for doing things such as saving and loading registers and memory maps, updating various tables and lists, flushing and reloading the memory cache, and so on. Suppose that this context switch takes 2ms, including switching memory maps, flushing and reloading the cache, etc. Also suppose that the quantum is set at 10ms. With these parameters, after doing 10ms of useful work, 2ms will be wasted on process switching. This makes the CPU very inefficient. To improve the efficiency the quantum should be set to a higher value.

9)

There will be three processes that will be created after executing the given segment of code.

P

/ \

P C

/ \ / \

P C C C

Answer 10

Wrapper functions are used to make writing computer programs easier. Wrapper functions are useful in the development of applications that use third-party library functions. A wrapper can be written for each of the third party functions and used in the native application. If a third party functions are updated or changed only the wrappers in the native application need to be modified as opposed to changing all instances of third party functions in the native application. So using wrapper function is more user-friendly as well as more efficient. execve() preserves all process attributes except:

*- The dispositions of any signals that are being caught.*

*- Any alternate signal stack.*

*- Memory mappings.*

*- Attached System V shared memory segments.*

*- POSIX shared memory regions.*

*- Open POSIX message queue descriptors.*

*- Any open POSIX named semaphores.*

*- POSIX timers.*

*- Any open directory streams.*

*- Memory locks.*

*- Exit handlers.*

*- The floating-point environment.*

execve() does not have a return because  if it fails to do what it is trying to do, i.e. replace the process with the new one, then it makes sense to return, to inform the caller that it failed. If it succeeds, then the code that called execve() will no longer be present, it has been replaced by the successful execution of that function, so obviously it cannot do anything any more.

Citations:

<http://www.informit.com/articles/article.aspx?p=25075&seqNum=4>

<http://unix.stackexchange.com/questions/121715/what-is-the-relationship-between-root-and-kernel>

<https://www.quora.com/Is-it-possible-to-add-a-System-Call-via-a-Loadable-Kernel-Module>

http://www.e-reading.club/chapter.php/143358/128/Tanenbaum\_-\_Distributed\_operating\_systems.html

https://www.quora.com/What-is-the-difference-between-threads-operating-in-user-space-kernel-space-and-hybrids

http://unix.stackexchange.com/questions/47701/adding-a-new-system-call-to-linux-3-2-x-with-a-loadable-kernel-module

<http://kernelnewbies.org/FAQ/LibraryFunctionsInKernel>

Man Pages