CS167 Homework Assignment 3

Due 11:59pm, Wednesday, April 9, 2025

1. [34%] In many Unix-based systems (such as Linux and Weenix), a process's address space is represented by a linked list of **vmarea** structures, each representing a separate piece of the address space and describing the access permissions and what has been mapped into the region of the address space. Such operating systems often employ "lazy evaluation" in which they postpone many operations in hopes of not having to do them.

   a. [10%] Assume the OS is running on a (32-bit) x86-based system that employs a two-level page-translation mechanism with a top-level page-directory table that has 2^{10} entries that each refer to page tables with 2^{10} entries, each of which refer to pages of size 2^{12} bytes. Consider the life of a process, from its creation via **fork** to its termination via **exit**. What are the events that cause the page directory to be allocated and what are the events that cause the page tables to be allocated? In particular, what actions, such as system calls, memory references, etc., by threads within the process cause the page directory to be allocated and what are the events that cause page tables to be allocated? (Note this doesn't happen all at once.)

   b. [8%] Recall that in 32-bit Linux, until recently all physical memory was directly mapped into the kernel's address space[1]. However, once it became affordable for the amount of physical memory in a machine to become greater than 2^{30} bytes (one gigabyte), this was no longer possible, and just the first gigabyte, minus a small amount, is directly mapped into the kernel. However, in 64-bit Linux, things are different: many, many gigabytes of physical memory can easily be directly mapped into the kernel's address space. Assume that by Moore's law the amount of physical memory one can afford doubles every two years, and assume one can now (in April 2025) afford up to sixteen gigabytes. (Further assume that Moore's law will continue to hold for the rest of the current century.) Will it still be possible to directly map all the physical memory one can afford at the end of the current century into the kernel address space of 64-bit Linux? (The last day of the current century is December 31, 2100.) Explain.

   c. [8%] It is possible (but you don't need to show how it's done) to allocate an individual thread's stack in a way so it doesn't need to occupy contiguous locations in memory. Thus thread stacks could be allocated from the heap (the dynamic region) on demand as they grow. (The first page of the stack could be in one part of the address space, the next page allocated from another part of the address space, and so forth.) This requires more processor time than in the standard approach, so it's not normally done. Explain why such an approach might be useful if one is to support thousands of concurrent threads on 32-bit Linux. (Keep in mind that the default stack size on Linux is 2^{21} bytes. Assume that by thousands of threads we mean 2^{14}). [Hint: not all threads need stacks as large as 2^{21} bytes.]

   d. [3%] We now want to support thousands of concurrent threads on 64-bit Linux (by "thousands", we mean 2^{14}). Can this be done without having to resort to the use of non-contiguous stacks? Explain.

[1] What this means is that the size of the usable portion of the kernel address space is equal to the amount of real memory in the system. The first byte of kernel virtual memory is mapped (via page tables) to the first byte of real memory, and consecutive byte of kernel virtual memory are mapped to consecutive bytes of real memory.

e. [5%] We should be concerned about the cost of creating a stack: is it cheaper to initially create a 10-kilobyte stack than it is to initially create a two-megabyte stack (the default size in Linux), assuming that in either case, the thread uses at most 10k of stack space? The architecture is x86-64, which is much like IA-32, except there are four levels of page tables rather than two.

2. [33%] It's been argued that the pattern of creating a new process by doing a combination of fork and exec is too expensive for some applications. We're going to examine the cost of this combination. Let's assume we are using the (32-bit) x86 architecture, with two-level paging and a page size of 4K. Each address space is represented by a one-page page-directory table containing 1024 (2^{10}) 4-byte entries, each of which may refer to a one-page page table containing 1024 4-byte entries, each of which may refer to a page of physical memory. We consider a process that has one read-only region of memory (containing executable code and constants) and two read-write regions of memory (one containing BSS, data, and dynamic storage, the other containing the stack). Assume that **mmap** is not used and the process is single-threaded. Assume that the stack region occupies 16 kilobytes (2^{14} bytes) and the other regions are one megabyte (2^{20} bytes) each. The first two (i.e., the larger) regions are at the low end of the address space, the stack region is at the high end (just below the kernel). Furthermore, assume that **esp**, the stack pointer register, points to a location that is 512 bytes from the low-addressed end of the stack (the stack grows from high addresses down to low addresses). Assume that it takes four microseconds ($4 \cdot 10^{-6}$ seconds) to copy one page of primary memory (or a fraction thereof).

   a. [6%] Recall that the effect of executing a **fork** system call is to create an exact copy of the calling process. Let's assume that our system does nothing to optimize the making of this copy, i.e., it does not employ copy on write and thus all the pages of the parent process are copied to create the child. How long would it take to copy all relevant memory when creating the child? (Note: page tables are included in relevant memory. They must be copied, then modified; we won't include the cost of modifying them.)

   b. [5%] Early Unix systems employed the **vfork** system call as an optimization of fork when the call to fork was to be followed by a call to exec. In this optimization, the child process executes in the parent's address space (on the same stack as the parent) until it calls exec, then a new address space is created for it. How long would it take to copy all relevant memory as part of the implementation of **vfork**?

   c. [6%] Weenix (and other Unix-based systems) optimize **fork** by using copy-on-write techniques. In particular, the child process's initial address space is virtually identical to that of the parent, but copy-on-write techniques are used to avoid copying any memory. (Note that new page tables and a new page directory are created.) How long would it take to copy all relevant memory as part of this optimized **fork**?

   d. [6%] Continuing with this optimized version of **fork**, suppose our process makes a few changes to file descriptors and then calls exec. Though nothing is modified in the first read-write region of memory, a maximum of 256 bytes of memory are pushed onto the stack. In addition to the cost of storing new items to the stack, how much time is spent copying relevant memory?

   e. [6%] We'd like to improve the optimization used in part c. What might we do to do so? (Hint: consider using copy on write on additional data structures, perhaps some used for address translation.) With your new optimization, how long would it take to copy all relevant memory as part of the fork implementation and then place its call to exec?

   f. [4%] There's a relatively new system call known as **posix_spawn** that has the effect of combining **fork** and **exec** into a single system call (it has all the arguments used by exec, as well as additional arguments indicating how file descriptors should be modified). If it were implemented optimally (with respect to the copying of the parent process's address space), how much cheaper would it be for our process than the optimized version of **fork** (followed by **exec**) used in part d?

3. [33%] In lecture we mentioned that it should be possible to set up a restricted protection domain in Unix in which a process has access only to what it needs. In this problem we will see if this really is possible. You might want to review lecture XX, starting with slide YY, or the textbook, starting on page 328.

   Let's assume the existence of a user ID that we'll call PUB. No files are owned by PUB (except for perhaps a few that we set up for this problem); no files explicitly give PUB any form of access; and PUB is not a member of any group. Thus a user running with the PUB user ID has access only to files for which all users have access.

   We'd like to create a shell command **secureRun** that has three arguments: a pathname of an input file, a pathname of an output file, and the pathname of a program we'd like run that will be able to read the input file as stdin, write to the output file as stdout, and have no access to any other files except for those that give all users access.

   a. [11%] You (user ID YOU) have downloaded a program into /home/you/malware. The program is supposed to modify, in a useful way, what it reads from stdin and write it to stdout. You don't trust it, but you'd like to run it, hopefully safely, using the command

      ```
      secureRun input output /home/you/malware
      ```

      /home/you/malware is executable by everyone (thus including PUB). YOU has access to input and output, but PUB does not. Thus PUB shouldn't have access to any of your other files.

      How can secureRun set its user ID to PUB, yet allow access to the files **input** and **output**? [Hint: it might be necessary for it to exec some other, intermediary, program that's been set up in a special way.]

   b. [11%] Although the process is running as user ID PUB and has access to the files **input** and **output**, is it prevented from accessing other files you own? Explain.

   c. [11%] What else should be done to make sure /home/you/malware cannot do anything other than what you intend for it to do? [Hint: read the man page for **setreuid**.]