Homework 3

Virtual Memory and I/O

1. Suppose we have a machine with a 2^{42} (42 bit) virtual address space, a maximum of 2^{42} bytes of physical memory (more than enough to load all of virtual memory), and a page size of 2^{24} bytes. Assume a single-level page table to begin. Please keep your PTE sizes to powers of 2. If your page table entry doesn't have a power-of-2 number of useful bits, add some unused padding to it.
    a. Draw a virtual address, splitting it into the virtual page number and the virtual page offset. Note how many bits are used for each.
    b. Draw a plausible page table entry, showing the length in bits of each of the fields. How many of your entries fit on a page?
    c. Assume virtual addresses from 0x10000000 to 0x10ffffff and from 0xf0000000 to 0xf0ffffff are in use. How much space does your page table occupy?
    d. Suppose there are five identical processes currently running, and each of them is using the same virtual addresses mentioned in part c. How much space is occupied by their page tables, in total? (Hint: easy question).
    e. Describe a plausible 2-level page table approach for this system and repeat a-d assuming that approach. For part a, note which part of the virtual page number is used for which level of the table.

2. The sigaction() system call registers a handler function to be run when a particular event is signaled by the kernel. It is like signal(), but we can arrange for the handler function to get additional arguments, such as the address of the instruction that was interrupted to deliver the signal. One kind of event is SIGILL, which indicates that the current instruction (the one that is interrupted) is invalid. If we attempt to execute an instruction that the hardware cannot decode, the hardware will generate an illegal instruction fault, and the kernel will cook it to a SIGILL signal for the process.

    The intern at our fintech is given code that occasionally triggers a SIGILL.[1] This is catastrophic since the default handler for SIGILL simply aborts the program, resulting in us losing lots of money. The intern tries to fix this problem with the following code:

```
void handler(int signo, char *ptr_to_instruction)
{
    *ptr_to_instruction = 0x90;   // 0x90 is the "no-op" instruction,
                                  // which does nothing
}

int main() {
   sigaction(SIGILL,handler);

   code_that_may_have_invalid_instructions();
}
```

    a. This does indeed eliminate the immediate problem---the program continues to run and appears to make progress. Why?
    b. However, as the program runs, it starts to make weirder and weirder trading decisions. Why?

[1] One reason why our code may have invalid instructions is because it has been compiled and optimized for a different target microarchitecture. For example, we may compile for a machine which has the fancy AVX512 vector instruction set, but then sometimes run the code on a machine that only has AVX2. Any AVX512 instruction will look like jibberish to this machine.

3. The fork() system call is used to create a new process that is a duplicate of the process that makes the system call. The created process (the child) and the original process (the parent) have the same memory content. However, their memory is **not** shared: when the child modifies its memory, these changes do not affect the memory of the parent, and vice versa. Here is an example:

```
int val = 0;
int ret = fork();
if (ret > 0) {
   // the parent
   val = 1;
} else if (ret == 0) {
   // the child
   val = 2;
   execl("/bin/ls", "/bin/ls", NULL);
} else {
   // ret < 0, fork() failed.
   exit(1);
}
```

Here, if fork() succeeds, the parent and the child will each hold an individual copy of the variable val. Note the parent sets val to 1 while the child sets val to 2. Each of the processes will see only its own change to val – these modifications will **not** interfere with each other. Notice this is very different from the pthread_create() function you are using in the SETILab.

A naïve implementation of fork() would simply allocate memory for the child, create a page table for it, and then copy all the memory contents of the parent process to the child process. Copying a process's memory is very expensive, which would make the fork() system call take a long time. However, Linux's implementation of fork() is extremely quick because it avoids this expense - its *immediate cost* is just the creation of the child process's page table. Its *long term cost* is that the first write to any page by the parent or the child will take longer than under the naïve implementation.

A. Can you describe how such a low overhead procedure could be implemented using virtual memory mechanisms? You may want to re-read the section on copy-on-write in your textbook.

B. It is a common case that the child process will call execl() to load and run a new program very soon after the fork()function call, like the above example does. execl() discards all current memory content in the process and its page table. Thus, even the fast mechanism in Linux (part A of this problem) is slower than it could be. vfork() is system call that can optimize this case even further. A vfork() is similar to a fork(), but after a vfork(), the child should not write any memory before it calls execl(). If it does, the results are specified as being ***undefined***.[2] How may vfork()be implemented?

4. Your book talks about mark-and-sweep garbage collection. A completely different approach is known as stop-and-copy garbage collection. In this approach, the heap memory is divided in half into the "working memory" and the "free memory". The program always uses just the working memory. When the program runs out of memory, it stops and garbage collection is invoked. As the garbage collector traverses the graph of nodes reachable from the root nodes (i.e., the non-garbage), it copies

[2]If you see the term "undefined" in a specification or manual, be aware that it is the committee-speak way of saying "all hell may break loose if you do this."

each reachable node to the free memory, placing the nodes sequentially in the free memory in the order in which they are traversed. By means of mechanisms that are not important here, each node is copied only once, and all the pointers between the reachable nodes are updated appropriately. Then, the garbage collector simply starts treating the free memory as the working memory and vice versa, essentially throwing away all the unreachable nodes en masse. One often finds that stop-and-copy greatly increases cache hit rates, especially for pointer-based data structures such as lists, trees, and graphs. Why?

5. Traceroute (/bin/traceroute) lets you trace the route which your packets take from source to destination. Ping (/usr/bin/ping) lets you measure the round-trip latency to a host. Dig (/usr/bin/dig) and whois (installed on some systems, also available via the web) let you find detailed information about DNS names. Try these commands out. (a) Pick three of your favorite web sites, or other destinations, and then run dig, whois, ping, and traceroute on them. Hand in the results. (b) Try to find a destination site that you think is very close to you, and one that you think is very far from you. Run traceroute and ping to them. How close are they, really? (Consider the speed of light). (c) determine the bandwidth and latency of your network connection. You can do this by searching for "speed test" in google. Document this.

6. Exceptional control flow invokes the kernel. Consider interrupts (exceptions due to external events). On your computer of choice, figure out how to determine the interrupt rate (e.g. using powermetrics on MacOS, /proc/interrupts on Linux, Performance Monitor on Windows, …), and document what it is, both when the computer is idle and when it is busy.

7. Run the equivalent of the Linux top/htop commands on your computer (e.g. activity monitor on mac, task manager on windows). Document what you see. How many processes and threads is your computer typically running? What percentage of the time is spent in user code, kernel code, or idle?