1 UNIX System Calls

1. Consider the C function printf() on UNIX. Is printf() implemented by the OS, or by an application-level library? What system call does printf() make internally?

2. Why are `mkdir`, `ln` and `rm` implemented as separate user-level programs, while `cd` is implemented as a built-in command?

3. On a linux machine, type the following command

   ```
   $ cat | tee output.file
   ```

   cat is a UNIX utility that prints the contents of STDIN to STDOUT. tee is a UNIX utility that prints the contents of STDIN to both STDOUT and to the file named by its argument (output.file). After you type this command, you can type in some characters followed by the newline character.

   a. While this command is running, examine the processes created:

   Use pstree to see the process hierarchy. Tell us what you find about the process hierarchy. Use 'ps x' to identify the process IDs of the processes created by cat and tee commands. Linux provides a proc pseudo-filesystem which can be used to examine the state of a process using filesystem namespace.

   Type the following command for both process-ids:

   ```
   $ ls -l /proc/pid-num/fd/
   ```

   b. What do you find? What are 0,1,2,...? What do they symlink to?

   ```
   $ ls -l /proc/self/fd/
   ```

   c. What do you find? What is 3 pointing to? Why?

   d. Which system calls are executed while you run `cat`, `tee`, and `ls` as in the above commands?

4. Assume you are given two text files containing newline-separated strings. You wish to write a program that takes as input two text files and outputs another text file that contains all the strings that are common in the two input files. For example, if the two text files are `foo` and `bar`, with the following contents:

   ```
   foo:
   hello
   os
   class

   bar:
   os
   is
   an
   interesting
   class
   ```

   Then, the output would be:

   ```
   output:
   os
   class
   ```

   (`os` and `class` are common words in the two files). Assume you are given the following functions:

i. A function `cat()` that takes input from its first argument and displays its content on the standard output (file-descriptor 1). e.g.,

```
cat(foo);
```

This function call will cause the contents of the file `foo` to get streamed to the standard output of the program.

ii. A function `sort()` that takes input from its standard input (file-descriptor 0), sorts the input text (assuming it consists of newline-separated words), and prints the output to standard output (file-descriptor 1).

iii. A function `intersect()` that takes input from file descriptors 3 and 4 (non-standard file descriptors). It assumes that both inputs (from each file descriptor) are sorted, and computes the intersection of the sorted streams of words, outputting to the standard output (file descriptor 1).

iv. The `dup()`, `pipe()` and `fork()` system calls.

You are not allowed to use any other system calls (e.g., `open`, `read`, etc.) or any other functions. Using these functions, implement a utility that given two filenames, outputs the common strings in the two files on the standard output.

5. What is the total number of processes at the end of the execution of the following program? Assume there is one process in the beginning that starts running at main. Also, assume that all system calls succeed.

```
main() {
  fork();
  fork();
  fork();
}
```

Explain.

6. Consider the following program:

```
main() {
    int fd;
    fd = open(outfile, O_RDWR)
    fork();
    write(fd, hello, 5);
    exit();
}
```

Assume all system calls finish successfully on a uniprocessor system. Also, assume that a system call cannot be interrupted in the middle of its execution. What will be the contents of the outfile file, after all processes have successfully exited? Explain briefly.

7. Consider the following program:

```
main() {
    int fd;
    fd = open(outfile, O_RDWR)
    fork();
    write(fd, hello, 5);
    exit();
}
```

Assume all system calls finish successfully on a uniprocessor system. Also, assume that a system call cannot be interrupted in the middle of its execution. What will be the contents of the outfile file, after all processes have successfully exited? Explain briefly.

8. Give two interesting applications where you think the user-defined SIGUSR1 and SIGUSR2 can be useful?

9. In UNIX, a child process may terminate before a parent calls wait(). When the parent calls wait() eventually, it still expects to read the correct exitcode that the child returned. To support this functionality, UNIX does not completely remove the process till it's parent has called wait() on it.

   Such processes that have completed execution but still have an entry in the process table are called zombie processes. Usually, the presence of zombie processes in the system for a long time indicates a bug in the program (it is a common error).

   UNIX also provides the SIGCHLD signal, which is received by the parent process whenever one of its children exits.

   In class we discussed that the shell implements "&" functionality by not calling wait() immediately. Should the shell never call wait()? When should it call wait()? Answer by providing short pseudo-code. (Hint: you may want to use the SIGCHLD signal).

10. UNIX uses `fork()` and `exec()` system calls to create new processes. On the other hand, Microsoft Windows has a system call called `CreateProcess()`. Here is a sample definition of the `CreateProcess()` call (this is not identical but similar to Windows system call):

```
Boolean CreateProcess(
    lpApplicationName,    /* Name of the executable. */
    lpCommandLine,        /* Command line parameters. */
    bInheritHandles,      /* Boolean variable indicating if the new process
                             should inherit the open file descriptors of the
                             calling process. */
    lpProcessAttributes,  /* attributes of security attributes of process. */
    lpCurrentDirectory,   /* current working directory of new process. */
    lpProcessInformation  /* A pointer to Process Information structure that
                             receives identification information about the
                             new process. */
     . . .                /* ignoring some other arguments. */
);
```

   Write code to implement the equivalent of this call on UNIX. You should show how you use `lpApplicationName`, `lpCommandLine`, `bInheritHandles`, `lpProcessAttributes`, `lpCurrentDirectory`, and `lpProcessInformation` on UNIX.

11. Write the pseudo-code for a program "`cp`" that takes two command-line arguments, say input-file and output-file, copies the input-file to the output-file.

```
Syntax:
$ cp ifile ofile

Program:

int main(int argc, char **argv)
{
  //your solution goes here.
  //Use UNIX system calls to implement the logic.
}
```

2 Threads

1. What is an address space? How does the operating system ensure that a variable of one program/process does not clobber a variable of another process?

2. Each process has a separate address space. When is the new address space created?

3. What are threads? How are they different from processes? What are the advantages? Give examples to motivate the use of threads.

4. What is the difference between user-level threads and kernel-level threads? Do user-level threads provide true physical concurrency? If not, why are they useful?

5. Threads can be implemented completely at the user level. i.e., we do not require privileged operations to implement a thread abstraction and schedule different threads. In other words, a process can provide multiple threads by implementing a scheduler. Let's see how this can be done.

   To implement threads, the process needs to provide the abstraction of multiple control-flow (program counter), multiple register sets and multiple stacks. This can be done if after every periodic time interval, one thread can be interrupted and saved and another thread can be loaded. Saving a thread involves saving it's program counter, registers and stack pointer. Similarly, loading a thread involves loading the new thread's program counter, registers and stack pointer. Neither the save operation, nor the load operation requires any privileged operation – we are just loading and saving registers.

   So the only remaining issue is how to periodically interrupt a running thread from within a process. For an OS, this interruption is done by the hardware timer device. A process can do this using the SIGALRM signal.

   Such threads implemented inside a process are called user-level threads. The OS cannot distinguish between multiple user-level threads and it can only see one process that is running which includes the thread scheduler and the different threads.

   Read the manpage of the signal, alarm, and setitimer. Understand how SIGALRM can help in implementing user-level threads. Briefly describe how you will do this (2-3 sentences and some pseudo-code).

3 PC Architecture and Compiler Conventions

1. Consider the following function:

```
int32_t global;

int32_t foo(int32_t a, int32_t *b) {
  int32_t c;

  c = global + a;

  return *(b + c);
}
```

   Assume that the variable global is allocated at a global address 0x12345. Write the assembly code for this function, with proper comments on which assembly code lines are implementing which C statement. Assume GCC calling conventions. You will need to be careful about properly naming all variables and arguments (e.g., using global addresses, stack offsets or frame pointer offsets), use proper opcodes and addressing modes, obey caller and callee-save conventions, etc.

2. A new compiler `hcc` is developed. What should `hcc` be careful about if it wants to use the libraries that were compiled using `gcc`?

3. To implement function calls, typically compilers use conventions about which registers to save. Some registers are saved by the caller (before the call instruction) and some registers are saved by the callee (in the function body). For example, gcc follows the following convention on x86:

   ```
   %eax, %ecx, %edx are "caller save" registers (saved by caller)
   %ebp, %ebx, %esi, %edi are "callee save" registers (saved by callee)
   ```

   Why is a convention needed? Give an example of a program where using the convention of caller and callee saved registers helps in reducing the number of saves and restores of registers during function calls. What should the compiler be careful about to maximize this optimization opportunity? Hint: Are the values of all registers useful (i.e., will they be used in future). If a value in a register will be used in subsequent instructions, that register is called live. If a value in a register will not be used in future (e.g., it will get overwritten by another value), that register is called dead. Can the caller save only live registers and ignore dead registers? What about callee? How does splitting into caller-saved and callee-saved registers improve this optimization opportunity? For example, which registers should the caller try to use first for storing its temporary values? Similarly, which registers should the callee try to use first for storing its temporary values? Why?)

4. Look at the following program:

   - C code

     ```
     int main(void) { return f(8)+1; }
     int f(int x) { return g(x); }
     int g(int x) { return x+3; }
     ```

   - Assembly code

     ```
     _main:
       __prologue__
       pushl %ebp
       movl %esp, %ebp
       __body__
       pushl $8
       call _f
       addl $1, %eax
       __epilogue__
       movl %ebp, %esp
       popl %ebp
       ret

     _f:
       __prologue__
       pushl %ebp
       movl %esp, %ebp
       __body__
       pushl 8(%esp)
       call _g
       __epilogue__
       movl %ebp, %esp
       popl %ebp
       ret

     _g:
     ```

```
__prologue__
pushl %ebp
movl %esp, %ebp
save %ebx
pushl %ebx
__body__
movl 8(%ebp), %ebx
addl $3, %ebx
movl %ebx, %eax
restore %ebx
popl %ebx
__epilogue__
movl %ebp, %esp
popl %ebp
ret
```

Notice that the function implementation only needs to obey the compiler conventions. Another smaller (and faster) but correct implementation of `g` is the following:

```
_g:
  movl 4(%esp), %eax
  addl $3, %eax
  ret
```

What would be the smallest possible (but correct) implementation of `_f`?

5. Consider the following function:

```
int32_t global
int32_t foo(int32_t a, int32_t *b) {
  int32_t c
  c = global + a
  return *(b + c)
}
```

Assume that the variable global is allocated at a global address 0x12345. Write the assembly code for this function, with proper comments on which assembly code lines are implementing which C statement. Assume GCC calling conventions. You will need to be careful about properly naming all variables and arguments (e.g., using global addresses, stack offsets or frame pointer offsets), use proper opcodes and addressing modes, obey caller and calleesave conventions, etc. It is okay to not be exactly correct in the use of x86 opcodes, but the general layout of the code and its logic should be correct.

4 Segmentation and Trap Handling

1. Consider the following program header of an ELF executable file `a.out`:

```
LOAD: offset    0x00001000
      vaddr 0x40100000
      paddr 0x00100000
      align 2**12
      filesize 0x0000b596
      memsize 0x000126fc
      flags rwx
```

Assume that this executable is loaded using the `exec(a.out, )` system call on 32-bit Linux. Also, assume that the Linux kernel is mapped starting at virtual address `0xc0000000`.

a. Draw the layout of the virtual address space of the process just after successful completion of the exec() system call. Indicate the sizes, and the contents of the memory regions, wherever possible.

b. Draw the layout of the physical memory of the computer just after this executable is loaded. Assume, segmentation is used for implementing virtual memory address spaces. Show the contents of the segment registers (CS, DS, SS, etc.), the global-descriptor table (GDT), and GDTR.

2. The hardware does not allow an (unprivileged) application to execute the `lgdt` instruction (to load the GDT). Why? What could happen if an application was allowed to execute the `lgdt` instruction?

3. How does the OS ensure through segmentation that one application cannot access another application's address space in the following situations:

   (a) The application tries to write to the physical address of the other application.

   (b) The application tries to modify the segment register

   (c) The application tries to overwrite GDT entries

   (d) The application tries to lower its privilege-level (i.e., tries to gain supervisor privileges).

4. The instruction to load the Interrupt Descriptor Table Register (IDTR) is "`lidt`" and is a privileged instruction, i.e., it can only be executed in privileged mode. Assume that it was possible to execute this instruction in user mode by an untrusted user process. Show an attack using this additional (hypothetical) capability, whereby:

   - A user process can crash the machine.
   - A user process can read the memory contents of another process.

   You should show the steps that the user process should follow to launch this attack in as much detail as possible

5 Paging

1. Assume a memory access latency of 100ns, and a 2level page table hierarchy on 32bit x86. What should be the TLB hit rate to ensure that the average memory access latency is 102ns. Assume there are no instruction/data caches in the hardware.

2. Consider the following program header of an ELF executable file `a.out`:

```
LOAD: offset    0x00001000
      vaddr 0x40100000
      paddr 0x00100000
      align 2**12
      filesize 0x0000b596
      memsize 0x000126fc
      flags rwx
```

   Assume that this executable is loaded using the `exec(a.out, )` system call on 32-bit Linux. Also, assume that the Linux kernel is mapped starting at virtual address `0xc0000000`.

   a. (repeat from Chapter 4) Draw the layout of the virtual address space of the process just after successful completion of the exec() system call. Indicate the sizes, and the contents of the memory regions, wherever possible.

   b. Assume that the operating system is using paging to map the pages of the executable on x86 using a twolevel page table. Also assume that it is not using large pages i.e., it is only using 4KB pages to map the process and kernels address space. Assume that the size of the physical memory is 4MB and it is entirely mapped in the kernel address space (starting at 0xc0000000). Also, assume that the kernels code and data takes 1MB of physical memory space (start at physical address

0x100000). Draw the page table and indicate the values stored in them. Especially, say which entries will be present and where they will be mapped (what are the likely values of these entries). Assume all space is mapped with rwx privileges (but of course, differing in user/kernel privileges).

3. What are the different ways in which an unpriviliged application can cause the CPU to start running in privileged mode?

4. On a trap, the hardware changes the contents of the ESP register if the trap causes an escalation in privilege. Why does it need to do so? Where is the new value of ESP obtained from?

5. In a certain OS, when executing in kernel mode, a trap on vector $0x80$ causes three words to get pushed on stack by hardware, namely CS, EIP and EFLAGS. On the other hand, when executing in user mode, a trap on vector $0x80$ causes five words to get pushed on stack by hardware, namely CS, EIP, EFLAGS, SS, and ESP. What does it say about the contents of the Interrupt Descriptor Table (IDT) at $0x80$ entry? Why does the hardware push a different number of words in the two cases?

6. Should an operating system allow an unprivileged (untrusted) application to modify the task-state segment (which is used to obtain the value of ESP on a trap)? Why or why not?

7. A process-model kernel maintains a separate kernel-side stack (the stack to which control transfers on a privileged trap) for each process. On the other hand, an interrupt-model kernel maintains a single stack for the kernel. What are the advantages and disadvantages of each design?

8. Consider the first question in Section 4. Draw the layout of the physical memory of the computer just after a trap occurs during the execution of this executable. Show the contents of the segment registers (CS, DS, SS, etc.), the global-descriptor table (GDT) and GDTR.

9. On a trap, certain parts of the processor state (CS, EIP, EFLAGS) are saved by the hardware. The other parts of the processor state (e.g., other segment registers, other general-purpose registers like EAX, etc.) are saved by software (i.e., the first few instructions executed after a trap). (Note that the first few instructions executed on a trap are a part of the trap handler). Why this distinction? Why can't all registers be saved by the software? Conversely, why does the hardware not save all elements of the state on a trap?

10. The trap-handling mechanism is used to implement system calls. Explain the sequence of steps that a kernel must follow to ensure that a user can invoke any system call at will, but cannot otherwise subvert the security of the computer.

11. Explain the sequence of steps executed by the hardware on the execution of the `iret` instruction.

12. *Advanced Question*: Traps can be caused due to exceptions (triggered by an application through a software instruction or a violation, e.g., segmentation fault) or by external interrupts (e.g., caused by a device like network card, disk, etc.). In both cases, the trap handling mechanism remains same. What are the advantages of using the same mechanism of trap handling for both internal exceptions and external interrupts?

13. *Advanced Question*: On debuggers like GDB, a programmer is allowed to set a "breakpoint", which means that if the execution of the program ever reaches the breakpoint address, then the execution is interrupted. One way to implement breakpoints is to replace the instruction at the address of the breakpoint with a software interrupt instruction. Whenever the execution reaches the breakpoint, a trap gets generated and the kernel can transfer control to the debugger. How does the debugger know the contents of the program at the time of reaching the breakpoint? Explain the sequence of steps required to save the program state, so the programmer can examine it.

14. How much virtual address space is named by one entry in the page-table (PTE)? How much virtual address space is named by one entry in the page-directory (PDE)?

15. What is the smallest page table structure required on 32-bit x86, to implement the following address space. By smallest, we mean that you should use the minimum amount of space for the page table structure.

16. Give examples where the same page in physical address space is pointed-to in multiple page-table entries or page-directory entries. The aliasing page-table or page-directory entries may be present in the same page table structure, or different page table structures.

17. The space overhead of a page table can be computed as:

$$\frac{VAbytes}{PTsize}$$

Here, $VAbytes$ are the total number of bytes in the virtual address space that are (validly) named by the page table. For example, if the process uses 3123 bytes of stack and 432 bytes of code, then the value of $VAbytes$ for that process is $3123 + 432 = 3555$. The $PTsize$ is the total number of bytes required for representing the page table — this includes number of bytes required to represent the page directory and the page table pages (including invalid/not-present entries). For example, two pages are required to represent an address space with valid bytes ranging from 0-100 (one for the page directory and one for the second-level page table). Thus the $PTsize$ in this case would be $2 * 4096 = 8192$ bytes.

Draw a page table and its corresponding VA space which:

(a) has the maximum page-table space overhead (as computed by the fraction $\frac{VAbytes}{PTsize}$). What is the space overhead in this case?

(b) has the minimum page-table space overhead (as computed by the fraction $\frac{VAbytes}{PTsize}$). What is the space overhead in this case?

18. Assume a memory access latency of 100ns, and a two-level page table hierarchy on 32-bit x86. What should be the TLB hit rate to ensure that the average memory access latency is 102ns. Assume there are no instruction/data caches in the hardware.

6 Kernel Structures for Implementing Processes

1. List a few in-memory data structures used by the kernel to store information about processes. Which of these data structures are visible to the process? How does the kernel ensure that all these structures are protected from the untrusted process? What would be the typical space-overheads of these data structures (you do not need to be exact, but some rough estimates are good enough)? Compare these to typical process sizes (again, use some representative programs that you use daily, and roughly estimate their size).

2. `malloc` and `free` are functions available both for the kernel and for the user programs. What is the difference between kernel's functions and user program's functions? Can they have identical implementations — what are some necessary differences?

3. Explain the problem of fragmentation, in the context of malloc() and free(). Compare and contrast this with the problem of fragmentation, as it exists with segmentation-based virtual memory.

4. Each process has a unique `pid`. Assume that the `pid` is a 32-bit integer. Does that mean that there can be only 2^{32} number of processes across the lifetime of execution of the computer? When can the `pid`'s be recycled (i.e., reused for new processes).

5. At any time, let m be the number of PCBs whose state is RUNNING and n be the number of CPUs in the computer. What is the relation between m and n?

6. An OS configures preemption by configuring the timer interrupt device to generate an interrupt periodically (e.g., every 10ms). On each interrupt, the trap is generated and the *handler* executes (by transferring control through IDT). What should the handler do to implement preemptive scheduling (i.e., a process can be interrupted at any time)?

7. The instruction to load the Interrupt Descriptor Table Register (IDTR) is `lidt` and is a privileged instruction, i.e., it can only be executed in privileged mode. Assume that it was possible to execute this instruction in user mode by an untrusted user process. Show an attack using this additional (hypothetical) capability, whereby:

   (a) A user process can crash the machine.

   (b) A user process can read the memory contents of another process.

   You should show the steps that the user process should follow to launch this attack in as much detail as possible.

8. How often does the value of `ss0` and `esp0` change in the task-state segment (TSS) in the

   (a) Interrupt-model kernel. i.e., one kernel stack per CPU

   (b) Process-model kernel. i.e., one kernel stack per process

   Assume a uniprocessor system.

9. Assume that a CPU is receiving external interrupts with uniform distribution at an average frequency of 100Hz. Assume that the kernel is using the process model (one kernel stack per process). Also assume that the kernel is non-preemptible, i.e., if a process is executing in kernel mode, it cannot be context-switched out until it returns back to user mode. During execution in kernel mode, the process uses its kernel stack to store its state. The kernel is using a 4KB stack.

   However, the kernel developer has forgotten to ensure that all external device interrupt handlers should execute with interrupts disabled. i.e., none of the IDT entries specify that interrupts should be disabled before transferring control to the handler.

   Answer the following questions:

   (a) If an external interrupt handler executes for roughly 10 microseconds, and does not disable interrupts during its execution, what is the probability that the kernel stack would overflow? Assume that one execution of an external interrupt handler pushes around 512 bytes to the stack in ramp-up/tear-down fashion (i.e., the kstack first grows by 512 bytes uniformly and then shrinks by 512 bytes uniformly distributed over the execution time of the interrupt handler).

   (b) A friend informs the kernel developer that this is a bug in his kernel; and so he fixes the bug by ensuring that the first instruction in all external interrupt handlers is `cli` and the last instruction in the interrupt handler restores the original value of the interrupt flag (before calling `iret`). Does this ensure correct execution in all cases? If not, what is the new probability that the kernel stack would overflow?

10. Because threads can access shared state concurrently, a bad thread interleaving could potentially result in incorrect program behavior (if the program is not written carefully). Such a situation is called a race condition. Assume that you are developing a new OS, lets call this YOS (your-own OS). Assume that YOS runs only on uniprocessor machines.

    (a) Is it possible for a multi-threaded application to have a race-condition when running on YOS (on a uniprocessor system)? Why/why not? Clearly state the assumptions you are making regarding your OS design to justify your answer.

    (b) One way of disallowing race conditions is to use locks. Your friend suggests that a simple way of implementing locks in YOS is to implement two system calls called `disable_interrupts()` and `enable_interrupts()`. He suggests that because YOS runs only on uniprocessor systems, an application programmer can use these system calls to ensure atomicity of its critical sections. If you agree with him, implement functions `lock(L)` and `unlock(L)` using the system calls `disable_interrupts()` and `enable_interrupts()`.

    (c) Is it a good/bad idea to provide such system calls (enable/disable interrupts) to the application developer? Why/why not?

7 Process Switching, Fork, Scheduler

1. Explain the steps involved in

   - A context-switch that occurs because a process voluntarily calls `yield()`.
   - A context-switch that occurs due to a pre-emptive context switch triggered by a hardware timer interrupt.
   - A context-switch that occurs because a process tries to read from the disk (slow device), which causes it to wait (and thus relinquish the CPU).

   For each step, also indicate the approximate time it takes to execute that step. For example, a trap takes a few micro-seconds (including saving and restoring trapframes), saving and restoring registers, reloading page table, etc. Also, briefly discuss the indirect cost of context-switch, e.g., TLB flush.

2. Estimate the cost of forking a new process. Compare the cost of fork, with and without the copy-on-write optimization.

3. How does a doubly linked-list help in implementing a round-robin scheduler? Why is a linked-list preferred over more complex data structures like min-heap or binary-search trees, to implement process lists in the kernel?

4. Explain how context-switch can be implemented simply by switching stacks. Do we need to save all registers, or only callee-saved registers on a context-switch? Why?

8 Creating the First Process

1. What is the typical functionality of the first process? What is its purpose?

2. How is the first process initialized? Explain, using an example program that can be used as a first process, while using UNIX abstractions.