SOLUTIONS BY SHUN (@shun4midx), HE MAY BE WRONG.

Multiple choice questions: (40 pts)

5pts each, deduct 3pts if one answer is wrong, the lowest score for each question is 0pts (Put down the answers - explanation is not required)

1. Which of the following about Unix is/are true?
   - X a. The first Unix Operating System was built by Linus Trovalds (It is Bell Labs instead)
   - X b. The PID of the shell process in a Unix system is usually 0 (0 is the swapper)
   - O c. In Unix, users often interact with the OS kernel via a special program called a shell. One can choose the shell that she wishes to use by default modifying the environment variable
   - O d. POSIX standardizes the services, including the system calls that a Unix system must provide
   - O e. In Unix, system calls generally take much longer than function calls to finish

2. Which of the following is/are true about a process?
   - X a. A process created by fork() always starts running immediately
   - O b. A process created by fork() inherits all file descriptors from its parent process (gets own fd but points to same entry anyway) fork ⇒ fd table copied, open file table entries and i-nodes shared
   - X c. vfork() shortens the process creation time because the kernel employs copy-on-write to avoid the memory copying overhead (fork() in modern systems also does COW)
   - X d. A process that exec() a program can never become an orphan process (exec only replaces current program image, not the process tree)
   - X e. In Unix, whenever a process terminates, the OS kernel recycles all of the process' respective resources, so the parent of the terminating process does not have to perform extra recycling work ☆ The kernel free most but not all: e.g. process control block (PCB) incl exit status — kept until parent calls wait() → zombies exist

3. Assuming that you log in to the linux1 server with your account, you can read or execute the file "hello" from the server. You are in a group created for all students (GID = 7), while all faculty members are in another group (GID = 8). Which of the following is/are true about the file? (rwx = 4 2 1)
   - X a. The UID of the file is the root user. The access permission is set to 4700 (4 here ⇒ SUID, 2=SGID, 1=sticky bit; we execute as UID ⇒ 7 = rwx)
   - X b. The UID of the file is the root user. The GID of the file is 3. The access permission is set to 4750 (other ⇒ 0)
   - O c. The UID of the file is you. The access permission is set to 555 (user = 5 ✓)
   - X d. The UID of the file is another student who takes SP. The GID of the file is 7. The access permission is set to 547 (group ⇒ 4, stopped at 4, never reaches 7 ⇒ no rtx; we are group)
   - O e. The UID of the file is Professor Pu-Jen Cheng. The GID of the file is 8. The access permission is set to 705 (other = 5 = rtx ✓; we are other)

   Takeaway: SUID/SGID only takes effect after the file begins executing, not before determining execution.

4. Which of the following is/are true about I/O performance?
   - X a. A slow system call, such as one that reads data from the disk, may be blocked forever — never forever, it must end!! Slow sys call = depends on I/O or external input (i.e. uncontrolled by CPU)
   - O b. A system call that waits for incoming network connections is a slow system call
   - X c. A database system could use synchronous writes (open() with the O_SYNC flag) to make file updates faster (sync only makes sure file updates are in sync → it's not faster)
   - O d. A system that enforces mandatory locking on file accesses may perform worse than a system that does not (↳ may slow you down)
   - O e. The OS kernel could perform read ahead if sequential reads are detected

SOLUTIONS BY SHUN (@shun4midx), HE MAY BE WRONG.

5. Which of the following is/are true about hard links and symbolic links?
   - O a. There is no actual file created in the file system when a new hard link is created
   - O b. There is an actual file created for a symbolic link when a new symbolic link is created
   - X c. One can use the command "ln" provided by your shell to create only symbolic links but not hard links "-s" can create sym links but w/o the flag defaults to hard links
   - O d. A symbolic link can link to a directory
   - X e. A hard link can link to a file on a different file system (hard link cannot link to a diff file sys, hence the appeal of sym link)

6. Which of the following is/are true about user identification?
   - X a. The OS kernel allocates a new UID whenever the user logs in each user has their unique UID stored in /etc/passwd... not reset at login
   - X b. For any process, its real UID is always equal to the effective UID (SUID exists...)
   - O c. The system searches for the default shell program to execute using the user's login ID (ex: root) During login, the env is not set yet, so determined by /etc/passwd (/etc/passwd)
   - X d. The process that executes a set-uid program will have its real UID updated No, that's the point of SUID...
   - O e. In Unix, different users must have unique UIDs. However, users may share the same GID

7. Which of the following is/are true about file I/Os?
   - X a. Consider two processes, A and B. Process A can pass its file descriptor to process B, so process B can use the file descriptor to access the file associated with the file descriptor fd are not universally unique across diff processes
   - X b. You can use the open() or creat() function to create a new file and specify the access permissions of the newly created file - the resulting file permissions of the file are always the same as specified by open() and creat() umask may result in diff perms
   - X c. Performing write() to a newly opened file with existing contents always writes the bytes from the beginning of the file (assuming the write() succeeds) O_APPEND writes to the end
   - X d. Assume a process opens a file a.txt successfully. It can then use lseek() alone to decrease the size of a.txt offset can be shifted but no information is actually lost
   - O e. Consider a case in which we read() 100 bytes from an opened regular file (size=50 bytes), with the current file offset set to 0. Assuming that the current read() returns 50. The next read from the file returns 0 (When we are at EOF, return 0 directly)

8. Which of the following is/are true about the Unix file system?
   - X a. One can set the sticky bit of a directory to ensure any files within the directory are readable to only its respective owner sticky bit only guards exec/rename, not read
   - X b. All open files of a given process share the same entry in the system open file table (Fd table: local to proc; global across proc stores)
   - X c. The contents of a file are stored directly in the file's associated i-node
   - X d. Assume that a program holds a file lock of a file a.txt but does not release the lock. After a process executes this program, no other program can ever open a.txt for reading (i-node contains metadata) The lock could be a reading lock, you can have >1 user reading at the same time when locked
   - O e. The device files are the files associated with hardware devices (Device files... by definition)

SOLUTIONS BY SHUN (@shun4midx), HE MAY BE WRONG.

Short answer questions: (60 pts)

A. "Everything is a file" is a key feature in Unix. Explain why this simplifies programming efforts. (5 pts)

Then, everything can be done with a shared interface: file I/O.

This means we can treat different devices or network connections as special files, so we can access them in the same way we access regular files.

B. Describe the key differences between blocking I/O, non-blocking I/O, and I/O multiplexing. Use an example to explain why using I/O multiplexing could provide better performance than the other two. (5 pts)

- Blocking I/O: Block the process until read/write is completed
- Non-Blocking I/O: Read and write as much as possible and return quickly to not block the process
- I/O Multiplexing: Block the process when there is no read/write available, return and resume the process once any read/write is ready. This is usually done using select() or poll()

Example

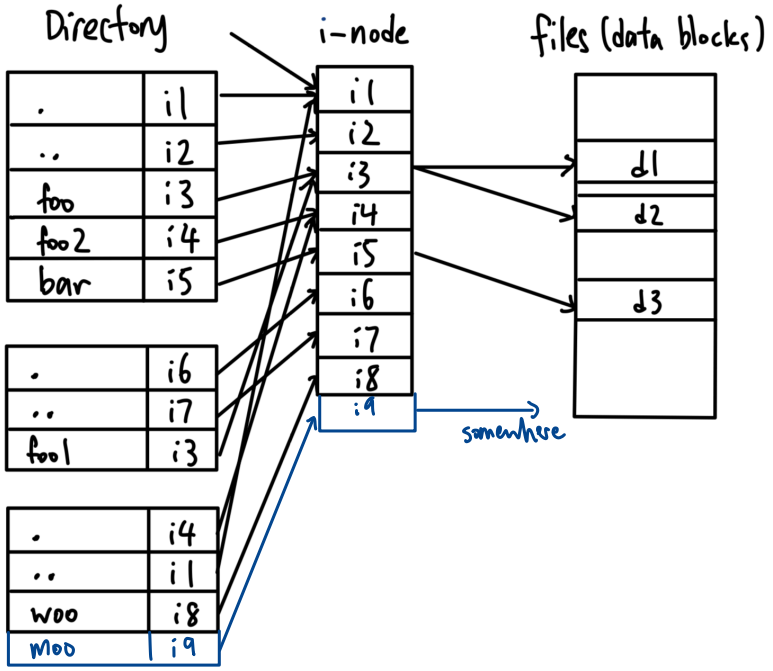
A server with many clients

↳ Blocking I/O: Handles one at a time → slow

↳ Non-blocking I/O: Wastes CPU polling

↳ I/O Multiplexing: Handle many processes at the same time (fast) + low CPU usage

SOLUTIONS BY SHUN (@shun4midx), HE MAY BE WRONG.



C. Please refer to the figure above. Assume “foo” is a regular file located within the directory “/home/sp02/”. (12 pts in total, 3pts each)
   a. i1 has three i-node links. Explain why.
   b. What does the figure look like after executing unlink(foo)?
   c. What is the absolute pathname for “woo”?
   d. Assume we want to create a symbolic “moo” in the directory where “woo” is within. Draw the changes in the figure after “moo” gets created.

a) Three different directories point to the directory as a file corr. to i1.
(from its parent, itself, and its child)

b) Only 

is removed, other two i3 i-node links remain.

c) /home/sp02/foo2/woo

d) As drawn

SOLUTIONS BY SHUN (@shun4midx), HE MAY BE WRONG.

D. Alice implements a function, setlock(), that tries to acquire a write lock on the file and returns 0 on success. If any other processes have already held locks on the file, setlock() immediately returns -1. However, Bob comes and points out that the implementation below is buggy. Explain why. (6 pts)

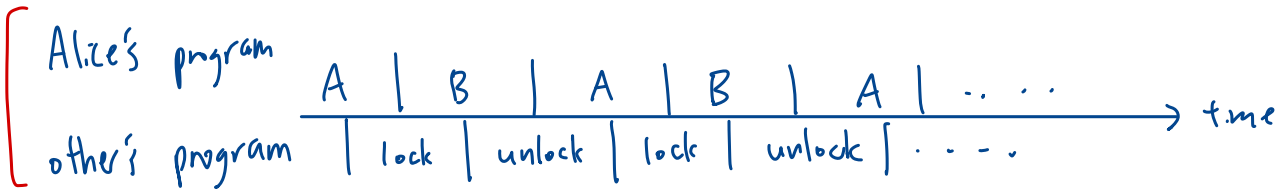
Need to consider multiprocess

Need not be syntax, can be race condition

```
int SP_FLOCK(int fd, int cmd, off_t start, off_t len) {
      struct flock fl;
      fl.l_type = F_WRLCK;
      fl.l_whence = SEEK_SET;
      fl.l_start = start;
      fl.l_len = len;
      if (cmd == F_SETLK)
            return fcntl(fd, cmd, &fl);
      else if (cmd == F_GETLK) {
            fcntl(fd, cmd, &fl);
            if (fl.l_type != F_UNLCK)
                  fprintf(stderr, "lock held by pid %d\n", fl.l_pid);
            return fl.l_type;
      }
}

int setlock(int fd, off_t start, off_t len) {
      while (1) {
            // Assume that arguments fd, start, len are always valid
            ret = SP_FLOCK(fd, F_GETLK, start, len);   <-- A
            if (ret == F_UNLCK) {
                  ret = SP_FLOCK(fd, F_SETLK, start, len);   <-- B
                  if (!ret)
                        return 0;
            } else
                  return -1;
      }
}
```

Notice, if there is another program running at the same time, some process may acquire read/write lock for the same file in between A and B, so this program needs to loop and try again. If we have the following scenario, the program won't respond immediately.



(reoccurable testing concept about file locking)

SOLUTIONS BY SHUN (@shun4midx), HE MAY BE WRONG.

E. Answer the questions below. Assuming a process has executed the code below. (the path points to a regular file) (8 pts)

```
fd1 = open(path, oflags);
fd2 = dup(f1);
fd3 = open(path, oflags);
link(path, path2); // path & path2 differ
dup2(fd2, 1);
```

// path2 ← path

// 1 ← fd2

1. Which time values of the file specified by *path* is/are modified? (3 pts) (access, modification, change time)
2. How many updates to the process' file descriptor table were made? (3 pts) (fd corr. to i-node)
3. Assuming the same process calls fcntl(fd1, F_SETFD, flags) after dup2(), which descriptor(s) is/are affected? (2 pts)

1) With open(path, oflags), this modifies the access time of the file specified by path. With link(path, path2), this modifies the i-node of the file, so change-status time is modified.

2) Line 1: Updates fd1
Line 2: Updates fd2
Line 3: Updates fd3
Line 4: NO UPDATE
Line 5: Updates fd=1
⇒ 4 updates

★ link modifies i-node (which corr. to a file) but not fd table itself

3) fd1 only ∵ FD-flags are per fd only, not per corr. entry. (open file table only updates when open, lseek, read, write with offset)

F. Why is running the following code dangerous? What does Unix do to limit the problem? (4 pts)

```
while (1) fork();
```

1. It will fork nonstop ⇒ the number of processes will increase exponentially and infinitely, which is a lot of overhead and may exceed the memory of the system
2. Unix sets a limit on the maximum number of processes a single process can create, so "infinitely many processes" is avoided

SOLUTIONS BY SHUN (@shun4midx), HE MAY BE WRONG.

G. Assume you are on the CSIE workstations. Your home directory is synchronized across all workstations, while other directories are not. The admins impose an upper limit on the number of processes that a single user is allowed to create. Suppose you have exceeded the limit on linux1.csie, and you're unable to login because the server refuses to create new processes for you. Meanwhile, other users are still able to run their programs successfully.

Given that you cannot get the admin privilege on linux1.csie or share your password with others. Describe the approach that you would take to re-enable login. (6pts)

1. Login to linux2.csie, and write some code that has SUID which kills my processes in linux1.csie and put it in my home dir
2. Tell someone else to login to linux1.csie and execute the program (with permission granted to them)

☆ Works because home dir can access all workstations, no matter which linux I'm on, it's still the same files in /home/ cuz mounted

H. The code below is buggy. List at least two potential issues. (8 pts)

```
int main(int argc, char **argv) {
      if (argc < 2) exit(1);
      struct stat statbuf;
      char *buf;
      ssize_t count;
      int fd;
      fd = open(argv[1], O_RDONLY);
      if (fd < 0) exit(1);
      fstat(fd, &statbuf);
      buf = (char *)malloc(statbuf.st_size);
      while ((count = read(fd, buf, 1)) > 0)
            buf += count;
      return 0;
}
```

☆ open, close, etc ⇒ single process

1. ssize_t count may = 0 ⇒ buf when malloc() may cause error
2. Not closing fd ⇒ fd leak
3. Not freeing buf before return ⇒ memory leak
4. Not checking if fstat was valid ⇒ may create undefined statbuf behavior

SOLUTIONS BY SHUN (@shun4midx), HE MAY BE WRONG.

I. Write a program in C that creates a zombie process. You should make sure that you can always find the zombie process via “ps” after the program is executed. You can ignore library headers. Make sure you use the parameters correctly. (6pts)

```
int main {
    if (fork()==0) {
        exit(0);
    } else {
        while(1) wait(1);
    }
    return 0;
}
```