Systems Programming (Fall, 2021)
Mid-Term Exam Sample

closed book & closed notes

1. Explanation (as clearly as possible for full credits).
   (a) System calls allow processes to trap into the kernel. Give two examples of system calls to explain under what circumstances one would trigger a context switch and the other would not.
   (b) Explain under what circumstances advisory lock is safe even though there are other processes probably violating a lock and trying to access the locked file.
   (c) Explain why communication through a pipe should be limited to processes that descend from a common ancestor.
   (d) Explain under what circumstances a UNIX-like system would automatically turn off the set-group-ID bit of a newly created file.
   (e) Explain why it's common for a process to create and open a new file and then immediately unlink it; however, it's not very useful for a process to create and open a directory and then immediately remove it.

2. The *cp* utility copies the content of a source file to a target file. When *cp* is implemented with unbuffered I/O system calls, the following four factors (A)~(D) would significantly affect its execution time. Please answer the questions.
   (A) The number of the while loops
   -- Each loop copies partial file content with *read*() or *write*()
   (B) The time to wait for data ready in memory
   (C) The time to copy data from kernel's buffer cache to user's buffer and vice versa
   -- User's buffer denotes the buffer specified in *read*() and *write*()
   (D) The time to move data from kernel's buffer cache to disk and vice versa
   (a) What factor(s) will significantly affect user CPU time?
   (b) What factor(s) will significantly affect system CPU time?
   (c) What factor(s) will significantly affect clock/response time?
   (d) What factor(s) will be significantly affected when nonblocking I/O is taken into account, compared with blocking I/O?
   (e) What time (user CPU time, system CPU time, and clock/response time) will be significantly affected by system call *fsync*()?
   (f) Suppose the target file is redirected to null device */dev/null*. We run the *cp* utility with different buffer sizes in blocking mode and get the following results.

Buffer size	User CPU (sec)	System CPU (sec)	Clock Time (sec)
1	124.89	161.65	288.64
2	63.10	80.96	145.81
32	4.13	5.01	9.76
128	1.01	1.27	6.82
4096	0.03	0.16	6.86
8192	0.01	0.18	6.67

(f.1) Increasing the buffer size beyond 4096 has little positive effect on system CPU time. Why?

(f.2) When the buffer size is small, the difference between clock time and total CPU time (user CPU + system CPU) is also small, e.g., the sum of 124.89 and 161.65 is close to 288.64. But such difference increases significantly when the buffer size comes to 128. Why?

(g) Suppose the time to read the source file is ignored. We run the *cp* utility with different sizes of source files in blocking mode and get the following results.

Source size	User CPU (sec)	System CPU (sec)	Clock Time (sec)
10M	0.000	0.008	0.010
300M	0.000	0.236	0.239
1000M	0.000	0.824	0.827

Disk I/O is assumed to be time consuming; however, clock time is close to system CPU time for different sizes. Why?

3. Alice and Bob propose a method to share files securely. Each of them creates two directories *OutBox* and *InBox* in her/his home directory. *OutBox* is the place to store the files to be shared; *InBox* is the place to store the links pointing to the shared files. If Alice wants to share her file *hw* with Bob, she can just put the file in *~Alice*/*OutBox* and then create a link pointing to *~Alice*/*OutBox*/*hw* in *~Bob/InBox*. For security issue, no one is able to traverse both of *OutBox* and *InBox* except the owner. Only the owner can remove his/her own files from *OutBox*. Assume Alice and Bob belong to different groups and they both have no superuser privileges. Please answer the following questions.
   (a) What is the advantage of creating a link in *InBox* over copying a file to *InBox*?
   (b) Alice can put a symbolic link or a hard link in Bob's *InBox*. Which choice is better? Explain your answer.
   (c) What **minimum** access rights (i.e., read, write, and execute) should be used for the following directories? Only consider the **others** class. Your answer should be in the form of "rwx", "r-x", or the like.
      (1) Directory *InBox*
      (2) Directory *OutBox* if hard links are adopted
      (3) Directory *OutBox* if symbolic links are adopted
   (d) If a file put in Bob's *InBox* is owned by Alice and its permission is "r-- --- ---," can *Bob* remove the file from his *InBox* directory? Why?
   (e) One major weakness of the method is that: (1) anyone may create files in *~Bob/InBox*, and (2) the owner of the link put in *~Bob/InBox* may not be Bob, who feels uncomfortable with this. Please develop a set-user-id program to help them solve these problems. No code is required here. Just describe how your method works in detail.

4. Alice plans to develop a function, which opens a unique temporary file in read/write mode and then returns its file descriptor. In her program, as shown below, *tmpnam*() is invoked to obtain a pointer pointing to a valid filename that doesn't exist at the time it is called. Error checking is ignored.

```
int temp_file() {
    char *fnptr;
    int fd;
    fnptr = tmpnam (NULL);
    fd = open (fnptr, O_CREAT | O_TRUNC | O_RDWR, 0600);
    return fd;
}
```

(a) Specify race condition.

(b) Give an example to explain why the code leads to a race condition.

(c) Propose a method to fix the problem without changing the way to call *tmpnam*() and *open*() in the code. That is, you cannot replace the arguments of the two functions with others. What system calls will be called in your method? Describe how your method works in detail.

5. Double fork is a common solution to the problem: the parent doesn't need to wait for the child to complete and the child doesn't become a zombie.

(a) Give two reasons to explain why UNIX-like systems need zombie processes.

(b) How does the *copy-on-write* technique improve the efficiency of *fork*()?

(c) Please implement function *double_fork*() with the prototype:

```
pid_t double_fork()
```

Your code should follow the rules:

(1) Call both *fork*() and *vfork*(). Each is invoked exactly once. Do not call *fork*() twice.

(2) Return 0 in the child. Return the process ID of the grandchild in the parent.

(3) Use a pipe and unbuffered I/O in blocking mode to avoid a race condition.

(4) Cannot produce any zombie processes.

(5) Close all of the unused file descriptors.

The errors returned from system calls can be ignored.

```
pid_t double_fork()
{




}
```

6. Alice and Bob are writing two programs to play chess in UNIX environment. In this game, their programs must alternately move one piece at a time until one is checkmated. Assume two functions, *thinking*() and *making_a_move*(), are provided already. The function *thinking*() is to evaluate how good further moves would be; *making_a_move*() is just to move one piece. After calling *making_a_move*(), one should inform the other "it's your turn now." One simple solution to process synchronization is using pipe, as shown below, where process Alice calls *write*() to send message 'P' to *Bob* through standard output. *Bob* will be woken up by reading the message from standard input and make the next move.

```
void main(void)
{  // Program Alice
    char buf;
    while (1) {
        thinking();
        making_a_move();
        buf='P';
        write(STDOUT_FILENO,buf,1);
        read(STDIN_FILENO,buf,1);
    }
}
```

```
void main(void)
{  // Program Bob
    char buf;
    while (1) {
        read(STDIN_FILENO,buf,1);
        thinking();
        making_a_move();
        buf='P';
        write(STDOUT_FILENO,buf,1);
    }
}
```

Please use *pipe*(), *fork*(), *dup2*(), *execv*(), and *wait*() to write a program, which forks two processes that can talk to each other with pipes. One process executes program Alice; the other executes program Bob. Your code doesn't need completeness and can ignore error returns. Unused file descriptors should be closed. Zombie processes are not allowed.

Some functions or system calls you might be interested in:

```
int  open(char *path, int oflag);
     oflag: O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_TRUNC,
         O_CREAT, O_EXCL, O_SYNC
int  close(int filedes);            ]
ssize_t  read(int filedes, void *buf, size_t nbytes);
ssize_t  write(int filedes, void *buf, size_t nbytes);
int  dup(int filedes);
int  dup2(int filedes, int filedes2);
int  fsync(int filedes);
int  fcntl(int filedes, int cmd, ... /* arg */ );
     cmd: F_GETLK, F_SETLK, F_SETLKW
     arg: struct flock *flockptr
     struct flock {
            short   l_type;     /* F_RDLCK, F_WRLCK, or F_UNLCK */
            off_t    l_start;   /* offset in bytes, relative to l_whence */
            short   l_whence;   /* SEEK_SET, SEEK_CUR, or SEEK_END */
            off_t    l_len;     /* length, in bytes; 0 means lock to EOF */
            pid_t   l_pid;      /* returned with F_GETLK */
     }
int  unlink(char *pathname);
FILE  *fopen(char *file, char *mode);
FILE  *fdopen(int filedes, char *mode);
int  fclose(FILE *stream);
int  fflush(FILE *stream);
char  *fgets(char *s, int n, FILE *stream);
int  fputs(char *s, FILE *stream);
int  fgetc(FILE *stream);
int  fputc(int c, FILE *stream)
void  *mmap(void *addr, size_t len, int prot, int flags, int filedes, off_t
offset);
mode_t  umask(mode_t cmask);
int  select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout);
pid_t  fork(void);
pid_t  vfork(void);
pid_t  wait(int *status);
pid_t  waitpid(pid_t wpid, int *status, int options);
void  exit(int status);
void  _exit(int status);
int  pipe(int filedes[2]);
int  execlp(const char *file, char *argv0, ..., (char *) 0) ;
int  execv(const char *pathname, char *const argv[]) ;
```