③ ④ ⑤

16 21 11

B07902048 李宥霆

67

Systems Programming (Fall, 2019)
Mid-Term Exam

-- There are 5 questions and 8 pages. Your response time is limited to 2.5 hours.

-- Do not do any IPC (inter-person/inter-phone communication) during the exam.

1. (15 pts) Are the following statements true or false? Explain your answer clearly.

   (a) (3 pts) As each process stores its open file descriptor table in its virtual memory, the open file descriptor table is in user space.

   3

   F，open file descriptor 屬於 kernel space，存的是 process 的 meta data。

   (b) (3 pts) A uni-tasking OS runs a series of processes one by one without manual intervention while a multi-tasking OS allows several processes to run simultaneously. In other words, given a set of programs, the multi-tasking OS needs less time to execute them.

   1

   F，multi-tasking OS 是以 time slice 來達成多工處理，且 multi-tasking 的 OS 所需的時間不會比 uni-tasking 還少

   (c) (3 pts) UNIX-like systems always follow the compatibility rule of exclusive write locks. But sometimes they might not follow the compatibility rule of shared read locks.

   1

   T，當系統上的是 mandatory lock 時，它就不會 follow

   (d) (3 pts) In addition to direct pointers, an i-node also points to an indirect block, which then point to more data blocks. Therefore, sequential access for a large file needs the indirect block to be read from the disk many times.

   3

   F，當要 access large file 時，系統會把整塊 indirect block (4K) load進 buffer cache 裡。

   (e) (3 pts) Temporary files that are opened and then deleted will never be written into a disk.

   3

   F，Temporary file 是在 open後 unlink，因此 ls不到，但仍可以把資料寫進 Disk

2. (10 pts) Comparison.

+0 (a) (2 pts) Compare the difference between ANSI C and POSIX.1 in providing portability of a program.

ANSI C是Unix系統本身就支援的語言，而POSJX.1則有點像外掛的標準，另外include進來的。

(b) (2 pts) Compare the difference between advisory locking and mandatory locking.

+2 advisory：還是可以對上lock的檔案做強制寫入

mandatory：lock住的檔案即無法對其進行修改

+2 (c) (2 pts) Compare the difference between hard links and symbolic links.

hard link：指到同一個 i-node，不能跨 partition

symbolic：開一個捷徑檔，裡面存指向的路徑，不用真的有該被指向的檔案也可以創建

(d) (4 pts) Compare the difference between built-in and external shell commands. Why are *umask* and *cd* built-in commands?

+4 built-in：shell本身看得懂的指令

external：shell fork出child process來執行的指令

Why：需要改變到shell本身的變數，(用) 用external會只改到child pro 的變數。

3. (25 pts) TA Alice writes a <u>set-user-id</u> program *run*, which allows students to execute their programs and write the results to Alice's file *resfile*. For example, student Bob can issue the following command to run his program *assignment*:

```
$ run assignment
```

Here is the code segment of program *run*. Error returns are ignored.

```
int main( int argc, char *argv[] ) {
    if ( access(argv[1], X_OK) < 0 ) return -1; // test for execute permission
    int fd = open( resfile, O_WRONLY | O_APPEND );
    // redirect fd to stdout & then close(fd) here
    execlp( argv[1], argv[1], (char*) 0);
}
```

Suppose that file *resfile* is writable only by Alice. Please answer the following questions.

(a) (4 pts) If program *assignment*'s set-user-ID bit is set, after *execlp*() is called: (1) What's the effective user ID of the process? (2) Can the process still write to the file *resfile*? why ?

+1

(1) ~~Alice~~ Bob

(2) 可以，因為effective UID default是會繼承，因此在原本的effective UID之下，可以對owner是Alice的resfile做寫入動作 ×

(b) (3 pts) Please specify race condition.

+3

當多個process同時進行，而進度不一，造成結果可能與預期有所不同的情況。

而進度先後會

(c) (4 pts) There is a race condition occurring in program *run*. Even though *access*() is invoked to make sure that Bob has the permission to execute *assignment*, it still happens that Bob can use the command to run another program that he has no permission to execute. Why?

+0

因為access跟open不是atomic，只能確保在呼叫access函式當下的權限是對的，而沒辦法確定在之後的程序中權限沒有被更改。

做法：先創一個捷徑檔指向一個有x權限的file，然後在access()執行後open前更改捷徑檔指向的file，這時候想指到誰就可以指到誰而不必在意權限，即可以open別人的檔案 非本題意

(d) (4 pts) Please propose a method to fix the problem mentioned in (c). Briefly describe what system calls you will use in the method. No code is needed here.

+4

方法有二：確保該捷徑檔的內容不能被更改

or 確保不能出現捷徑檔

可以用readlink()把捷徑檔路徑讀出來，與access讀到的路徑比較，(真正路徑)

若不同則return -1，就可以確保中間沒有出現捷徑檔

(e) (4 pts) There is another security hole in program *run*. Bob finds a way to run the command to do what only Alice can do such as deleting Alice's files. Why? And how do you fix the problem?

+2

在執行run program時，effective UID是Alice，而我寫一個program來刪除Alice的file，因為exec()會繼承effective UID，所以可以用Alice的權限exec Bob寫的程式來刪除Alice的檔案

fix?

-6

(f) (6 pts) If Bob issues the following command to delete Alice's file *~Alice/SC/resfile*:

```
$ run ~Bob/SP/assignment1
```

What are minimum access rights for file *resfile* and directories *SP* and *SC*? Only consider the owner or other class. Your answer should be in the form of "rwx", "r-x", or the like.

resfile: Owner: ---

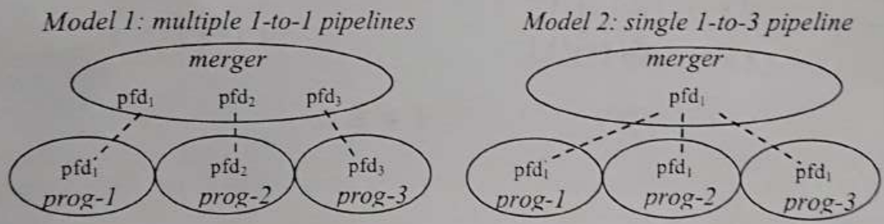
SC: Owner: -wx

SP: Other: --x

4. (32 pts) Alice plans to write a program *merger* that aggregates the outputs from its child processes. When issuing the following command, she wants *merger* to call *fork*() to create three child processes. Each child process calls *exec*() to execute one program, i.e., *prog-i* (*i*=1..3).

```
$ merger prog-1 prog-2 prog-3
```

Each child process is asked to write its output to its standard output. The output will then be sent to *merger* through a pipe. Alice takes two models into account, as shown below, where pfd_j (j=1..3) is the file descriptor used by *pipe*(). Model 1 has three pipelines while Model 2 has only one. The two models should allow the child processes to send data simultaneously and don't generate any zombie processes. The *merger* program, i.e., *merger.c*, is not complete.




```
int main( int argc, char *argv[] )     // merger.c
{
   int i, pid;

   // Section A
   for (i=1; i<argc, i++) {
      // Section B
      pid = fork();
      if ( pid == 0 ) {
         // Section C
         execlp( argv[i], argv[i], (char *) 0 );
      }
      // Section D
   }
   // Section E
}
```

(a) (12 pts) Fill the following form to complete the program for supporting Model 1 and Model 2, respectively. Unused file descriptors should be closed. You could declare your own variables in Section A and put your code in Sections A~E. Errors can be ignored.

+12

	Model 1	Model 2

Section A	int pfd[4][2];
	char buf;
	int cnt = 0;
		int pfd[2];
		char buf;
		pipe(pfd[2]);
		int cnt=0;

Section B	pipe(pfd[i][2]);	

Section C	dup2(pfd[i][1], 1);
	close(pfd[i][0]);
	close(pfd[i][1]);
		dup2(pfd[1], 1);
		close(pfd[0]);
		close(pfd[1]);

Section D	close(pfd[i][1]);	close(pfd[0]); // in section E

Section E	while(1){
	for(i=1; i < argc; i++){
	if(read(pfd[i][0], &buf, size of(char))>0){
	wait(NULL); close(pfd[i][0]);
	cnt++;}}
	if(cnt ≥3) break;}
	exit(0);
		while(1){
		if(read(pfd[0], &buf, sizeof(char))>0){
		wait(NULL);
		cnt++}
		if(cnt ≥3){
		close(pfd[0]); break;}
		}

(b) (4 pts) Suppose program *merger* opens a file for writing (O_WRONLY) in Section A and gets a new file descriptor *fd*. Alice writes a function *append_data*() that allows the child processes to simultaneously "append" data to the file via file descriptor *fd*. Assume that all system calls are atomic and calling *append_data*() is the only way for children to write data to the file. Should function *append_data*() be an atomic operation? Please give an example to verify your answer.

```
ssize_t append_data ( int fd, void *buf, size_t nbytes ) {
    // append nbytes of data from buffer buf to file descriptor fd
    // return the number of bytes written
{
    if ( lseek( fd, 0, SEEK_END ) < 0 ) return -1;
    return ( write( fd, buf, nbytes ) );
}
```

+0

Yes，若沒有atomic，可能會造成race condition.

當process 1在剛完讀寫頭，準備要開始寫時，Process 2正好把內容寫進file，造成檔尾的位置與process 1 lseek的位置不同，process 2寫進去的檔案將被process 1覆蓋

(c) (7 pts) Consider Model 1. Alice makes some changes in *merge.c*, including changing *fork*() into *vfork*(), changing *execlp*() into *_exit*(0), and calling *write*() in Section C to send data via pipe. Please (1) specify deadlock and (2) explain why such changes may lead to a deadlock in detail.

+3 (1) 當多個process共同競爭同一個檔案所造成你等我，我等你的情況
(須有4個條件同時成立)

+2 (2) 當prog1遲遲不寫時，因為vfork要等child process先死的特性，造成其他prog無法運作

(d) (7 pts) Consider Model 2. When the child processes send data to their parent at the same time, there would be intermixing of the outputs from them. (1) Under what condition will such intermixing lead to the problem of race condition? (2) Instead of the use of *fcntl*(), *flock*() or *lockf*(), Alice wants to create a unique file as a lock. If the lock file exists, it means someone is sending data; otherwise, no one is writing to the pipe. What system calls are required by this solution? Explain how and why your solution fixes race condition in detail.

(1) 當要output的內容太大時(超過pipe的buffer)，pipe就無法保證atomic，因此在兩個人同時要寫內容進同一個pipe時就會造成$\underbrace{\text{順序}}_{\text{output}}$可能會不一樣。

+3

(2)
mkfifo();
當有人要寫時，則創建一個fifo檔對他做寫入，而parent對他讀出，這時其他人如果看到存在這個fifo檔時則暫時讓他先寫，等到寫完fifo檔被刪除時其他人再重複上述步驟。

+0

+1 (e) (2 pts) Consider the three I/O models we talked about in class. Please recommend an appropriate I/O model for *merger* in Model 1 and Model 2, respectively.

Model 1: Multiplexing I/O

Model 2: Non-blocking I/O

5. (18 pts) The *cp* utility copies the content of a source file to a target file. The following six factors (A)~(F) 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*()/*write*() or *fgets*()/*fputs*().
   - ✓(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 (used in *read*() and *write*()) and vice versa
   - (D) The time to copy data from kernel's buffer cache to standard I/O library's buffer and vice versa
   - (E) The time to copy data from standard I/O library's buffer to user's buffer (used in *fgets*() and *fputs*()) and vice versa
   - ✓(F) The time to move data from kernel's buffer cache to disk and vice versa

+3 (a) (3 pts) Without special setting, *write*() performs delayed write. What is delayed write?

delayed write：當把所有資料都寫進kernel buffer時就先行return，系統再慢慢把把資料搬進Disk

+2 (b) (3 pts) What factors will significantly affect user CPU, system CPU and response time, respectively?

User CPU time: A E

System CPU time: A C D

Response time: ~~A~~ B ~~C~~ ~~D~~ F

+2 (c) (4 pts) What factor(s) will be significantly affected by system calls *open*() with O_SYNC, *sync*() and *fflush*(), respectively?

open() with O_SYNC: BF

sync(): ~~F~~

fflush(): DE

(d) (4 pts) Consider blocking I/O, non-blocking I/O and multiplexing I/O.

+2 Which model saves the most in user CPU time? blocking I/O

Which model wastes the most in user CPU time? ~~multiplexing~~ I/O

Which model needs longer time addressed in (C)? non-blocking I/O

(e) (4 pts) Give two reasons to explain why

system CPU time + user CPU time ≤ wall clock time

+2 when program *cp* is run.

- cp 會需要搬動 disk 裡的資料到 kernel buffer，這個時間既不算在 user CP
  也不算在 system CPU time 裡
- 但在 cp 時會有 read ahead 跟 delay write 的功能，能夠讓 disk 到 kernel buffe
  的等待時間縮短，減低左式與右式的差距。

```
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, or F_SETLKW
int     unlink(char *pathname);
mode_t  umask(mode_t cmask);
int     select(int nfds, fd_set *readfds,
               fd_set *writefds, fd_set *exceptfds,
               ...ct timeval *timeout);
pi...          ...);
               ...id);
               ...t *status);
               ...pid(pid_t wpid, int *status, intoptions);
               ...(int status);
               _exit(int status);
               pipe(int filedes[2]);
...t    execl(char *pathname, char *argv0, ..., (char *) 0) ;
int     execlp(char *file, char *argv0, ..., (char *) 0) ;

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     fputs(char *s, FILE *stream);
char    *fgets(char *s, int n, FILE *stream);
ssize_t readlink(char* pathname, char * buf, size_t bufsize);
int     sync(void);
int     execv(char *pathname, char *argv[]) ;
off_t   lseek(int filedes, off_t offset, int whence);
                 // whence: SEEK_SET, SEEK_CUR, SEEK_END
```