

IC221: Systems Programming
12-Week Written Exam
[SOLUTIONS]

April 2, 2015

Answer the questions in the spaces provided on the question sheets. If you run out of room for an answer, continue on the back of the page. Show all work, but please be legible.

Microscopic writing will not be graded.

You are allowed a single crib sheet for this exam on one-side of an 8.5"x11" sheet of paper, hand written. You must turn in your crib sheet with your exam.

Name: _____

Section: _____

Alpha: _____

Question	Points	Score
1	20	
2	20	
3	20	
4	20	
5	20	
Total:	100	

1. (a) [4 points] Consider the execution of the following commands, what is the output of jobs?
(*HINT: the argument - for cat says to read from standard input*)

```
#> head -c 10 /dev/urandom > rand
#> sleep 200 &
#> cat - rand &
#> sleep 100
^Z
#> bg
#> jobs
```

```
[1] Running sleep 200
[2] Stopped cat
[3] Running sleep 100
[+2 for stopped and cat, +2 for running sleep]
```

- (b) [2 points] Based on the above commands, what shell command will bring sleep 200 to the foreground?

```
fg 1 or fg %1
[+2 for fg; +1 for indicating job number]
```

- (c) [3 points] What happens to a background process when it attempts to read from standard input? Why does this happen for standard input but not for standard output?

```
The process is sent a signal to stop. Only the foreground process can read from standard input, all process can write to standard output
[+2 for stopped, +1 for explanation, -1 for "blocked" instead of "stopped" there is a technical difference]
```

- (d) [2 points] For how many seconds does the following pipeline run for? **Explain.**

```
sleep 5 | sleep 6 | sleep 7 | cat /proc/cpuinfo | grep processor
```

```
7 seconds because all run in parallel
[+1 for 7 seconds, +1 for parallel]
```

- (e) [3 points] Using the following pipeline of commands and their pid (indicated in italics above each command), what is the pgid of each process in the pipeline? Why?

```
1994      1995      1996      1997
sleep 20 | sleep 30 | sleep 40 | sleep 50
```

```
The pgid is 1994, the pid of the first process in the pipeline.
[+2 for 1994, +1 for explain]
```

- (f) [2 points] For the pipeline from the previous question, will all the processes terminate when a Ctrl-C is used? Why or why not?

```
Yes. The terminal signal is sent to the foreground process group.
[+1 for yes, +1 for explain]
```

- (g) [4 points] For each of the MARKs in the program to the left, what is the most likely process state?

```
int main(){
  if(fork() == 0){
    while(1); //MARK 1
  }else{
    wait(NULL); //MARK 2
  }
}
```

```
Running at Mark 1, busy loop, and Blocked and Waiting at Mark 2, wait is a blocking system call.
[+2 for mark 1 with explanation, +2 for mark 2 with explanation]
```

2. Consider the following program

```
int ticks=0;
void handler(int signum){
    if ( ++ticks > 5){
        raise(9); //<- MARK 1
    }

    alarm(1); //<- MARK 2
}

int main(){

    //MARK 3
    signal(SIGALRM, handler);

    alarm(1);

    //MARK 4
    while(1){
        pause();
        printf("tick tock: %d\n", ticks);
    }
}
```

- (a) [3 points] At MARK 3, what is the purpose of the call to `signal()` with respect to future deliveries of SIGALRM from the O.S.?

Registers a signal handler with the O.S., `handler` is called whenever a SIGALRM is delivered. [+2 Handler is called when SIGALRM is delivered, +1 register with OS (not SIGALRM being called, that's the signal)]

- (b) [4 points] What is the result of the system call at MARK 1? What is the *name* of the signal being raised?

Sends the current process the designated signal, SIGKILL whose number value is 9 [+2 raise description, +2 SIGKILL, -1 if not describing signal being sent]

- (c) [3 points] How many times does tick tock print? Explain.

5 times because 5 SIGALRMs are delivered before the raise [+2 5 SIGALRMs, +1 for explanation]

- (d) [3 points] If the code at MARK 2 were replaced by the following code, how long would it be (in seconds) until next SIGALRM is delivered? **Explain.**

```
alarm(1); //<- MARK 2
alarm(2);
alarm(3);
alarm(2);
```

2 seconds because calls to `alarm()` reset prior calls [+2 for 2 seconds, +1 for explanation]

- (e) [4 points] Consider replacing the code at MARK 4 with the following below. What is the output of the program if the user does not enter a phrase within 1 second? **Explain.**

```
//MARK 4
char phrase[BUF_LEN];
printf("Enter Phrase:\n");
if ( scanf("%s",phrase) < 1 ){
    fprintf("ERROR: scanf\n");
    exit(1);
}

while(1){
    pause();
    printf("%s: %d\n",
           phrase, ticks);
}
```

The program will print ERROR ... and exit because the `scanf()` was interrupted by the system call [+2 for error, +2 for explanation]

- (f) [3 points] To fix the program, the code at MARK 3 was replaced with the following:

```
\\MARK 3
struct sigaction sa;
sa.sa_handler=handler;
sa.sa_flags = SA_RESTART;
sigaction(SIGALRM, &sa, NULL);
```

Explain how this correction fixes the program.

The flag `SA_RESTART` will restart the read system call, avoiding the error. [+2 SA_RESTART, +1 for restarting system call, -1 does not describe "system call" being restarted]

3.

(a) [3 points] For the program to the left, what is the output? **Explain.**

```
int main(){
  int cpid;

  int value = 20;

  cpid = fork();
  if(cpid == 0){
    //Child

    value = 30;
    return 0;
  }else{
    //Parent

    printf("Value:%d",
           value);
  }

  return 0;
}
```

Value: 20
 Value is duplicated but not shared between parent and child [+1 for value, +2 for explanation (should mention no sharing or variables are duplicated, no credit for describing a race condition)]

(b) [2 points] Why does fork() return twice?

Returns once in child and one in parent.
 [+2 for explain, +1 for each return value]

(c) [2 points] What values are returned for each of the returns?

Returns 0 in child, and process id of child in parent.
 ##### HEAD [+1 for each return value — FULL CREDIT DUE TO AMBIGUITY] ===== [+1 for each return value (+1 for say 20 and 30 due to ambiguity in wording, no credit for 0)] ##### 41373e741320db51e3956276cd537648feae54ba

(d) [2 points] For the program to the right, how many total processes (including the initial processes) result from execution? **Explain.**

Returns 4. Initial parent forks twice, one child forks once, totaling 4.
 [+1 for value, +1 for explanation (only -1 for 5 instead of 4)]

```
int main(){
  int i;
  for(i=0;i<2;i++){
    fork();
  }

  while(1);
}
```

(e) [2 points] If the for loop in the program to the right were changed to $i < 4$, how many total process (including the initial process) result from execution? **Explain.**

16, the number of processes doubles in each loop, $2^4 = 16$
 [+1 for value, +1 for explanation (only -1 for 17 instead of 16)]

```

int main(){
    int i,status;

    char * sleep[] = {"sleep",
                      "i",
                      NULL};

    for(i=0;i<2;i++){
        if(fork() == 0){
            //child

            execvp(sleep[0],sleep);
            perror("execvp");
        }

        //parent

        wait(&status);

        sleep(2); //sleep for 2 seconds

        return 0;
    }
}

```

- (f) [3 points] For the program to the left, how long does the program run for? **Explain.**

3 seconds; 1 second for the wait() and 2 seconds for the sleep. [+2 for answer, +1 for explanation (-2 for 4 seconds)]

- (g) [3 points] What is a zombie process? Does the program to the left result in any zombie processes? **Explain.**

Zombie is a process that has terminated but the parent has not waited on it. Yes, this process results in 1 zombie because two forks and 1 wait. [+2 for answer zombie explain, +1 for yes]

- (h) [3 points] What is an orphan process and who “inherits” orphan processes? Does the program to the left produce any orphan processes? **Explain.**

Orphan is a process whose parent has terminated first. They are inherited by the init. Yes. The program does produce a orphan zombie process. [+1 for orphan, +1 for init, and +1 for yes, zombie orphan]

4.

```

int main(int argc, char * argv[]){

    int src,dest;

    //MARK 1
    src = open(argv[1],
               O_RDONLY);

    if ( src < 0){
        fprintf(stderr, "ERROR: open src");
    }

    //MARK 2
    dest = open(argv[2],
               O_WRONLY| O_TRUNC | O_CREAT,
               0666);

    if( dest < 0){
        fprintf(stderr, "ERROR: open dest");
    }

    int n;
    char buf[BUF_SIZE];
    while( (n = read(src, buf, BUF_SIZE)){
        write(dest,buf,n);
    }

    close(src);
    close(dest);
}

```

- (a) **[3 points]** What does the program to the left do with regard to its command line arguments `argv[1]` and `argv[2]`?

It will copy the file specified as `argv[1]` to a new file `argv[2]`
[+3 for copy, -1 describes program mostly but doesn't say copies]

- (b) **[4 points]** For the `open()` at MARK 1 and MARK 2 what does each of the option flags mean?

`O_RDONLY`: read only; `O_WRONLY`: write only;
`O_TRUNC`: truncate; `O_CREAT`: create the file
[+1 for each]

- (c) **[4 points]** At MARK 2, how are the option flags combined? That is, what is the specific operation used and how is it encoded? **Give a small example.**

Combined using bitwise OR or ORing. Encode in each bit, e.g., 1000 — 0100 produce 1100
[+1 for each, -2 doesn't include ORing (or bitwise OR) but does describe process generally]

- (d) **[3 points]** If the `umask` is set to 0037, what will the permission mode for the newly created file be? **Show work or explain.**

$0640 = 0666 \& \sim 0037$
[+2 for right answer, +1 for work, -1 for 0740 if didn't notice 0666 in program]

- (e) **[2 points]** What is the purpose of the `umask` with respect to the security of newly created files?

Ensures that newly created files are not over provisioned with permissions.
[+2 for right answer]

- (f) **[4 points]** Consider again the call to `open()` at MARK 2, write the equivalent `fopen()` line of code to match the options. What is the return type?

`fopen(argv[2], "w");` return type is `FILE *`
[+3 for `fopen()`, +1 for `FILE *`, -1 for including permission settings, e.g., 0666]

5. Consider the program

```

int main(){
    int fd;
    int pfd[2];
    pid_t cpid;

    char * cat[] = {"cat",
                    NULL};

    fd = open("input.txt",
             RD_ONLY);

    //MARK 1
    close(0);
    dup2(fd,0);

    pipe(pfd);

    cpid = fork();
    if( !cpid ){
        //child

        //MARK 2
        close(pfd[0]);
        close(1);
        dup2(pfd[1],1)

        execvp(cat[0],cat);
    }else{
        //parent

        //MARK 3
        close(pfd[1]);

        int n;
        char buf[BUF_SIZE];
        while( (n = read(pfd[0],
                       buf,
                       BUF_SIZE)){
            write(1,buf,n);
        }

        close(pfd[1]);
    }
}

```

- (a) [3 points] What does the `dup2()` system call do? Use the code at MARK 1 in your explanation.

`dup2()` duplicates a file descriptor onto another. Here the `input.txt` file is duplicated onto standard input.
[+2 for `dup2`, +1 for code example]

- (b) [3 points] A pipe is an array of file descriptor, which index is the read end and which is the write end?

`pfd[0]` is read end and `pd[1]` is write
[+1 for one, +3 for both]

- (c) [4 points] At MARK 2 and MARK 3 one end of the pipe is closed. What is this called? Why are alternate ends of the pipe closed in the parent and child?

widowing the pipe. Alternate because close the end not being used.
[+2 for right answer]

- (d) [4 points] From what file or standard file descriptor (e.g. `stdin`, `stdout`, `stderr`) does this program read? And, to what file or standard file descriptor does this program write? **Explain.**

Reads from `input.txt` writes to `stdout.txt` and piped through `cat` [+1 for read, +1 for write, +2 for explain]

- (e) [2 points] For the program to the right, why is it the case that at MARK 3 the while loop will break?

Will read 0 bytes or EOF from the pipe. [+2 for read 0 — FULL CREDIT DUE TO ERROR]

- (f) [2 points] Consider MARK 1: what happens to the child if the kernel buffer for the pipe is less than 4096 bytes in size?

The child will block until any data is read from the pipe. [+2 for block]

- (g) [2 points] Consider removing the `wait()` in the parent at MARK 2: What happens to the parent if it reads from the pipe before the child has written anything?

The parent will block until data has been written to the pipe. [+2 for block — FULL CREDIT DUE TO ERROR IN QUESTION]

```
int main(){
    int i;
    int pfd[2];
    char c;

    pipe(pfd);
    if(fork() == 0){//child
        close(pfd[0]);

        //MARK 1
        for(i=0;i<4096;i++){
            write(pfd[1], 'A', 1);
        }
        close(pfd[1]);
        exit(1);
    }else{ //parent

        //MARK 2
        wait(NULL);

        //MARK 3
        while(read(pfd[1], &c, 1)){
            write(1, &c, 1);
        }
    }
}
```