Exam 1 Feb 16, 2023

COMS 4118 Operating Systems I Jae Woo Lee & Hans J. Montero
Spring 2023 Columbia University

There are 4 problems totaling 99 points + 1 point for taking the exam:

Attendance:	1 points
Problem 1:	30 points
Problem 2:	27 points
Problem 3:	27 points
Problem 4:	15 points

Assume the following programming environment unless noted otherwise:

- Current stable release of Debian GNU/Linux, 64-bit version, running under VMware, with two or more CPUs assigned to the VM.

- All user level programs are compiled with gcc with no optimization.

- All library function calls and system calls are successful when they are invoked correctly. For example, you can assume that fork() will successfully create a child process and malloc() will not return NULL when it is called with a reasonable argument.

- Some of the programs omit #include statements to save space. Assume that all necessary #includes are there.

What to hand in and what to keep:

- At the end of the exam, you will hand in only the answer sheet, which is the last two pages (one sheet printed double-sided) of the exam booklet.

- Make sure you write your name & UNI on both sides of the answer sheet.

- All other pages (i.e., the rest of this exam booklet and any scratch papers you have used during the exam) are yours to keep.

- Before you hand in your answer sheet, please copy down your answers back onto the exam booklet so that you can verify your grade when the solution is published in the mailing list.

- Please be clear and succinct on your answer sheet. If a question asks for a single number or a single word, do not write anything else. If a question asks for a short explanation, keep it short and precise. If you give two different answers, we will take the one that will result in a LOWER grade. If you give a vague explanation that can be interpreted in multiple ways, we will choose the interpretation that will result in the LOWEST possible grade.

```
+--------------------------------------------------------------------+
|  PLEASE DO NOT OPEN THIS EXAM BOOKLET UNTIL YOU ARE TOLD TO DO SO!  |
+--------------------------------------------------------------------+
```

References

```
----------------------------------------------------------------------------

unsigned int alarm(unsigned int sec);

    DESCRIPTION

       alarm() arranges for a SIGALRM signal to be delivered to the calling
       process in sec seconds.

       If sec is zero, any pending alarm is canceled.

       When called, any previously set alarm() is canceled.

    RETURN VALUE

       alarm() returns the number of seconds remaining until the previously
       scheduled alarm was due to be delivered, or zero if there was no
       previously scheduled alarm.

int sem_wait(sem_t *sem);

    RETURN VALUE

       Return 0 on success; on error, the value of the semaphore is left
       unchanged, -1 is returned, and errno is set to indicate the error.
       For example, errno is set to EINTR if the call was interrupted by a
       signal.

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);

    DESCRIPTION

       sigprocmask() is used to fetch and/or change the signal mask of the
       caller. The signal mask is the set of signals whose delivery is
       currently blocked for the caller.
```

Problem [1] (30 points)

Consider the following C program, p0.c:

```
void reader(int fd, pid_t pid)
{
    char buf[5] = {0, 0, 0, 0, 0};

    if (pid) sleep(1);

    char c;
    for (int i = 0; read(fd, &c, 1) == 1; i++) {
        buf[i] = c;
        sleep(2);
    }

    if (pid) waitpid(pid, NULL, 0);

    printf("%s:\t%s\n", pid ? "parent" : "child", buf);
}

int main()
{
    pid_t pid = fork();
    int fd = open("foo.txt", O_RDONLY);

    reader(fd, pid);

    close(fd);
}
```

and the following shell session:

```
$ # create foo.txt with 4 bytes: ’X’, ’D’, ’X’, ’D’

$ echo -n "XDXD" > foo.txt

$ ls -l
total 8
-rw------- 1 hans hans   4 Feb 11 17:56 foo.txt
-rw------- 1 hans hans 552 Feb 11 17:52 p0.c

$ # note that there is no newline in foo.txt

$ gcc -g -Wall p0.c && ./a.out
child:  XDXD
parent: XDXD
```

p0 consistently prints this output.

(1.1)-(1.5) present p1-p5, which are variations of p0. For each program, complete the output lines for child and parent.

If an output line varies from run to run or the program can crash, write "UNPREDICTABLE".

If the child process or parent process doesn't produce additional output after "child:" or "parent:", respectively, write "BLANK". You must write the word "BLANK" to indicate that the respective process didn't produce additional output. Leaving the answer blank is not acceptable.

Notes:
- Error-checking is omitted for brevity.
- Assume p0-p5 are run on a lightly-loaded system.

(1.1) p1's reader() function is the same as p0's.

p1's main() function below switches the order of open() and fork() from p0.

```
int main()
{
    int fd = open("foo.txt", O_RDONLY);
    pid_t pid = fork();

    reader(fd, pid);

    close(fd);
}
```

(1.2) p2's reader() function remains the same as p0's.

p2's main() function below is similar to p1's, but calls open() twice.

```
int main()
{
    int fd1 = open("foo.txt", O_RDONLY);
    int fd2 = open("foo.txt", O_RDONLY);
    pid_t pid = fork();

    reader(pid ? fd1 : fd2, pid);

    close(fd1);
    close(fd2);
}
```

(1.3) p3's reader() function remains the same as p0's.

p3's main() function below changes the second open() from p2 to dup().

```
int main()
{
    int fd1 = open("foo.txt", O_RDONLY);
    int fd2 = dup(fd1);
    pid_t pid = fork();

    reader(pid ? fd1 : fd2, pid);

    close(fd1);
    close(fd2);
}
```

(1.4)

p4, shown in its entirety below, is similar to p0, but is reimplemented using the C standard I/O library.

```
    void reader(FILE *fp, pid_t pid)
    {
        char buf[5] = {0, 0, 0, 0, 0};

        if (pid) sleep(1);

        char c;
        for (int i = 0; fread(&c, 1, 1, fp) == 1; i++) {
            buf[i] = c;
            sleep(2);
        }

        if (pid) waitpid(pid, NULL, 0);

        printf("%s:\t%s\n", pid ? "parent" : "child", buf);
    }

    int main()
    {
        pid_t pid = fork();
        FILE *fp = fopen("foo.txt", "r");

        reader(fp, pid);

        fclose(fp);
    }
```

(1.5) p5’s reader() function is the same as p4’s.

p5’s main() function below switches the order of fopen() and fork() from p4.

```
    int main()
    {
        FILE *fp = fopen("foo.txt", "r");
        pid_t pid = fork();

        reader(fp, pid);

        fclose(fp);
    }
```

Problem [2] (27 points)

Consider the following process hierarchy currently running on the system, taken from the output of the ps command (with appropriate options):

```
    \_ bash
        \_ A
        |   \_ B
        |   |   \_ C
        |   \_ D
        |   |   \_ E
        |   \_ F
        |       \_ G
        \_ H
            \_ I
```

The following kernel function, given a task_struct pointer, prints the name of the task’s parent and the task itself.

```
    void print_tasks_0(struct task_struct *proc) {
            pr_info("%s\n", proc->parent->comm);
            pr_info("%s\n", proc->comm);
    }
```

When invoked with the task_struct pointer of process "I", it generates the following output in the kernel log buffer:

```
    H
    I
```

(2.1)-(2.3) present three variations of print_tasks_0. For each kernel function, write the output generated by the function if we pass in the task_struct pointer of process "D".

If output depends on information not provided, the output varies from run to run, or the kernel can crash or hang indefinitely, write "UNPREDICTABLE".

If a function does not generate any output, write "BLANK". You must write the word "BLANK" to indicate that the respective process didn’t produce additional output. Leaving the answer blank is not acceptable.

Assume that the ps command lists siblings from top to bottom in the order they appear in the linked list of siblings.

(2.1) Write the output when proc points to D’s task_struct.

```
void print_tasks_1(struct task_struct *proc) {
        struct list_head *cur;
        struct task_struct *p;

        cur = proc->parent->children.next;
        while (1) {
                p = container_of(cur, struct task_struct, sibling);
                pr_info("%s\n", p->comm);

                cur = p->children.next;
                if (cur == &p->children)
                        break;
        }
}
```

(2.2) Write the output when proc points to D’s task_struct.

```
void print_tasks_2(struct task_struct *proc) {
        struct list_head *cur;
        struct task_struct *p;

        cur = proc->parent->children.next;
        while (cur != &proc->parent->children) {
                p = container_of(cur, struct task_struct, sibling);
                pr_info("%s\n", p->comm);

                cur = p->sibling.next;
        }
}
```

(2.3) Write the output when proc points to D’s task_struct.

```
void print_tasks_3(struct task_struct *proc) {
        struct list_head *cur;
        struct task_struct *p;

        cur = proc->sibling.next;
        while (cur != &proc->sibling) {
                p = container_of(cur, struct task_struct, sibling);
                pr_info("%s\n", p->comm);

                cur = p->sibling.next;
        }
}
```

Problem [3] (27 points)

Consider sem_wait_with_timeout(), which augments sem_wait() with a limit on the number of seconds that the call should block if the decrement cannot be performed immediately. This function is meant to be a library function included in various UNIX systems.

```
void sig_alrm_noop(int signo) {}

int sem_wait_with_timeout(sem_t *sem, int timeout)
{
    signal(SIGALRM, &sig_alrm_noop);

    alarm(timeout);
    int ret = sem_wait(sem);
    unsigned int remainder = alarm(0);

    if (ret < 0) {
        if (errno == EINTR && remainder == 0) {
            errno = ETIMEDOUT;
        }
        return -1;
    } else {
        return 0;
    }
}
```

sem_wait_with_timeout() is implemented with the following semantics in mind:

- Returns 0 on success and the semaphore’s value is decremented

- Returns -1 on error, and the semaphore’s value is unchanged

    - If the call is interrupted by a signal, errno is set to EINTR

    - If the call times out before the semaphore could be acquired, errno is set to ETIMEDOUT

The library function’s use of signals is not ideal. List three issues with it on the answer sheet.

Hints and requirements:

- Assume that the library function is only used in single-threaded programs.

- Error-checking is omitted for brevity. Assume that the sem and timeout arguments are valid.

- There are more than three possible answers, but please only list three. If you list more than three, we will take the three that will result in the lowest grade.

- Be succinct. Limit each answer to no more than 20 words.

- Each answer must describe a problem in concrete terms. That is, answers of the form "X is bad" or "uses X instead of Y" will receive no credit. You need to explain what about "X" is problematic in the context of sem_wait_with_timeout().

Problem [4] (15 points)

Please write "True" or "False" on the answer sheet for each of the following five statements.

(4.1)

Assume that an open file descriptor was passed from process A to process B using a UNIX domain socket between them. If A closes its file descriptor, the corresponding file descriptor in B will also be closed.

(4.2)

All thread-safe functions are also async-signal-safe functions.

(4.3)

Assume that A and B are related processes. If B wants to block execution until A tells it to continue, A and B can synchronize using an unnamed pipe.

(4.4)

Two unrelated processes can achieve full-duplex data passing by using two FIFOs.

(4.5)

Setting O_NONBLOCK on the write end of a pipe allows the kernel to dynamically extend the internal buffer so that writes always succeed without blocking.

[blank page]

COMS 4118, Spring 2023: Exam 1 Answer Sheet, page 1 of 2

UNI: Name:

[1]

(1.1) child: ______________________________

parent: ______________________________

(1.2) child: ______________________________

parent: ______________________________

(1.3) child: ______________________________

parent: ______________________________

(1.4) child: ______________________________

parent: ______________________________

(1.5) child: ______________________________

parent: ______________________________

[2]

(2.1)

(2.2)

(2.3)

[4]

(4.1)

(4.2)

(4.3)

(4.4)

(4.5)

```
COMS 4118, Spring 2023                     +---+---+---+---+---+---+---+---+
Exam 1                          Your UNI:  |   |   |   |   |   |   |   |   |
Answer Sheet, page 2 of 2                  +---+---+---+---+---+---+---+---+
```

Your Name: ____

front->UNI: ________________

left->UNI: ________________   [You]   right->UNI: ________________

back->UNI: ________________

[3]

(3.1)

(3.2)

(3.3)