

Northern Michigan University (Marquette Co, MI)

CS426-01-26F: Operating Systems (Andrew A. Poe)
Practice Midterm Examination (Exam 1) Page 1/2

Name: _____
Monday 5 October 2026 3:00 P.M. EDT

Time: 50 minutes.

1. Using only low-level system calls, write the following program in C: It executes the program specified in the first argument and passes the string in the second argument as the first argument in the new program. It blocks while the new program is running, and terminates when the child dies. For example, if the program were called `xeq`, then `xeq ./frog 4` would run `./frog` with argument 4.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

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

    int pid = fork();
    if (pid==0) execl (argv[1],argv[1],argv[2],NULL);
    wait (NULL);
    return 0;
}
```

2. In C, write a complete program, using only low-level system calls, that forks a child. The parent opens file called "input.txt" and copies output byte-by-byte to a pipe to the child. The child reads the pipe and copies it into a file it creates called "output.txt". The parent and child should terminate without creating a zombie.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <fcntl.h>

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

    int p2c[2];

    pipe (p2c);
    fork ();
    if (pid==0) {
        close (p2c[1]);
        int out = open ("output.txt",O_WRONLY|O_CREAT|O_TRUNC,0700);
        char buf;
        while (read (p2c[0],&buf,1) > 0) write (out,&buf,1);
        close (p2c[0]);
        close (out);
    } else {
        close (p2c[0]);
        int in = open ("input.txt",O_RDONLY);
        char buf;
```

Northern Michigan University (Marquette Co, MI)

CS426-01-26F: Operating Systems (Andrew A. Poe)
Practice Midterm Examination (Exam 1) Page 2/2

Name: _____
Monday 5 October 2026 3:00 P.M. EDT

```
while (read (in,&buf,1) > 0) write (p2c[1],&buf,1);
close (p2c[1]);
close (in);
wait (NULL);
}
}
```

3. If I have a memory space of 25 bytes, initially clear, what will the results be with this sequence of commands under Best Fit?

```
alloc 15
alloc 10
dealloc 0
alloc 10
dealloc 10
dealloc 15
alloc 10
alloc 10
```

```
alloc 15: returns 0. <0,14> is allocated. <15,24> is free.
alloc 10: returns 15. <0,14><15,24> are allocated. No free space.
dealloc 0: returns 15. <15,24> is allocated. <0,14> is free.
alloc 10: returns 0. <0,9><15,24> are allocated. <10,14> is free.
dealloc 10: ERROR. There is no allocated space beginning at 10.
dealloc 15: returns 10. <0,9> is allocated. <10,24> is free.
alloc 10: returns 10. <0,9><10,19> are allocated. <20,24> is free.
alloc 10: ERROR: There aren't ten available bytes.
```

4. Linux allows both hard links and symbolic (soft links) to data files, but only soft links to directories. What would a very large problem be with allowing hard links to directories? If you wanted to adjust Linux to allow hard links to folders, how would you adjust (perhaps in a slight way) how files and directories are stored to mitigate the very large problem?

The big problem with allowing hard links to directories is that if there is more than one path to a directory, any program that recursively scans directories could go into an infinite loop. If /A/B/C is the same physical directory as /A, then we would scan /A, /A/B, /A/B/C, /A/B/C/B, /A/B/C/B/C, /A/B/C/B/C/B, etc. With a soft link, a program could verify that's the link is soft and not recursively scan that directory. Hard links aren't distinguishable from the original directory and can't be verified.

If you really wanted hard links to directories, one way that would work is that every physical directory could be given a unique id number by the OS. When you are doing a recursive directory scan, you could keep track of the ID numbers you've seen, and not entering a directory you've already seen.