cs154: Introduction to Computer Systems **Homework 9**
Autumn 2019 (Assigned Nov 26)
Due Monday Dec 2nd 11:59pm

Submit your work by committing files in the `hw9` sub-directory of your `CNETID-cs154-aut-19` svn repository. The filename should be either `hw9.txt` or `hw9.pdf` for answers in plain ASCII text, or PDF, respectively. PDFs of scanned hand-written pages must not exceed 6 megabytes. Not following directions will result in losing points.

(1) (16 points)

```
#include "csapp.h"
void echo(int connfd);

void sigchld_handler(int sig)
{
    while (waitpid(-1, 0, WNOHANG) > 0)
        ;
    return;
}

int main(int argc, char **argv)
{
    int listenfd, connfd, port;
    socklen_t clientlen=sizeof(struct sockaddr_in);
    struct sockaddr_in clientaddr;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    port = atoi(argv[1]);

    Signal(SIGCHLD, sigchld_handler);
    listenfd = Open_listenfd(port);
    while (1) {
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        if (Fork() == 0) {
            Close(listenfd); /* Child closes its listening socket */
            echo(connfd);    /* Child services client */
            Close(connfd);   /* Child closes connection with client */
            exit(0);         /* Child exits */
        }
        Close(connfd); /* Parent closes connected socket (important!) */
    }
}
```

This question refers to the code in textbook Figure 12.5 (reproduced above), and the corresponding discussion in lec23-sy1.pptx. `Accept()` and `Close()` are just wrappers around system calls `accept()` and `close()` that perform error checking.

Assume that an open file descriptor structure is 50 bytes in the OS, and that 100 clients have previously connected (i.e. the child processes have terminated). For the two parts below, answer with a non-negative (≥ 0) number of bytes, and a roughly 10-20 word justification.

A. Suppose the "`Close(connfd)`" line has been omitted from the code for the parent process. How large a memory leak has been created by this omission (i.e. how many bytes linger in the OS without being used)?

B. Now assume we fix the bug from the previous part (so the `Close(connfd)` is performed by the parent), but we accidentally delete `Close(listenfd)` in the child process. If another 100 clients connect with this new configuration, how much memory is leaked?

.

(2) (18 points)

```
1  #include "csapp.h"
2  #define N 2
3  void *thread(void *vargp);
4
5  char **ptr;  /* global variable */
6
7  int main()
8  {
9      int i;
10     pthread_t tid;
11     char *msgs[N] = {
12         "Hello from foo",
13         "Hello from bar"
14     };
15
16     ptr = msgs;
17     for (i = 0; i < N; i++)
18         Pthread_create(& tid, NULL, thread, (void *)&i);
19     Pthread_exit(NULL);
20 }
21
22 void *thread(void *vargp)
23 {
24     int myid = *((int*)vargp);
25     int cnt = 0;
26     printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
27     return NULL;
28 }
```

This question refers to the above *modified* version of the code in textbook Figure 12.15. Modifications were made to lines 18, 24, and 25.

Using the analysis from Section 12.4, fill each entry in the following table with "Yes" or "No" for the code above. In the first column, the notation `v.t` denotes an instance of variable `v` residing on the local stack for thread `t`, where `t` is either `m` (main thread), `t0` (peer thread 0), or `t1` (peer thread 1). This is a modification of Practice Problem 12.6.A.

Variable instance	Referenced by main thread?	Referenced by peer thread 0?	Referenced by peer thread 1?
`cnt.t0`	________	________	________
`cnt.t1`	________	________	________
`i.m`	________	________	________
`msgs.m`	________	________	________
`myid.t0`	________	________	________
`myid.t1`	________	________	________

(3) (15 points)

Consider the following use of locks/mutexes in two threads in light of the "Mutex lock ordering rule" from textbook Section 12.7.

```
Initially: a = 1, b = 1, c = 1, d = 1.
Thread 1:          Thread 2:
   P(a);              P(a);
   P(b);              V(a);
   P(c);              P(d);
   V(c);              P(c);
   V(b);              P(b);
   P(d);              V(b);
   V(d);              V(c);
   V(a);              V(d);
```

A. List all pairs of locks/mutexes that Thread 1 can hold, and all pairs of locks/mutexes that Thread 2 can hold.

B. Is there overlap between the sets of pairs of locks/mutexes? If yes, are the locks/mutexes (within the pair) locked in the same order?

C. Given your analysis, is there a potential for deadlock (Yes or No)?