COSC 301: Operating Systems

Lab 2: The process API and more fun with pointers

1. Consider the following program (also in the class git repo, named `labs/lab02/fork01.c`). Compile and run it, then explain its behavior:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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

    char *s = "I am a process!";
    int value = 100;

    pid_t pid = fork();
    if (pid == 0) {
        s = "Or am I?";
        value -= 50;
    } else {
        s = "Who are you?";
        value += 50;
    }

    printf("%s %d\n", s, value);
    return 0;
}
```

2. Write an ill-tempered program that uses the `fork` system call in an effort to make the operating system crash (or run very slowly).

3. The following code (also in the repo as `labs/lab02/fork03.c`) uses the `execv` system call to start and run the `ps` program. Modify the program so that it first forks a child process, which then does the `execv`. The parent process should wait for the child process to finish running (using the `waitpid` system call), print a message like "`Child process finished`", then exit.

The `waitpid` system call takes three parameters: the process ID of the child to wait for, a pointer to an int which is filled with the return value of the child process, and an options bitmask (an int). See the `man` page for more on `waitpid`.

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

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

    char *cmd[] = { "/bin/ps", "-ef", ".", NULL };
```

```
    if (execv(cmd[0], cmd) < 0) {
        fprintf(stderr, "execv failed: %s\n", strerror(errno));
    }

    return 0;
}
```

4. The following code is broken. Explain what is wrong and how to fix the problem:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

char *remove_whitespace(char *s) {
    char buffer[strlen(s) + 1];
    int i = 0, j = 0;
    for ( ; i < strlen(s); i += 1) {
        if (!isspace(s[i])) {
            buffer[j] = s[i];
            j += 1;
        }
    }
    buffer[j] = '\0';
    printf("%s\n", buffer);
    return buffer;
}

int main(int argc, char **argv) {
    char *s = strdup("  the \tinternet\t\nis a series of tubes  ");
    char *newstr = remove_whitespace(s);
    printf("%s\n", newstr);
    return 0;
}
```

5. Inserting an element into a linked list.

Say that you have the following `struct`:

```
struct node {
    char name[128];
    struct node *next;
};
```

Write a function called `insert` that takes a C string and the head of the list and (1) creates a new `struct node` element, and (2) inserts the element at the head of the list. The `insert` function **should not return anything**. The key challenge for the problem is to determine the parameter types for `insert` and how it should be called in order to ensure that new elements are correctly inserted at the head of the list.

You can assume that the list is declared in `main` like:

```
struct node *head = NULL;
```