CMPUT 379 Assignment 1

Name: Bach Wongwandanee
CCID: waridh
SID: 1603722

Part 1

Question 1

1. Assume we have a variable declaration `'PCB_t pcb'` where `PCB_t` defines a C-struct with various members (e.g., integers, character arrays, etc.). Write a C-code fragment that uses the standard C library function `memset` to initialize variable `pcb` to all zeros.

Answer

```
1. PCB_t pcb;
2. memset(&pcb, 0, sizeof(pcb));
3.
```

This C fragment will change every value stored in the struct to 0.

Question 2

2. Often, a program needs to read a sequence of text lines (say, from the `stdin`), and process each line in a certain way. A simple C-fragment to read the lines is:

```
#define MAXLINE  128
#define MAXWORD  20
char buf[MAXLINE];
while(fgets(buf,MAXLINE,stdin) != NULL) { ... }
```

However, `fgets` does not remove a terminating newline character `'\n'` from `buf` (which may be undesirable). Assuming that each line has at most `MAXLINE` − 2 characters, and each line terminates with a newline, write a C-fragment to delete the terminating newline character from `buf`.

Answer

```
1. buf[strcspn(buf, "\n")] = 0;
2.
```

This C fragment will get rid of the terminating newline character from buf.

Question 3

3. In the above question scenarios, after reading a text line in `buf`, one may need to split each input line into tokens, each delimited by a character stored in a string of *field separators*. For example, the string `char WSPACE[]= "\n \t"` may be used to define 3 field separators (the newline, space, and tab characters). So, for example, one may want to split an input string like

`timeout  10  do_work bash.man 7` into 5 tokens separated by one (or more) of the above white space characters.

Explain how to use the standard C library function `strtok` to tokenize an input string, called `inStr`, around separators in `WSPACE`, while keeping track of the number of obtained tokens in variable `count`. Assuming that each token has at most `MAXWORD` − 1 characters, specify a simple data structure for storing the obtained tokens.

Answer

```
 1. // Assignment 1 Part 1 Question 3
 2.
 3. #define MAXLINE 128
 4. #define MAXWORD 20
 5.
 6. // Simple data structure for storing obtained tokens
 7. typedef struct  {
 8.       /*
 9.       The size of the character matrix is based on the max number of words in the
10.       string and the max length of a word. The max number of words is the
11.       character limit per line given in question 2 divided by two, as each word
12.       must have an accompanying delimiter. The other axis size is based on the
13.       max length of characters per word. This is just going to be 20 based on the question
14.       2.
15.       */
16. char toker[MAXLINE/2][MAXWORD];
17.
18. }  token;
19.
20. // Initialize the token type
21. token tok;
22. // Clearing the structure.
23. memset(&tok, 0, sizeof(tok));
24.
25. // Initialize the count
26. unsigned int count = 0;
27. // The delimiters
28. char WSPACE[] = "\n \t";
29.
30. // Creating the output in sigular token.
31. char *buffet = strtok(inStr, WSPACE);
32.
33. while (buffet != NULL)  {
34.       // Using strcpy to move the token into the struct
35.       strcpy(tok.toker[count], buffet);
36.       // Adding to the count
37.       count++;
38.       // Obtaining the next token
39. buffet = strtok(NULL, WSPACE);
40. };
```

```
41.
```

This C fragment shows the initialization of the struct that will contain the tokens and the way to separate the strings into word tokens using the given whitespace characters. This fragment assumes that the inStr variable has been declared already. Here we have a struct for containing the tokens and the variable count, keeping track of the number of tokens that have been read.

Question 4

4. When encountering an error, many Unix system calls return -1 and set the integer `errno` to a value indicating the reason of the error (cf. Section 1.7 of [APUE 3/E]). Give a C-code fragment to print on the `stdout` (rather than the `stderr`) an error message string corresponding to the `errno` value.

Answer

```
1. printf("Error: %s\n", strerror(errno));
2.
```

This C fragment of a single line will output the error message corresponding to whatever the errno is at the moment to stdout.

Part 2

Command-line: “a1p2 w”				
	PID	PPID	State	CMD
1 (a1p2 process)	2750385	2750372	S	./a1p2 w
2 (child process)	2750386	2750385	S	/bin/sh ./myclock out1
3 (grandchild process)	2751369	2750386	S	sleep 2
4 (a1p2 process)	N/A	N/A	N/A	N/A
5 (child process)	2750386	1	S	/bin/sh ./myclock out1
6 (grandchild process)	2763026	2750386	S	sleep 2

Experiment 1 deliverables

1. As seen above, the a1p2 w program will call myclock as a child process. This process will have a1p2 as the PPID, spawning the sleep process every two seconds. We know that sleep belongs to myclock because its PPID is that of myclock. This will make the sleep process a grandchild of the a1p2 process. After the a1p2 process was killed, the child process, myclock, changed its PPID to 1 and keeps running.
2. Every process state in this experiment is “S” for interruptible sleep. This means that they are waiting for an event to complete before continuing. This is likely because the grandchild process sleep is waiting for two seconds to pass, and the child process is waiting for sleep to be done before writing to out1. The parent process a1p2 waits for the child process to end before terminating. After the a1p2 program was killed, the child process continued running and waiting for the sleep process to write to out1.

Command-line: “a1p2 s”				
	PID	PPID	State	CMD
1 (a1p2 process)	2779609	2779608	S	./a1p2 s
2 (child process)	2779610	2779609	S	/bin/sh ./myclock out1
3 (grandchild process)	2779849	2779610	S	sleep 2
4 (a1p2 process)	2779609	2779608	S	./a1p2 s
5 (child process)	2779610	2779609	Z	[myclock] <defunct>
6 (grandchild process)	N/A	N/A	N/A	N/A

Experiment 2 deliverables

1. As seen above, the a1p2 program will call myclock as a child process. This process will have a1p2 as the PPID, spawning the sleep process every two seconds. We know that sleep belongs to myclock because its PPID is that of myclock. This will make the sleep process a grandchild of the a1p2 process. When killing the child process, myclock, the grandchild process stops being recalled and thus will no longer be running. The child process is still there because it is now a zombie process, waiting for its parent to finish its termination.
2. Before killing the child process, the State of all the processes was S for interruptible sleep. This was because they were waiting for the sleep process to finish waiting two seconds. After the kill command was used on the child process, myclock, the sleep process stopped appearing as it is no longer being recalled. The child process is now a defunct “zombie” process. It will stay this way until the parent finishes the two minutes wait and runs again, where it will complete the program termination for the child process.

Part 3

Case 0

‘./a1p3 0 < a1p3.data2’	
Recorded time:	
Real:	0.00 sec
User:	0.00 sec
Sys:	0.00 sec
Child User:	0.00 sec
Child Sys:	0.00 sec

The following table shows the CPU time output for the 0 command line argument in the part 3 program using the custom input that takes at least 5 seconds to run. Both the real and CPU time for both the parent and child process is zero here because the program does not bother to wait for its child processes to finish and moves on. This almost immediately ends the program as the parent program doesn’t take up much CPU time.

Case 1

‘./a1p3 1 < a1p3.data2’	
Recorded time:	
Real:	5.01 sec
User:	0.00 sec
Sys:	0.00 sec
Child User:	0.00 sec
Child Sys:	0.00 sec

The following table shows the CPU time output for the 1 command line argument in the part 3 program using the custom input that takes at least 5 seconds to run. The real time of the process was 5.01 seconds; however, user and sys time for both the parents and child are still at zero. For the parent, the system’s CPU

spent almost no time running the program. As for the child, we are seeing zero because the program that was first to return was myclock, which takes up almost no CPU time when running as it is a light program.

Case -1

‘./a1p3 0 < a1p3.data2’	
Recorded time:	
Real:	5.01 sec
User:	0.00 sec
Sys:	0.00 sec
Child User:	6.18 sec
Child Sys:	8.80 sec

The following table shows the CPU time output for the -1 command line arguments in the part 3 program using the custom input that takes at least 5 seconds to run. The real time was recorded as 5.01 seconds because it took every program 5 seconds to run, but as they are running concurrently, it will only take as long as the longest program to run to completion. The parent’s CPU time is zero because the main program does not take many CPU resources, thus, less CPU time. The child process’s CPU time is significantly higher because three of the input command lines run the example do_work program from eclass set to a high max iteration limit. For the child process, user time is the time the CPU spent executing code in user mode, while sys time is the time the CPU spends in super-user/kernel mode in the process.

Acknowledgements

Operating Systems Concepts 10th Edition and Advanced Programming in the Unix Environment 3rd Edition were vital knowledgebases for completing this assignment. GeeksforGeeks was used as a manual for functions. I would also like to acknowledge James Thompson for answering many of my questions.