Objective 3 Comments

Merrill McKee - TA
Dr. Montagne
Operating Systems
Fall 2007

1 Compare Utility

Many file compare utilities exist and they are extremely useful for anyone working in software. Some of the specific and general reasons for using a file compare utility are listed below:

Comparing Output Files Using "make compare OBJ=3" is the ultimate measure for this class. However, if differences exist, sometimes it is unclear what these differences are and where in the output file they occur. A good utility can quickly let you visually inspect the location of the differences and compare the correct output against your output side-by-side.

Version Control - Working Alone Have you ever made changes that caused your code to crash and been unable to get back to your last stable version? A compare utility will not solve this problem. I recommend saving your source files, data files, and output files at regular intervals and keeping them in their own descriptive folder. I work from a separate working folder rather than any particular version folder. If you introduce a bug into your code, a file compare (actually folder compare) utility will let you visually compare two folders of multiple files. It will show you which files are identical and which are different. You can then go inspect differing files and only focus on areas of code that have changed reducing your debugging time.

Version Control - Group Coding If you are one of multiple people editing the same source code, the issue comes up when both people work on the same code or people run their code using different settings. The file compare utility can help you see these differences. A utility that also merges files, will allow you to quickly take the portions of code from each file that you want to keep. This also helps you visually ensure both team members were running under the same settings in the case where one team member's code is crashing.

1.1 WinMerge

My favorite utility is one a co-worker at Lockheed Martin told me about. It is called WinMerge and is available at www.winmerge.org for free. Others available include Windiff (Microsoft), Fcompare, and Unix's diff command. Countless others exist.

The basic idea is you give it two files (of identical or differing names) or two folders to compare. In the latter case, it compares files of the same name in different folders. There are also settings for whitespace, blank lines, case, and carriage returns in the options menu.

The differences will be highlighted. You can either scroll to the differences or use Alt-Up and Alt-Down to quickly jump to each block of differences. Merging blocks is simple with Alt-Left or Alt-Right. Alt-Left takes the block from the file in the right window pane and copies it to the file

in the left window pane. You can also copy and paste or type in differences smaller than a highlighted block in size.

2 Debugging Utility

I recommend becoming familiar with at least one debugging utility. The three mentioned in class are GDB, Microsoft Visual C++, and Eclipse. The last two are windows based debuggers while GDB is a command line debugger in Unix (or most Unix emulators). Below I list some reasons:

Stepping Through Someone Else's Code If you're given code that someone else wrote and the task to understand it, fix it, or improve it, then you need to understand the code. Since code is not sequential, reading it is sometimes difficult. However, running the code and stepping through it with a debugger can quickly give you a picture of which routines are at the highest levels and which ones are rarely called.

In your case, you've been given simulator.c along with osdefs.h and externs.h and you need to add to it through the objective source files. In a work setting, you are much more likely to edit existing code than to code something from scratch. Knowing how to understand someone else's code is important.

Where Did My Code Crash? Running your program in a debug mode will allow you to see exactly where your program crashed. Not only can you see what line was last executed, but you can also see the call stack and the values of all the local and global variables.

Figuring out why your code crashed is not as simple, but this information can move you in the right direction.

2.1 Debugging Information

A quick disclaimer here. The only thing that matters is how your code runs on Olympus. Learning a new debugger for this project may not be the best form of time management, especially nearing a deadline. However, becoming familiar and skilled with at least one debugger is a useful skill to have. If you're starting early on your objective or completely stuck, then maybe a debugger will come in handy.

I'll mention the following generic debugger terms. These are identical to what you would see in Microsoft Visual C++ and most other debuggers will have terms by identical or similar names.

Step Over This command will step line-by-line through the function (or method) you are currently in. *[VC++ - F10]*

Step Into This command will step line-by-line through the function (or method) you are currently in. However, if the line you are at calls another function, then it will step into this function. You will see your window jump to the new function. *[VC++ - F11]*

New Breakpoint This command will set a breakpoint at the current line. *[VC++ - F9]*

Start Debugging Program This command will run your program until it finishes, crashes, or gets to a breakpoint. *[VC++ - F5]*

Watch Window A watch window will allow you to type variables and see their current values. It will display physical addresses of pointers. Some watch windows will allow you to "open" all of the members of a structure or object to view each individual value.

Call Stack Window When you call a function, this function is placed on the top of the call stack.

Within your own program, the main function is typically at the bottom of the call stack. A recursive program will have a call stack that quickly increases in height. Most non-recursive programs will only have a call stack that goes a few levels higher than the main function.

In debug mode, you can click on any of the functions on the call stack and look at the values of the variables that are visible to this function. This is useful for debugging variable visibility issues. It may answer the question as to what values are successfully being returned through pointers from a function.

The most useful application of the call stack in debug mode is on a crash. When your code crashes in debug mode, you are taken to the last line executed. This function may be called repeatedly and only be crashing on the n-th execution of the function. Viewing the call stack to see the which function calls preceded the current one is useful for gauging where in your overall program execution the crash occurred.

3 Objective 3

As reference, two powerpoint presentations from former semesters have been posted.

For objective 3, just about everything you'll need is written in the comments preceding the functions. Below, I'll add some comments.

3.1 Get_Script

At the beginning of this function, write the two lines given to you under "NOTE:".

Simulator.c opens all the files you'll ever use for any objective. In this case, scriptfp is the name of the file pointer that points to script.dat.

The set of relevant strings is given by pgmidtab. Use this array of strings to determine the integer index. pcb->script[] is array of integer indices into pgmidtab.

This function reads a single script, not the entire script.dat file!

3.2 Logon_Service

For number one, go ahead and set the pcb->user string according to the value of AGENT.

This function is called by simulator.c and starts the ball rolling! It calls Get_script and Next_pgm. How often it is called depends on the logon.dat file which you had to look at for objective 2.

3.3 Next_pgm

3.4 Get_memory

This function along with loader will read a program file. This function is responsible for reading the first half of the program file. It creates the segment table.

After finishing, loader is called (by next_pgm).

3.5 Alloc_seg

This function will allocate a program segment to memory. It won't actually write to memory, but instead will find the appropriate location in memory (MEM) to begin writing.

It will either change the information of one node in the FreeMem linked list or it will delete a node in the FreeMem linked list.

3.6 Loader

This function, along with Get_memory will read a program file. This function is responsible for reading the instructions in the program file with the Get_Instr command. Loader will store the instructions in physical memory (MEM) based on the information in the segment table.

3.7 Dealloc_pgm

Anytime you use a free command, you should follow it by setting the pointer to NULL rather than assuming ”free”ing the pointer will automatically set it to NULL.

3.8 Dealloc_seg

3.9 Merge_seg

E[i] represents a free segment
b[i] represents the physical location of this segment or ”segptr”
len[i] represents the size of this free segment

When using a while loop, you can include a timeout to prevent an infinite loop. Any single pass of the FreeMem linked list can take at most 2*MAXSEGMENTS iterations. In reality it can take less, but all you’re [optionally] doing is including a counter to prevent an infinite loop.

3.10 End_Service

For part two, you’ll figure out that it should be Purge_rb(pcb). For part three, using rblist will give you a compiler error. Instead use firstrb.

3.11 Abend_Service