Objective 3 Directions

Objective 3 Overview

In this objective you will develop functions that handle LOGON, PGMEND, SEGFAULT, and ADRFAULT events and load the programs for each user as specified in `script.dat`. Several functions such as `Display_pgm()` and `Get_Instr()` from previous objectives will be used again in new ways and several Objective 2 specific functions, such as `Boot()`, will no longer be used.

In the `main()` function of `simulator.c` an `InterruptHandler()` (in `simulator.c`) function is called which in turn calls your `Logon_Service()` function in response to LOGON events. `Logon_Service()` will create a Process Control Block (PCB) for each user that logs on, and will read the list of programs that the user is going to execute from `script.dat` and store this information in the PCB. Finally, `Next_pgm()` is called from `Logon_Service()` to advance to the next program in the PCB to execute.

`Next_pgm()` will handle deallocating programs from memory that are done executing and will load the next program into memory. `Next_pgm()` will call `Get_memory()` and based on the information in the PCB, `Get_memory()` will open the correct program file and based upon the SEGMENT statements determine if there is enough free memory to load the requested program. If there is enough memory available `Get_memory()` will called `Alloc_seg()` for each segment in the program file to allocate memory to load the program into. Once the call to `Get_memory()` completes, the `Loader()` function is called which loads the program from the program file into the areas of `MEM` that were allocated by `Alloc_seg()` and that are described in the segment table of the PCB.

PGMEND, SEGFAULT, and ADRFAULT events are also handled in the `Interrupt_Handler()` function and call your `End_Service()` and `Abend_Service()` similarly.

To get an example of what the output should look like run: make correct−output to create an ossim.out file with the correct output.

Important data types and files

PCB − This data type (struct pcb_type) is defined in osdefs.h and has many data members. Most of them are either used by the simulator or are used in later objectives. In Objective 3 you are primarily concerned with the following:

```
char  user[5];  /* User terminal "Uxxx" */
int *script;   /* Pointer to process script */
struct segment_type *segtable; /* segment table ptr*/
unsigned int segtab_len;/* segment table length  */
```

You will initialize several other data members but won't use them much. `user[5]` is simply a string like U001 or U002 just as before. `*script` is an array of integers that will take on the values of the program ids (script program codes) defined in `osdefs.h` as:

```
#define   EDITOR      0
#define   PRINTER     1
#define   COMPILER    2
#define   LINKER      3
#define   USER        4
#define   BOOT        5
#define   LOGOFF      6
```

Programs will be loaded from these script files into `MEM`. The PCB contains a variable named `segtable` which is an array of `segment_type` structs, just like `MEMMAP`, and is of length `segtab_len`.

script.dat − This is simply a text file that contains the programs codes as listed above. When the first user logs on, this file is read until a LOGOFF event reached. As additional users logon, the process is repeated, with the file pointer remaining where it was left by the previous logon actions. The fact that the events are broken up across several lines makes no difference, the important thing to know is that the LOGOFF event is the delimiter between users' scripts. You don't need to worry about running out of program codes, there are enough for all the users that will logon during the simulation.

editor.dat −
compiler.dat
linker.dat
user.dat

These files have the same format as boot.dat, although there are often more than one program stored in a file. The first user to use the editor for example, will load the first program in editor.dat. The second user to use the editor will load the second program, etc. You don't need to worry about running out of program instances in a script file, there will be enough for the simulated users in this objective.

Get_Script

```
void Get_Script(struct pcb_type *pcb)
```

The comments in `obj3.c` *describe this function and what you basically need to do.*

Directions:

1. Initialize the pcb as described in the comments.
2. Read from script.dat and convert the program codes from their string representation to their corresponding integer one (shown above). Store these integers in pcb −> script. After reading and storing a LOGOFF event stop reading from script.dat. script.dat can be accessed through the scriptfp file pointer and does not need to be opened or closed in your code.

3. Print the script to simout as shown in ossim.out

Logon_Service

```
void Logon_Service()
```

This function handles LOGON events.

Directions:

1. Follow the instructions given in `obj3.c` in the comments surrounding `Logon_Service()` function.
2. Make sure to set `pcb -> user` as well, to U00x where x is AGENT.

Next_pgm

```
int Next_pgm(struct pcb_type *pcb)
```

This function makes the transition to the next program in the script for (pcb), if there is one.

Directions:

1. Follow the instructions given in `obj3.c` in the comments surrounding the `Next_pgm()` function.
2. The format for the banner message is shown in `ossim.out`.

Get_memory

```
void Get_memory(struct pcb_type *pcb)
```

This function allocates space for a new program by reading a segment table from disk.

Directions:

1. Follow the instructions given in `obj2.c` in the comments surrounding `XPGM()` function.
2. You don't need to worry about opening or closing any of the PROGM_FILE[] files, that is taken care of for you.
3. Remember to skip the blank lines between segments in the script files

Alloc_seg

```
int Alloc_seg(int len)
```

This function searches the list of free memory segments for a segment of at least length `len` *and returns the address of this free segment. If the request can't be satisfied, return −1;*

Directions:

1. Follow the instructions given in `obj3.c` in the comments surrounding the Alloc_seg`()` function.

Loader

```
void Loader(struct pcb_type *pcb)
```

This function reads a program from a program file into memory.

Directions:

1. Follow the instructions given in `obj3.c` in the comments surrounding `Loader()` function.
2. Remember to skip the blank lines between segments in the script files
3. Be sure to output which program was loaded and for what process as shown in `ossim.out`.

Dealloc_pgm

```
void Dealloc_pgm(struct pcb_type *pcb)
```

This function frees all allocated segments for the current program.

Directions:

1. Follow the instructions given in `obj3.c` in the comments surrounding `Dealloc_pgm()` function

Dealloc_seg

```
void Dealloc_seg(int base, int len)
```

This function returns a segment at base of length `len` *to the free list, and then merges any continuous free segments.*

Directions:

1. Follow the instructions given in `obj3.c` in the comments surrounding `Dealloc_seg()` function

Merge_seg

```
void Merge_seg()
```

This function scans FreeMem and joins contiguous blocks of free memory.

Directions:

1. Follow the instructions given in `obj3.c` in the comments surrounding `Merge_seg()` function.

End_Service

```
void End_Service()
```

This function services PGMEND events.

Directions:

1. Follow the instructions given in `obj3.c` in the comments surrounding `End_Service()` function.
2. Be sure to output note as in `ossim.out` that a program ended for a user.

Abend_Service

```
void Abend_Service()
```

This function services SEGFAULT and ADRFAULT events

Directions:

1. Follow the instructions given in `obj3.c` in the comments surrounding `Abend_Service()` function.