Lab 6: Advanced C (ECSE Continuation 3)

1. Preliminary Notes

- We skipped lab 5 (ECSE Continuation 2) due to scheduling issues
- It was requested by students who have not used C in a long time that a review of that language should be given at the beginning of this course. It is, however, assumed that students have taken a C or C++ course in the past.
- **This lab is optional.** You do not need to attend the lab if you are comfortable with C.

2. Victim Structures

With this lab we will explore basic memory mapping and linear victim selection techniques. This will be a helpful exploration of concepts to prepare you for assignment 3. This is not a replacement of assignment 3, as it asks you to build your software in a particular way. This lab is helpful in how it allows you to both practice C, data structures, and explore process victimization.

Assume you are given the following data structures:

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

int ram[100];

int maxPCB;

struct PCB {
  int pid;
  int startAddress;
  int length;
} pcbList[20];
```

Three standard C libraries are included. Specifically stdio.h for printf() and time.h for its random number generator, which we will look at soon.

Two data structures representing "RAM-ownership" and a list of running PCBs (this is not the ready-list). To make this lab easy, it was decided to keep everything in arrays. For example, the PCB list would be better suited as a linked list in real applications.

The ram[] array is a **membership** data structure. When a process is loaded into RAM for the first time its source code is copied into RAM. We will use the data structure ram[] as the method by which the OS keeps track where a process was saved in RAM. Note I am using RAM to mean the physical memory and ram[] to indicate the data structure that tracks which process is in RAM and where it is in RAM. Assume that the index number of the array ram[] symbolize the physical address of RAM. Assigning the process ID number to a cell of the ram[] array indicates that code from that process is currently in RAM at that address. For example: assume PID=5, length of code = 3, and the process was loaded at address 10 in RAM, then cells 10, 11, 12 are assigned the number 5.

The struct PCB is an array that records important information for each running process. Specifically, it stores the PID, start-address in RAM where the program was initially loaded, and the number of instructions in the program (length). As we said, this data structure would be better as a linked-list of nodes, but for simplicity we are using an array of struct.

Assume you are given the following main() program:

```
int main() {
     int i, length;
     time_t t;

     // Part 1 - initialize ram[]

     for(i=0;i<100;i++) ram[i]=-1;
     maxPCB = 0;

     // Part 2 - populate ram with 5 processes

     srand((unsigned) time(&t));

     for(i=0; i<5; i++) {
             length = rand() % 20;
             addProcess(i,length); // addProcess(pid, size)
     }

     // Part 3 - print ram and pcb list (verify)

     printRamPcb();

     // Part 4 - generate a random new process

     i=6;
     length = rand() % 20;
     printf("***New Process: pid: %d, length: %d\n", i, length);

     // Part 5 - select a victim and overwrite (if space) otherwise add

     if (victim(i,length)) printf("Victim found\n");
     else printf("Victim not found. Added to end.");

     // Part 6 - check ram and pcb list (verify)

     printRamPcb();
}
```

The above main() program is divided into 6 parts. Each part is identified by a comment.

Part one initializes the ram[] array to -1 indicating that no process has ownership of the cells of the array. The number of running PCBs (maxPCB) is initialized to zero to indicate that nothing is running.

Part two uses a random number generator to create 5 processes. The PID for each process starts at 0 and increments to 4. The random number generator determines the length of each process. It then calls the function addProcess() to find space in ram[] for that program, and to add the process to the pcbList[i]. The function addProcess() does a linear search from the beginning of the array ram[] to find the first available cell that can fit the size of the program. If there is no room in the array then the process is not loaded and a -1 is saved in the start address of the PCB for that process (yes, it still creates a PCB for that process).

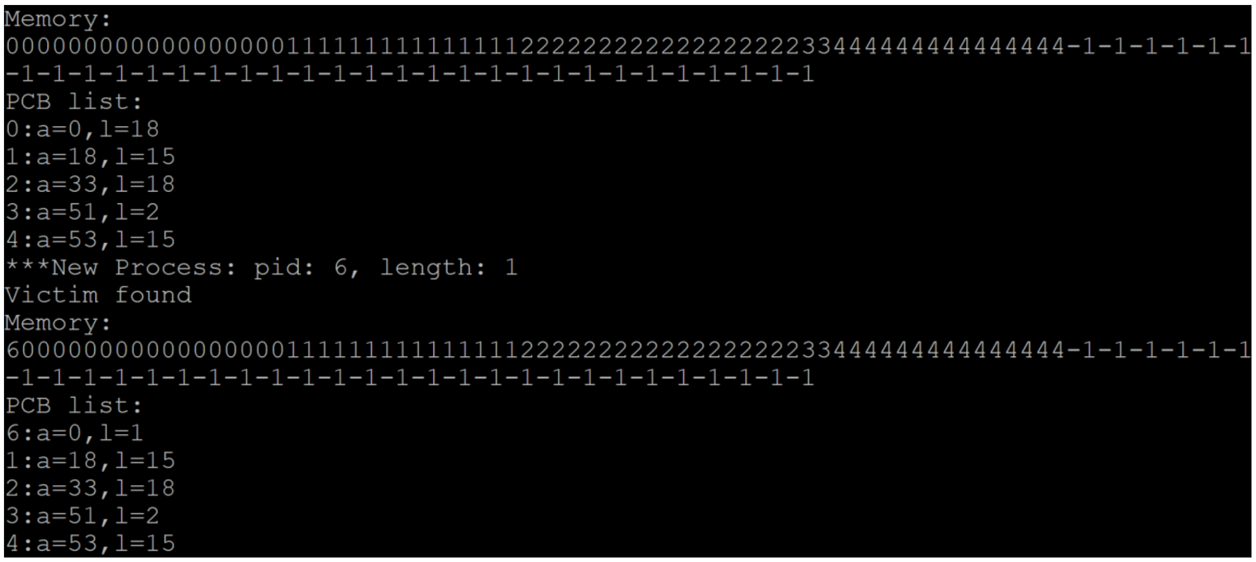
Part three prints the contents of ram[] and pcbList[] arrays. This is done to verify that parts one and two completed as expected.

Part 4 uses the random number generator to create an additional process with the PID set to 6.

Part 5 calls the victim() function to find a victim from the initial 5 processes. The function scans the pcbList[] array from the beginning looking for the first PCB that has a length greater than or equal to the new PID 6. If one is found, it overwrites the old PCB values and ram[] values with PID 6's information, otherwise it calls addProcess() to add PID 6 to the end of the pcbList[] and ram[] arrays.

Part 6 ends by printing the ram[] and pcbList[] array to see if it works.

The output looks like this:



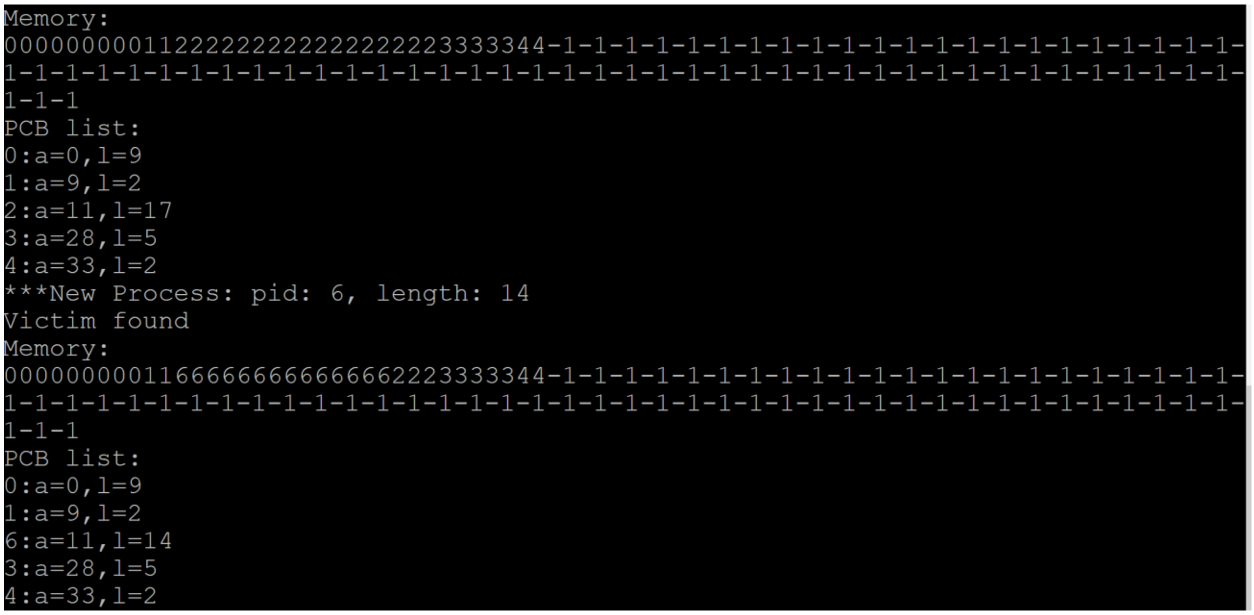
```
Memory:
00000000000000000011111111111111122222222222222222233444444444444444-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
PCB list:
0:a=0,l=18
1:a=18,l=15
2:a=33,l=18
3:a=51,l=2
4:a=53,l=15
***New Process: pid: 6, length: 1
Victim found
Memory:
60000000000000000011111111111111122222222222222222233444444444444444-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
PCB list:
6:a=0,l=1
1:a=18,l=15
2:a=33,l=18
3:a=51,l=2
4:a=53,l=15
```

Notice how it runs:

- The user begins by typing ./a.out to run the program.
- The first printout from part 3 outputs ram[] showing the cells filled with zeros, ones, twos, threes and fours. The numbers printed correspond to the length of each program.
- The PCB list is also outputted. As you can see process 0 output as 0:a=0,l=10, meaning: display the process ID number, the start address (a=0) and the length of the program (l=18). Notice

there are 18 zeros starting from cell 0 of ram[]. Verify that this is true for all the other processes.
- Then a new process length is generated for PID number 6.
- We are notified if a victim is found or if one was not found. In the above example we found a victim since PID 6 was so small (length = 1).
- The program ends printing out ram[] and pcbList[].
- Notice that cell 0 of ram[] has one 6, since the length of PID 6 is 1. Notice that the remaining zeros were not overwritten!! This is a normal strategy employed by operating systems. Why do you think?

Here is another run:



```
Memory:
00000000011222222222222222223333344-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-
1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-
1-1-1
PCB list:
0:a=0,l=9
1:a=9,l=2
2:a=11,l=17
3:a=28,l=5
4:a=33,l=2
***New Process: pid: 6, length: 14
Victim found
Memory:
00000000011666666666666662223333344-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-
1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-
1-1-1
PCB list:
0:a=0,l=9
1:a=9,l=2
6:a=11,l=14
3:a=28,l=5
4:a=33,l=2
```

This time the new process (PID 6) length is 14 and it found the victim to be process PID 2. Notice in the second printout of ram[] and pcbList[], that process 2 has been replaced by process 6 in ram[] and in the pcbList[].

Note how there is still some of 2 remaining in ram[]. This is true in real operating systems. This is for speed. This is also why there is a carrier in forensics.

For this lab, write the functions: addProcess(), victim() and printRamPcb().