CSE 511 Project 1 Report

Thread Migrator System

Name- Niramay Vaidya
PSU ID- 939687597

Table of Contents

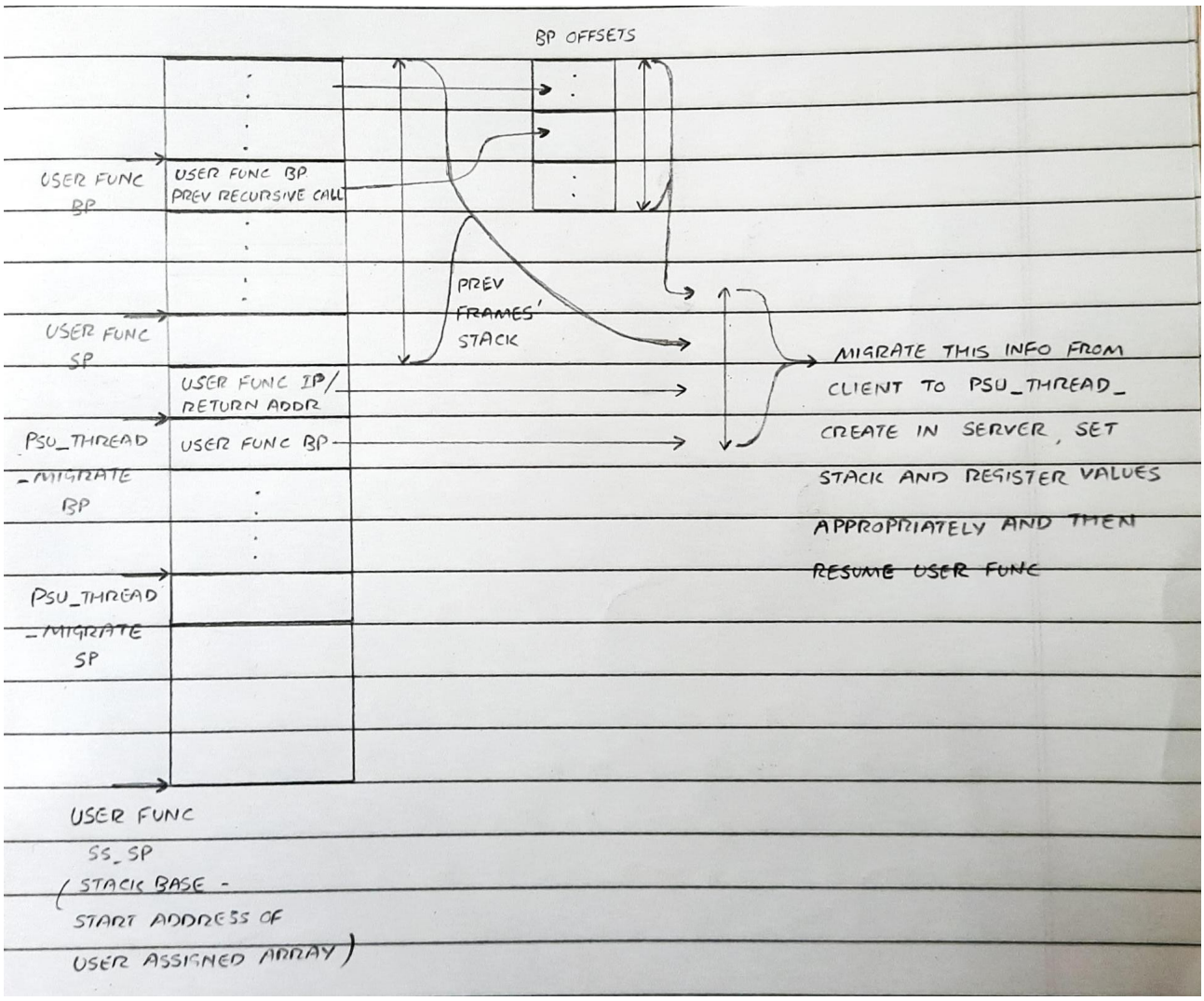
1. Overall logic of the implementation
2. Difficulties faced and how they were overcome
3. Special cases that the code does not work for
4. Extra work done beyond the requirements of the project
5. Distribution of work between the team members

1. Overall logic of the implementation-

Client- In psu_thread_create, the context for the user func is created by getting it using getcontext, then setting the user defined stack for it, then assigning the user func to its context via makecontext and then finally switching to its context using swapcontext. The crux of the thread migration code at the client side lies within psu_thread_migrate where the required values for the previous frame's base pointer, the return address immediately after psu_thread_migrate's return into its caller i.e. the user func (either within the 1st call or within a nested or inner recursive call), the stack to be migrated by determining previous frame's stack pointer (which includes all the frames prior to the one for psu_thread_migrate), and then the base pointer offsets stored within the stack to be migrated (stored base pointers of previous frames) are found out and sent over to the server (the communication model used with the server is a request reply one, which involves ACKs/NACKs for every piece of data received by the server from the client). Once the transfer has been done to the server, the client exits immediately. Refer to the image below for more details on what and how information is determined and sent over to the server by the client.

Server- Majority of the server's code is in psu_thread_create. The server receives the required data from the client one by one, ACKs/NACKs accordingly. The context for the user func has already been gotten before this and the user defined stack has been set, and the main's context has also been linked to this context. Makecontext is then done to assign the user func to its context. The obtained register values are then set appropriately, along with populating the current stack with the received stack and base pointer offsets properly. Finally, the user function is resumed from the correct location by doing swapcontext. Psu_thread_migrate has no code on the server side since the user func is anyways resumed at the point immediately after psu_thead_migrate returns in it. Refer to the image below for more details.

Diagrammatic representation-



Apart from this, there are a few other components implemented in the code which act as helper functions and are fairly straightforward, along with certain #defines to control the print statements.

2. Difficulties faced and how they were overcome-

The primary difficulty was precisely figuring out how the stack is managed, and in what order does the data get stored on to the stack i.e. what sequence is followed when pushing the arguments, if any, register values (base pointer and program counter or instruction pointer), and then the local variables during a function call and after it, when the function starts executing. It also took some effort to figure out how the stack pointer exactly moves. In addition to this, understanding of how the stack frames look when there are multiple function calls, either nested but different, or recursive, was also required.
Which portion of the stack needs to be transferred from the client to the server also needed some prior analysis, in terms of how the stack pointer moves on the test machines (since there are two ways that can be followed, either increment first and then push the value, where the stack pointer then points to the topmost value on the stack, or push the value first and then increment, where the stack pointer then points to the first empty slot on the stack).
Finally, it took some time to determine where the previous frame's base pointer, instruction pointer and stack pointer would lie with respect to the current frame's base pointer.

3. Special cases that the code does not work for-

Even though the case where the stack base i.e. the start address of the user defined stack (the array) across two different machines may change has been handled by transferring the offsets of the base pointers stored in the portion of the stack to be migrated and then calculating the corresponding absolute addresses with respect to the server's stack base, instead of storing the absolute addresses of the base pointers (plural meaning the base pointer stored on to the stack before every nested or recursive function call) present at the client side, the same case for the base addresses of the functions defined has not been handled. Offsets for return addresses or instruction pointer values stored on the stack have not been calculated with respect to their respective functions' base addresses. Instead, absolute return addresses have been sent from the client to the server (as part of the migrated stack). As a result, in case the base addresses of the same functions defined at the client and server differ from each other, the current implementation will run into a problem (fortunately, this has not been the case when running the client and the server on two different W135 lab machines).

4. Extra work done beyond the requirements of the project-

There was a thought of registering a signal handler for the SIGSEGV signal in order to detect a stack overflow error caused because of the user assigned stack's size not being sufficient enough, in which case this size could have been extended from within the handler via realloc, had the initial stack been malloced (in the current implementation, this isn't the case, an array has been used for the stack). Once this is done, some way would need to have been devised in order to re-execute the instruction that actually caused the stack overflow (segmentation fault) to occur. In this case, all the previous stack's addresses stored within the stack itself i.e. the base pointer values would need to be updated according to the base address of the reallocated stack (in this case, the offsets of the base pointer values would have to be stored before reallocation and then they would have to be reflected in their correct positions within the stack by calculating the new absolute addresses from them). Though, all of this has just been given a rough thought, and has not reflected in the actual implementation, so I'm assuming it does not really count as any extra work done beyond the requirements of the project.
I am not even sure whether creating extra test cases to check robustness of the code counts as extra work in any way or not.

5. Distribution of work between the team members-

Though initially, me and Aman Pandey had planned to work together as a team, we both ended up coding the entire implementation separately. Apart from minimal collaboration pertaining to some aspects of the logic to begin with, our GitHub repositories are different where both of us only have commits in our own repositories. The primary idea of implementing the thread migration in itself is different in both our codes, the major difference in this aspect being where the user function resumes at the server side. In my implementation, I resume it immediately after the psu_thread_migrate function returns, whereas in his code, he resumes it within psu_thread_migrate itself.
Apart from this main logic (even in the main logic, other than the major difference mentioned above, rest of the implementation also has almost no resemblance), our codes also starkly differ in their structures i.e. distribution of the code in multiple files, socket programming constructs, how the input values are taken in and processed (hostname, port and mode), the thread info struct, error handling, supporting functions, print levels (debug/info/warn/error), git components (.gitignore and README.md), the Makefile, and the additional custom test app files (there is a

high chance that I might have missed something in this list, but which is also different when both are codes are compared).