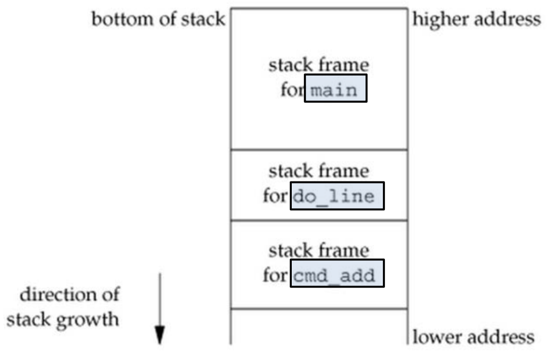
This summary was made by Dr. Dai, William Devlin and Dr. Urgaonkar, to help student digest the slides.

The slides first briefly reviewed address space layout of a program written as a single process comprising multiple user level threads. The following is a brief summary.

1) The h/w context and stack for each thread would be stored in heap or data segments of the process address space. The process heap is preferred, so that a varying number of threads could be dynamically allocated in the process address space.
2) In addition to their stacks, there would be a need to separately store the h/w context and some other scheduling related information for each thread. For this, there would be a thread control block (TCB) per thread kept in the process heap.
3) User-level threads library would be part of the code segment of the process.

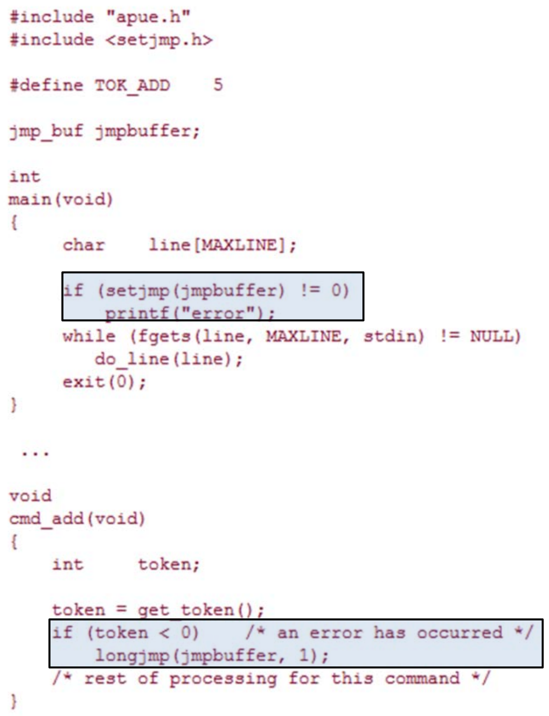
Then, the slides continued the description of user-level context switching using setjmp() and longjmp(). The function prototypes were given in slides, as well as Example 1. Here, we follow the slides to Example 2: Deeply-nested function calls. The example goes this way:

1) There are three functions: main, do_line, and cmd_add, which are called in a nested fashion. At a point when we are inside the function cmd_add, the stack layout is as follows:



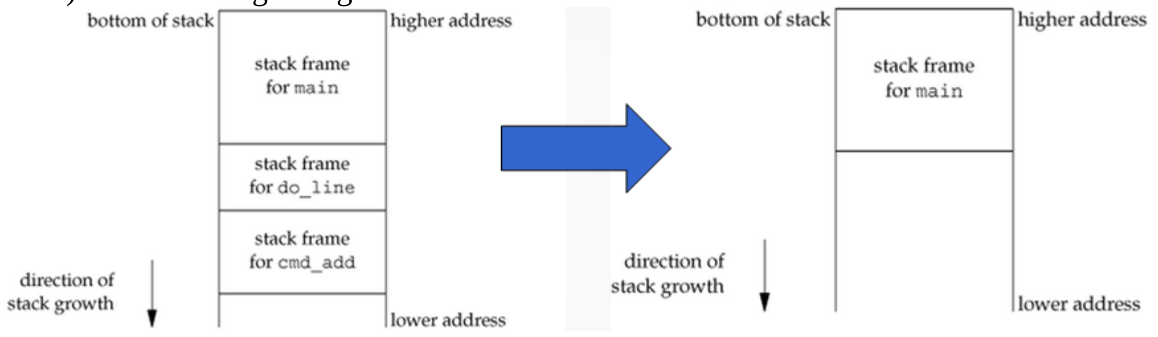
2) What if when cmd_add encountered a non-fatal error, we would like to ignore the rest of the line, and return directly to main? The following two solutions would be infeasible or undesirable because:
    a. goto would only return within cmd_add;
    b. using special return value would involve going up the entire set of nested calls.
3) Therefore, we expect a non-local goto facility. Luckily, setjmp() and longjmp(), as standard library routines, provide this facility. Specifically, they are useful for handling error conditions in deeply-nested function calls. Let's see the setjmp() and longjmp()-implemented error handling code added to the above Example 2, illustrated in the following figure.
4) The error handling using setjmp() and longjmp() goes this way:

a. setjmp() records whatever information it needs to in jmpbuffer and returns 0;
b. Upon an error, longjmp() causes the stack to be “unwound” back to the main function, throwing away the stack frames for cmd_add and do_line;
c. Calling longjmp() causes the setjmp() in main to return with a value of 1.



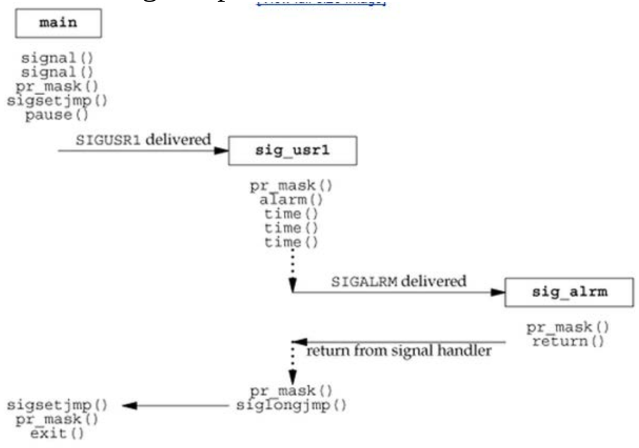
```
#include "apue.h"
#include <setjmp.h>

#define TOK_ADD    5

jmp_buf jmpbuffer;

int
main(void)
{
     char    line[MAXLINE];

     if (setjmp(jmpbuffer) != 0)
         printf("error");
     while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
     exit(0);
}

 ...

void
cmd_add(void)
{
    int     token;

    token = get_token();
    if (token < 0)    /* an error has occurred */
        longjmp(jmpbuffer, 1);
    /* rest of processing for this command */
}
```

5) The resulting change of the stack frame is shown below:



Next the slides introduced some shortcomings of the longjmp function. While longjmp does help us restore the processes h/w context to that specified in the jmpbuffer that is passed as its first argument, there is one aspect of the processes environment that longjmp does not restore on some systems (like linux): the signal mask. The signal mask is an array of bits stored in each PCB which indicate whether a signal should or should not be delivered to the process’s signal handlers. (In certain scenarios, a process may not want a signal delivered to it if it is already handling a signal of that type). Because of this limited functionality, we were introduced to a new pair of functions, sigsetjmp() and siglongjmp(). These functions, while quite similar to setjmp() and longjmp(), have one key difference. These allow the processes signal mask to be restored when the siglongjmp function is called within a processes signal handler.

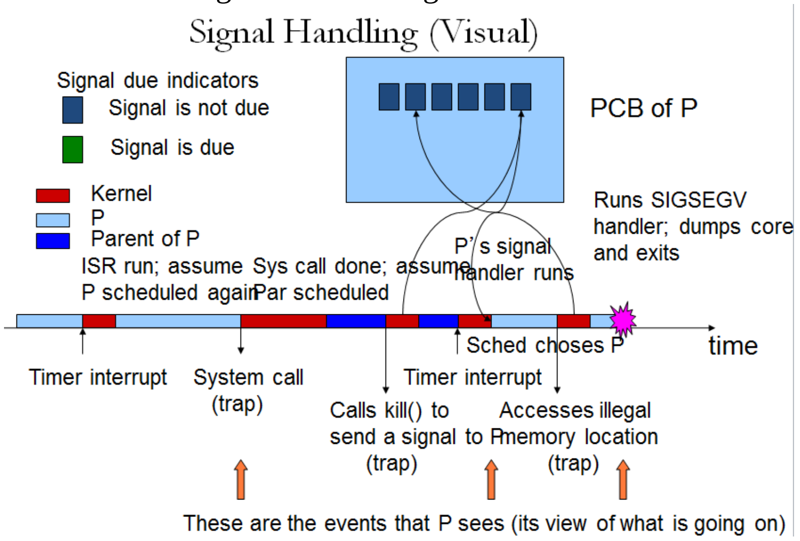
We then saw an example of the sigsetjmp/siglongjmp functions in use in the program snippet shown in slides. You can see the custom signal handler for SIGUSR1 calling the siglongjmp() function right inside the signal handler as discussed earlier. Without the siglongjmp function the program would not be able to restore its signal mask. In the next slide you can see the output of the program. First you can see the program being run from a separate terminal with "./a.out &" then from the original terminal the user sends the "kill" system call to send the SIGUSR1 signal to the process (using the processes process ID). Below you can see the scope of the process as it passes through different functions and handles the signals pushed by the OS. In the program you can see that the "signal" system call was used to handle signals of a specified type in a new way. Just like overloading a function, a new function can be passed to the signal system call giving the OS a new handler for the signal specified.



A Detailed Look at How the OS Helps Processes with Signal Delivery and Handling:
Recall that processes send signals to each other using the "kill" system call. When this system call is made, only the sending and the OS are aware of the existence of this signal. As part of the system call, the OS records the information about this signal in the recipient process's PCB. The instant the OS chooses the process with the pending signal to run next, the OS, instead of preforming all of the usual context switching, takes the following actions to make the process aware of this signal and help its signal handler run before normal resumption. The OS makes a special change to the program counter for the process to point it to the beginning of the signal handler (i.e., the process's code meant to handle the matching signal). Not only that, but the OS also temporarily saves the registers AND the stack that the process was using so the signal handler for the process does not overwrite any information needed by the process. Though it's running separate code, the signal handler is NOT a separate thread of the process. It is simply a separate function (or collection of functions) that need to execute before resuming normal execution of the process.

We then saw the example on slide 27giving us a picture to follow for the signal handling procedure. This diagram also shows us what happens after the function meant to handle the system call is finished executing. Because the process needs its original stack and registers to continue executing, it must give control back to the OS to allow it to restore its original context from before the signal. This is done by the process sending the system call sys_sigreturn(). This function tells the OS to undo its psudo-context switch, and to do so, the OS removes the signal flag from the process's PCB (to indicate that this signal has now been handled so the handler does not run again) and restores its original stack.

Now that we know how signals are requested and handled at both the user and kernel level, we can see it all come together in the diagram on slide 28.


Signal Handling (Visual)
Signal due indicators
Signal is not due
Signal is due
PCB of P
Kernel
P
Parent of P
Runs SIGSEGV handler; dumps core and exits
P's signal handler runs
ISR run; assume P scheduled again
Sys call done; assume Par scheduled
Sched choses P
time
Timer interrupt
System call (trap)
Timer interrupt
Calls kill() to send a signal to P (trap)
Accesses illegal memory location (trap)
These are the events that P sees (its view of what is going on)

Based on the elaborate explanations from the slides, user-level thread implementation can be summarized as follows:

1) Use "alarm" facility provided by the OS to ensure the user-level scheduling/context switching code gets to run from time to time;
2) Implement this code as part of (or called from within) the signal handler;
3) Define your own thread control block data structure to maintain the hardware context and scheduling state for each thread;
4) Use sigsetjmp() and siglongjmp() for saving/restoring hardware context and for context switching in the thread chosen by your CPU scheduler.