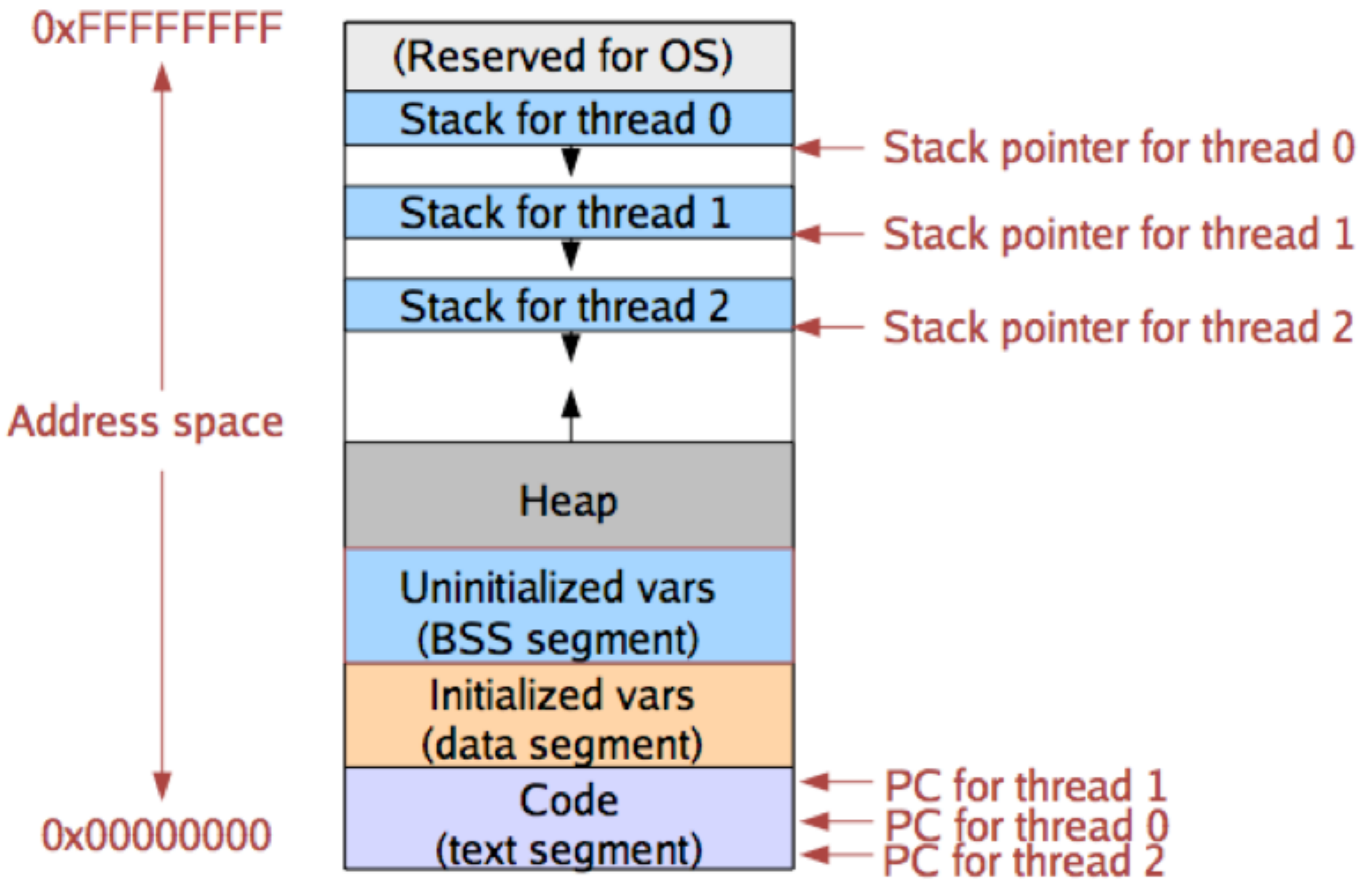
Operating Systems Principles

CPU Management

User-level Threads

Slides are adapted from Prof. Bhuvan Urgaonkar

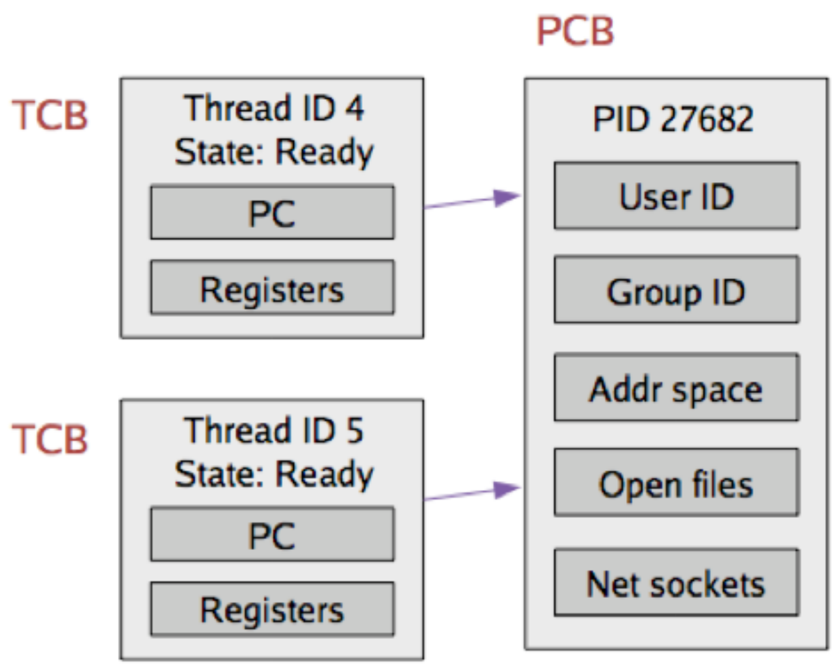
(New) Address Space with Threads



- All threads in a process share the same address space

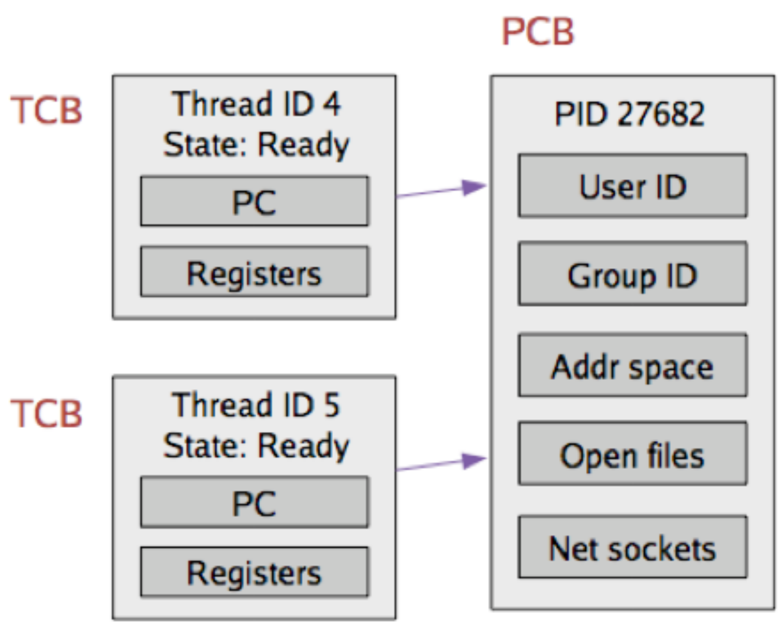
Implementing Threads

- Given what we know about processes, implementing threads is "easy"
- Idea: Break the PCB into two pieces:
  - Thread-specific stuff: Processor state
  - Process-specific state: Address space and OS resources (e.g., open files)



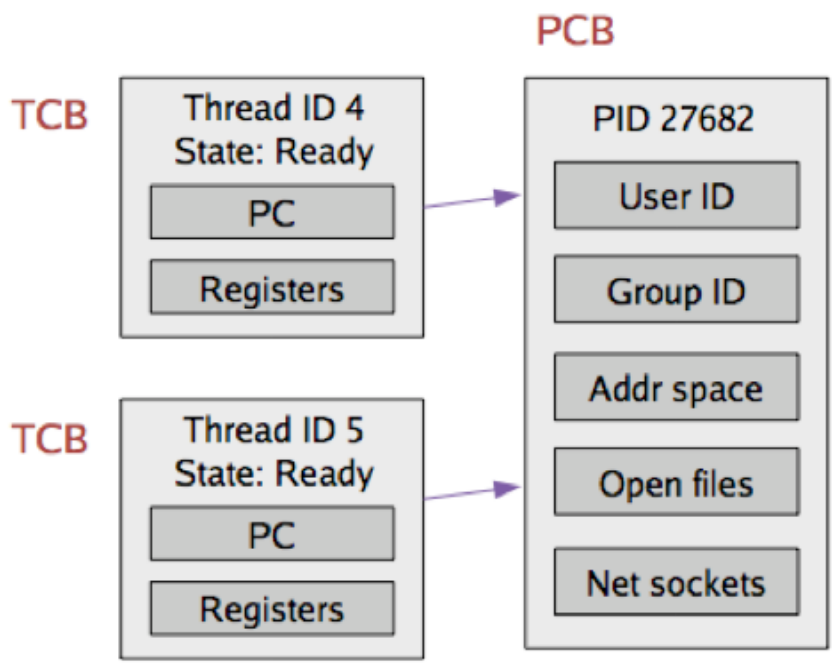
Thread Control Block (TCB)

- TCB contains info on a single thread
  - Thread id
  - Scheduling state
  - H/W context (registers)
  - A pointer to corresponding PCB
- PCB contains info on the containing process
  - Address space and OS resources, but NO processor state!



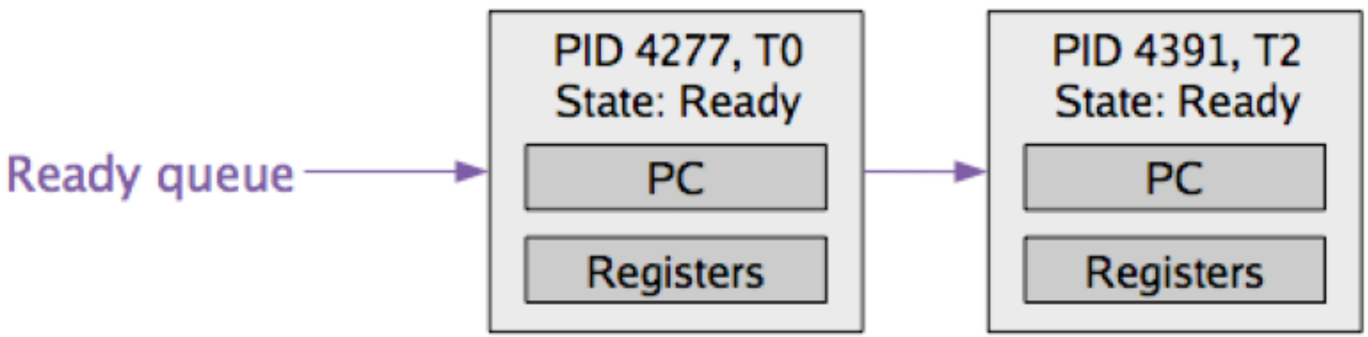
Thread Control Block (TCB)

- TCBs are smaller and cheaper than PCBs
  - E.g., For some recent version of Linux:
    - Linux TCB (thread_struct) has 24 fields
    - Linux PCB (task_struct) has 106 fields



Context Switching

- TCB is now the unit of a context switch
  - Ready queue, wait queues, etc. now contain pointers to TCBs
  - Context switch causes CPU state to be copied to/from the TCB



- Context switch between two threads of the same process
  - No need to change address space
    - No TLB flush
- Context switch between two threads of different processes
  - Must change address space, possibly causing cache and TLB pollution

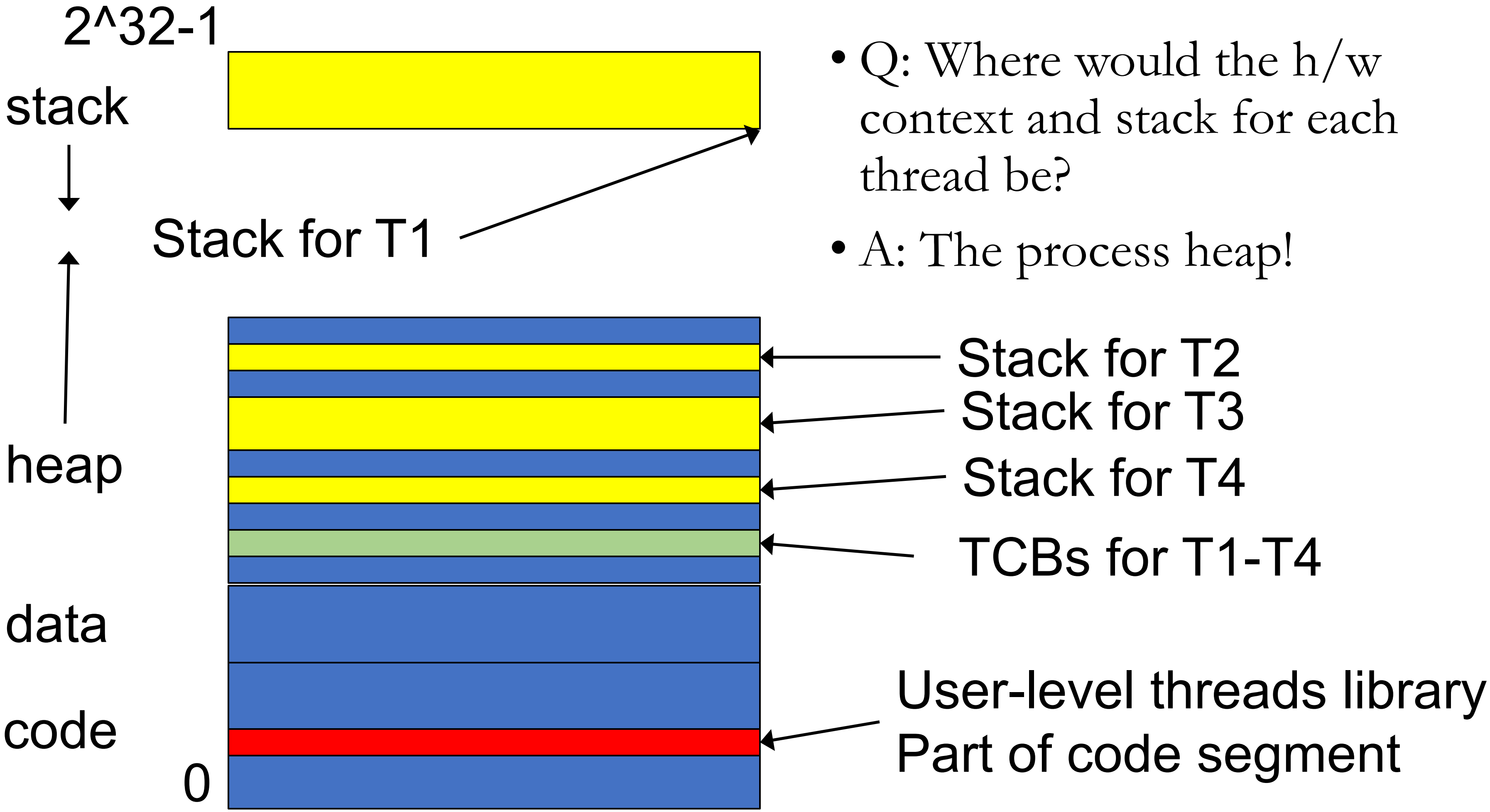
Implementing User-level Threads

- Alternate to kernel-level threads
  - Implement all thread functions as a user-level library
    - E.g., libpthread.a
  - OS thinks the process has a single thread
    - Use the same PCB structure as when we studied processes
  - OS need not know anything about multiple threads in a process!

Implementing User-level Threads

- It should be clear that we would need the following:
  - #1: Scheduling and context switching code as part of process code
    - E.g., a library that we link against our process
  - #2: Room to store hardware context and stack for each thread in process's own address space

Examples: #1, #2



- Q: Where would the h/w context and stack for each thread be?
- A: The process heap!

Implementing User-level Threads

- It should be clear that we would need the following:
  - ~~#1: Scheduling and context switching code as part of process code~~
    - ~~E.g., a library that we link against our process~~
  - ~~#2: Room to store hardware context and stack for each thread in process's own address space~~
  - #3: Facility for this code to intervene execution of threads from time to time and run itself (analogous to timer interrupt)
  - #4: Ability to save/restore hardware context while remaining in user space
    - This includes switching PC to address for thread being restored

#3: SIGALRM signal

- #3: Facility for this code to intervene execution of threads from time to time and run itself (analogous to timer interrupt)
  - Request the OS to send periodic "alarm" signals to the process (SIGALRM)
  - Implement a signal handler for SIGALRM (part of our code)
  - Whenever the OS context switches this process in, if there is a signal pending, this handler would run before resuming execution
  - **This is our opportunity to run our scheduler/context switching code and pick a thread to run!**

```
#include <setjmp.h>
#include <signal.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>

bool gotit = false;

void timer_handler(int sig)
{
  int ret_val;
  gotit = true;
  printf("Timer expired\n");
}
```

```
int main()
{
   signal(SIGVTALRM, timer_handler);

   struct itimerval tv;
   tv.it_value.tv_sec = 2; //time of first timer
   tv.it_value.tv_usec = 0; //time of first timer
   tv.it_interval.tv_sec = 2; //time of all timers but the first
   tv.it_interval.tv_usec = 0; //time of all timers but the first

   setitimer(ITIMER_VIRTUAL, &tv, NULL);
   for(; ;) {
      if (gotit) {
         printf("Got it!\n");
         gotit = false;
      }
   }
   return 0;
}
```

#4: User-level context switching

- How to switch between user-level threads?
- Need some way to swap CPU state
- Fortunately, this does not require any privileged instructions
  - So the threads library can use the same instructions as the OS to save or load the CPU state into the TCB
- Why is it safe to let the user switch the CPU state?
- How does the user-level scheduler get control?

setjmp() and longjmp()

- In C, we can't use the goto keyword to change execution to code outside the current funtion
- setjmp() and longjmp() are C standard library routines that allow this
- Useful for handling error conditions in deeply-nested function calls
- Lets understand them first and then see how they can help realize user-level threads

setjmp() and longjmp()

- int setjmp (jmp_buf env);
  - Save current CPU state in the "jmp_buf" structure
- void longjmp (jmp_buf env, int retval);
  - Restore CPU state from "jmp_buf" structure, causing corresponding setjmp() call to return with return value "retval"
  - Note: setjmp returns twice!
- struct jmp_buf { … }
  - Contains CPU specific fields for saving registers, PC.

Example 1: Basic Usage

```
int main(int argc, void *argv) {
 int i, restored = 0;
  jmp_buf saved;

  for (i = 0; i < 10; i++) {
    printf("Value of i is now %d\n", i);
    if (i == 5) {
      printf("OK, saving state...\n");
      if (setjmp(saved) == 0) {
        printf("Saved CPU state and breaking from loop.\n");
        break;
      } else {
        printf("Restored CPU state, continuing where we saved\n");
        restored = 1;
      }
    }
  }
  if (!restored) longjmp(saved, 1);
}
```

Example 1: Basic Usage

```
Value of i is now 0
Value of i is now 1
Value of i is now 2
Value of i is now 3
Value of i is now 4
Value of i is now 5
OK, saving state...
Saved CPU state and breaking from loop.
Restored CPU state, continuing where we saved
Value of i is now 6
Value of i is now 7
Value of i is now 8
Value of i is now 9
```

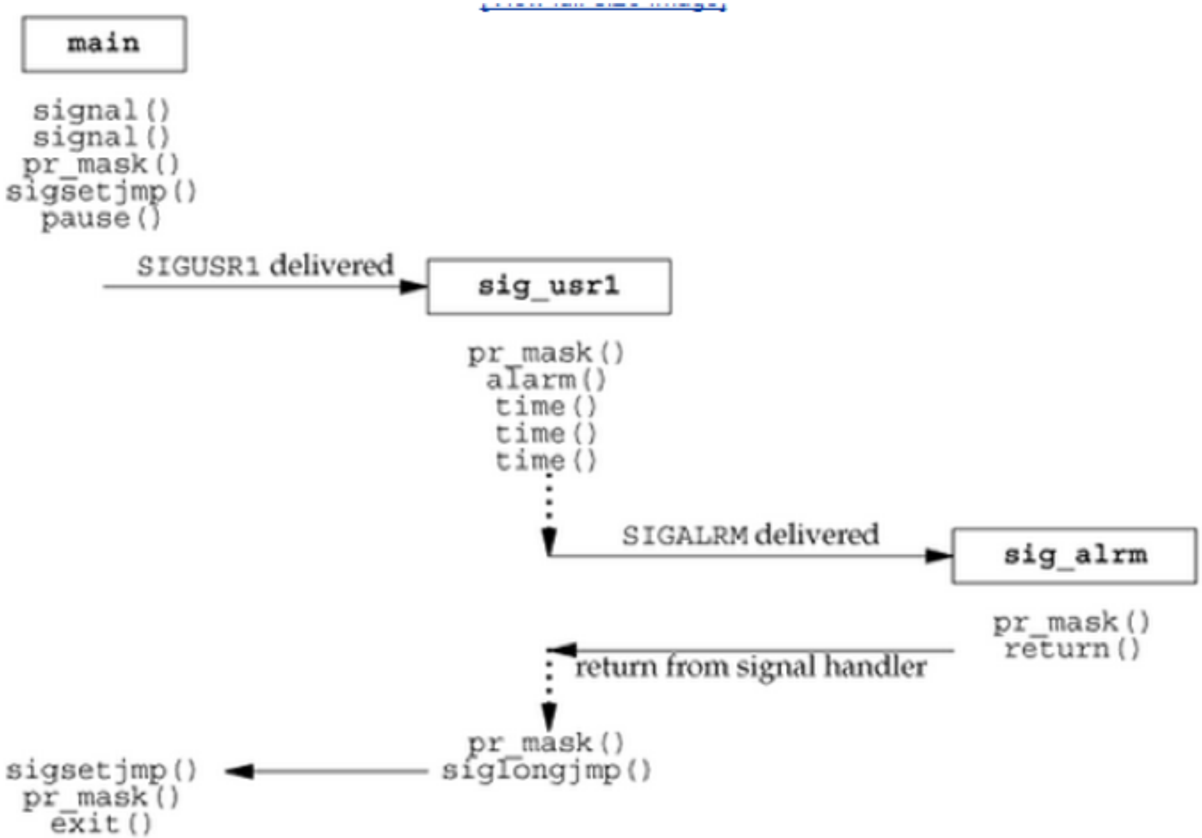
sigsetjmp() and siglongjmp()

- A problem with longjmp:
  - when a signal is caught the signal handler is entered with the current signal added to the signal mask for the process
  - i.e., Subsequent occurrences of the same signal will not interrupt the signal handler
  - Some OSes do not save/restore the mask when longjmp is called from a signal handler (e.g., Linux)
- sigsetjmp and siglongjmp allow the signal mask for the process to be restored when siglongjmp is called from a signal handler

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

static void                         sig_usr1(int), sig_alrm(int);
static sigjmp_buf                   jmpbuf;
static volatile sig_atomic_t        canjump;

int
main(void)
{
    if (signal(SIGUSR1, sig_usr1) == SIG_ERR)
        err_sys("signal(SIGUSR1) error");
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");
    pr_mask("starting main: ");     /* Figure 10.14 */

    if (sigsetjmp(jmpbuf, 1)) {
        pr_mask("ending main: ");
        exit(0);
    }
    canjump = 1;         /* now sigsetjmp() is OK */

    for ( ; ; )
        pause();
}
static void
sig_usr1(int signo)
{
    time_t  starttime;

    if (canjump == 0)
        return;     /* unexpected signal, ignore */

    pr_mask("starting sig_usr1: ");
    alarm(3);               /* SIGALRM in 3 seconds */
    starttime = time(NULL);
    for ( ; ; )             /* busy wait for 5 seconds */
        if (time(NULL) > starttime + 5)
            break;
    pr_mask("finishing sig_usr1: ");

    canjump = 0;
    siglongjmp(jmpbuf, 1);  /* jump back to main, don't return */
}

static void
sig_alrm(int signo)
{
    pr_mask("in sig_alrm: ");
}
```

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

void
pr_mask(const char *str)
{
    sigset_t    sigset;
    int         errno_save;

    errno_save = errno;     /* we can be called by signal handlers */
    if (sigprocmask(0, NULL, &sigset) < 0)
        err_sys("sigprocmask error");

    printf("%s", str);
    if (sigismember(&sigset, SIGINT))   printf("SIGINT ");
    if (sigismember(&sigset, SIGQUIT))  printf("SIGQUIT ");
    if (sigismember(&sigset, SIGUSR1))  printf("SIGUSR1 ");
    if (sigismember(&sigset, SIGALRM))  printf("SIGALRM ");

    /* remaining signals can go here */

    printf("\n");
    errno = errno_save;
}
```

```
$ ./a.out &                          start process in background
starting main:
[1]   531                            the job-control shell prints its process ID
$ kill -USR1 531                     send the process SIGUSR1
starting sig_usr1: SIGUSR1
$ in sig_alrm: SIGUSR1 SIGALRM
finishing sig_usr1: SIGUSR1
ending main:
                                     just press RETURN
[1] + Done          ./a.out &
```



Process vs K thread vs U thread

- Context switch
  - Compare direct overheads
    - Saving/restoring registers, executing the scheduler
  - Compare indirect overheads
    - TLB flush, Cache pollution
    - Kernel/User mode transitions
- Memory needs
  - Compare
    - User space memory
    - Kernel memory
- Parallelism and scheduling
  - What happens upon blocking I/O

Quiz

- Q1: Recall user-level threads. Why is it safe to let a user switch the CPU? (Hint: what abstraction that we have studied ensures this safety and who enforces it?)

- Q2: Process scheduling is usually described as selection from a queue of processes. For each of these scheduling methods, describe the selection method, the insertion method (if required), further properties of the queue (if required), and whether a simple queue should be replaced by a more sophisticated data structure.
  - round-robin time-sharing
  - shortest-job-first
  - multilevel feedback methods in general

- Q3: So far we have said that all registers including the PC need to be saved when context switching out a process. However, one can typically get away without saving the PC (meaning it gets saved in some other way). Where do you think it gets saved? Who (what entity) saves it?

Quiz

- Q4: Which of process/K-level thread/U-level thread would you pick to design an application whose constituent activities
  - Are mostly CPU-bound
  - Are mostly CPU-bound and need to meet timeliness guarantees
  - Do a lot of blocking I/O

- Q5:

A process on an average runs for time T before blocking for I/O. A context switch takes time S which is effectively a waste (overhead). For round robin scheduling with quantum Q (S is not included in Q), give a formula for CPU efficiency for each of the following:

i. $Q = \text{inf}$

ii. $Q > T$

iii. $S < Q < T$

iv. $Q = S$

v. Q is nearly 0