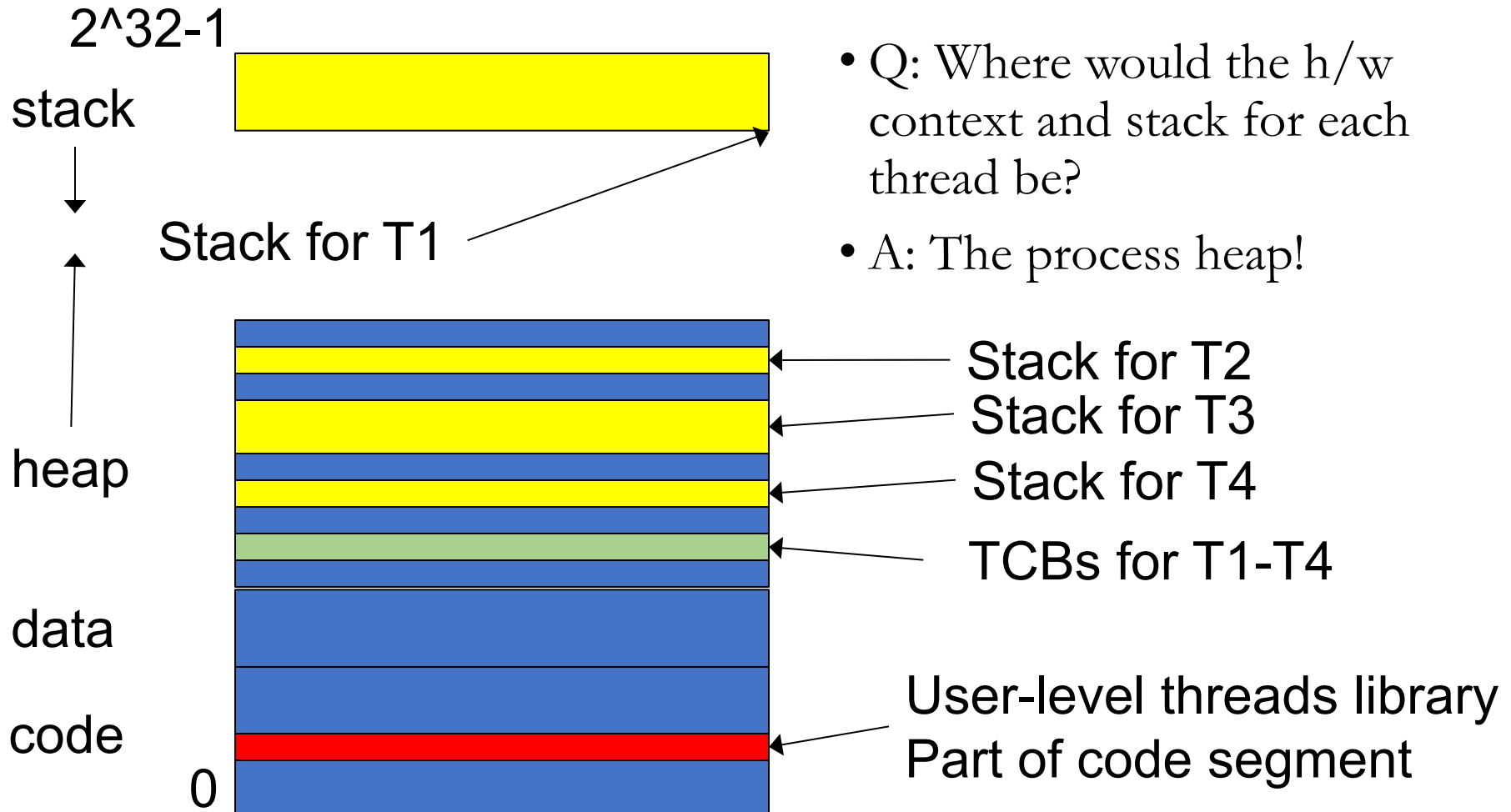


Operating Systems Principles

CPU Management

User-level Threads (2), CPU Scheduling (3)

Examples: #1, #2



• Q: Where would the h/w context and stack for each thread be?

• A: The process heap!

#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 function
- 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;

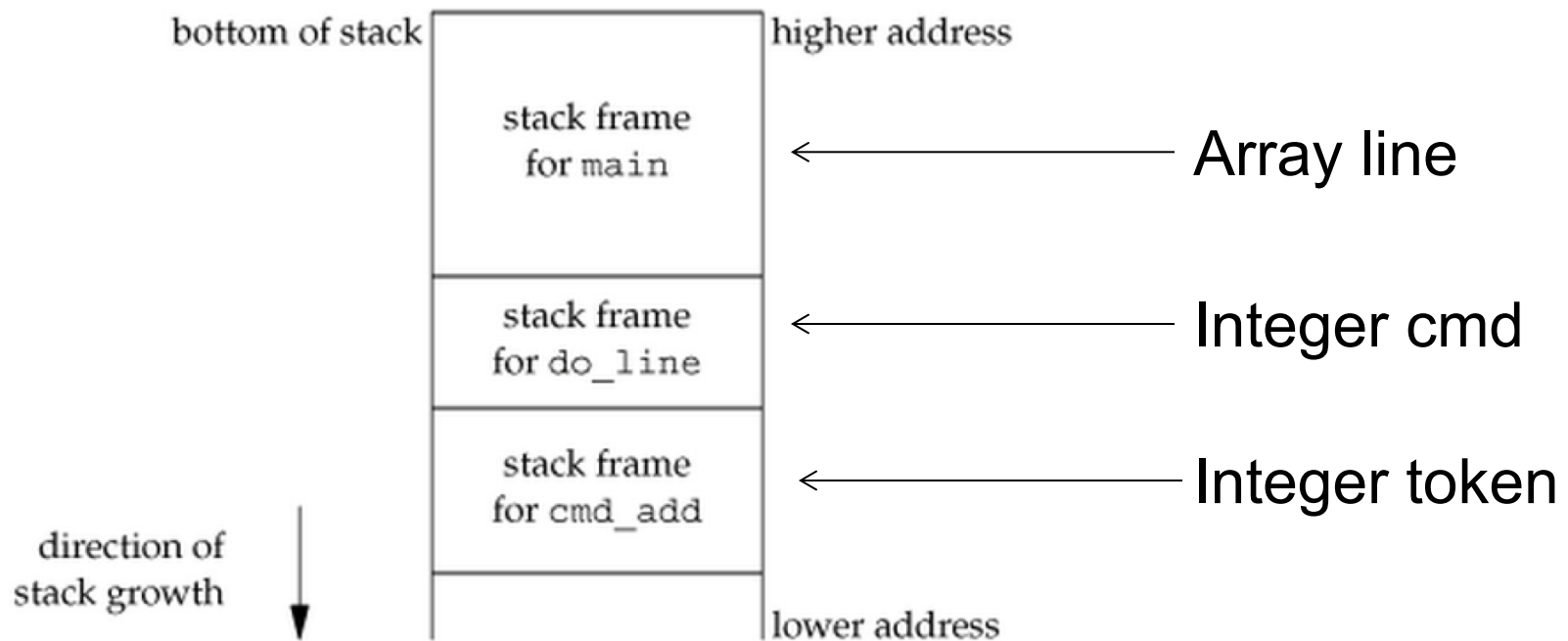
    for (i = 0; i < 10; i++) {
        printf("Value of i is now %d\n", i);
        if (i == 5) {
            printf("OK, saving state...\n");
            if (setjmp(saved) == 0) {
                printf("Saved CPU state and breaking from loop.\n");
                break;
            } else {
          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

Example 1: Basic Usage

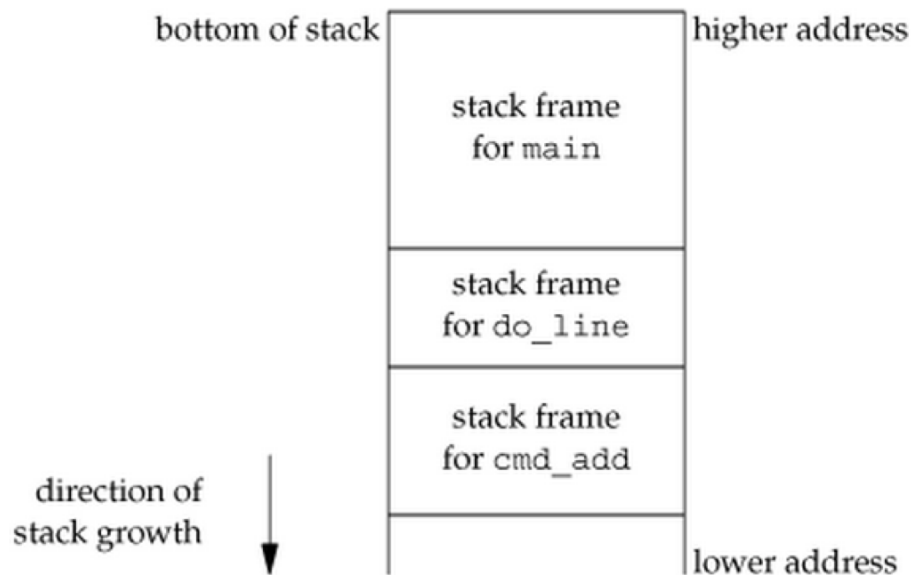
```
Value of i is now 0  
Value of i is now 1  
Value of i is now 2  
Value of i is now 3  
Value of i is now 4  
Value of i is now 5  
OK, saving state...  
Saved CPU state and breaking from loop.  
Restored CPU state, continuing where we saved  
Value of i is now 6  
Value of i is now 7  
Value of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

Example 2: Deeply-nested function calls

- This is what the stack may look like after `cmd_add()` has been called
 - Where are various local variables?



- What if when `cmd_add` encountered a non-fatal error, we would like to ignore the rest of the line and return to main?
 - `goto` would only return within `cmd_add`
 - Using special return value would involve going up the entire set of nested calls!
 - Would like a non-local goto: `setjmp/longjmp` provide this



```

#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);
}

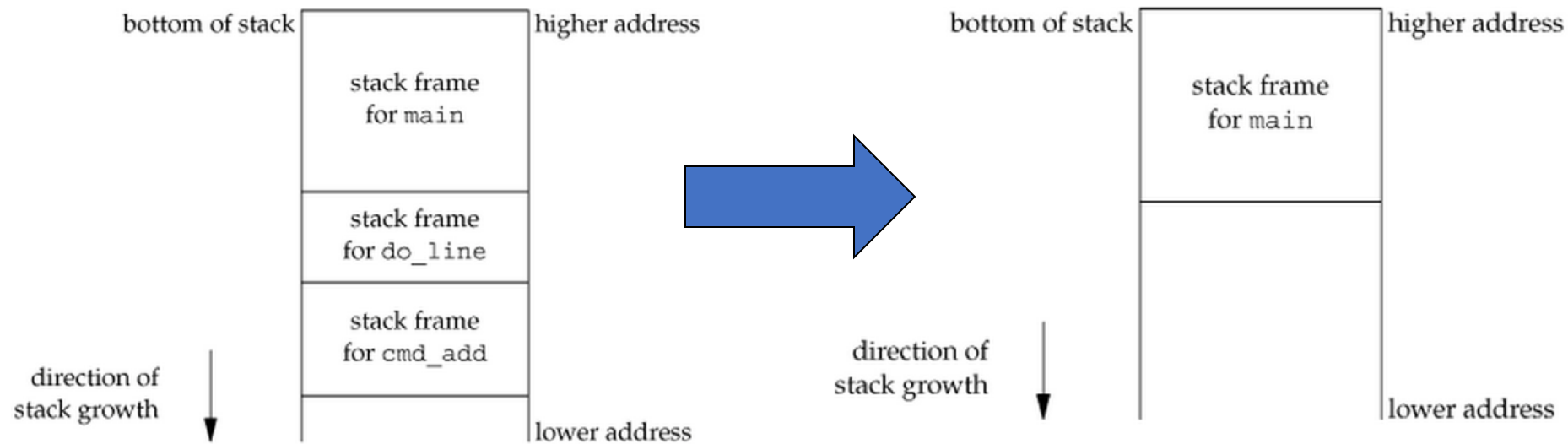
...

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 */
}

```

- What's going on?
 - Setjmp records whatever information it needs to in jmpbuffer and returns 0
 - 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
 - Calling longjmp causes the setjmp in main to return with a value of 1



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;
}

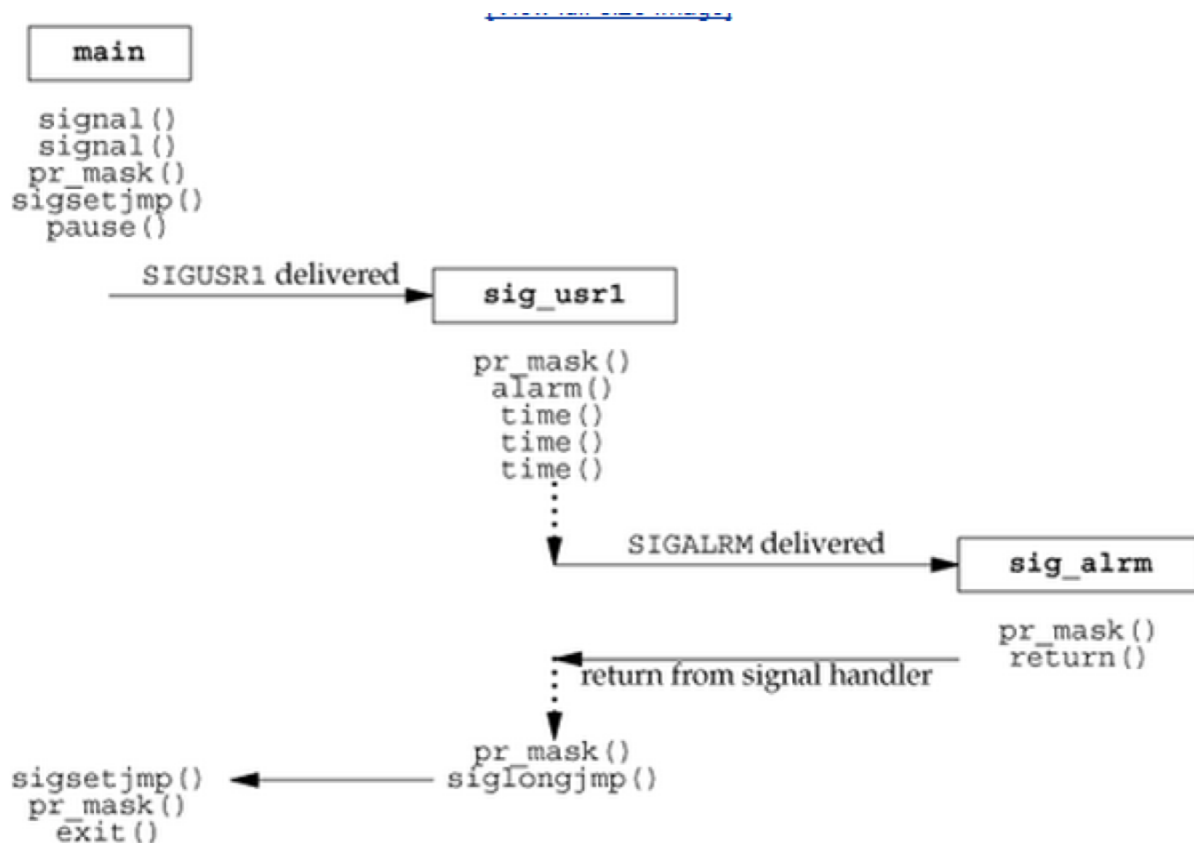
```

```

$ ./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_alm: SIGUSR1 SIGALRM
finishing sig_usr1: SIGUSR1
ending main:

[1] + Done                                   just press RETURN
./a.out &

```



System calls related to signals

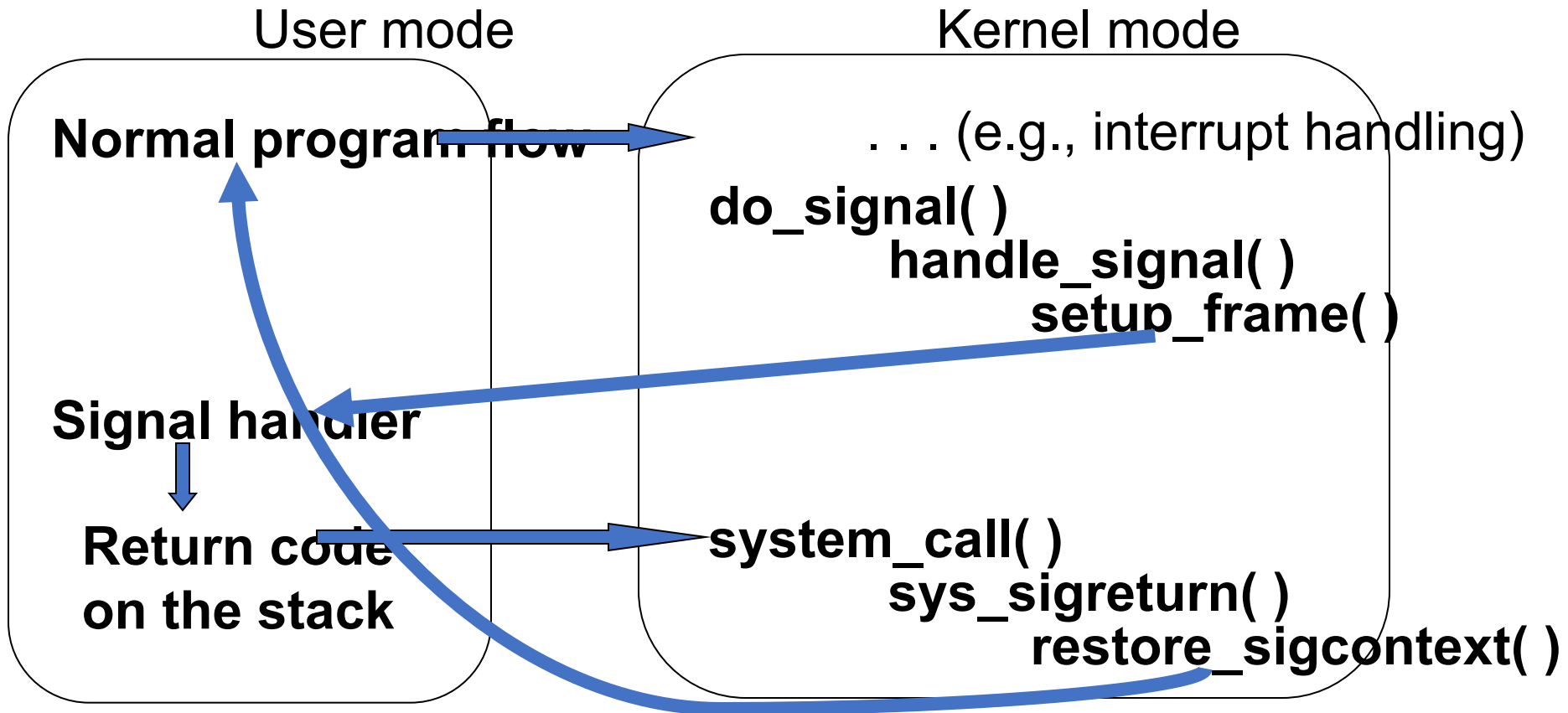
- `kill(signal_num, pid)` - to send a signal
- `signal(signal_num, handler)` - to handle it

Signal Handling (more)

- When does a process handle a signal?
 - Whenever it gets scheduled next after the generation of the signal
- We said the OS marks some members of the PCB to indicate that a signal is due
 - And we said the process will execute the signal handler when it gets scheduled
 - But its PC had some other address!
 - The address of the instruction the process was executing when it was scheduled last
 - Complex task due to the need to juggle stacks carefully while switching between user and kernel mode

Signal Handling (more)

- Remember that signal handlers are functions defined by processes and included in the user mode code segment
 - Executed in user mode in the process's context
- The OS **forces** the handler's starting address into the program counter
 - The user mode stack is modified by the OS so that the process execution starts at the signal handler



- **setup_frame**: sets up the user-mode stack
 - Forces Signal handler's address into PC and some "return code"
 - After the handler is done, this return code gets executed
 - It makes a system call such as **sigreturn** (in Linux) that does the following:
 - 1. Restores signal pending info in the PCB for the process
 - 2. Restores the User mode stack to its original state
 - When the system call terminates, the normal program execution can continue

Signal Handling (Visual)

Signal due indicators



Signal is not due



Signal is due



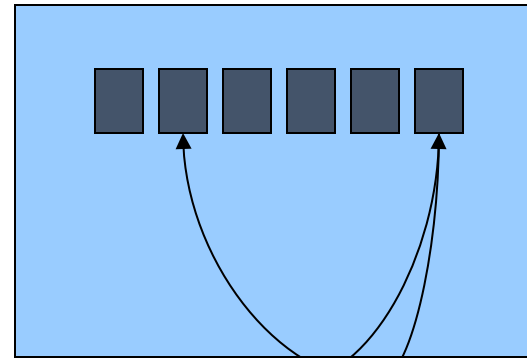
Kernel



P



Parent of P



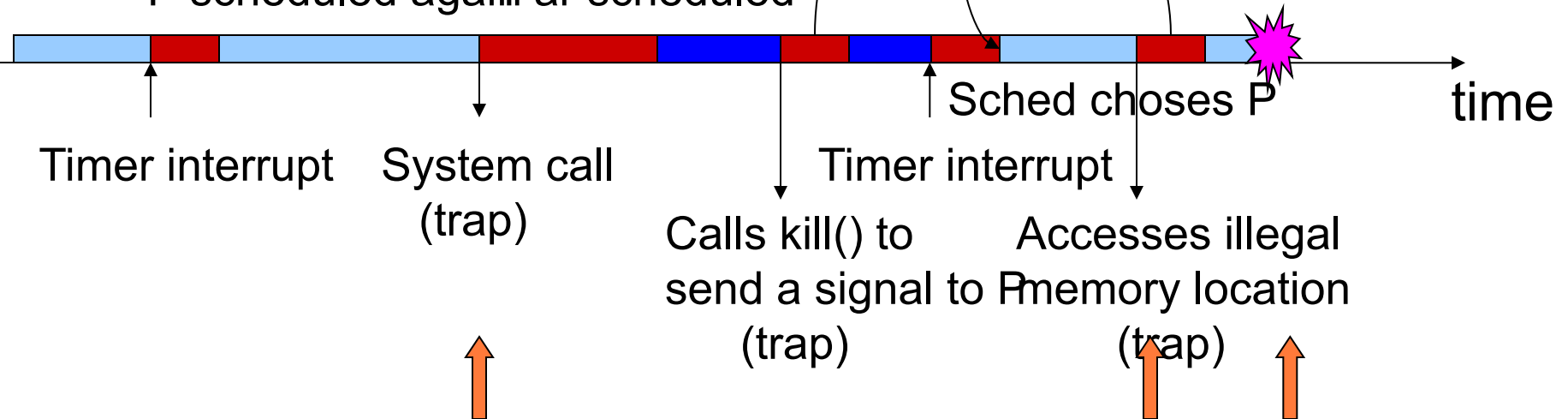
PCB of P

Runs SIGSEGV handler; dumps core and exits

P's signal handler runs

ISR run; assume Sys call done; assume P scheduled again

Par scheduled



These are the events that P sees (its view of what is going on)

User-level Threads Implementation: Putting it all Together

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

Process v K thread v 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 on a multi-CPU system?
 - What happens upon blocking I/O?

Process v K thread v U thread

			Multi-process	Multi-K-thread	Multi-U-thread
context switch	direct overheads		yes	yes	yes
	indirect overheads	TLB flush/cache pollution	yes	depends	no
		kernel/user mode transition	yes	yes	no
memory needs	user space memory		bigger	equal	
	kernel memory		biggest	middle	smallest
parallelism and scheduling	What happens on multi-processor system?		yes	yes	no
	What happens upon blocking I/O?		no	no	yes

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. $Q = \inf$
- ii. $Q > T$
- iii. $S < Q < T$
- iv. $Q = S$
- v. Q is nearly 0