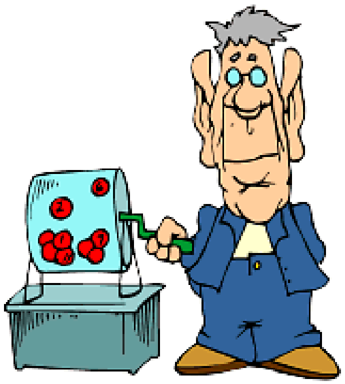
Lottery Scheduling

- Perhaps the simplest proportional-share scheduler
- Create lottery tickets equal to the sum of the weights of all processes
- Draw a lottery ticket and schedule the process that owns that ticket

Lottery Scheduling Example

P1=6 *P2=9*

P1		P2		
1	4	7	10	13
2	5	8	11	14
3	6	9	12	15

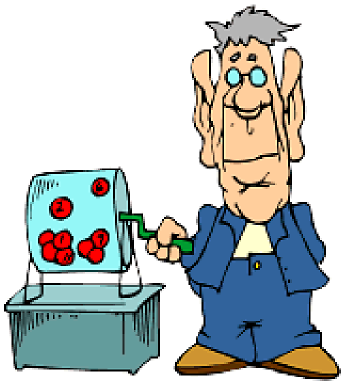


9

Schedule P2

Lottery Scheduling Example

P1=6 *P2=9*



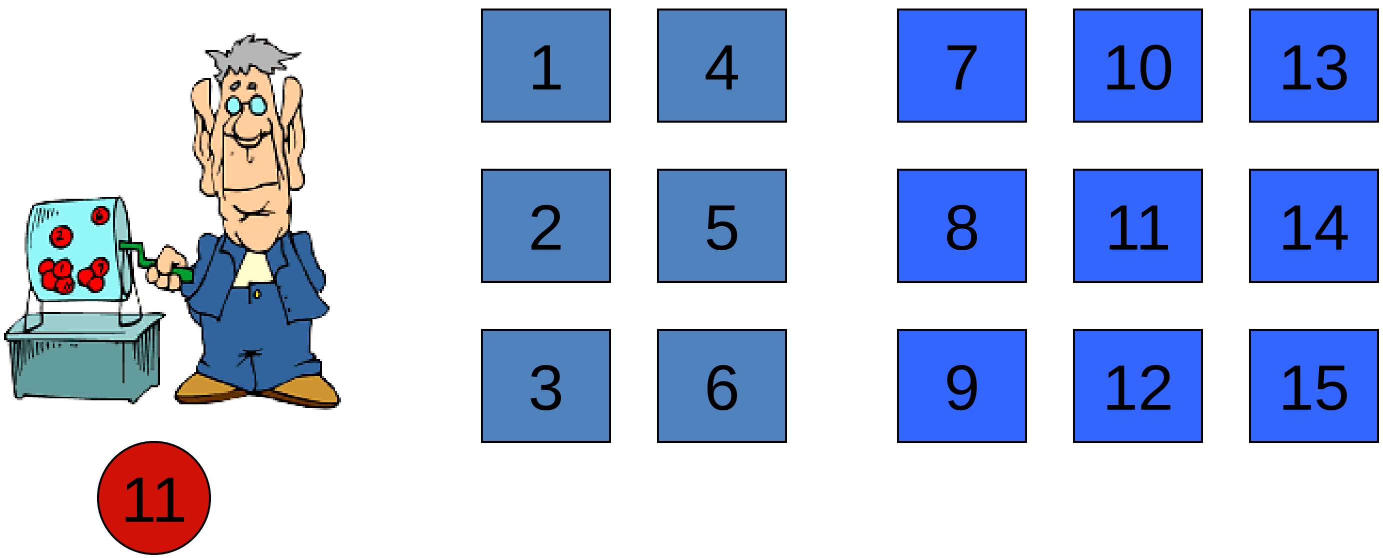
1	4	7	10	13
2	5	8	11	14
3	6	9	12	15

3

Schedule P1

Lottery Scheduling Example

P1=6 *P2=9*



- As t $\longrightarrow \infty$, processes will get their share (unless they were blocked a lot)
- Problem with Lottery scheduling: Only probabilistic guarantee
- What does the scheduler have to do
  - When a new process arrives?
  - When a process terminates?

Schedule P2

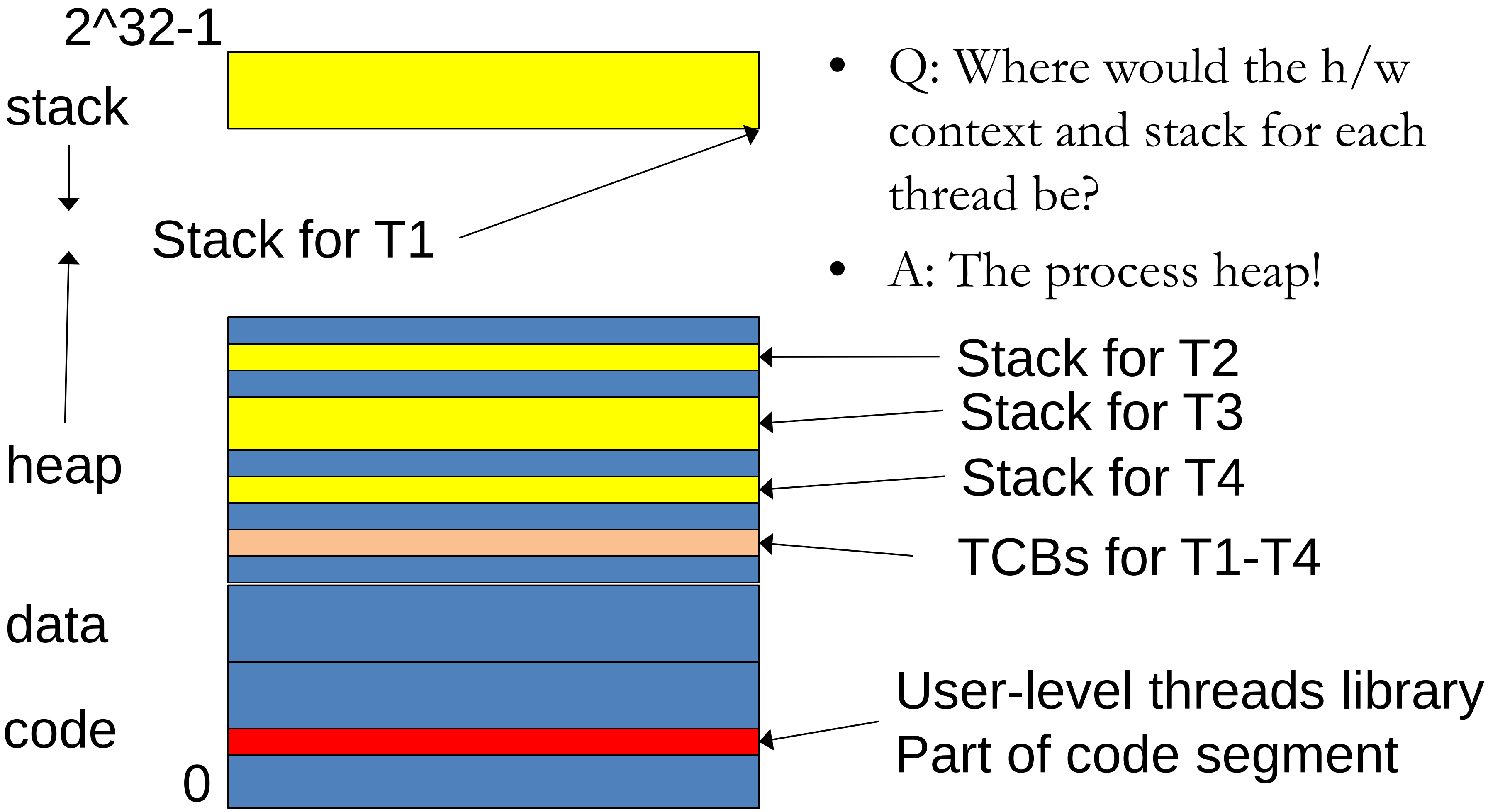
Implementing User-level Threads

- 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!**

```c
#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");
}
```

```c
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

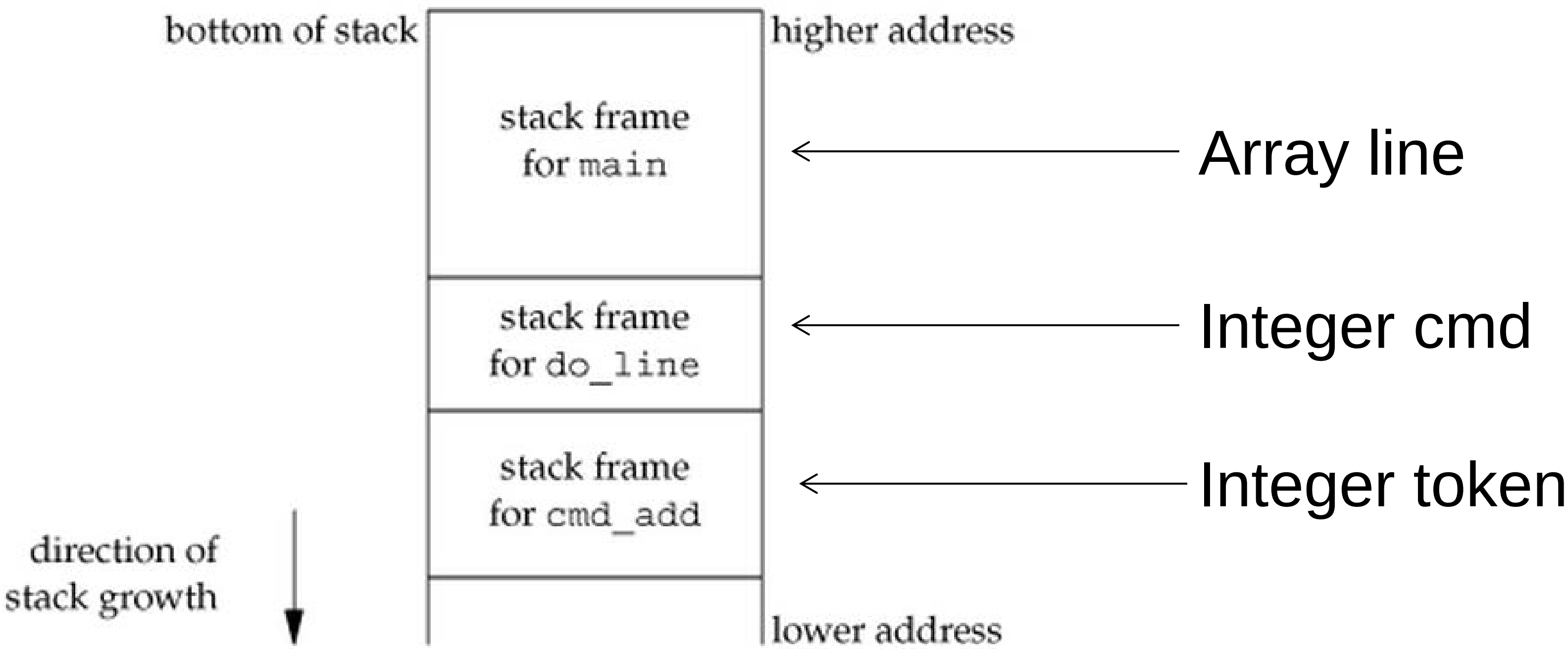
```
Value of i is now 0
Value of i is now 1
Value of i is now 2
Value of i is now 3
Value of i is now 4
Value of i is now 5
OK, saving state...
Saved CPU state and breaking from loop.
Restored CPU state, continuing where we saved
Value of i is now 6
Value of i is now 7
Value of i is now 8
Value of i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

Example 2: Deeply-nested function calls

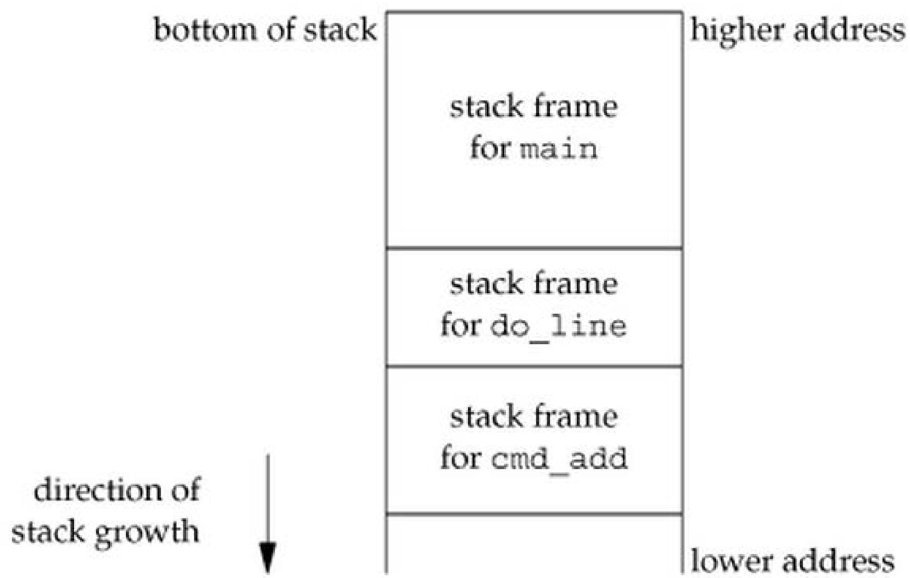
- Consider a program calling functions in a nested way:
  - main() call do_line()
  - do_line call cmd_add()

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?

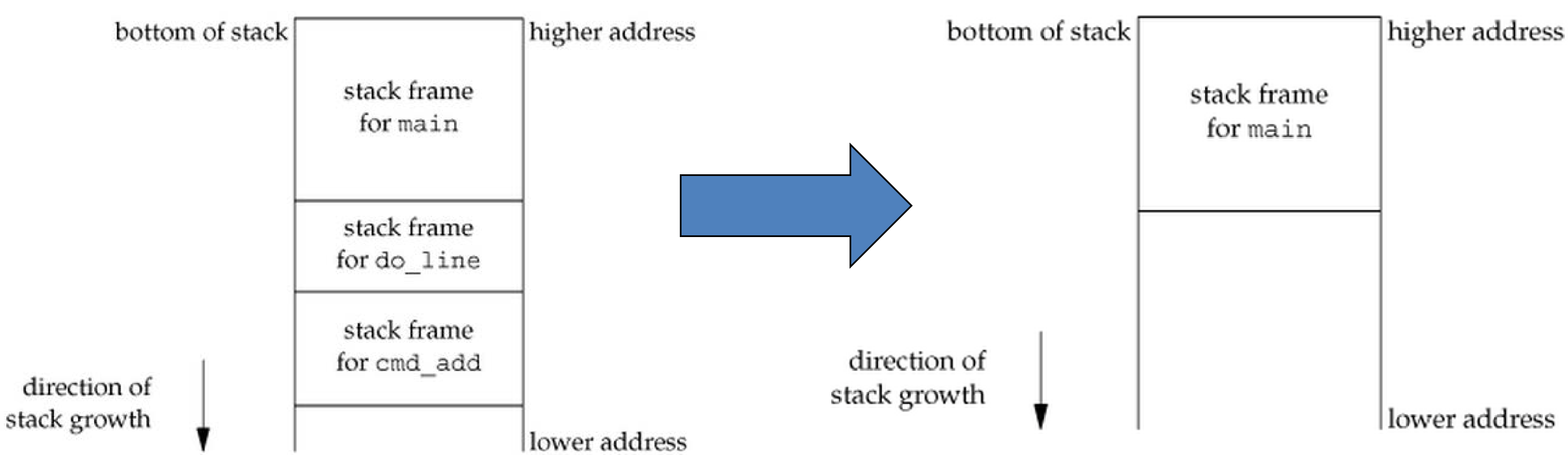


- What if when cmd_add encountered a non-fatal error, we would like to ignore the rest of the line and return to main?
  - goto would only return within cmd_add
  - Using special return value would involve going up the entire set of nested calls!
  - Would like a non-local goto: setjmp/longjmp provide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



```c
#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


bottom of stack
higher address
stack frame
for main
stack frame
for do_line
stack frame
for cmd_add
direction of
stack growth
lower address
bottom of stack
higher address
stack frame
for main
direction of
stack growth
lower address

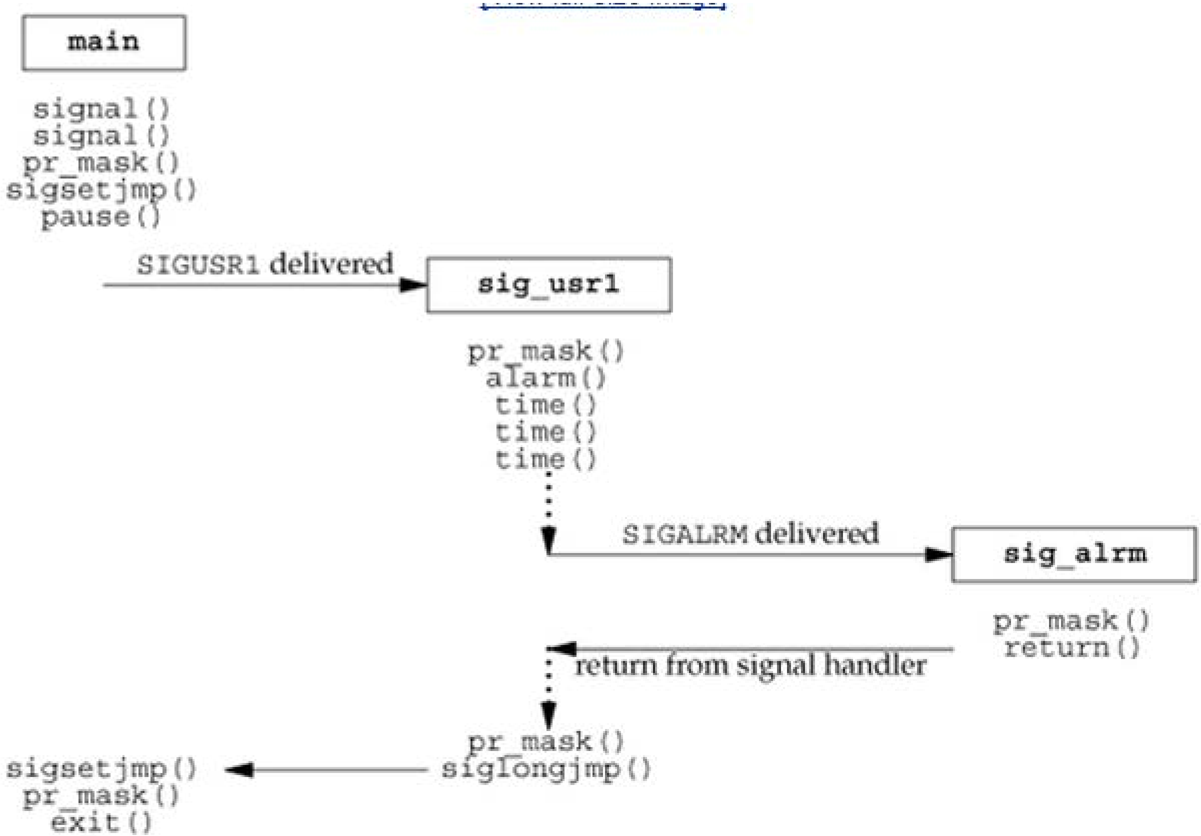
sigsetjmp() and siglongjmp()

- 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

```c
#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: ");
}
```

```c
#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 &
```



System calls related to signals

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

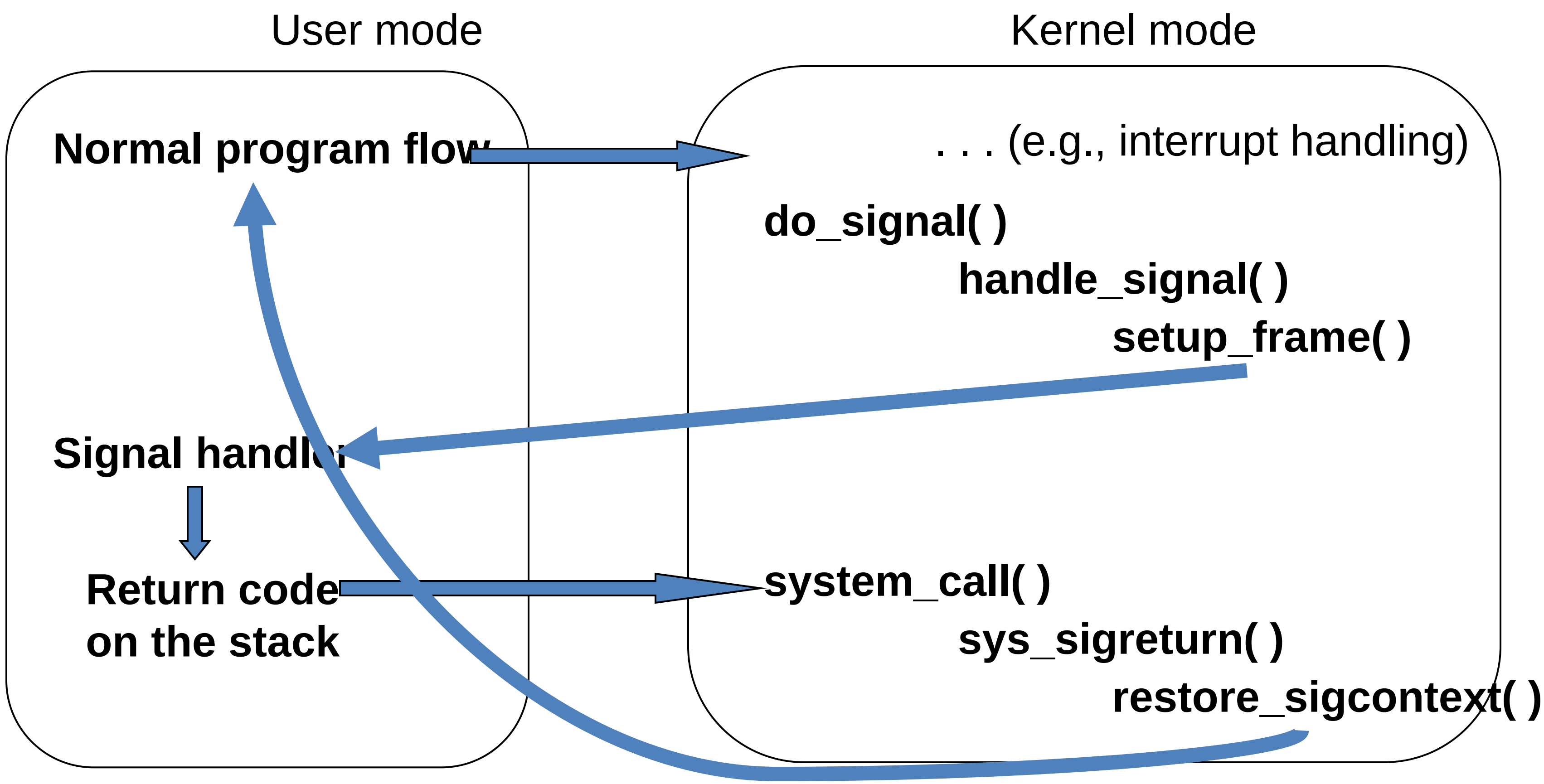
Signal Handling (more)

- 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)

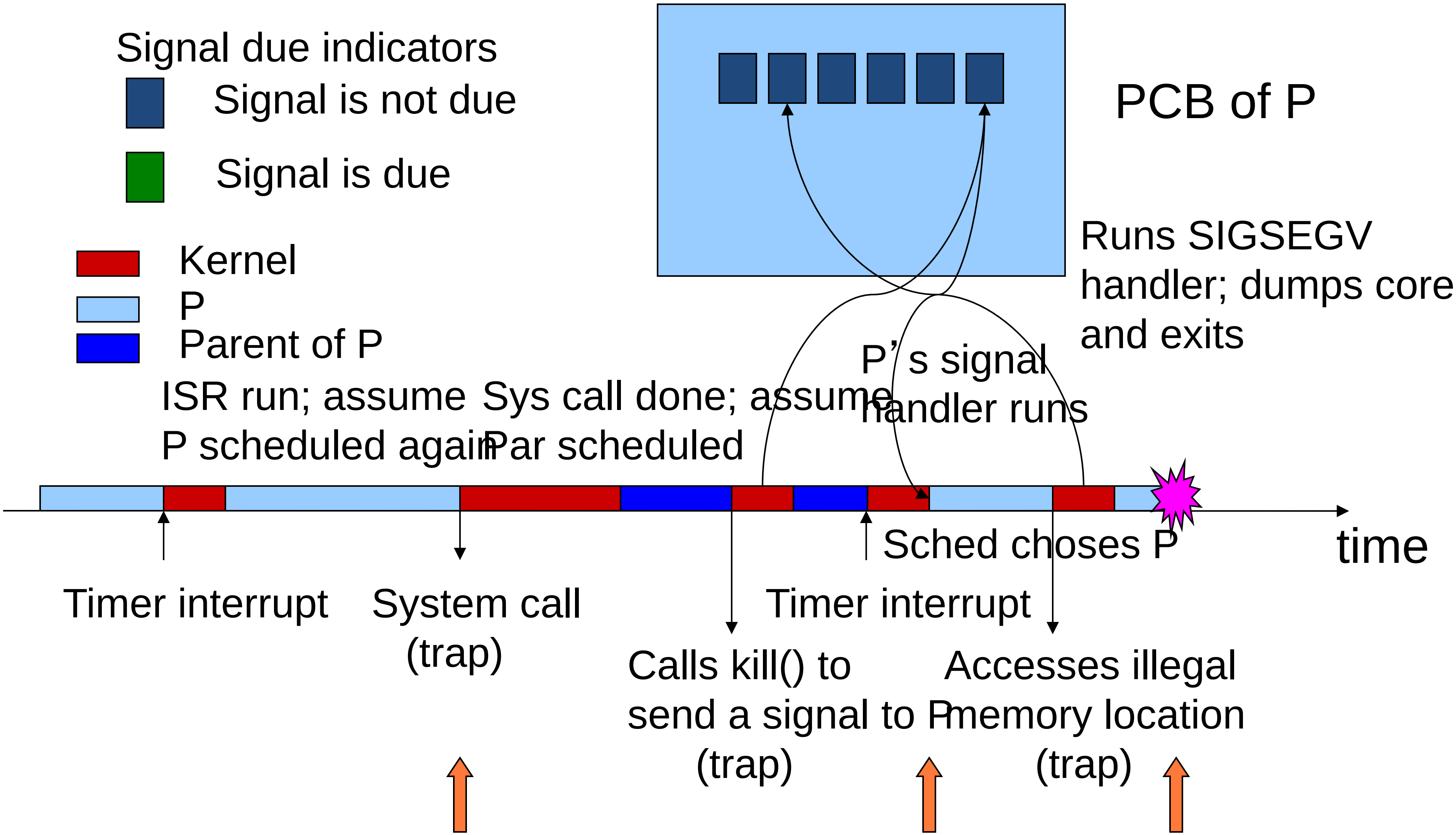
- Remember that signal handlers are functions defined by processes and included in the user mode code segment
  - Executed in user mode in the process's context

- 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)



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