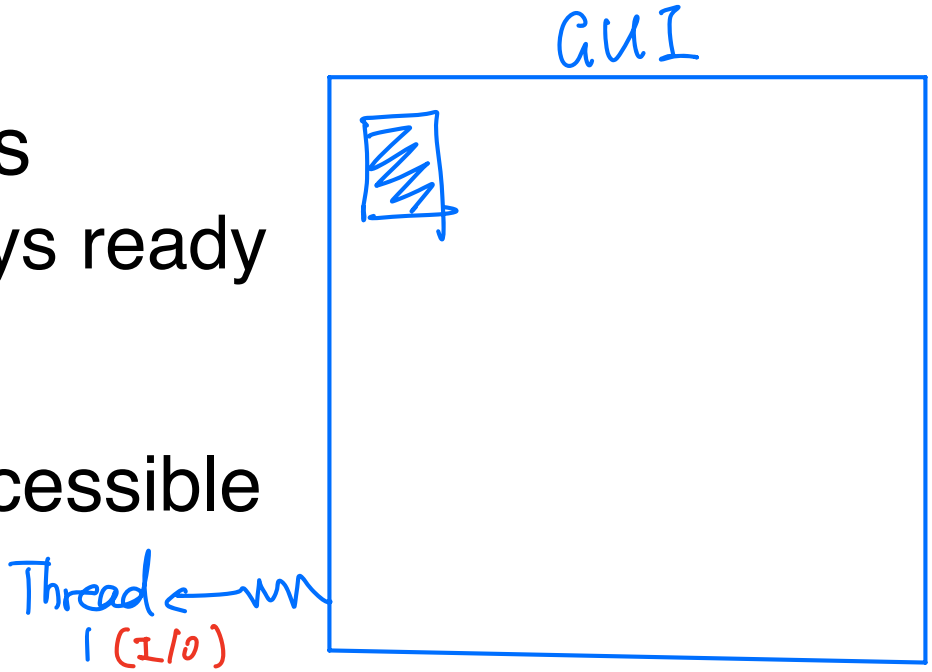
PENNSTATE

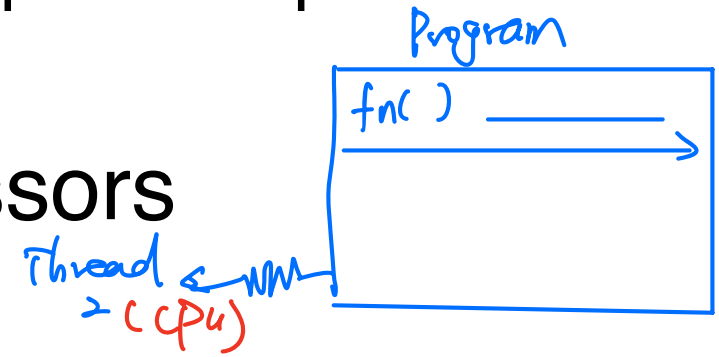
Threads

Why Threads?

Advantages of Threads

- Improve Responsiveness
  - Ideally, a thread is always ready
- Resource Sharing
  - All the stuff is easily accessible
- Economy of Resources
  - Thread resources are cheaper than process resources
- Utilization of Multiprocessors
  - Get all of them running

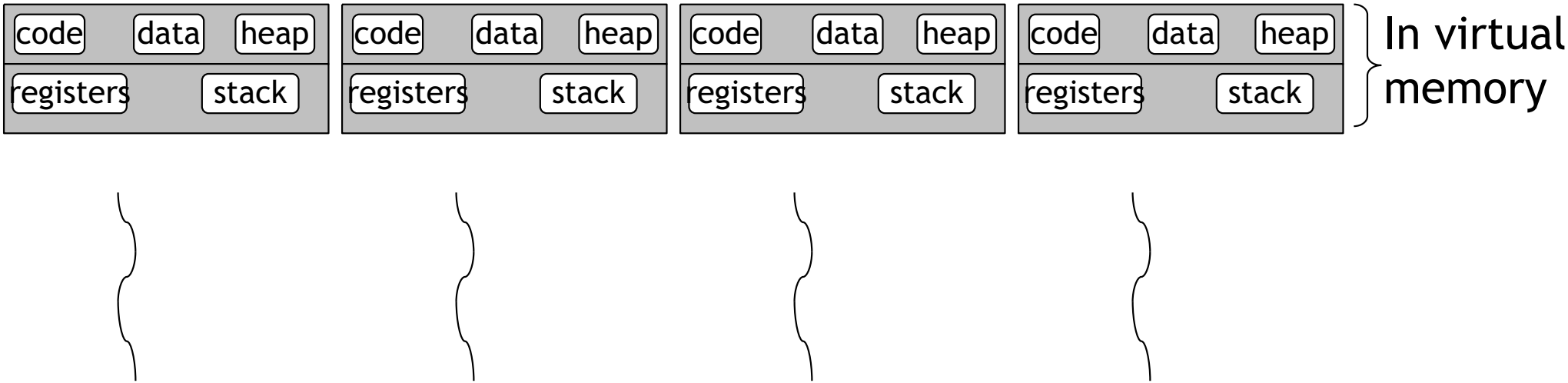




PENNSTATE

Old Approach: Multiple Cooperating Processes

E.g., MP Web browser



URL parsing process

Network sending process

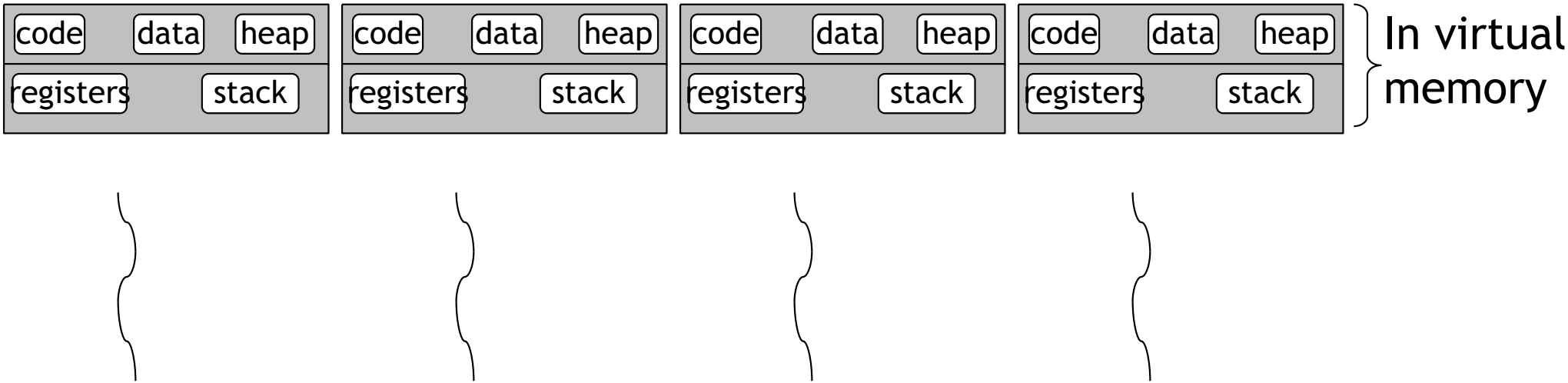
Network reception process

Interprets response, composes media together and displays on browser screen

code
data
heap
> share

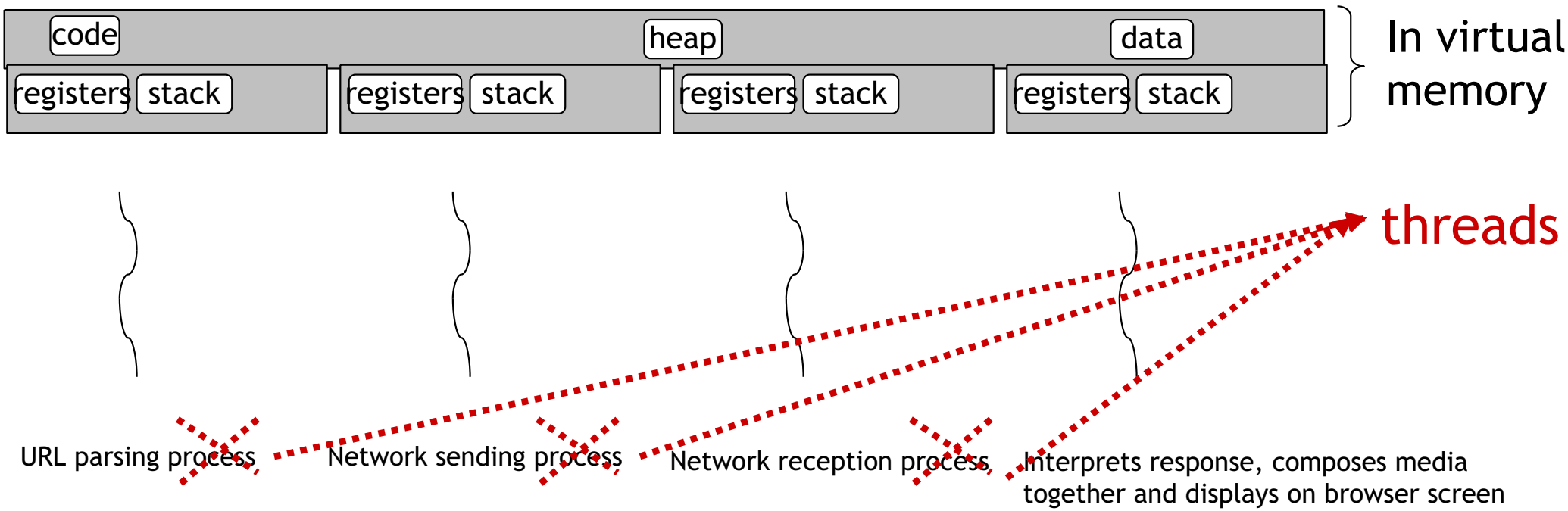
stack
registers
> not share

E.g., MP Web browser



Advantages and Disadvantages: TLB Hits, Physical Memory Usage, Security

Share!

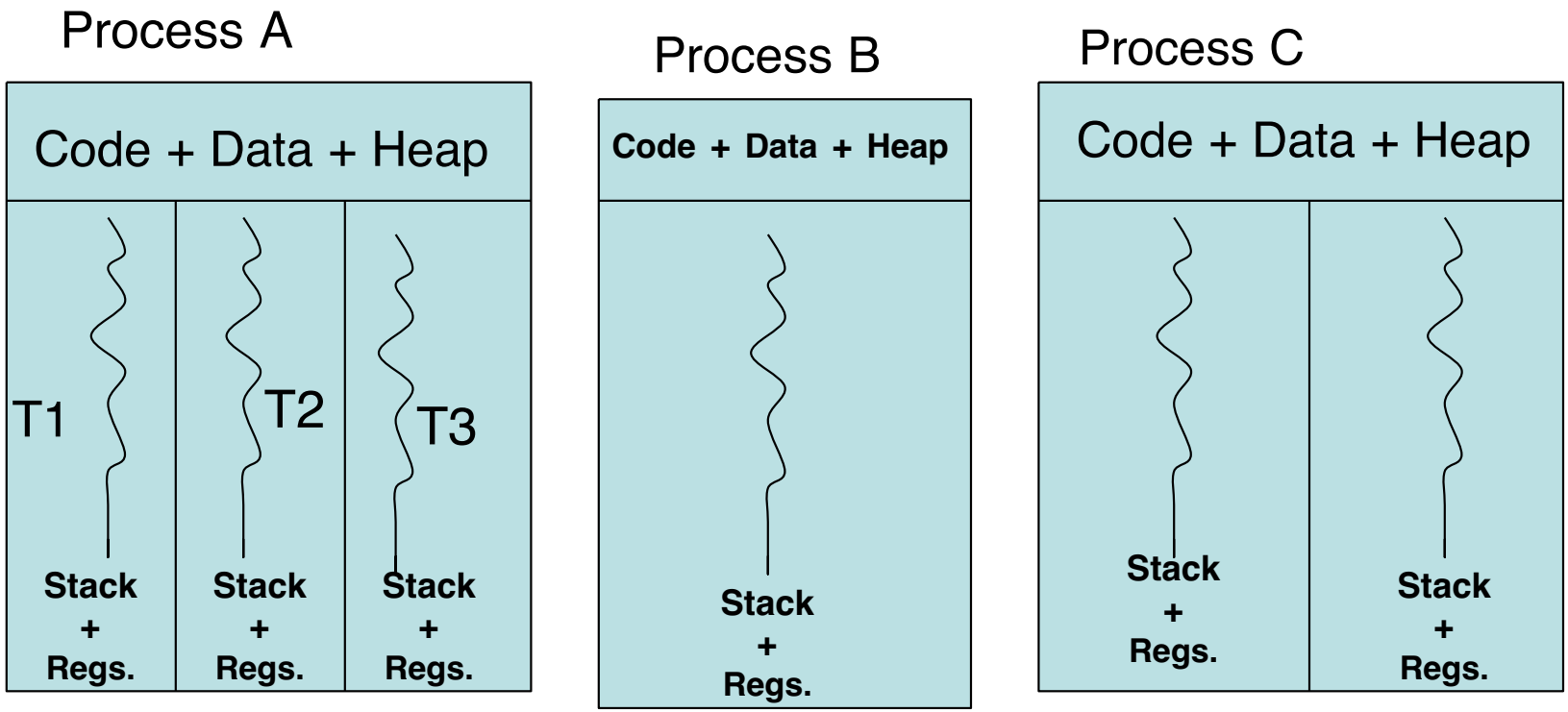


- Share code, data, heap in same address space
  - Only registers and stack must be per thread
  - Why?

PENNSTATE

New Approach: Multiple Cooperating Threads in One Process

Threads



Operating System

Hardware

TCB

Program

pthread_create

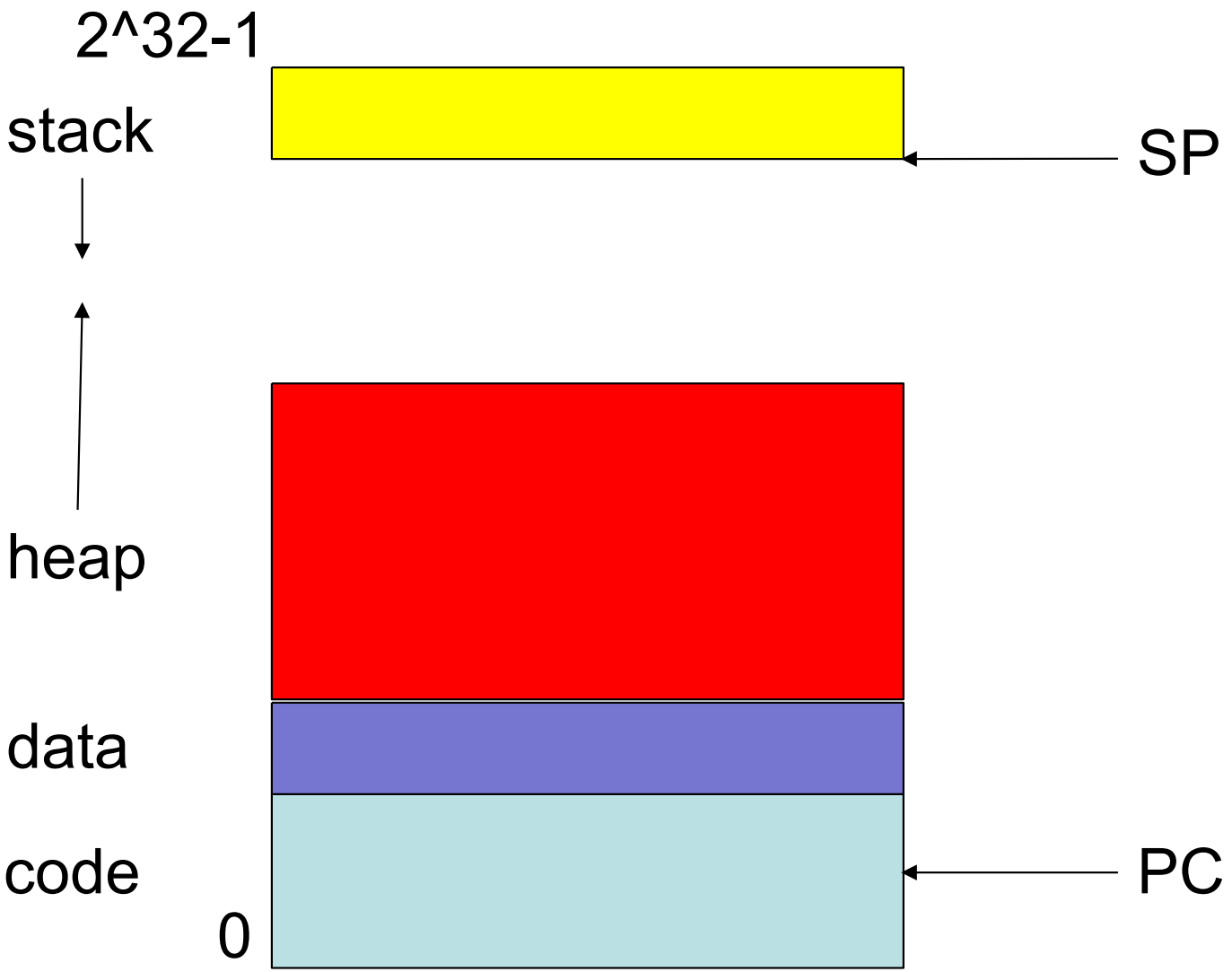
A B

PCB

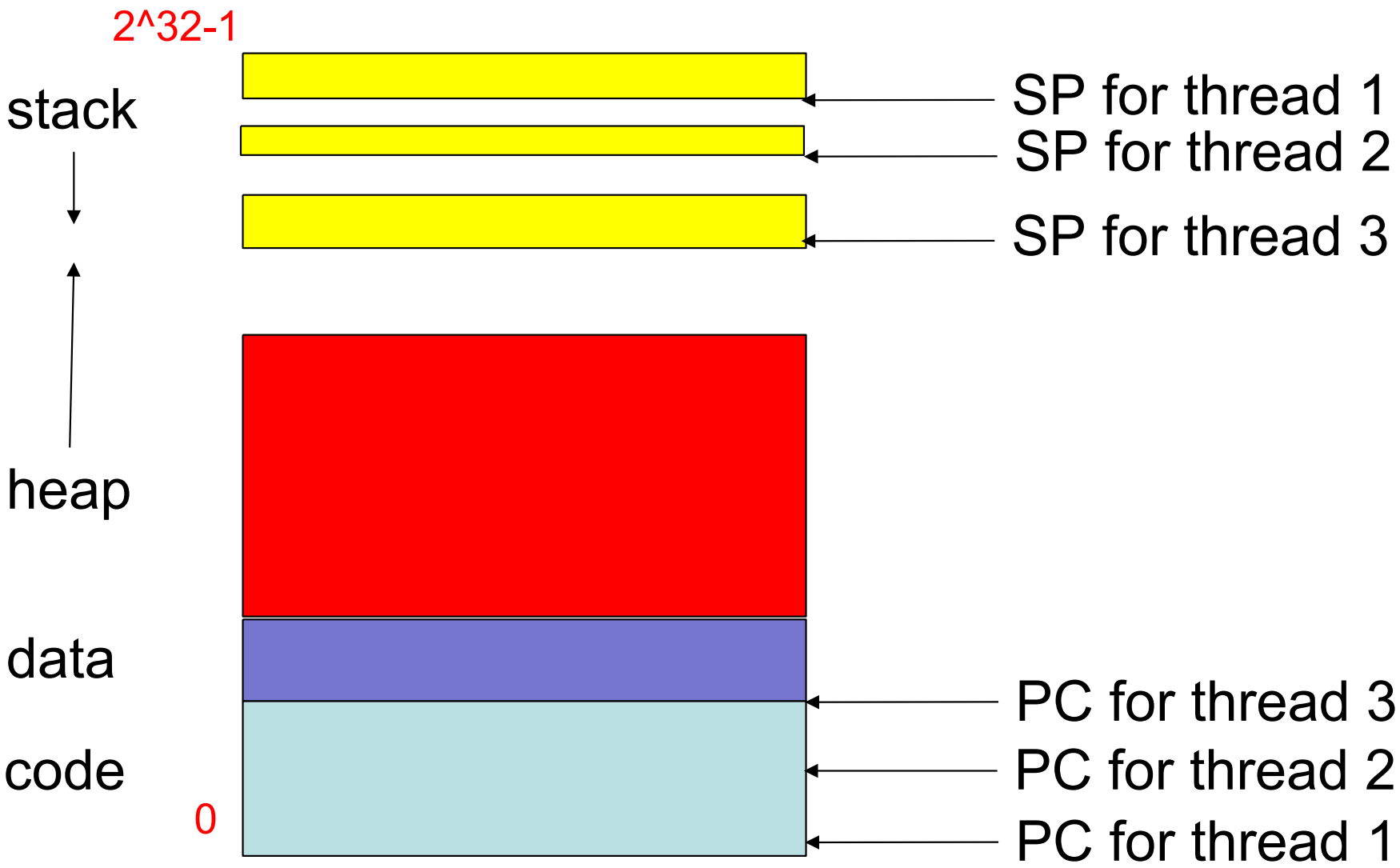
address space

page table
Malloc → sbrk → "promise"
Open Files / Sockets.

(Old) Process Address Space



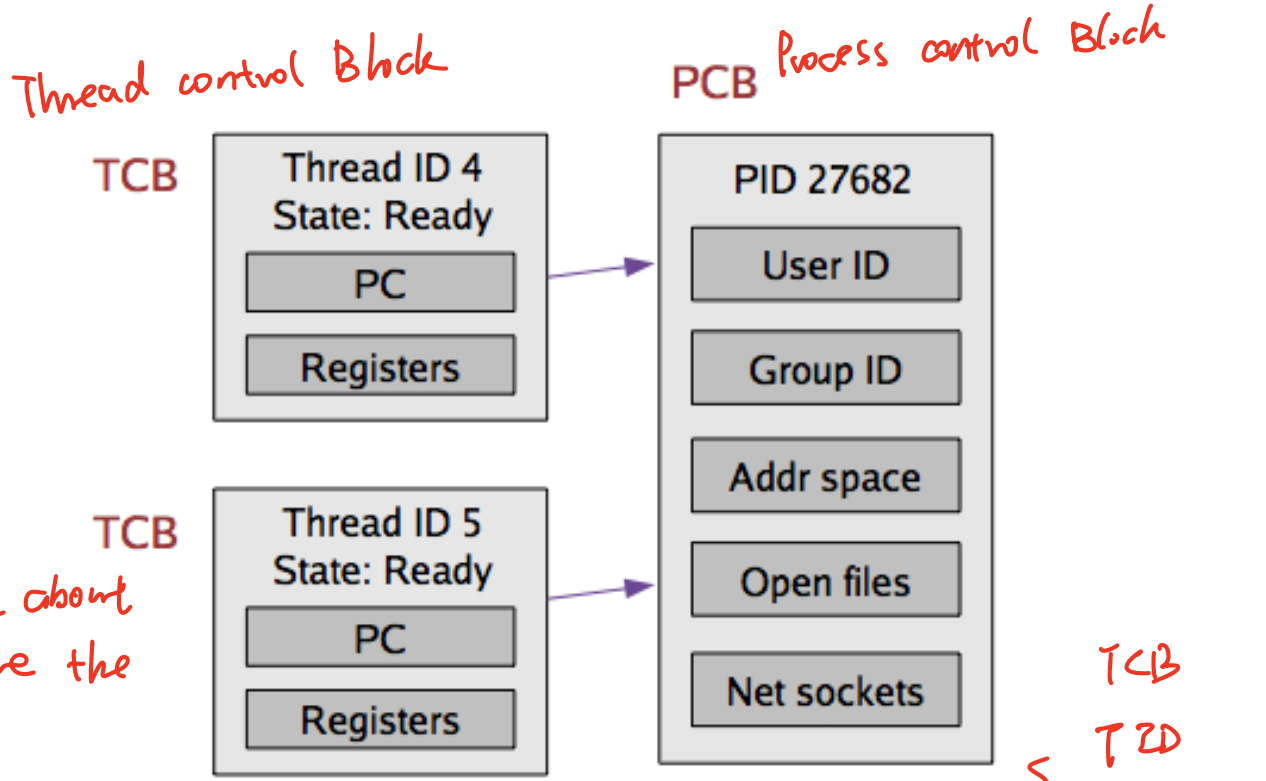
(New) Address Space w/ Threads



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)



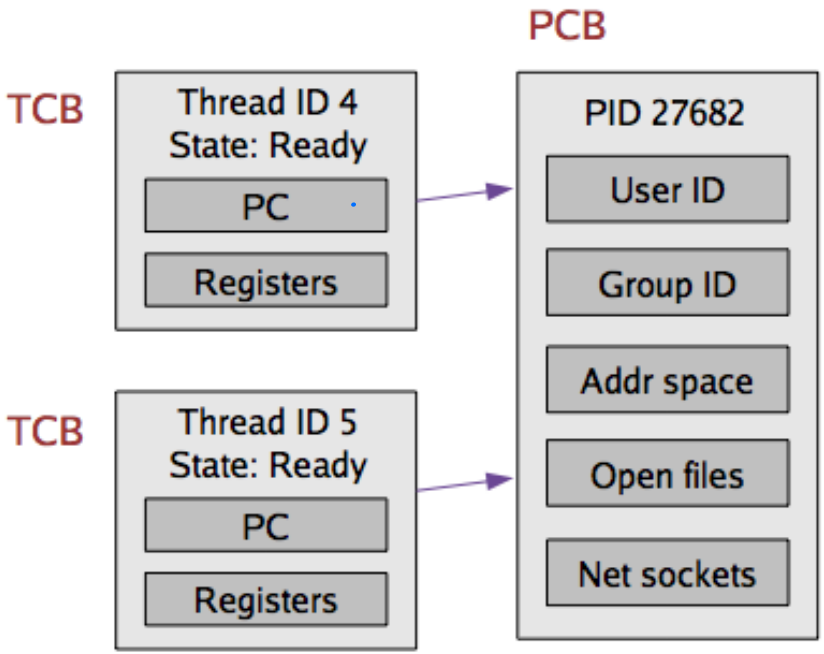
context switching care about TCB. since TCB have the context info.

TCB
TID
state
PC
register

PENNSTATE

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!

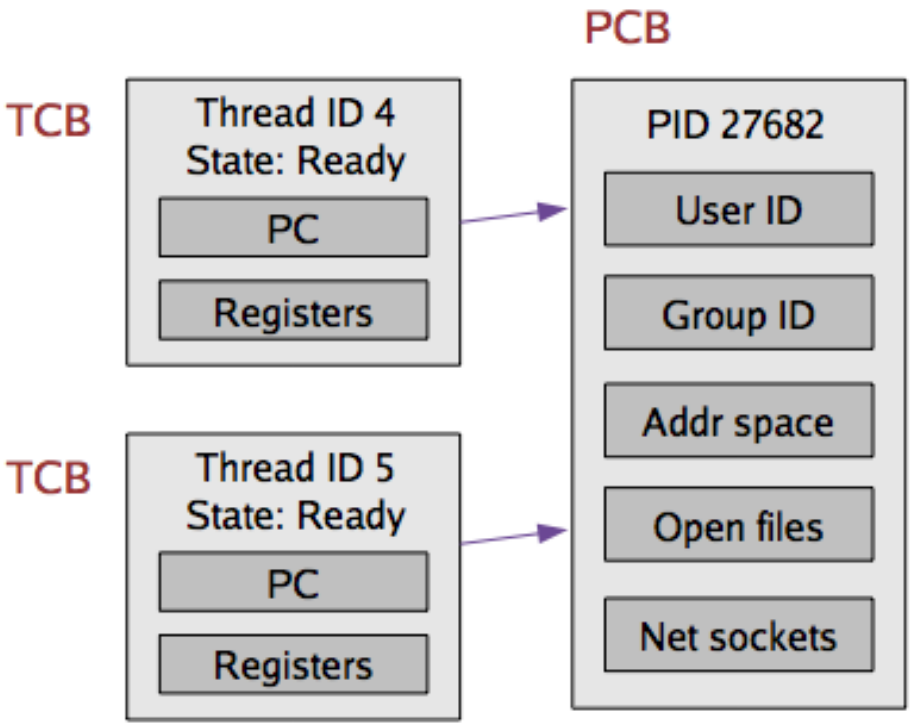


PENNSTATE

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

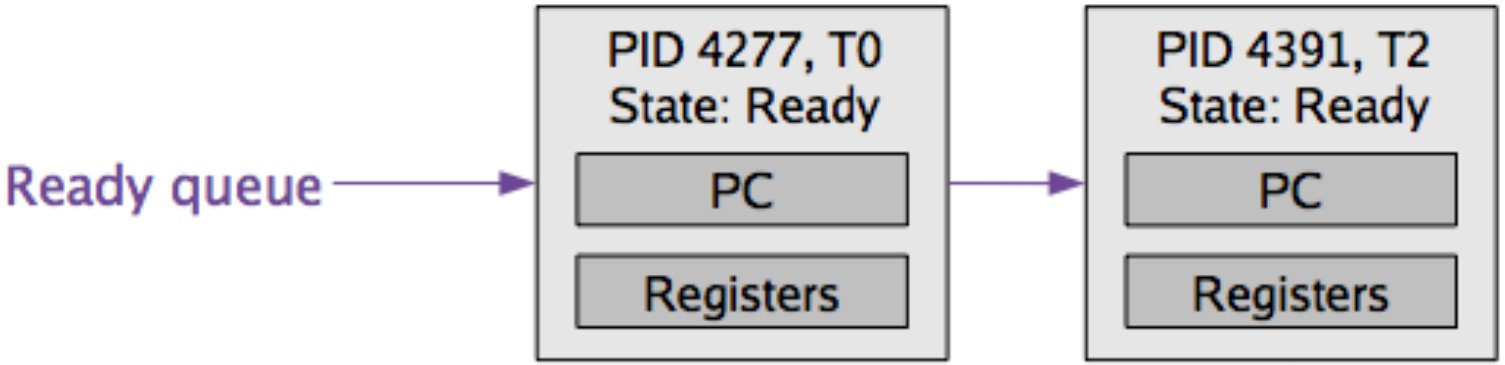
TCB is constrained to PCB



PENNSTATE

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



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

PENNSTATE

Security

- What about security?
- What happens when a bug occurs
  - in one process of a set of cooperating processes?
  - in one thread in a process?
- Crash
- Exploit

- Good news is that new hardware features are being developed to obtain some isolation between threads in the same process

PENNSTATE

Threading Models

PENNSTATE

Threading Models

- Programming: **Library or system call interface**
  - User-Space Threading
    - Thread management support in user-space library
    - Linked into your program
  - Kernel Threading
    - Thread management support in the kernel
    - Invoked via system call
- Scheduling: **Application or kernel scheduling**
  - May create user-level or kernel-level threads
    - NOTE: CPU virtualization only runs kernel threads!

PENNSTATE

User-Space Threads

- Thread management support in user-space library
  - Sets of functions for creating, invoking, and switching among threads
- Linked into your program
  - Thread libraries
- Examples
  - POSIX Threads (PThreads)
  - Win32 Threads
  - Java Threads

PENNSTATE

Kernel Threads

- Thread management support in kernel
  - Sets of system calls for creating, invoking, and switching among threads
- Supported and managed directly by the OS
  - Thread objects in the kernel
- Nearly all OS support a notion of threads
  - Linux -- thread and process abstractions are mixed
  - Solaris
  - Mac OS X
  - Windows
  - …

three models

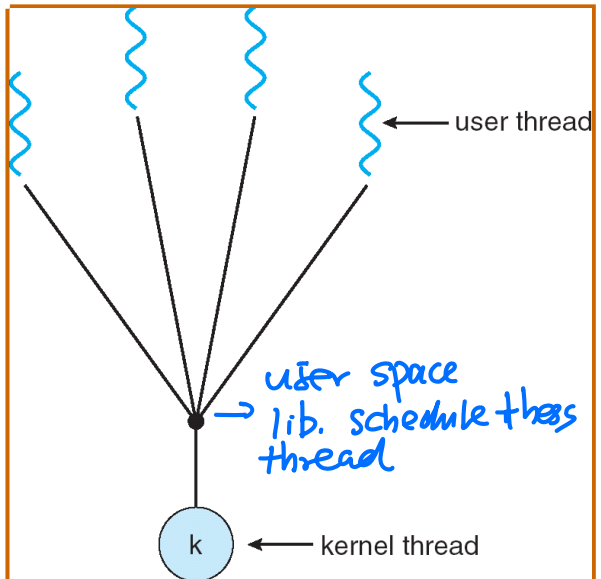
1

Many-to-One Thread Model

- Many user-level threads correspond to a single kernel thread
  - Kernel is not aware of the mapping
  - Handled by a thread library
- How does it work?
  - Create and execute a new thread
  - Upon *yield*, switch to another thread in the same process
    - Kernel is unaware
  - Upon *wait*, all threads are blocked
    - Kernel is unaware there are other options
    - Can't wait and run at the same time

voluntary stand by when can't access data while others are modifying it

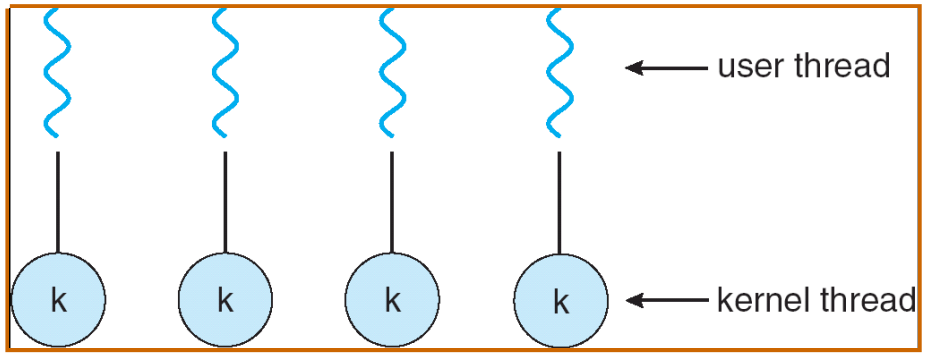
I/O burst waiting



we distinct user thread and kernel thread because of portability. we can run user thread on different system without change the OS. since user thread is support by pthread lib.

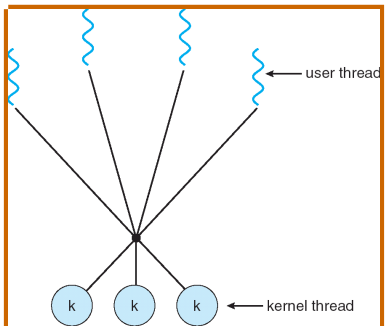
One-to-One Thread Model

- One user-level thread per kernel thread
  - A kernel thread is allocated for every user-level thread
  - Must get the kernel to allocate resources for each new user-level thread
- How does it work?
  - Create new thread, including system call to kernel
  - Upon *yield*, switch to another thread in system
    - Kernel is aware
  - Upon *wait*, another thread in the process may run
    - Only the single kernel thread is blocked
    - Kernel is aware there are other options in this process



Many-to-Many Thread Model

- A pool of user-level threads maps to a pool of kernel threads
  - Pool sizes can be different (kernel pool is no larger)
  - A kernel thread is pool is allocated for every user-level thread
  - No need for the kernel to allocate resources for each new user-level thread
- How does it work?
  - Create new thread (may map to kernel thread dynamically)
  - Upon *yield*, switch to another thread in system
    - Kernel is aware
  - Upon *wait*, another thread in the process may run
    - If a kernel thread is available to be scheduled to that process
    - Kernel is aware of the mapping between process threads and kernel threads



PENNSTATE

Threading Systems

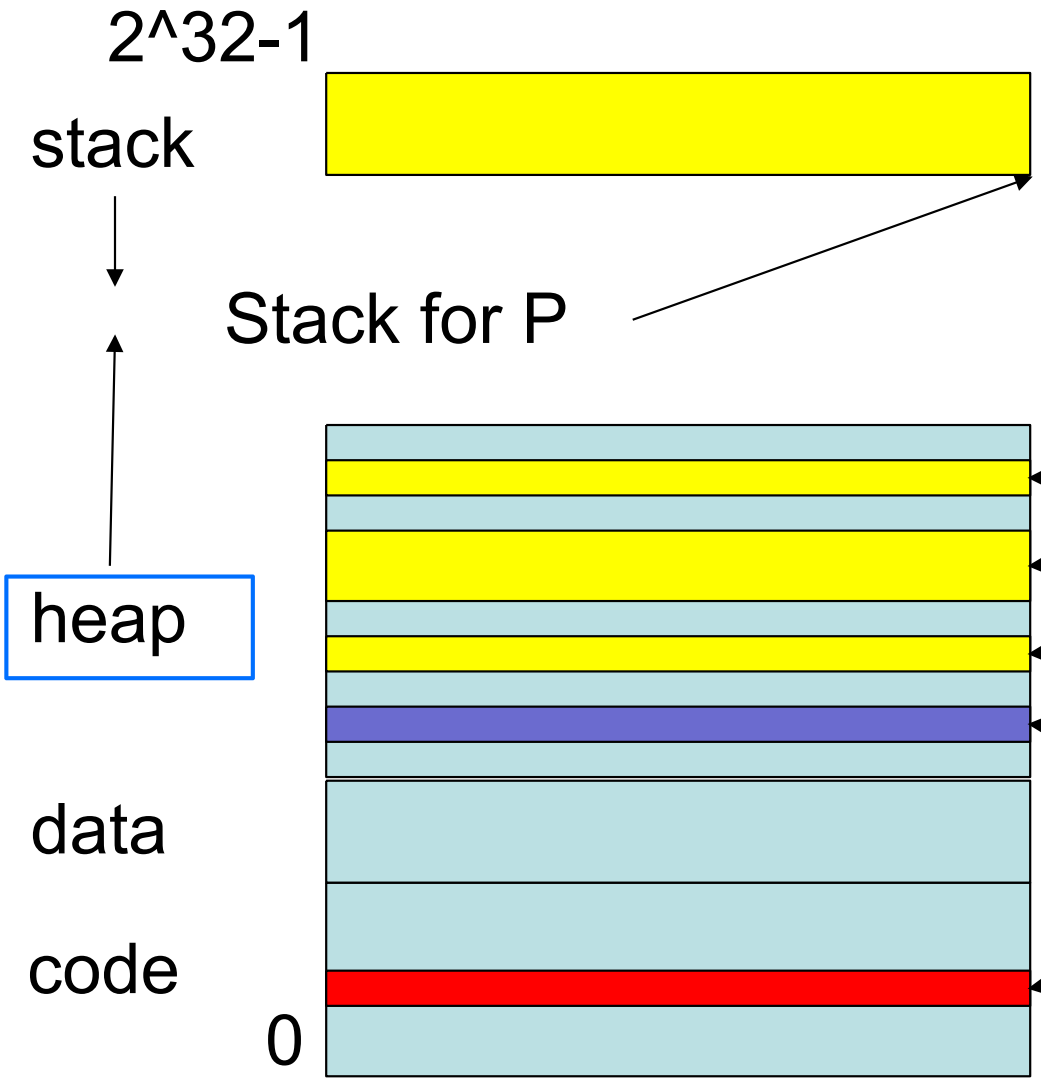
PENNSTATE

Implementing User-level Threads

- It should be clear that we would need the following:
  - UT.1: Scheduling and context switching as part ofeach process
    - E.g., a library that we link against our process
  - UT.2: Store hardware context and stack for each thread in each process's own address space
  - UT.3: Intervene execution of threads from time to time and run itself (like timer interrupt handling by OS)
  - UT.4: Restore hardware context in user space process

① schedule and context switch
② store hardware context and stack
③ intervene execution of thread
④ restore hardware context

Examples: UT.1, UT.2



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

Stack for T1
Stack for T2
Stack for T3
TCBs for T1-T3
in heap

User-level thread library
- Part of code segment

PENNSTATE

Implementing User-level Threads

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

PENNSTATE

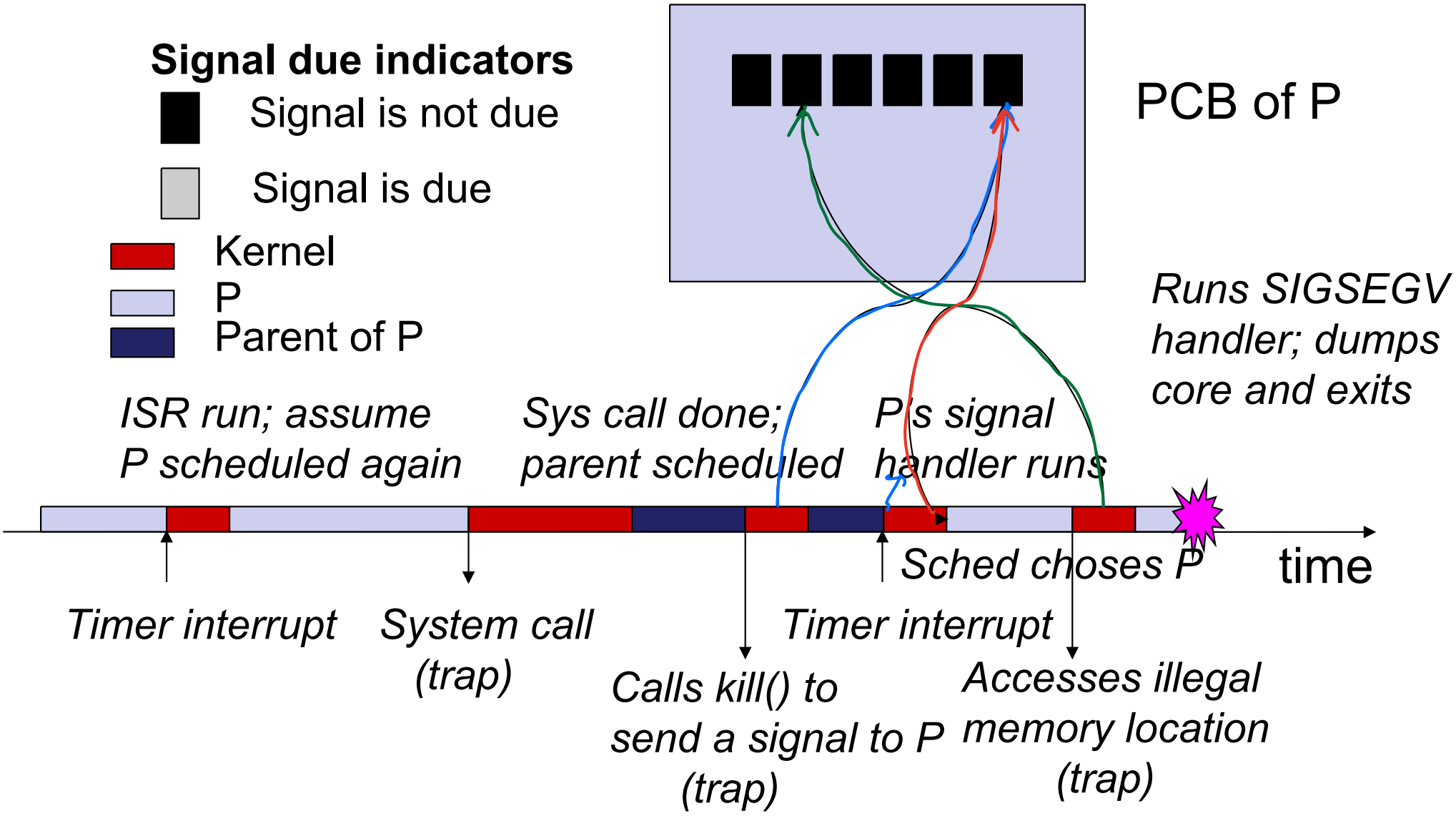
UT.3: SIGALRM signal

- UT.3: Intervene execution of threads from time to time and run itself (like timer interrupt)
  - Employ "virtual" interrupts, a.k.a. signals
    - Request the OS to send periodic "alarm" signals to the process (SIGALRM)
  - Implement a signal handler for SIGALRM
  - 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!**

Signal Handling

- What's a signal?
  - A form of IPC
  - Send a particular signal to another process
- The receiver's signal handler processes the signal on receipt
- Example
  - Tell the Internet daemon (**inetd**) to reread its config file
  - Send signal to **inetd**: kill -SIGHUP <pid>
  - **Inetd**'s signal handler for the SIGHUP signal re-reads the config file
- Note: some signals cannot be handled by the receiving process, so they cause default action (i.e., kill the process)

Signal Handling (Visual)



PENNSTATE

System Calls for Signals

- kill(signal_num, pid) - to send a signal
- signal(signal_num, handler) - to handle it
  - man pages recommend using sigaction structure

PENNSTATE

Signal Handling

- Synchronous Signals
  - Generated by the kernel for the process
  - E.g., due to an exception -- divide by 0
    - Events caused by the thread receiving the signal
- Asynchronous Signals
  - Generated by another process
- Asynchronous signals are more difficult for multithreading

Asynchronous

Synchronous

PENNSTATE

Signal Handling and Threads

- So, you send a signal to a process
  - Which thread should it be delivered to?
- Choices
  - Thread to which the signal applies
  - Every thread in the process
  - Certain threads in the process
  - A specific signal receiving thread
- It depends…

PENNSTATE

Signal Handling and Threads

- Synchronous vs. Asynchronous Cases
- Synchronous
  - Signal is delivered to the same process that caused the signal
  - Which thread(s) would you deliver the signal to?
- Asynchronous
  - Signal generated by another process
  - Which thread(s) in this case?

```
#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;
}
```

PENNSTATE

Signal Handling

- When does a process handle a signal?
  - Whenever it gets scheduled next after the generation of the signal

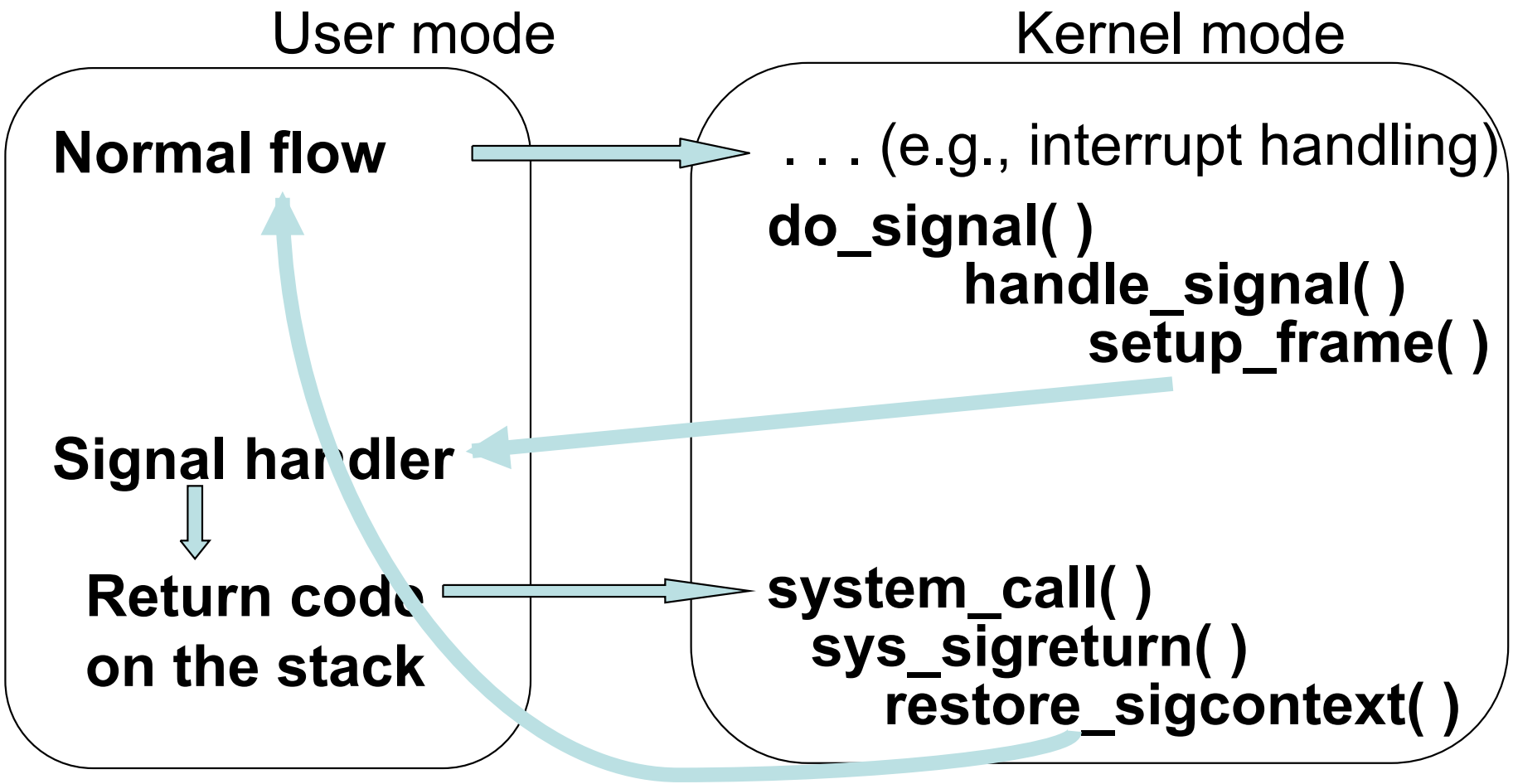
- The OS marks some members of the PCB to indicate that a signal is due
  - And we said the process will execute the signal handler when it gets scheduled
  - But its PC had some other address!
    - The address of the instruction the process was executing when it was scheduled last
  - Complex task due to the need to juggle stacks carefully while switching between user and kernel mode
    - E.g., there is a special signal stack to store stack for signal handlers

PENNSTATE

Signal Handling

- 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



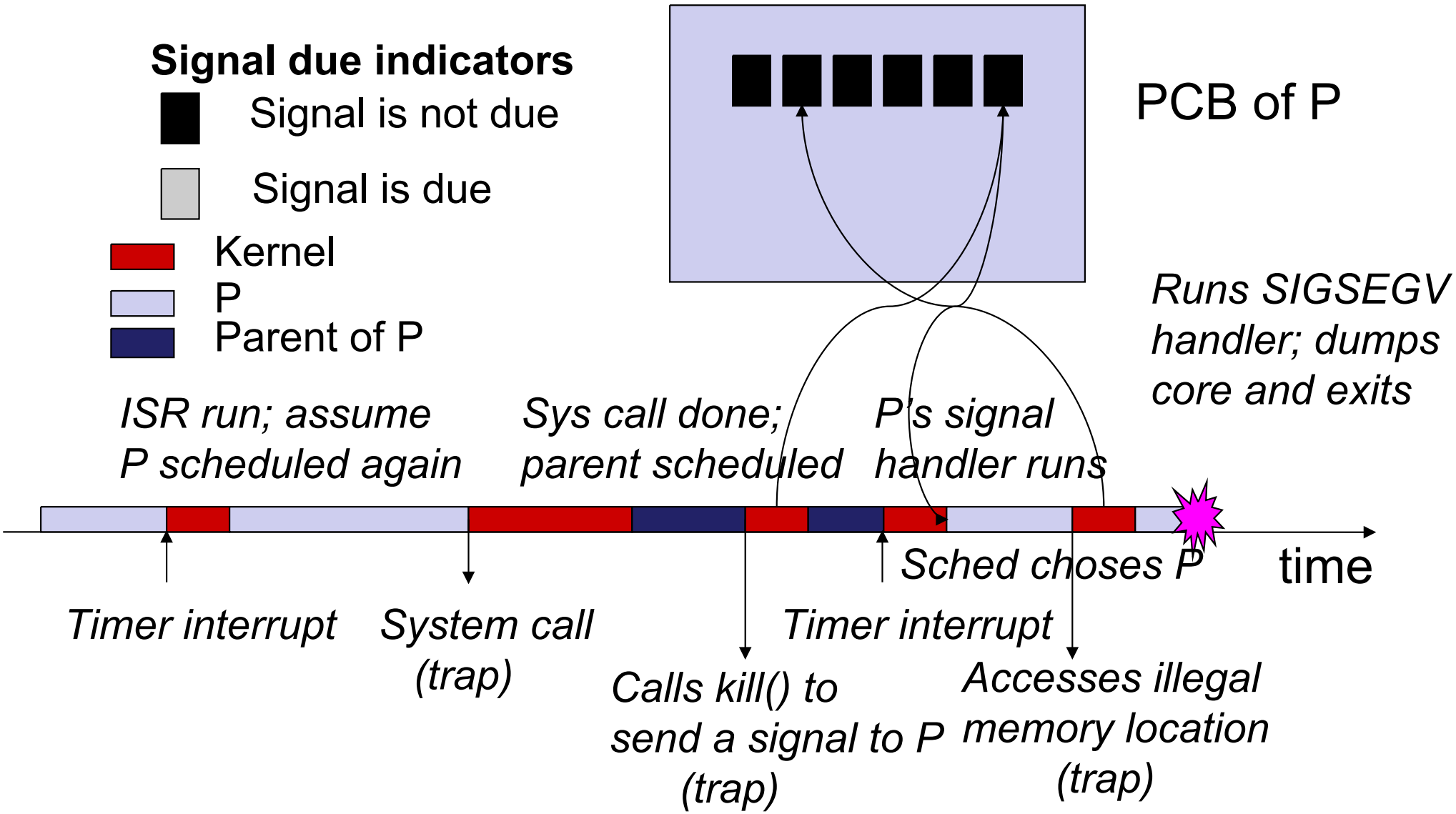
- setup_frame: sets up the user-mode (signal handler) stack
  - Forces signal handler's address into PC and some "return code"
  - After the handler is done, this return code gets executed
    - It makes a system call such as sigreturn (in Linux) that does the following:
      - 1. Restores signal pending 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

PENNSTATE

Signal Masking

- A process may request to the OS that certain signals not be delivered to it
  - Represented by a "signal mask" in the PCB
  - Analog: the OS may interrupt certain interrupts
- Interruptible vs non-interruptible interrupts and signals

Signal Handling (Visual)



PENNSTATE

Implementing User-level Threads

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

PENNSTATE

UT.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 thread library can use the same instructions as the OS to save or load the CPU state into the TCB
- Typical C functions that offer this facility
  - setcontext(), getcontext(), makecontext(), swapcontext()
  - setjmp() and longjmp()
  - sigsetjmp() and siglongjmp()

PENNSTATE

Interlude: 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
- **Let's understand them first and then see how they can help realize user-level threads**

PENNSTATE

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.

PENNSTATE

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);
}
```

PENNSTATE

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
```

PENNSTATE

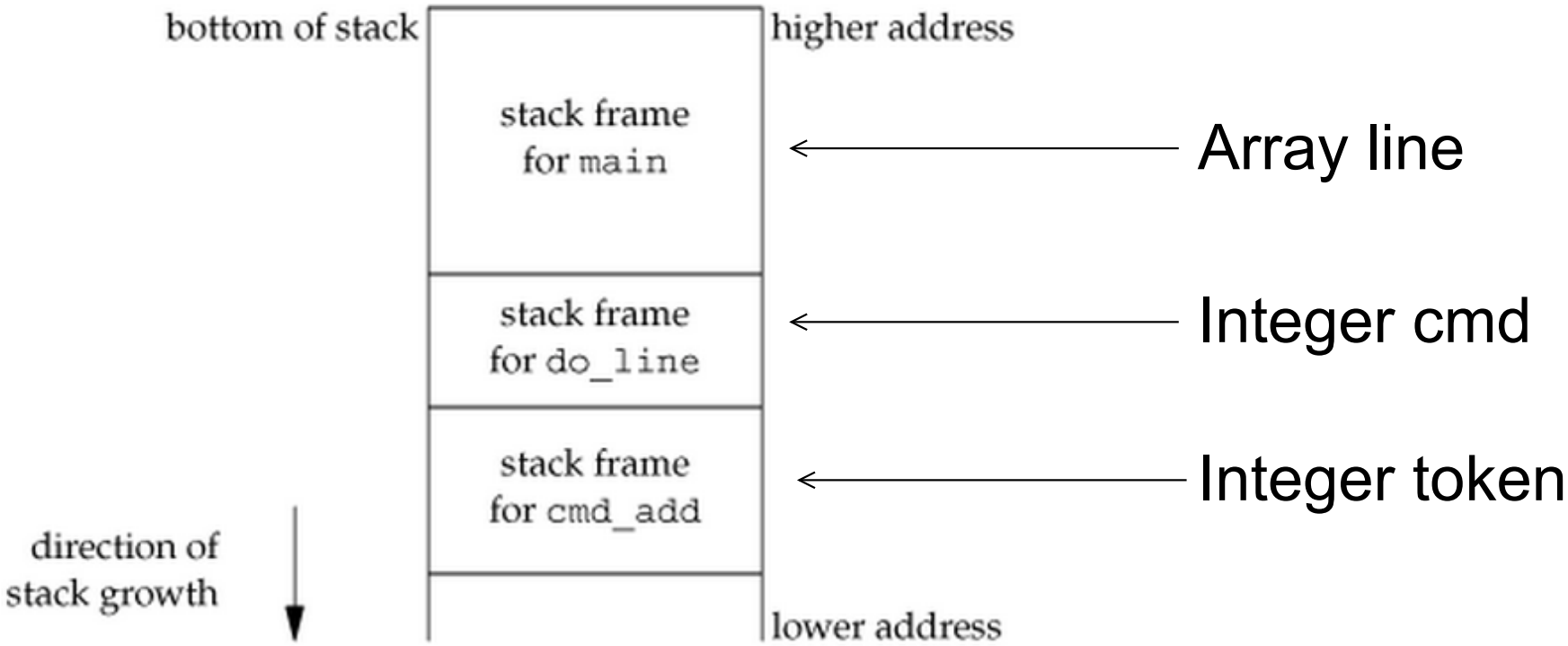
Example 2:
Deeply nested function calls

```
#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 */
}
```

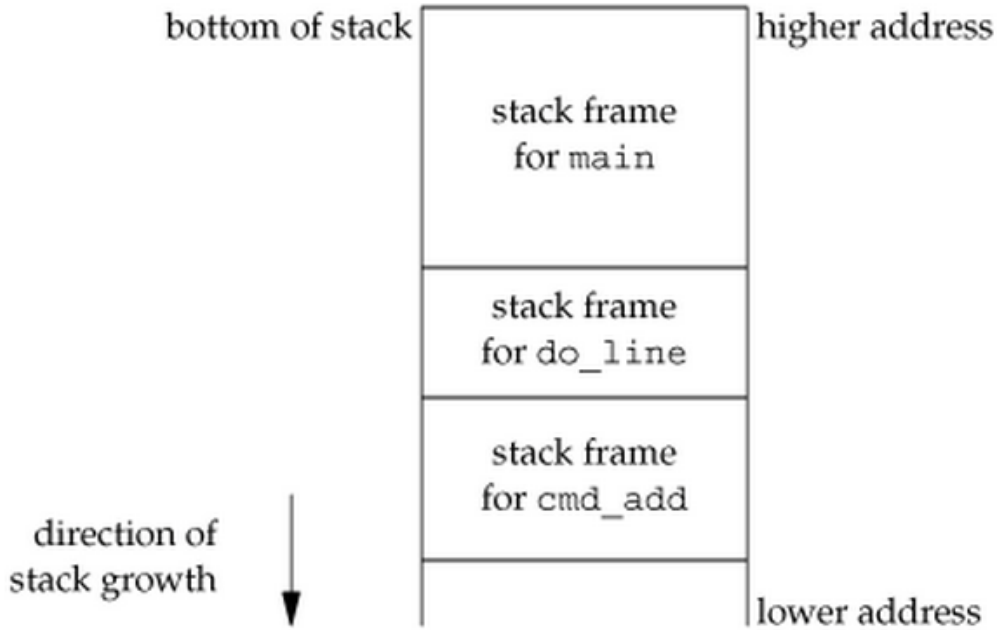
Note: do_line calls cmd_add

Example 2: Deeply-nested Function Calls

- This is what the stack may look like after cmd_add() has been called
  - Where is jmpbuffer? Local variables?



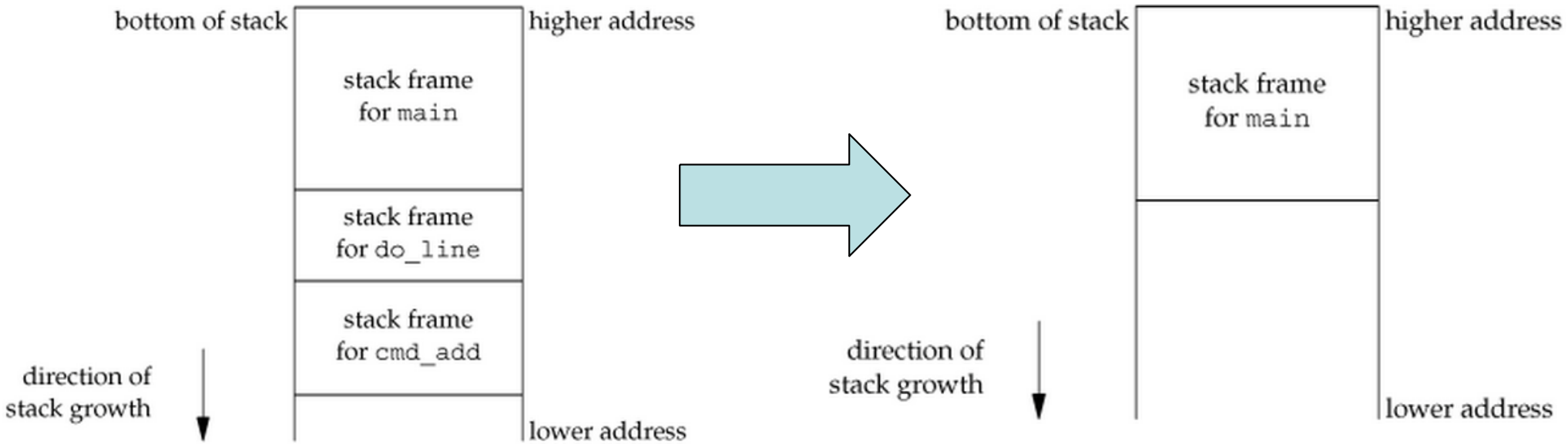
- What if when cmd_add encountered a non-fatal error, we would like to ignore the rest of the line and return to main?
  - goto would only return within cmd_add
  - Using special return value would involve going up the entire set of nested calls!
  - Would like a non-local goto: setjmp/longjmp provide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


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

PENNSTATE

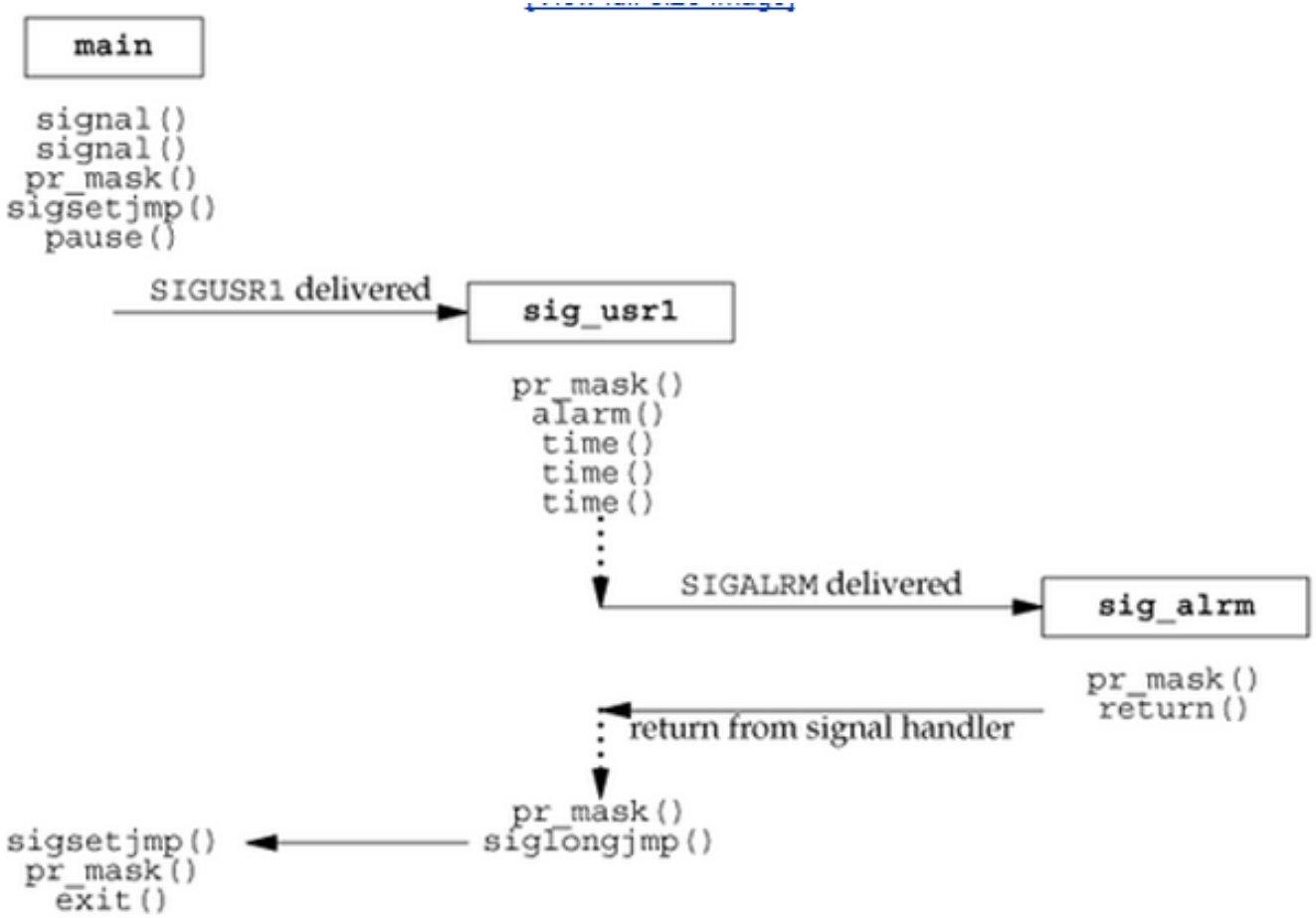
sigsetjmp() and siglongjmp()

- 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 &
```



PENNSTATE

User-level Threads: 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

PENNSTATE

Questions?

PENNSTATE

Pthread Library

- pthread_create () // create a thread
- pthread_exit () //terminate a thread
- pthread_join () // wait for a thread to exit
- pthread_self() // returns thread id
- …

Example: find max number

```
# include <pthread.h>
# define SIZE 1000
int num[SIZE];

void *find_max(void *id)
{
   int index = (int) id;
   int len = SIZE/2;
   int max = num[index * len];

   for(i=index*len+1;i<(index+1)*len;i++)
    {
        if(num[i] > max)
            max=num[i];
    }
    pthread_exit((void*)max);
 }

int main()
{
    pthread_t tid0, tid1;
    int max0,max1;
    int max;

pthread_create(&tid0,NULL,find_max,(void*)0);

pthread_create(&tid1,NULL,find_max,(void*)1);

    pthread_join(tid0,(void**)&max0);
    pthread_join(tid1,(void**)&max1);
    max = max0;
   if(max1 > max) max=max1;

   return max
}
```