Introduction to OS161

Threads

Virtual Memory

Load Exe

Outline

- Kernel threads
  - Thread library
- User processes
  - Context switch
  - Syscalls & traps
  - Address spaces and address translation
  - ELF files and exec load

What is a thread (process)?

- Represents the control state of an executing program
- Has an associated ***context*** (state)
  - Processor’s **CPU** state: program counter (PC), stack pointer, other registers, execution mode (priviledged/non-priviledged)
  - **Stack**, located in the address space of the process
- Memory
  - Program code (out of context)
  - Program data (out of context)
  - Program stack containing procedure activation records (within context)

User Threads and Kernel Threads

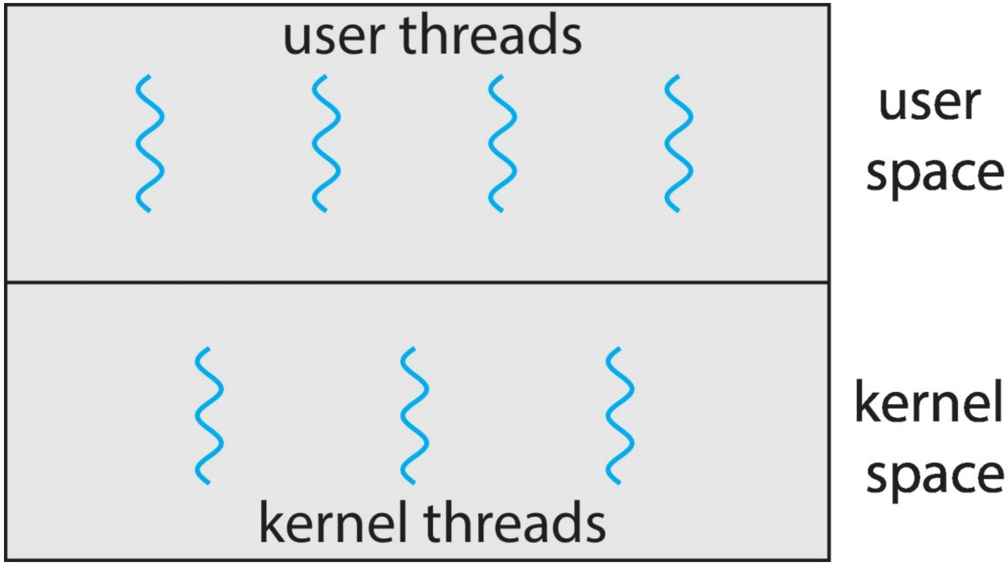
- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads

kernel threads are part of the process and user threads are threads. P threads means you have a process which means you are creating threads. User threads can be done with posix window and ava threads.

You have many multithreading models that we are not going to consider where there are different ways orchestrating threads.

- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Linux
  - Mac OS X
  - iOS
  - Android

User and Kernel Threads



Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Process Concept

- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

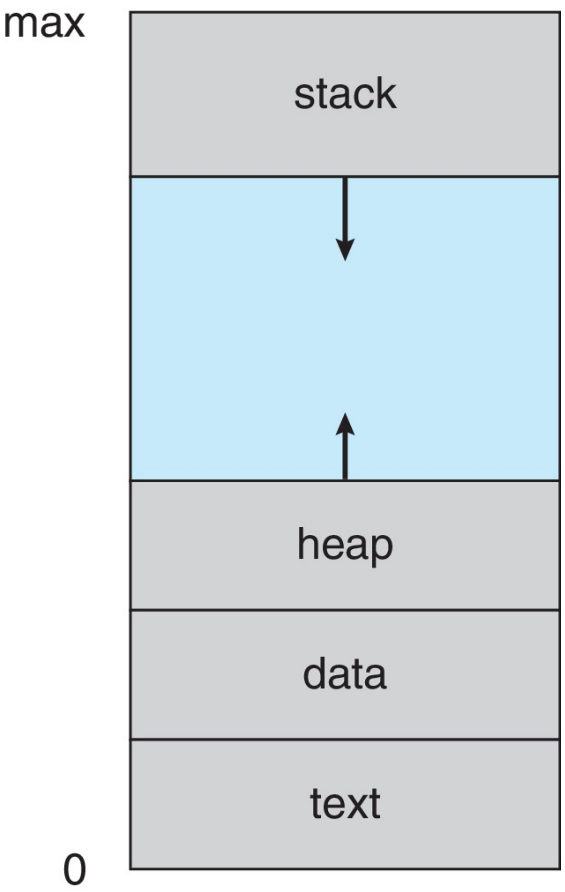
A process is something characterized by text section. Data section containing global variables.

Process Concept (Cont.)

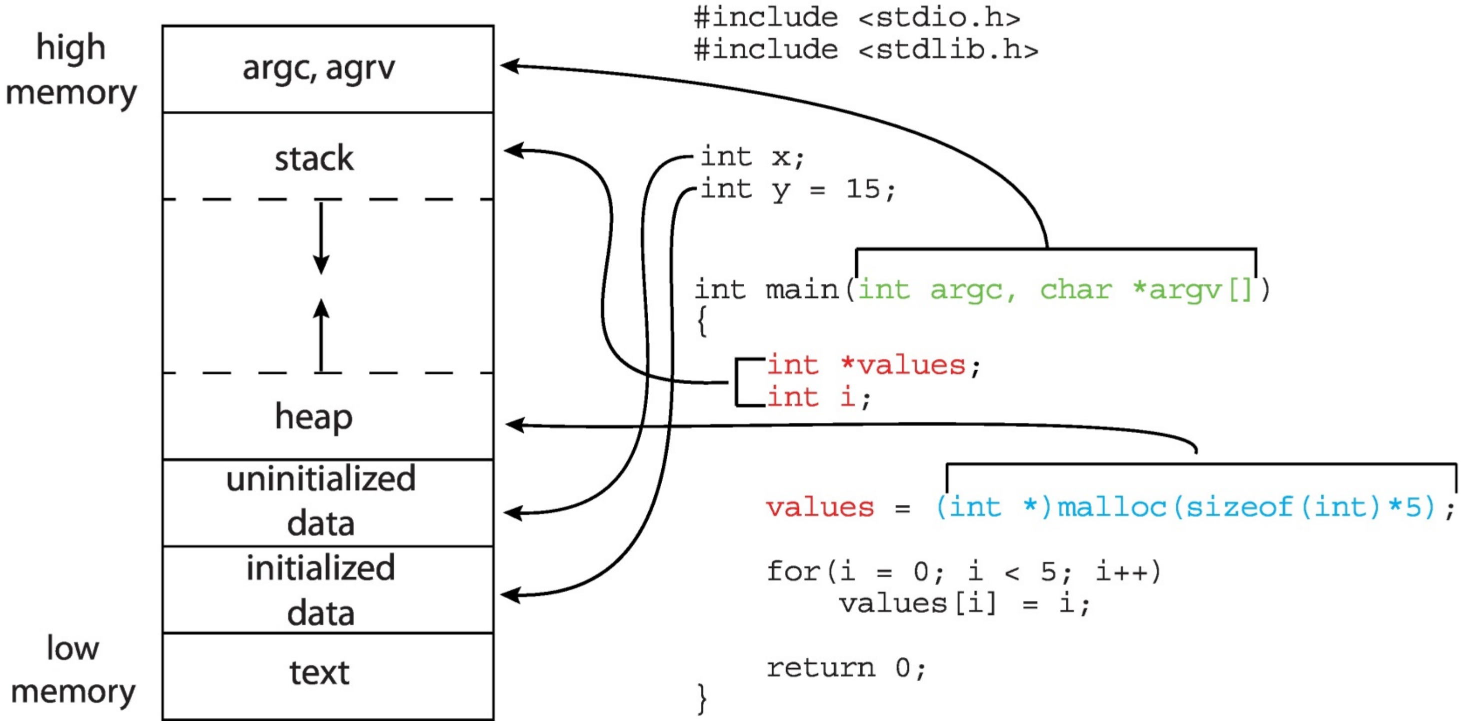
- Program is ***passive*** entity stored on disk (**executable file**); process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

Process in Memory

Heap and stack are typically located in such a way that they can grow. this is the memory layout of a c program.

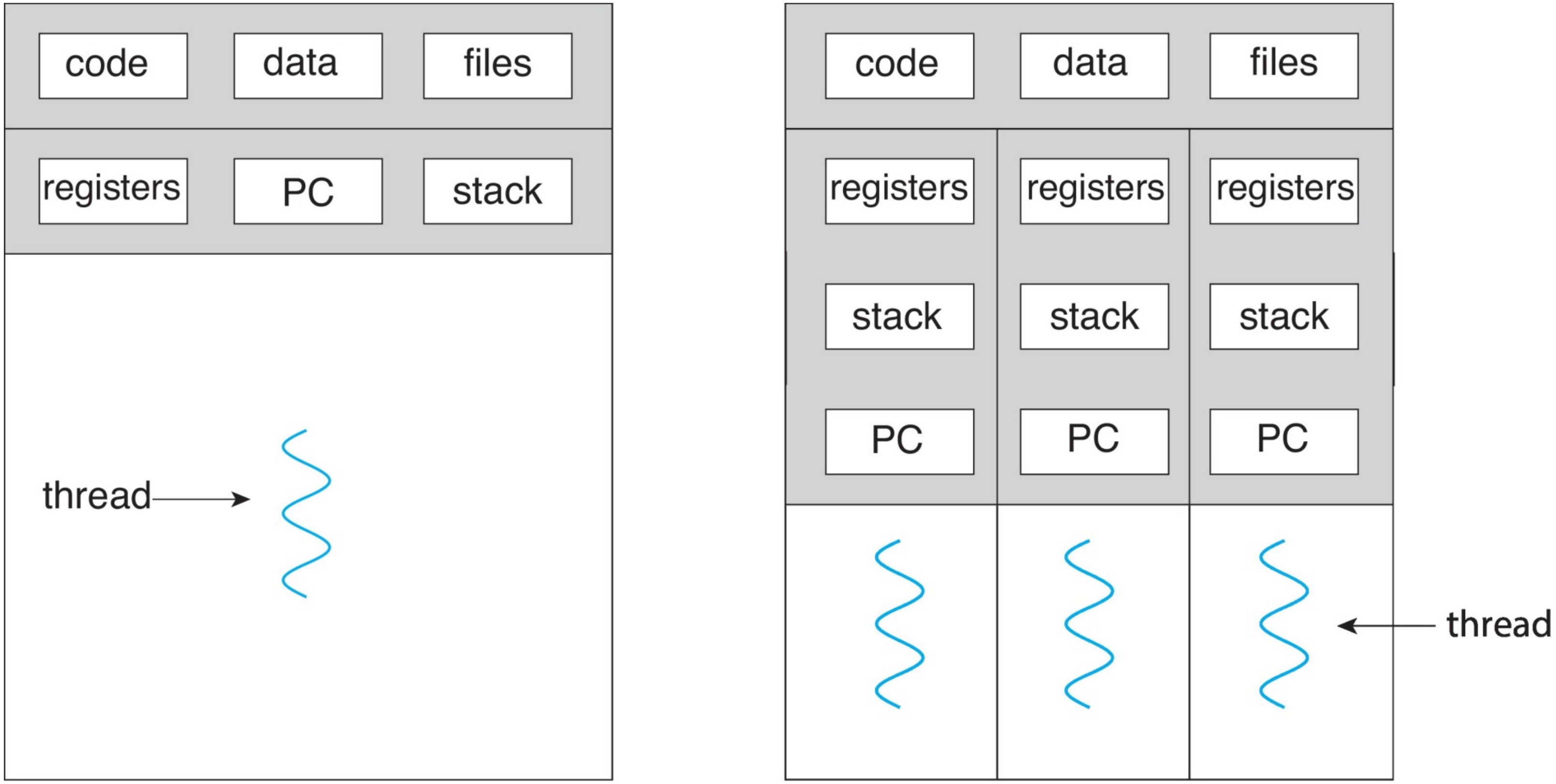


Memory Layout of a C Program



```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

Single and Multithreaded Processes

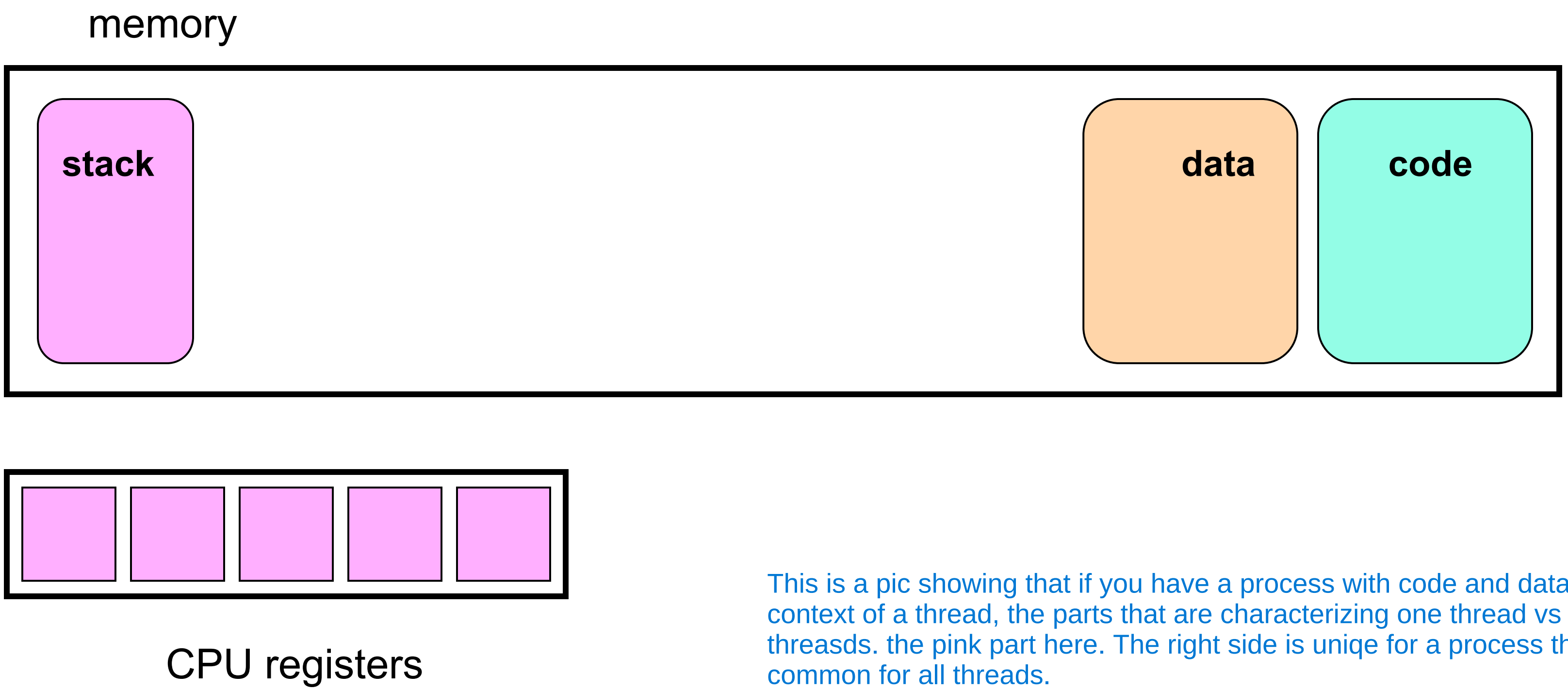


single-threaded process

multithreaded process

Program counter is the one telling you where the counter is being executed. The stack is related to the functions that we are inside. Here we see a multithreaded process and it has registers stack and pc for each of thread whereas code data and files are common which tells the story if you write concurrent software based on thread global values. global variables are common for threads in a given process whereas two different threads. There is a way to share memory. They have same code but diff global counters. Not same local variables. There are user threads and kernel thread.

Thread context



This is a pic showing that if you have a process with code and data the context of a thread, the parts that are characterizing one thread vs other threasds. the pink part here. The right side is uniqe for a process the left is common for all threads.

Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

we need thread library that's providing help for manipulating threads. So you need a library that allows to create and manipulate. two ways of implementing library for threads. something entirely in user whihc means essentially its just a matter of user program support. or you need interaction with the kernel.

Implementing threads

- Threads are implemented by a thread library
- Thread library stores threads’ contexts (or pointers to threads’ contexts) when not running
- Data structure used by thread library to store a thread context
  - Thread control block
- In OS161 thread control block is a ***thread*** structure

Threads are implemented by thread library, and in os161 there is thread control block called thread structure.

The OS161 thread Structure

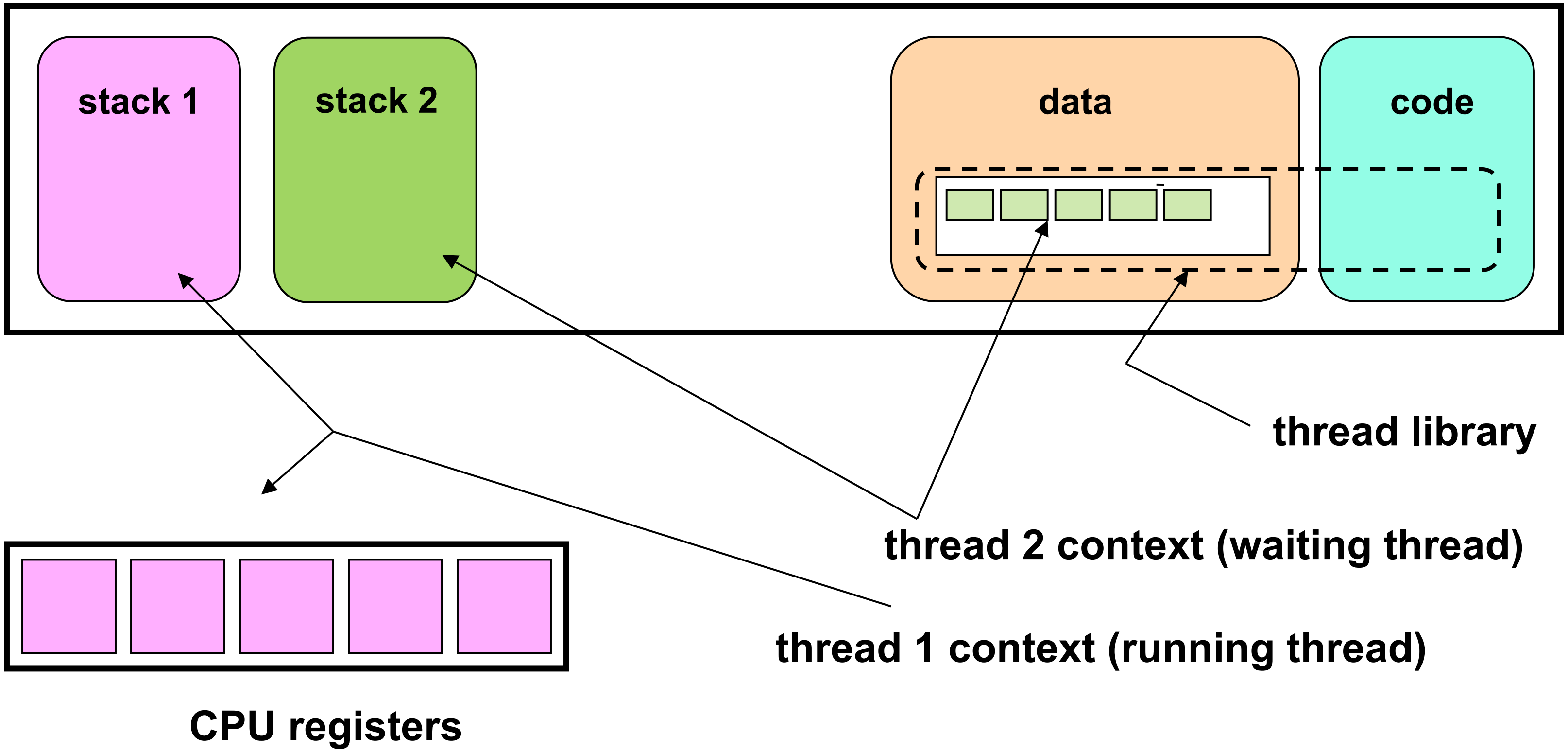
```
/* see kern/include/thread.h */

struct thread {
  char *t_name;                    /* Name of this thread */
  const char *t_wchan_name;        /* Name of wait channel, if sleeping */
  threadstate_t t_state;           /* State this thread is in */
```

here you will see where system is collecting info to handle the threads

```
  /* Thread subsystem internal fields. */
  struct thread_machdep t_machdep;
  struct threadlistnode t_listnode;
  void *t_stack;                   /* Kernel-level stack */
  struct switchframe *t_context;   /* Saved register context (on stack) */
  struct cpu *t_cpu;               /* CPU thread runs on */
  struct proc *t_proc;             /* Process thread belongs to */
  ...
};
```

This is a pointer to the context caleled switch frame , this is a pointer to where data is stored in the memory when the thread is either saved or restored by context switch.

Thread library and two threads (generic)


stack 1
stack 2
data
code
thread library
thread 2 context (waiting thread)
thread 1 context (running thread)
CPU registers

This pic shows while thread 1 is running, thread 1 iss in the cpu. so the cpu registers are loaded with data of thread 1. It is running, thread 2 in this moment is saved. The context the stack are saved. The stack is saved what does it mean the stack is already in register so doesn't have to be saved. So you could read this pic in two ways. thread 1 and thread 2 belong to different process! Process 1 with thread 1 is running, thread 2 is not running. Here we don't see code and data of stack 1. It could be either way and we don't care right now. The context of the thread could be running. The thread library is os161 has an interface.

OS161 Thread Library

- Interface for bootstrap/shutdown (or panic)

  `thread_bootstrap`

  `thread_start_cpus`

  `thread_panic`

  `thread_shutdown`

- Interface for thread handling

  `thread_fork`

  `thread_exit`

  `thread_yield`

  `thread_consider_migration`

- Internal functions

  `thread_create`

  `thread_destroy`

  `thread_make_runnable`

  `thread_switch`

Wehn the operating system boots and ends execution it is essentially doing initialization.

if you want to write a program that manipulates a thread you can call the methods.

these are functions avaliable to the kernel, since we are developing.

thread yield is to stop momentarily the execution and let another thread run.

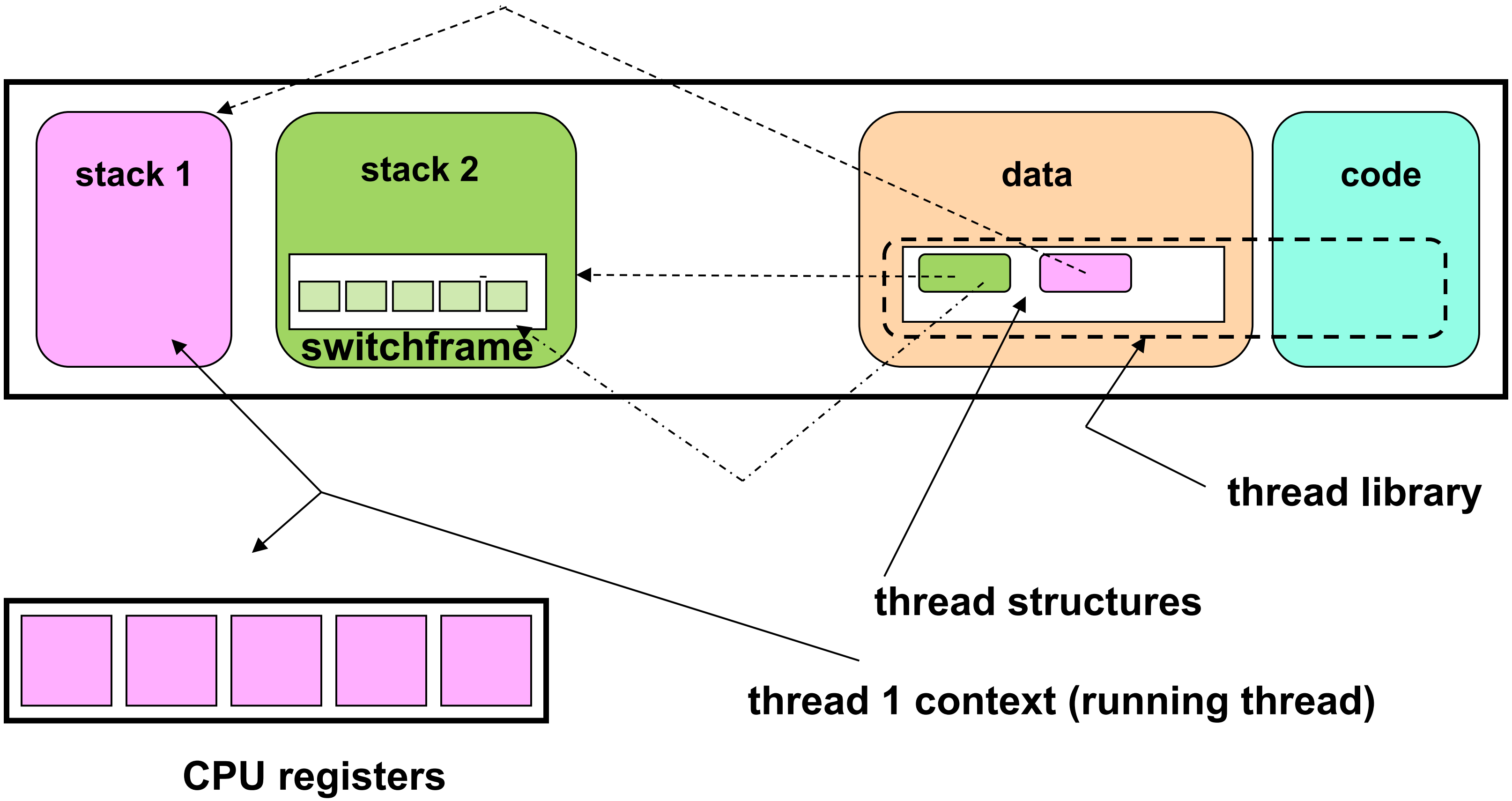
Internal func simply means functions not available directly but indirectly used by other functions.

The OS161 thread Structure

```
/* see kern/include/thread.h */

struct thread {
  char *t_name;                    /* Name of this thread */
  const char *t_wchan_name;        /* Name of wait channel, if sleeping */
  threadstate_t t_state;           /* State this thread is in */

  /* Thread subsystem internal fields. */
  struct thread_machdep t_machdep;
  struct threadlistnode t_listnode;
  void *t_stack;                   /* Kernel-level stack */
  struct switchframe *t_context;   /* Saved register context (on stack) */
  struct cpu *t_cpu;               /* CPU thread runs on */
  struct proc *t_proc;             /* Process thread belongs to */
  ...
};
```

the data structure is local and not global. Two possibilities put everything here in data. If you have this data it means that the context should be saved in a global data structure. Going with the stack is easier if you think about in the end about process with multiple thread.

Thread library and two threads (OS161)



The OS161 Thread Interface (incomplete)

Thread fork could either be used create my own process or thread new process. When you create a thread you have to essentially say. Entry point is the pointer for association of thread.

thread fork is a func that makes a new thread and when you call thread fork you have to possiblity make a new thread for the current process lke the kernel. When you run the kernel with main.k.main. As i showed you monday you can create a kernel thread that belongs to the same process. That running thread is going to create. But the same strategy will be used in os161 to create user processes.

```
/* see kern/thread/thread.c */
```

- /* Make a new thread, which will start executing at “entrypoint". The thread will belong to the process "proc", or to the current thread's process if "proc" is null. The "data" arguments (one pointer, one *number) are passed to the function. */

  ```
  int thread_fork (const char *name, struct proc *proc,
         void (*entrypoint)(void *, unsigned long),
         void *data1, unsigned long data2);
  ```

- /* Cause the current thread to exit. Interrupts need not be disabled. */

  ```
  __DEAD void thread_exit(void);
  ```

- /* Cause the current thread to yield to the next runnable thread, but itself stay runnable. Interrupts need not be disabled. */

  ```
  void thread_yield(void);
  ```

Creating Threads Using thread_fork()

this is a func that is creating a certain num of threads and this is seen in lab of week2. A func that creates kernel threads. called in two ways loud or quite. which means we have to modes of execution. So simply two different functions. In os161 we already have semaphores implemented. It is the main thread and it's waiting for certain number of these.

```
runthreads(int doloud){
  char name[16];
  int i, result;
  for (i=0; i<NTHREADS; i++) {
     snprintf(name, sizeof(name), "threadtest%d", i);
     result = thread_fork(name, NULL,
                doloud ? loudthread : quietthread, NULL, i);
     if (result) {
       panic("threadtest: thread_fork failed %s)\n",
       strerror(result));
     }
  }
  for (i=0; i<NTHREADS; i++) {
    P(tsem);
  }
}
```

From thread fork() to thread execution (ready state)

```
thread_fork(..., void (*entrypoint)(void *, unsigned long),
            void *data1, unsigned long data2) {
   ...
   newthread = thread_create(...);
   ...
   switchframe_init(newthread, entrypoint, data1, data2);
   thread_make_runnable(newthread, false);
}
thread_create(...) {
   thread = kmalloc(sizeof(*thread));
   thread->... = ...;
   return thread;
}
switchframe_init(...) {
   /* setup switchframe in stack */
}
thread_make_runnable(struct thread *target) {
   ...
   target->t_state = S_READY;
   threadlist_addtail(&targetcpu->c_runqueue, target);
   ...
}
```

How is the thread fork activating! End of lab1. put a break point on thread fork. Thread fork is calling internally. It is only allocating the thread data structure. New thread is a pointer to the thread data structure.
It is initializing a switch frame.
Thread calling thread make roundable. Thread create if you see is essentially allocating k malloc. Only datastructure management. Switchframe in the stack is preparing.
Thread make runnable is preparing the execution of the new thread from the entry point. The problem is essentially this. We are in thread fork. So the thread under execution right now .

From ready to execution: thread_switch()

```
thread_switch(threadstate_t newstate, ...) {
   struct thread *cur, *next;

   cur = curthread;
   /* Put the thread in the right place. */
   switch (newstate) {
       case S_RUN:
       panic("Illegal S_RUN in thread_switch\n");
       case S_READY:
       thread_make_runnable(cur, true /*have lock*/);
       break;
   }
   next = threadlist_remhead(&curcpu->c_runqueue);

   /* do the switch (in assembler in switch.S) */
   switchframe_switch(&cur->t_context, &next->t_context);
   ...
}
```

Kernel thread tests

from os161 menu

- `tt1`: call `threadtest->runthreads(1/*loud*/)` to generate `NTHREADS` (8) threads executing `loudthread`.
  8 threads mixing output of chars (120 chars each)
  (see `kern/test/threadtest.c`)

- `tt2`: call `threadtest2->runthreads(0/*quiet*/)` to generate `NTHREADS` (8) threads executing `quierthread`.
  8 threads doing busy wait (200000 for iterations) followed by output of 1 char (0..7)
  (see `kern/test/threadtest.c`)

- `tt3`: call `threadtest3->runtest3` to generate a certain number of threads doing sleep or work and synchronization
  (see `kern/test/tt3.c`)

From ready to execution: thread_switch()

```
thread_switch(threadstate_t newstate, ...) {
   struct thread *c

   cur = curthread;
   /* Put the threa
   switch (newstate
       case S_RUN:
       panic("Illega                                    );
       case S_READY:
       thread_make_runnable(c            *have lock*/);
       break;
   }
   next = threadlist_remhead(&    cpu->c_runqueue);

   /* do the switch (in assembler in switch.S) */
   switchframe_switch(&cur->t_context, &next->t_context);
   ...
}
```

Assembler because switch done on **registers**

```
cur->t_context = registers; /* save */
registers = next->t_context;/* restore */
```

Review: MIPS Register Usage

See also: kern/arch/mips/include/kern/regdefs.h

```
R0, zero = ## zero (always returns 0)↰
R1, at = ## reserved for use by assembler
R2, v0 = ## return value / system call number
R3, v1 = ## return value
R4, a0 = ## 1st argument (to subroutine)↰
R5, a1 = ## 2nd argument
R6, a2 = ## 3rd argument
R7, a3 = ## 4th argument
```

Review: MIPS Register Usage

```
R08-R15, t0-t7 = ## temps (not preserved by subroutines)↰
R24-R25, t8-t9 = ## temps (not preserved by subroutines)↰
                 ## can be used without saving
R16-R23, s0-s7 = ## preserved by subroutines
                 ## save before using,
                 ## restore before return
R26-27,  k0-k1 = ## reserved for interrupt handler
R28,     gp =    ## global pointer
                 ## (for easy access to some variables)↰
R29,     sp =    ## stack pointer
R30,     s8/fp   ## 9th subroutine reg / frame pointer
R31,     ra =    ## return addr (used by jal)↰
```

Dispatching on the MIPS (1 of 2)

```
/* see kern/thread/thread.c */
thread_switch(threadstate_t newstate, ...) {
   ...
   /* do the switch (in assembler in switch.S) */
   switchframe_switch(&cur->t_context, &next->t_context);
}

/* see kern/arch/mips/thread/switch.S */
switchframe_switch:
   /* a0/a1 point to old/new thread’s switchframe (control block) */
   /* Allocate stack space for saving 10 registers. 10*4 = 40 */
   addi sp, sp, -44

   /* Save the registers */
   sw ra, 36(sp)
   sw gp, 32(sp)
   sw s8, 28(sp)
   ...
   sw s1, 4(sp)
   sw s0, 0(sp)
   /* Store the old stack pointer in the old thread */
   sw sp, 0(a0) /* cur->t_context = s0; */
```

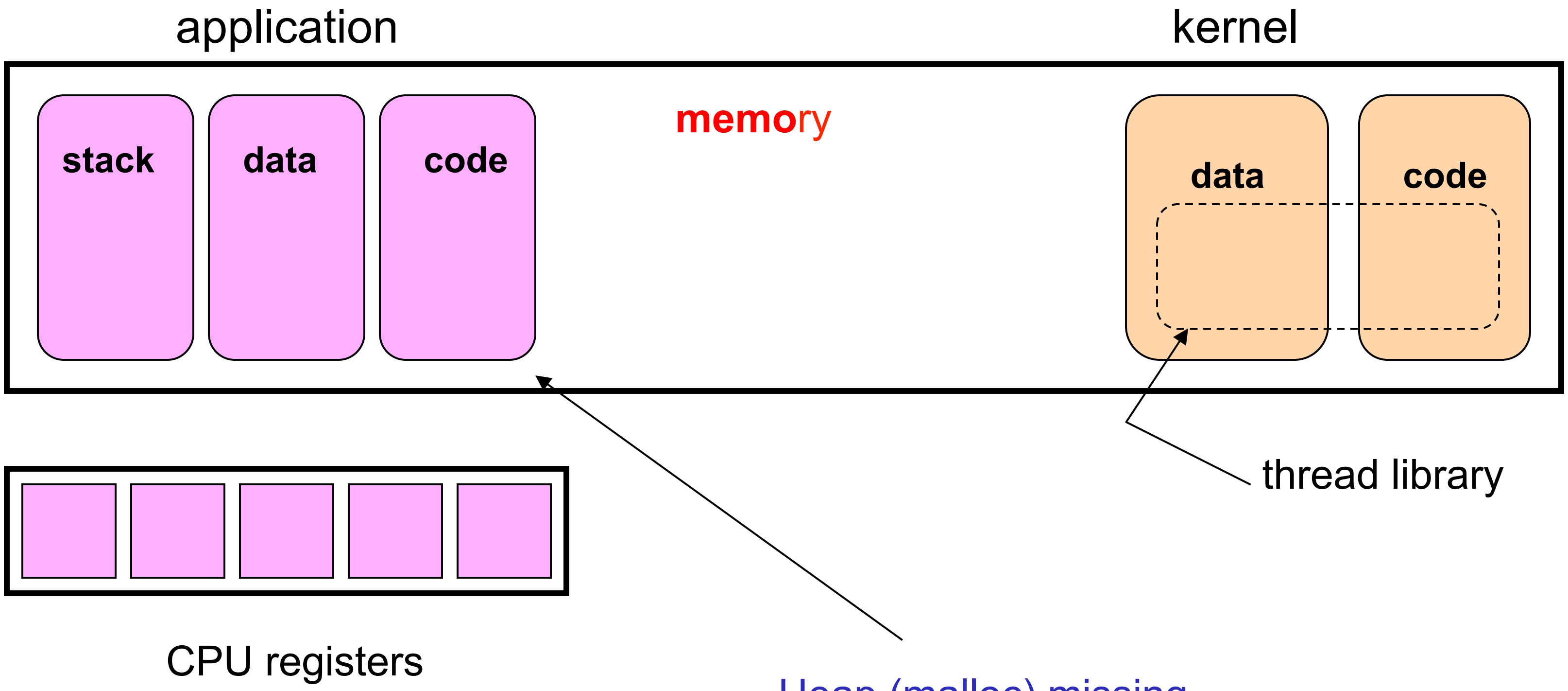
Dispatching on the MIPS (1 of 2)

```
/* see kern/thread/thread.c */
thread_switch(threadstate_t newstate, ...) {
   ...
   /* do the switch (in assembler in switch.S) */
   switchframe_switch(&cur->t_context, &next->t_context);
}

/* see kern/arch/mips/thread/switch.S */
switchframe_switch:
   /* a0/a1 point to old/new thread's switchframe (control block) */
   /* Allocate stack space for saving 10 registers. 10*4 = 40 */
   addi sp, sp, -44

   /* Save the registers */
   sw ra, 36(sp)
   sw gp, 32(sp)
   sw s8, 28(sp)
   ...
   sw s1, 4(sp)
   sw s0, 0(sp)
   /* Store the old stack pointer in the old thread */
   sw sp, 0(a0) /* cur->t_context = sp; */⌉  *a0 = sp;
```

Dispatching on the MIPS (2 of 2)

```
/* Get the new stack pointer from the new thread */
lw sp, 0(a1) /* sp = next->t_context; */

nop /* delay slot for load */

/* Now, restore the registers */
lw s0, 0(sp)
lw s1, 4(sp)
lw s2, 8(sp)
lw s3, 12(sp)
lw s4, 16(sp)
lw s5, 20(sp)
lw s6, 24(sp)
lw s8, 28(sp)
lw gp, 32(sp)
lw ra, 36(sp)
nop /* delay slot for load */

j ra /* and return. */
addi sp, sp, 40 /* in delay slot */
.end switchframe_switch
```

Application (User process) and Kernel



thread library

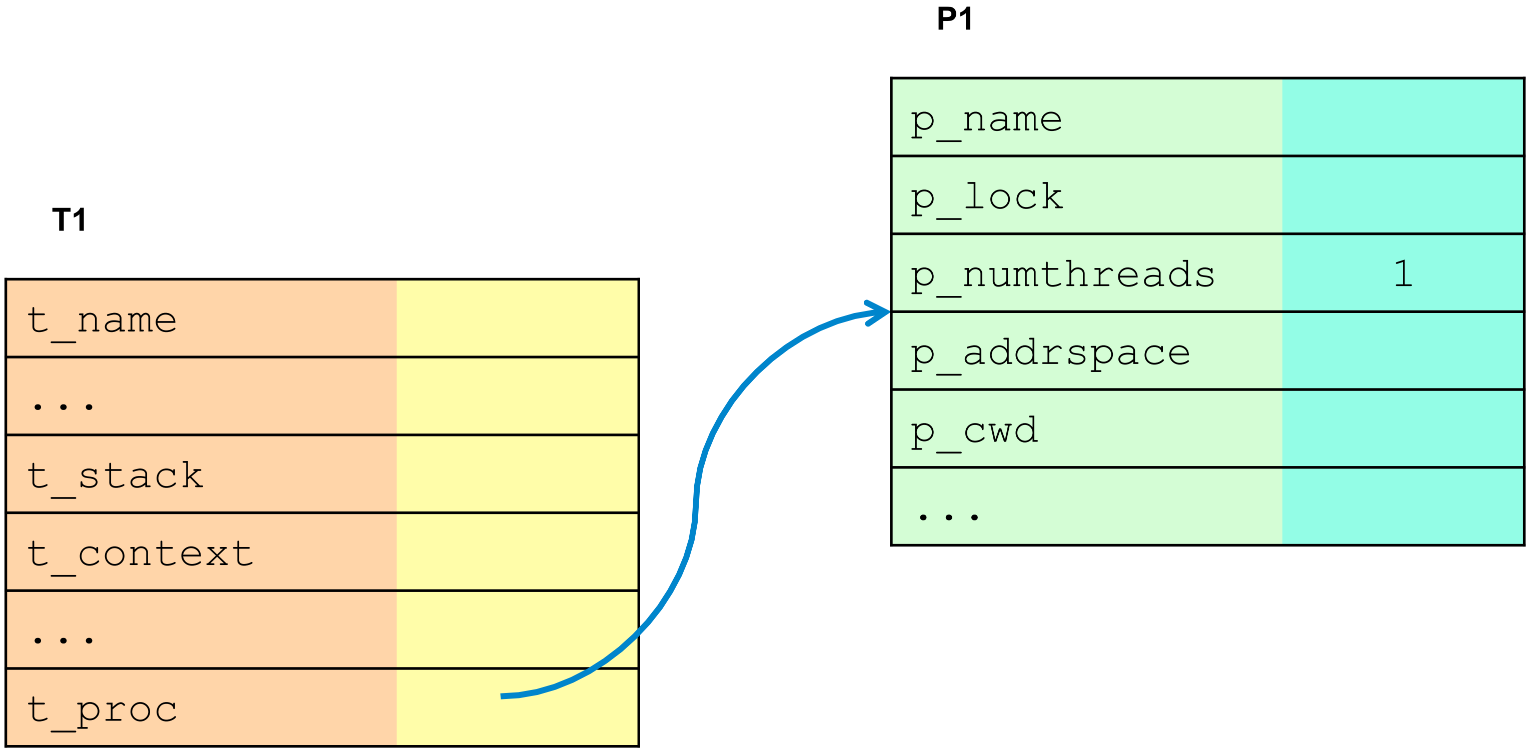
Heap (malloc) missing
TODO

The OS161 proc Structure
(PCB: Process Control Block)

```
/* see kern/include/proc.h */

struct proc {
  char *p_name;                   /* Name of this process */
  struct spinlock p_lock;         /* Lock for this structure */
  unsigned p_numthreads;          /* Number of threads in this process */

  /* VM */
  struct addrspace *p_addrspace;  /* virtual address space */

  /* VFS */
  struct vnode *p_cwd;            /* current working directory */

  /* add more material here as needed */
  ...
};
```

Single threaded Process

```
struct proc P1;
struct thread T1;
```



T1

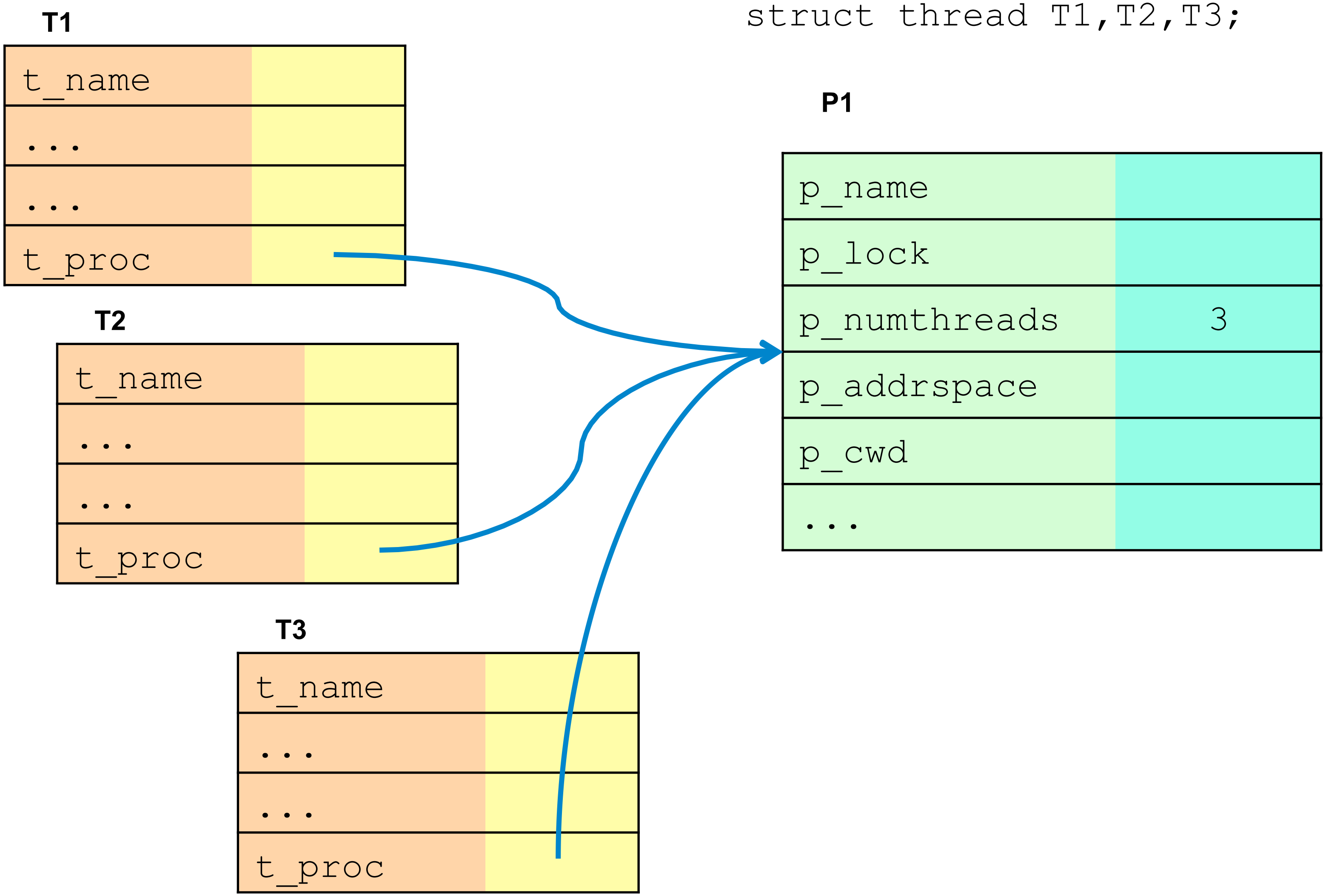
t_name	
...	
t_stack	
t_context	
...	
t_proc	

P1

p_name	
p_lock	
p_numthreads	1
p_addrspace	
p_cwd	
...	

Multi-threaded Process

```
struct proc P1;
struct thread T1,T2,T3;
```



T1

t_name
...
...
t_proc

T2

t_name
...
...
t_proc

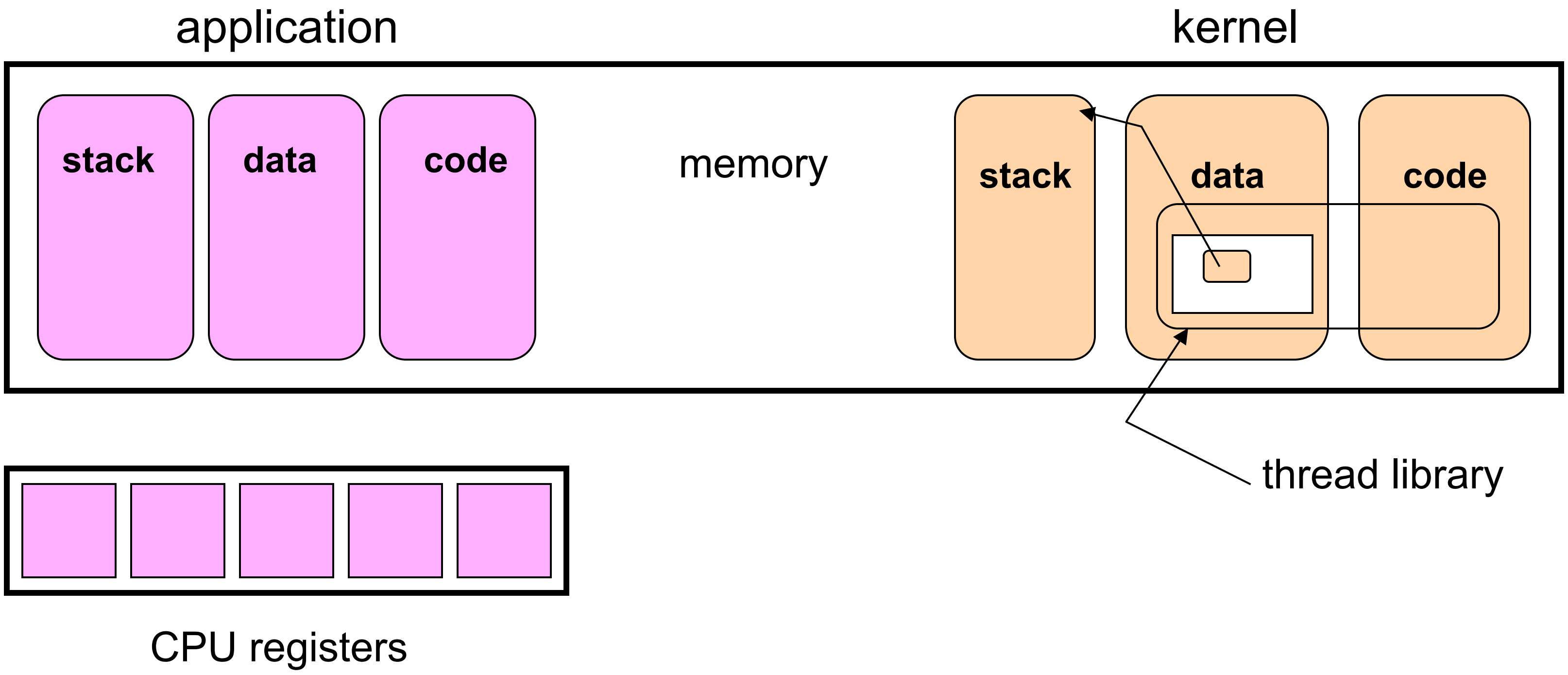
T3

t_name
...
...
t_proc

P1

p_name
p_lock
p_numthreads 3
p_addrspace
p_cwd
...

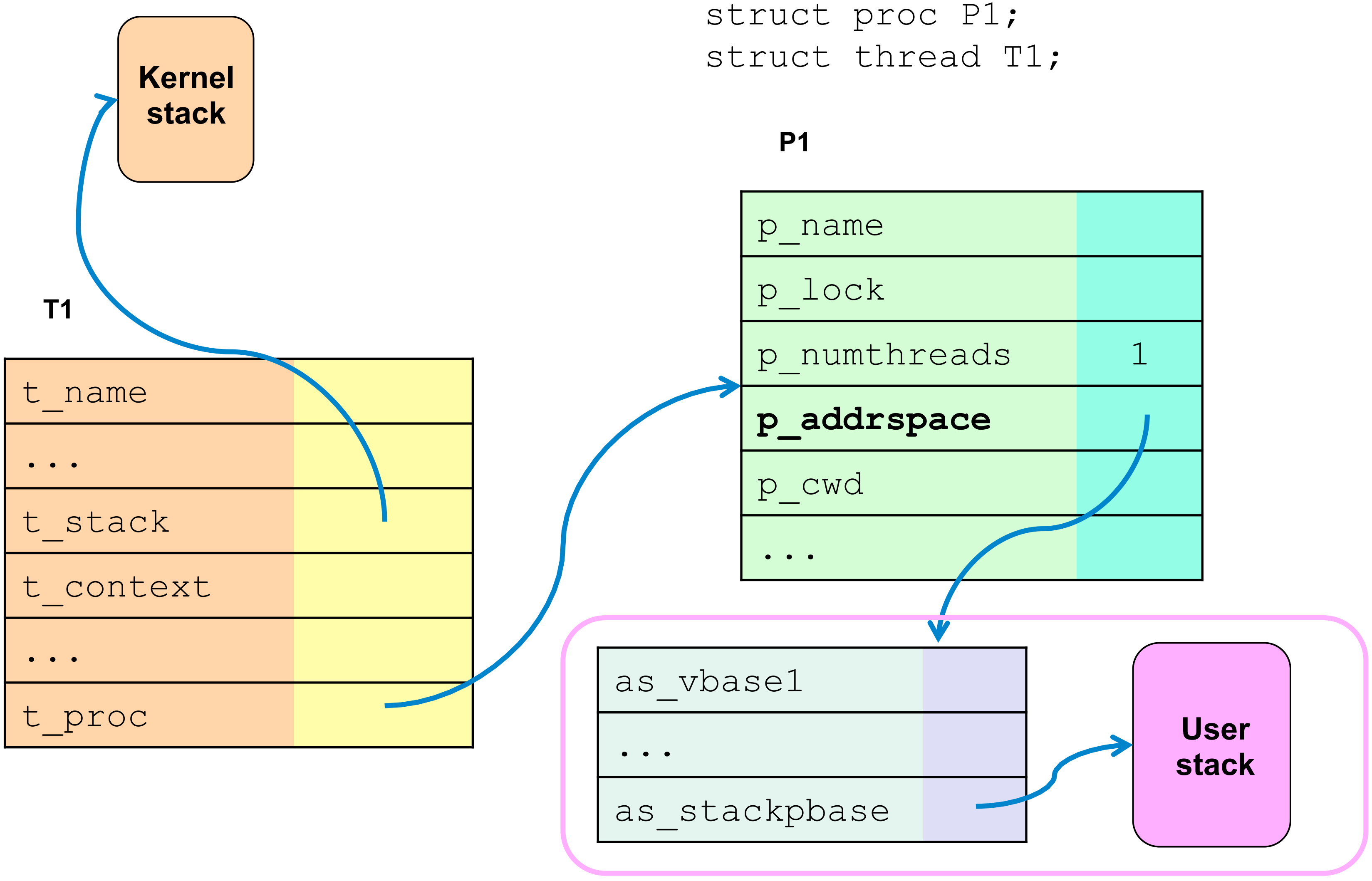
OS161 User and Kernel Thread Stacks



Each OS161 thread has two stacks, one that is used while the thread is executing unprivileged application code, and another that is used while the thread is executing privileged kernel code.

Process Stack(s)

```
struct proc P1;
struct thread T1;
```



Running a user program

(GDB: set breakpoint on load_elf)
from os161 menu

- `p <elf_file> {<args>}`:
    - p bin/cat <filename>
    - p testbin/palin
- Menu calls `cmd_prog->common_prog`
    - `proc_create_runprogram`: create user process
    - `thread_fork:`
      thread executes `cmd_progthread->runprogram`
        - Generate address space
        - Read ELF file
        - Enter new process (kernel thread becomes USER thread)

runprogram (running a user program)

```
/* see kern/syscall/runprogram.c */
int runprogram(char *progname) {
  struct addrspace *as;
  struct vnode *v;
  vaddr_t entrypoint, stackptr;
  int result;

  /* Open the file. */
  result = vfs_open(progname, O_RDONLY, 0, &v);
  ...
  /* Create a new address space. */
  as = as_create();
  ...
  /* Switch to it and activate it. */
  proc_setas(as);
  as_activate();
```

runprogram

```
  /* Load the executable. */
  result = load_elf(v, &entrypoint);
  ...
  /* Done with the file now. */
  vfs_close(v);

  /* Define the user stack in the address space */
  result = as_define_stack(as, &stackptr);
  ...
  /* Warp to user mode. */
  enter_new_process(0/*argc*/, NULL/*userspace addr of argv*/,
            NULL /*userspace addr of environment*/,
            stackptr, entrypoint);
  /* enter_new_process does not return. */
  panic("enter_new_process returned\n");
  return EINVAL;
}
```

enter_new_process

```
/* see kern/arch/mips/locore/trap.c */
void enter_new_process(int argc, userptr_t argv, userptr_t env,
          vaddr_t stack, vaddr_t entry) {
    struct trapframe tf;

    bzero(&tf, sizeof(tf));

    tf.tf_status = CST_IRQMASK | CST_IEp | CST_KUp;
    tf.tf_epc = entry;
    tf.tf_a0 = argc;
    tf.tf_a1 = (vaddr_t)argv;
    tf.tf_a2 = (vaddr_t)env;
    tf.tf_sp = stack;

    mips_usermode(&tf);
}
void mips_usermode(struct trapframe *tf) {
   ...
    /* This actually does it. See exception-*.S. */
    asm_usermode(tf);
}
```

enter_new_process

```
/* see kern/arch/mips/locore/trap.c */
void enter_new_process(int argc, userptr_t argv, userptr_t env,
          vaddr_t stack, vaddr_t entry) {
    struct trapframe tf;

    bzero(&tf, sizeof(tf))

    tf.tf_status = CST_IRQ
    tf.tf_epc = entry;
    tf.tf_a0 = argc;
    tf.tf_a1 = (vaddr_t)argv;
    tf.tf_a2 = (vaddr_t)env;
    tf.tf_sp = stack;

    mips_usermode(&tf);
}
void mips_usermode(struct trapframe *tfp) {
   ...
    /* This actually does it. See exception-*.S. */
    asm_usermode(tfp);
}
```

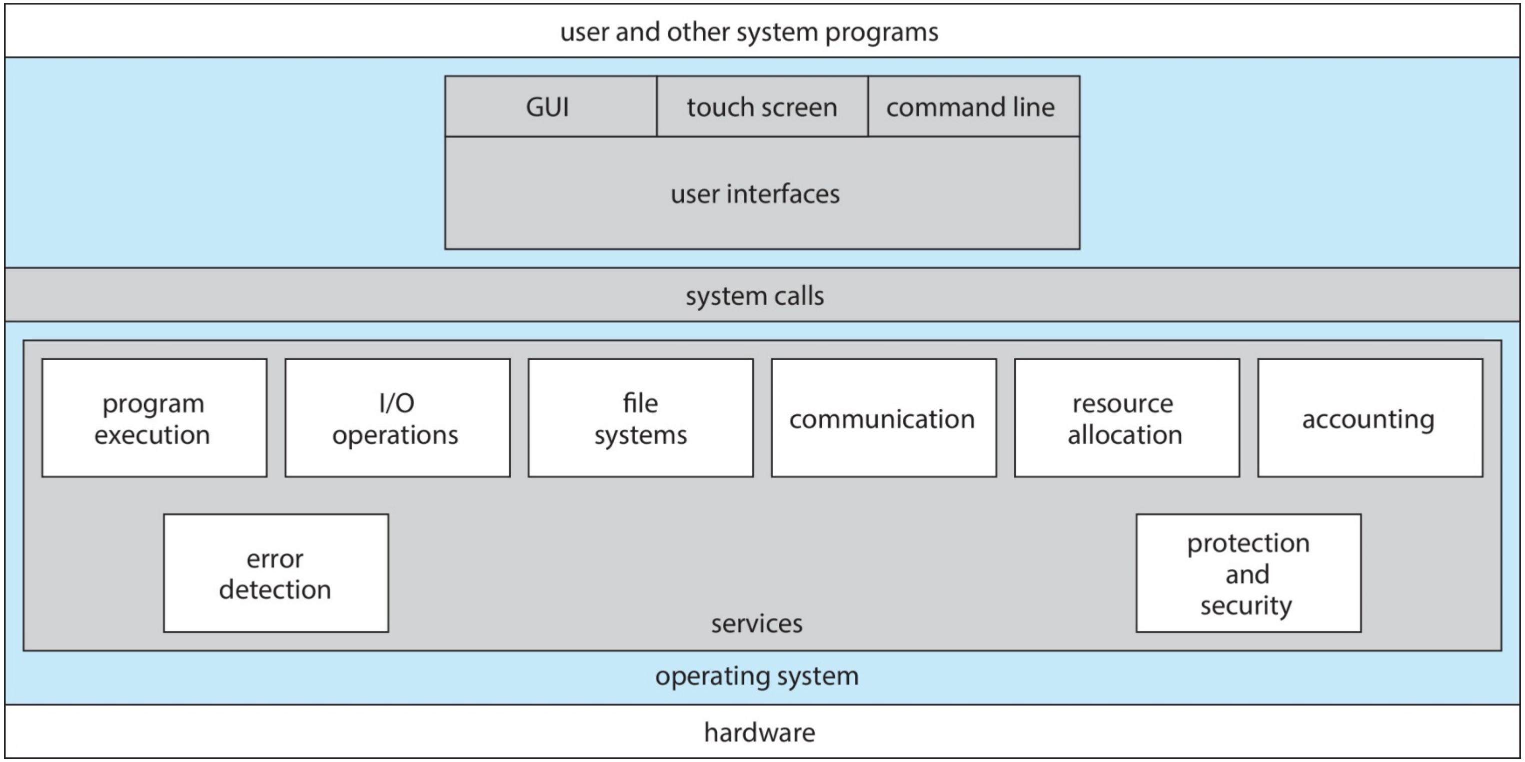
Assembler because working on **registers**
Usermode done as if returning from exception
(restoring user mode)

```
registers = tf;
return from exception;
```

asm_usermode on the MIPS

```
/* see kern/arch/mips/locore/exception-mips1.S */
asm_usermode:
   /* a0 is the address of a trapframe to use for exception "return".
    * It's allocated on our stack.
    * Move it to the stack pointer - we don't need the actual stack
    * position any more. (When we come back from usermode, cpustacks[]
    * will be used to reinitialize our stack pointer, and that was
    * set by mips_usermode.)
    * Then just jump to the exception return code above.
    */
   j exception_return
   addiu sp, a0, -16              /* in delay slot */
   .end asm_usermode

exception_return:
   ... /* restore registers from trapframe */
   /* done */
   jr k0  /* jump (register) back */
   rfe    /* in delay slot: return from exception - resume user mode (if
   needed) */
   .end common_exception
```

A View of Operating System Services



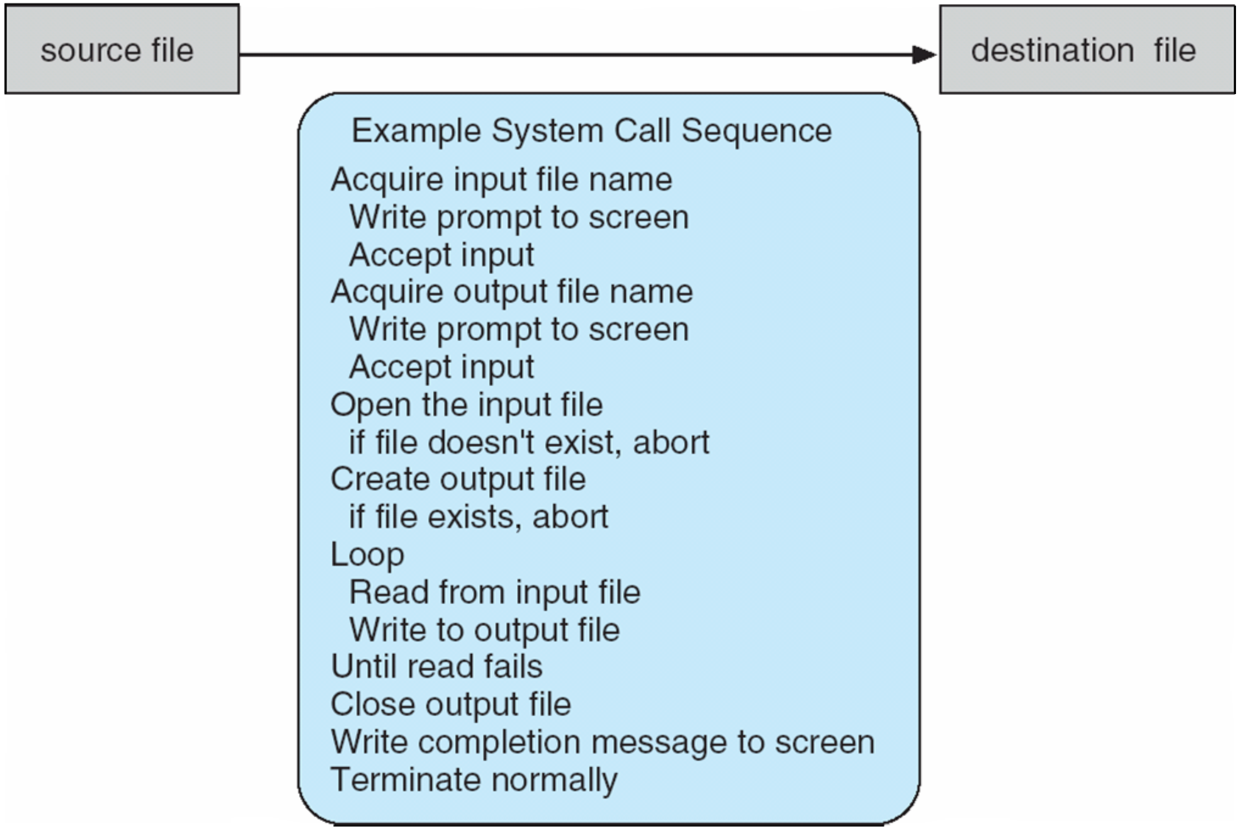
System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

Note that the system-call names used throughout this text are generic

Example of System Calls

- System call sequence to copy the contents of one file to another file



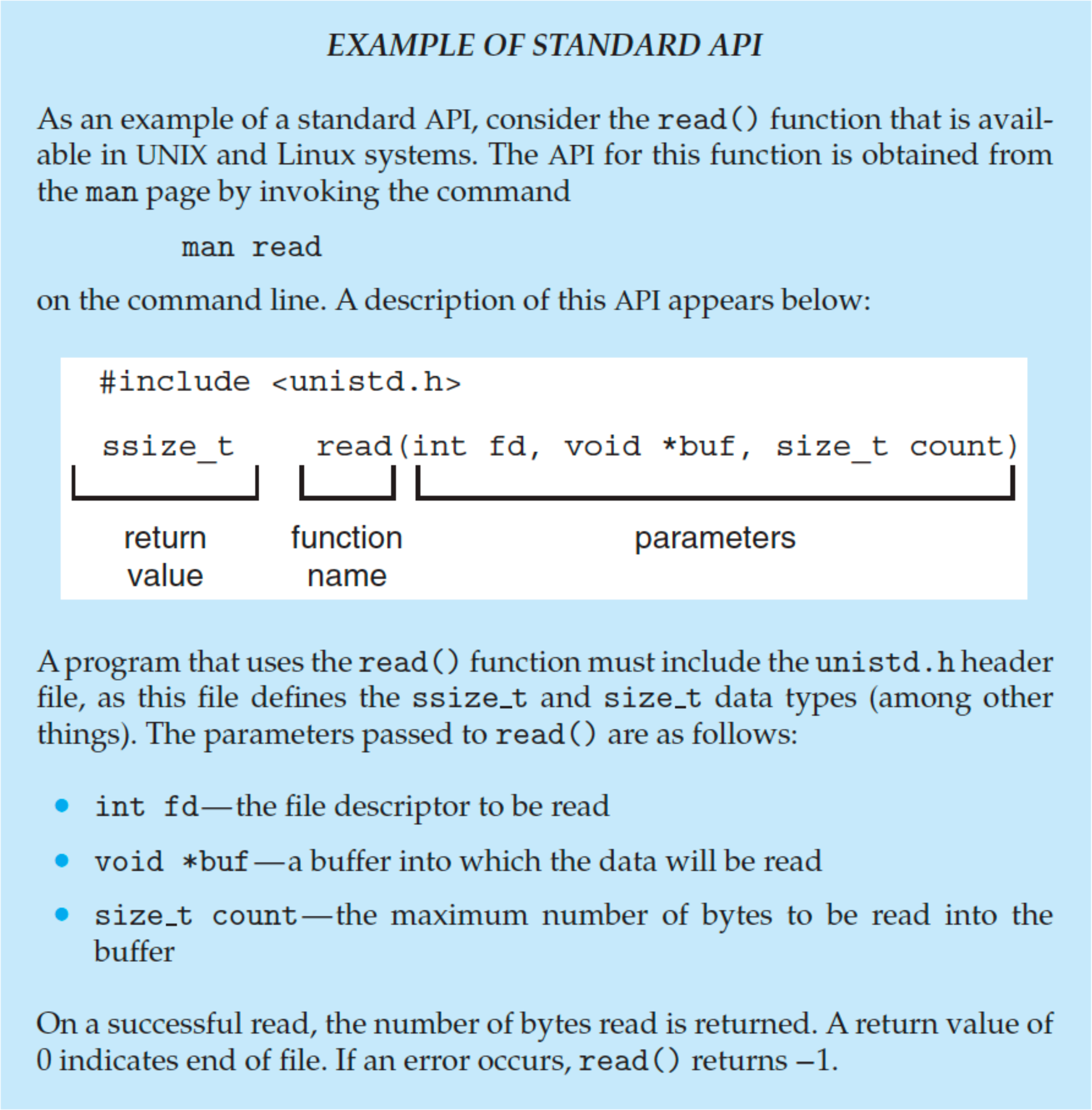
source file

destination file

Example System Call Sequence

```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

Example of Standard API



EXAMPLE OF STANDARD API

As an example of a standard API, consider the read() function that is available in UNIX and Linux systems. The API for this function is obtained from the man page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

return value

function name

parameters

A program that uses the read() function must include the unistd.h header file, as this file defines the ssize_t and size_t data types (among other things). The parameters passed to read() are as follows:

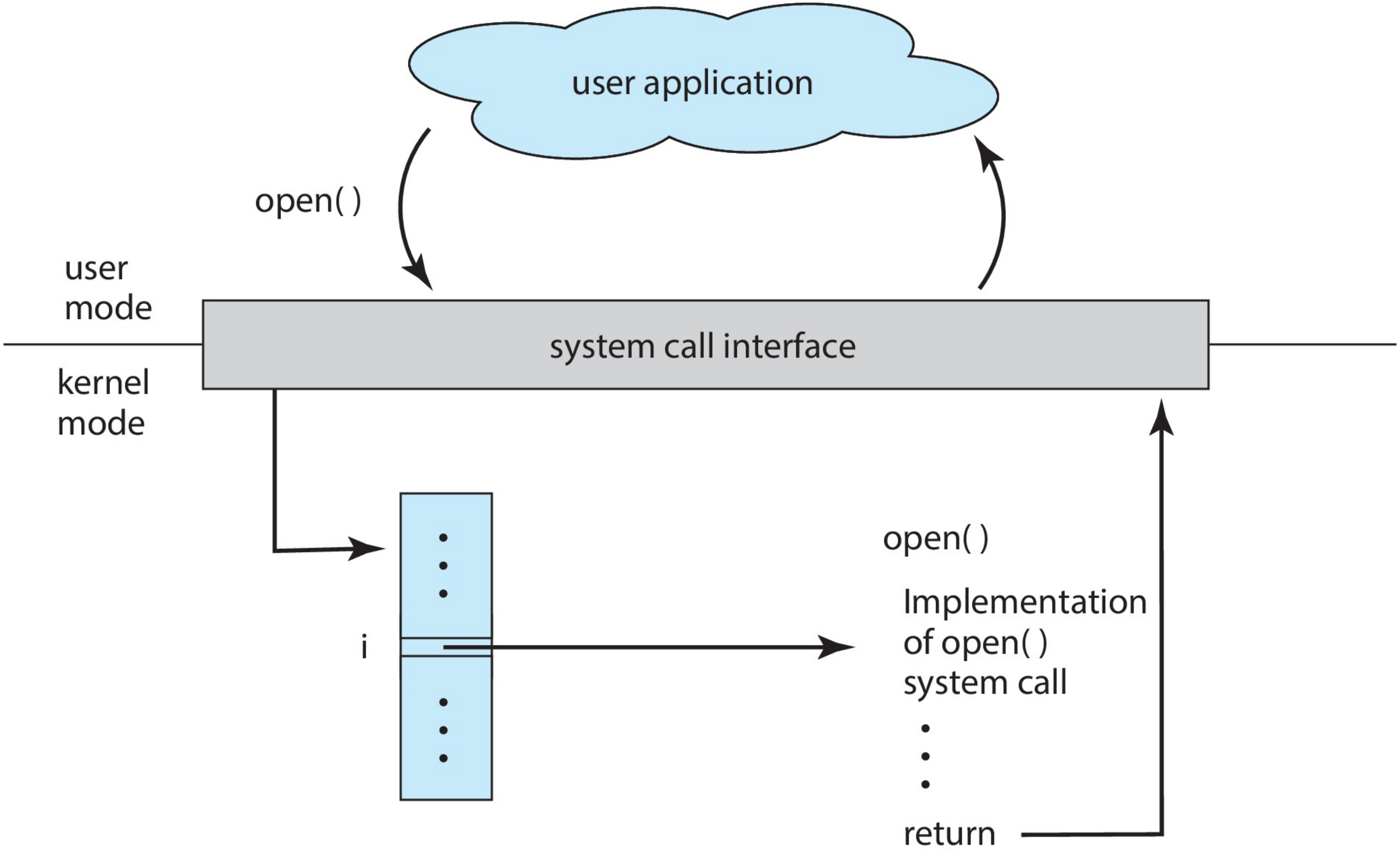
- int fd—the file descriptor to be read
- void *buf—a buffer into which the data will be read
- size_t count—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, read() returns −1.

System Call Implementation

- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

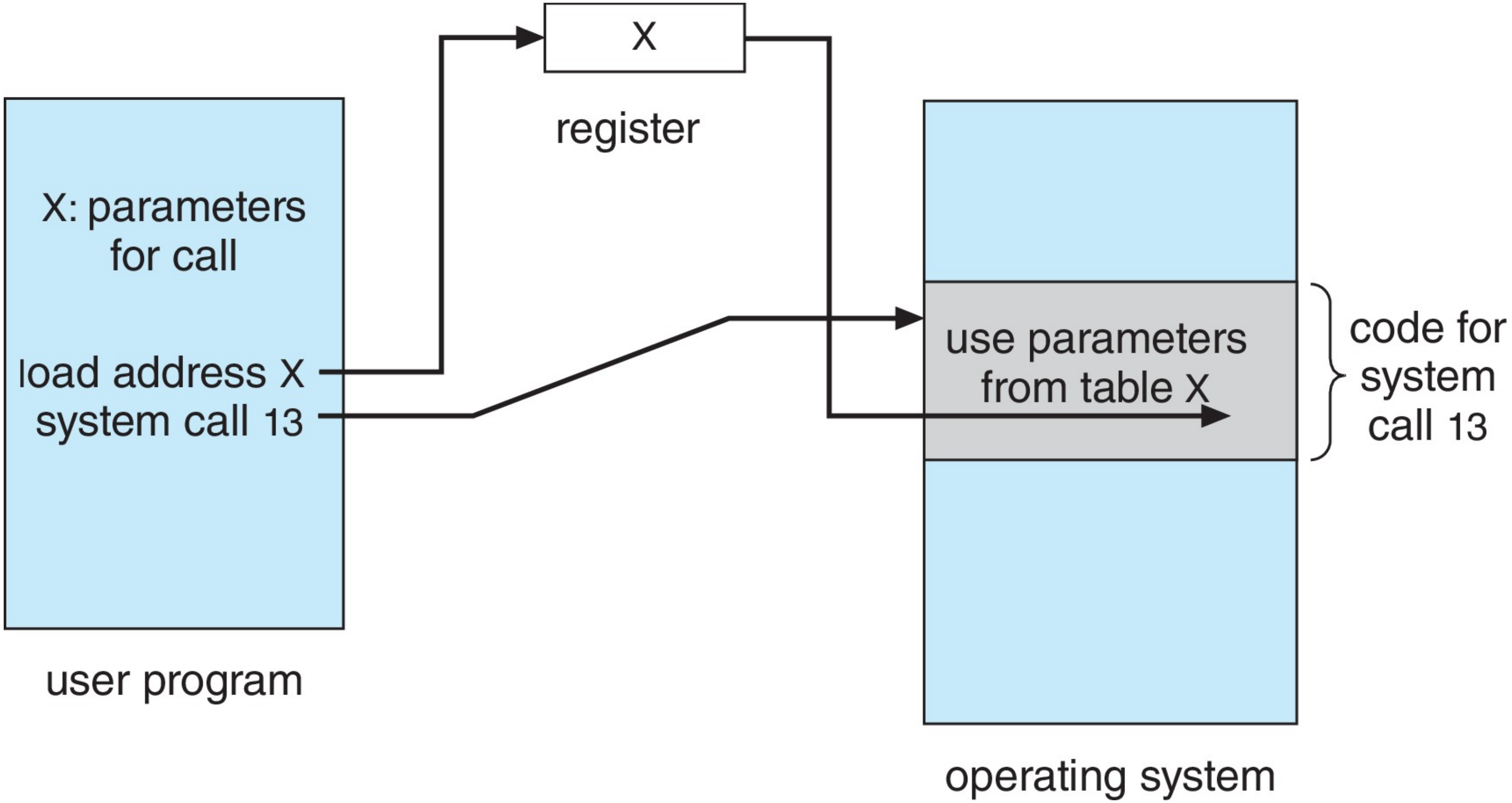
API – System Call – OS Relationship



System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
    - In some cases, may be more parameters than registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table



Types of System Calls

- Process control
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
  - **Debugger** for determining **bugs, single step** execution
  - **Locks** for managing access to shared data between processes

Types of System Calls (cont.)

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

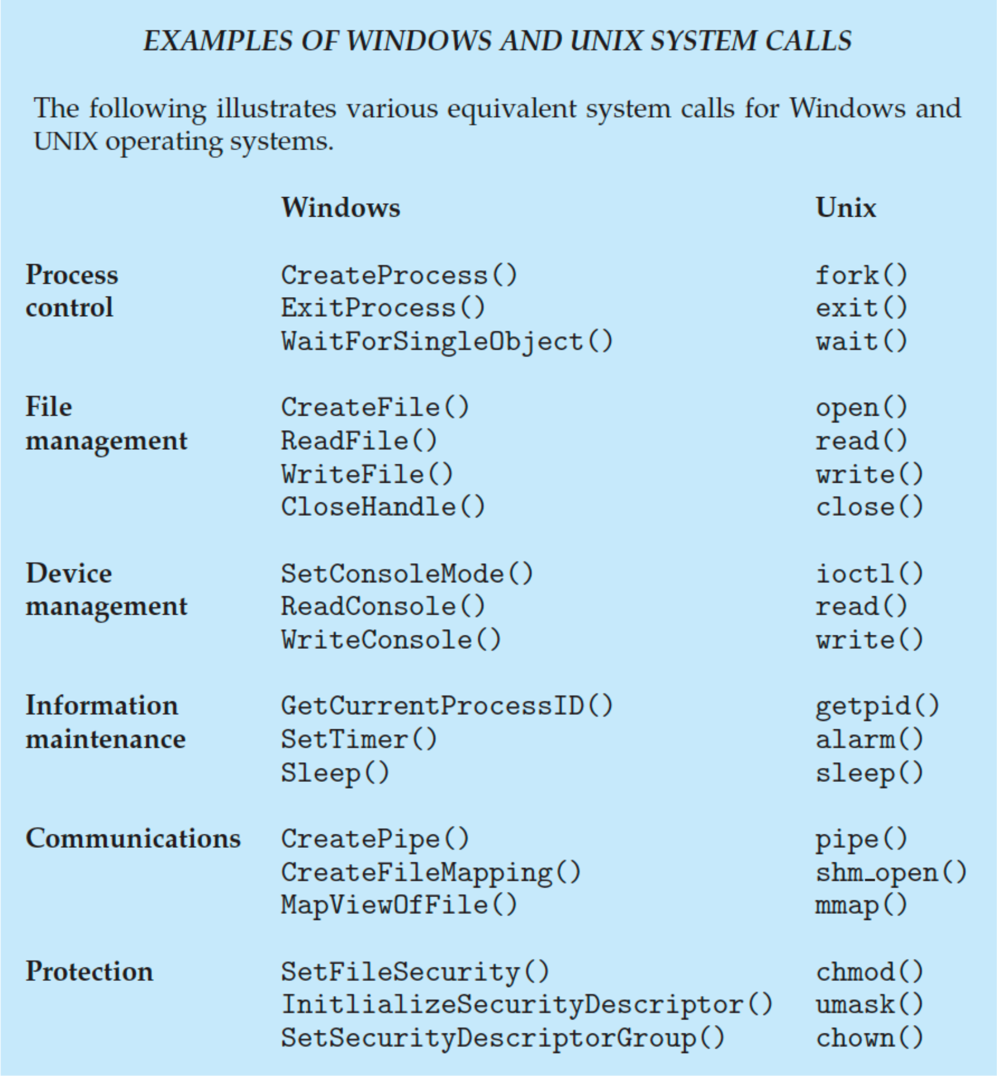
Types of System Calls (Cont.)

- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages if **message passing model** to **host name** or **process name**
    - From **client** to **server**
  - **Shared-memory model** create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices

Types of System Calls (Cont.)

- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

Examples of Windows and Unix System Calls



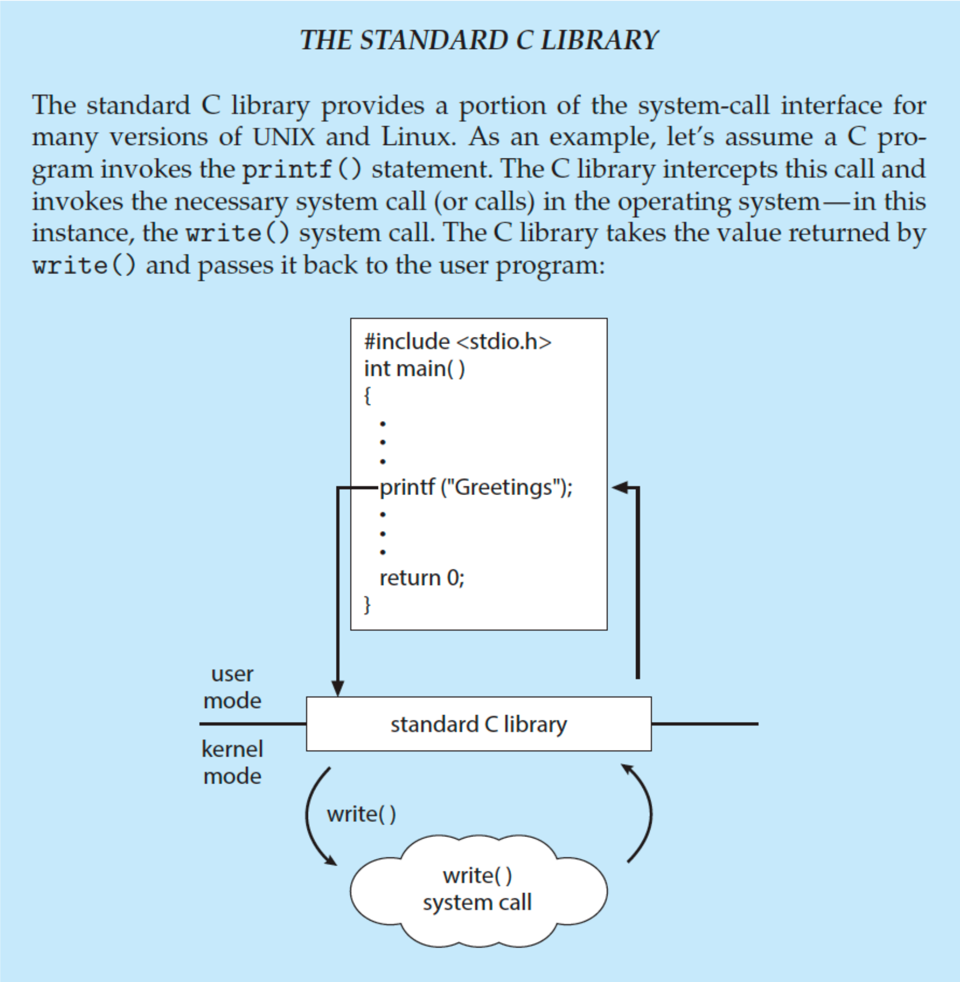
EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	**Unix**
Process control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File management	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device management	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`
	`WriteConsole()`	`write()`
Information maintenance	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`
Communications	`CreatePipe()`	`pipe()`
	`CreateFileMapping()`	`shm_open()`
	`MapViewOfFile()`	`mmap()`
Protection	`SetFileSecurity()`	`chmod()`
	`InitlializeSecurityDescriptor()`	`umask()`
	`SetSecurityDescriptorGroup()`	`chown()`

Standard C Library Example

- C program invoking printf() library call, which calls write() system call



THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:

```
#include <stdio.h>
int main( )
{
   •
   •
   •
   printf ("Greetings");
   •
   •
   •
   return 0;
}
```

Mips trap: Handling System Calls, Exceptions, and Interrupts

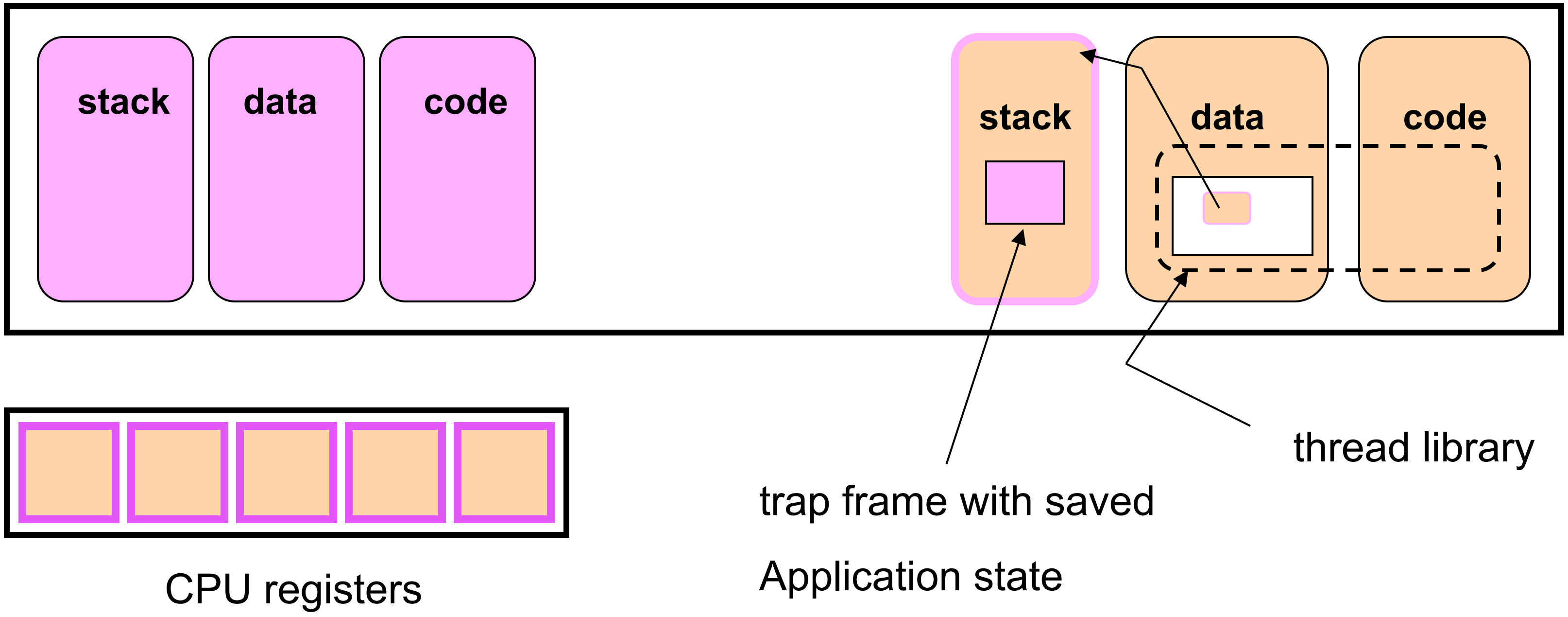
- On the MIPS, the same exception handler is invoked to handle system calls,exceptions and interrupt

- The hardware sets a code to indicate the reason (system call, exception, or interrupt) that the exception handler has been invoked

- OS161 has a handler function corresponding to each of these reasons. The mips trap function tests the reason code and calls the appropriate function: the system call handler (mips syscall) in the case of a system call.

- Mips trap can be found in kern/arch/mips/locore/trap.c.

OS161 Trap Frame



thread library

trap frame with saved

Application state

CPU registers

While the kernel handles the system call, the application's CPU state is saved in a trap frame on the thread's kernel stack, and the CPU registers are available to hold kernel execution state.

OS161 MIPS System Call Handler

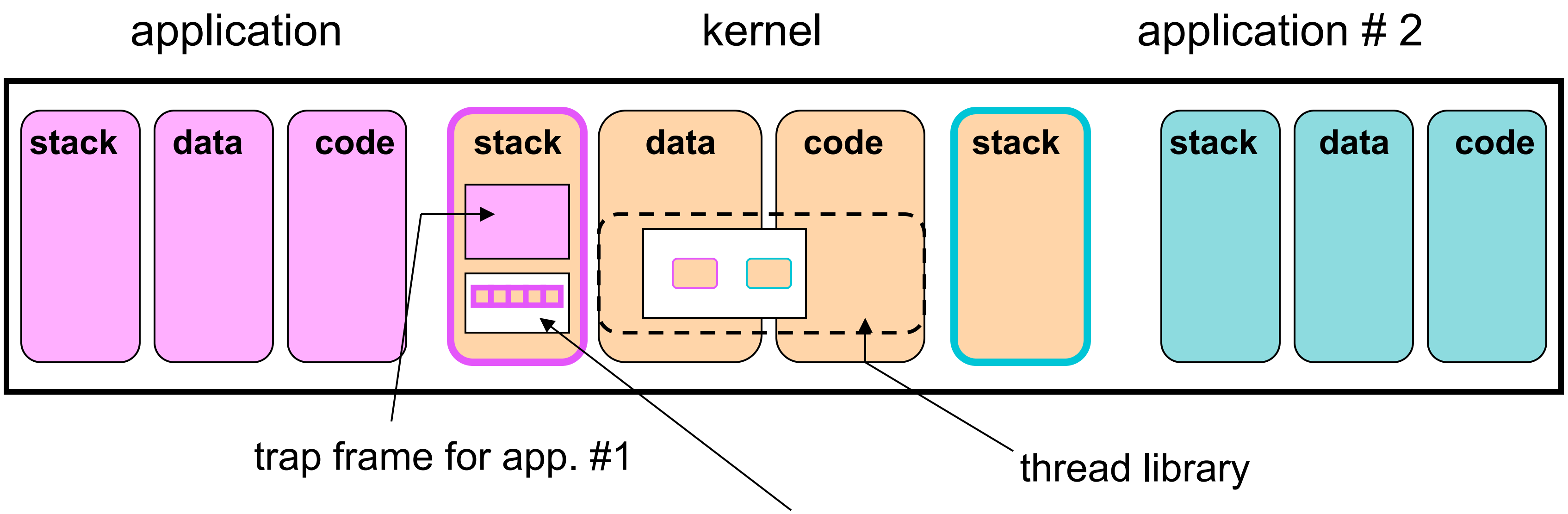
```
void
syscall(struct trapframe *tf) {
  ..
  callno = tf->tf_v0; retval = 0;
  switch (callno) {
    case SYS_reboot:
      err = sys_reboot(tf->tf_a0); /* in kern/main/main.c */
      break;

    /* Add stuff here */
    default:
      kprintf("Unknown syscall %d\n", callno);
      err = ENOSYS;
      break;
 }
```

MIPS Exceptions

```
EX_IRQ      0    /* Interrupt */
EX_MOD      1    /* TLB Modify (write to read-only page) */
EX_TLBL     2    /* TLB miss on load */
EX_TLBS     3    /* TLB miss on store */
EX_ADEL     4    /* Address error on load */
EX_ADES     5    /* Address error on store */
EX_IBE      6    /* Bus error on instruction fetch */
EX_DBE      7    /* Bus error on data load *or* store */
EX_SYS      8    /* Syscall */
EX_BP       9    /* Breakpoint */
EX_RI       10   /* Reserved (illegal) instruction */
EX_CPU      11   /* Coprocessor unusable */
EX_OVF      12   /* Arithmetic overflow */
```

In OS161, mips trap uses these codes to decide whether it has been invoked because of an interrupt, a system call, or an exception.

Two Processes in OS161

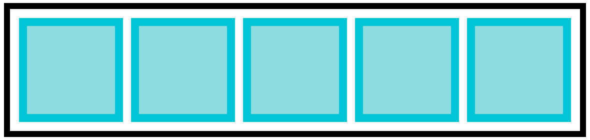


trap frame for app. #1

thread library

saved kernel thread

context for thread # 1



CPU registers

System Calls for Process Management

	linux	OS161
Creation	fork,execve	fork,execv
Destruction	_exit,kill	_exit
Synchronization	wait,waitpid,pause,…	waitpid
Attribute Mgmt	getpid,getuid,nice,getrusage,...	getpid

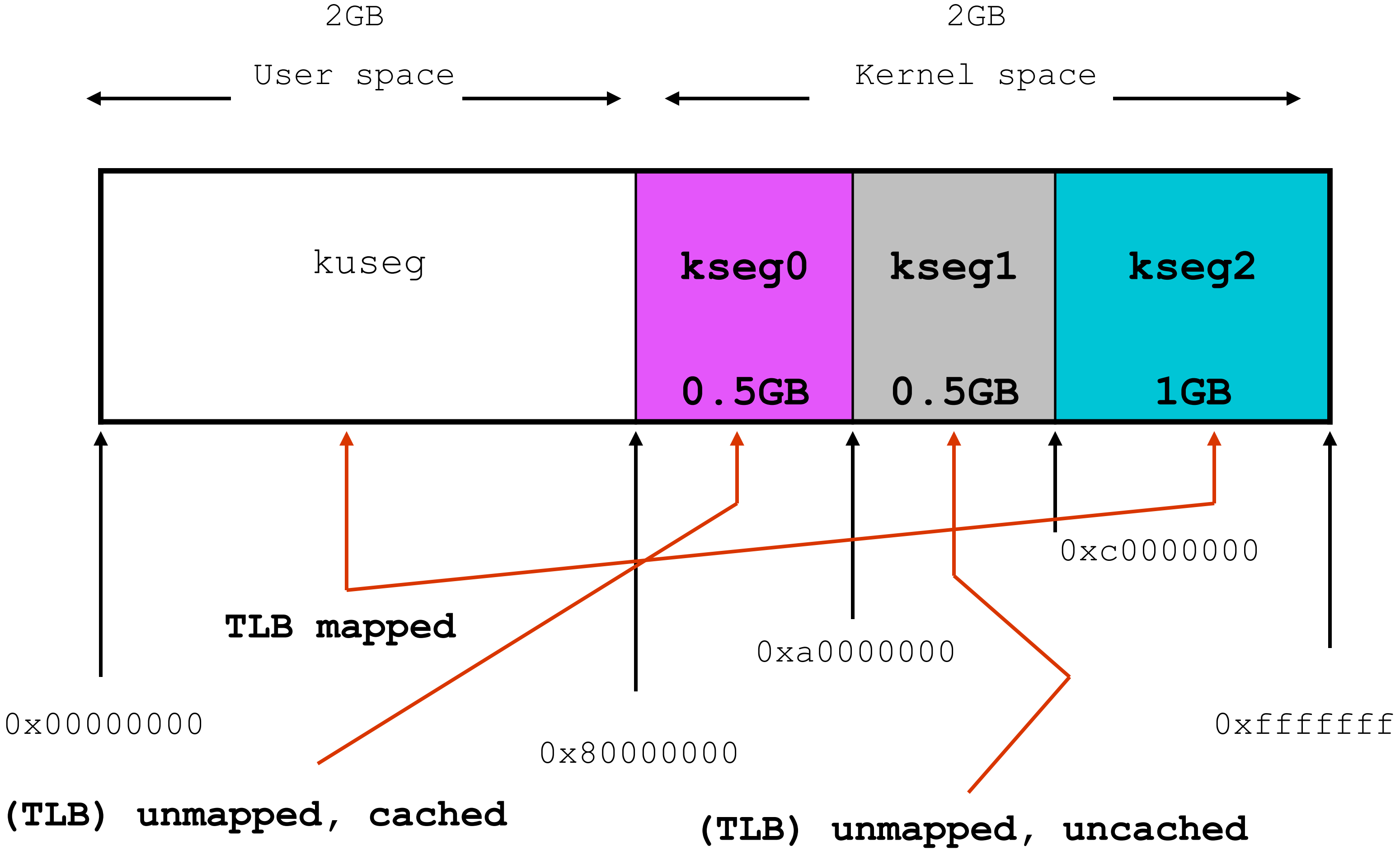
OS161 Memory Management

- Kernel/User address spaces
- Mips logical addresses
- Mips TLB
- DUMBVM virtual memory management
- Loading an ELF file into a (process) address space

An Address Space for the Kernel

- Each process has its own address space. What about the kernel?
- two possibilities
  - **Kernel in physical space: disable address translation in privileged system**
    - execution mode, enable it in unprivileged mode
  - **Kernel in separate virtual address space: need a way to change address**
    - translation (e.g., switch page tables) when moving between privileged and
    - unprivileged code
- OS161, Linux, and other operating systems use a third approach: the kernel is mapped into a portion of the virtual address space of *every process*
- memory protection mechanism is used to isolate the kernel from applications
- one advantage of this approach: application virtual addresses (e.g., system call parameters) are easy for the kernel to use

Address Translation on the MIPS R3000



2GB

User space

2GB

Kernel space

kuseg

kseg0

0.5GB

kseg1

0.5GB

kseg2

1GB

0xc0000000

TLB mapped

0xa0000000

0x00000000

0x80000000

0xfffffff

(TLB) unmapped, cached

(TLB) unmapped, uncached

In OS/161, user programs live in kuseg, kernel code and data structures live in kseg0, devices are accessed through kseg1, and kseg2 is not used.

The MIPS R3000 TLB

- The MIPS has a software-controlled TLB than can hold 64 entries.
- Each TLB entry includes a virtual page number, a physical frame number, an address space identifier (not used by OS161), and several flags (valid,read-only)↰
- OS161 provides low-level functions for managing the TLB:

tlb_write(): modify a specified TLB entry

tlb_random(): modify a random TLB entry

tlb_read(): read a specified TLB entry

tlb_probe(): look for a page number in the TLB

- If the MMU cannot translate a virtual address using the TLB it raises an exception, which must be handled by OS161

See kern/arch/mips/include/tlb.h

OS161 Address Spaces: dumbvm

- OS161 starts with a very simple virtual memory implementation
- virtual address spaces are described by addrspace objects, which record the mappings from virtual to physical addresses

```
struct addrspace {
#if OPT_DUMBVM
  vaddr_t as_vbase1; /* base virtual address of code segment */
  paddr_t as_pbase1; /* base physical address of code segment */
  size_t as_npages1; /* size (in pages) of code segment */
  vaddr_t as_vbase2; /* base virtual address of data segment */
  paddr_t as_pbase2; /* base physical address of data segment */
  size_t as_npages2; /* size (in pages) of data segment */
  paddr_t as_stackpbase; /* base physical address of stack */
#else
  /* Put stuff here for your VM system */
#endif
};
```

This amounts to a slightly generalized version of simple dynamic relocation, with three bases rather than one.

See kern/include/addrspace.h

Address Translation Under dumbvm

- the MIPS MMU tries to translate each virtual address using the entries in the TLB

- If there is no valid entry for the page the MMU is trying to translate, the MMU generates a page fault (called an *address exception*)↰

- The vm fault function (see kern/arch/mips/vm/dumbvm.c) handles this exception for the OS161 kernel. It uses information from the current process' addrspace to construct and load a TLB entry for the page.

- On return from exception, the MIPS retries the instruction that caused the page fault. This time, it may succeed.

vm fault is not very sophisticated. If the TLB fills up, OS161 will crash!

Loading a Program into an Address Space

- When the kernel creates a process to run a particular program, it must create an address space for the process, and load the program’s code and data into that address space

- A program’s code and data is described in an *executable file*, which is created when the program is compiled and linked

- OS161 (and other operating systems) expect executable files to be in ELF(**E**xecutable and **L**inking **F**ormat)format

- the OS161 execv system call, which re-initializes the address space of a process

```
#include <unistd.h>
int execv(const char *program, char **args);⌉
```

- The program parameter of the execv system call should be the name of the ELF executable file for the program that is to be loaded into the address space.

ELF Files

- ELF files contain address space segment descriptions, which are useful to the kernel when it is loading a new address space

- the ELF file identifies the (virtual) address of the program’s first instruction

- the ELF file also contains lots of other information (e.g., section descriptors,symbol tables) that is useful to compilers, linkers, debuggers, loaders and other tools used to build programs

Address Space Segments in ELF Files

- Each ELF segment describes a contiguous region of the virtual address space.
- For each segment, the ELF file includes a segment *image* and a header, which describes:

- the virtual address of the start of the segment
- the length of the segment in the virtual address space
- the location of the start of the image in the ELF file
- the length of the image in the ELF file

- the image is an exact copy of the binary data that should be loaded into the specified portion of the virtual address space
- the image may be smaller than the address space segment, in which case the rest of the address space segment is expected to be zero-filled

To initialize an address space, the kernel copies images from the ELF file to the specifed portions of the virtual address space

ELF Files and OS161

- OS161's dumbvm implementation assumes that an ELF file contains two segments:
  - a *text segment*, containing the program code and any read-only data
  - a *data segment*, containing any other global program data
- the ELF file does not describe the stack (why not?)
- dumbvm creates a *stack segment* for each process. It is 12 pages long, ending at virtual address 0x7fffffff

Look at kern/syscall/loadelf.c to see how OS161 loads segments from ELF files