Introduction to OS161

Politecnico di Torino
Department of Control and Computer Engineering

System Programming - Sarah Azimi

CAD & Reliability Group
DAUIN– Politecnico di Torino

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)?

- Thread or 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)

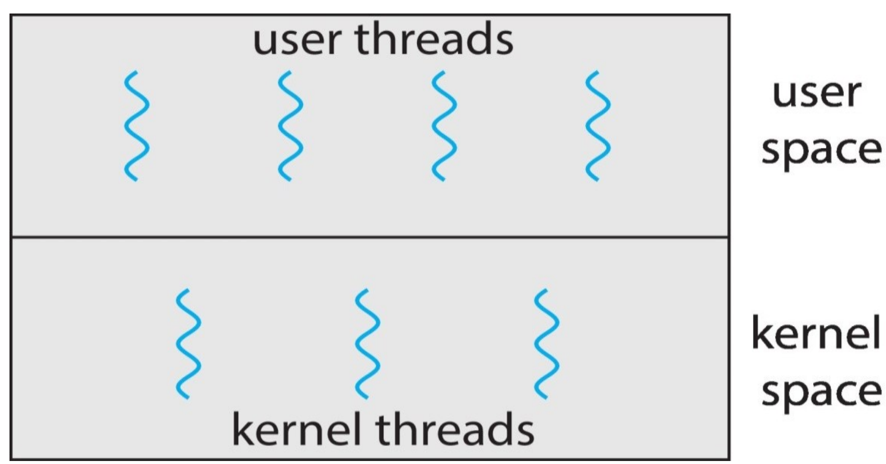
User Threads and Kernel Threads

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

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

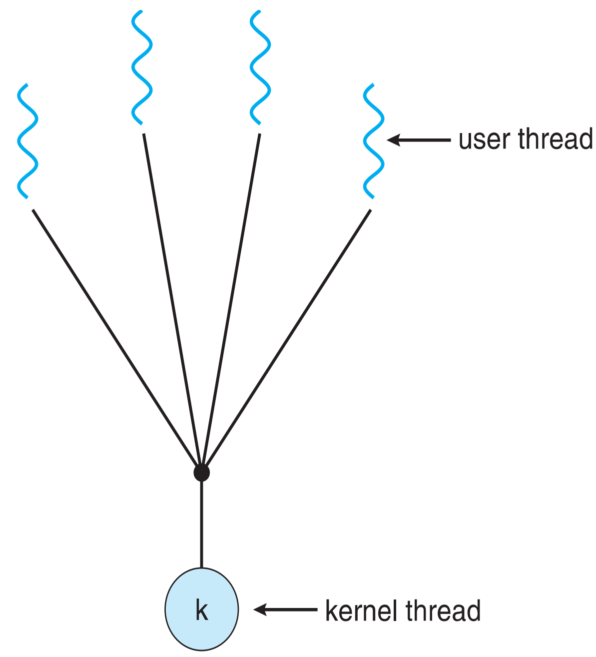
User and Kernel Threads

- User threads are supported above the kernel and are managed without kernel support.
- Kernel threads are supported and managed directly by the operating system.
- A relationship must exist between user threads and kernel threads.
- A user thread should be mapped to the kernel thread.



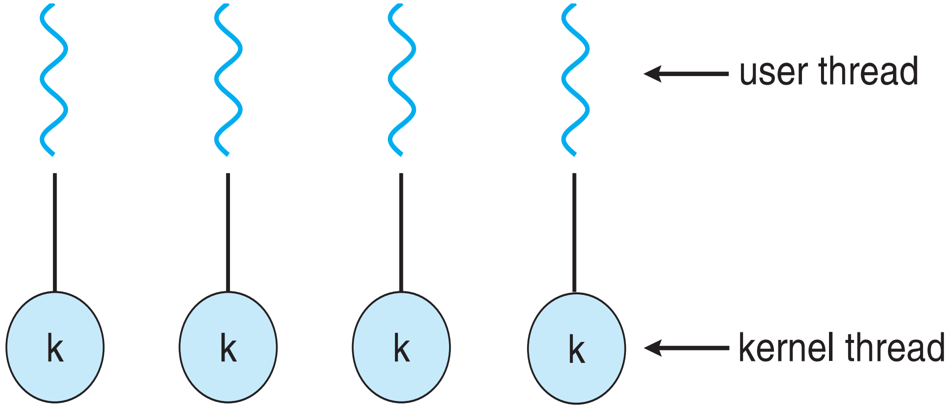
Multithreading Models

- Three common ways to establish a relationship between kernel threads and user threads:
  - Many-to-One: mapping many user-level threads to one kernel thread.
    - Only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.
    - One thread blocking causes all to block
    - Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time



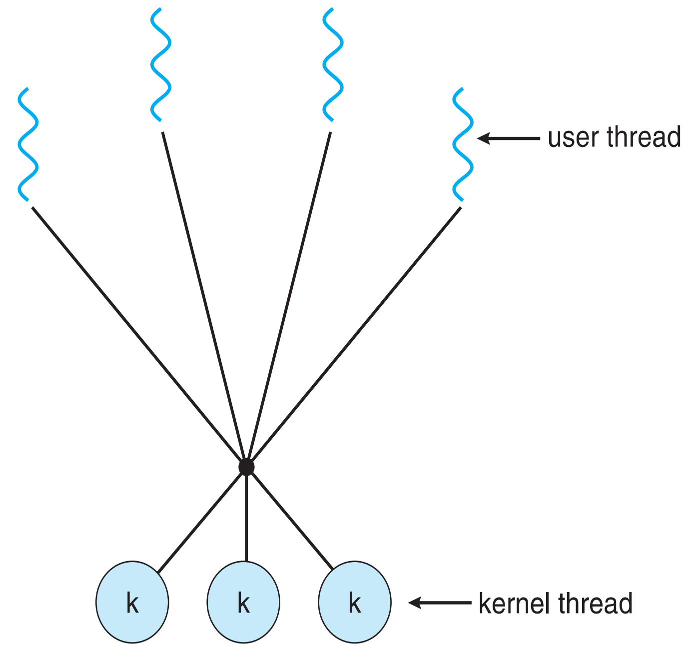
Multithreading Models

- Three common ways to establish a relationship between kernel threads and user threads:
  - One-to-One: Mapping each user thread to a kernel thread.
    - More concurrency than many-to-one
    - Number of threads per process sometimes restricted due to overhead
    - While allowing multiple threads to run in parallel, the drawback is that creating a user thread requires creating the corresponding kernel thread and a large number of kernel threads may burden the performance of a system.



Multithreading Models

- Three common ways to establish a relationship between kernel threads and user threads:
  - Many-to-Many: mapping many user-level threads to a smaller or equal number of kernel threads
    - Allows the operating system to create a sufficient number of kernel threads
    - Developers can create as many user threads as necessary and the corresponding kernel threads can run in parallel on a multiprocessor.



Process Concept

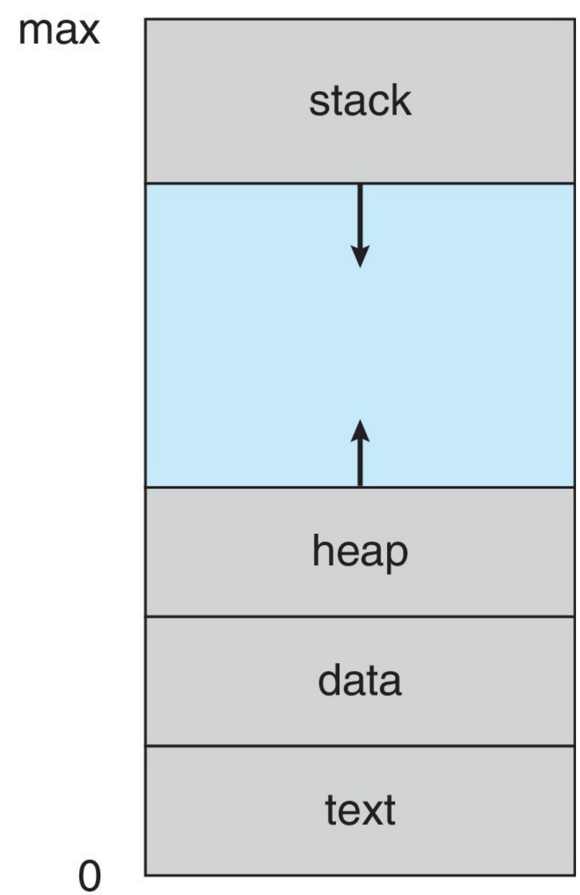
- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution; process execution must progress in a sequential fashion
- Multiple parts
  - The program code, also called the **text section**
  - Current activity including **program counter**, processor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

Process Concept

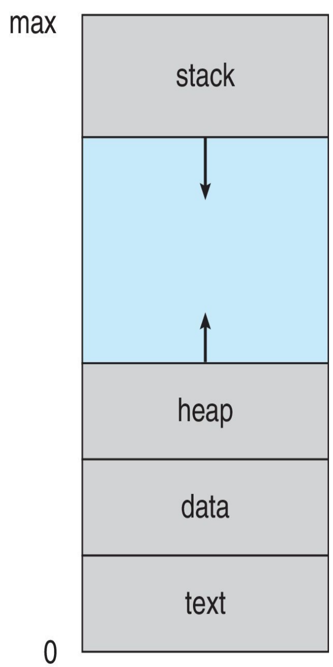
- Program is ***passive*** entity stored on disk (**executable file**); process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider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

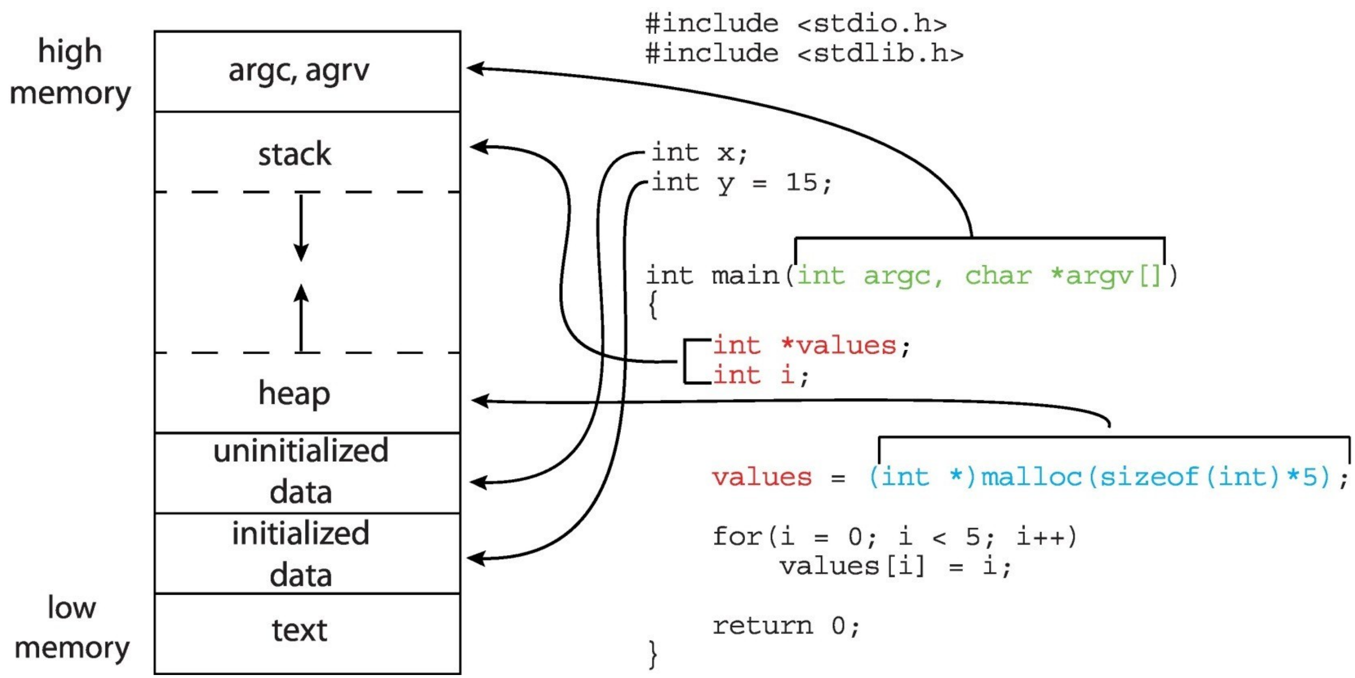
Process in Memory

- User side layout of the memory
- Logical memory address of processes are organized as:
  - Text Where the code is
  - Data where the global variables are
  - Stack
  - Heap
  - Stack and Heap can increase the size



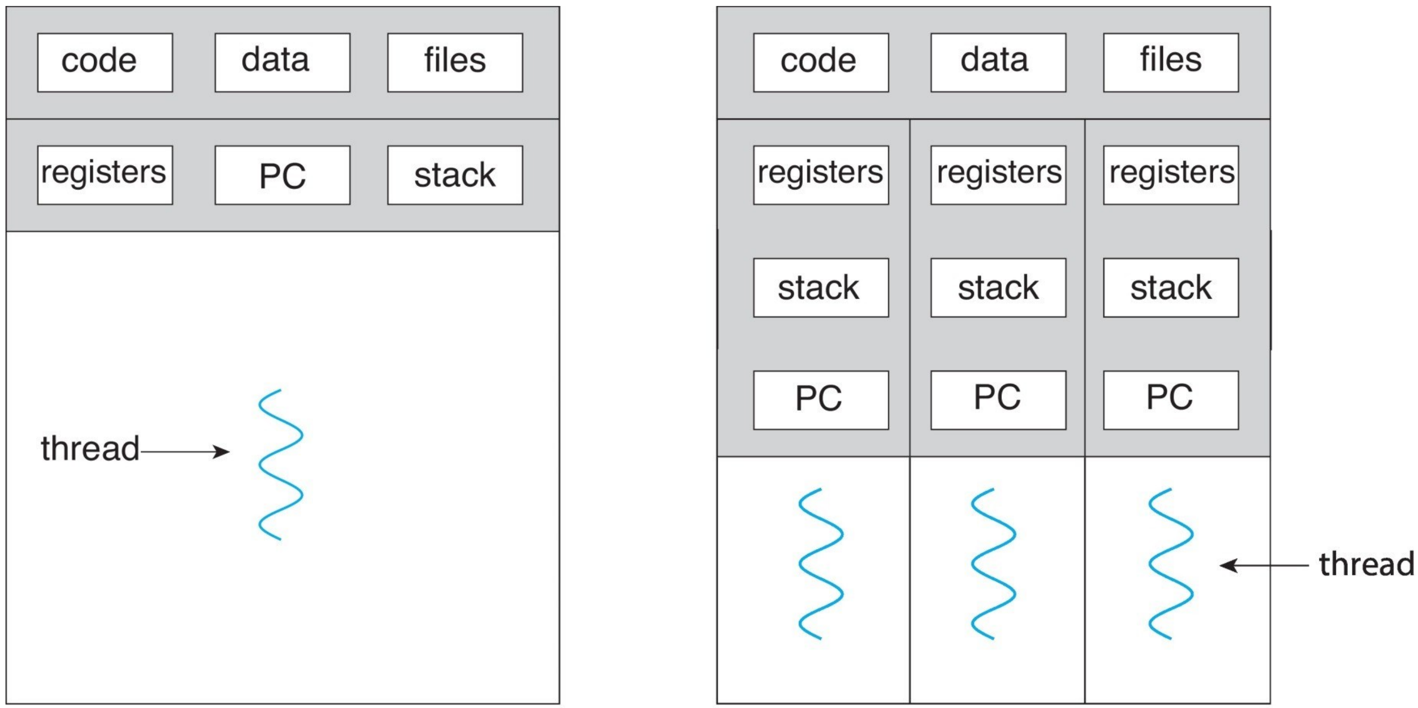
Memory Layout of a C Program





Single and Multithreaded Processes

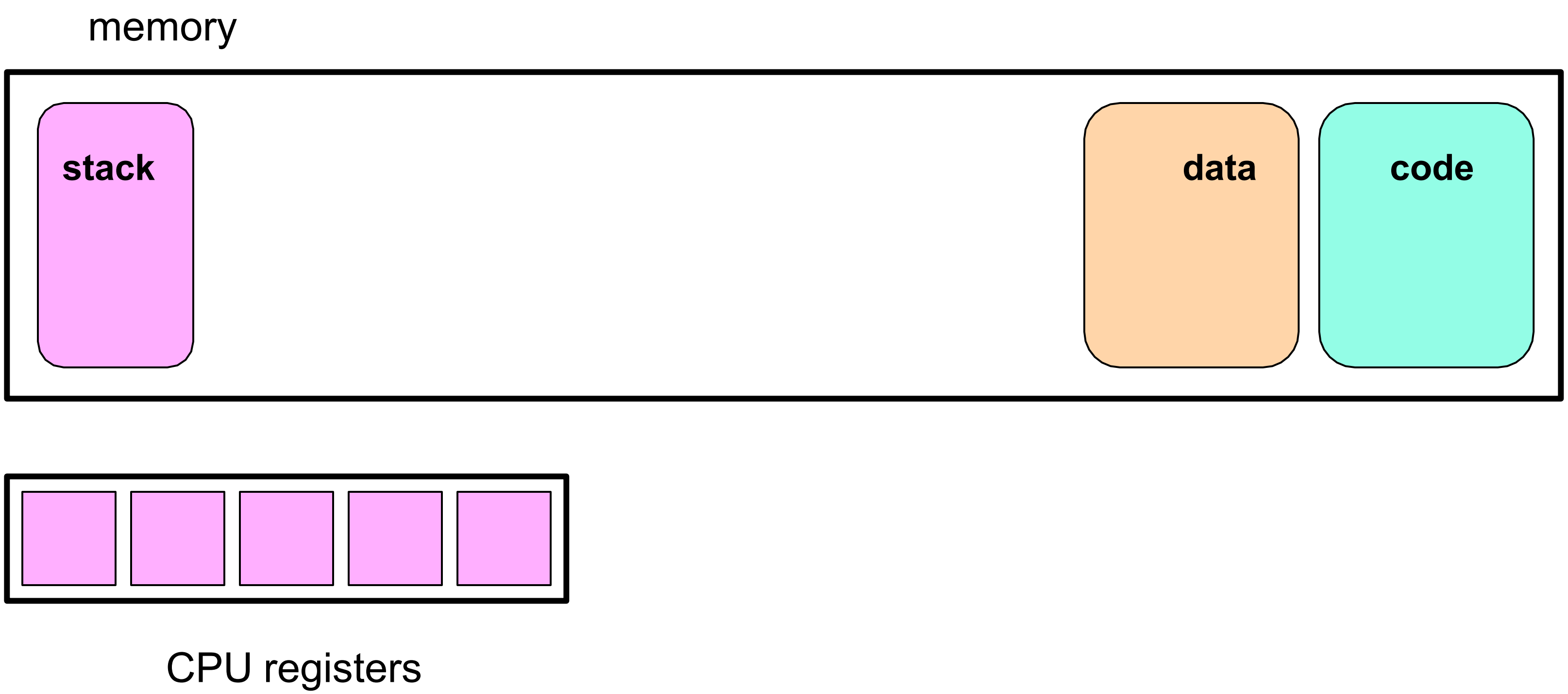
- Single thread single process
- Single process multiple threads



single-threaded process

multithreaded process

Thread Context



Thread Libraries

- **Thread library** provides the 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

Implementing Threads

- Moving toward thread on OS161:
- A thread library is responsible for implementing threads.(managing threads at kernel level through library)
- Having data structure for saving thread information: the thread library stores threads' contexts (or pointers to the threads' contexts) when they are not running.
- The Data structure used by thread library to store a thread context is called **Thread Control Block**
- In the OS/161 kernel's thread implementation, thread contexts are stored in thread structures

The OS161 thread Structure

All instructions are written in C, the structure that defines thread can be find in “**thread.h**”

```
/* 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 */ Every
                                       thread has a state such as
                                       running, sleeping, ready.
  /* 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 OS161 thread Structure

Data structure for saving thread information.
Each thread has name, state of execution (running, waiting, ..)

```
/* 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 */ Every
                                           thread has a state such as
                                           running, sleeping, ready.
  /* 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 OS161 thread Structure

Each thread is associated to a stack, so there is a pointer at TCB that points to the area of stack, as part of kernel memory.

```
/* 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 */ Every
                                      thread has a state such as
                                      running, sleeping, ready.

  /* 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 OS161 thread Structure

Structure allows the OS to save the registers of the CPU associated to the thread. Switching from one thread to another thread, OS saves all the registers in this structure memorized as part of kernel memory.

```
/* 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 */ Every
                                            thread has a state such as
                                            running, sleeping, ready.

  /* 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 OS161 thread Structure

If a thread is in execution, in which CPU the thread is executed in written here. (in case of multi core processor)

```
/* 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 */ Every
                                     thread has a state such as
                                     running, sleeping, ready.

  /* 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 OS161 thread Structure

In os161, each thread knows its process. While thread knows its process, the process does not know its thread.

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

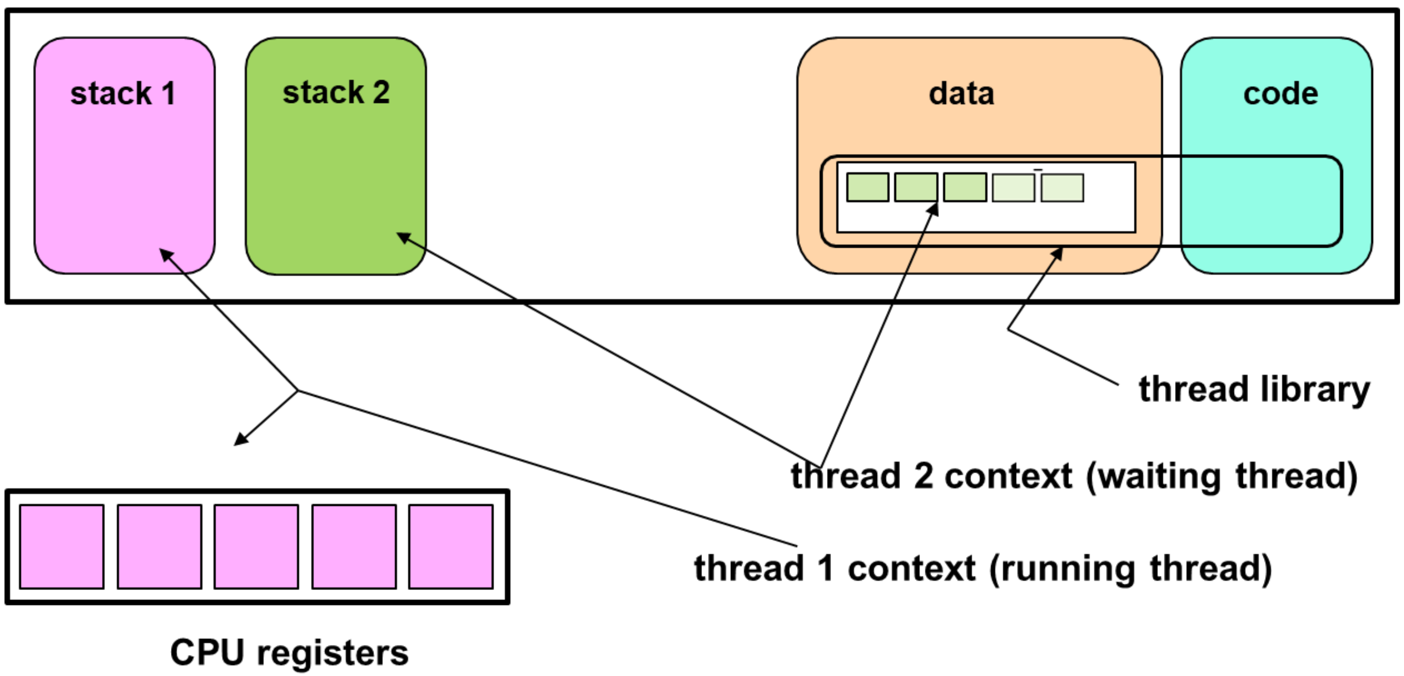
struct thread {
  char *t_name;                       /* Name of this thread */
  const char *t_wchan_name;           /* Name of wait channel, if sleeping */
  threadstate_t t_state;              /* State this thread is in */ Every
                                         thread has a state such as
                                         running, sleeping, ready.

  /* 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 */
  ...
};
```

Thread library and two threads (generic)

A processor with two active threads:

1. Having two stack for the threads.
2. In thread library, one of the two thread is active (violet one)
3. The inactive one should save its context and registers in the its stack.
4. Switching thread in OS, recovering the context of one of the threads from stack, loading it to the CPU while saving the context of the running thread and saving it in its stack.



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`

The functions that manage the thread in OS161

The OS161 thread Structure

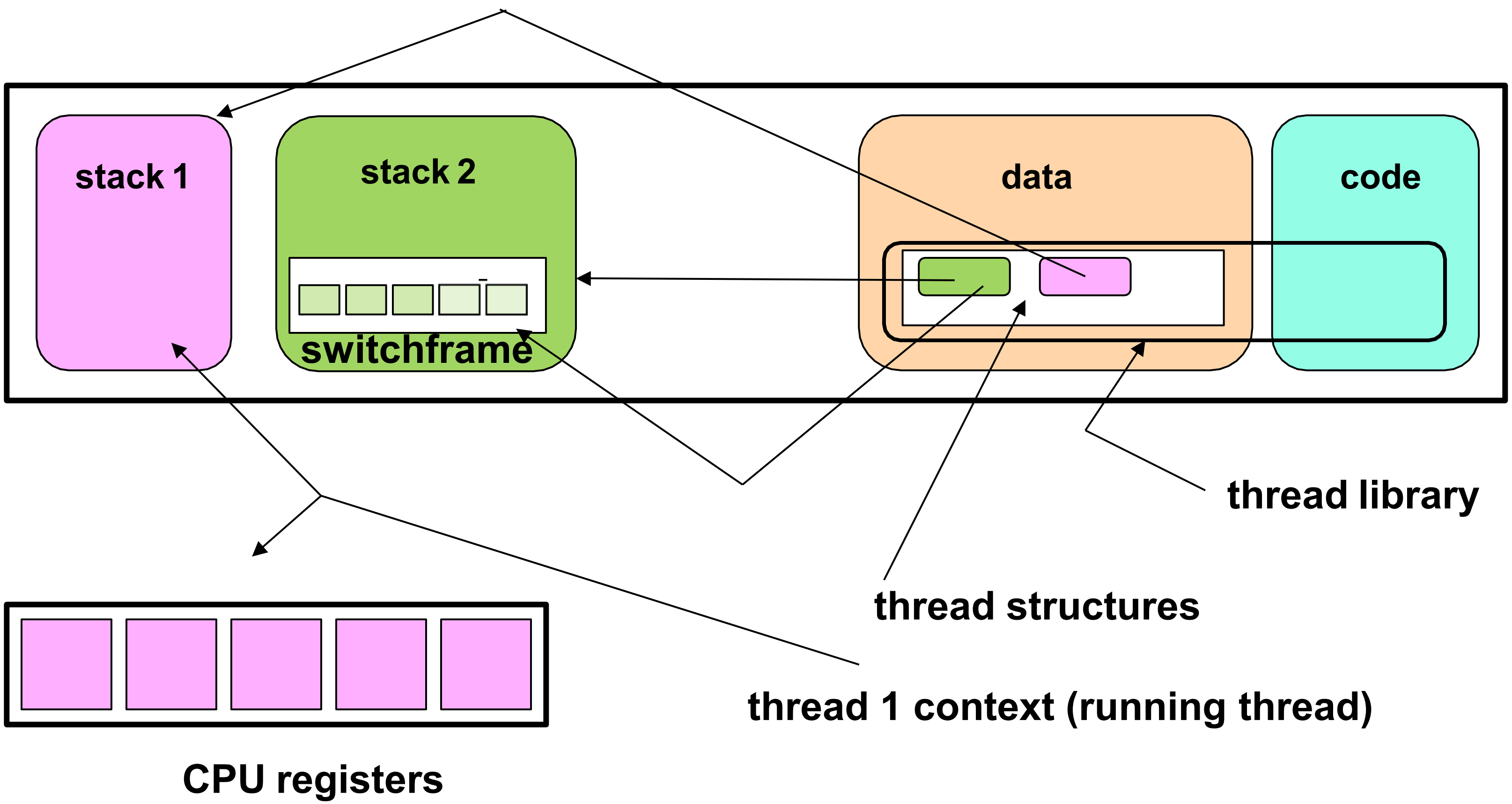
```
/* see kern/include/thread.h */

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 two elements that allows saving the context in thread: kernel level stack and switch frame

Thread library and two threads (OS161)



The OS161 Thread Interface (incomplete)

The library of thread in OS161 is not complete. There are some functions to be done by US.

```
/* 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);
  ```
  ```

The OS161 Thread Interface (incomplete)

```
/* 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);
```

Thread_fork, for creating a new thread. First step, defining an entry point which is a pointer to a zone of a memory (third parameter of the **thread_fork** function, a pointer to the address of a memory that contains the codes of the processor). In os161, the entry point has a fixed structure, expecting two parameters: first, a pointer to the data memory (the function to execute). The second is an integer number (a pointer to the parameters of the function).

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

The OS161 Thread Interface (incomplete)

```
/* 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. */

Exiting/disabling the thread

```
char *name, struct proc *proc,
t)(void *, unsigned long),
igned 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);
```

The OS161 Thread Interface (incomplete)

```
/* 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,`

Forcing the context switch. Calling the **thread_yields** puts the calling thread on pause and ask the OS to schedule the next thread.

- `__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()

Creating "i" number of threads

```
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);
  }
}
```

Creating Threads Using thread_fork()

Defining names for the threads

```
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);
  }
}
```

Creating Threads Using thread_fork()

Calling thread_fork, passing the name, Null (associating the thread to the current processor), pointer to the function, Null (no paramtere passed), i as the variable)

```
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);
   ...
}
```

Thread_fork is a wrapper for **thread_create, switchframe_init, and thread_make_runnable.**

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);
   ...
}
```

Thread_create is for creating the thread, allocating the TCB and the necessary memory for managing the thread.

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);
   ...
}
```

Creating the switch frame, allowing to memorize the values of all the registers through the switchframe_init to initialize the frames.

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);
   ...
}
```

Changing the status of the thread to ready and inserting the thread in the ready list to be scheduled

From ready to execution: thread_switch()

When there are multiple threads and thread switch is required:
Thread_switch is called that change the state of the current thread, taking a new thread from the list.
Calling the **switchframe_switch** to change the context.

```
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);
   ...
}
```

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);
   ...
}
```

While all the functions that manage the threads are written in C, the functions that manage the context is written in assembly.

For saving the context, we should know the architecture of the processor.

OS161 >> processor with MIPS architecture

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()

The functions to switch frames are architecture dependent:
Saving the current switch frame and restoring the switch frame of the next thread.

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

   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
```

The organization of registers in MIPS architecture

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)
```

In MIPS there are 31 registers, each group of registers has a meaning. Each register has two names, one is the actual name, the other one represents the characteristics of the register. When we switch the context, we save the status of these registers and then restore the new values.

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; */
```

Switchframe_switch implemented in OS161:
Allocating the space for switch frame, 10 registers 32 bits (40 byte). At start, extra information is saved, so 44 byte is used.
The function is saving all the registers that are important to do context switch (10 registers). In addition, the stack pointer of the process is saved in switch frame. Therefore, there are extra 4 bytes.

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
```

Switchframe_switch

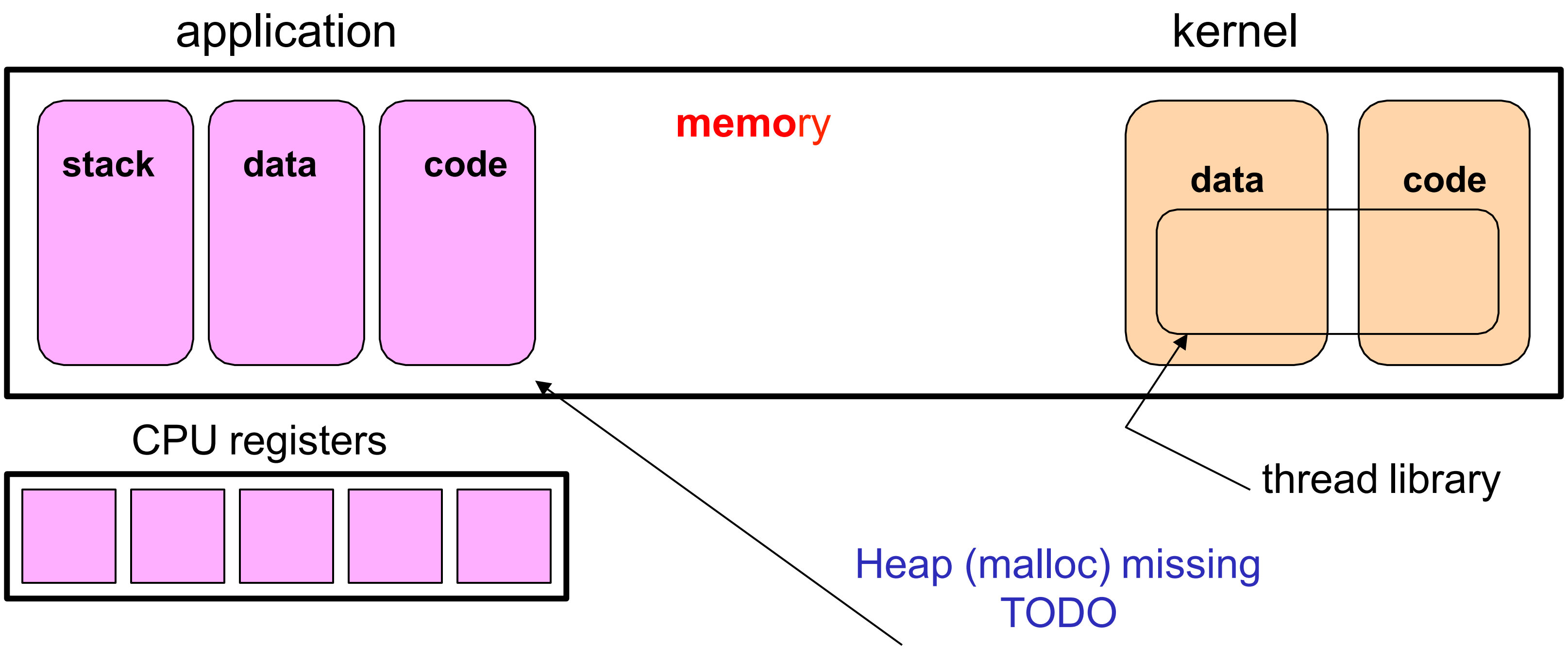
Each thread has its own stack and stack pointer.

Saving the context of the thread is saving all the registers and stack pointer.

Loading the context of a thread is loading the stack pointer and from there, loading all the values of the registers and loading it in the memory.

Once all done, jump to return the address to continue the execution.

Application (User process) and Kernel



Layout the memory in OS161:
For each process, OS 161 can see the kernel and user side:
User side has stack, data and code.
The kernel side, the memory associated to the kernel.

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 */
```

Every time a program is launched, a process is created >> data structure (PCB) is created. The data structure includes:
Process name, number of active thread (not knowing which threads), synchronization (spinlock), pointer to the virtual address space of the process (where the process is mapped in virtual address space, files)

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 */
   ...
```

Every time a program is launched, a process is created >> data structure (PCB) is created. The data structure includes:
Process name, number of active thread (not knowing which threads), synchronization (spinlock), pointer to the virtual address space of the process (where the process is mapped in virtual address space, files)

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 */
   ...
```

Every time a program is launched, a process is created >> data structure (PCB) is created. The data structure includes:
Process name, number of active thread (not knowing which threads), synchronization (spinlock), pointer to the virtual address space of the process (where the process is mapped in virtual address space, files)

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 */
   ...
```

Every time a program is launched, a process is created >> data structure (PCB) is created. The data structure includes:
Process name, number of active thread (not knowing which threads), synchronization (spinlock), pointer to the virtual address space of the process (where the process is mapped in virtual address space, files)

The OS161 proc Structure

(PCB: Process Control Block)

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

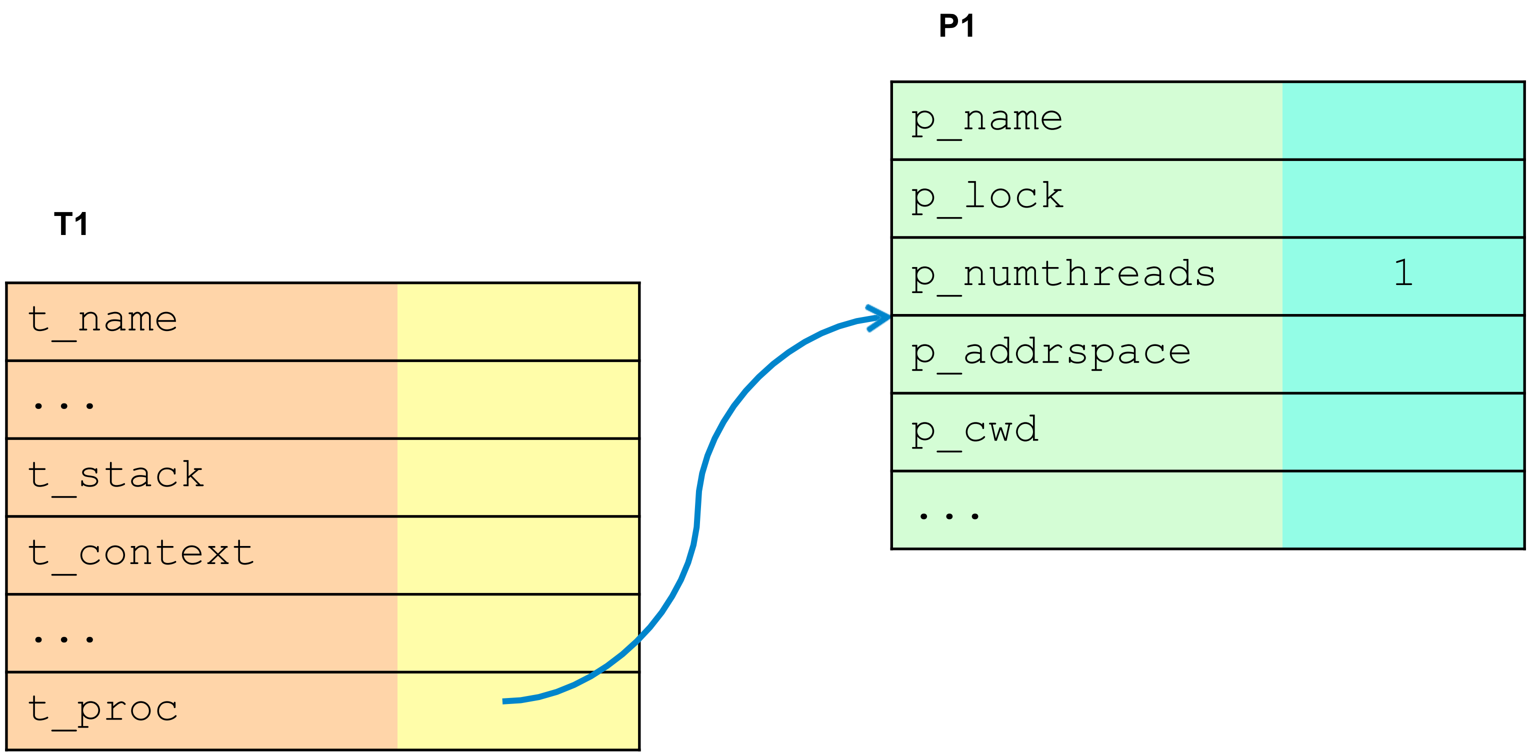
 struct proc {
   char *p_name;                      /* Name of this process */
   struct spinlock p_lock;            /* Lock for this structure */
   unsigned p_numthreads;             /* Number of threads in this process */
   /* VM */
   struct addrspace *p_addrspace;    /* virtual address space */
   /* VFS */
   struct vnode *p_cwd;               /* current working directory */

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

Every time a program is launched, a process is created >> data structure (PCB) is created. The data structure includes:
Process name, number of active thread (not knowing which threads), synchronization (spinlock), pointer to the virtual address space of the process (where the process is mapped in virtual address space, files)

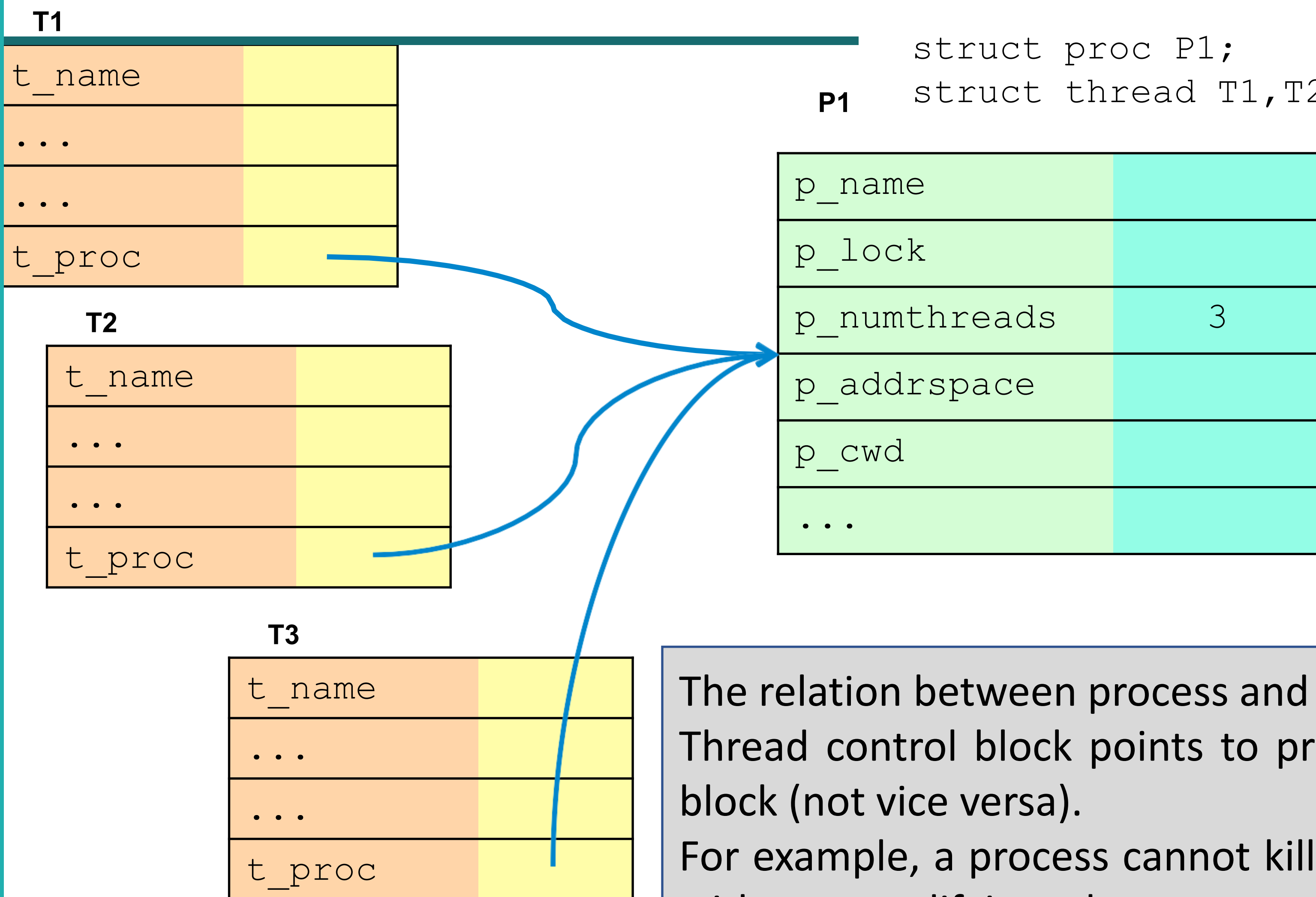
Single threaded Process

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



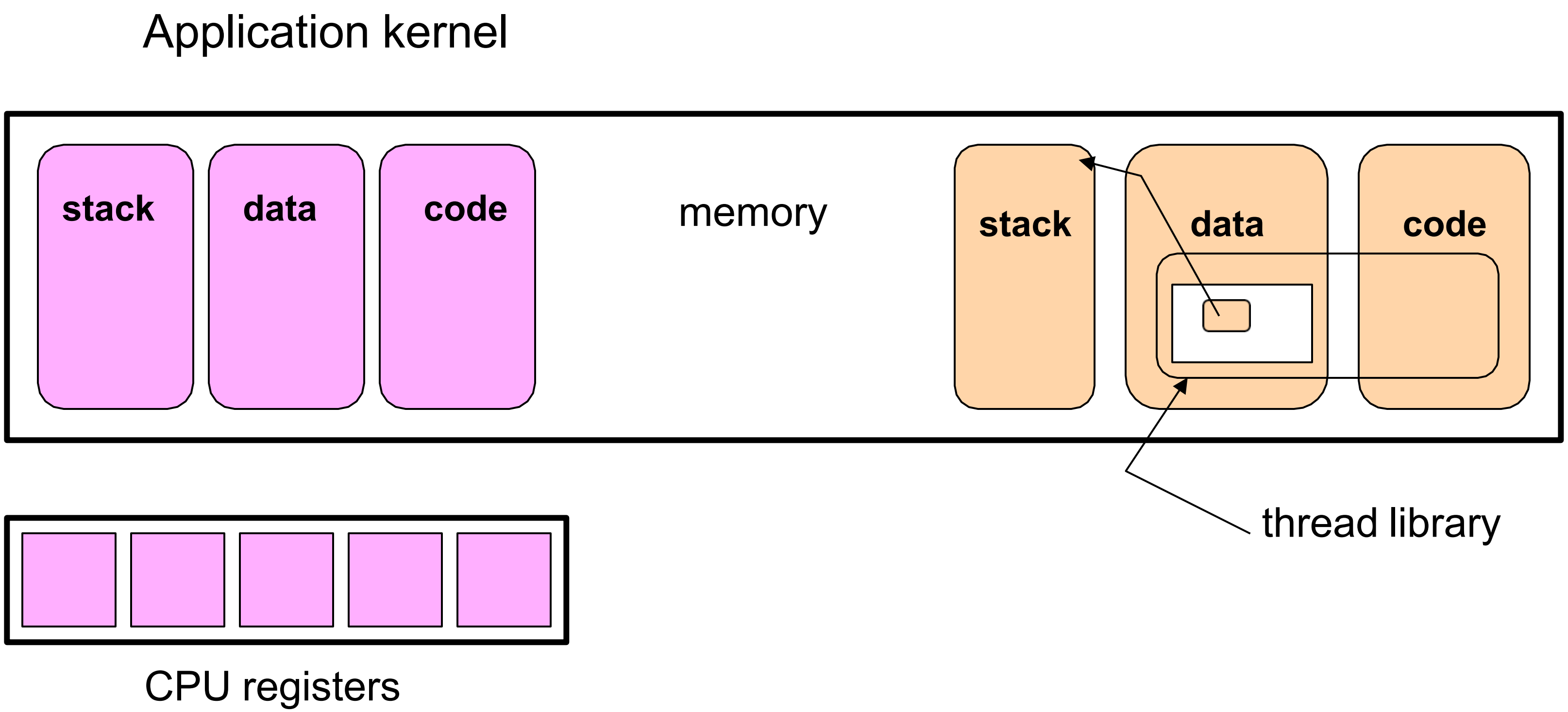
Multi threaded Process

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



The relation between process and thread:
Thread control block points to process control block (not vice versa).
For example, a process cannot kill all its thread without modifying the current structure of process since the process is not able to find all its thread.

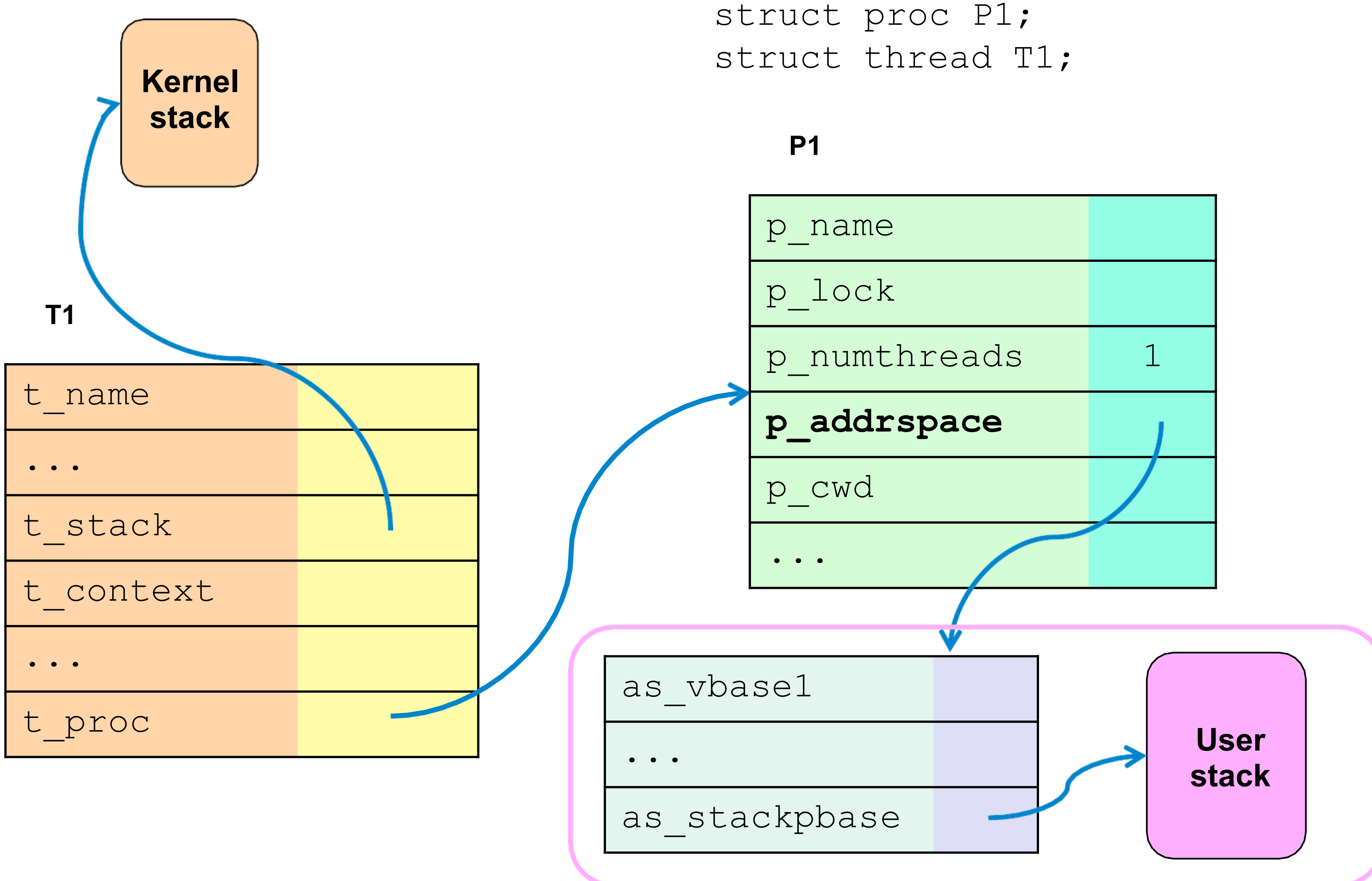
OS161 User and Kernel Thread Stacks



Each OS161 thread has two stacks, one (user side) that is used while the thread is executing unprivileged (user) application code, and another (kernel side) 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)

What happens when we execute a program(p file name (.elf):

1. OS calls **proc_create_runprogram** to create a user process (create PCB and allocating memory)
2. Each process should have at least one thread, therefore, calling **thread_fork**.
3. **run_program** execution: generating the address space, loading the elf file into memory, starting the execution of the thread.

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)

What happens when we execute a program(p file name (.elf):

1. OS calls **proc_create_runprogram** to create a user process (create PCB and allocating memory)
2. Each process should have at least one thread, therefore, calling **thread_fork**.
3. **run_program** execution: generating the address space, loading the elf file into memory, starting the execution of the thread.

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)

What happens when we execute a program(p file name (.elf):

1. OS calls **proc_create_runprogram** to create a user process (create PCB and allocating memory)
2. Each process should have at least one thread, therefore, calling **thread_fork**.
3. **run_program** execution: generating the address space, loading the elf file into memory, starting the execution of the thread.

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 is a system call for defining the structure of the program:

1. Addrspace, a data structure that memory manager of OS161 uses to manage the space of a program (which zone of memory is associated to that process).
2. Entry point, address of the first instructor to execute.
3. Stack pointer that shows where the stack (user) is allocated.

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 is

1. open the elf file, passing the name of the program
2. creating an address space (as_create)
3. associating the address space to the processor (proc_setas)
4. activate it (as_activate)

Running a user program

```
/* 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;
```

Runprogram is

1. Reading the content of elf file, allocating it in the entrypoint (load_elf)
2. Defining the stack (as_define_stack)
3. Starting the execution of the process (enter_ new_peocess)

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) {
```

The function **enter_new_process for allowing the start of the execution of the process.**

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

registers = tf;

return from exception;

```
    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);
}
```

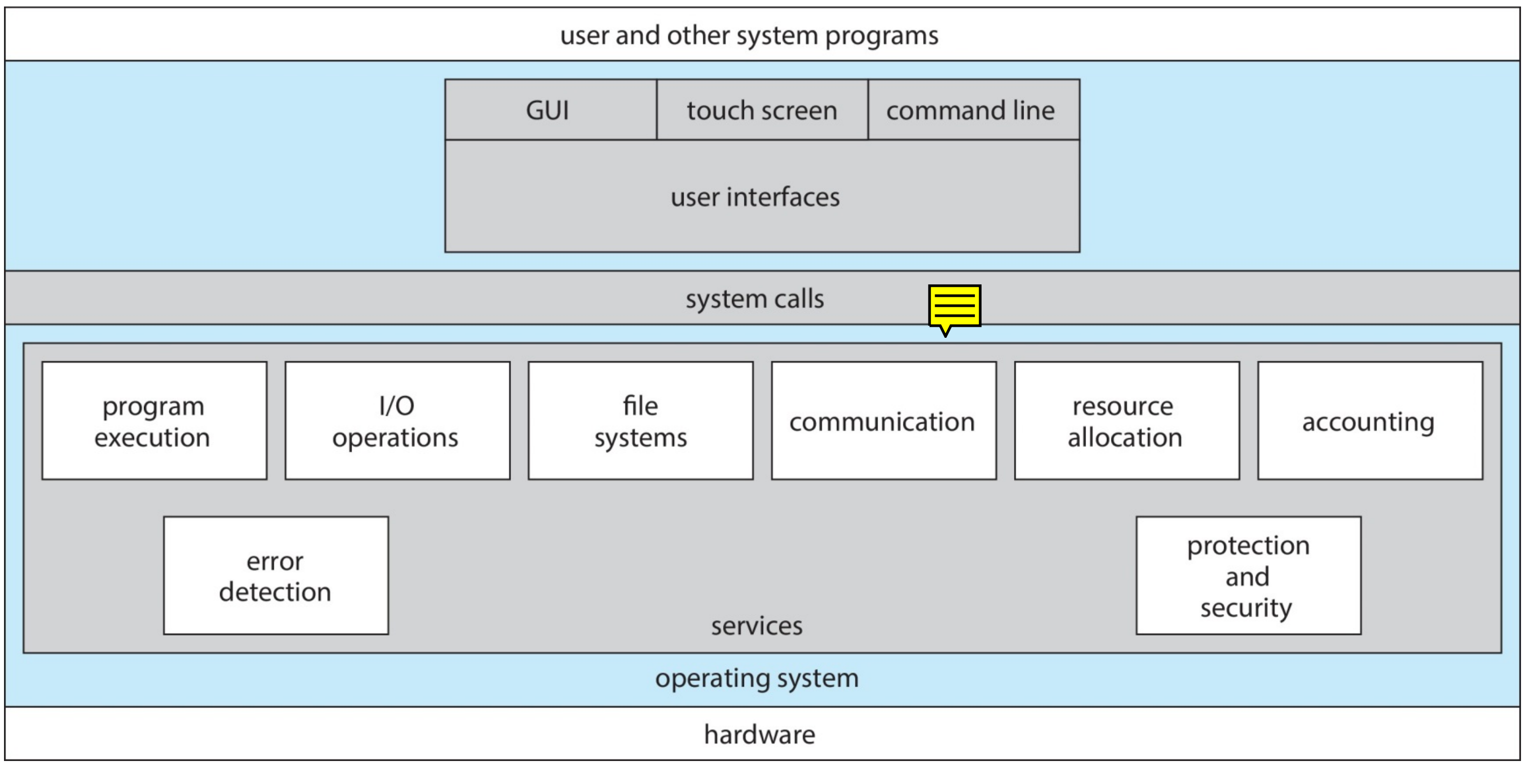
Asm_usermode on the MIPS

```
/* 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

- An operating system provides an environment for the execution of programs.
- It makes certain services available to programs and users of those programs.
- System calls provide an interface to the services made available by an operating system.
- Typically written in a high-level language (C or C++)

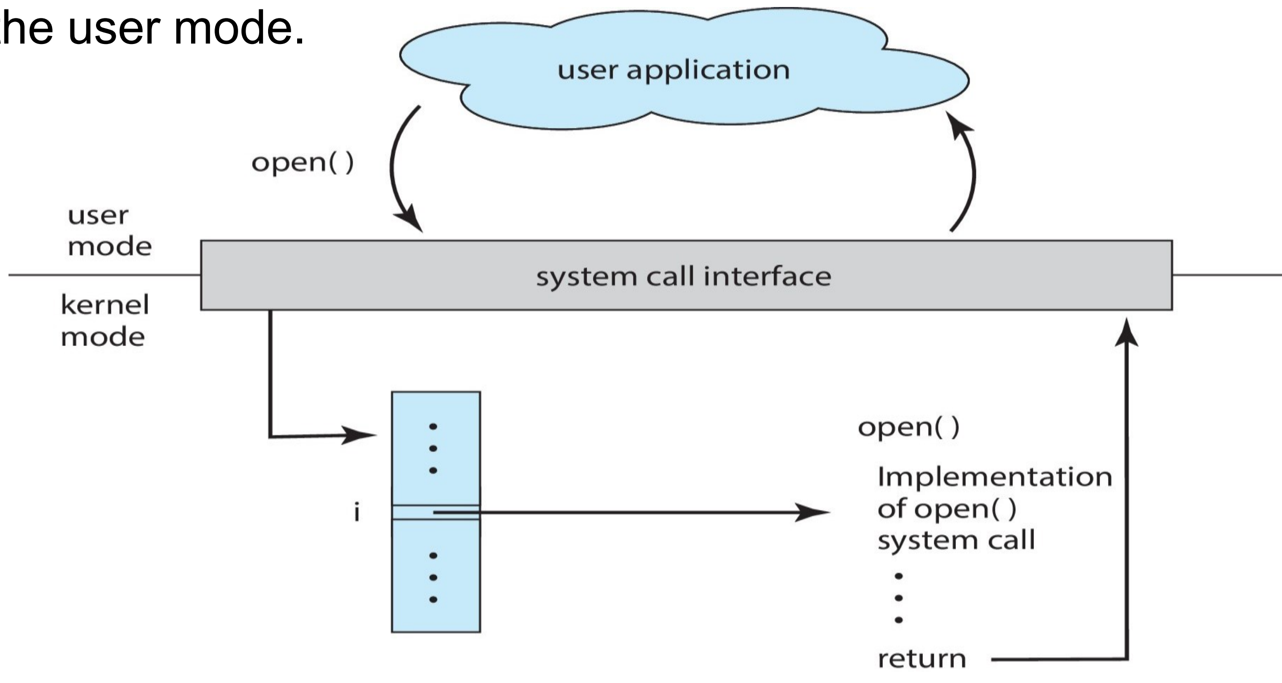


System Call Implementation

- 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

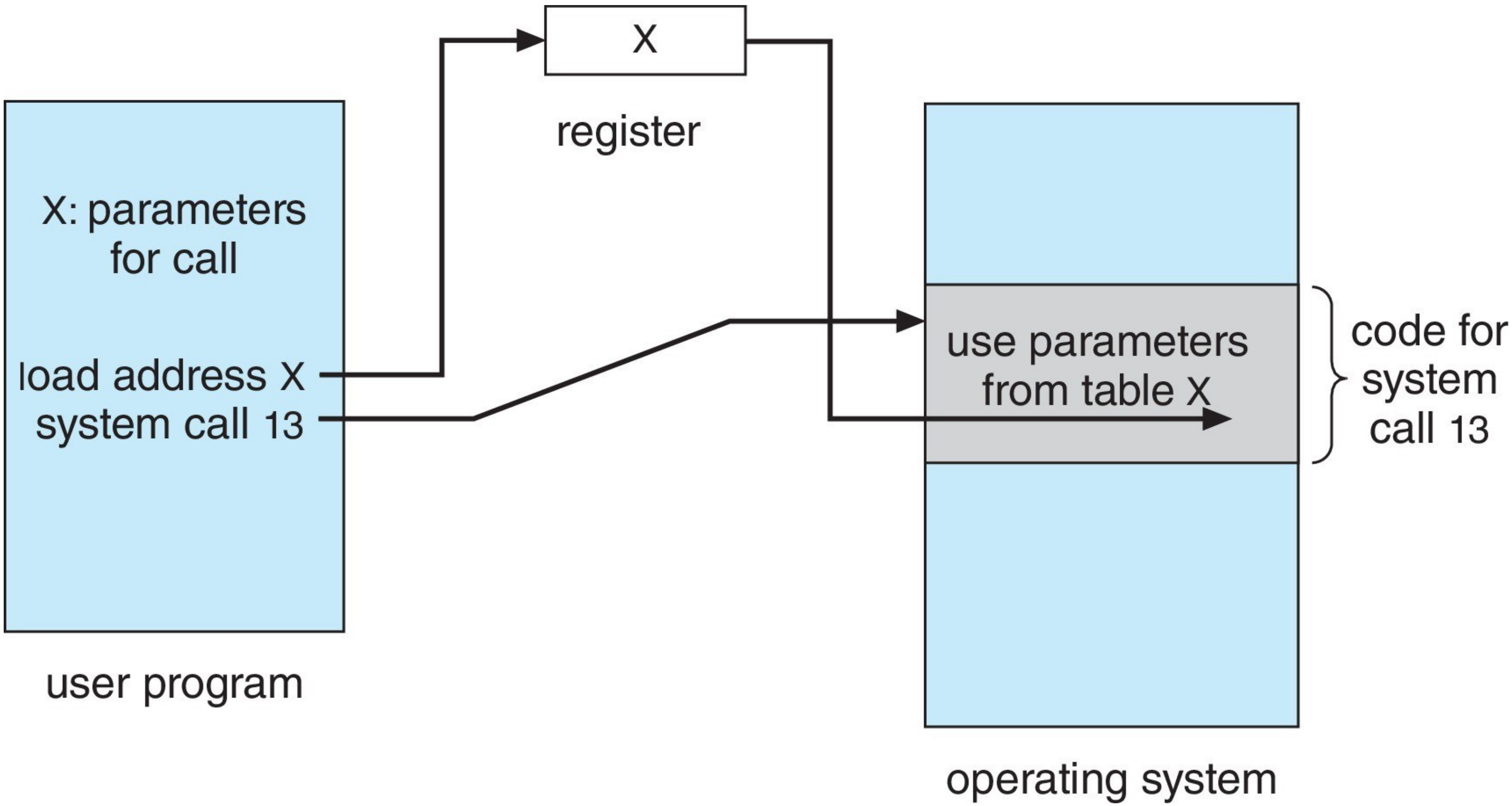
- The OS associate a number or ID to each system call, also a table at the kernel space that says for each number of system call, what should be done.
- Every time that the program execute a system call (indecently from the system call), context switch from user mode to kernel mode is done, while passing the number associated to the system call.
- OS goes to the data structure of the system call number and executes the operations.
- When done, passing to the user mode.



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

- 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

- 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

- 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()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File management	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device management	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

Information maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communications	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

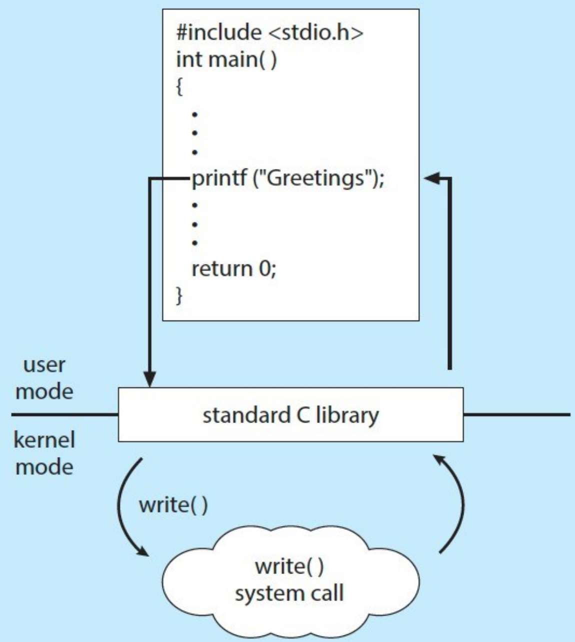
Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

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:

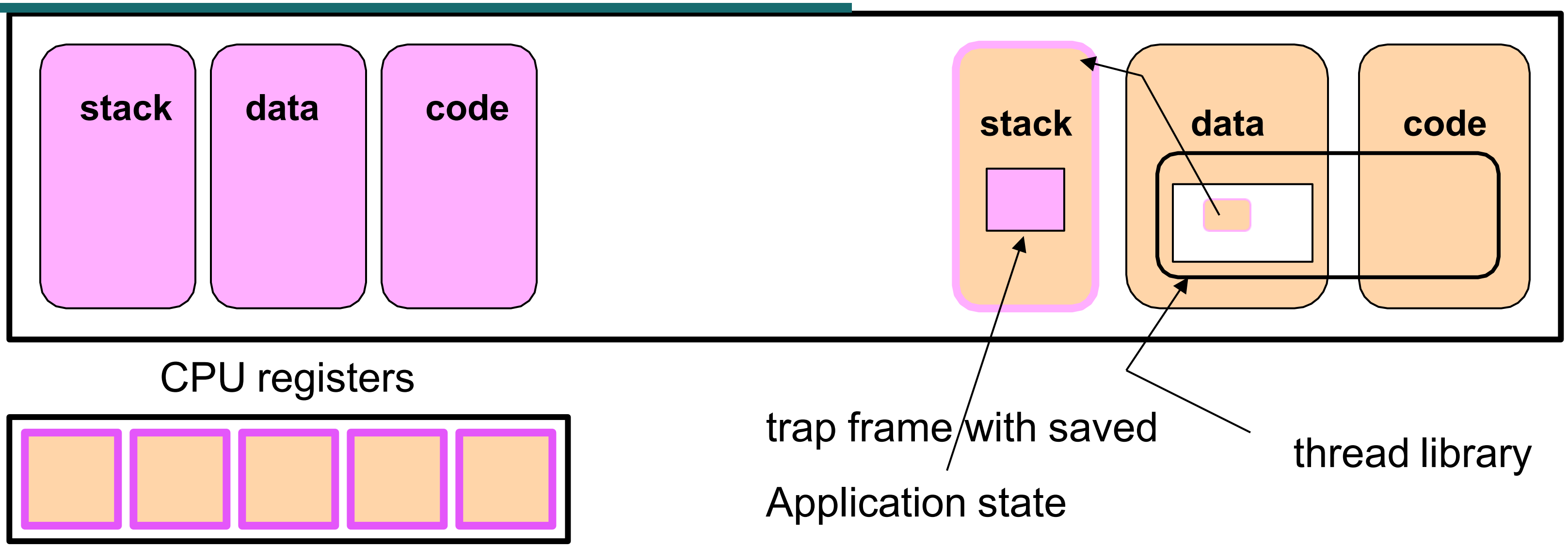


Mips trap: Handling System Calls, Exceptions, and Interrupts

- 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.

In OS161 implemented on MIPS, there is not difference between system calls, exceptions and interrupts.
In case each one of these three happens, the process is going to execute a particular code (handler).
A specific code is implemented to specify the reason of the exceptions. Therefore, three handler is defined for the three mentioned events.

OS161 Trap Frame



Every time a system call is called, the execution is passed from user mode to kernel mode. Therefore, the status of CPU should be saved through trap frame.

Trap frame will be saved at the stack of kernel thread.

So, while a thread is being executed, it is possible that the execution is interrupted (by system call, exception, interrupt).

Therefore, processors state should be saved for the execution of the system call and reloading it after.

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;
 }
```

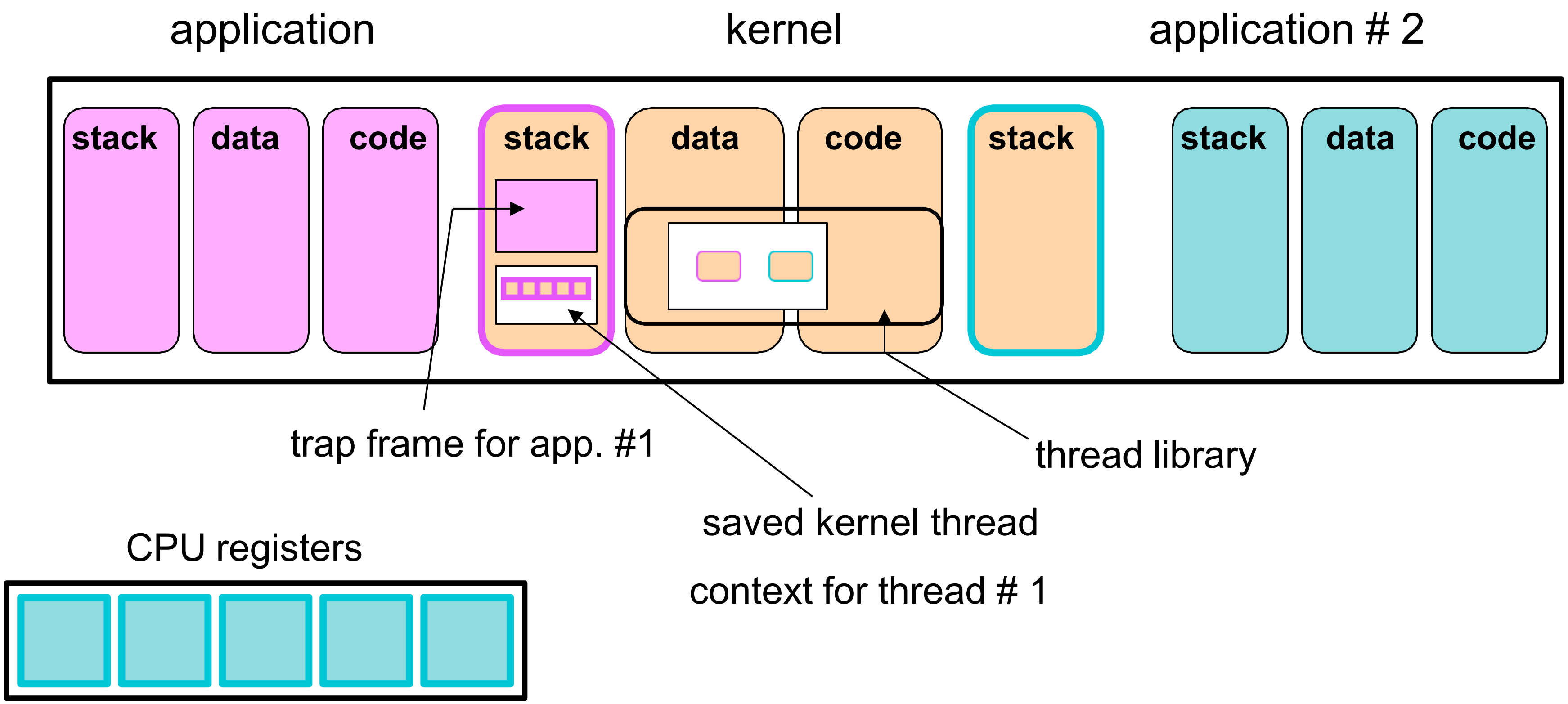
Handler for system call in MIPS through switch case:
When a system call is called, the handler is executed which save the context of the thread in a trap frame, executing the code related to the system call (#),

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



More than one program in execution:

1. Two user memory address for the two program (application).
2. Data and code section.
3. Two kernel for the two threads for storing different data structure such as switch frame or trap frame.

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

The available system calls of OS161

OS161 Memory Management

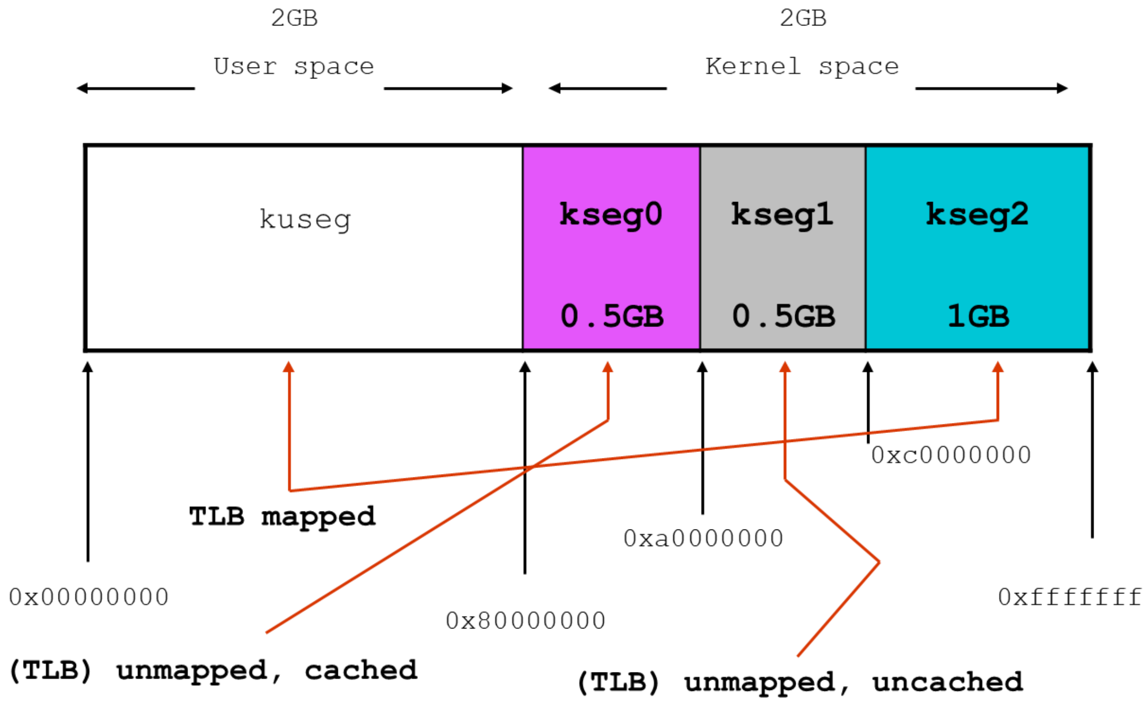
- Kernel/User address spaces
- Mips logical addresses
- Mips TLB
- DUMBVM virtual memory management
- Loading an ELF file into a (process) address space

How the memory is organized in OS161:
The memory is divided into kernel and user space.
In Mips, a small MMU and TLB is provided.
The virtual memory management is organized by DUMBVM that performs pagination on contiguous allocation, but not allocating memory (lab 02).

An Address Space for the Kernel

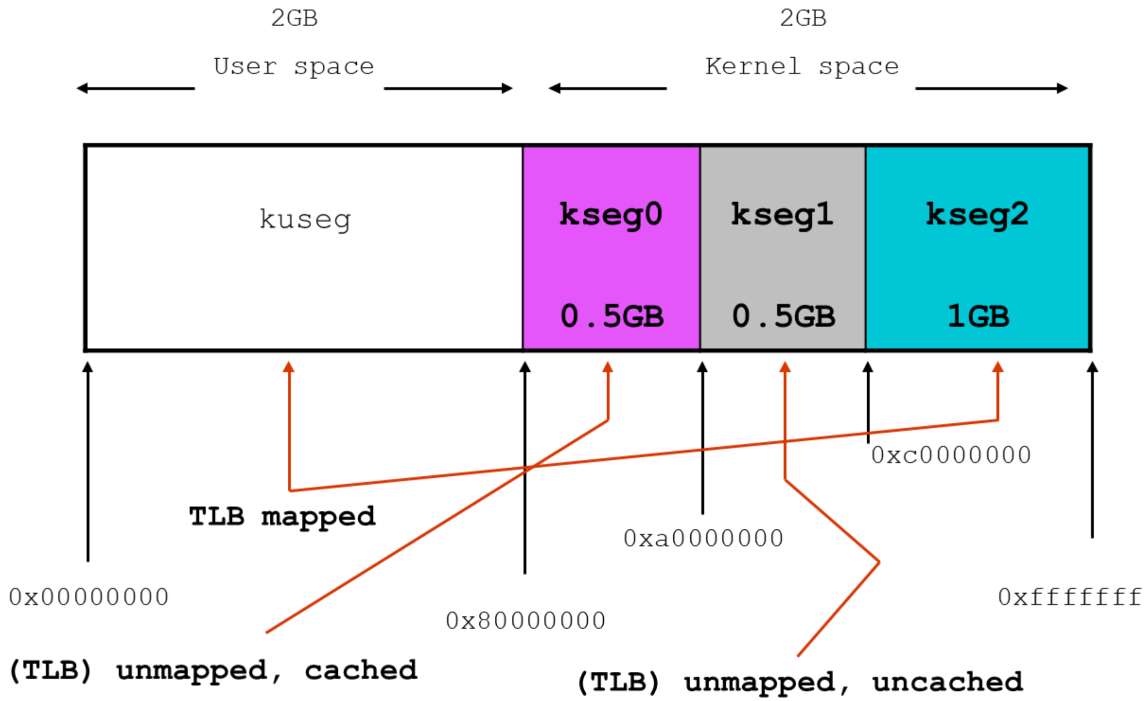
- Each process has its own address space. What about the kernel?
- two possibilities
  - **Kernel in physical space: disable address translation in privileged system**
    - execution mode, enable it in unprivileged mode
  - **Kernel in separate virtual address space: need a way to change address**
    - 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



OS161 is running on Mips, a 32 bits processor and addresses at 32 bits. Therefore, 4GB of addresses.

Address Translation on the MIPS R3000



Dividing the memory to fix sections, using some sections for user and some sections for kernel, 2 GB user space, and 2 GB kernel space.

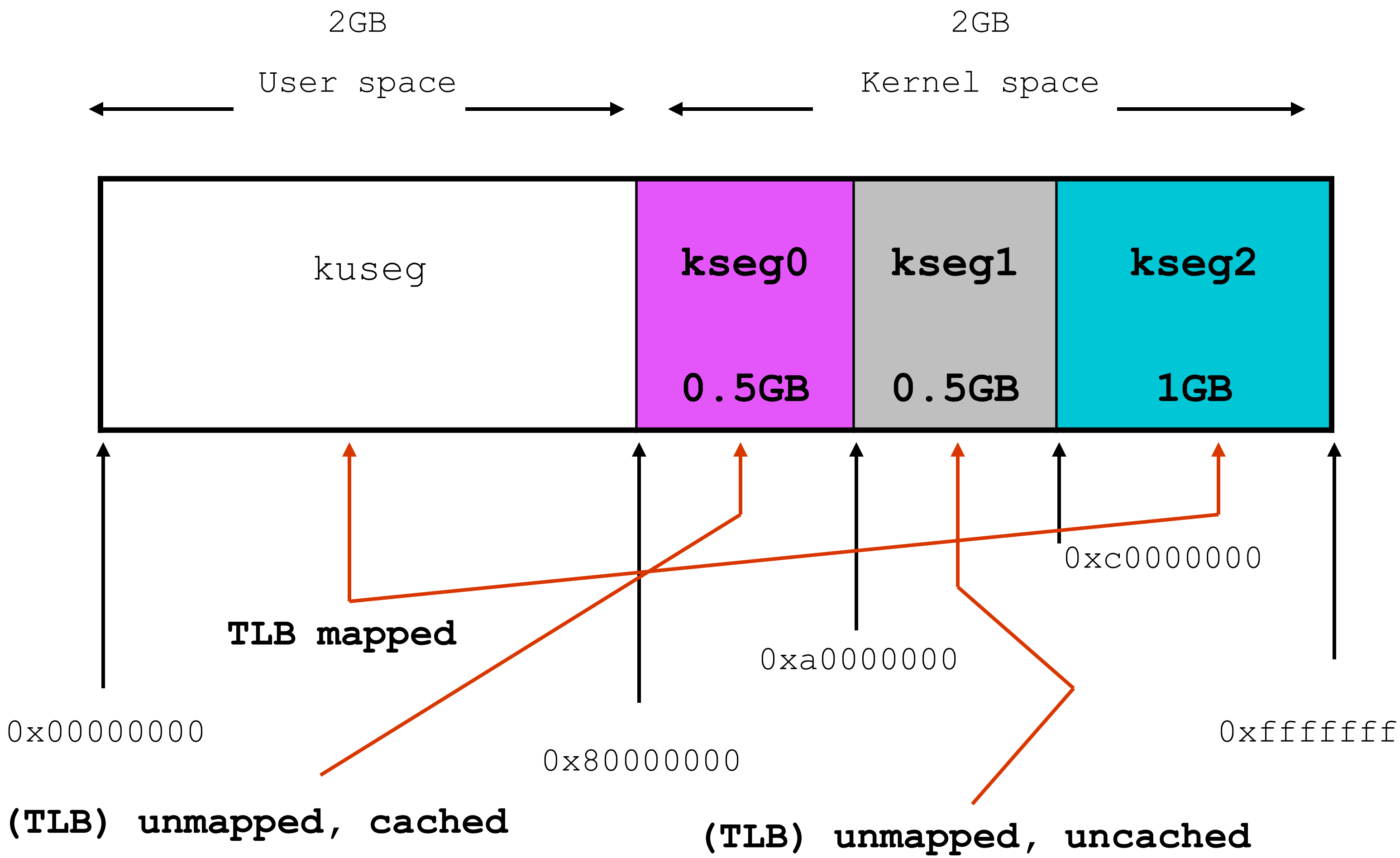
2GB kernel space is divided to subsections.

Kseg0, 0.5GB are not passed through TLB but cache (direct access to physical address).

Kseg1, 0.5 GB, not passed through TLB not cache, for mapping I/O devices.

Kseg2, 1 GB, not used in OS161.

Address Translation on the MIPS R3000



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.
- **It is not hardware and transparent.** It is software-based.
- You should call functions for using it.

- OS161 provides low-level functions for managing the TLB:

tlb_write(): modifyaspecifiedTLB entry
tlb_random(): modifya 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