Processes and the Kernel

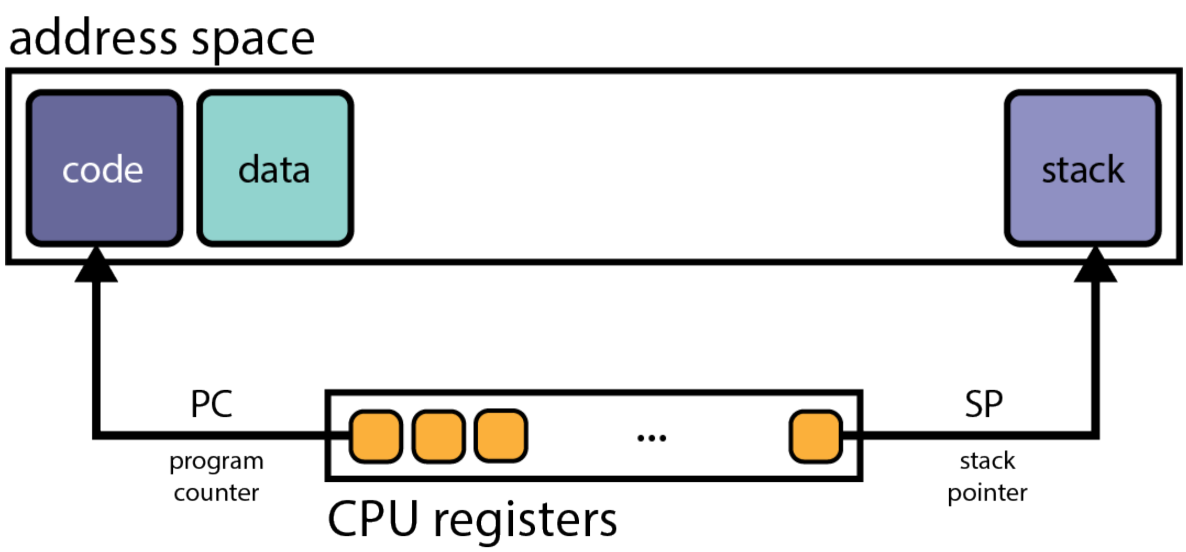
key concepts: process,system call,processor exception,fork/execv,multiprocessing

Zille Huma Kamal

David R. Cheriton School of Computer Science
University of Waterloo

Spring 2022

Recall: Sequential Program Execution

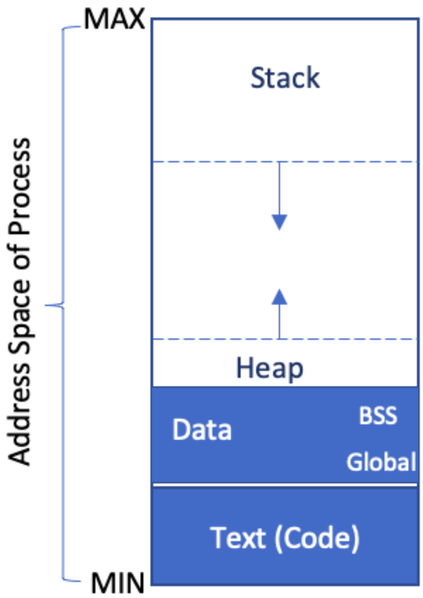


The Fetch/Execute Cycle

1. **fetch** instruction PC points to
2. decode and **execute** instruction
3. increment the PC

Process view of the world

A **process** is the execution of a program.



- a process includes virtualized **resources** :
  - virtual processor, for executing instructions
  - virtual memory, for an address space for the program's code and data
  - other resources, e.g., file and socket descriptors
- processes are created and managed by the kernel
- processes are isolated from each other but they can interact with each other: **interprocess communication**
  - **shared memory (e.g. `mmap`)**
  - **message passing (eg. pipe operator |)**
  - **sockets**

Kernel's view of a process

The kernel maintains a **process control block (PCB)** data structure for each process.

Process Related	Memory Related	File
PID	**Segment Pointers:** Text Data Stack	**File Descriptors:** Root directory Working directory Open files
State: Running Ready Blocked	**Pointers:** Base and Bound Page Table	
Context: PC SP Registers		
Scheduling Parameters: priority CPU time used		
Management Information: PPID Owner Group Creation date and time		

- a process includes virtualized **resources** :
  - virtual processor, for executing instructions
  - virtual memory, for an address space for the program's code and data
  - other resources, e.g., file and socket descriptors
- processes are created and managed by the kernel
- processes are isolated from each other but they can interact with each other: **interprocess communication**
  - **shared memory (e.g. `mmap`)**
  - **message passing (eg. pipe operator |)**
  - **sockets**

Process Management Calls

Processes can be created, managed, and destroyed. Each OS supports a variety of functions to perform these tasks.

	Linux	OS/161
Creation	fork,execv	fork,execv
Destruction	_exit,kill	_exit
Synchronization	wait,waitpid,pause,...	waitpid
Attribute Mgmt	getpid,getuid,nice,getrusage,...	getpid

The OS/161 process management calls are **NOT** implemented yet.

fork,_exit

- `int fork (void);`
  - `fork` creates a new process (the **child**) that is a clone of the original (the **parent**)
  - after `fork`, both parent and child are executing copies of the same program
  - virtual memories of parent and child are identical at the time of the fork, but may diverge afterwards
  - `fork` is called by the parent, but returns in **both** the parent and the child
  - parent and child see different return values from fork
- `_exit` terminates the process that calls it
  - process can supply an exit status code when it exits
  - kernel records the exit status code in case another process asks for it (via `waitpid`)

waitpid

- `int waitpid (int pid, int *stat, int opt);`
  - pid – process to wait for, or -1 for any
  - stat – will contain exit value, or signal
  - opt – usually 0 or `WNOHANG`
  - Returns process ID or -1 on error
  - waitpid lets a process wait for another to terminate, and retrieve its exit status code

The fork, _exit, getpid and waitpid system call - example

```
main() {
   rc = fork();  /* returns 0 to child, pid to parent */
   if (rc == 0) {  /* child executes this code */
      my_pid = getpid();
      x = child_code();
      _exit(x);
   } else {   /* parent executes this code */
     child_pid = rc;
     parent_pid = getpid();
     parent_code();
     p = waitpid(child_pid,&child_exit,0);
     if (WIFEXITED(child_exit))
       printf("child exit status was %d\n",
              WEXITSTATUS(child_exit))
   }
}
```

The execv system call - run a process

In Linux, `execv` has many variants:

- `int execve (char *prog, char **argv, char **envp)`
  envp – environment variables, e.g., PATH, HOME
- `int execvp (char *prog, char **argv);`
  Search `PATH` for prog, use current environment
- `int execlp (char *prog, char *arg, ...);`
  List arguments one at a time, finish with `NULL`
- `prog` – full pathname of program to run
- `argv` – argument vector that gets passed to `main`
- `envp` – environment variables, e.g.,

The execv system call - run a process (continued)

- Generally called through wrapper functions
- `execv` changes the program that a process is running
- The calling process's current virtual memory is destroyed
- The process gets a new virtual memory, initialized with the code and data of the new program to run
- After `execv`, the new program starts executing

> The process ID stays the same.

> `execv` can pass arguments to the new program, if required

execv example

```
int main()
{
  int rc = 0;
  char *args[4];

  args[0] = (char *) "/testbin/argtest";
  args[1] = (char *) "first";
  args[2] = (char *) "second";
  args[3] = 0;

  rc = execv("/testbin/argtest", args);
  printf("If you see this execv failed\n");
  printf("rc = %d errno = %d\n", rc, errno);
  exit(0);
}
```


Combining fork and execv - an example

```
main()
{
   char *args[4];
   /* set args here */
   rc = fork();  /* returns 0 to child, pid to parent */
   if (rc == 0) {
     status = execv("/testbin/argtest",args);
     printf("If you see this execv failed\n");
     printf("status = %d errno = %d\n", status, errno);
     exit(0);
   } else {
     child_pid = rc;
     parent_code();
     p = waitpid(child_pid,&child_exit,0);
   }
}
```

Putting it all together - a simplified shell (minish.c)

Parent Process (PID 5)

```
pid_t pid; char **av;
void doexec() {
  execvp(av[0], av);
  perror(av[0]);
  exit(1);
}

    /* ... main loop: */
    for (;;) {
      parse_input(&av, stdin);
      switch (pid = fork()) {
      case -1:
        perror("fork"); break;
      case 0:
        doexec();
      default: // ← After Fork (pid = 5)
        waitpid(pid, NULL, 0); break;
      }
    }
```

Child Process (PID 6)

```
pid_t pid; char **av;
void doexec() {
  execvp(av[0], av);
  perror(av[0]);
  exit(1);
}

    /* ... main loop: */
    for (;;) {
      parse_input(&av, stdin);
      switch (pid = fork()) {
      case -1:
        perror("fork"); break;
      case 0: // ← [PID=6] After Fork
        doexec();
      default:
        waitpid(pid, NULL, 0); break;
      }
    }
```


Inter-Process Communication (IPC)

Processes are isolated from each other. But, what if they want to communicate (share data) with each other?

IPC or inter-process communication is a family of methods used to send data between processes.

- **File:** data to be shared is written to a file, accessed by both processes
- **Socket:** data is sent via network interface between processes
- **Pipe:** data is sent, unidirectionally, from one process to another via OS-managed data buffer
- **Shared Memory:** data is sent via block of shared memory visible to both processes
- **Message Passing/Queue:** a queue/data stream provided by the OS to send data between processes

Interprocess communication - e.g. pipe operator |

Interprocess

- Manipulating file descriptors
- `int dup2 (int oldfd, int newfd);`
  - Closes `newfd`, if it was a valid descriptor
  - Makes `newfd` an exact copy of `oldfd`
  - Two file descriptors will share same offset
- Example: `redirsh.c`
  - Loop that reads a command and executes it
  - Recognizes input, output redirection

Example - redirsh.c

```c
void doexec (void) {
  int fd;
  if (infile) {    /* non-NULL for "command < infile" */
    if ((fd = open(infile, O_RDONLY)) < 0) {
      perror(infile);
      exit(1);
    }
    if (fd != 0) {
      dup2(fd, 0);
      close(fd);
    }
  }

  /* ... do same for outfile→fd 1, errfile→fd 2 ... */
  execvp (av[0], av);
  perror (av[0]);
  exit (1);
}
```

Deleting Processes

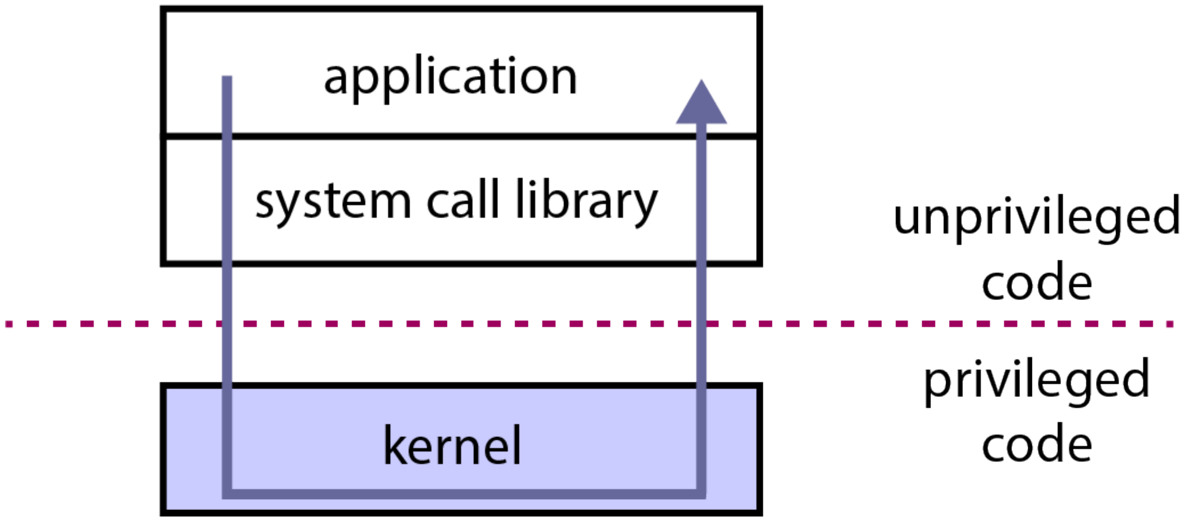
- `void exit (int status);`
  - Current process ceases to exist
  - `status` shows up in `waitpid` (shifted)
  - By convention, `status` of 0 is success, non-zero error
- `int kill (int pid, int sig);`
  - Sends signal `sig` to process `pid`
  - `SIGTERM (sig=15)` most common value, kills process by default (but application can catch it for "cleanup")
  - `SIGKILL (sig=9)` stronger, always kills a process, e.g. from the shell `kill -9 pid`

System Calls

Process management calls, e.g., `fork`, are called by user programs. They are also **system calls**. **System calls are the interface between processes and the kernel.**

Service	**OS/161 Examples**
create,destroy,manage processes	`fork,execv,waitpid,getpid`
create,destroy,read,write files	`open,close,remove,read,write`
manage file system and directories	`mkdir,rmdir,link,sync`
interprocess communication	`pipe,read,write`
manage virtual memory	`sbrk`
query,manage system	`reboot,__time`

System Call Software Stack



Kernel Privilege

- The CPU implements different levels (or rings) of **execution privilege** as a security and isolation mechanism.
- Kernel code runs at the highest privilege level.
- Application code runs at a lower privilege level because user programs should **not** be permitted to perform certain tasks such as:
  - modifying the page tables that the kernel uses to implement process virtual memories (address spaces)
  - halting the CPU
- Programs cannot execute code or instructions belonging to a higher-level of privilege. These restrictions allow the kernel to keep processes isolated from one another - and from the kernel.
  - Application programs cannot directly call kernel functions or access kernel data structures.

> The Meltdown vulnerability found on Intel chips lets user applications bypass execution privilege and access any address in physical memory.

How System Calls Work (Part 1)

> Since application programs can't directly call the kernel, how does a program make a system call such as `fork`?

- There are only two things that make kernel code run:
  1. **Interrupts**
     - interrupts are generated by devices when they need attention
  2. **Exceptions**
     - exceptions are caused by instruction execution when a running program needs attention

Recall: Interrupts

- Interrupts are raised by devices (hardware)
- An interrupt causes the hardware to transfer control to a fixed location in memory, where an **interrupt handler** is located
- Interrupt handlers are part of the kernel
  - If an interrupt occurs while an application program is running, control will jump from the application to the kernel's interrupt handler
- When an interrupt occurs, the processor switches to privileged execution mode when it transfers control to the interrupt handler
  - This is how the kernel gets its execution privilege

Exceptions

- Exceptions are conditions that occur during the execution of a program instruction.
  - Examples: arithmetic overflows, illegal instructions, or page faults (to be discussed later).
- Exceptions are detected by the CPU during instruction execution
- The CPU handles exceptions like it handles interrupts:
  - control is transferred to a fixed location, where an **exception handler** is located
  - the processor is switched to privileged execution mode
- The exception handler is part of the kernel

MIPS Exception Types

```
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 */
```

> On the MIPS, the **same** mechanism handles exceptions and interrupts, and there is a single handler for both in the kernel. The handler uses these codes to determine what triggered it to run.

How System Calls Work (Part 2)

- To perform a system call, the application program needs to cause an exception to make the kernel execute:
  - on the MIPS, `EX_SYS` is the system call exception
- To cause this exception on the MIPS, the application executes a special purpose instruction: `syscall`
  - other processor instruction sets include similar instructions, e.g., `syscall` on x86
- The kernel's exception handler checks the exception code (set by the CPU when the exception is generated) to distinguish system call exceptions from other types of exceptions.

Which System Call?

- There is only one `syscall` exception. `fork` and `getpid` are both system calls. How does the kernel know **which** system call the application is requesting?
- **Answer:** system call codes
  - the kernel defines a code for each system call it understands
  - the kernel expects the application to place a code in a specified location before executing the `syscall` instruction
    - for OS/161 on the MIPS, the code goes in register `v0`
  - the kernel's exception handler checks this code to determine which system call has been requested
  - the codes and code location are part of the **kernel ABI** (Application Binary Interface)

Example: loading a system call code

Example: `li v0, 0` loads the system call code for `fork` into `v0`.

Some OS/161 System Call Codes

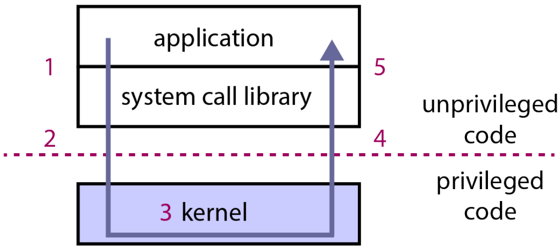
```
...
#define SYS_fork         0
#define SYS_vfork        1
#define SYS_execv        2
#define SYS__exit        3
#define SYS_waitpid      4
#define SYS_getpid       5
...
```

This comes from `kern/include/kern/syscall.h`. The files in `kern/include/kern` define things (like system call codes) that must be known by both the kernel and applications.

System Call Parameters

- System calls take parameters and return values, like function calls. How does this work, since system calls are really just exceptions?
- **Answer:** The application places parameter values in kernel-specified locations before the `syscall`, and looks for return values in kernel-specified locations after the exception handler returns
  - The locations are part of the kernel ABI
  - Parameter and return value placement is handled by the application system call library functions
  - On MIPS, parameters go in registers a0,a1,a2,a3
    - result success/fail code is in a3 on return
    - return value or error code is in `v0` on return

System Call Software Stack (again)



System calls are expensive

Which is faster?

N separate `print` calls, or forming a string of *N* numbers and a single `print`.

1. application calls library wrapper function for desired system call
2. library function performs `syscall` instruction
3. kernel exception handler runs
   (a) creates trap frame to save application program state
   (b) determines that this is a system call exception
   (c) determines which system call is being requested
   (d) does the work for the requested system call
   (e) restores the application program state from the trap frame
   (f) returns from the exception
4. library wrapper function finishes and returns from its call
5. application continues execution

User and Kernel Stacks

- Every OS/161 process thread has two stacks, although it only uses one at a time
  - **User (Application) Stack:** used while application code is executing
    - this stack is located in the application's virtual memory
    - it holds activation records for application functions
    - the kernel creates this stack when it sets up the virtual address memory for the process
  - **Kernel Stack:** used while the thread is executing kernel code, after an exception or interrupt
    - this stack is a kernel structure
    - in OS/161, the `t_stack` field of the `thread` structure points to this stack
    - this stack holds activation records for kernel functions
    - this stack also holds **trap frames** and **switch frames** (because the kernel creates trap frames and switch frames)

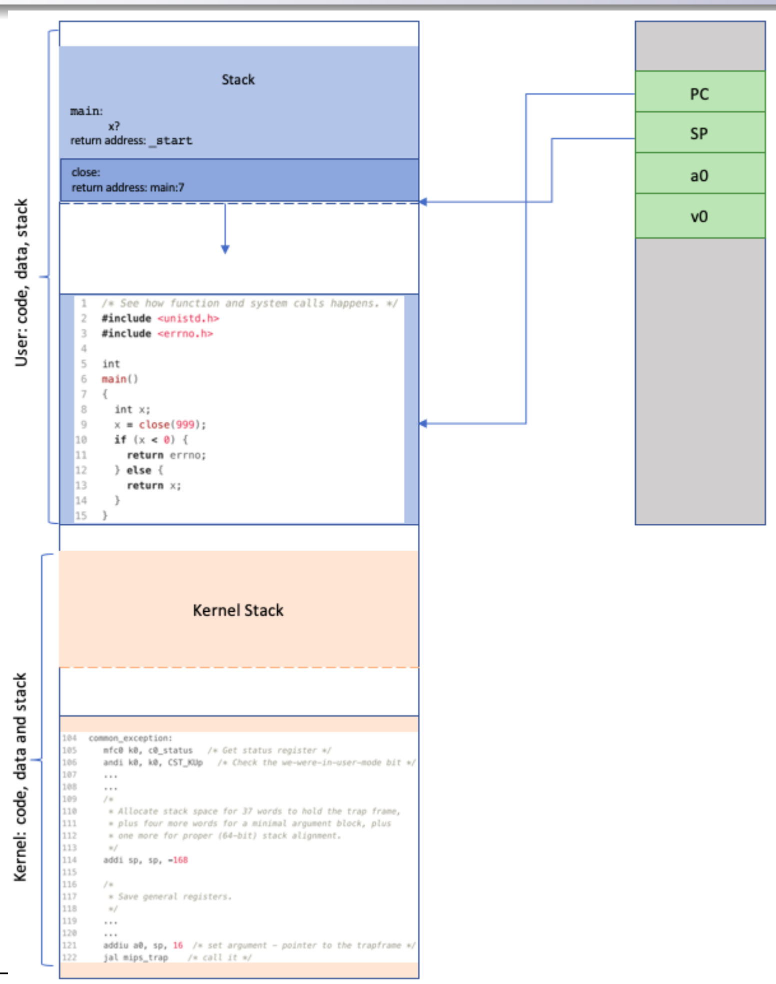
Exception Handling in OS/161

- first to run is careful assembly code that
  - saves the application stack pointer
  - switches the stack pointer to point to the thread's kernel stack
  - carefully saves application state and the address of the instruction that was interrupted in a trap frame on the thread's kernel stack
  - calls `mips_trap`, passing a pointer to the trap frame as a parameter
- after `mips_trap` is finished, the handler will
  - restore application state (including the application stack pointer) from the trap frame on the thread's kernel stack
  - jump back to the application instruction that was interrupted, and switch back to unprivileged execution mode
- see `kern/arch/mips/locore/exception-mips1.S`

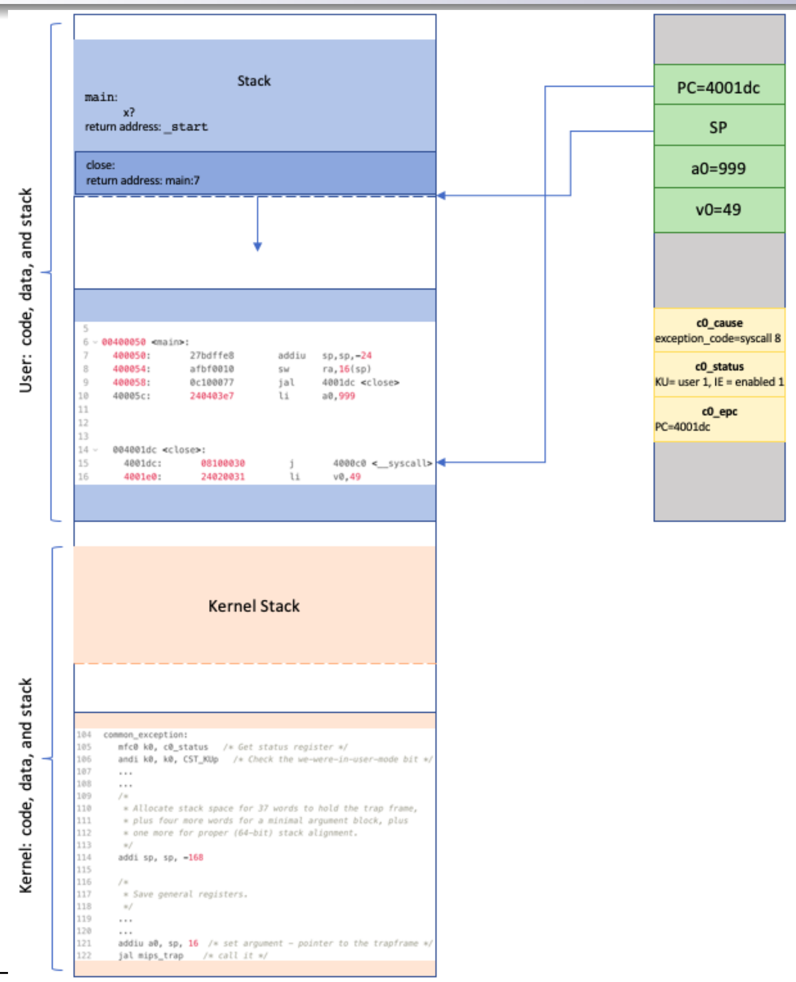
mips_trap

- `mips_trap` determines what type of exception this is by looking at the exception code: interrupt? system call? something else?
- there is a separate handler in the kernel for each type of exception:
  - interrupt? call `mainbus_interrupt`
  - address translation exception? call `vm_fault` (important for later assignments!)
  - system call? call `syscall` (kernel function), passing it the trap frame pointer
  - syscall is in `kern/arch/mips/syscall/syscall.c`
- see `kern/arch/mips/locore/trap.c`

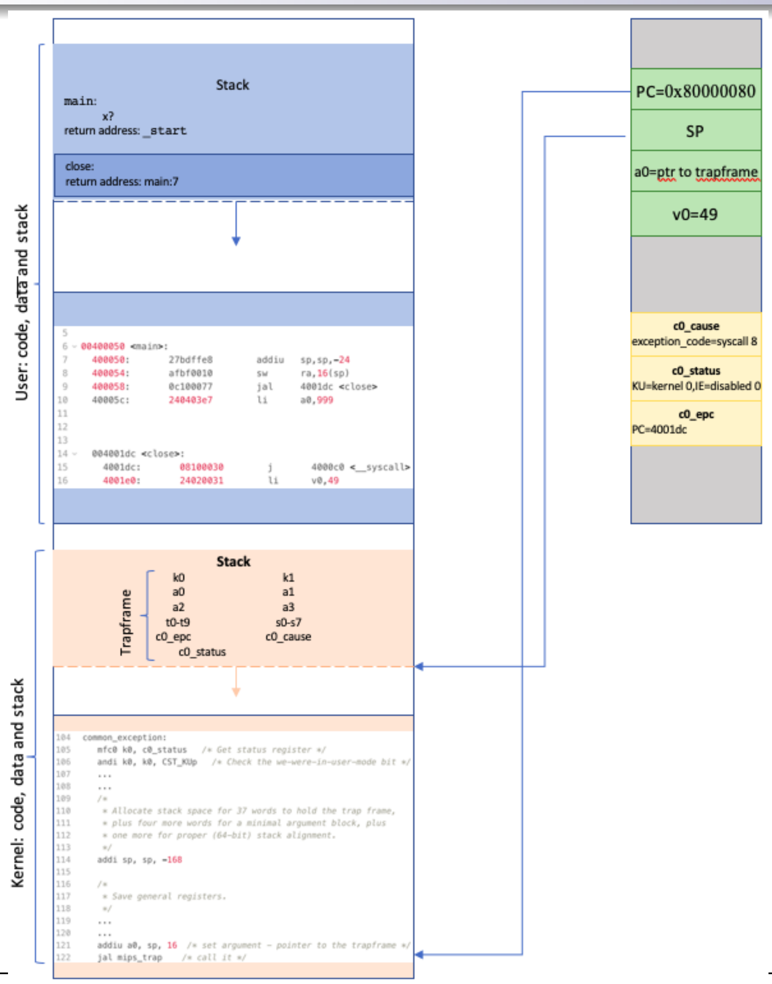
Tracing a system call



Tracing a system call



Tracing a system call

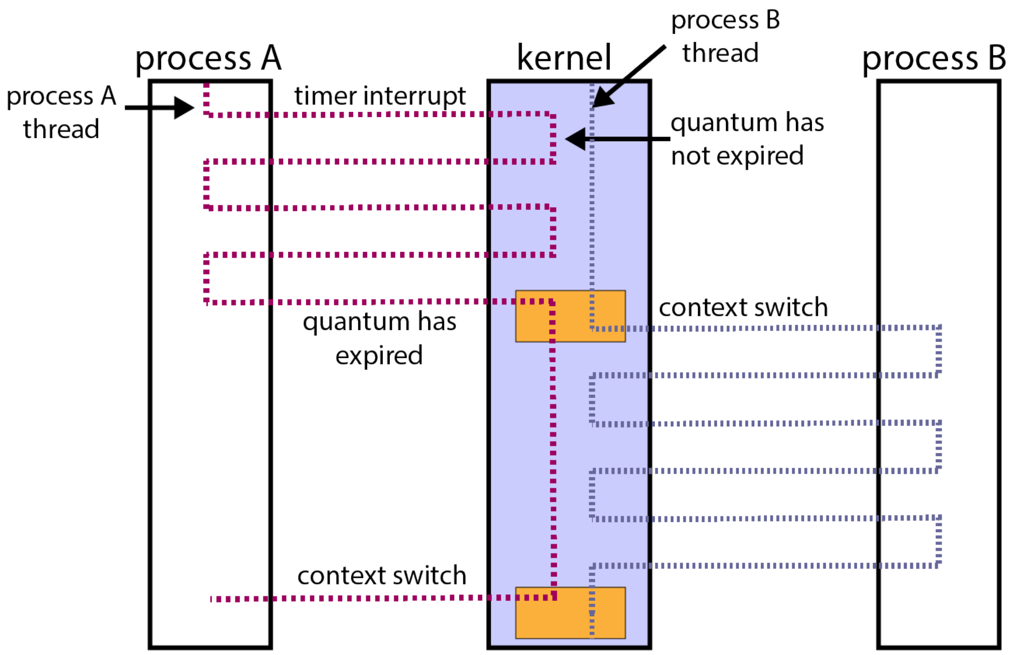


Multiprocessing

- Multiprocessing (or multitasking) means having multiple processes existing at the same time
- All processes share the available hardware resources, with the sharing coordinated by the operating system:
  - Each process' virtual memory is implemented using some of the available physical memory. The OS decides how much memory each process gets.
  - Each process' threads are scheduled onto the available CPUs (or CPU cores) by the OS.
  - Processes share access to other resources (e.g., disks, network devices, I/O devices) by making system calls. The OS controls this sharing.
- The OS ensures that processes are isolated from one another. Interprocess communication should be possible, but only at the explicit request of the processes involved.

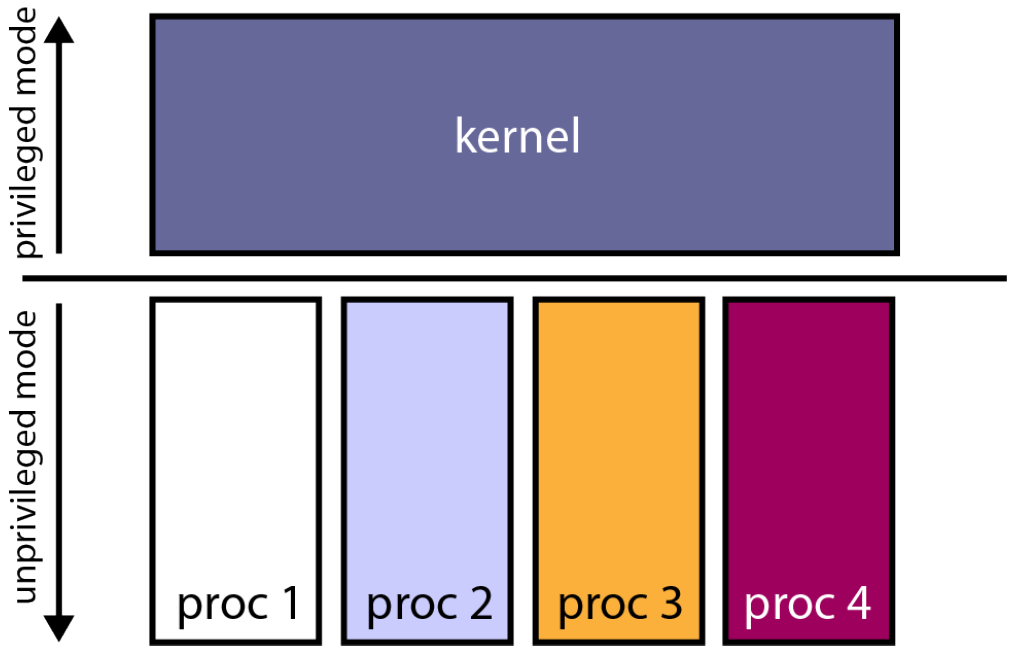
> Processes can have many threads, but must have at least one to execute. OS/161 only supports a single thread per process.

Two-Process Example

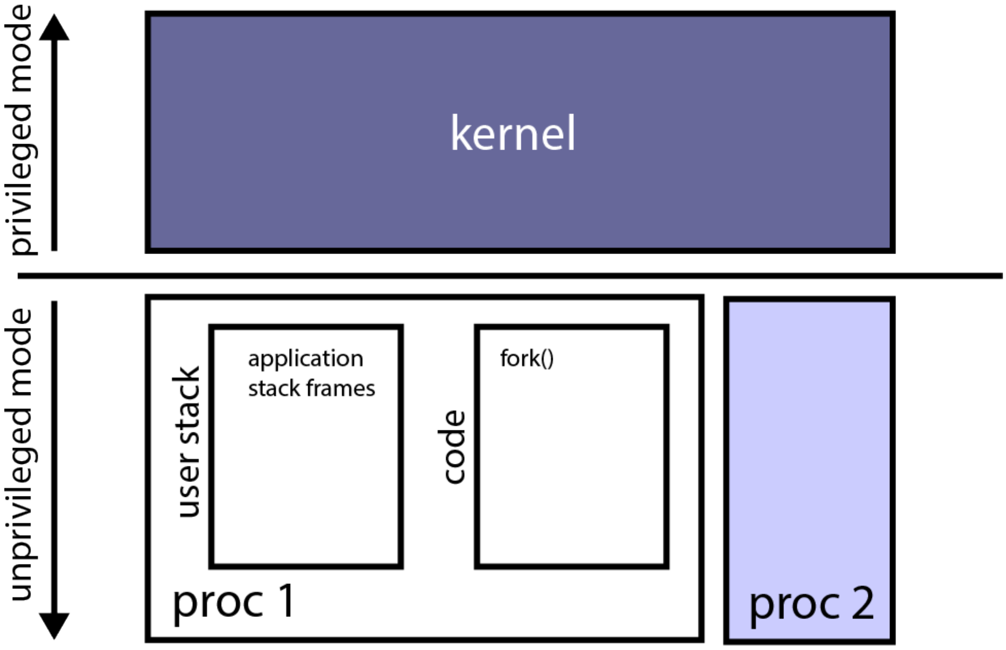


Threads "waiting in" the kernel are **ready**.

Example: System Calls (1/27)

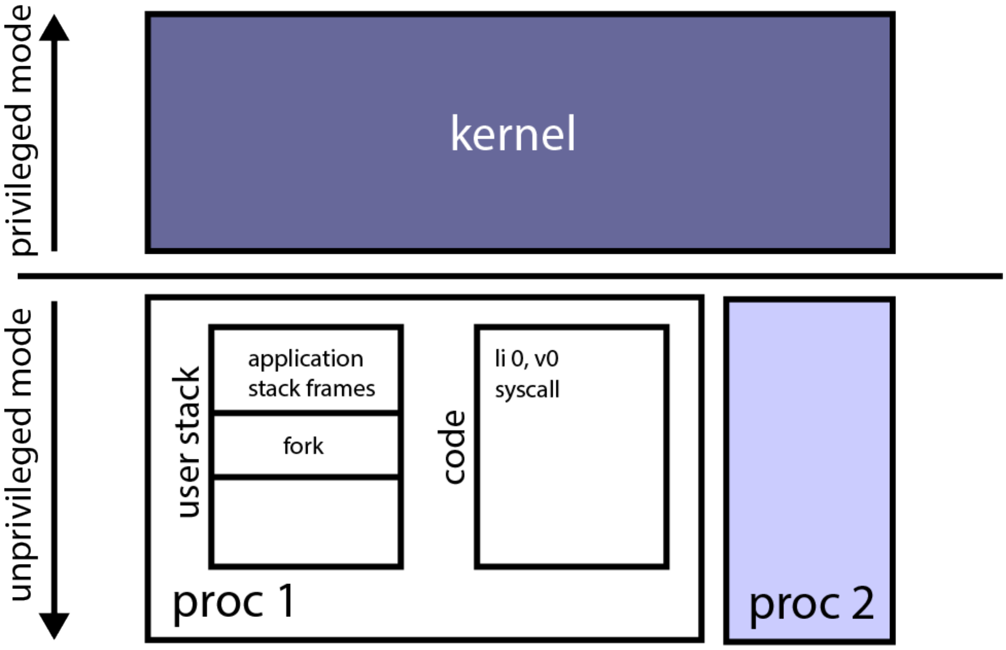


Example: System Calls (2/27)



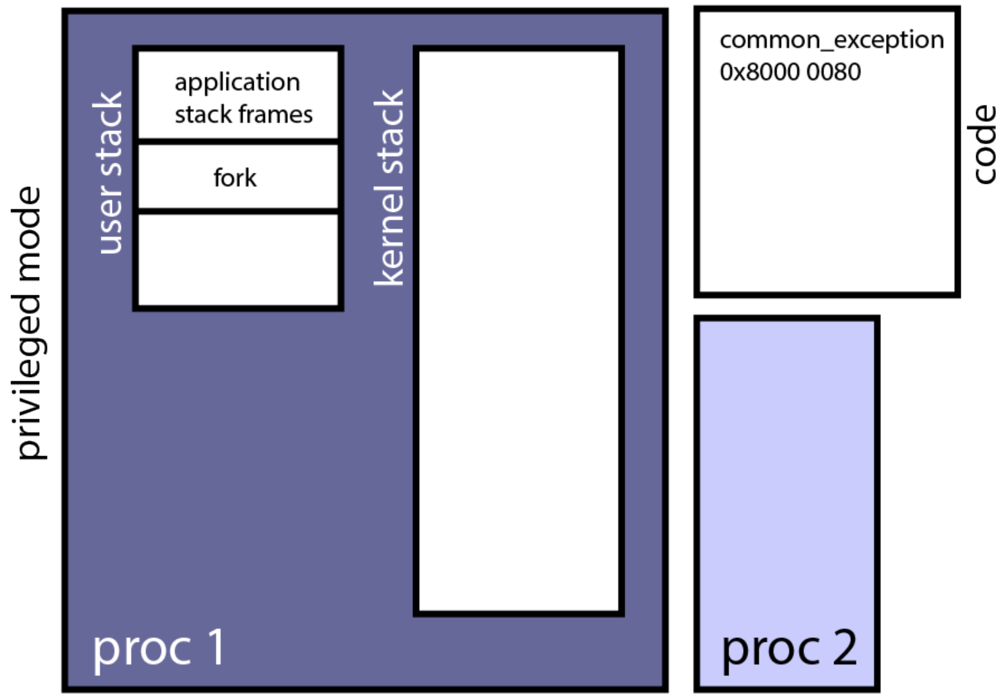
Proc A calls `fork`, a system call.

Example: System Calls (3/27)



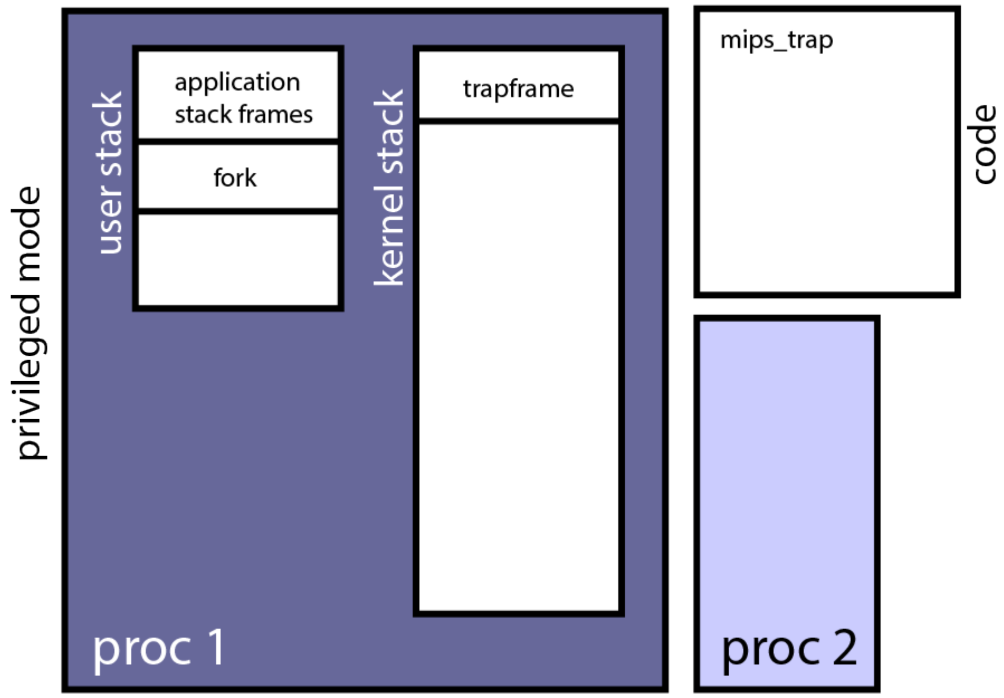
`fork` is a system call library function. It puts the system call code in register `v0` and raises the exception.

Example: System Calls (4/27)



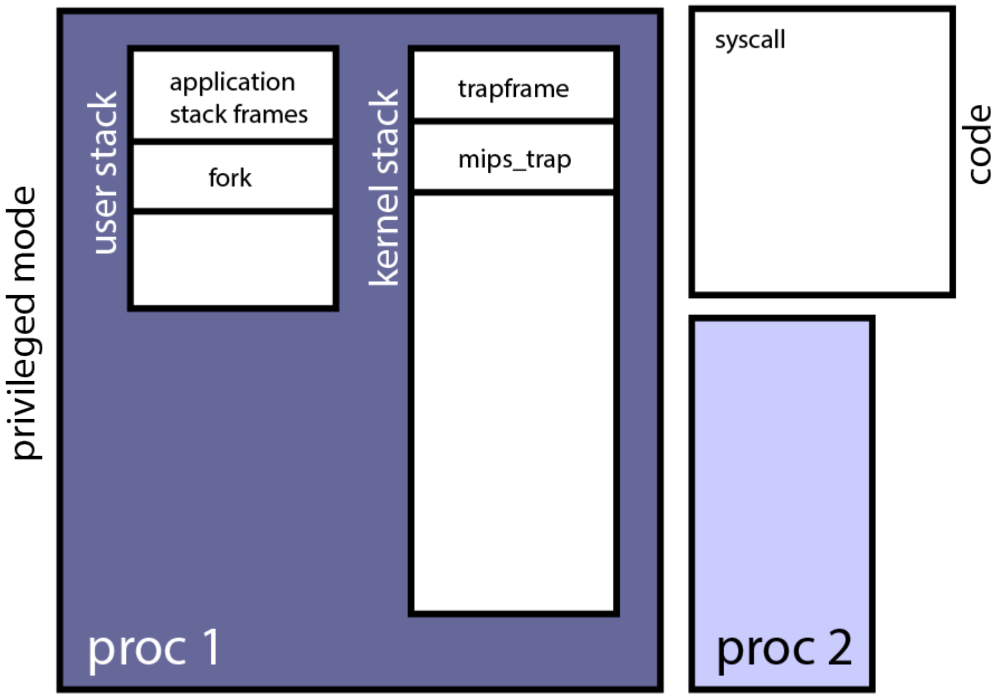
Exception is raised, the CPU executes `common_exception`. The CPU goes into privileged mode and interrupts are turned off. Switch from user to kernel stack. Save trapframe.

Example: System Calls (5/27)



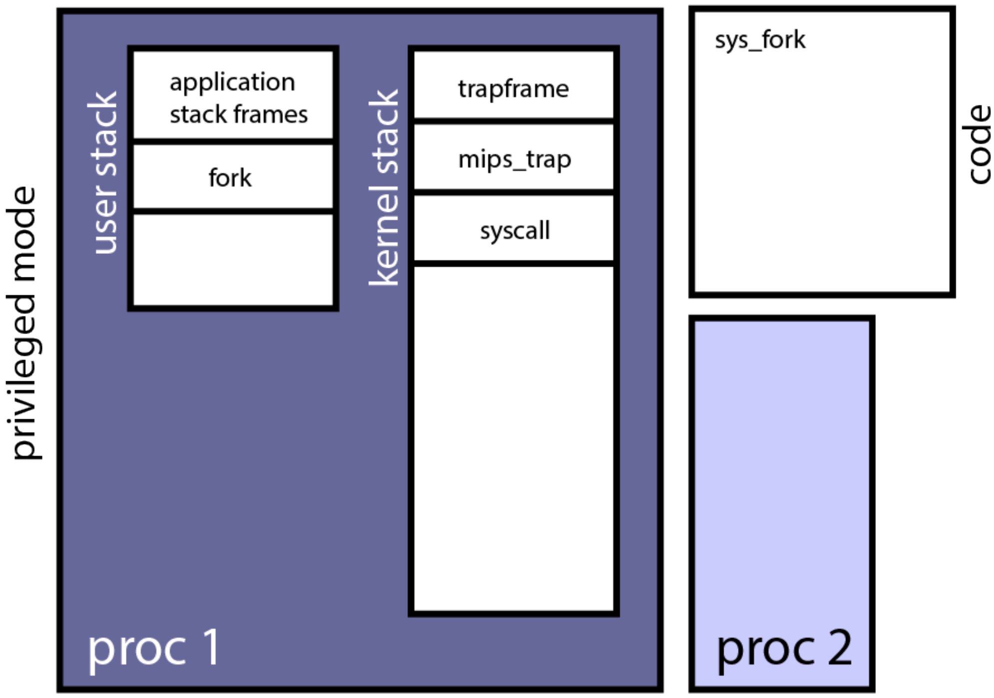
After saving the state `common_exception` calls `mips_trap` to determine what kind of exception was raised. For a system call, turn interrupts back on.

Example: System Calls (6/27)



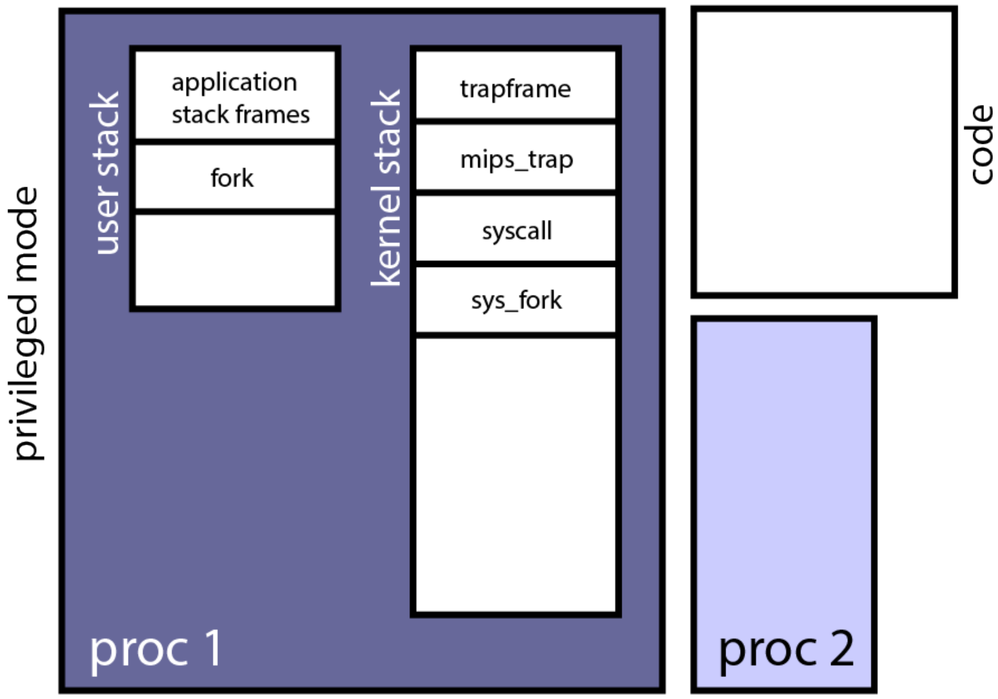
`mips_trap` determines exception is a system call. Calls `syscall`, a kernel function to dispatch the correct function.

Example: System Calls (7/27)



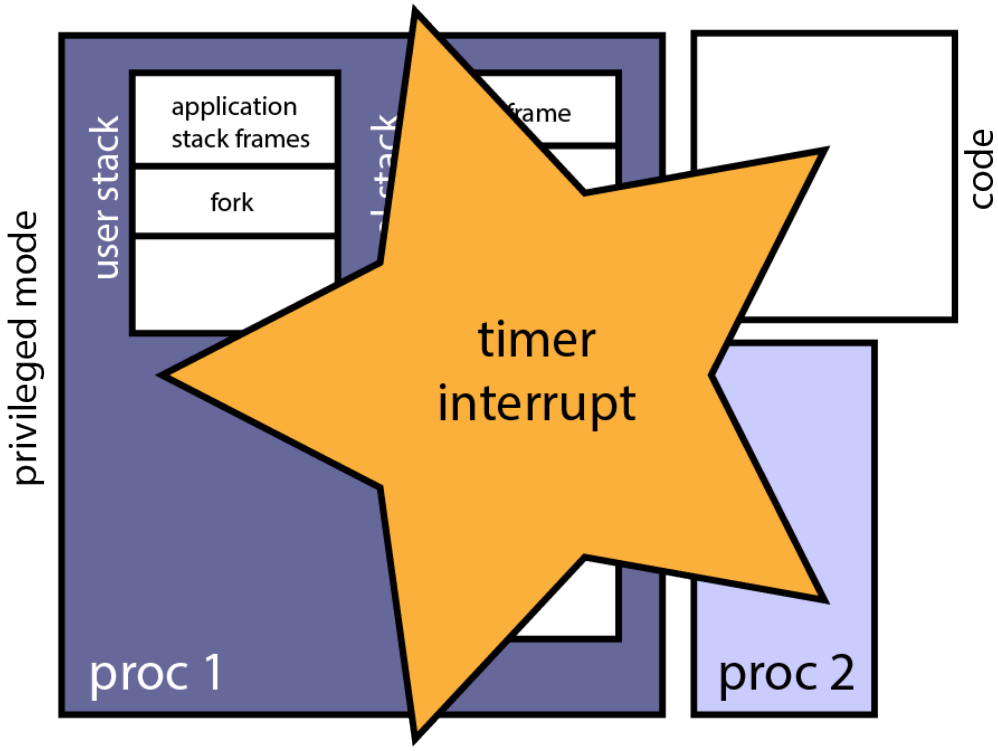
`syscall`, the system call dispatcher, calls the appropriate handler for the system call code provided in `v0`. In this case, `sys_fork` is called.

Example: System Calls (8/27)



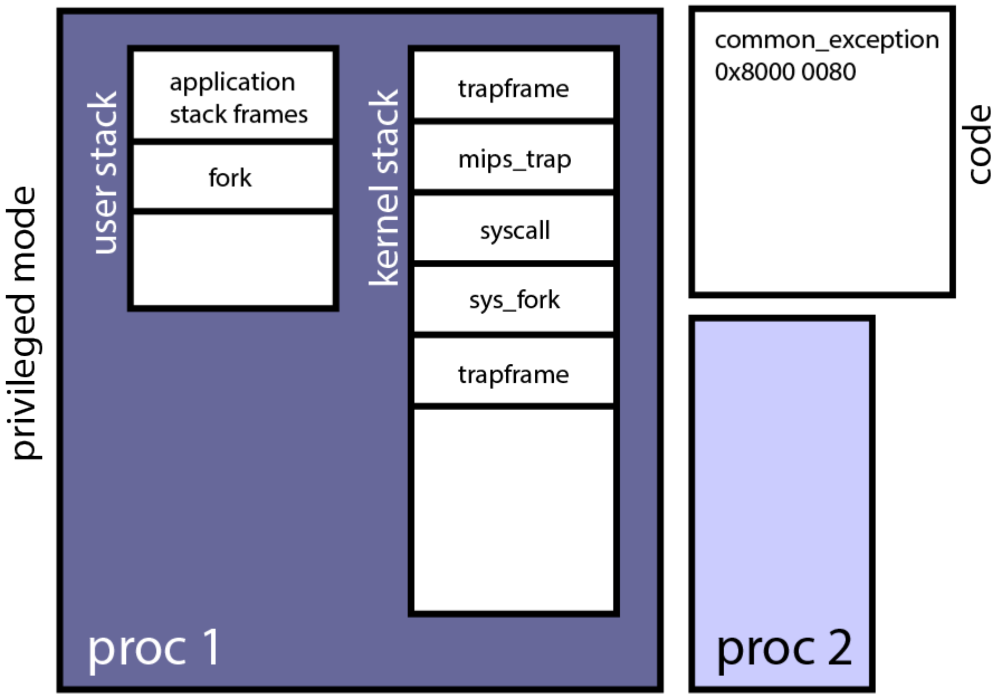
The system call is finally executed by the kernel.

Example: System Calls (9/27)



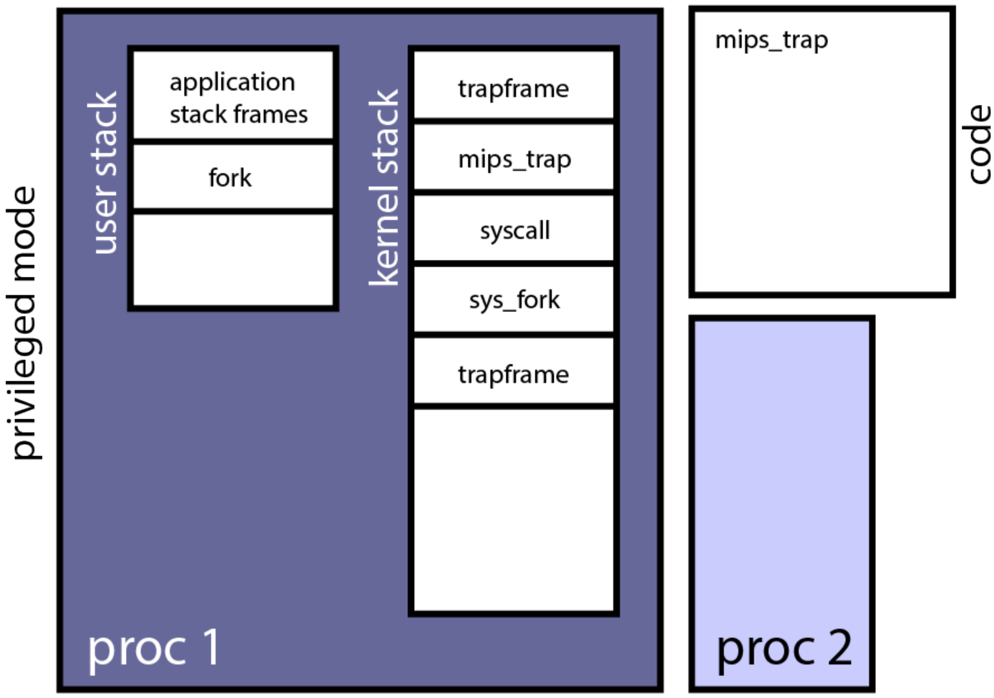
A timer interrupt occurs.

Example: System Calls (10/27)



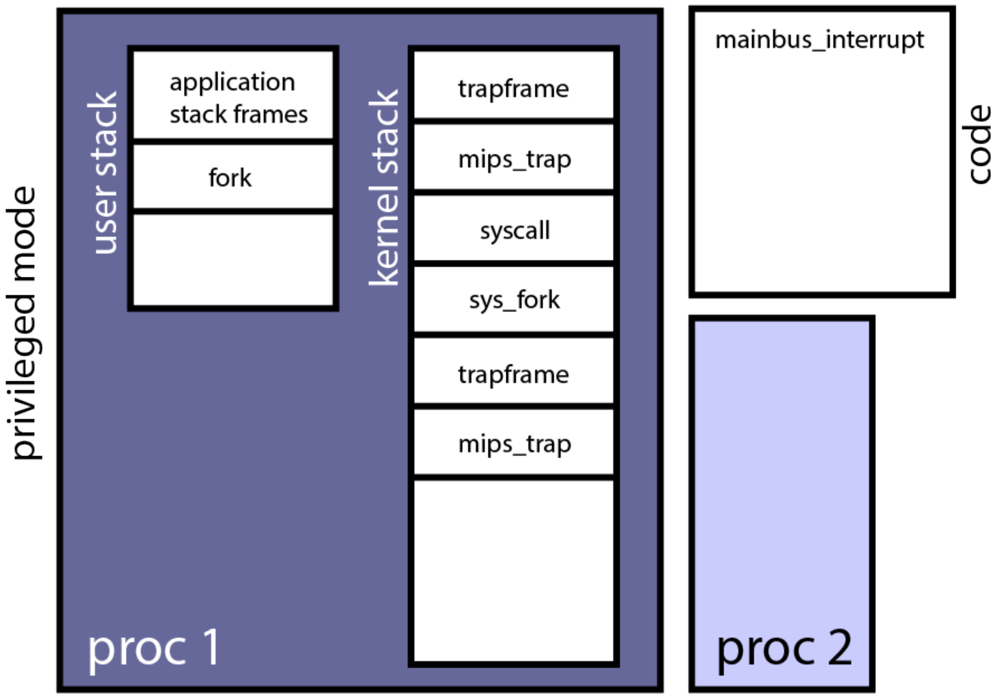
CPU executes `common_exception`. Interrupts are turned off. Save trapframe.

Example: System Calls (11/27)



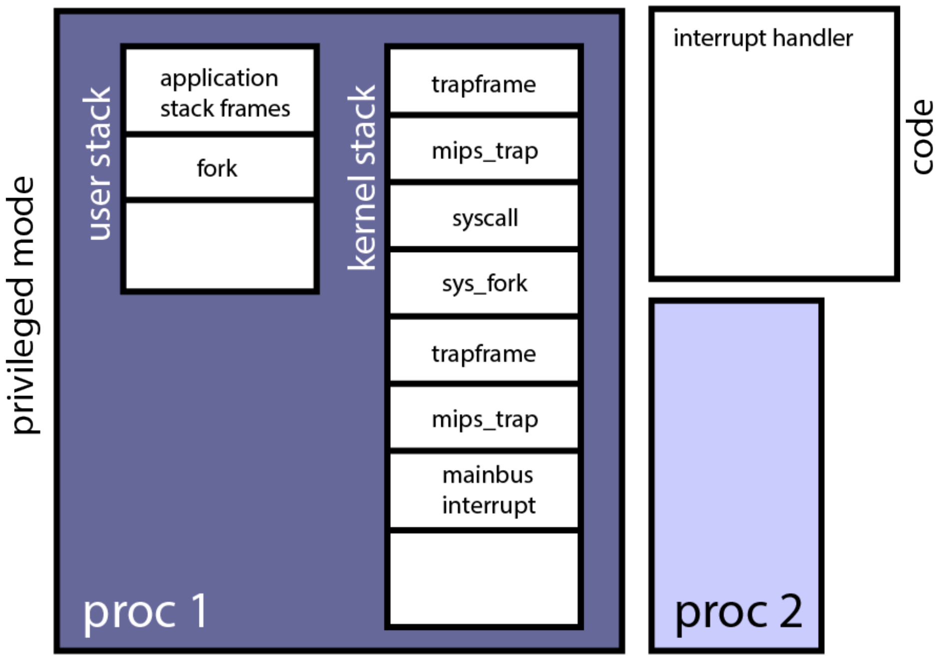
`mips_trap` determines which exception has been raised. In this case, a timer interrupt.

Example: System Calls (12/27)



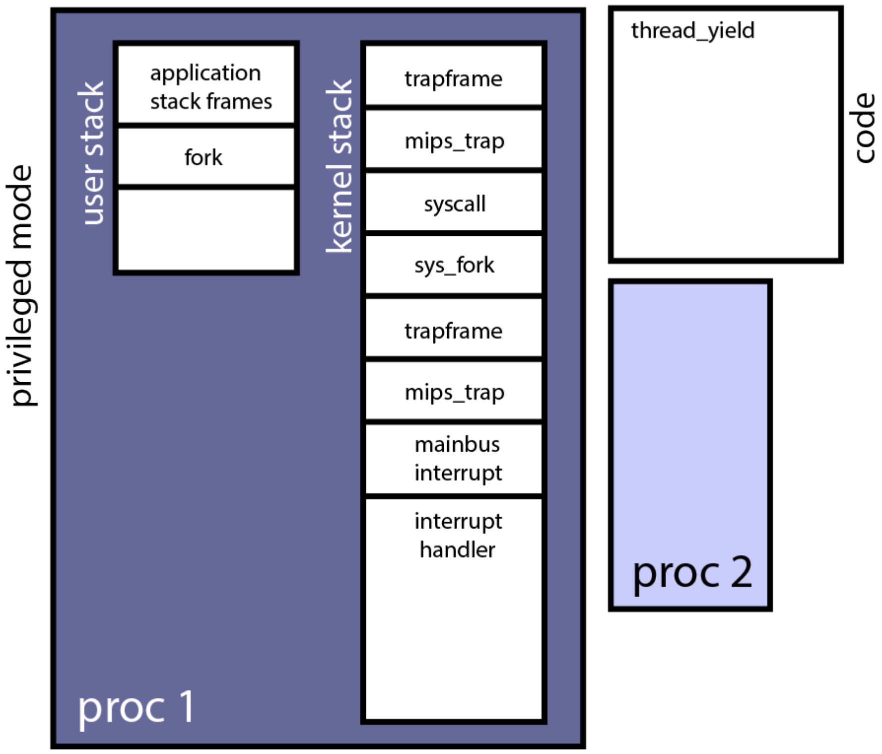
mainbus_interrupt determines which device threw the interrupt, then calls the appropriate handler.

Example: System Calls (13/27)



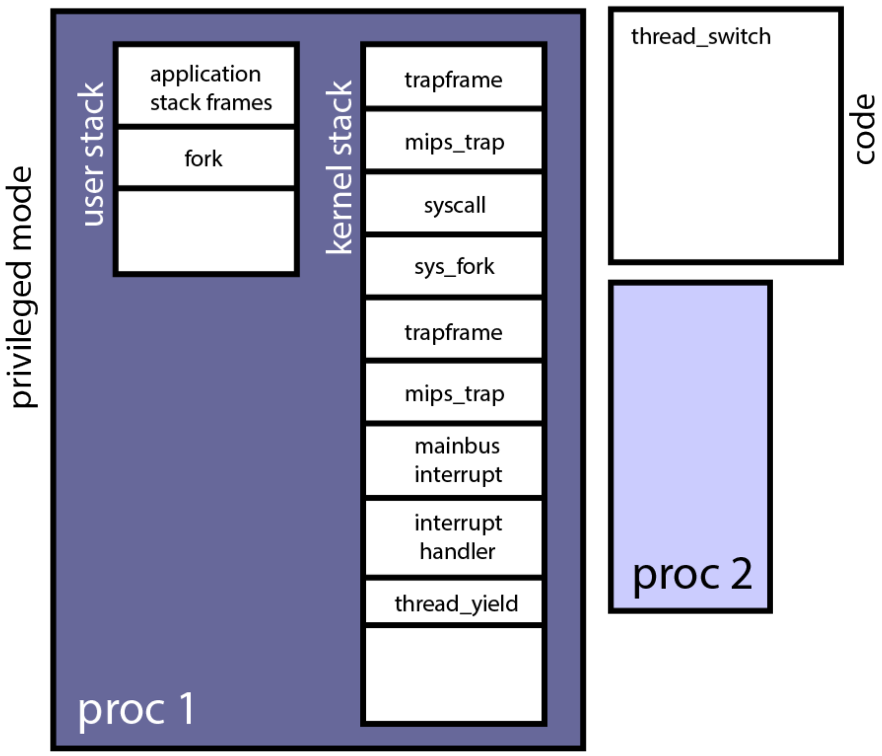
The device interrupt handler runs. Thread quantum has expired.

Example: System Calls (14/27)



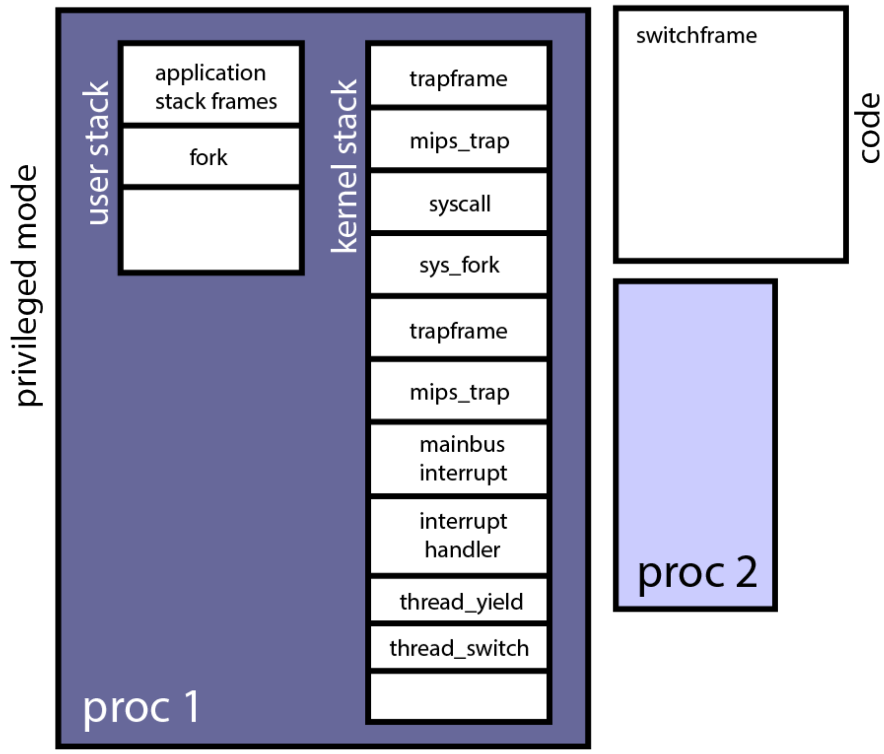
Quantum expired. `thread_yield` is called to perform context switch.

Example: System Calls (15/27)



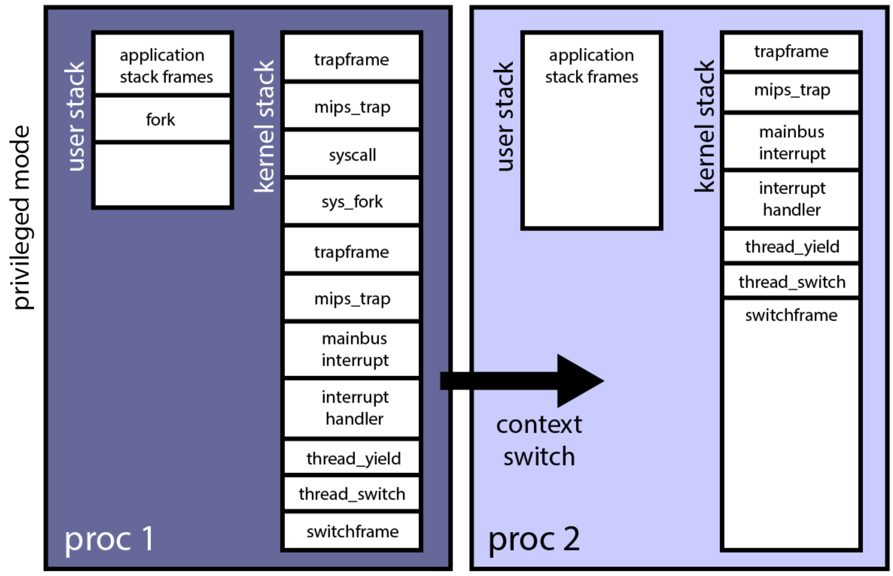
`thread_yield` calls `thread_switch`.

Example: System Calls (16/27)



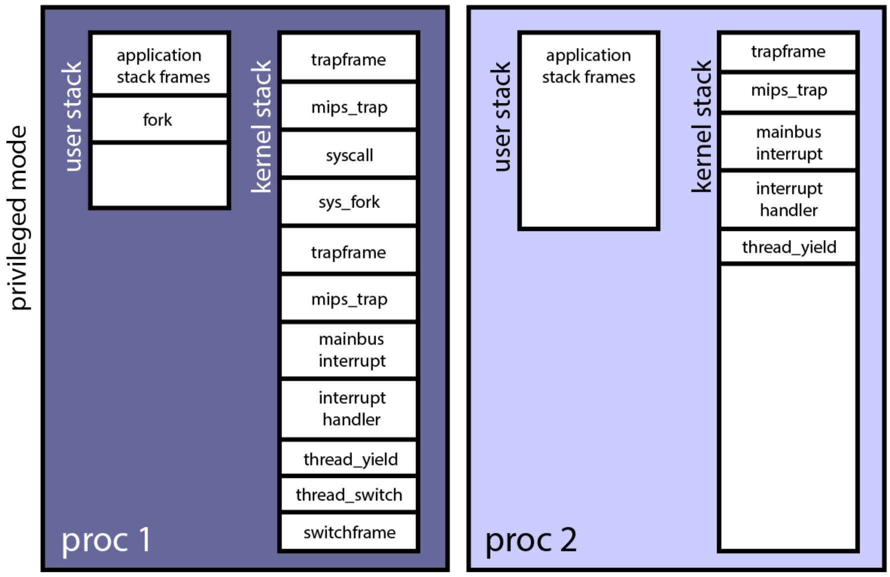
`thread_switch` calls `switchframe_switch`.

Example: System Calls (17/27)



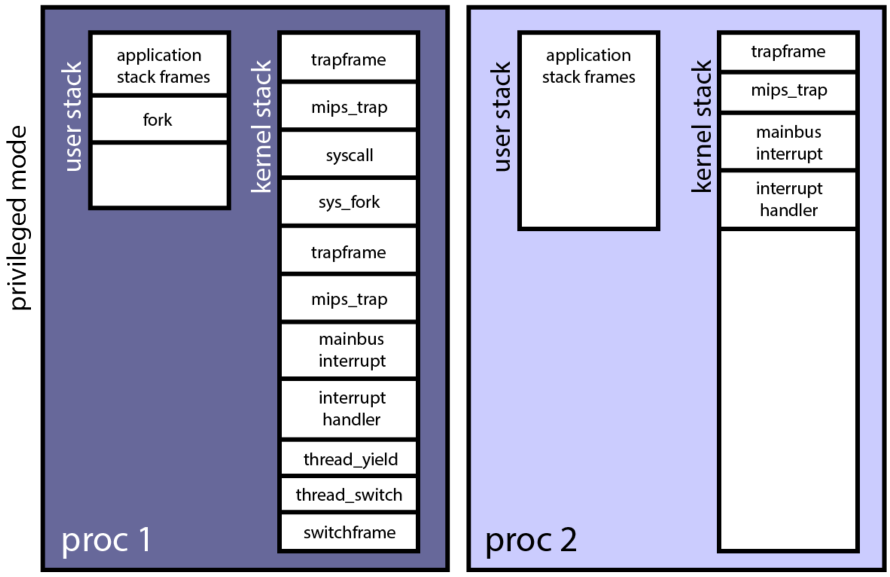
State of current thread saved, context switch occurs.

Example: System Calls (18/27)



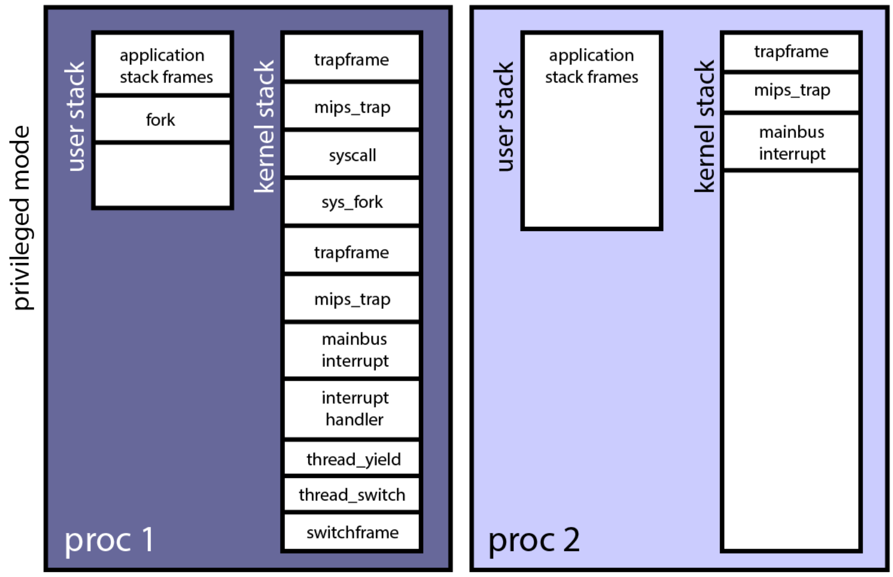
State of new thread restored, return to `thread_yield`.

Example: System Calls (19/27)



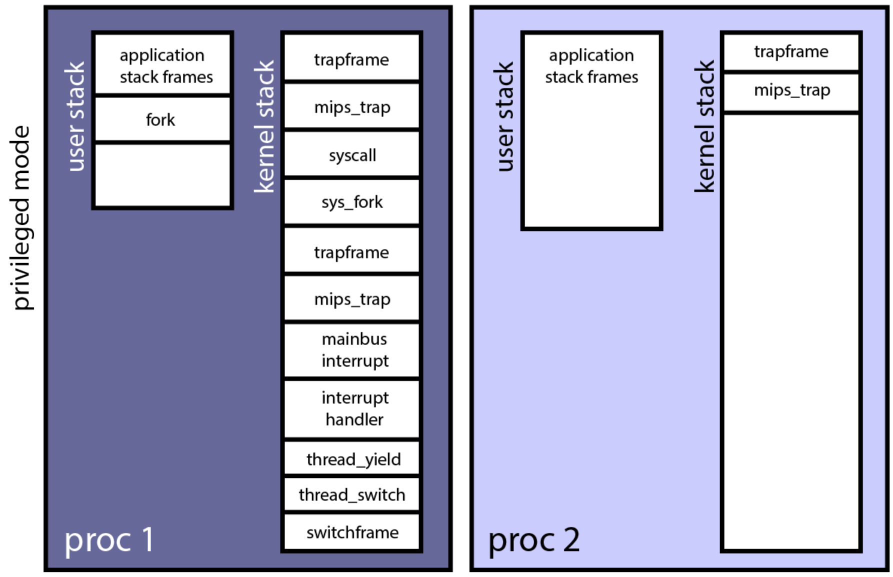
`thread_yield` returns to interrupt handler.

Example: System Calls (20/27)



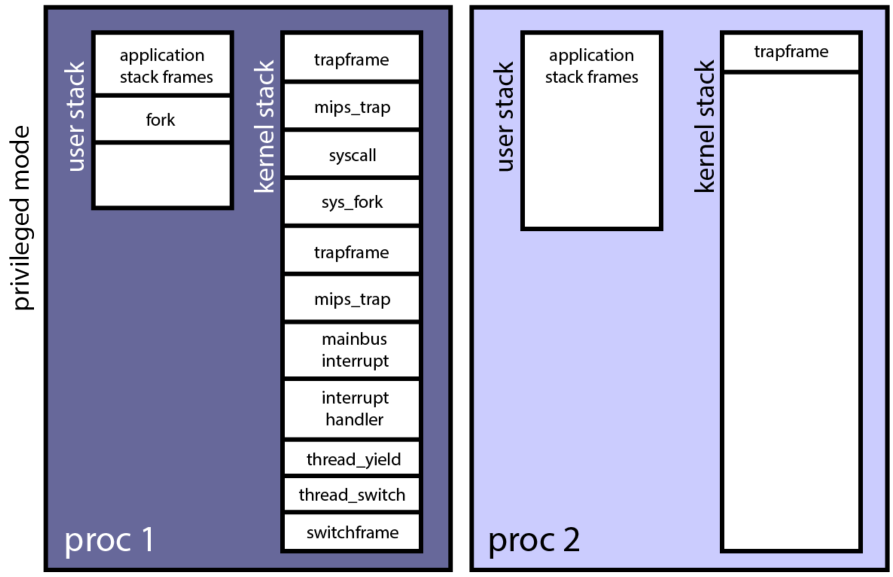
The interrupt handler returns to `mainbus_interrupt`.

Example: System Calls (21/27)



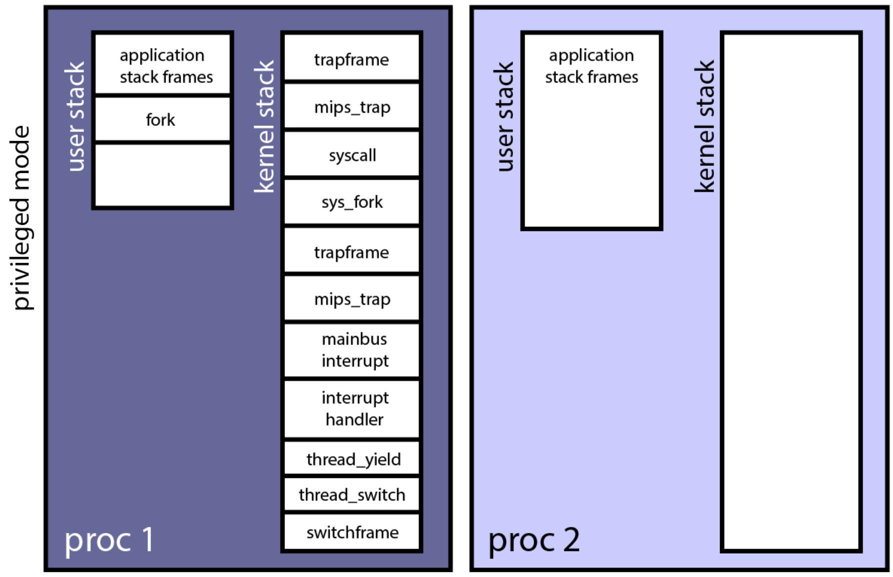
`mainbus_interrupt` returns to `mips_trap`.

Example: System Calls (22/27)



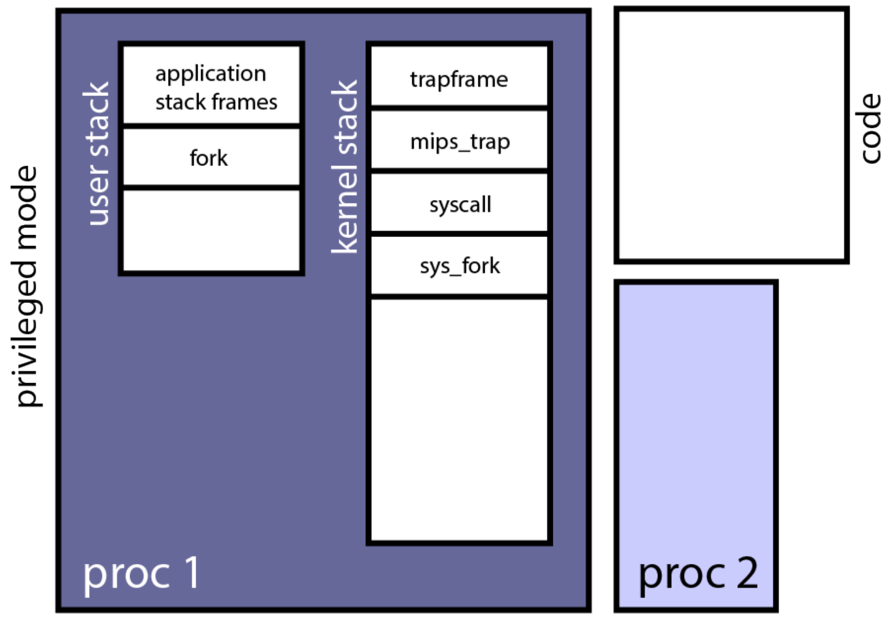
`mips_trap` returns to `common_exception`.

Example: System Calls (23/27)



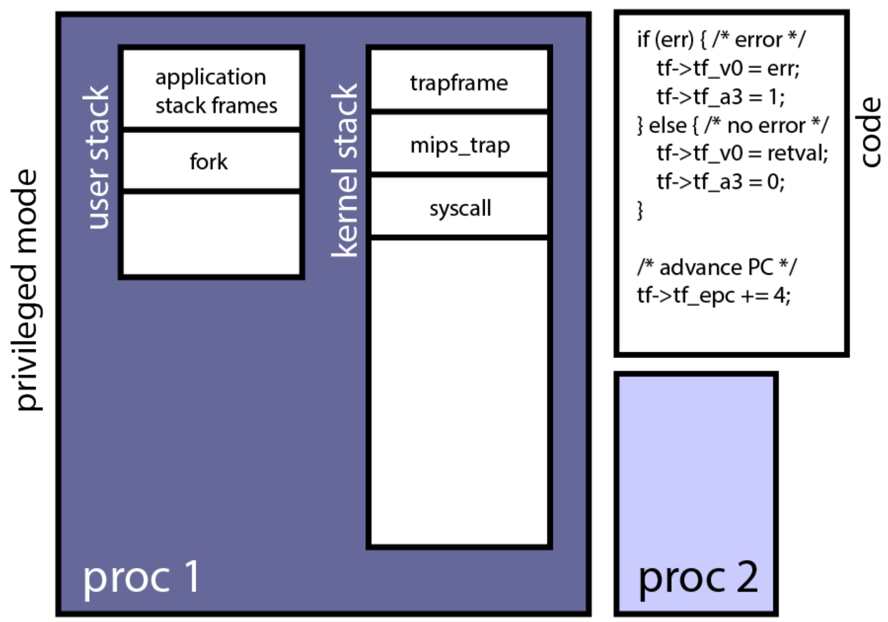
Thread context is restored from trapframe. Switch from kernel to user stacks. Switch to unprivileged mode. User code continues execution.

Example: System Calls (24/27)



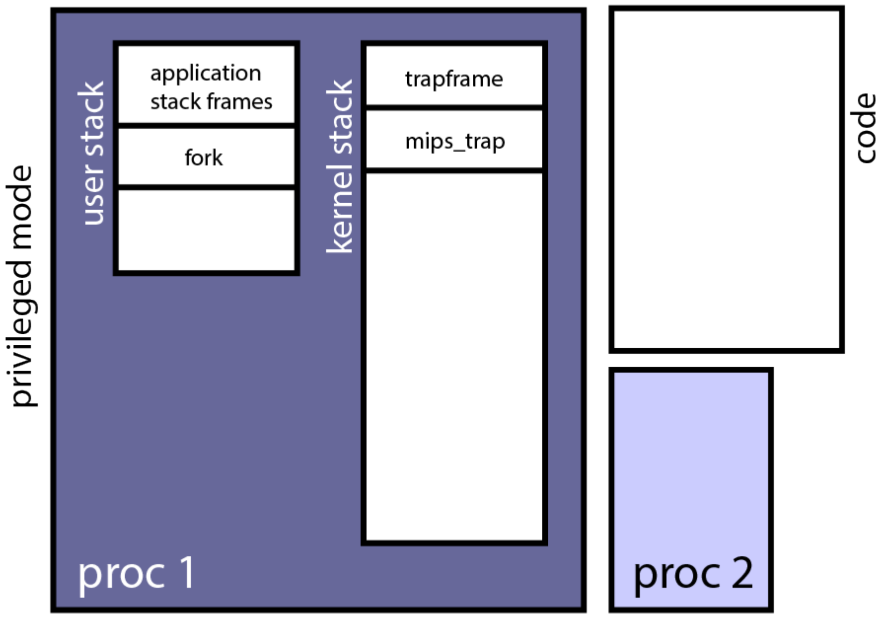
Suppose the timer interrupt did **NOT** occur.

Example: System Calls (25/27)



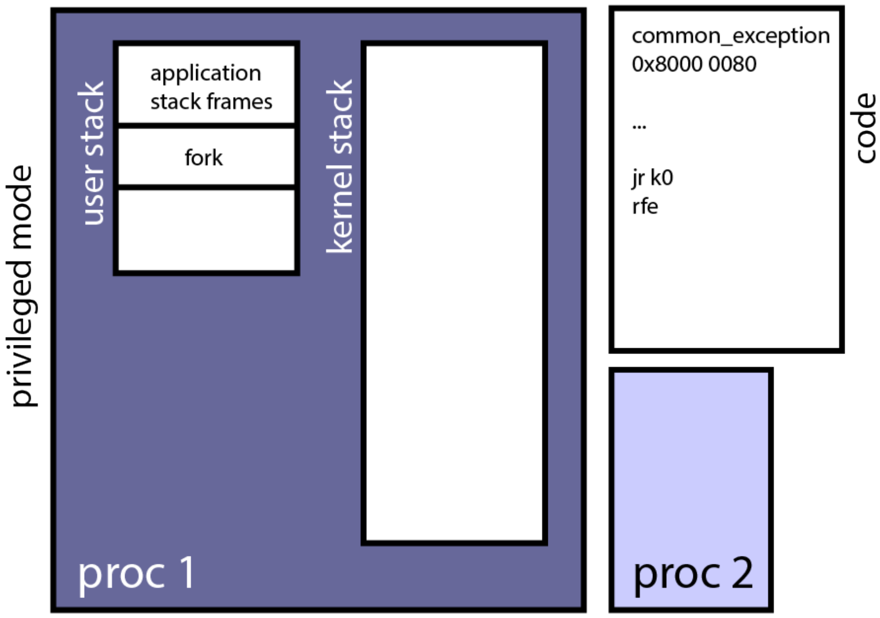
`sys_fork` returns to `syscall`. `syscall` sets up the return value/error code and result. It also increments the PC.

Example: System Calls (26/27)



syscall returns to mips_trap.

Example: System Calls (27/27)



mips_trap returns to common_exception. The trapframe data is restored. Switch from kernel to user stack. Switch to unprivileged mode (rfe). User code continues execution.

Food for thought

- System calls allow user-level processes to interact with the kernel to perform privileged operations
- Do we need to deliver events from kernel to user, asynchronously?
- How do you implement user-level exception handling?
- Upcalls:
  - Unix: signals
  - Windows: asynchronous events