

## What is a Process?

**Answer 1:** a process is an abstraction of a program in execution

**Answer 2:** a process consists of

- an *address space*, which represents the memory that holds the program's code and data
- a *thread* of execution (possibly several threads)
- other resources associated with the running program. For example:
  - open files
  - sockets
  - attributes, such as a name (process identifier)
  - ...

---

---

A process with one thread is a *sequential* process. A process with more than one thread is a *concurrent* process.

---

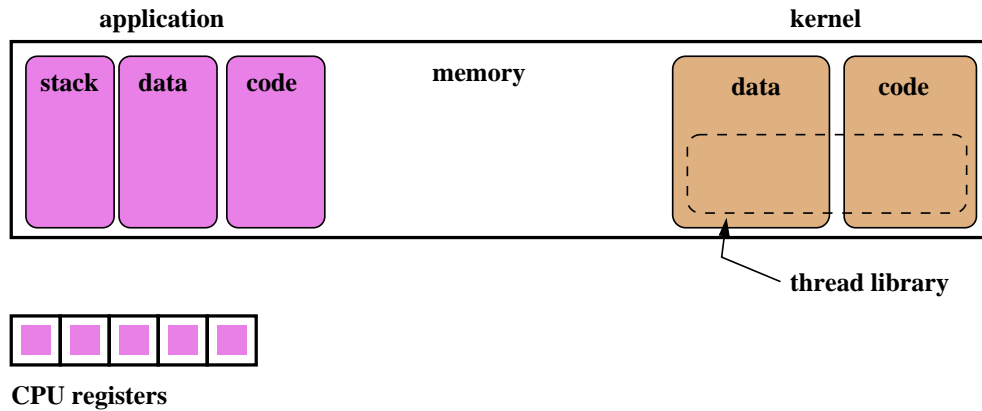
---

## Multiprogramming

- multiprogramming means having multiple processes existing at the same time
- most modern, general purpose operating systems support multiprogramming
- all processes share the available hardware resources, with the sharing coordinated by the operating system:
  - Each process uses some of the available memory to hold its address space. The OS decides which memory and how much memory each process gets
  - OS can coordinate shared access to devices (keyboards, disks), since processes use these devices indirectly, by making system calls.
  - Processes *timeshare* the processor(s). Again, timesharing is controlled by the operating system.
- OS ensures that processes are isolated from one another. Interprocess communication should be possible, but only at the explicit request of the processes involved.

## The OS Kernel

- The kernel is a program. It has code and data like any other program.
- Usually kernel code runs in a privileged execution mode, while other programs do not



## Kernel Privilege, Kernel Protection

- What does it mean to run in privileged mode?
- Kernel uses privilege to
  - control hardware
  - protect and isolate itself from processes
- privileges vary from platform to platform, but may include:
  - ability to execute special instructions (like `halt`)
  - ability to manipulate processor state (like execution mode)
  - ability to access memory addresses that can't be accessed otherwise
- kernel ensures that it is *isolated* from processes. No process can execute or change kernel code, or read or write kernel data, except through controlled mechanisms like system calls.

## System Calls

- System calls are an interface between processes and the kernel.
- A process uses system calls to request operating system services.
- Some examples:

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

## How System Calls Work

- The hardware provides a mechanism that a running program can use to cause a system call. Often, it is a special instruction, e.g., the MIPS `syscall` instruction.
- What happens on a system call:
  - the processor is switched to system (privileged) execution mode
  - key parts of the current thread context, such as the program counter, are saved
  - the program counter is set to a fixed (specified by the hardware) memory address, which is within the kernel's address space

## System Call Execution and Return

- Once a system call occurs, the calling thread will be executing a system call handler, which is part of the kernel, in privileged mode.
- The kernel's handler determines which service the calling process wanted, and performs that service.
- When the kernel is finished, it returns from the system call. This means:
  - restore the key parts of the thread context that were saved when the system call was made
  - switch the processor back to unprivileged (user) execution mode
- Now the thread is executing the calling process' program again, picking up where it left off when it made the system call.

---

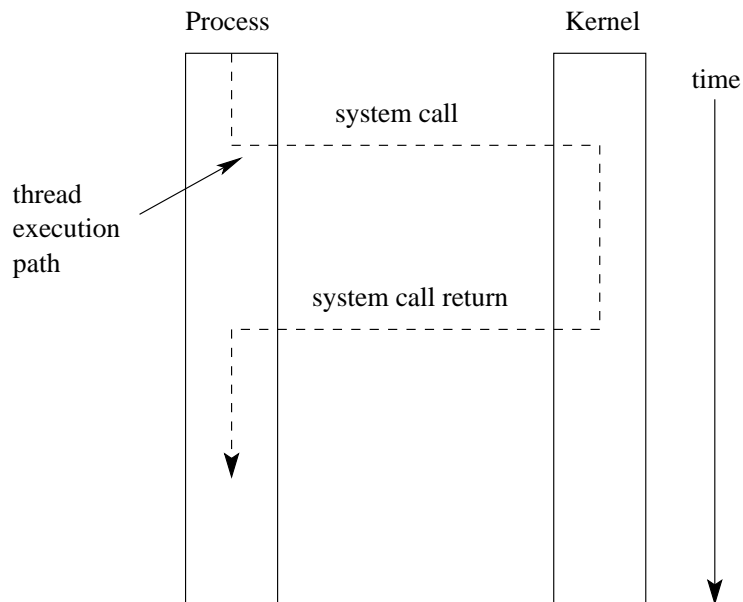
---

A system call causes a thread to stop executing application code and to start executing kernel code in privileged mode. The system call return switches the thread back to executing application code in unprivileged mode.

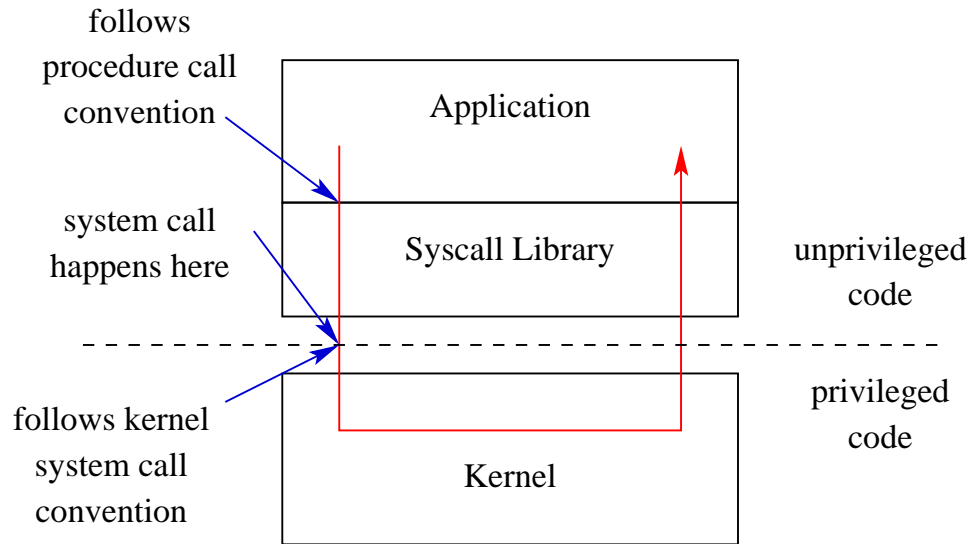
---

---

## System Call Diagram



### System Call Software Stack



### OS/161 `close` System Call Description

**Library:** standard C library (libc)

**Synopsis:**

```
#include <unistd.h>
int
close(int fd);
```

**Description:** The file handle `fd` is closed. ...

**Return Values:** On success, `close` returns 0. On error, -1 is returned and `errno` is set according to the error encountered.

**Errors:**

**EBADF:** `fd` is not a valid file handle

**EIO:** A hard I/O error occurred

### An Example System Call: A Tiny OS/161 Application that Uses `close`

```

/* Program: user/uw-testbin/syscall.c */
#include <unistd.h>
#include <errno.h>

int
main()
{
    int x;
    x = close(999);
    if (x < 0) {
        return errno;
    }
    return x;
}

```

### Disassembly listing of user/uw-testbin/syscall

```

00400050 <main>:
400050: 27bdf0e8  addiu sp,sp,-24
400054: afbf0010  sw  ra,16(sp)
400058: 0c100077  jal 4001dc <close>
40005c: 240403e7  li  a0,999
400060: 04410003  bgez v0,400070 <main+0x20>
400064: 00000000  nop
400068: 3c021000  lui  v0,0x1000
40006c: 8c420000  lw  v0,0(v0)
400070: 8fbf0010  lw  ra,16(sp)
400074: 00000000  nop
400078: 03e00008  jr  ra
40007c: 27bd0018  addiu sp,sp,24

```

---

MIPS procedure call convention: arguments in `a0,a1,...`, return value in `v0`.

---



---

The above can be obtained using `cs350-objdump -d`.

---

## OS/161 MIPS System Call Conventions

- When the `syscall` instruction occurs:
  - An integer system call code should be located in register R2 (v0)
  - Any system call arguments should be located in registers R4 (a0), R5 (a1), R6 (a2), and R7 (a3), much like procedure call arguments.
- When the system call returns
  - register R7 (a3) will contain a 0 if the system call succeeded, or a 1 if the system call failed
  - register R2 (v0) will contain the system call return value if the system call succeeded, or an error number (`errno`) if the system call failed.

## OS/161 System Call Code Definitions

```
/* Contains a number for every more-or-less standard */  
/* Unix system call (you will implement some subset). */  
...  
#define SYS_close          49  
#define SYS_read           50  
#define SYS_pread          51  
//#define SYS_readv        52 /* won't be implementing */  
//#define SYS_preadv       53 /* won't be implementing */  
#define SYS_getdirent     54  
#define SYS_write         55  
...
```

---

---

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 Wrapper Functions from the Standard Library

```
...
004001dc <close>:
    4001dc: 08100030  j 4000c0 <__syscall>
    4001e0: 24020031  li  v0,49

004001e4 <read>:
    4001e4: 08100030  j 4000c0 <__syscall>
    4001e8: 24020032  li  v0,50
...
```

---

---

The above is disassembled code from the standard C library (libc), which is linked with user/uw-testbin/syscall.o.

---

---

## The OS/161 System Call and Return Processing

```
004000c0 <__syscall>:
    4000c0: 0000000c  syscall
    4000c4: 10e00005  beqz  a3,4000dc <__syscall+0x1c>
    4000c8: 00000000  nop
    4000cc: 3c011000  lui  at,0x1000
    4000d0: ac220000  sw   v0,0(at)
    4000d4: 2403ffff  li   v1,-1
    4000d8: 2402ffff  li   v0,-1
    4000dc: 03e00008  jr   ra
    4000e0: 00000000  nop
```

---

---

The system call and return processing, from the standard C library. Like the rest of the library, this is unprivileged, user-level code.

---

---



## OS/161 MIPS Exception Handler

```

common_exception:
    mfc0 k0, c0_status /* Get status register */
    andi k0, k0, CST_KUp /* Check the we-were-in-user-mode bit */
    beq k0, $0, 1f /* If clear, from kernel, already have stack */
                    /* 1f is branch forward to label 1: */
    nop             /* delay slot */

    /* Coming from user mode - find kernel stack */
    mfc0 k1, c0_context /* we keep the CPU number here */
    srl k1, k1, CTX_PTBASESHIFT /* shift to get the CPU number */
    sll k1, k1, 2             /* shift back to make array index */
    lui k0, %hi(cpustacks) /* get base address of cpustacks[] */
    addu k0, k0, k1         /* index it */
    move k1, sp             /* Save previous stack pointer */
    b 2f                   /* Skip to common code */
    lw sp, %lo(cpustacks)(k0) /* Load kernel sp (in delay slot) */

```

## OS/161 MIPS Exception Handler

```

1:
    /* Coming from kernel mode - just save previous stuff */
    move k1, sp /* Save previous stack in k1 (delay slot) */
2:

    /* At this point:
    * Interrupts are off. (The processor did this for us.)
    * k0 contains the value for curthread, to go into s7.
    * k1 contains the old stack pointer.
    * sp points into the kernel stack.
    * All other registers are untouched.
    */

```

---



---

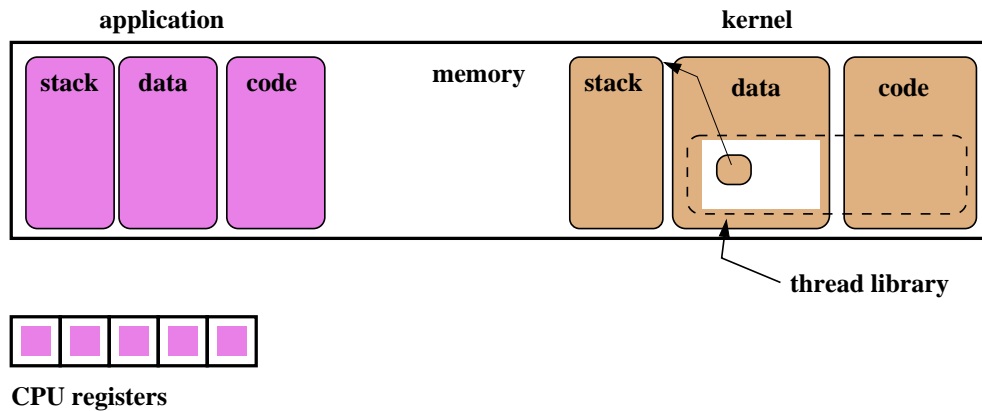
When the syscall instruction occurs, the MIPS transfers control to address 0x80000080. This kernel exception handler lives there. See kern/arch/mips/locore/exception-mips1.S

---



---

## OS/161 User and Kernel Thread Stacks




---

Each OS/161 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.

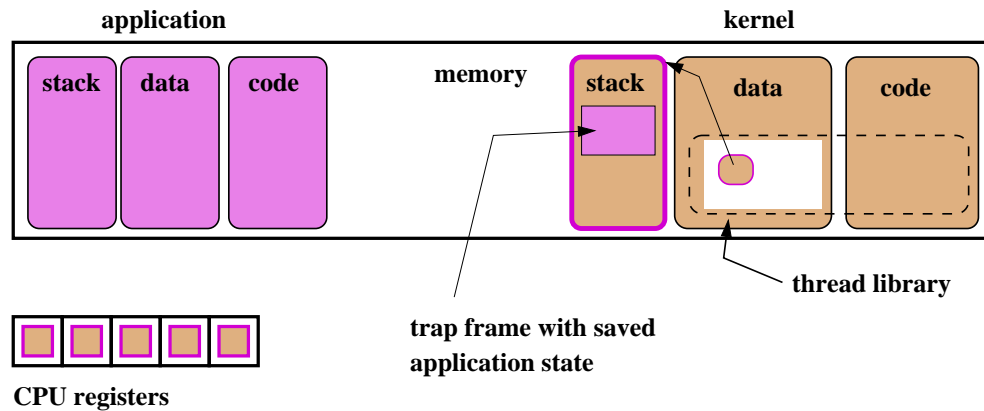
---

## OS/161 MIPS Exception Handler (cont'd)

The `common_exception` code then does the following:

1. allocates a *trap frame* on the thread's kernel stack and saves the user-level application's complete processor state (all registers except `k0` and `k1`) into the trap frame.
2. calls the `mips_trap` function to continue processing the exception.
3. when `mips_trap` returns, restores the application processor state from the trap frame to the registers
4. issues MIPS `jr` and `rfe` (restore from exception) instructions to return control to the application code. The `jr` instruction takes control back to the location specified by the application program counter when the `syscall` occurred (i.e., exception PC) and the `rfe` (which happens in the delay slot of the `jr`) restores the processor to unprivileged mode

## OS/161 Trap Frame




---

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.

---

## `mips_trap`: Handling System Calls, Exceptions, and Interrupts

- On the MIPS, the same exception handler is invoked to handle system calls, exceptions and interrupts
- The hardware sets a code to indicate the reason (system call, exception, or interrupt) that the exception handler has been invoked
- OS/161 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 (`syscall`) in the case of a system call.
- `mips_trap` can be found in `kern/arch/mips/locore/trap.c`.

---

Interrupts and exceptions will be presented shortly

---

### OS/161 System Call Handler

```

syscall(struct trapframe *tf)
{  callno = tf->tf_v0; retval = 0;
  switch (callno) {
    case SYS_reboot:
      err = sys_reboot(tf->tf_a0);
      break;
    case SYS___time:
      err = sys___time((userptr_t)tf->tf_a0,
        (userptr_t)tf->tf_a1);
      break;

    /* Add stuff here */

  default:
    kprintf("Unknown syscall %d\n", callno);
    err = ENOSYS;
    break;
  }
}

```

---

syscall checks system call code and invokes a handler for the indicated system call. See kern/arch/mips/syscall/syscall.c

---

### OS/161 MIPS System Call Return Handling

```

if (err) {
  tf->tf_v0 = err;
  tf->tf_a3 = 1;      /* signal an error */
} else {
  /* Success. */
  tf->tf_v0 = retval;
  tf->tf_a3 = 0;      /* signal no error */
}

/* Advance the PC, to avoid the syscall again. */
tf->tf_epc += 4;

/* Make sure the syscall code didn't forget to lower spl */
KASSERT(curthread->t_curspl == 0);
/* ...or leak any spinlocks */
KASSERT(curthread->t_iphigh_count == 0);
}

```

---

syscall must ensure that the kernel adheres to the system call return convention.

---

## Exceptions

- Exceptions are another way that control is transferred from a process to the kernel.
- Exceptions are conditions that occur during the execution of an instruction by a process. For example, arithmetic overflows, illegal instructions, or page faults (to be discussed later).
- Exceptions are detected by the hardware.
- When an exception is detected, the hardware transfers control to a specific address.
- Normally, a kernel exception handler is located at that address.

---



---

Exception handling is similar to, but not identical to, system call handling.  
(What is different?)

---



---

## 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 OS/161, `mips_trap` uses these codes to decide whether it has been invoked because of an interrupt, a system call, or an exception.

---



---

### Interrupts (Revisited)

- Interrupts are a third mechanism by which control may be transferred to the kernel
- Interrupts are similar to exceptions. However, they are caused by hardware devices, not by the execution of a program. For example:
  - a network interface may generate an interrupt when a network packet arrives
  - a disk controller may generate an interrupt to indicate that it has finished writing data to the disk
  - a timer may generate an interrupt to indicate that time has passed
- Interrupt handling is similar to exception handling - current execution context is saved, and control is transferred to a kernel interrupt handler at a fixed address.

### Interrupts, Exceptions, and System Calls: Summary

- Interrupts, exceptions and system calls are three mechanisms by which control is transferred from an application program to the kernel
- When these events occur, the hardware switches the CPU into privileged mode and transfers control to a predefined location, at which a kernel *handler* should be located.
- The kernel handler creates a *trap frame* and uses it to save the application thread context so that the handler code can be executed on the CPU.
- Just before the kernel handler finishes executing, it restores the application thread context from the trap frame, before returning control to the application.

---

---

In OS/161, trap frames are placed on the *kernel stack* of the thread performed the system call, or of the thread that was running when the interrupt or exception occurred

---

---

### System Calls for Process Management

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

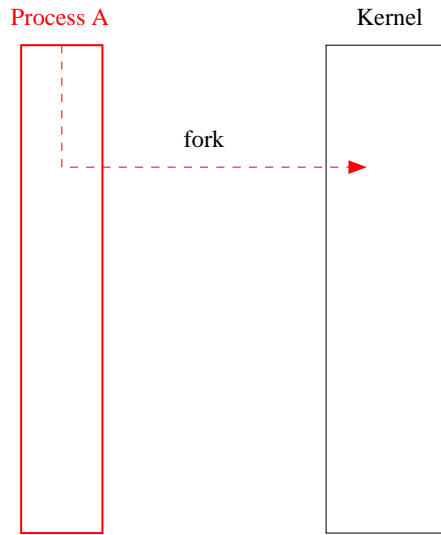
### The fork, \_exit, getpid and waitpid system calls

```

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))
    }
}

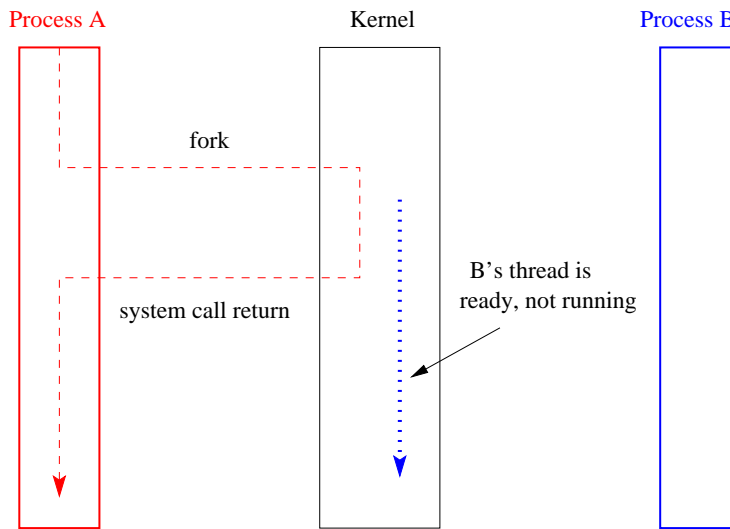
```

### Process Creation Example (Part 1)



Parent process (Process A) requests creation of a new process.

### Process Creation Example (Part 2)



Kernel creates new process (Process B)



### The execv system call

```
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

```
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);
    }
}
```

## Implementation of Processes

- The kernel maintains information about all of the processes in the system in a data structure often called the process table.
- Per-process information may include:
  - process identifier and owner
  - the address space for the process
  - threads belonging to the process
  - lists of resources allocated to the process, such as open files
  - accounting information

## OS/161 Process

```
/* From kern/include/proc.h */
struct proc {
    char *p_name; /* Name of this process */
    struct spinlock p_lock; /* Lock for this structure */
    struct threadarray p_threads; /* Threads in process */

    struct addrspace *p_addrspace; /* virtual address space */
    struct vnode *p_cwd; /* current working directory */

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

## OS/161 Process

```
/* From kern/include/proc.h */
/* Create a fresh process for use by runprogram() */
struct proc *proc_create_runprogram(const char *name);

/* Destroy a process */
void proc_destroy(struct proc *proc);

/* Attach a thread to a process */
/* Must not already have a process */
int proc_addthread(struct proc *proc, struct thread *t);

/* Detach a thread from its process */
void proc_remthread(struct thread *t);
...
```

## Implementing Timesharing

- whenever a system call, exception, or interrupt occurs, control is transferred from the running program to the kernel
- at these points, the kernel has the ability to cause a context switch from the running process' thread to another process' thread
- notice that these context switches always occur while a process' thread is executing kernel code

---

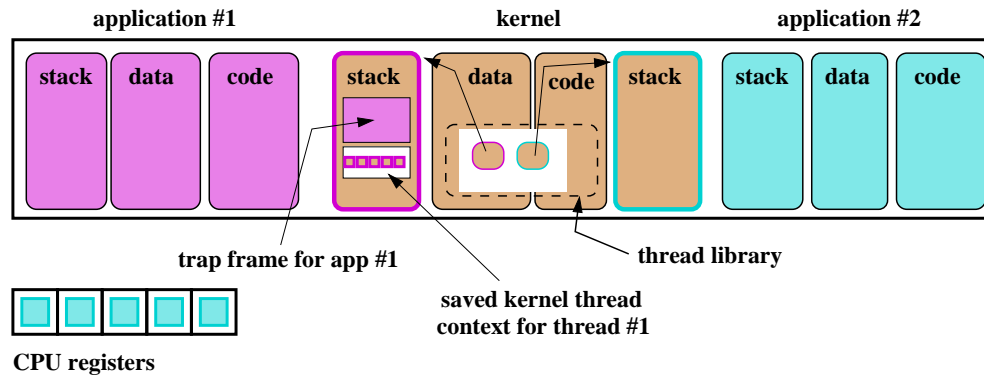
---

By switching from one process's thread to another process's thread, the kernel timeshares the processor among multiple processes.

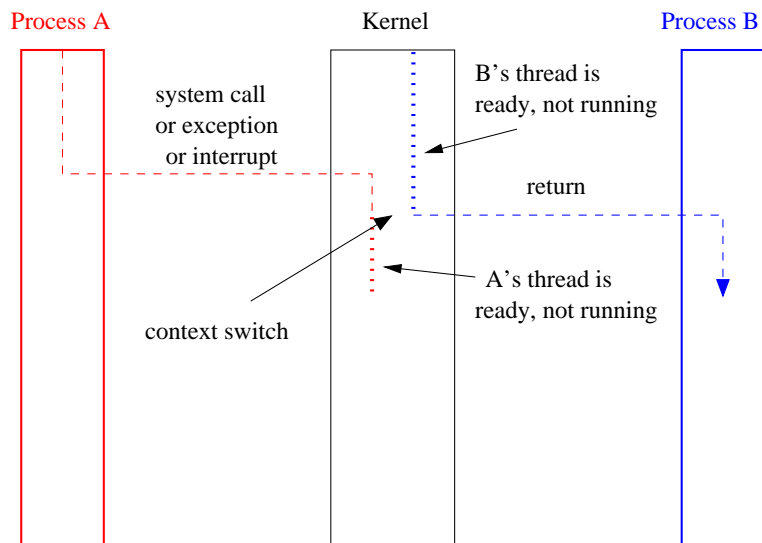
---

---

## Two Processes in OS/161

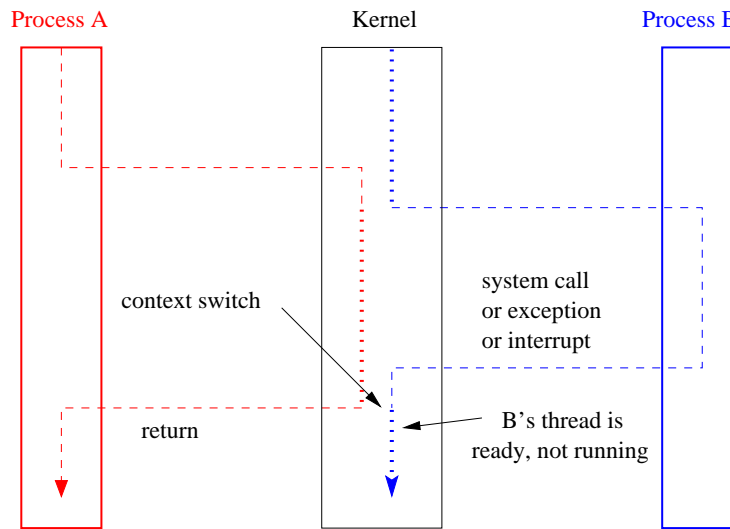


## Timesharing Example (Part 1)



Kernel switches execution context to Process B.

## Timesharing Example (Part 2)



Kernel switches execution context back to process A.

## Implementing Preemption

- the kernel uses interrupts from the system timer to measure the passage of time and to determine whether the running process's quantum has expired.
- a timer interrupt (like any other interrupt) transfers control from the running program to the kernel.
- this gives the kernel the opportunity to preempt the running thread and dispatch a new one.

### Preemptive Multiprogramming Example

