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.

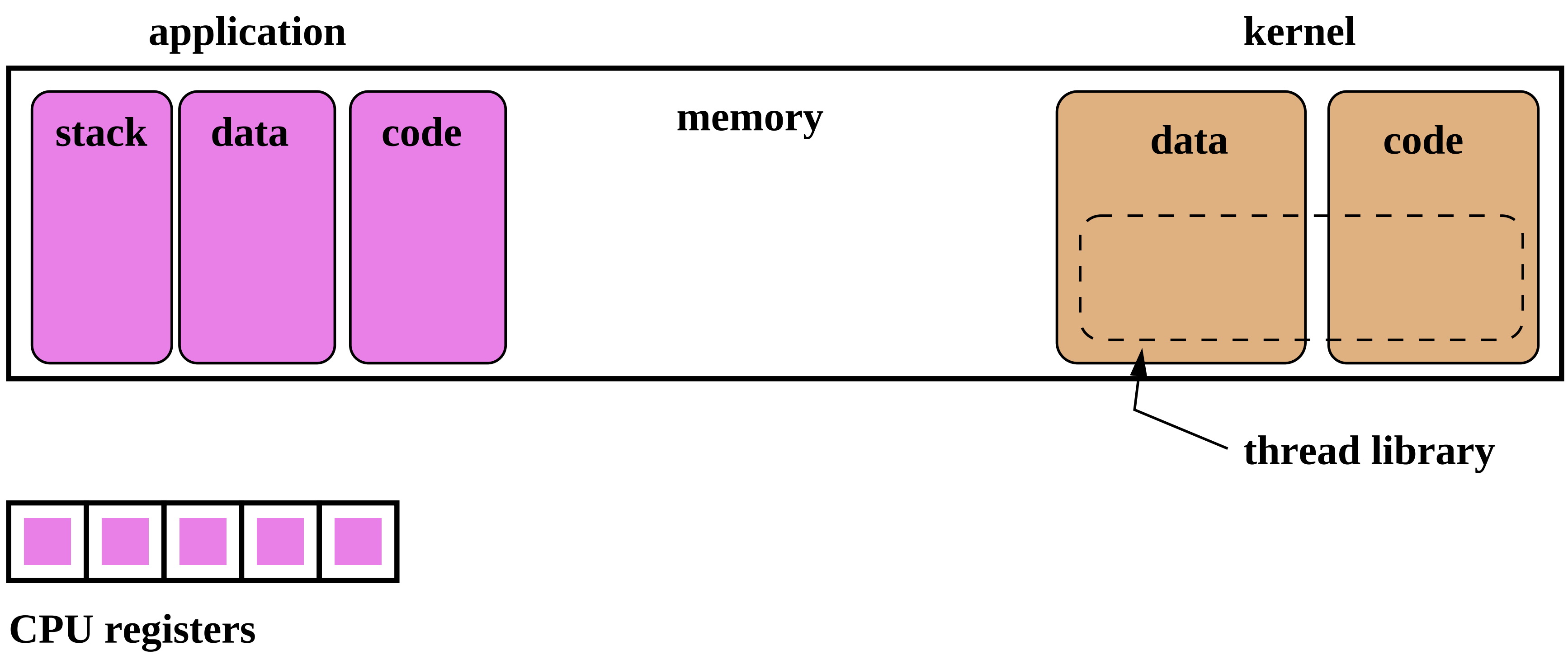
Multiprogramming

- 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
- For now, think of the kernel as a program that resides in its own address space, separate from the address spaces of processes that are running on the system. Later, we will elaborate on the relationship between the kernel's address space and process address spaces.

An Application and the Kernel



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.
- From point of view of the process, these services are used to manipulate the abstractions that are part of its execution environment. For example, a process might use a system call to
  - open a file
  - send a message over a pipe
  - create another process
  - increase the size of its address space

How System Calls Work

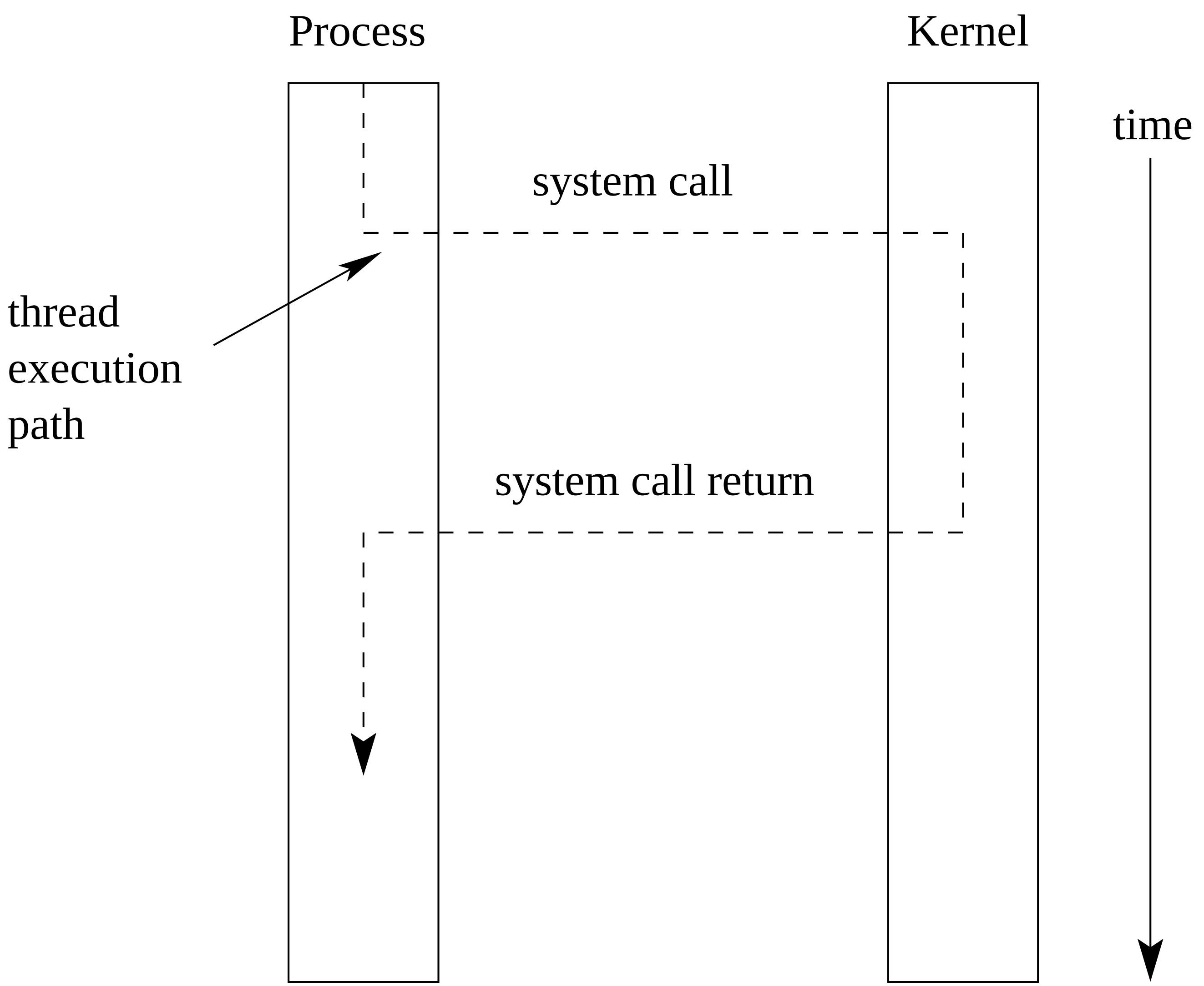
- The hardware provides a mechanism that a running program can use to cause a system call. Often, it is a special instruction, e.g., the MIPS `syscall` instruction.
- What happens on a system call:
  - the processor is switched to system (privileged) execution mode
  - key parts of the current thread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



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

A Tiny OS/161 Application that Uses `close`: SyscallExample

```
/* Program: SyscallExample */
#include <unistd.h>
#include <errno.h>

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

SyscallExample, Disassembled

```
00400100 <main>:
  400100: 27bdffe8  addiu sp,sp,-24
  400104: afbf0010  sw ra,16(sp)
  400108: 0c100077  jal 4001dc <close>
  40010c: 240403e7  li a0,999
  400110: 04400005  bltz v0,400128 <main+0x28>
  400114: 00401821  move v1,v0
  400118: 8fbf0010  lw ra,16(sp)
  40011c: 00601021  move v0,v1
  400120: 03e00008  jr ra
  400124: 27bd0018  addiu sp,sp,24
  400128: 3c031000  lui v1,0x1000
  40012c: 8c630000  lw v1,0(v1)
  400130: 08100046  j 400118 <main+0x18>
  400134: 00000000  nop
```

The above can be obtained by disassembling the compiled `SyscallExample` executable file using `cs350-objdump -d`

System Call Wrapper Functions from the Standard Library

```
...
004001d4 <write>:
  4001d4: 08100060  j 400180 <__syscall>
  4001d8: 24020006  li v0,6
004001dc <close>:
  4001dc: 08100060  j 400180 <__syscall>
  4001e0: 24020007  li v0,7
004001e4 <reboot>:
  4001e4: 08100060  j 400180 <__syscall>
  4001e8: 24020008  li v0,8
...
```

The above is disassembled code from the standard C library (libc), which is linked with `SyscallExample`. See `lib/libc/syscalls.S` for more information about how the standard C library is implemented.

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

```
...
#define SYS_read        5
#define SYS_write       6
#define SYS_close       7
#define SYS_reboot      8
#define SYS_sync        9
#define SYS_sbrk        10
...
```

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

The OS/161 System Call and Return Processing

```
00400180 <__syscall>:
  400180: 0000000c  syscall
  400184: 10e00005  beqz a3,40019c <__syscall+0x1c>
  400188: 00000000  nop
  40018c: 3c011000  lui at,0x1000
  400190: ac220000  sw v0,0(at)
  400194: 2403ffff  li v1,-1
  400198: 2402ffff  li v0,-1
  40019c: 03e00008  jr ra
  4001a0: 00000000  nop
```

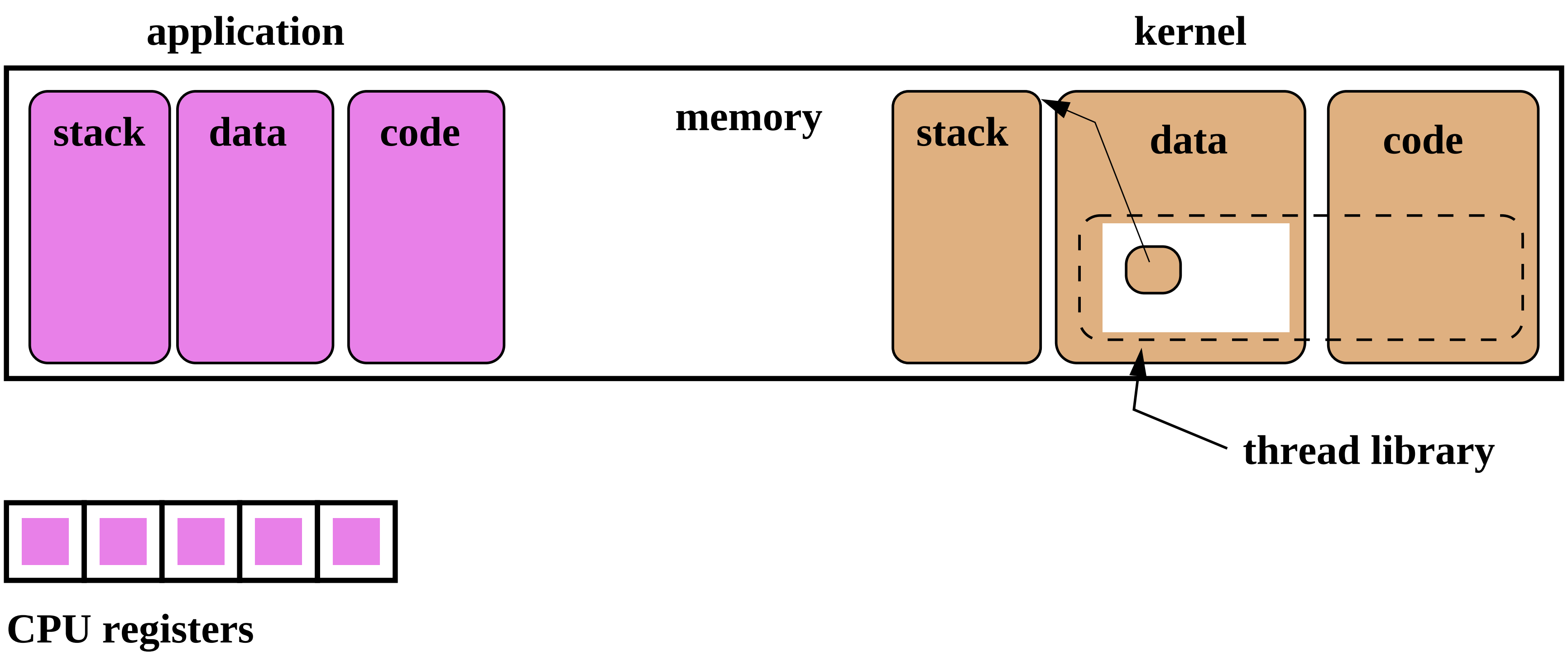
The system call and return processing, from the standard C library. Like the rest of the library, this is unprivileged, user-level code.

OS/161 MIPS Exception Handler

```
exception:
  move k1, sp         /* Save previous stack pointer in k1 */
  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
  nop                  /* delay slot */
  /* Coming from user mode - load kernel stack into sp */
  la k0, curkstack     /* get address of "curkstack" */
  lw sp, 0(k0)         /* get its value */
  nop                  /* delay slot for the load */
1:
  mfc0 k0, c0_cause    /* Now, load the exception cause. */
  j common_exception   /* Skip to common code */
  nop                  /* delay slot */
```

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

OS/161 User and Kernel Thread Stacks



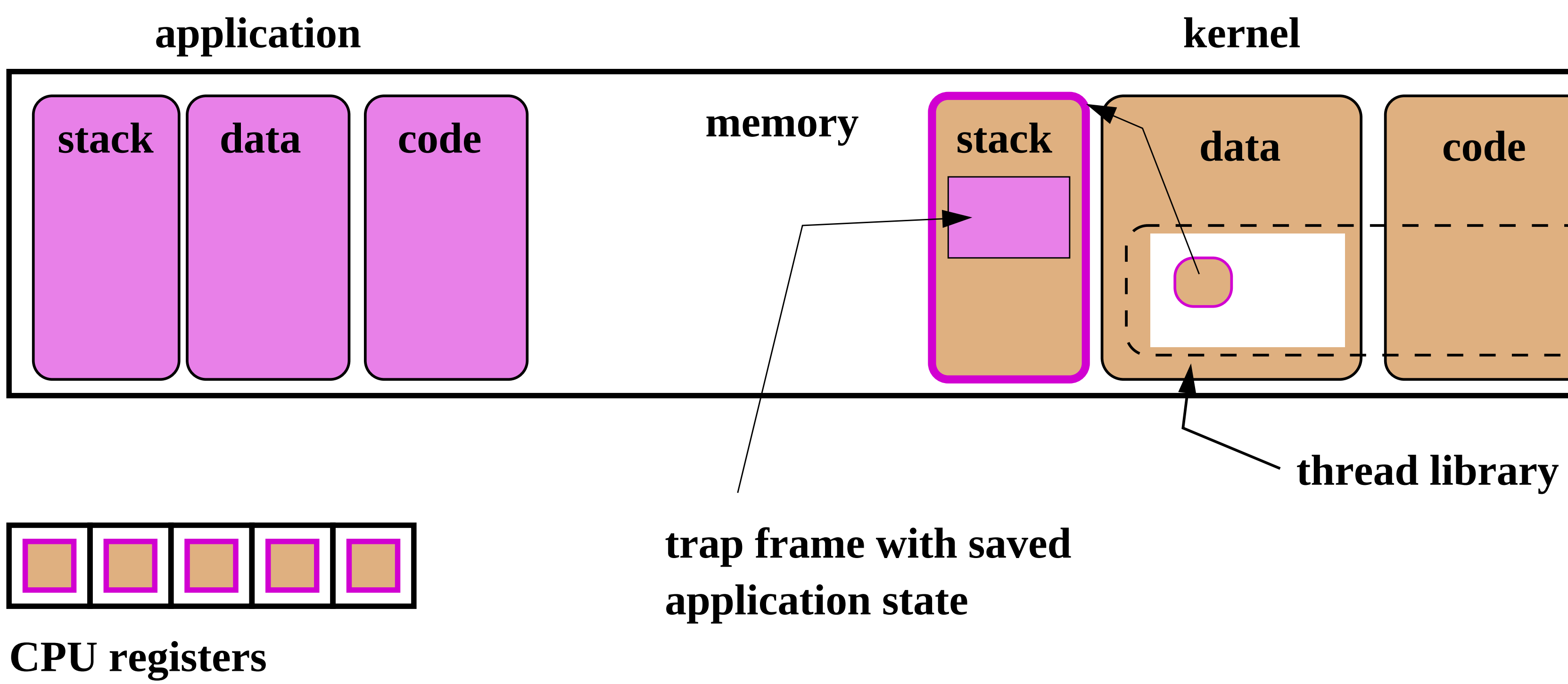
Each OS/161 thread has two stacks, one that is used while the thread is executing unprivileged application code, and another that is used while the thread is executing privileged kernel code.

OS/161 MIPS Exception Handler (cont'd)

The `common_exception` code 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, restore the application processor state from the trap from to the registers
4. issue MIPS `jr` and `rfe` (restore from exception) instructions to return control to the application code. The `jr` instruction takes control back to location specified by the application program counter when the `syscall` occurred, 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 (`mips_syscall`) in the case of a system call.
- `mips_trap` can be found in `kern/arch/mips/mips/trap.c`.

Interrupts and exceptions will be presented shortly

OS/161 MIPS System Call Handler

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

    /* Add stuff here */

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

`mips_syscall` checks the system call code and invokes a handler for the indicated system call. See `kern/arch/mips/mips/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 *
  assert(curspl==0);
}
```

`mips_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

- 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 handler saves the application thread context so that the kernel code can be executed on the CPU, and restores the application thread context just before control is returned to the application

Implementation of Processes

- The kernel maintains information about all of the processes in the system in a data structure often called the process table.
- Information about individual processes is stored in a structure that is sometimes called a *process control block (PCB)*. In practice, however, information about a process may not all be located in a single data structure.
- Per-process information may include:
  - process identifier and owner
  - current process state and other scheduling information
  - lists of resources allocated to the process, such as open files
  - accounting information

In OS/161, some process information (e.g., an address space pointer) is kept in the `thread` structure. This works only because each OS/161 process has a single thread.

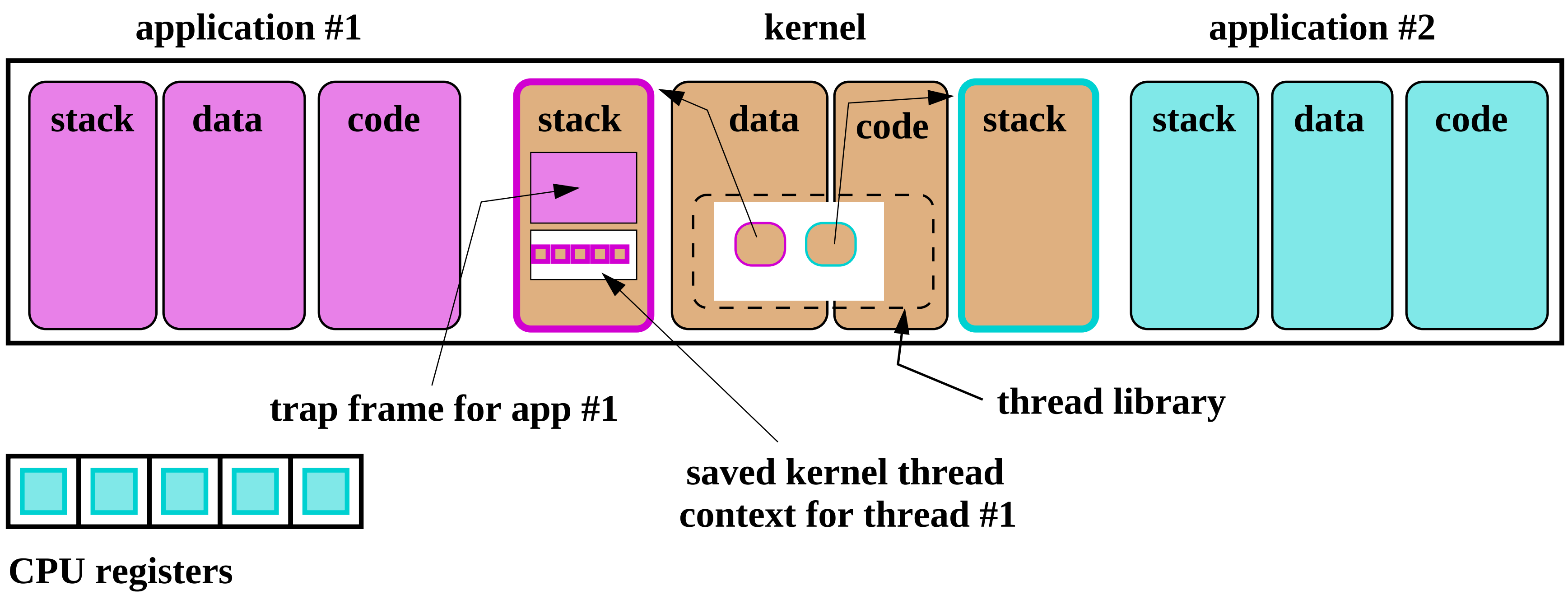
Implementing Timesharing

- 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

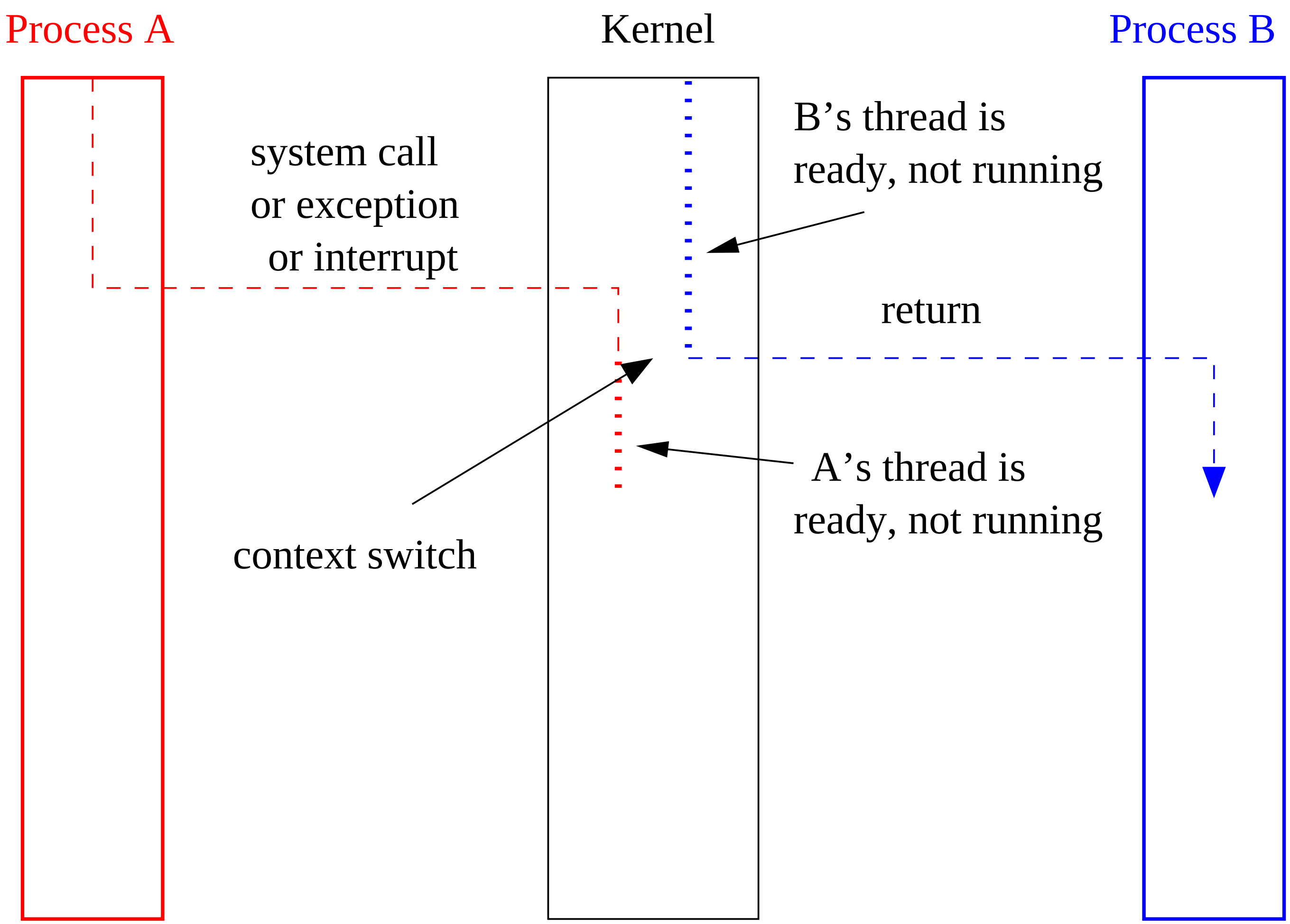
---

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

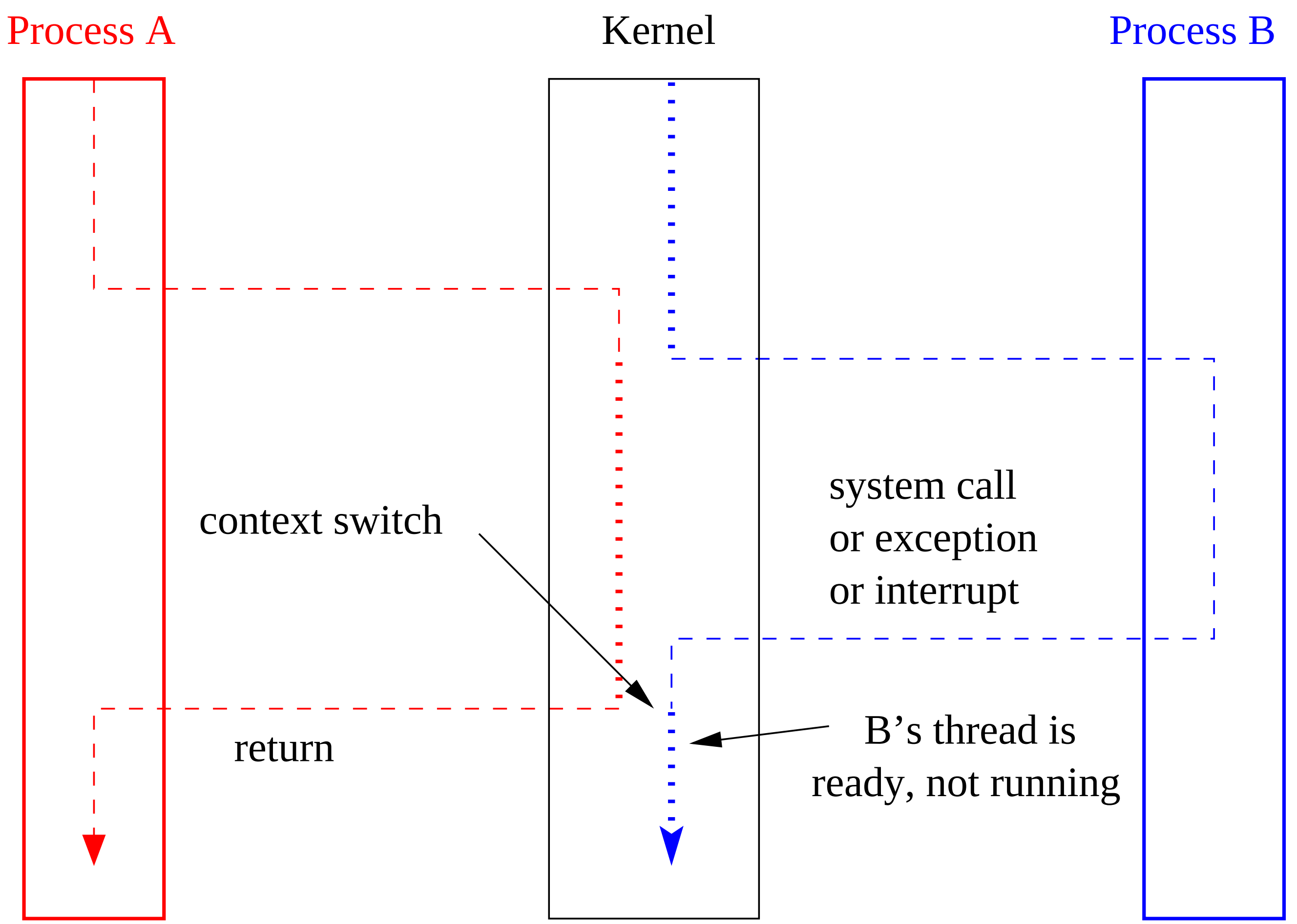


Timesharing Example (Part 1)



Kernel switches execution context to Process B.

Timesharing Example (Part 2)

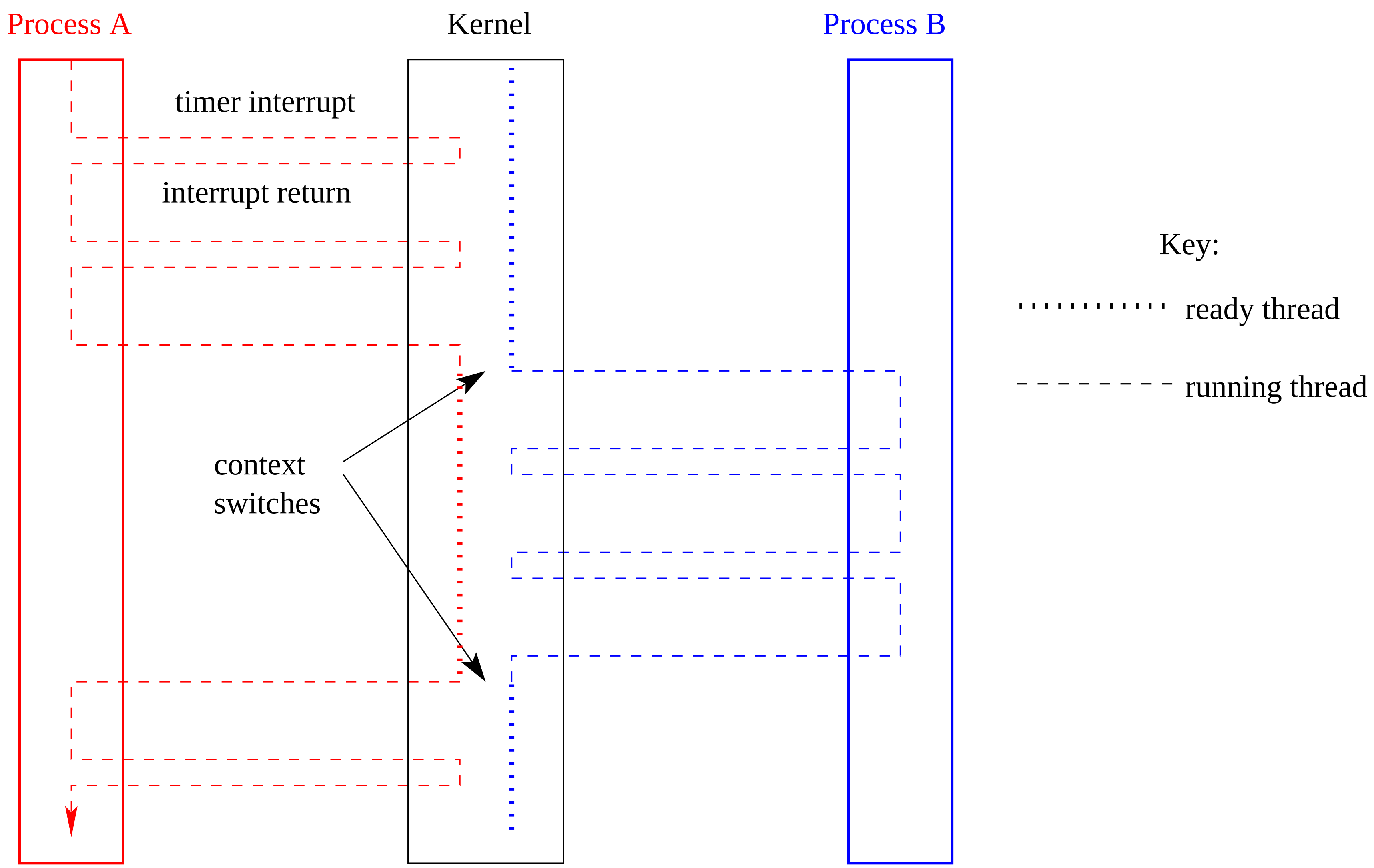


Kernel switches execution context back to process A.

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



System Calls for Process Management

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

The Process Model

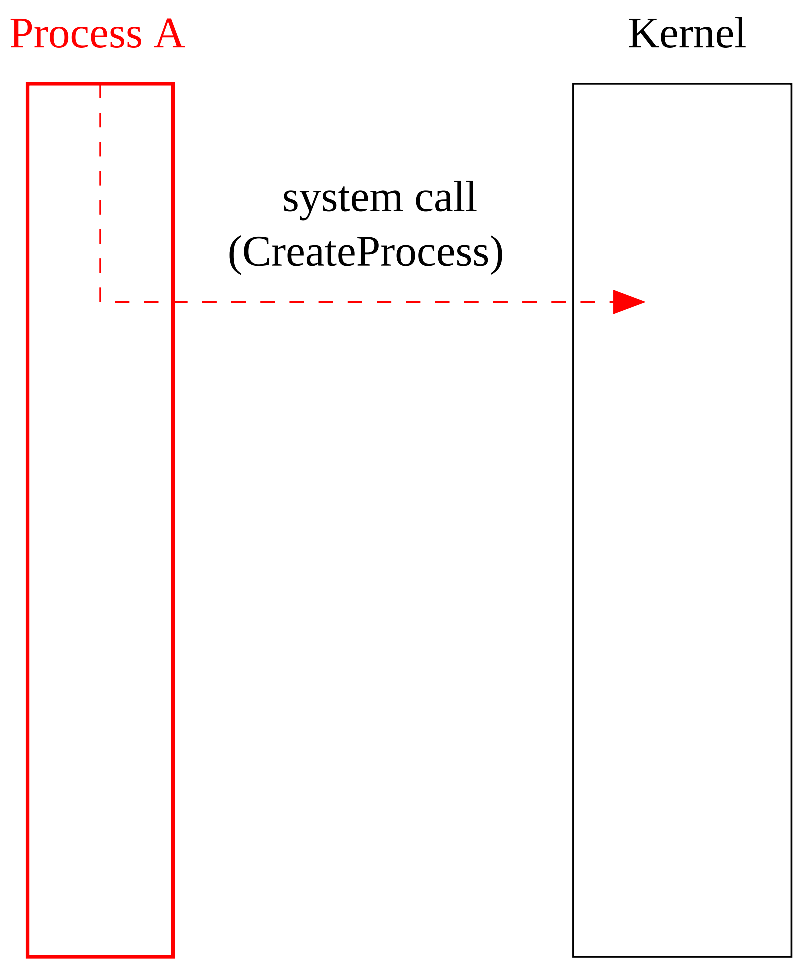
- Although the general operations supported by the process interface are straightforward, there are some less obvious aspects of process behaviour that must be defined by an operating system.

  Process Initialization: When a new process is created, how is it initialized? What is in the address space? What is the initial thread context? Does it have any other resources?

  Multithreading: Are concurrent processes supported, or is each process limited to a single thread?

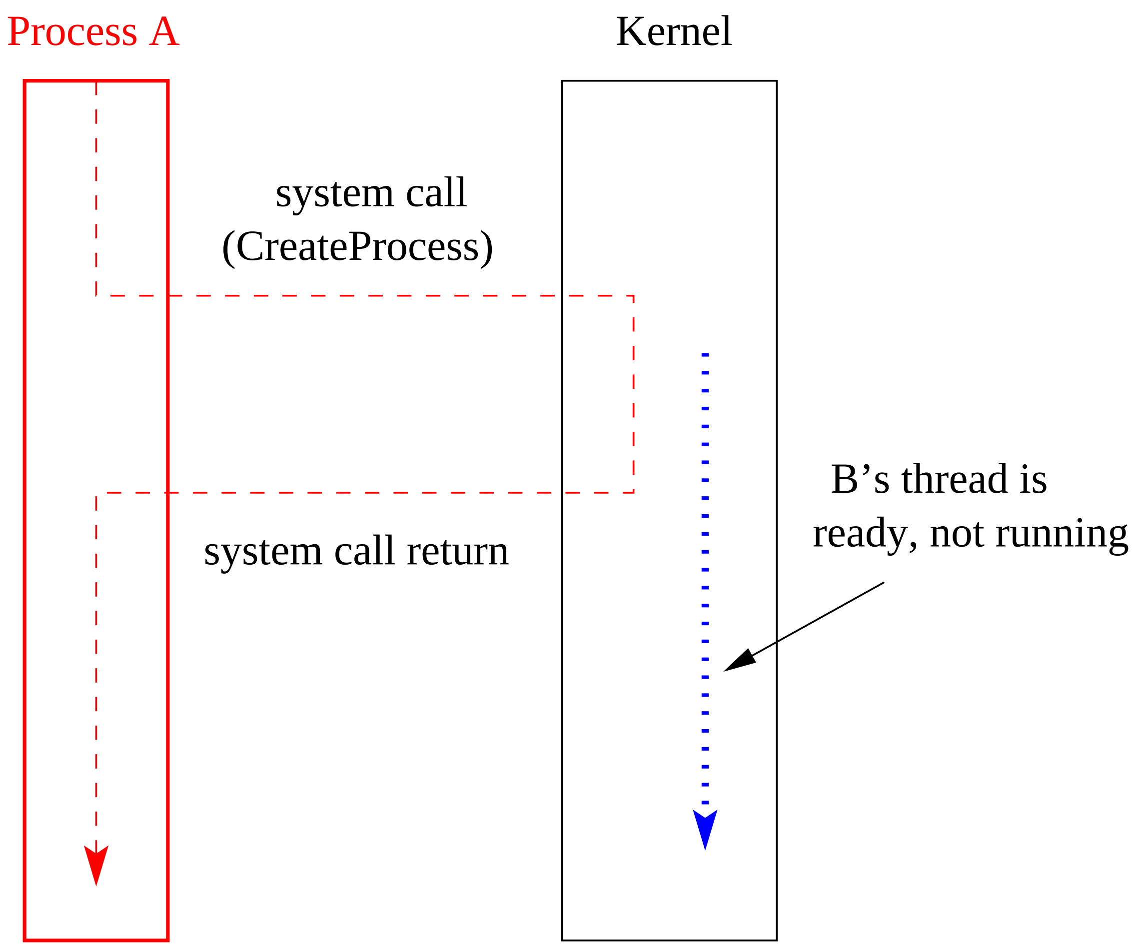
  Inter-Process Relationships: Are there relationships among processes, e.g, parent/child? If so, what do these relationships mean?

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)