The Operating System and the Kernel

- We will use the following terminology:

  kernel: The operating system kernel is the part of the operating system that responds to system calls, interrupts and exceptions.

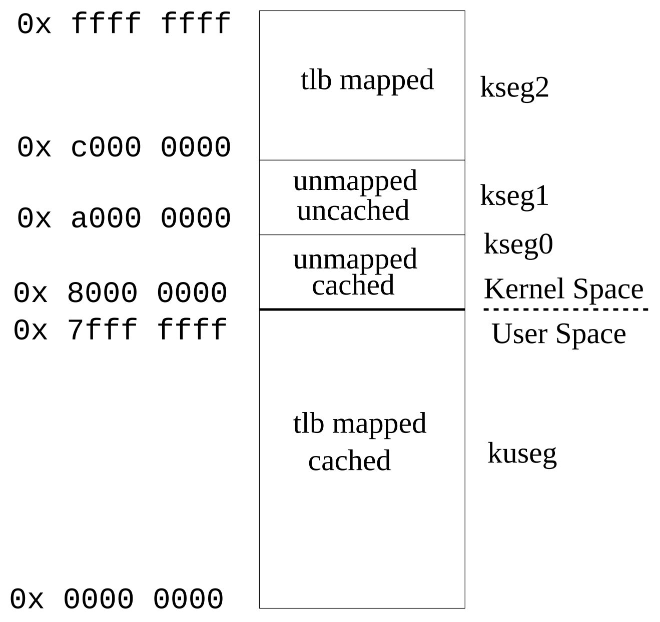
  operating system: The operating system as a whole includes the kernel, and may include other related programs that provide services for applications. This may include things like:
  - utility programs
  - command interpreters
  - programming libraries

The OS Kernel

- Usually kernel code runs in a privileged execution mode, while the rest of the operating system does not.
- The kernel is a program. It has code and data like any other program.
- 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.

MIPS Address Spaces and Protection

- On 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



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 virtual 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 the 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

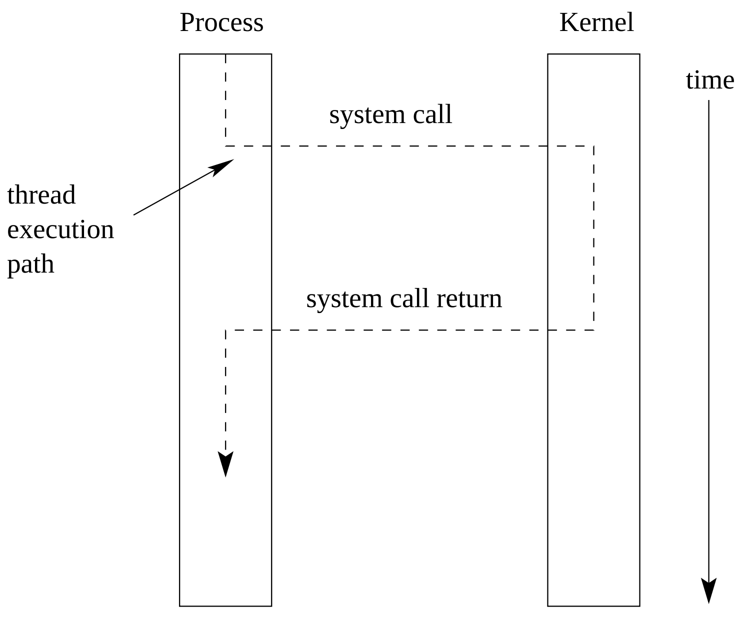
How System Calls Work

- 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, like the program counter and the stack pointer, are saved
  - the thread context is changed so that:
    - ∗ the program counter is set to a fixed (determined by the hardware) memory address, which is within the kernel's address space
    - ∗ the stack pointer is pointed at a stack in 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.

System Call Diagram



How a System Call Works

- Review: MIPS Register Usage

```
See also: kern/arch/mips/include/asmdefs.h
R0 =
R1 =
R2 =
R3 =
R4 =
R5 =
R6 =
R7 =
```

How a System Call Works

- Review: MIPS Register Usage

```
R08-R15 =
R24-R25 =

R16-R23 =

R26-27  =
R28     =

R29     =
R30     =
R31     =
```

How a System Call Works: User Code

```
004000b0 <__start>:
  4000b0: 3c1c1000  lui gp,0x1000
  4000b4: 279c7ff0  addiu gp,gp,32752
  4000b8: 3c08ffff  lui t0,0xffff
  4000bc: 3508fff8  ori t0,t0,0xfff8
  4000c0: 03a8e824  and sp,sp,t0
  4000c4: 27bdfff0  addiu sp,sp,-16
  4000c8: 3c011000  lui at,0x1000
  4000cc: ac250004  sw  a1,4(at)
```

How a System Call Works: User Code

```
  4000d0: 0c100040  jal 400100 <main>   # Call main
  4000d4: 00000000  nop
  4000d8: 00408021  move  s0,v0
  4000dc: 0c100050  jal 400140 <exit>
  4000e0: 02002021  move  a0,s0
  4000e4: 0c100069  jal 4001a4 <_exit>
  4000e8: 02002021  move  a0,s0
  4000ec: 02002021  move  a0,s0
  4000f0: 24020000  li  v0,0
  4000f4: 0000000c  syscall
  4000f8: 0810003b  j 4000ec <__start+0x3c>
  4000fc: 00000000  nop
```

How a System Call Works: User Code

```
/* See how a function/system call happens. */
#include <unistd.h>
#include <errno.h>

int
main()
{
  int x;
  int y;
  x = close(999);
  y = errno;

  return x;
}
```

How a System Call Works: User Code

```
% cs350-objdump -d syscall > syscall.out
% cat syscall.out

00400100 <main>:
  400100:     27bdffe0     addiu   sp,sp,-32
  400104:     afbf001c     sw      ra,28(sp)
  400108:     afbe0018     sw      s8,24(sp)
  40010c:     03a0f021     move    s8,sp
  400110:     0c10007b     jal     4001ec <close>
  400114:     240403e7     li      a0,999
  400118:     afc20010     sw      v0,16(s8)
  40011c:     3c021000     lui     v0,0x1000
  400120:     8c420000     lw      v0,0(v0)
  400124:     00000000     nop
```

How a System Call Works: User Code

```
<main> continued

  400128:     afc20014     sw      v0,20(s8)
  40012c:     8fc20010     lw      v0,16(s8)
  400130:     03c0e821     move    sp,s8
  400134:     8fbf001c     lw      ra,28(sp)
  400138:     8fbe0018     lw      s8,24(sp)
  40013c:     03e00008     jr      ra
  400140:     27bd0020     addiu   sp,sp,32
```

How a System Call Works: User Code

```
## See lib/libc/syscalls.S for details/comments */
## At bit easier to understand from disassembled code.

% cs350-objdump -d syscall > syscall.S
% cat syscall.S
...
0040022c <close>:
  40022c: 08100074  j 4001d0 <__syscall>
  400230: 24020007  li  v0,7

00400234 <reboot>:
  400234: 08100074  j 4001d0 <__syscall>
  400238: 24020008  li  v0,8
...
```

How a System Call Works: User Code

```
From lib/libc/syscalls.S
   .set noreorder
   .text
   .type __syscall,@function
   .ent __syscall

__syscall:
   syscall         /* make system call */
   beq a3, $0, 1f  /* if a3 is zero, call succeeded */
   nop             /* delay slot */
   sw v0, errno    /* call failed: store errno */
   li v1, -1       /* and force return value to -1 */
   li v0, -1
1:
   j ra            /* return */
   nop             /* delay slot */
   .end __syscall
   .set reorder
```

How a System Call Works: User Code

```
From lib/libc/syscalls.S

SYSCALL(close, 7)
SYSCALL(reboot, 8)
SYSCALL(sync, 9)
SYSCALL(sbrk, 10)
SYSCALL(getpid, 11)
```

How a System Call Works: Kernel Code

```
syscall instruction generates an exception.
Processor begins execution at virtual address 0x8000 0080
From: kern/arch/mips/mips/exception.S
## Q: where does this address live?

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 stac
  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 */
```

How a System Call Works: Kernel Code

```
From: kern/arch/mips/mips/exception.S
common_exception:
 o saves the contents of the registers
 o calls mips_trap (C code in kern/arch/mips/mips/trap.c)
 o restores the contents of the saved registers
 o rfe (return from exception)

From: kern/arch/mips/mips/trap.c
mips_trap:
  o figures out the exception type/cause
  o calls the appropriate handing function
    (for system call this is mips_syscall).
```

How a System Call Works: Kernel Code

```
From: kern/arch/mips/mips/syscall.c

mips_syscall(struct trapframe *tf)
{
  assert(curspl==0);
  callno = tf->tf_v0; retval = 0;

  switch (callno) {
    case SYS_reboot:
      /* is in kern/main/main.c */
      err = sys_reboot(tf->tf_a0);
      break;

    /* Add stuff here */

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

How a System Call Works: Kernel Code

```
  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);
}
```

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 error, e.g, overflow
  - illegal instruction
  - memory protection violation
  - page fault (to be discussed later)
- exceptions are detected by the hardware

Exceptions (cont'd)

- when an exception occurs, control is transferred (by the hardware) to a fixed address in the kernel (0x8000 0080 on MIPS & OS/161)
- transfer of control happens in much the same way as it does for a system call. In fact, a system call can be thought of as a type of exception, and they are sometimes implemented that way (e.g., on the MIPS).
- in the kernel, an exception handler determines which exception has occurred and what to do about it. For example, it may choose to destroy a process that attempts to execute an illegal instruction.

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.

Summary of Hardware Features Used by the Kernel

Interrupts and Exceptions, such as timer interrupts, give the kernel the opportunity to regain control from user programs.

Memory management features, such as memory protection, allow the kernel to protect its address space from user programs.

Privileged execution mode allows the kernel to reserve critical machine functions (e.g, halt) for its own use.

Independent I/O devices allow the kernel to schedule other work while I/O operations are on-going.