

# Processes and System Calls

## key concepts

process, system call, processor exception, fork/execv, multiprocessing

## reading

Three Easy Pieces: Chapter 4 (Processes), Chapter 5 (Process API), Chapter 6 (Direct Execution)

## What is a Process?

---

---

A process is an environment in which an application program runs.

---

---

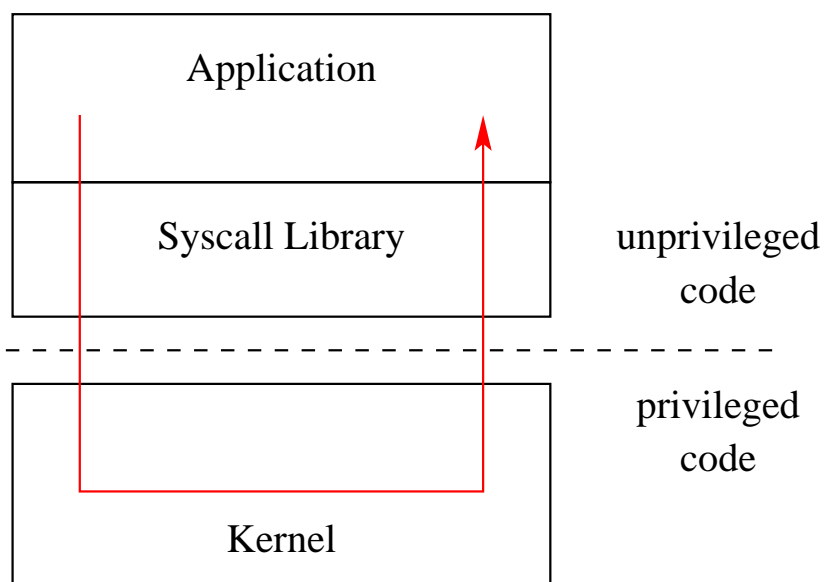
- a process includes virtualized *resources* that its program can use:
  - one (or more) threads
  - virtual memory, used for the program's code and data
  - other resources, e.g., file and socket descriptors
- processes are created and managed by the kernel
- each program's process *isolates* it from other programs in other processes

## System Calls

- System calls are the 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	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

- Kernel code runs at a higher level of *execution privilege* than application code
  - privilege levels are implemented by the CPU
- The kernel's higher privilege level allows it to do things that the CPU prevents less-privileged (application) programs from doing. For example:
  - application programs cannot modify the page tables that the kernel uses to implement process virtual memories
  - application programs cannot halt the CPU
- 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.

---

---

## How System Calls Work (Part 1)

---

---

Since application programs can't directly call the kernel, how does a program make a system call?

---

---

- There are only two things that make kernel code run:
  - **Interrupts**
    - \* interrupts are generated by devices
    - \* an interrupt means a device (hardware) needs attention
  - **Exceptions**
    - \* exceptions are caused by instruction execution
    - \* an exception means that a running program needs attention

## Interrupts, Revisited

- We have described interrupts already. Remember:
  - 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.

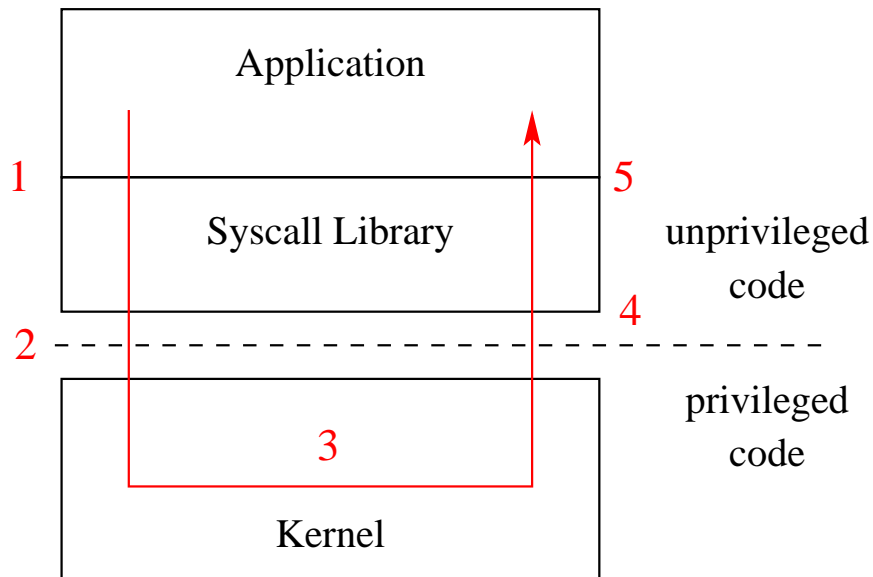
---

---

## 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.

### System Call Software Stack (again)



### System Call Timeline

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

### Which System Call?

- Q. There are many different system calls, but only one `syscall` exception. How does the kernel know *which* system call the application is requesting?
- A. 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)

### 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

- Q. System calls take parameters and return values, like function calls. How does this work, since system calls are really just exceptions?
- A. 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 the 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

## 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`

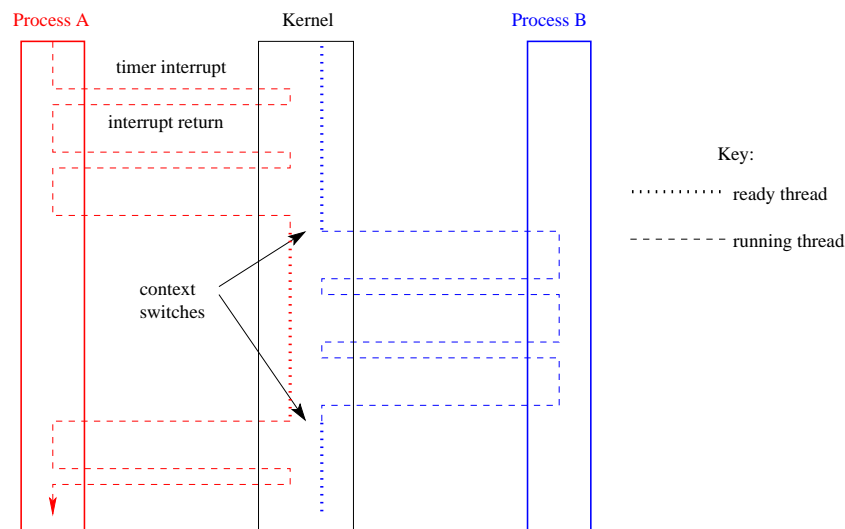
## `mips_trap`

- `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`

## 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.

## Multiprocessing Example



**Two process' threads timesharing a single CPU.**

## 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

## fork, \_exit, and waitpid

- `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` lets a process wait for another to terminate, and retrieve its exit status code

### 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))
    }
}
```

### The execv system call

- `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

**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**

```
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);
    }
}
```