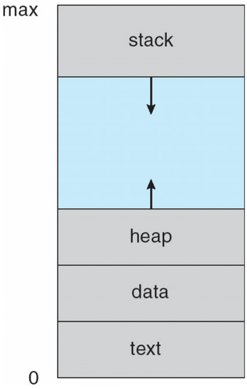
Processes

- **A *process* is an instance of a program running**
- **Modern OSes run multiple processes simultaneously**
- **Very early OSes only ran one process at a time**
- **Examples (can all run simultaneously):**
    - `emacs` – text editor
    - `firefox` – web browser
- **Non-examples (implemented as one process):**
    - Multiple firefox windows or emacs frames (still one process)
- **Why processes?**
    - Simplicity of programming
    - Speed: Higher throughput, lower latency

A process's view of the world

- **Each process has own view of machine**
  - Its own address space
  - Its own open files
  - Its own virtual CPU (through preemptive multitasking)
- `*(char *)0xc000` **different in P_1 & P_2**
- **Simplifies programming model**
  - `gcc` does not care that `firefox` is running



- **Sometimes want interaction between processes**
  - Simplest is through files: `emacs` edits file, `gcc` compiles it
  - More complicated: Shell/command, Window manager/app.

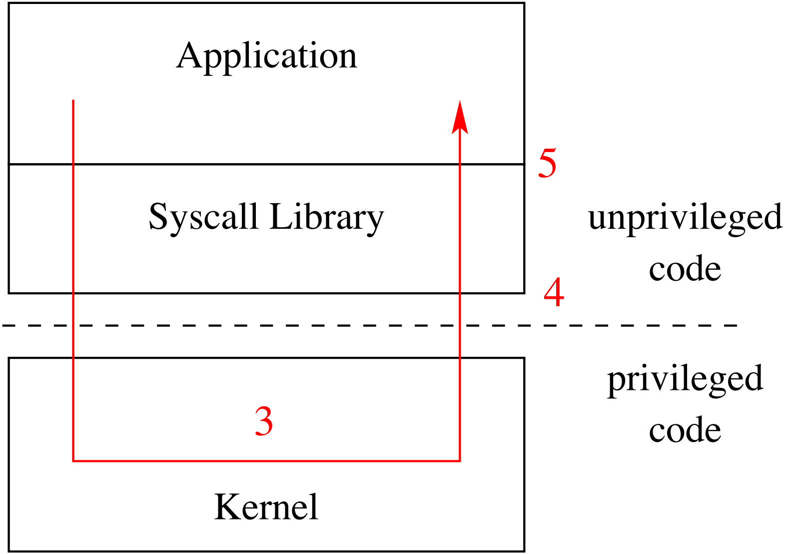
Outline

1. Application/Kernel Interface
2. User view of processes
3. Kernel view of processes

System Calls

- **Systems calls are the interface between processes and the kernel**
- **A process invokes a system call to request operating system services**
- `fork()`, `waitpid()`, `open()`, `close()`
- **Note: Signals are another common mechanism to allow the kernel to notify the application of an important event (e.g., Ctrl-C)**
  - Signals are like interrupts/exceptions for application code

System Call Software Stack



Kernel Privilege

- **Hardware provides two or more privilege levels (or protection rings)**
- **Kernel code runs at a higher privilege level than applications**
- **Typically called *Kernel Mode* vs. *User Mode***
- **Code running in kernel mode gains access to certain CPU features**
  - Accessing restricted features (e.g. Co-processor 0)
  - Disabling interrupts, setup interrupt handlers
  - Modifying the TLB (for virtual memory management)
- **Allows the kernel to isolate processes from one another and from the kernel**
  - Processes cannot read/write kernel memory
  - Processes cannot directly call kernel functions

How System Calls Work

- **The kernel only runs through well defined entry points**
- **Interrupts**
  - Interrupts are generated by devices to signal needing attention
  - E.g. Keyboard input is ready
- **Exceptions**
  - Exceptions are caused by the processor executing code
  - E.g. Divide by zero, page fault, etc.

Interrupts

- **An interrupt or exception causes the hardware to transfer control to a fixed location in memory, where the *interrupt handler* is located**
- **Interrupt handlers are part of the kernel**
- **When an interrupt occurs, the processor switches to kernel mode (or privileged mode) allowing the kernel to take over**
  - This is how the kernel gets run with privileges
  - Interrupts can still be delivered while running the kernel
  - Exception is that spinlocks disabled interrupts

Exceptions

- **Exceptions are conditions that occur during the execution of a program (or kernel) that require attention**
  - E.g. divide by zero, page faults, illegal instructions, etc.
- **Exceptions are detected by the CPU during execution**
- **CPU handles exceptions just like interrupts by transferring control to the kernel**
  - Control is transferred to a fixed location where the exception handler is located
  - Processor is switches into privileged mode

MIPS Exception Vectors

```
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 */
```

- **Interrupts, exceptions, and system calls are handled through the same mechanism**
- **Some processors specially handle system calls for performance reasons**

How System Calls Work Continued

- **System calls are performed by triggering an exception**
- **Applications execute the `syscall` instruction to trigger the `EX_SYS` exception**
  - Many processors include a similar instruction
  - For example, x86 contains the `syscall` and/or `sysenter` instructions, but with an optimized implementation

Hardware Handling

- **Exception handlers in the R3000 are at fixed locations**
- **The processor jumps to these addresses whenever an exception is encountered**
  - `0x8000_0000` User TLB Handler
  - `0x8000_0080` General Exception Handler
- **Remember that in MIPS `0x8000_0000-0x9FFF_FFFF` is mapped to the first 512 MBs of physical memory.**

Hardware Handling Continued

- **System Control Coprocessor (CP0) contains all the information regarding the exception**
  - Use the mfc0/mtc0 (Move from/to co-processor 0) instruction
  - `c0_status` CPU status including kernel/user mode flag
  - `c0_cause` Cause of the exception
  - `c0_epc` PC where the exception occurred
  - `c0_vaddr` Virtual address associated with fault (e.g. page fault)
  - `c0_context` Used by OS to store the CPU number

System Call Operations

- **Application calls into C library (e.g. calls `write()`)**
- **Library executes the `syscall` instruction**
- **Kernel exception handler `0x8000_0080` runs**
  - Switch to kernel stack
  - Create a trap frame to save program state
  - Determine the type of system call
  - Determine which system call is being invoked
  - Process call
  - Restore application state from trap frame
  - Return from exception
- **Library wrapper function returns to application**

Application Binary Interface/Calling Conventions

- **Each architecture and OS define calling conventions**
- **Describes how registers are used in function calls and system calls**
- **MIPS+OS/161 Calling Conventions**
  - System call number in `v0`
  - First four arguments in `a0`, `a1`, `a2`, `a3`
  - Remaining arguments passed on stack
  - Result success/fail in `a3` and return value/error code in `v0`
- **Number for each system call in kern/include/kern/syscall.h**

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

OS/161 Code Walkthrough

- `kern/arch/sys161/startup/start.S`
- `kern/arch/mips/locore/exception-mips1.S`
- `kern/arch/mips/locore/trap.c`
- `kern/arch/mips/syscall/syscall.c`

Outline

1. Application/Kernel Interface
2. User view of processes
3. Kernel view of processes

Creating processes

- Original UNIX paper is a great reference on core system calls
- `int fork (void);`
  - Create new process that is exact copy of current one
  - Returns *process ID* of new process in “parent”
  - Returns 0 in “child”
- `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

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` most common value, kills process by default (but application can catch it for “cleanup”)
  - `SIGKILL` stronger, kills process always
- `pid_t getpid(void);`
  - Get the current process ID
- `pid_t getppid(void);`
  - Get the process ID of the parent process

Running programs

- `int execve (char *prog, char **argv, char **envp);`
  - `prog` – full pathname of program to run
  - `argv` – argument vector that gets passed to `main`
  - `envp` – environment variables, e.g., `PATH`, `HOME`
- **Generally called through a wrapper functions**
  - `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`
- **Example:** `minish.c`
  - Loop that reads a command, then executes it

Process Startup: user/lib/crt0/mips/crt0.S

```
__start:
  /* Load the "global pointer" register */
  la gp, _gp

  /* argc in a0 and argv in a1 */
  li t0, 0xfffffff8          /* mask for stack alignment */
  and sp, sp, t0                    /* align the stack */
  addiu sp, sp, -16          /* create our frame */

  sw a1, __argv /* save second arg (argv) in __argv */

  jal main     /* call main */
  nop          /* delay slot */
```

Process Exit: user/lib/crt0/mips/crt0.S

```
  move s0, v0  /* save return value */
  jal exit     /* call exit() */
  move a0, s0         /* Set argument (in delay slot) */

  jal _exit    /* Try _exit() */
  move a0, s0  /* Set argument (in delay slot) */

1:
  move a0, s0
  li v0, SYS__exit
  syscall

  j 1b         /* loop back */
  nop          /* delay slot */
```

`minishell.c` (simplified)

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

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

UNIX file I/O

- **Applications "open" files (or devices) by name**
  - I/O happens through open files
- `int open(char *path, int flags, /*mode*/...);`
  - `flags`: `O_RDONLY`, `O_WRONLY`, `O_RDWR`
  - `O_CREAT`: create the file if non-existent
  - `O_EXCL`: (w. `O_CREAT`) create if file exists already
  - `O_TRUNC`: Truncate the file
  - `O_APPEND`: Start writing from end of file
  - `mode`: final argument with `O_CREAT`
- **Returns file descriptor—used for all I/O to file**

Error returns

- **What if `open` fails? Returns -1 (invalid fd)**
- **Most system calls return -1 on failure**
  - Specific kind of error in global int `errno`
- `#include <sys/errno.h>` **for possible values**
  - 2 = `ENOENT` "No such file or directory"
  - 13 = `EACCES` "Permission Denied"
- `perror` **function prints human-readable message**
  - `perror ("initfile");`
    → "`initfile: No such file or directory`"
- **Details:**
  - Typically `errno` is a thread local variable
  - FreeBSD: C macro that calls `__errno()` to return the result

System Calls: lib/libc/arch/mips/syscalls-mips.S

```
#define SYSCALL(sym, num) \
   .globl sym                ; \
   .type sym,@function       ; \
sym:                         ; \
   j __syscall               ; \
   addiu v0, $0, SYS_##sym   ; \
   .end sym                  ; \
   .set reorder

__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 */
```

Operations on file descriptors

- `int read (int fd, void *buf, int nbytes);`
  - Returns number of bytes read
  - Returns 0 bytes at end of file, or -1 on error
- `int write (int fd, void *buf, int nbytes);`
  - Returns number of bytes written, -1 on error
- `off_t lseek (int fd, off_t pos, int whence);`
  - `whence`: 0 – start, 1 – current, 2 – end
    - Returns previous file offset, or -1 on error
- `int close (int fd);`

File descriptor numbers

- **File descriptors are inherited by processes**
  - When one process spawns another, same fds by default
- **Descriptors 0, 1, and 2 have special meaning**
  - 0 – “standard input” (`stdin` in ANSI C)
  - 1 – “standard output” (`stdout, printf` in ANSI C)
  - 2 – “standard error” (`stderr, perror` in ANSI C)
  - Normally all three attached to terminal
- **Example:** `type.c`
  - Prints the contents of a file to `stdout`

type.c

```
void
typefile (char *filename)
{
    int fd, nread;
    char buf[1024];

    fd = open (filename, O_RDONLY);
    if (fd == -1) {
        perror (filename);
        return;
    }

    while ((nread = read (fd, buf, sizeof (buf))) > 0)
        write (1, buf, nread);

    close (fd);
}
```

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 (`lseek` on one will affect both)
- `int fcntl (int fd, F_SETFD, int val)`
  - Sets *close on exec* flag if `val` = 1, clears if `val` = 0
  - Makes file descriptor non-inheritable by spawned programs
- **Example:** `redirsh.c`
  - Loop that reads a command and executes it
  - Recognizes `command < input > output 2> errlog`

redirsh.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);
}
```

Pipes

- `int pipe (int fds[2]);`
  - Returns two file descriptors in `fds[0]` and `fds[1]`
  - Data written to `fds[1]` will be returned by `read` on `fds[0]`
  - When last copy of `fds[1]` closed, `fds[0]` will return EOF
  - Returns 0 on success, -1 on error
- **Operations on pipes**
  - `read`/`write`/`close` – as with files
  - When `fds[1]` closed, `read(fds[0])` returns 0 bytes
  - When `fds[0]` closed, `write(fds[1])`:
    - ▷ Kills process with `SIGPIPE`
    - ▷ Or if signal ignored, fails with EPIPE
- **Example:** `pipesh.c`
  - Sets up pipeline `command1 | command2 | command3 ...`

`pipesh.c` (simplified)

```
void doexec (void) {
  while (outcmd) {
    int pipefds[2]; pipe (pipefds);
    switch (fork ()) {
    case -1:
      perror ("fork"); exit (1);
    case 0:
      dup2 (pipefds[1], 1);
      close (pipefds[0]); close (pipefds[1]);
      outcmd = NULL;
      break;
    default:
      dup2 (pipefds[0], 0);
      close (pipefds[0]); close (pipefds[1]);
      parse_command_line (&av, &outcmd, outcmd);
      break;
    }
  }
  ⋮
```

Why fork?

- **Most calls to `fork` followed by `execve`**
- **Could also combine into one *spawn* system call**
- **Occasionally useful to fork one process**
  - Unix *dump* utility backs up file system to tape
  - If tape fills up, must restart at some logical point
  - Implemented by forking to revert to old state if tape ends
- **Real win is simplicity of interface**
  - Tons of things you might want to do to child: Manipulate file descriptors, set environment variables, reduce privileges, …
  - Yet `fork` requires *no* arguments at all

Spawning a process without fork

- **Without fork, needs tons of different options for new process**
- **Example: Windows `CreateProcess` system call**
  - Also `CreateProcessAsUser`, `CreateProcessWithLogonW`, `CreateProcessWithTokenW`, ...

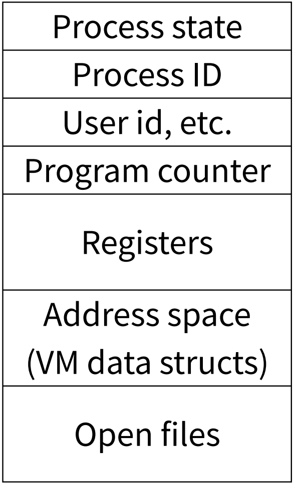
```
BOOL WINAPI CreateProcess(
  _In_opt_      LPCTSTR lpApplicationName,
  _Inout_opt_   LPTSTR lpCommandLine,
  _In_opt_      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  _In_opt_      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  _In_          BOOL bInheritHandles,
  _In_          DWORD dwCreationFlags,
  _In_opt_      LPVOID lpEnvironment,
  _In_opt_      LPCTSTR lpCurrentDirectory,
  _In_          LPSTARTUPINFO lpStartupInfo,
  _Out_         LPPROCESS_INFORMATION lpProcessInformation
);
```

Outline

1. Application/Kernel Interface
2. User view of processes
3. Kernel view of processes

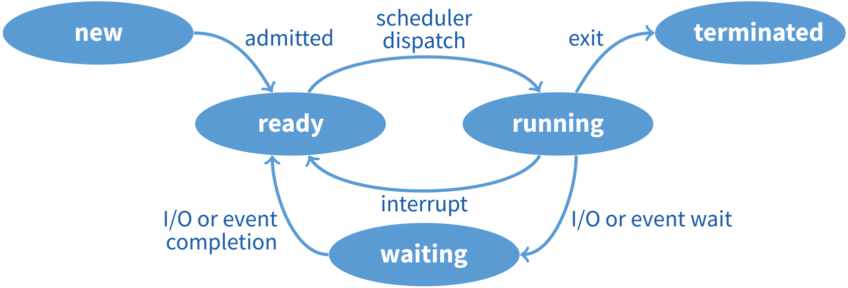
Implementing processes

- **Keep a data structure for each process**
  - Process Control Block (PCB)
  - Called `proc` in Unix, `task_struct` in Linux
- **Tracks *state* of the process**
  - Running, ready (runnable), waiting, etc.
- **Includes information necessary to run**
  - Registers, virtual memory mappings, etc.
  - Open files (including memory mapped files)
- **Various other data about the process**
  - Credentials (user/group ID), signal mask, controlling terminal, priority, accounting statistics, whether being debugged, which system call binary emulation in use, …


Process state
Process ID
User id, etc.
Program counter
Registers
Address space (VM data structs)
Open files

PCB

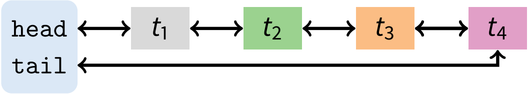
Process states



- **Process can be in one of several states**
  - *new* & *terminated* at beginning & end of life
  - *running* – currently executing (or will execute on kernel return)
  - *ready* – can run, but kernel has chosen different process to run
  - *waiting* – needs async event (e.g., disk operation) to proceed
- **Which process should kernel run?**
  - if 0 runnable, run idle loop (or halt CPU), if 1 runnable, run it
  - if $>$1 runnable, must make scheduling decision

Scheduling

- **How to pick which process to run**
- **Scan process table for first runnable?**
  - Expensive. Weird priorities (small pids do better)
  - Divide into runnable and blocked processes
- **FIFO?**
  - Put threads on back of list, pull them from front:



- **Priority?**
  - Give some threads a better shot at the CPU

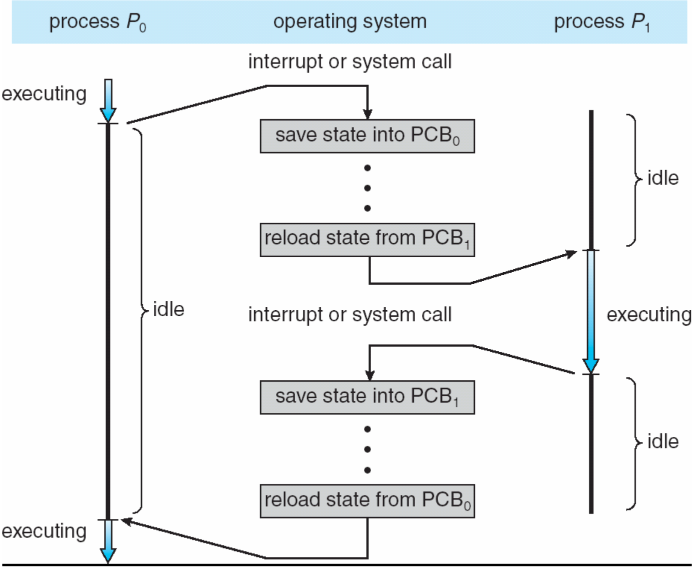
Scheduling policy

- **Want to balance multiple goals**
  - *Fairness* – don't starve processes
  - *Priority* – reflect relative importance of procs
  - *Deadlines* – must do *X* (play audio) by certain time
  - *Throughput* – want good overall performance
  - *Efficiency* – minimize overhead of scheduler itself
- **No universal policy**
  - Many variables, can't optimize for all
  - Conflicting goals (e.g., throughput or priority vs. fairness)
- **We will spend a whole lecture on this topic**

Preemption

- **Can preempt a process when kernel gets control**
- **Running process can vector control to kernel**
  - System call, page fault, illegal instruction, etc.
  - May put current process to sleep—e.g., read from disk
  - May make other process runnable—e.g., fork, write to pipe
- **Periodic timer interrupt**
  - If running process used up quantum, schedule another
- **Device interrupt**
  - Disk request completed, or packet arrived on network
  - Previously waiting process becomes runnable
  - Schedule if higher priority than current running proc.
- **Changing running process is called a *context switch***

Context switch



Context switch details

- **Very machine dependent. Typical things include:**
    - Save program counter and integer registers (always)
    - Save floating point or other special registers
    - Save condition codes
    - Change virtual address translations
- **Non-negligible cost**
    - Save/restore floating point registers expensive
        - Optimization: only save if process used floating point
    - May require flushing TLB (memory translation hardware)
        - HW Optimization 1: don't flush kernel's own data from TLB
        - HW Optimization 2: use tag to avoid flushing any data
    - Usually causes more cache misses (switch working sets)