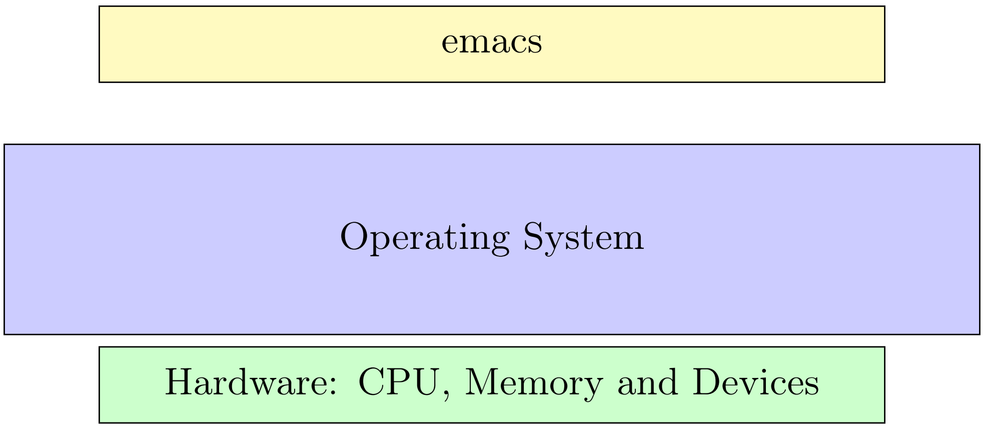
CS350: Processes

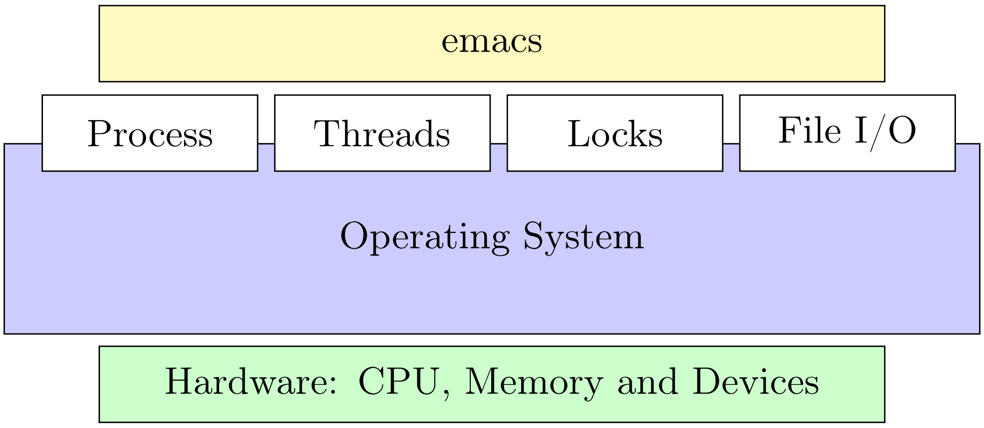
Zille Huma Kamal and Emil Tsalapatis

University of Waterloo

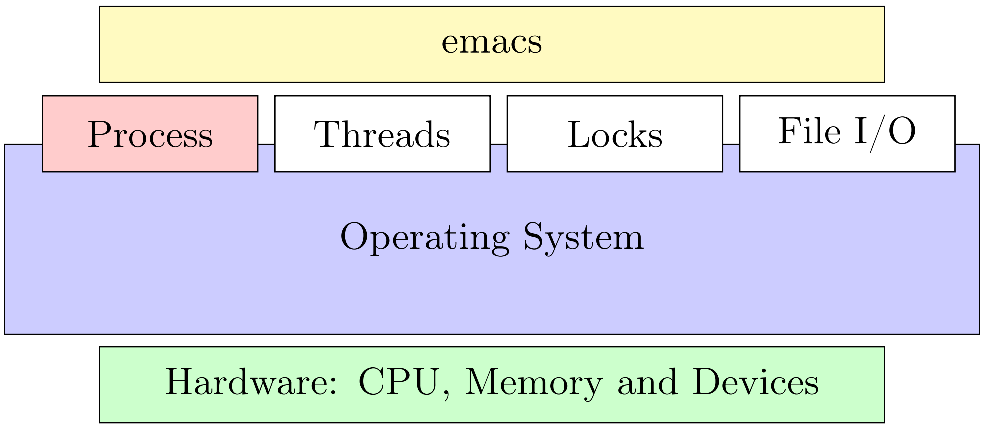
Operating System



Operating System: Basic Abstractions and APIs



Today: Introduce the Process Abstraction



Processes

A process is an instance of a program running

Examples (can all run simultaneously):

- `gcc file_A.c` – compiler running on file A
- `gcc file_B.c` – compiler running on file B
- `emacs` – text editor
- `firefox` – web browser

Non-examples (implemented as one process):

- Multiple firefox windows or emacs frames (still one process)

Modern OSes run multiple processes simultaneously

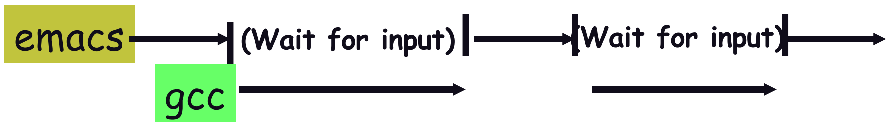
Why processes?

- Simplicity of programming
- Higher throughput (better CPU utilization), lower latency

Speed

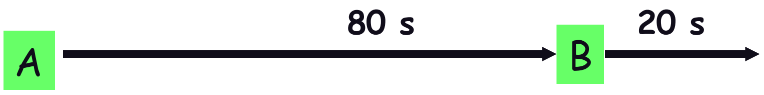
Multiple processes can increase CPU utilization

- Overlap one process's computation with another's wait

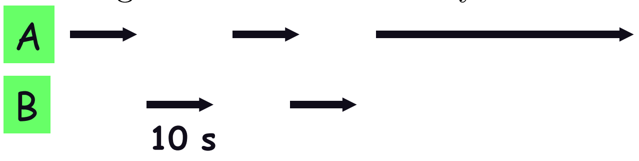


Multiple processes can reduce latency

- Running A then B requires 100 sec for B to complete



- Running A and B concurrently makes B finish faster



- A is slower than if it had whole machine to itself,
  but still $<$ 00 sec unless both A and B completely CPU-bound

Concurrency and parallelism

Parallelism fact of life much longer than OSes have been around

- E.g., say takes 1 worker 10 months to make 1 widget
- Latency for first widget   0 month
- Company may hire 100 workers to make 100 widgets
- Throughput may be $<$ 0 widgets per month
  (if can't perfectly parallelize task)
- And 100 workers making 10,000 widgets may achieve   0 widgets/month

Most computers, laptops, and phones are multi-core!

Computer with 4 cores can run 4 processes in parallel

Result: $\times$ throughput

Lecture Objectives

Process's view of the world

Kernel view of processes

- Implementing processes in the kernel

User view of processes

- Crash course in basic Unix/Linux system call interface
- How to create, kill, and communicate between processes
- Running example: how to implement a shell

Outline

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)

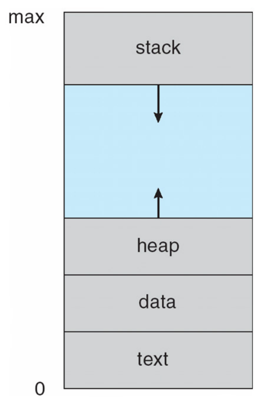
`*(char *)0xc000` different in P & P_2

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. Process's view of the world
2. Kernel view of processes
3. User view of processes

Implementing processes

OS keeps data structure for each proc

- Process Control Block (PCB)
- Called `proc` in Unix, `task_struct` in Linux, and just `struct thread` in OS/161

Tracks state of the process

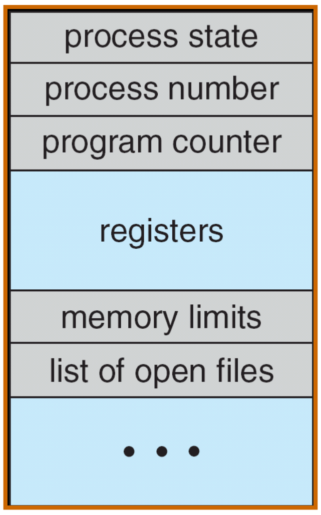
- Running, ready (runnable), blocked, etc.

Includes information necessary to run

- Registers, virtual memory mappings, etc.
- Open files (including memory mapped files)

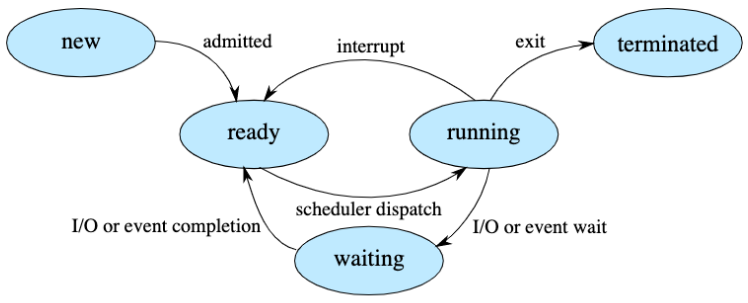
Various other data about the process

- Credentials (user/group ID), signal mask, controlling terminal, priority, accounting statistics, whether being debugged, which system call binary emulation in use, ...



PCB

Mutliprogramming - 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

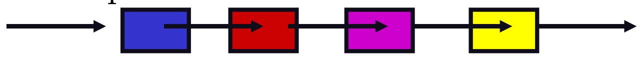
How to pick which process to run

Scan process table for first runnable?

- Expensive. Weird priorities (small pids do better)
- Divide into runnable and blocked processes

FIFO/Round-Robin?

- Select process to run based on order of arrival to the ready queue



Priority?

- Give some processes a better shot at the CPU

We will spend a whole lecture on the topic of Scheduling

Time Sharing - Preemption

Preemptive Kernel

- interrupts—periodic timer interrupt
- system call—e.g. read from disk, write to stdout buffer
- Schedule if higher priority than current running process

Periodic timer interrupt

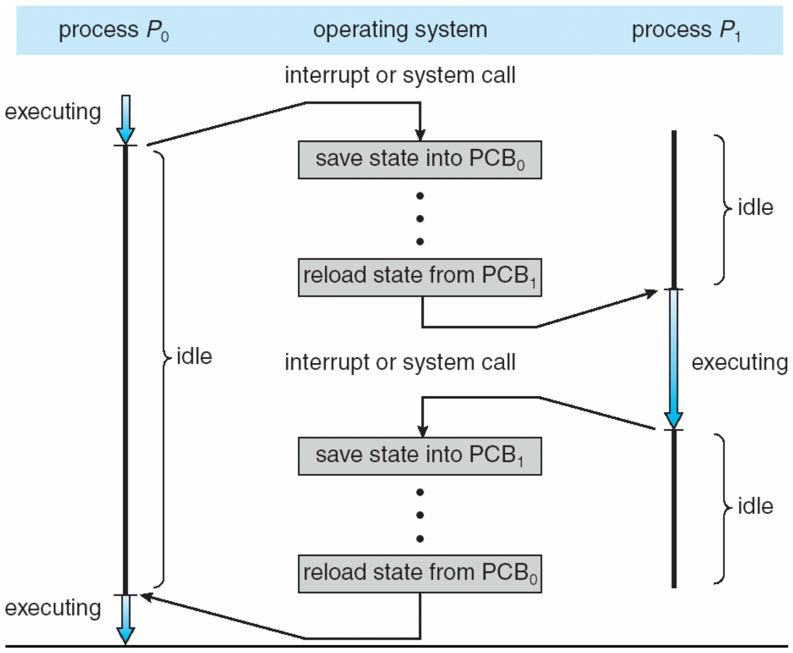
- If running process used up quantum, schedule another

Device interrupt

- Disk request completed, or packet arrived on network
- Previously waiting process becomes runnable

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

Protection - Privilege Modes

Hardware provides multiple protection modes

At least two modes:

- Kernel Mode or Privledged Mode – Operating System
- User Mode – Applications

Kernel Mode can access privileged CPU features

- Access all restricted CPU features
- Enable/disable interrupts, setup interrupt handlers
- Control system call interface
- Modify the TLB (virtual memory ... future lecture)

Allows kernel to protect itself and isolate processes

- Processes cannot read/write kernel memory
- Processes cannot directly call kernel functions

Mode Transitions

Kernel Mode can only be entered through well defined entry points

Two classes of entry points provided by the processor:

Interrupts

- Interrupts are generated by devices to signal needing attention
- E.g. Keyboard input is ready
- More on this during our IO lecture!

Exceptions:

- Exceptions are caused by processor
- E.g. Divide by zero, page faults, internal CPU errors

Interrupts and exceptions cause hardware to transfer control to the interrupt/exception handler, a fixed entry point in the kernel.

Interrupts

Interrupt are raised by devices

Interrupt handler is a function in the kernel that services a device request

Interrupt Process:

- Device signals the processor through a physical pin or bus message
- Processor interrupts the current program
- Processor begins executing the interrupt handler in privileged mode

Most interrupts can be disabled, but not all

- Non-maskable interrupts (NMI) is for urgent system requests

Exceptions

Exceptions (or faults) are conditions encountered during execution of a program

- Exceptions are due to multiple reasons:
- Program Errors: Divide-by-zero, illegal instructions
- Operating System Requests: Page faults
- Hardware Errors: System check (bad memory or internal CPU failures)

CPU handles exceptions similar to interrupts

- Processor stops at the instruction that triggered the exception (usually)
- Control is transferred to a fixed location where the exception handler is located in privledged mode

System calls are a class of exceptions!

Execution Contexts

Execution Context: The environment where functions execute including their arguments, local variables, memory.

Context is a unique set of CPU registers and a stack pointer

Multiple execution contexts:

- Application Context: user level process
- Kernel Context: privileged instructions, software interrupts, etc
- Interrupt Context: Interrupt handler

Kernel and Interrupts usually the same context

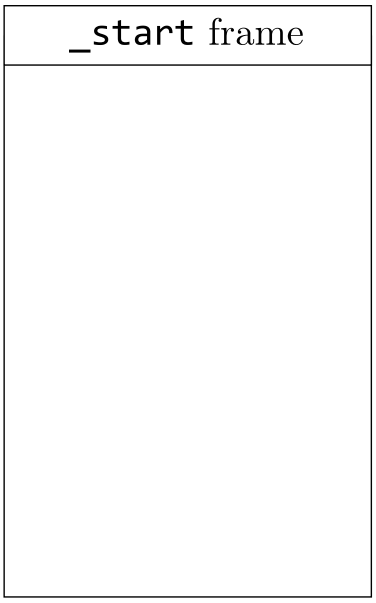
Context transitions:

- Context switch: a transitions between contexts

Application Stack

Stack made of up frames containing locals, arguments, and spilled registers

Programs begin execution at `_start`

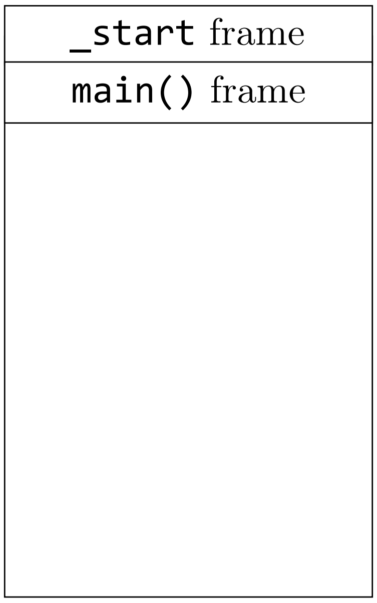


User Stack

Application Stack

Stack made of up frames containing locals, arguments, and spilled registers

Programs begin execution at `_start`

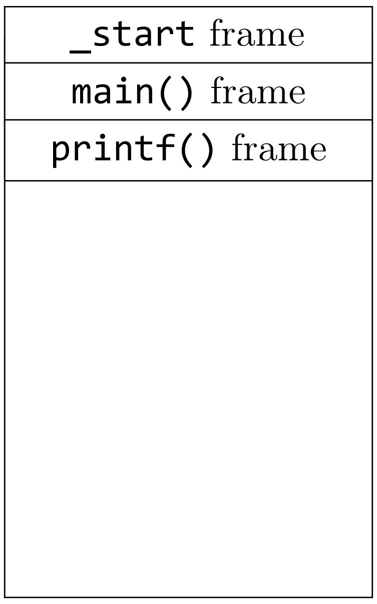


User Stack

Application Stack

Stack made of up frames containing locals, arguments, and spilled registers

Programs begin execution at `_start`

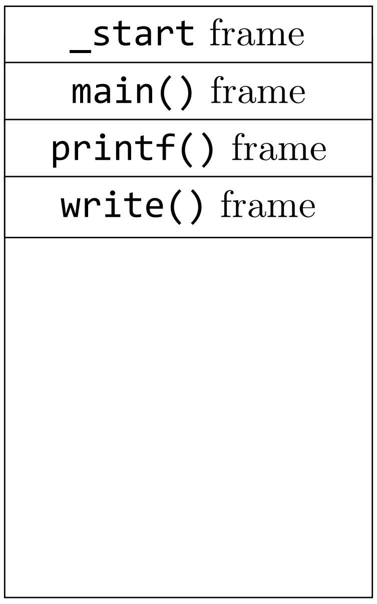


User Stack

Application Stack

Stack made of up frames containing locals, arguments, and spilled registers

Programs begin execution at `_start`

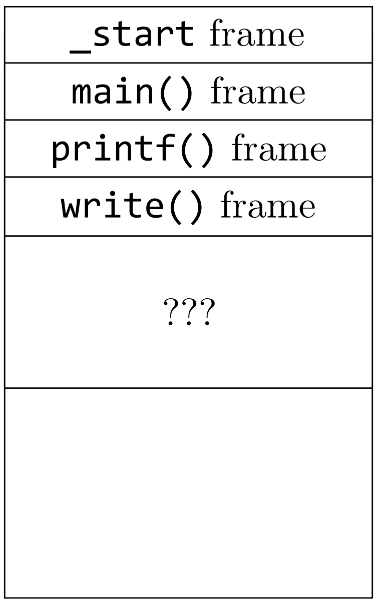


User Stack

Application Stack

Stack made of up frames containing locals, arguments, and spilled registers

Programs begin execution at `_start`

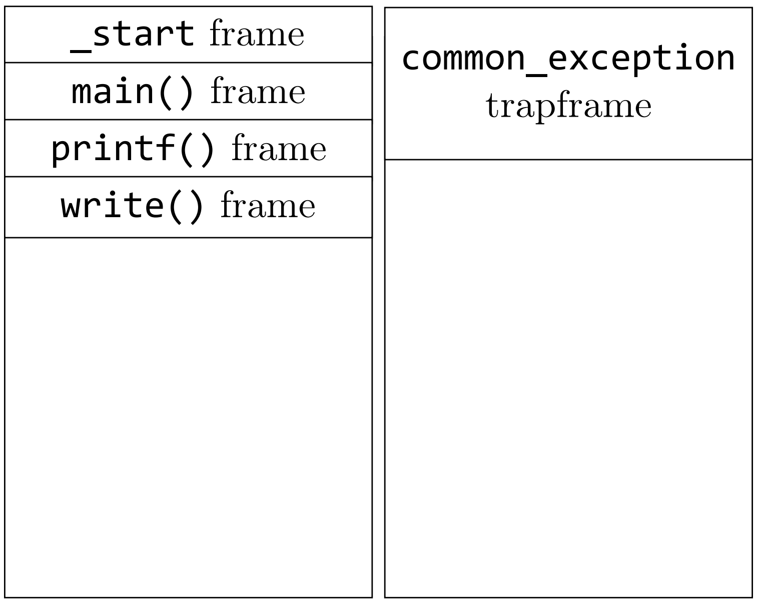


User Stack

Context Switch: User to Kernel

trapframe: Saves the application context

`syscall` instruction triggers the exception handler

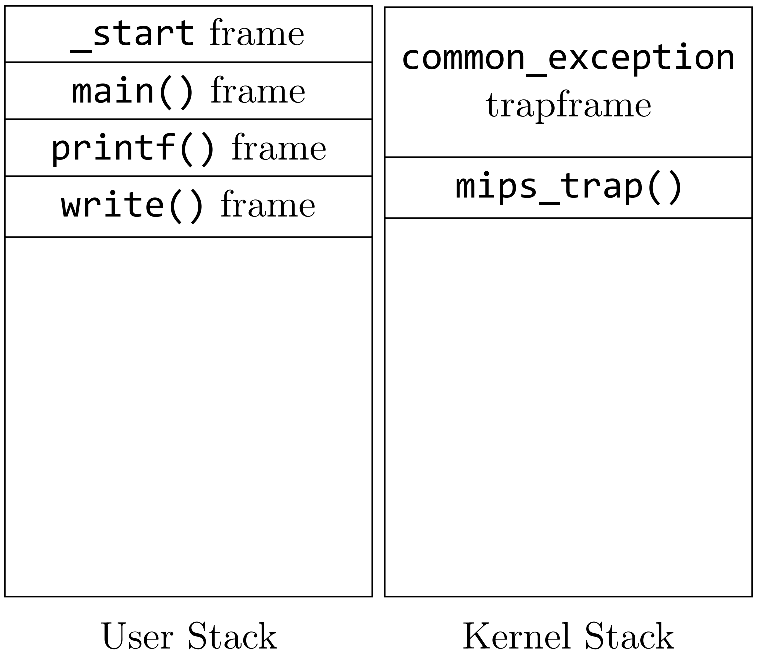


User Stack Kernel Stack

Context Switch: User to Kernel

trapframe: Saves the application context

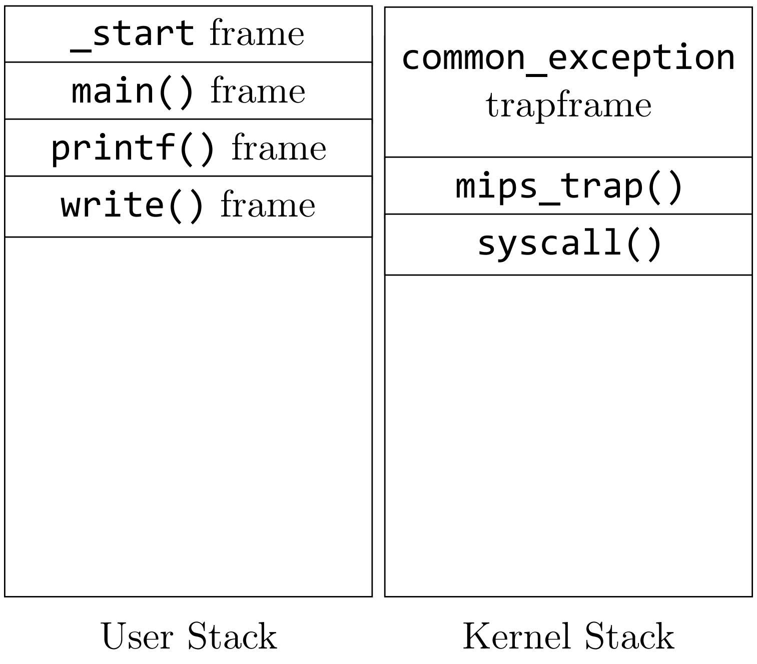
`common_exception` saves trapframe on the kernel stack!



Context Switch: User to Kernel

trapframe: Saves the application context

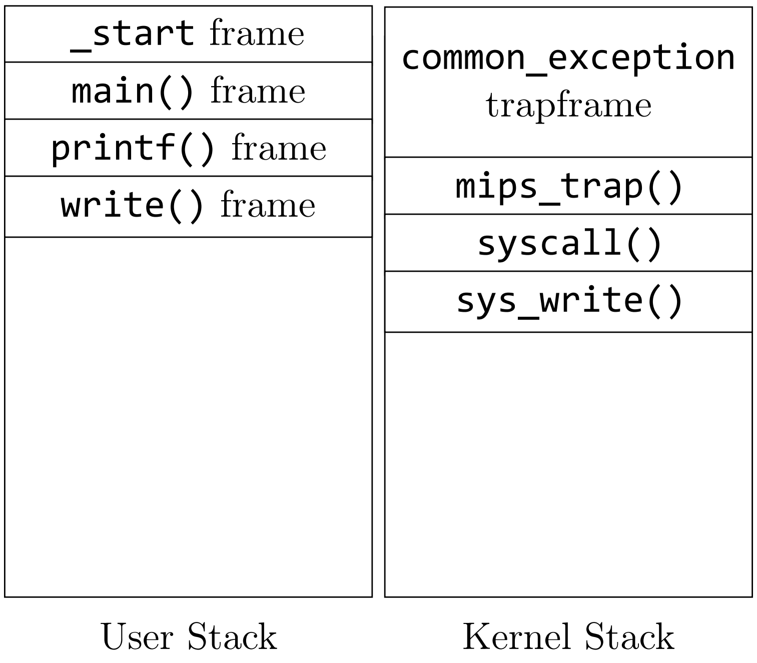
Calls `mips_trap()` to decode trap and `syscall()`



Context Switch: User to Kernel

trapframe: Saves the application context

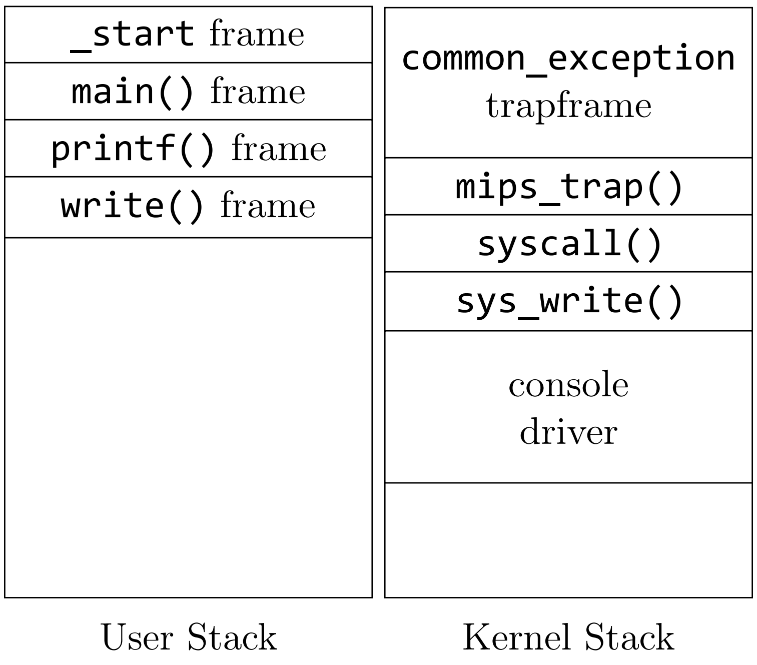
`syscall()` decodes arguments and calls `sys_write()`



Context Switch: Returning to User Mode

trapframe: Saves the application context

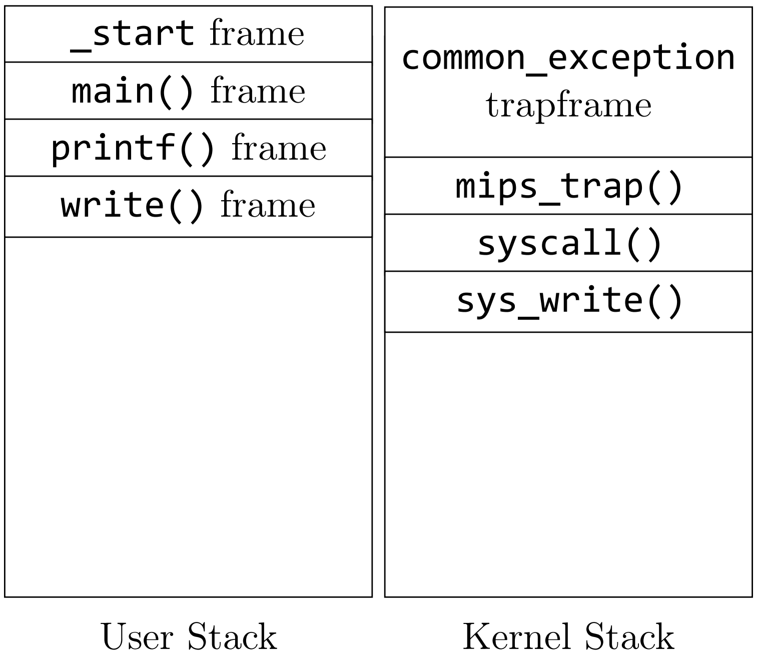
`sys_write()` writes text to console



Context Switch: Returning to User Mode

trapframe: Saves the application context

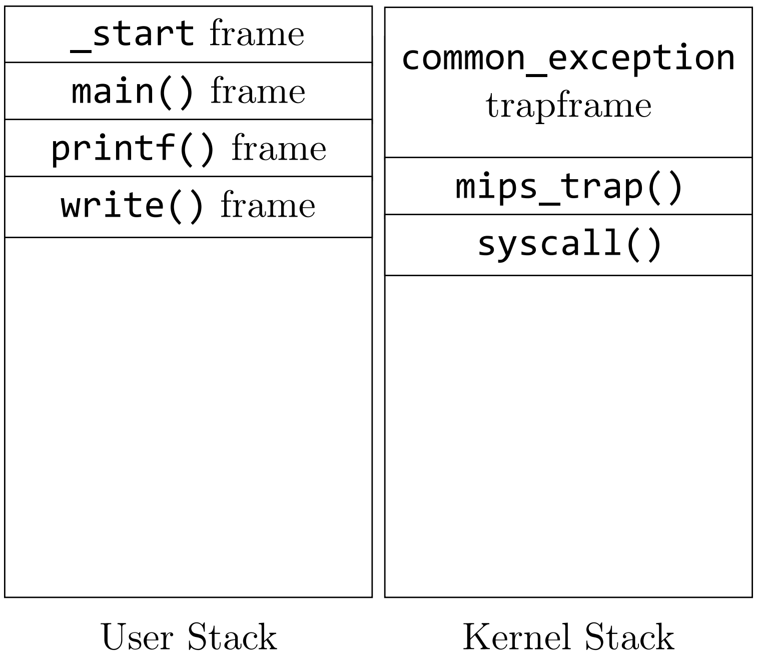
Return from `sys_write()`



Context Switch: Returning to User Mode

`syscall()` stores return value and error in trapframe

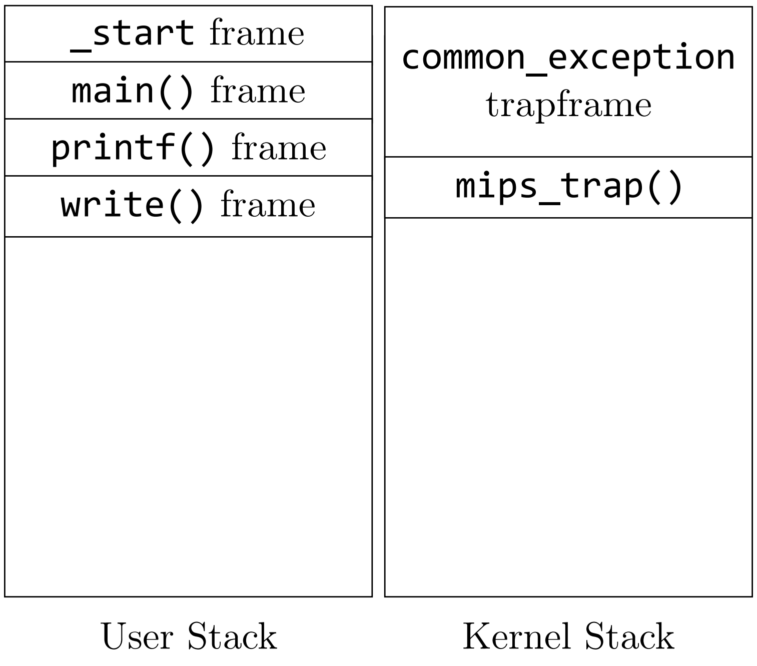
`v0`: return value/error code, `a3`: success (1) or failure



Context Switch: Returning to User Mode

`mips_trap()` returns to the instruction following `syscall`

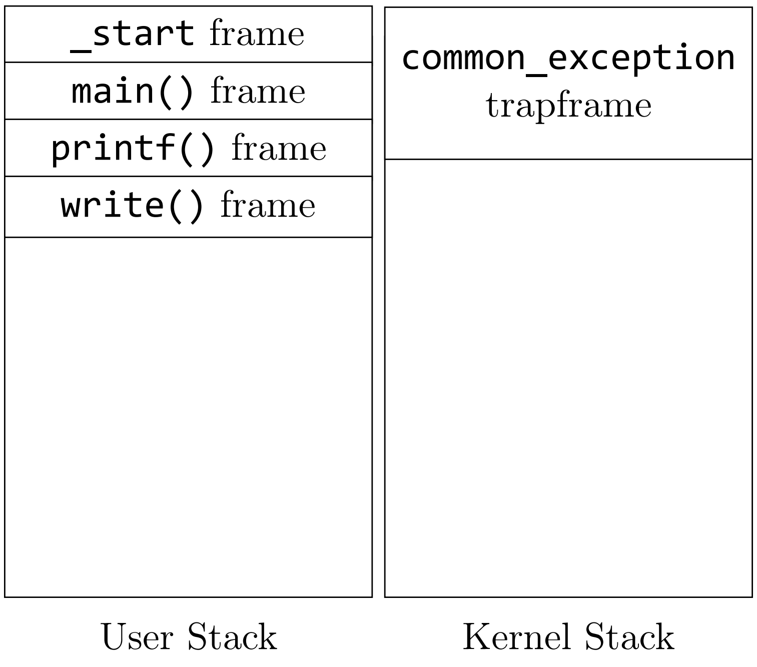
`v0`: return value/error code, `a3`: success (1) or failure



Context Switch: Returning to User Mode

`common_exception` restores the application context

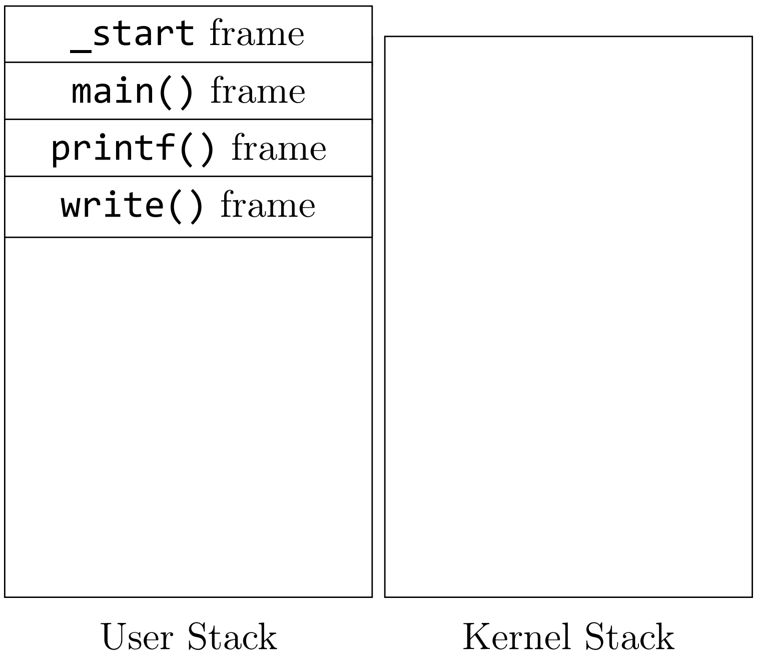
Restores all CPU state from the trapframe



Context Switch: Returning to User Mode

`write()` decodes `v0` and `a3` and updates `errno`

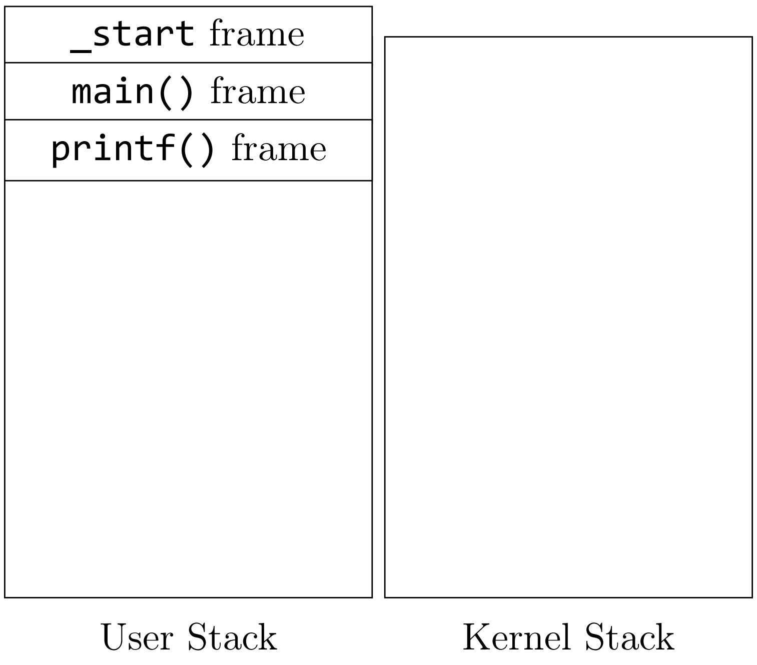
errno is where error codes are stored in POSIX



Context Switch: Returning to User Mode

errno is where error codes are stored in POSIX

`printf()` gets return value, if -1 then see `errno`



Outline

1. Process's view of the world
2. Kernel view of processes
3. User view of processes

System Call Interface

System Calls: Application programmer interface (API) that programmers use to interact with the operating system.

Processes invoke system calls

Examples: `fork()`, `waitpid()`, `open()`, `close()`, ...

System call interface can have complex calls

- `sysctl()` Exposes operating system configuration
- `ioctl()` Controlling devices

Need a mechanism to safely enter and exit the kernel

- Applications don't call kernel functions directly!
- Remember: kernels provide protection

Creating processes

`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

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

`minish.c` (simplified)

Parent Process (PID 5)

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

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

minish.c (simplified)

Parent Process (PID 5)

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

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

Child Process (PID 6)

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

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

minish.c (simplified)

Parent Process (PID 5)

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

    /* ... main loop: */
    for (;;) {
      parse_input(&av, stdin);
      switch (pid = fork()) {
      case -1:
        perror("fork"); break;
      case 0:
        doexec();
      default: //    After Fork pid   6
        waitpid(pid, NULL, 0); break;
      }
    }
```

Child Process (PID 6)

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

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

`minish.c` (simplified)

Parent Process (PID 5)

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

    /* ... main loop: */
    for (;;) {
      parse_input(&av, stdin);
      switch (pid = fork()) {
      case -1:
        perror("fork"); break;
      case 0:
        doexec();
      default: //    After Fork  pid   6
        waitpid(pid, NULL, 0); break;
      }
    }
```

Child Process (PID 6)

```
pid_t pid; char **av;
void doexec() {
  execvp(av[0], av); //    After Fork
  perror(av[0]); // Never executes!
  exit(1);
}

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

`minish.c` (simplified)

Parent Process (PID 5)

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

    /* ... main loop: */
    for (;;) {
      parse_input(&av, stdin);
      switch (pid = fork()) {
      case -1:
        perror("fork"); break;
      case 0:
        doexec();
      default: //    After Fork  pid   6
        waitpid(pid, NULL, 0); break;
      }
    }
```

Child Process (PID 6)

Replaced by the new program

```
int
main(int argc, const char *argv[])
{
  //    Starts here!
  ...
  exit(0);
}
```

`minish.c` (simplified)

Parent Process (PID 5)

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

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

Child Process (PID 6)

Replaced by the new program

```
int
main(int argc, const char *argv[])
{

  ...
  exit(0); //    Wake up waitpid
}
```

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` $= 0$
- Sets file descriptor non-inheritable by new program

Example: `redirsh.c`

- Loop that reads a command and executes it
- Recognizes input, output redirection

redirsh.c

```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]`
- Writes to `fds[1]` will be 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 ...`

Why fork?

Most calls to `fork` followed by `execve`

Could also combine into one spawn system call

Occasionally useful to fork one process

- Pre-forked Webservers for parallelism
- Creates one process per core to serve clients
- Lots of uses: Nginx, PostgreSQL, etc.

Real win is simplicity of interface

- Tons of things you might want to do to child:
  Manipulate file descriptors, environment, resource limits, etc.
- Yet `fork` requires no arguments at all

Spawning process w/o fork

Without fork, require tons of different options

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
);
```