EECS-343 Operating Systems
Lecture 2:
Processes & System Calls

Steve Tarzia

Spring 2019

Northwestern

Covers Chapter 4 and 5 in Three Easy Pieces

Announcements

- Midterm is on Thursday, May 2nd.
- First two parts of project 1 are due on Monday.
- Project part 2 (the big part) was posted, due the following Monday.
- TA and Peer Mentor office hours will be in the Wilkinson Lab (Tech M338)
- Another open OS book that will help you greatly:
  - "xv6: a simple, Unix-like teaching operating system" by Cox et al.
  - Google "xv6 book rev11"
- Don't forget to read the book! This lecture covers chapters 4-5.

Operating systems roles

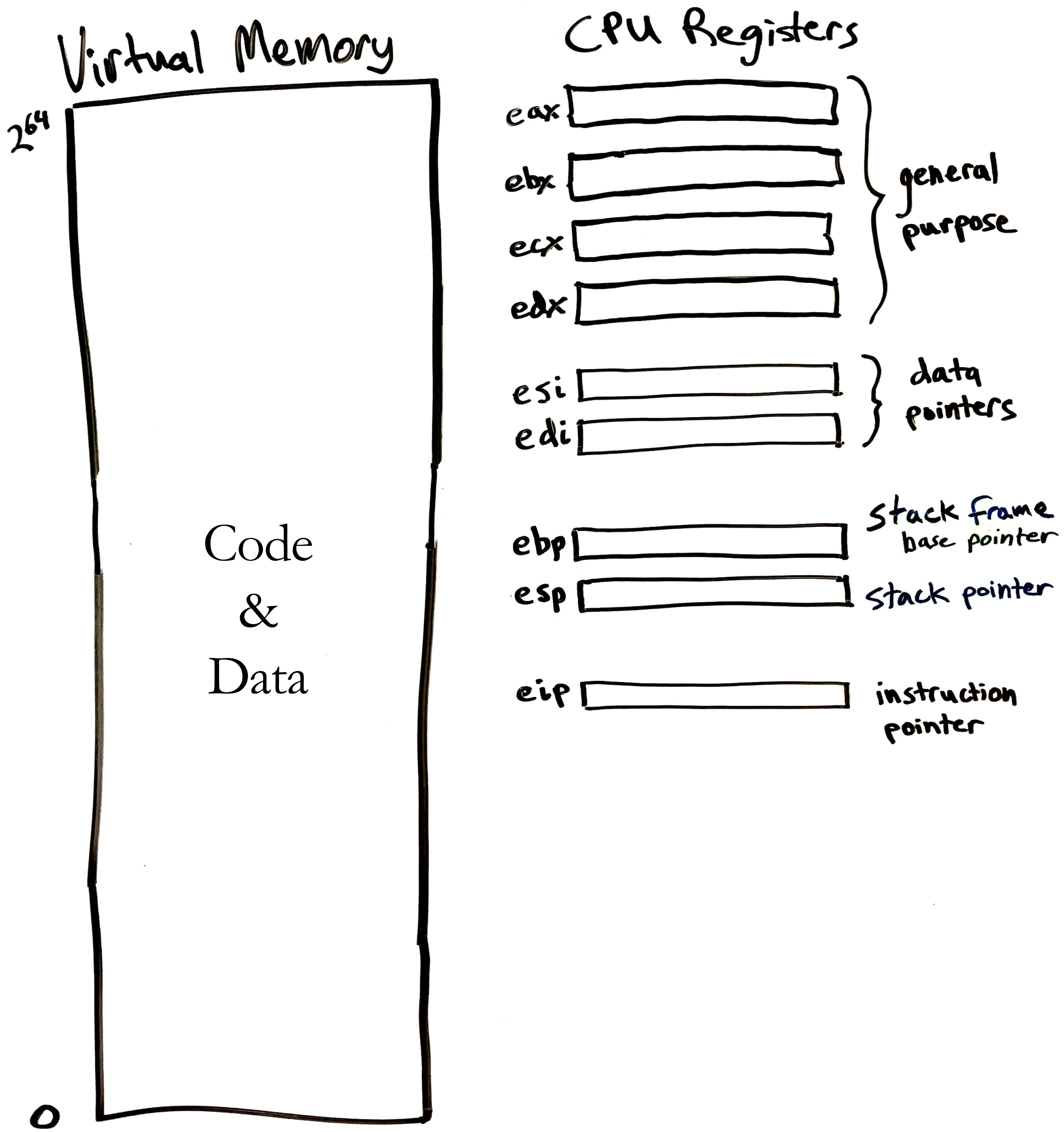
- A *user interface* for humans to run programs
- A *resource manager* allowing multiple programs to share one set of hardware.
- A *programming interface* (API) for programs to access the hardware and other services.

In this lecture we'll start talking about how OS provides programs with the illusion that they have the entire CPU and Memory to themselves.

Operating systems run ***processes***

- A **process** is a program in execution
  - On Linux/Mac run "ps" or "top" to see the active processes
  - On Windows use the Task Manager
- OS assigns each new process a numeric **process id**.
- Each process has it own private view of the computer:
  - **CPU register** values, including
    - General purpose registers (eax, ebx, ecx, edx), index registers (edi, esi)
    - Stack pointer (esp, ebp), used to locate local variables, function return address, etc.
    - Instruction Pointer, indicating the next instruction that will execute
  - Virtual memory **address space**
    - We will explore virtual memory in detail in a couple of weeks.

A program's view of the computer



- Machine code (after compilation) is a sequence of simple instructions understood by the CPU circuitry.
  - Arithmetic, copy to/from memory, and conditional jumps
- Operates on a few *fast* registers, and a large block of memory.
  - You may have heard of CPU caches. There are optional and *hidden* from the program.
- Values stored in registers and in memory represent a program's state
  (We're ignoring the OS for a moment)
- Each program has its own view like this. The OS and CPU together create this *illusion*.

Let's think at the assembly level for a moment

C Language:

```
void function1() {
  int A = 10;
  A += 66;
}
```

→

x86 assembly:

```
function1:
     pushl %ebp #
     movl %esp, %ebp #,
     subl $4, %esp #,
     movl $10, -4(%ebp) #, A
     leal -4(%ebp), %eax #,
     addl $66, (%eax) #, A
     movl %ebp, %esp
     popl %ebp
     ret
```

Example is from http://www.hep.wisc.edu/~pinghc/x86AssmTutorial.htm

"Limited direct execution"

- This is how the OS supports multi-tasking (concurrent execution)
- (In this class we usually assume a machine has one CPU core)
- Let a process run for a while with exclusive use of the CPU
  - Can use all the CPU registers
- Eventually, the OS pauses that process to let another process run
  - Then *that* process will use CPU for a while …
- This is a **context switch:** the OS/kernel has changed the active process.

Multitasking

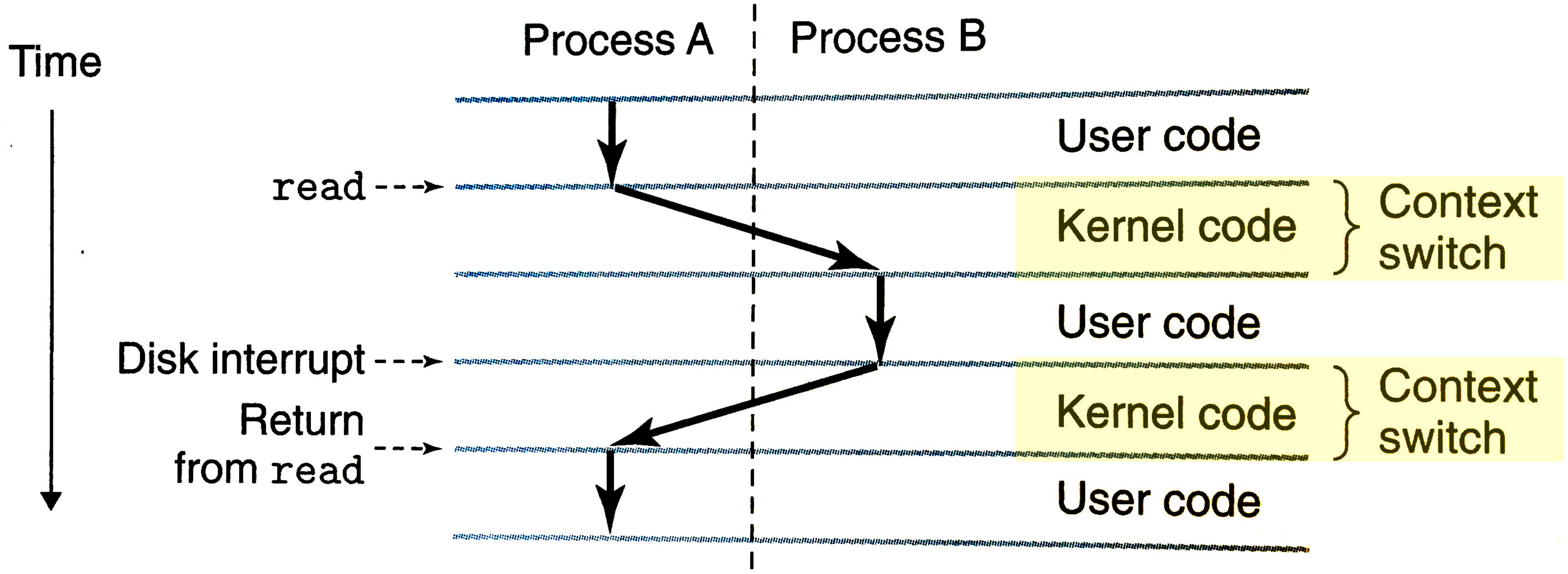
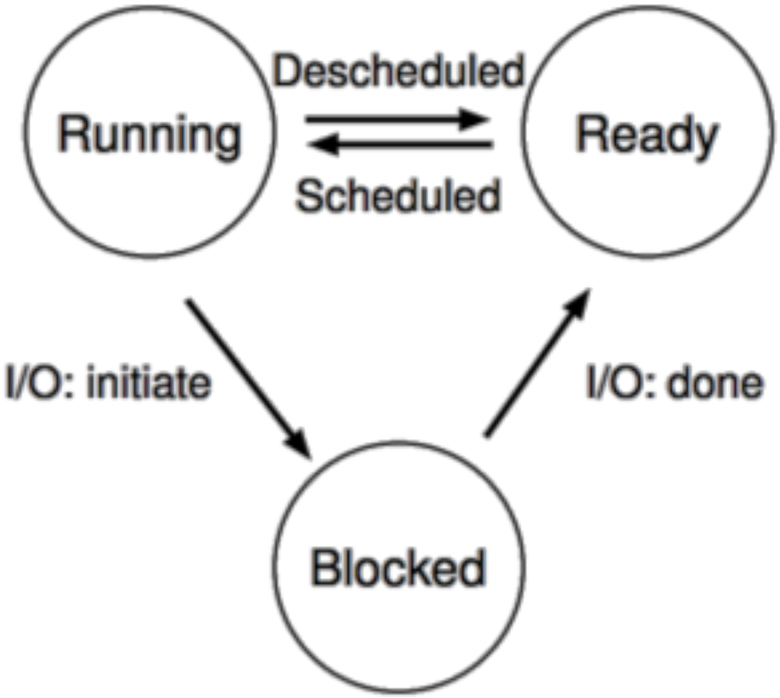


Diagram from Bryant & O'Hallaron book

Processes don't run all the time

The three basic process states:



- OS ***schedules*** processes
  - Decides which of many competing processes to run.
- A ***blocked*** process is not ready to run.
- I/O means input/output – anything other than computing.
  - For example, reading/writing disk, sending network packet, waiting for keystroke, updating display.
  - While waiting for results, the process often cannot do anything, so it **blocks**, telling the OS to let someone else run.

Process execution is *limited*

- **User processes** execute with CPU in "**user** mode"
  - Can only execute basic arithmetic, branching, and memory read/write instructions (within a specific range/segment).
  - CPU will not execute certain privileged instructions when in user mode..
- **OS kernel** runs with the CPU in "**privileged**/kernel mode"
  - Allows all instructions, including:
    - Changing registers that control which memory is accessible
    - Performing I/O, switching CPU mode.
- Early CPUs lacked multiple modes, so could not support a real OS.
  - For example, the first IBM PC had an Intel 8088 CPU lacking this feature, so PC DOS was a very limited OS.
  - Intel 386 processor in 1985 enabled a true OS for PCs (OS/2 and Windows).

Things a program cannot do itself

- Print "hello world"
  - *because the display is a shared resource.*
- Download a web page
  - *because the network card is a shared resource.*
- Save or read a file
  - *because the filesystem is a shared resource and the OS wants to check file permissions first.*
- Launch another program
  - *because processes are managed by the OS*
- Send data to another program
  - *because each program runs in isolation, one at a time*

Break time

"It must be you. The computer, it so happens, is user-friendly."

Interrupts

A way for the CPU to be, well, *interrupted*.

- CPU switches to privileged mode (kernel activates)
  - Now any instruction can be executed, including privileged ones.
- Execution jumps to a predefined location
  - (specified in the CPU's interrupt vector table)
  - This ensures that the kernel code starts running.
  - Interrupts are the only way the kernel is activated (after boot-up)
- Used to support asynchronous I/O
  - Lets a hardware device tell the CPU that some data is ready
  - Remember that a disk operation is millions of times slower than an *add*.
- CPU has an electrical pin for hardware interrupts.
- There is also an instruction for *software* interrupts.

Interrupts numbers in xv6 (traps.h)

```c
// Processor-defined:
#define T_DIVIDE         0      // divide error
#define T_DEBUG          1      // debug exception
#define T_NMI            2      // non-maskable interrupt
#define T_BRKPT          3      // breakpoint
#define T_OFLOW          4      // overflow
#define T_BOUND          5      // bounds check
#define T_ILLOP          6      // illegal opcode
#define T_DEVICE         7      // device not available
#define T_DBLFLT         8      // double fault
// #define T_COPROC      9      // reserved (not used since 486)
#define T_TSS           10      // invalid task switch segment
#define T_SEGNP         11      // segment not present
#define T_STACK         12      // stack exception
#define T_GPFLT         13      // general protection fault
#define T_PGFLT         14      // page fault
// #define T_RES        15      // reserved
#define T_FPERR         16      // floating point error
#define T_ALIGN         17      // aligment check
#define T_MCHK          18      // machine check
#define T_SIMDERR       19      // SIMD floating point error
```

CPU *exceptions* trigger interrupts, eg.:

- arithmetic overflow
- invalid memory access (general protection fault)

```c
// These are arbitrarily chosen, but with care not to overlap
// processor defined exceptions or interrupt vectors.
#define T_SYSCALL       64      // system call
#define T_DEFAULT      500      // catchall

#define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ

#define IRQ_TIMER        0
#define IRQ_KBD          1
#define IRQ_COM1         4
#define IRQ_IDE         14
#define IRQ_ERROR       19
#define IRQ_SPURIOUS    31
```

External hardware:

- Keyboard
- IDE disk

System Calls (syscalls)

…are the way that processes ask the OS to do things for them.

- Run a **Software Interrupt** (a.k.a. **trap**) instruction (`int` on x86)
- Syscall **number** and parameters are loaded into pre-defined registers
- Kernel takes over during the interrupt handler routine

- A regular function call into a library would be insufficient because it would run in the same process, in user mode.

Syscalls in xv6

- `user.h` defines function prototypes for syscalls:
- User processes can call these functions in C code
- But these are not regular functions!
  - They're just wrappers that trigger software interrupts.

```c
// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(char*, int);
int mknod(char*, short, short);
int unlink(char*);
int fstat(int fd, struct stat*);
int link(char*, char*);
int mkdir(char*);
int chdir(char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
```

Implementation of syscall user functions is in assembly

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
```

- `usys.S`
- There's some funky C-preprocessor syntax here.
- Will generate this code for `kill(int)`:

```
 .globl kill
 kill: movl $SYS_kill, %eax
       int $T_SYSCALL
       ret
```

- ".globl" makes the symbol visible to the linker, so it's like writing a C function.

Syscall numbers are defined in `syscall.h`

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_write   5
#define SYS_read    6
#define SYS_close   7
#define SYS_kill    8
#define SYS_exec    9
#define SYS_open   10
#define SYS_mknod  11
#define SYS_unlink 12
#define SYS_fstat  13
#define SYS_link   14
#define SYS_mkdir  15
#define SYS_chdir  16
#define SYS_dup    17
#define SYS_getpid 18
#define SYS_sbrk   19
#define SYS_sleep  20
#define SYS_uptime 21
```

Syscall table is defined in `syscall.c`

```c
// array of function pointers to handlers for all the syscalls
static int (*syscalls[])(void) = {
[SYS_chdir]   sys_chdir,
[SYS_close]   sys_close,
[SYS_dup]     sys_dup,
[SYS_exec]    sys_exec,
[SYS_exit]    sys_exit,
[SYS_fork]    sys_fork,
[SYS_fstat]   sys_fstat,
[SYS_getpid]  sys_getpid,
[SYS_kill]    sys_kill,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_mknod]   sys_mknod,
[SYS_open]    sys_open,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_unlink]  sys_unlink,
[SYS_wait]    sys_wait,
[SYS_write]   sys_write,
[SYS_uptime]  sys_uptime,
};
```

- There is one software interrupt handler function in the kernel, but it looks at the `@eax` register value to determine which of many syscalls was intended.
- This table tells the kernel which kernel function to call (sys_*) for each numbered syscall (SYS_*)

How syscall is handled in xv6

This C code in a user program:

```
kill(43);
```

Will compile to something like:

```
       push 43
       call kill

       ...
kill:  movl 8, %eax
       int 64
       ret
```

$SYS_kill = 8 is the syscall number for "kill"

$T_SYSCALL = 64 is the interrupt number chosen by xv6 for syscalls

The `int`(errupt) command switches control to the OS

Having just received an interrupt, the CPU will:

- Switch to privileged (kernel) mode
- Get the interrupt handler address by checking the interrupt vector table (in this case using the 64th entry)
- Check the `%eax` register for the syscall number (8 in this case)
- Call the appropriate syscall handler function
  - In this case, the 8th handler gives `sys_kill()`
- The kernel function `sys_kill()` gets the parameter left by the user process on the stack ("43") and handles it accordingly.
- When the syscall handler is done, we switch back to user mode and resume execution of the user process (using `iret` instruction).

Interrupts trigger *context switches*

- How are context switches implemented?
- Somehow we have to move processes on and off and on the CPU.

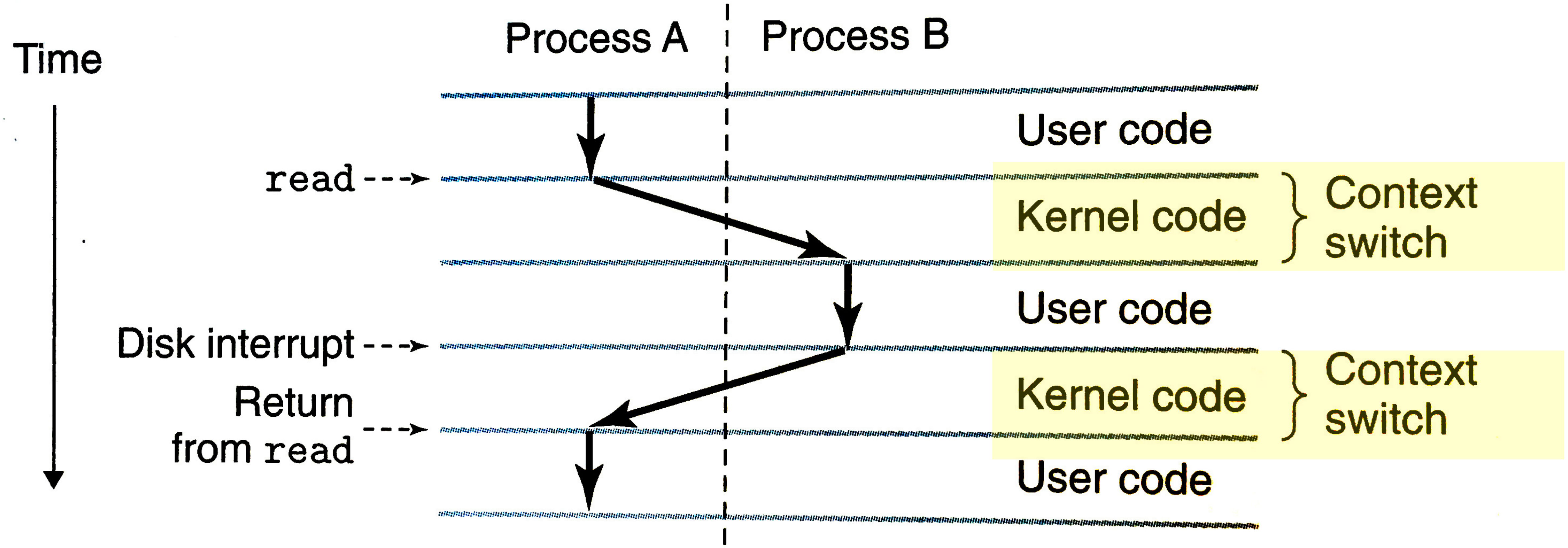


Diagram from Bryant & O'Hallaron book

Inactive process state

- OS has a process list in kernel memory to store the CPU state of processes that are not currently running.
- Context switches read and write this process state.
- In xv6's proc.h:

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  volatile int pid;            // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

…and in x86.h:

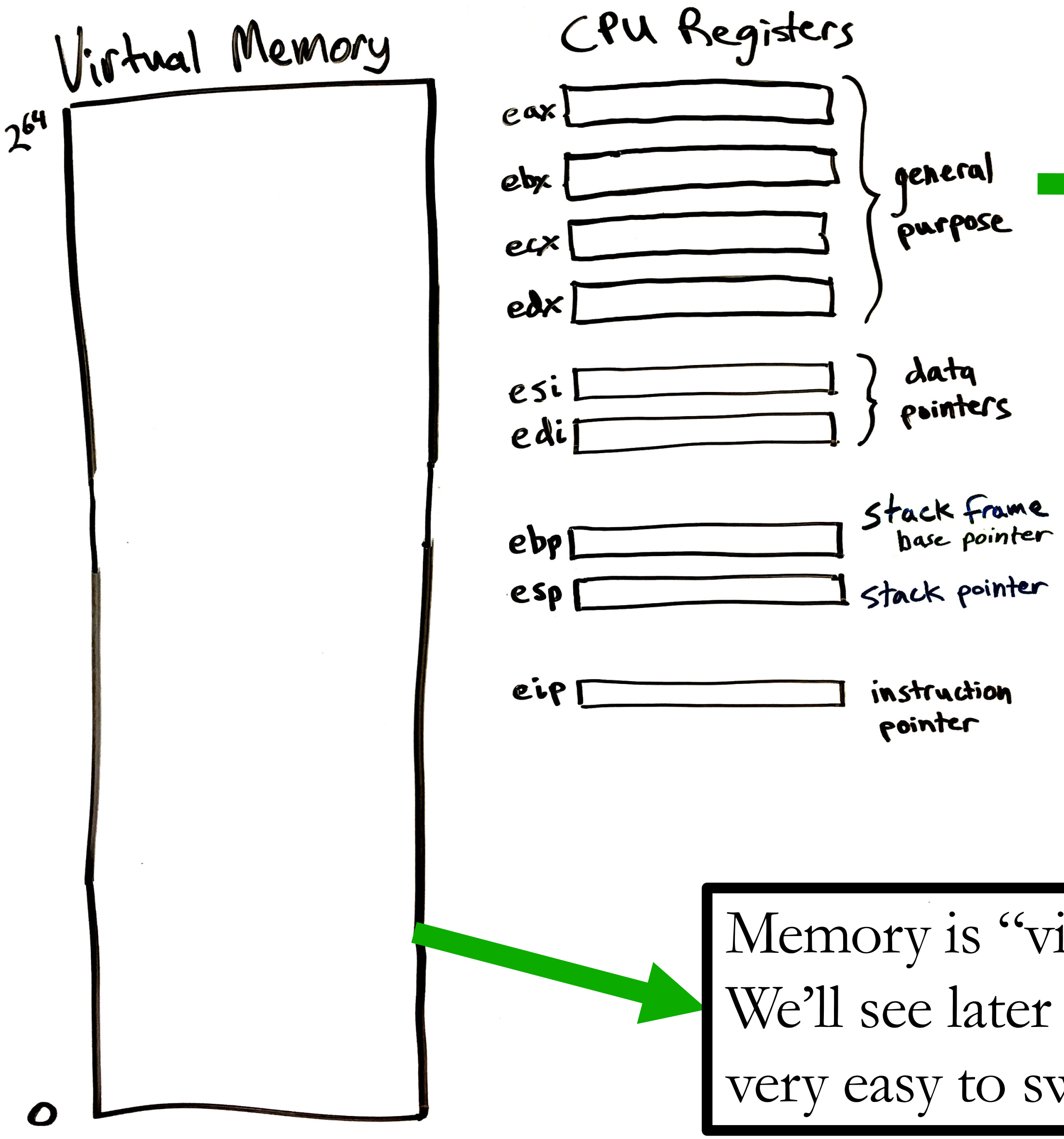
```c
// Layout of the trap frame built on the stack by the
// hardware and by trapasm.S, and passed to trap().
struct trapframe {
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort gs;
  ushort padding1;
  ushort fs;
  ushort padding2;
  ushort es;
  ushort padding3;
  ushort ds;
  ushort padding4;
  uint trapno;
```

```c
  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding5;
  uint eflags;

  // below here only when crossing rings, such as from user to kernel
  uint esp;
  ushort ss;
  ushort padding6;
};
```

CPU’s state is switched during context switch



Stored on the top of the process’ stack

```c
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

Memory is “virtual.” We’ll see later that it’s very easy to switch.

Stop and think

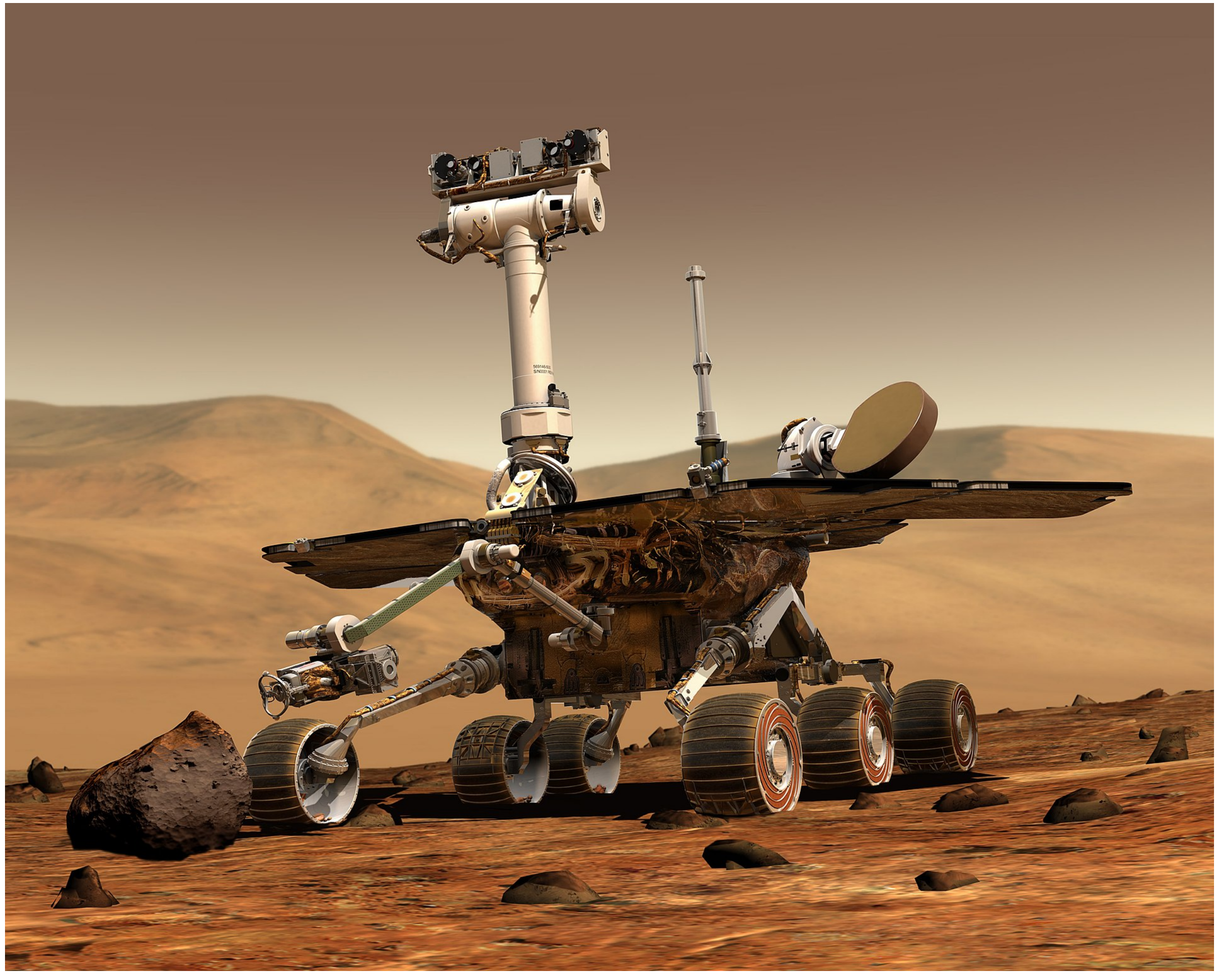
- So far, we've seen the kernel's mechanism for switching processes and these are called **context switches.**
- We've seen than the kernel gets control after the CPU gets an interrupt
  - **Hardware interrupts** can be triggered by I/O devices
  - **Software interrupts** are created by programs making **system calls** to ask the OS to do something that the user program is not privileged to do.
- After an interrupt, the kernel can choose to do a context switch, and thus schedule another process.
- But what's to prevent a program from hogging the CPU forever?
- What is the program never does any I/O or systems calls?
- No interrupts will happen and the kernel will never run! … right???

Solution: programmable **timer interrupt**

- The programmable **timer** is another hardware feature for the OS.
- Timer is a hardware device that can be programmed to generate an interrupt after a certain amount of time.
  - Perhaps after 1 to 10 milliseconds
- Before context switching to a user process, the kernel sets the timer.
- Timer interrupt ensures that the kernel gets an opportunity to act.
  - Recall that kernel only runs in response to interrupts
  - Timer is necessary to implement process scheduling policies

    (Give another process a chance to run)
- Prevents a user process from getting stuck in an infinite loop

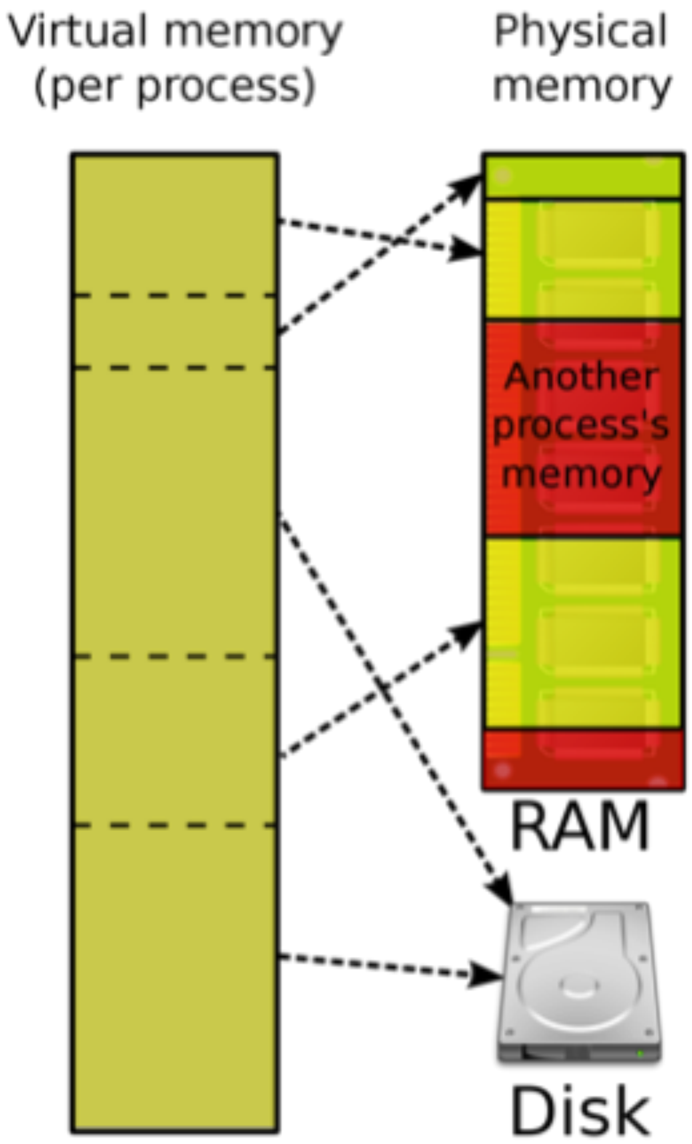
Aside: Watchdog Timer



- PC's timer generates a periodic interrupt to let the OS check on things.
- Similarly, embedded systems often have a *watchdog timer*:
  - A running countdown to *reboot*
  - Software is supposed to reset it periodically.
  - If software/hardware is hung, watchdog timer will expire and reboot the CPU.
- Left: Mars Exploration Rover

What about memory?

- We have shown how each process gets it’s own copy of CPU registers.
  - However registers only store a little data
- Each process also has its own *virtual memory*
- Virtual memory gives the illusions that:
  - Each process has exclusive use of the memory
  - Processes have “infinite” memory available
- OS and CPU handle virtual memory mapping using *page tables.*
- This is a complex topic that we will discuss starting in Lecture 6 or 7.



Recap

- **Process** is a program in execution
- **Limited direct execution** is a strategy whereby a process usually operates as if it has full use of the CPU & memory.
- CPUs have user and kernel **modes** to prevent user processes from running privileged instructions, thus *limiting* execution.
- **Interrupts** are events that cause the kernel to run
- **System Calls** (or traps) are software interrupts called by a user program to ask the OS to do something on its behalf.
- **Timer Interrupt** ensures that the kernel eventually runs.
- *Next time:* process creation and process memory layout.