Implementing Time Sharing

Two central challenges –

- performance
  - seek to avoid excessive overhead
- control
  - a process should not be allowed to take over and hog system resources
  - a process should not be allowed to access information belonging to other processes, other users, or the system

Direct Execution

- *direct execution* – run the program directly on the CPU
  - sometimes the CPU is running OS code, sometimes it is running user process code

when a program is run

OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute **call** main()	
	Run main()
	Execute **return** from main
Free memory of process	
Remove from process list	

  - the overhead is minimal – just the setup and cleanup for the process
  - but…

Limited Direct Execution

- issue #1: restricted operations
  - rationale: there are some things a user process should not be able to access freely – disk, memory, CPU
    - e.g. can't enforce file permissions without a mechanism to check a disk I/O request before it is allowed to proceed
  - mechanism: *user mode* vs *kernel mode*
    - a process running in user mode is not allowed to do *privileged operations* like issue disk I/O requests
    - OS code runs in kernel mode and can do anything

User vs Kernel Mode

- the CPU switches to kernel mode when privileged operations are needed
  - user process needs OS services
    - user process makes a *system call* in order to be allowed to do privileged operations in a controlled way
  - an external device needs attention
    - device raises an interrupt
  - something has gone wrong during program execution
- this is managed through *traps*
  - the *trap* instruction jumps into kernel code and sets the privilege level to kernel mode
  - the *return-from-trap* instruction sets the privilege level to user mode and returns back to the caller
- observation: the OS is not a separate process
  - OS instructions run when another process is in kernel mode

System Calls

- system calls appear to a program just like regular function calls
  - but the implementation in the C library contains special assembly code to set up, issue, and return from a trap

System Calls

- for a regular function call –
  - allocate memory on the stack for function arguments, local variables, etc
  - push the return address, PC jumps to function code
  - push register context
  - execute function code
  - when returning from the function, pop the return address and register context
- systems calls also –
  - push register context
  - save old PC, change PC to OS system call handler
  - trap! to run system call
  - restore context back to user code

System Calls

- for a regular function call –
  - allocate memory on the stack for function arguments, local variables, etc
  - push the return address, PC jumps to function code
  - push register context
  - **execute function code** → instructions in function body run in user mode, nbd
  - when returning from the function, pop the return address and register context
- systems calls also –
  - push register context
  - save old PC, change PC to OS system call handler
  - trap! to run system call
  - restore context back to user code → instructions run in kernel mode – have to be careful!

System Calls

- for a regular function call –
  - allocate memory on the stack for function arguments, local variables, etc
  - push the return address, PC jumps to function code
  - push register context
  - execute function code
  - when returning from the function, pop the return address and register context

  } on user stack, only accessed in user mode, nbd
- systems calls also –
  - push register context
  - save old PC, change PC to OS system call handler
  - trap! to run system call
  - restore context back to user code

  } don't want to use user stack, what if they've set up something malicious?

Protected Control Transfer

- use a separate *kernel stack* for running kernel code
  - kept in OS memory not accessible in user mode
- locations to jump to in order to handle system calls are defined in the *trap table*
  - set up in hardware on boot
    - start in kernel mode on boot to do key OS setup stuff
  - trap instruction is invoked with an argument indicating the trap number – address to jump to is looked up
- system must also check parameters to system calls
  - e.g. `write` is passed the address of the buffer whose contents are to be written – must be an address within the process' address space (and not the kernel's!)

Limited Direct Execution

OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list		
Allocate memory for program		
Load program into memory		
Setup user stack with argv		
Fill kernel stack with reg/PC		
return-from-trap		
	restore regs	
	(from kernel stack)	
	move to user mode	
	jump to main	
		Run main()
		...
		Call system call
		trap into OS
	save regs	
	(to kernel stack)	
	move to kernel mode	
	jump to trap handler	
Handle trap		
Do work of syscall		
return-from-trap		
	restore regs	
	(from kernel stack)	
	move to user mode	
	jump to PC after trap	
		...
		return from main
		trap (via `exit()`)
Free memory of process		
Remove from process list		

Limited Direct Execution

- issue #2: switching between processes
  - to implement time sharing, the OS may need to regain control of the CPU outside of a system call
  - *cooperative approach* – wait for system call
    - system calls typically happen fairly often
    - includes a `yield` system call for a process to voluntarily give up the CPU in the absence of other system calls
  - non-cooperative approach – *timer interrupt*
    - hardware generates an interrupt at particular intervals
    - when the interrupt occurs, control switches to a pre-configured *interrupt handler*

Context Switches

- interrupts and system calls transfer control to the kernel
- when returning from kernel mode –
  - may *need* to return to a different process
    - e.g. current process has exited
    - e.g. current process must be terminated (due to illegal instruction or memory access)
    - e.g. current process has made a blocking system call
  - may *want* to return to a different process
    - for time sharing – current process has run for too long

Context Switches

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember addresses of... syscall handler timer handler	
start interrupt timer		
	start timer interrupt CPU in X ms	
OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A ...
	timer interrupt save regs(A) → k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call `switch()` routine save regs(A) → proc_t(A) restore regs(B) ← proc_t(B) switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) ← k-stack(B) move to user mode jump to B's PC	
		Process B ...

two sets of register save/restore steps

registers are saved to the kernel stack by the hardware when timer interrupt occurs, as with trap – *user context* captures where execution stopped in user mode

restored by return-from-trap

registers are saved to the process structure when the context switch is initiated – *kernel context* captures where the execution stopped in kernel mode

restored by the context switch

Context Switches

```
# void swtch(struct context *old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
  # Save old registers
  movl 4(%esp), %eax  # put old ptr into eax
  popl 0(%eax)        # save the old IP
  movl %esp, 4(%eax)  # and stack
  movl %ebx, 8(%eax)  # and other registers
  movl %ecx, 12(%eax)
  movl %edx, 16(%eax)
  movl %esi, 20(%eax)
  movl %edi, 24(%eax)
  movl %ebp, 28(%eax)

  # Load new registers
  movl 4(%esp), %eax  # put new ptr into eax
  movl 28(%eax), %ebp # restore other registers
  movl 24(%eax), %edi
  movl 20(%eax), %esi
  movl 16(%eax), %edx
  movl 12(%eax), %ecx
  movl 8(%eax), %ebx
  movl 4(%eax), %esp  # stack is switched here
  pushl 0(%eax)       # return addr put in place
  ret                 # finally return into new ctxt
```

general-purpose registers

- xv6 is a modern reimplementation of UNIX V6 (originally released in 1975 for the PDP-11) used as a teaching OS
- registers – e for extended (32-bit x86)
  - `eax` – accumulator
  - `ebx` – base (base pointer for memory access)
  - `ecx` – counter (loop counter, shifts)
  - `edx` – data
  - `esi`, `edi` – source, destination index for string ops
  - `esp` – stack pointer (top of program stack)
  - `ebp` – frame pointer (base of current stack frame)

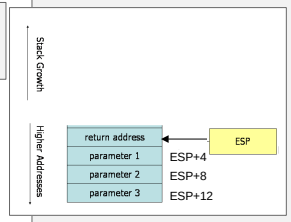
Context Switches

```
# void swtch(struct context *old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
  # Save old registers
  movl 4(%esp), %eax  # put old ptr into eax
  popl 0(%eax)        # save the old IP
  movl %esp, 4(%eax)  # and stack
  movl %ebx, 8(%eax)  # and other registers
  movl %ecx, 12(%eax)
  movl %edx, 16(%eax)
  movl %esi, 20(%eax)
  movl %edi, 24(%eax)
  movl %ebp, 28(%eax)

  # Load new registers
  movl 4(%esp), %eax  # put new ptr into eax
  movl 28(%eax), %ebp # restore other registers
  movl 24(%eax), %edi
  movl 20(%eax), %esi
  movl 16(%eax), %edx
  movl 12(%eax), %ecx
  movl 8(%eax), %ebx
  movl 4(%eax), %esp  # stack is switched here
  pushl 0(%eax)       # return addr put in place
  ret                 # finally return into new ctxt
```

CALL instruction sets up:
return address at `0(%esp)`
parameters `old`, `new` at `4(%exp)`, `8(%exp)`

`popl 0(%eax)` pops the top of the stack (return address), adding 4 to `%esp`...so `4(%esp)` now refers to new



adapted from https://www.cs.virginia.edu/~evans/cs216/guides/x86.html

Concurrency Concerns

- issue #3 – what if an interrupt occurs during a system call or when another interrupt is being handled?
  - option: don't let that happen – disable interrupts when an interrupt is being handled
    - problem: can lose interrupts if handling is disable for too long
  - option: use locking schemes to protect concurrent access to internal data structures
    - problem: can be complex and hard to debug