ITP 30002 Operating System

Limited Direct Execution and Context Switching

OSTEP Chapter 6

Shin Hong

Mechanisms for Process

- time-sharing
  - run one process for a while, and then switch to another one
  - issue
    - managing control
    - obtaining performance
- limited direct execution
  - run an application program directly on the CPU with some restriction
  - restriction
    - restricted memory accesses (H/W manipulation, resource allocation, etc.)
    - restricted instruction

Dual Mode Operation: User Mode & Kernel Mode

- Most contemporary processors provide dual mode operation

- User mode
    - for application program execution
    - restriction is enforced
        - a trap occurs when a process executes a restricted instruction under user mode

- Kernel model
    - for kernel execution
    - all privileged operations can be executed

- What if an application program needs to execute privileged operations?

System Call

- A way of an application program to call a kernel to get a system service
    - not possible to do this with the procedure call mechanism

- to execute a system call, a program must execute a trap instruction
  - the operation of a trap instruction
    - change the mode into the kernel mode
    - store the current PC at a kernel stack
    - jump to a predefined program location for handling the trap
      - trap table
      - trap handler
  - each system call is identified by its unique number (i.e. system-call number)

Example of System Call Workflow

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		

Switching Between Processes

- How can OS regain control of the CPU when it is given to an application program?
- Natural chance: blocked operation
- Periodic scheduling
  - Cooperative approach
    - blocked operation
  - Preemptive scheduling approach
    - exploit a timer interrupt and its interrupt handler
    - requires HW support

Preemptive Scheduling

- scheduler: a kernel module that determines which process to dispatch at a chance (e.g., timer interrupt)

- context switch: a kernel module that is executed to switch processes running on a CPU
  - registered as a timer interrupt handler
  - steps
    - store the process status of a currently-running process to memory
      - CPU states
    - find the stored status of the next process from the memory
    - restore the stored status at the CPU
    - return the control back to the application program

Example

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 ...

The xv6 Context Switch Code

```
# 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
```