Beginnings of Operating Systems

history of operating systems
1. Goal: To look at the history of operating systems and see why they developed as they did; to see the basic functions and designs of operating systems
2. First Generation (vacuum tubes)
   a. hardware only, open shop
3. Second Generation (transistors)
   a. separation of programmers and operators
   b. batching, satellite systems, buffering
   c. device independence, resident loaders, first operating system, JCL
   d. Atlas system: extracodes, interrupts, virtual memory
4. Third Generation (integrated circuits)
   a. spooling, job scheduling, multiprogramming
   b. protection, traps, fence registers, privileges, system calls, time sharing
   c. virtual machines: levels of abstraction, THE system
   d. customer service, compatibility
5. Fourth Generation (VLSI)
   a. minicomputers: rise of the UNIX operating system
   b. microcomputers: workstations, personal computers, open operating systems
6. Networks and Distributed Systems
   a. networked vs. distributed operating systems

Operating Systems Functions
1. Goal: To look at common operating system functions, see what a good operating system needs to do, desirable features.
2. I/O Functions
   a. polling, interrupt-driven
   b. DMA, I/O controller
3. Process Functions
   a. creation
   b. deletion
   c. scheduling
   d. synchronization, communication
4. Memory Functions
   a. allocation
   b. deallocation
   c. management
5. Secondary Storage Functions
   a. bringing data in and out
   b. address translation
6. User Interface Functions
   a. command interpreter
   b. job control language
7. Desirable Features
   a. efficiency
   b. reliability
   c. maintainability
   d. smallness
**Organization of operating systems**

1. monolithic: processes are subroutines
2. kernel: operating system calls are subroutines
3. process hierarchy: virtualize lower level resources
4. object oriented: view system as collection of objects (processes, procedures, pages, devices, messages, *etc.*) and capabilities (pointers to objects, also containing rights).
5. client-server model: kernel just passes messages

**User Interface**

1. controls how the user interacts with the system
2. types of software
   a. kernel
   b. essential utilities
   c. optional utilities
3. Command Interpreter
   a. batch job control language
   b. interactive job control language
4. Invoking programs
   a. search path
   b. global and local environments
   c. user principle
   d. little languages

**The kernel**

1. Question: how do processes interact with the hardware of the computer? What does the program that mediates this interaction (kernel) look like?
2. The system kernel
   a. interface between machine hardware and the operating system
   b. first-level interrupt handler, dispatcher, interprocess communication primitives
3. Processes in the kernel
   a. process control blocks
   b. ready queue, device queues, other queues
   c. CPU/IO Burst Cycle
4. Dispatcher
MINIX Context Switch Routine

Introduction

This is the MINIX context switch routine; it is lines 6319–6345 of the MINIX operating system (see the text, p. 596). My comments follow the program.

The basic idea is to execute any pending interrupts. If there are any, the process receiving the interrupt becomes the current task; otherwise, the current task is the one already chosen. Load the segment descriptors for the task to be run, reset the interrupt vector so the next interrupt will cause the process state for the task to be saved properly, reset the registers, and return to the next task.

Program

```
06319 |*=================================================================================* |
06320 |*                              restart                                    * |
06321 |*=================================================================================* |
06322 _restart:                           |
06323
06324 ! Flush any held-up interrupts.          |
06325 ! This reenables interrupts, so the current interrupt handler may reenter.       |
06326 ! This does not matter, because the current handler is about to exit and no    |
06327 ! other handlers can reenter since flushing is done only when k_reenter == 0.   |
06328
06329 cmp (_held_head),0 ! do fast test to usually avoid function call       |
06330 jz over_call_unhold       |
06331 call _unhold ! this is rare so overhead acceptable         |
06332 over_call_unhold:               |
06333 mov esp,(_proc_ptr) ! will assume P_STACKBASE == 0            |
06334 lldt P_LDT_SEL(esp) ! enable segment descriptors for task       |
06335 lea eax,P_STACKTOP(esp) ! arrange for next interrupt         |
06336 mov (_tss+TSS3_S_SPO),eax ! to save state in process table     |
06337 restart1:                       |
06338     decb (_k_reenter)           |
06339     o16 pop gs                  |
06340     o16 pop fs                  |
06341     o16 pop es                  |
06342     o16 pop ds                  |
06343     popad                       |
06344     add esp, 4 ! skip return adr |
06345     iretd ! continue process    |
```

Comments

06331 If MINIX takes this call, it enters a routine that pulls the first process off the interrupt queue and puts it into the ready position (see the routines over_call_unhold at 07397–07419 and interrupt at 06935–07003).

06332–06345 If the restoration of registers is interrupted, then the system will have to know which ones were changed and which ones were not. This is not possible, so interrupts are ignored.

06333–06337 These lines prepare interrupt addresses and task segments for the next task to run.

06338 The control is about to leave the kernel, _k_reenter is the number of times the kernel has reentered (invoked itself). If this variable is 1, the next task is not a kernel task. Otherwise, it is. While within the kernel, user-level interrupts are disabled.

06339–06343 These restore all the registers of the process to be run except the return address, which must be handled specially.

06344 Here the stack pointer is reset.

06345 This returns control to the next task. It takes the return address from the stack (which is what the previous instruction sets up) and puts it in the program counter.