Today's plan

- Minix context switch
- Minix interrupt handling
- seL4 interrupt handling
- Minix system call handling
- Minix kernel synchronization
- Minix process suspension and reactivation
- a.out and kernel executable formats
- booting

Minix context switch

- context switch on (hardware) interrupt or on system call (software interrupt)
- interrupt from user mode will reload stack pointer from a given location in memory (book, p. 168-169)
- Minix updates this location every time a new process is started so registers are automatically saved in the process descriptor, the process table entry for the currently running process
- some registers are saved automatically, and others are pushed by save, which also increments _k_reenter (p. 712)
- the result is in sigregs (p 658) format
  - newer version: trapframe/intrframe

Interrupts in Minix

- see comments in kernel/mpx386.s, p. 707
  - minix/kernel/arch/i386/mpx.S at https://github.com/Stichting-MINIX-Research-Foundation/minix/
- hardware interrupt causes execution of hwint00..hwint15
- these routines call save (p. 712, or SAVE_PROCESS_CTX in minix/kernel/arch/i386/sconst.h), which saves the register, sets up a kernel stack if needed (if _k_reenter >= 0), adds a return address that calls _restart, and returns
- these routines then call intr_handle (kernel/i8259.c, p. 735), which is written in C, with an argument to tell it which interrupt it is handling (or call irq_handle in minix/kernel/interrupt.c, then switch_to_user in minix/kernel/proc.c)
- once intr_handle returns, these routines disable the specific interrupt (should be re-enabled by the device driver), re-enable interrupts in general, and return
- the return goes to the address set up by save, which will either return to the kernel code that was interrupted, or start or restart a process, often not the one that was interrupted -- for example, a newly awakened task, specified by next_ptr in restart (p. 713)
  - in the current version, irq_handle transfers control to switch_to_user, in minix/kernel/proc.c, which eventually calls arch_finish_switch_to_user in minix/kernel/arch/i386/arch_system.c, which finally enables interrupts (line 512)
- this new task will typically be a device driver

Device-Specific Interrupt Handling and intr_handle

- intr_handle is given a list of hooks (function pointers), stored in the irq_handlers array, and calls them in turn
- irq_handlers are set up by calls to put_irq_handler which, except for the clock device, is called by do_irqctl in kernel/system/do_irqctl.c (not in book)
- do_irqctl calls generic_handler, which transforms interrupts into messages by calling lock_notify in kernel/proc.c
- generic_handler then conditionally re-enables the specific interrupt by returning a bit
  - in the current version, interrupts are re-enabled as long as none of the hooks returns false
- lock_notify calls mini_notify after "locking" (disabling interrupts) but only if we are not already re-entering (lock is declared on line 4861 of kernel/const.h, p. 692
- mini_notify either creates the message and makes the receiver ready to execute, or saves the message information (one bit) if the receiver is already active
- in any case, intr_handle never blocks -- if other kernel code is executing, it simply returns after recording that the interrupt was held back

seL4 Interrupt Handling

- from https://github.com/seL4/seL4
- assembly `handle_interrupt` in `arch/x86/64/traps.S` saves registers, loads the kernel stack, copies the irq and syscall arguments to the registers where C expects them, and calls
- `c_x64_handle_interrupt` in `arch/x86/64/c_traps.c`, which eventually calls
- `c_handle_interrupt` in `arch/x86/c_traps.c`, which locks the CPU and (in normal cases) calls `handleVMFaultEvent` or `handleInterruptEntry` or `handleUnknownSyscall`
- `handleInterruptEntry` in `api/syscall.c`, which calls
- `handleInterrupt` in `object/interrupt.c`, which calls `sendSignal` or `timerTick` or a few other special cases, and ends by calling a machine-dependent `ackInterrupt`
- the signal then wakes up the device driver that processes the interrupt

Minix System Calls

- hardware syscalls transfer control into the kernel
  - `sys_call` (p. 724) is called from the system call (soft interrupt) handler (p. 712) in a process similar to a hardware interrupt, but with interrupts enabled
  - its three arguments are a "call number" which includes a function (send, receive, or both: sendrec) and flags (non-blocking), the task with which to communicate, and a message
  - system calls from user processes can only go to one of the servers, and can only be sendrec (send, then receive)
  - send, receive, or both may block
- posix syscalls always transfer control to one of the servers

Synchronization of reentrant processes in (textbook) Minix

- `_k_reenter` incremented by `_s_call` before enabling interrupts
- `_k_reenter` checked by `save` (to decide which stack to use) and `lock_notify`, if less than zero, `lock_notify` disables interrupts (which if called from an interrupt, are already disabled)
- since interrupts are disabled whenever `sys_call` and the other functions in `kernel/proc.c` are called, these can modify global variables, especially the process queue, `rdy_head`, `rdy_tail`, and the current and next process, `curr_ptr` and `next_ptr`
- `sys_call` may block and queue its caller, but `sys_call` itself never blocks and never executes another process (until after it returns), so (and because interrupts are suspended) `sys_call` does not need to worry about reentrant calls

Process Blocking in Minix

- literally, a process blocks when it makes a system call or is interrupted, because the kernel is a separate thread with its own stack
- logically, a process blocks when it sends a message to a process that is not receiving, or when it receives a message and no sender is ready
- a process is awakened when the receiver of its message receives it, or, if blocked receiving a message, when a process sends it a message
- blocking a process means removing it from the appropriate queue (`dequeue`, p. 729), as well as setting the appropriate flags (`SENDING`, `RECEIVING`, or both) and, if sending, putting it on the receiver's queue (so the `ANY` receive semantics can be implemented in O(1) time)
- the process that is blocked at the system call is the one that will be reawakened, though perhaps after many other processes get to execute

Process Reawakening in Minix

- awakening a process means inserting it into the ready queue, perhaps after filling its receive buffer
- the semantics of send and receive are such that a process cannot receive until it is done sending
- to avoid deadlocks, `mini_send` will fail if the receiver is trying to send to us (`mini_send`, p 726, lines 7605-7610), perhaps transitively (A sending to B sending to C -- C is not allowed to send to A)

`a.out` format

- described in `include/a.out.h` (not in book)
- 2-byte magic number helps avoid inadvertent execution of text and other random files
- 1-byte flags field identifies different styles of executables, including dynamic linking or position-independent code (PIC)
- 1-byte CPU field identifies architecture on which it is meant to run`aout_mids.h`
- 1-byte header length (no longer present) allowed for header variability
- segment lengths for text, data, bss segments (below)
- entry point records the address to jump to when executing the file
- text segment contains executable code
- data segment contains initialized data, with initial values
- bss segment contains data initialized to zero, so only the size is recorded in the file, no actual data is stored

memory layout of Minix kernel

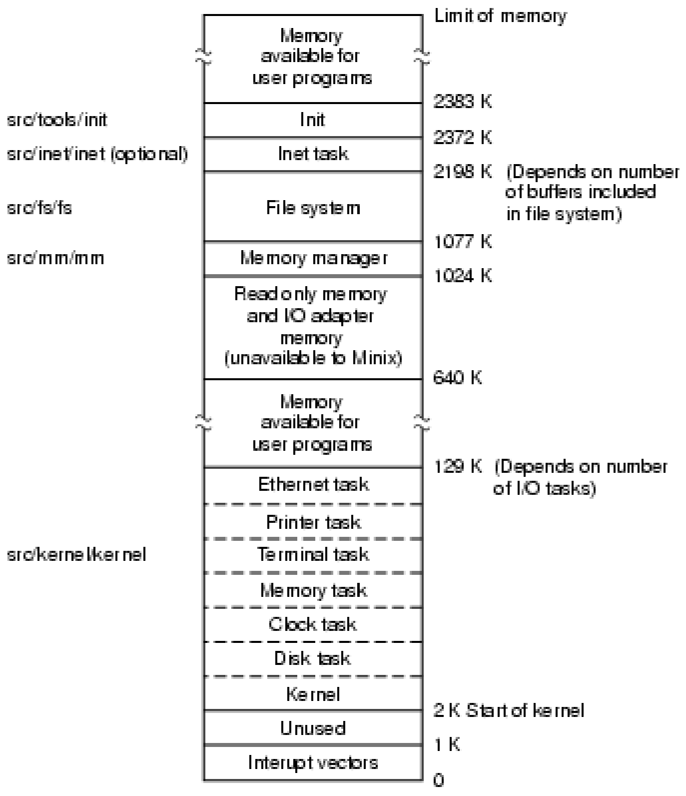

Figure 2-27. Memory layout after MINIX has been loaded from the disk into memory.

figure 2-31 on p. 129

kernel file format

- kernel on disk is simply a concatenation of:
  - one a.out for the kernel and the tasks and some drivers
  - one a.out for the memory/process manager (pm)
  - one a.out for the file system (fs)
  - one a.out for the inet server
  - one a.out for the restart (rs) process
  - one a.out for the init process
  - anything else the OS is configured for
- entry point was the label MINIX in `mpx386.s` (p. 709), which then called `cstart` (p. 716) and finally `main` (p. 718)
- in current code, the label MINIX is in head.s. That code calls `pre_init` and then `kmain` in `kernel/main.c`
- `kmain` sets up process table entries for the operating system processes, then calls `bsp_finish_booting` which then calls `switch_to_user`, which is what is called at the end of each system call and interrupt, and runs the scheduler

booting

- ROM built-in to the machine (the BIOS) loads the first sector (512 bytes) from the floppy disk and executes it
- or reads a floppy's worth of data from a CD drive and treats it as a floppy
- or reads the partition table from the hard disk, locates the active partition, loads the boot block (512 bytes) from the active partition, and executes it
- boot sector program is hardcoded with the sectors of a program called `boot` which does the actual initialization -- a different boot sector is written by `installboot` depending on where `boot` is on the disk
- `boot` understands the minix file system, and searches for a file named `/boot/image` or, in a `/boot/image/ directory`, the newest file, or whatever file is specified by the boot parameters
- `boot` copies this file to memory (at location 2K for Minix) and jumps to the entry point
- **diskless workstations** need enough networking in the ROM to request a kernel image from a server