Timers

Jiffies

- Each timer tick, a variable called `jiffies` is incremented
- It is thus (roughly) the number of HZ since system boot
- A 32-bit counter incremented at 1000 Hz wraps around in about 50 days
- We need 64 bits — but there's a problem

Potent and Evil Magic

- A 64-bit value cannot be accessed atomically on a 32-bit machine
- A spin-lock is used to synchronize access to `jiffies_64`; kernel routines call `get_jiffies_64()`
- But we don't want to have to increment two variables each tick
- Linker magic is used to make `jiffies` the low-order 32 bits of `jiffies_64`
- Ugly!

Time of Day

- The time of day is stored in `xtime`, which is a `struct timespec`
- It's incremented once per tick
- Again, a spin-lock is used to synchronize access to it
- The apparent tick rate can be adjusted slightly, via the `adjtimex()` system call

4 / 46

Kernel Timers

- Two types of timers use by kernel routines
- Dynamic timer — call some routine after a particular interval
- Delay loops — tight spin loops for very short delays
- User-mode interval timers are similar to kernel dynamic timers

5 / 46

Dynamic Timers

- Specify an interval, a subroutine to call, and a parameter to pass to that subroutine
- Parameter used to differentiate different instantiations of the same timer — if you have 4 Ethernet cards creating dynamic timers, the parameter is typically the address of the per-card data structure
- Timers can be cancelled; there is (as usual) a delicate synchronization dance on multiprocessors. See the text for details

6 / 46

Delay Functions

- Spin in a tight loop for a short time — microseconds or nanoseconds
- Nothing else can use that CPU during that time, except via interrupt
- Used only when the overhead of creating a dynamic timer is too great for a very short delay

7 / 46

System Calls

- `time()` and `gettimeofday()`
- `adjtimex()` — tweaks apparent clock rate (even the best crystals drift; see the *Remote Physical Device Fingerprinting* paper from my COMS E6184 class
- `setitimer()` and `alarm()` — interval timers for applications

8 / 46

The Test

9 / 46

Conditions

- Closed book, closed notes, calculator ok
- 75 minutes

10 / 46

Format

- Approximately 8 questions
- I'm not asking you to write programs
- Three types of questions
  - Explanations of certain concepts, somewhat above the pure memorization level
  - Carrying out tasks based on things discussed in class
  - Design questions (i.e., ones intended to make you think)

11 / 46

Material

- If it's in my slides or I said it in class, you're responsible for it
- There will be some material based more on Tanenbaum; there won't be much from Linux internals
- You're responsible for the assigned readings at about the level of class coverage.
- I'm not going to ask you to memorize the crazy synchronization algorithms, but if one shows up you should be prepared to explain it

12 / 46

Limits

- I can't quiz you on everything I've covered in a dozen lectures
- I can't review 15 hours of class time today
- I'm to some extent limited by the kinds of things it's feasible to ask on an exam

13 / 46

Review

14 / 46

What's an Operating System?

- Resource manger
- Multiplexor
- Fairness
- Protection

15 / 46

Pieces of an OS

- Kernel
- Scheduler
- Memory management
- File systems
- Device drivers

16 / 46

History of Operating Systems

- Single-user
- Punchcard+printer — spooling to mag tape
- Early "monitor" programs
- Lack of memory protection
- I/O libraries

17 / 46

3rd Generation Computers

- Disks, memory protection, schedulers
- Reasons: efficiency; CPUs were expensive
- Timesharing

18 / 46

Minicomputers

- Much cheaper
- Used similarly to old systems — no memory protection, small disks if any, etc.

19 / 46

Microcomputers

- Same evolutionary path
- Took almost 20 years to get modern OS features

20 / 46

Hardware Features

- Memory protection
- Privileged operations
- Interrupts and system calls
- Timers

21 / 46

Multiprogramming

- Run several programs at once; precise order and timing don't matter
- What's a process? Separately scheduled, protection, isolation, distinct from threads
- Lots of per-process state
- User- and kernel-level stacks
- Process creation and inheritance

22 / 46

Processes

- Separately scheduled; isolated and protected
- Lots of per-process state
- Kernel stack per process
- Process creation: copying, inheritance, etc.
- Process relationships

23 / 46

Threads

- Non-preemptive scheduling
- Used in most graphical applications
- Implementable three different ways: user, kernel, scheduler activations
- The need for reentrancy

24 / 46

Signals

- Interrupts to process
- Catchable; different default actions
- Watch for race conditions

25 / 46

Linux Processes and Interrupts

- Reality is more complex: efficiency
- Lightweight processes: used for threads; less to copy
- Several data structures, some indirect; kernel stack pointer is the handle for most of them
- Wait queues for sleeping processes
- Processes put themselves to sleep

26 / 46

Switching Processes

- Save general registers on the stack; other hardware context in `task_struct`
- Loading new state switches processes; the stack pointer determines most of the state
- Changing the kernel stack pointer changes the process

27 / 46

Creating a Process

- Fork() vs. vfork() vs. clone()
- Copy on write for memory — why?
- Exiting — must close files; process is a zombie until `wait()` call issued

28 / 46

Interrupts

- Synchronous (faults and traps) vs. asynchronous
- IRQs and priorities
- Interrupt masking
- Entering an interrupt handler; returning from interrupts
- First- vs. second-level interrupt handlers
- No process context for interrupts
- Deferred work — softirqs, tasklets, work queues

29 / 46

System Calls

- Why?
- C code vs. assembler glue
- Calling conventions: stack vs. registers
- Copying data to/from user space

30 / 46

Race Conditions; Critical Regions

- Why they happen
- Where they can happen
- Critical regions and blocking
- Desired properties

31 / 46

Lock Variables

- Care needed; must have atomicity
- Test and Set Lock — locks memory bus
- Strict alternation; properties of Peterson's Algorithm
- Spin locks and priority inversion

32 / 46

Semaphores

- Operations: **down** and **up**; what they do
- Different uses of semaphores: counting vs. exclusion (mutex)

33 / 46

Monitors

- Programming language construct
- Operations: **wait** and **signal**

34 / 46

Synchronization Problems

- Dining philosophers
- Watch for deadlocks and starvation
- Readers and writers

35 / 46

Deadlock

- What's a deadlock?
- Conditions for deadlocks
- Recovery, avoidance, prevention
- Release and rerequest
- Two-phase locking

36 / 46

Scheduling

- What is it?
- Types of schedulers and their goals
- Batch, interactive, real-time
- First come, first served
- Shortest first
- Shortest remaining time

37 / 46

Interactive Scheduling

- Round robin
- Quantum length issues: responsiveness vs. overhead; variable size
- Priority scheduling and aging
- Interactivity boost
- Security issues

38 / 46

More Scheduling

- Process history — moving average
- Guaranteed scheduling
- Lottery scheduling
- Fair share scheduling

39 / 46

Real-Time Scheduling

- Hard vs. soft real-time
- Periodic vs. aperiodic; mixing with non-real-time
- Rate Monotonic
- Earliest Deadline First

40 / 46

Multiprocessing

- Asymmetric vs. symmetric
- Locking: fine-grained vs. coarse

41 / 46

Measuring Time

- Statistical time
- I/O, memory, and the scheduler

42 / 46

Evaluation

- What's the metric? What are you optimizing for?
- Deterministic, queueing theory, simulation, build it
- Limits to evaluation

43 / 46

Linux Scheduler

- O(1)
- Run queue per processor
- 140 run queues per processor
- High priority processes get long quanta
- Dynamic priority for quick boost
- Run each queue once before starting at the high priority again

44 / 46

Sleeping and Waking

- Processes sleep on queues
- Queues are awakened, not processes; must check if condition is satisfied

45 / 46

Timers

- Time of day and interval timer
- Types of timers
- What happens at each tick?
- Dynamic timers

46 / 46