The different Unix contexts

- User-level
- Kernel “top half”
  - System call, page fault handler, kernel-only process, etc.
- Software interrupt
- Device interrupt
- Timer interrupt (hardclock)
- Context switch code
Transitions between contexts

- User $\rightarrow$ top half: syscall, page fault
- User/top half $\rightarrow$ device/timer interrupt: hardware
- Top half $\rightarrow$ user/context switch: return
- Top half $\rightarrow$ context switch: sleep
- Context switch $\rightarrow$ user/top half
Top/bottom half synchronization

- Top half kernel procedures can mask interrupts

  ```
  int x = splhigh();
  /* ... */
  splx(x);
  ```

- splhigh disables all interrupts, but also splnet, splbio, splsoftnet, ...

- Masking interrupts in hardware can be expensive
  - Optimistic implementation – set mask flag on splhigh, check interrupted flag on splx
Kernel Synchronization

- Need to relinquish CPU when waiting for events
  - Disk read, network packet arrival, pipe write, signal, etc.

- `int tsleep(void *ident, int priority, ...);`
  - Switches to another process
  - `ident` is arbitrary pointer—e.g., buffer address
  - `priority` is priority at which to run when woken up
  - PCATCH, if ORed into `priority`, means wake up on signal
  - Returns 0 if awakened, or ERESTART/EINTR on signal

- `int wakeup(void *ident);`
  - Awakens all processes sleeping on `ident`
  - Restores SPL a time they went to sleep
    (so fine to sleep at splhigh)
Process scheduling

- **Goal: High throughput**
  - Minimize context switches to avoid wasting CPU, TLB misses, cache misses, even page faults.

- **Goal: Low latency**
  - People typing at editors want fast response
  - Network services can be latency-bound, not CPU-bound

- **BSD time quantum:** 1/10 sec (since ~1980)
  - Empirically longest tolerable latency
  - Computers now faster, but job queues also shorter
Scheduling algorithms

- Round-robin
- Priority scheduling
- Shortest process next (if you can estimate it)
- Fair-Share Schedule (try to be fair at level of users, not processes)
Multilevel feedback queues (BSD)

- Every runnable proc. on one of 32 run queues
  - Kernel runs proc. on highest-priority non-empty queue
  - Round-robins among processes on same queue

- Process priorities dynamically computed
  - Processes moved between queues to reflect priority changes
  - If a proc. gets higher priority than running proc., run it

- Idea: Favor interactive jobs that use less CPU
Process priority

- **p\_nice** – user-settable weighting factor

- **p\_estcpu** – per-process estimated CPU usage
  - Incremented whenever timer interrupt found proc. running
  - Decayed every second while process runnable

  \[
  p\_estcpu \leftarrow \left( \frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1} \right) p\_estcpu + p\_nice
  \]

- Run queue determined by **p\_usrpri/4**

  \[
  p\_usrpri \leftarrow 50 + \left( \frac{p\_estcpu}{4} \right) + 2 \cdot p\_nice
  \]

  (value clipped if over 127)
Sleeping process increases priority

- **p_estcpu not updated while asleep**
  - Instead `p_slptime` keeps count of sleep time

- **When process becomes runnable**

  \[ p_{estcpu} \leftarrow \left( \frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1} \right)^{p_{slptime}} \times p_{estcpu} \]

  - Approximates decay ignoring nice and past loads
Discussion

- 10 people running vi have 1 sec latency?
- How do UNIX signals work?
  - What if signal arrives while process in “top half”
- Does UNIX kernel suffer from priority inversion?
Real-time scheduling

- **Two categories:**
  - *Soft real time*—miss deadline and CD will sound funny
  - *Hard real time*—miss deadline and plane will crash

- **System must handle periodic and aperiodic events**
  - E.g., procs A, B, C must be scheduled every 100, 200, 500 msec, require 50, 30, 100 msec respectively
  - *Schedulable* if \( \sum \frac{CPU}{period} \leq 1 \) (not counting switch time)

- **Variety of scheduling strategies**
  - E.g., first deadline first (works if schedulable)
Multiprocessor scheduling issues

• For TLB and cache, care about which CPU
  - Affinity scheduling—try to keep threads on same CPU

• Want related processes scheduled together
  - Good if threads access same resources (e.g., cached files)
  - Even more important if threads communicate often
    (otherwise would spend all their time waiting)

• Gang scheduling—schedule all CPUs synchronously
  - With synchronized quanta, easier to schedule related processes/threads together
Lottery scheduling

- **Issue lottery tickets to processes**
  - Let $p_i$ have $t_i$ tickets, let $T = \sum_i t_i$
  - Chance of winning next quantum is $t_i/T$.

- **Control avg. proportion CPU for each process**
  - Can also group processes hierarchically for control
  - Subdivide lottery tickets allocated to a particular process
  - Modeled as currencies, funded through other currencies

- **Can transfer tickets to other processes**
  - Perfect for IPC
  - Avoids priority inversion with mutexes
Compensation tickets

• What if proc. only uses fraction $f$ of quantum
  - Say $A$ and $B$ have same number of lottery tickets
  - Proc. $A$ uses full quantum, proc. $B$ uses $f$ fraction
  - Each wins the lottery as often
  - $B$ gets fraction $f$ of $B$’s CPU time. No fair!

• Solution: Compensation tickets
  - If $B$ uses $f$ of quantum, inflate $B$’s tickets by $1/f$ until it next wins CPU
  - E.g., process that uses half of quantup gets schedules twice as often