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

- **Solaris SVR4: 1/100 sec**
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)
- Fancy combinations of the above (e.g., SMART)
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
Multilevel feedback queues (BSD)

- Every runnable proc. on one of 32 run queues
  - Kernel runs proc. on highest-priority non-empty queue
  - Round-robin 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
Limitations of BSD scheduler

- Hard to have isolation / prevent interference
  - Priorities are absolute

- Can’t transfer priority (e.g., to server on RPC)

- No flexible control
  - E.g., In monte carlo simulations, error is $1/\sqrt{N}$ after $N$ trials
  - Want to get quick estimate from new computation
  - Leave a bunch running for a while to get more accurate results

- Multimedia applications
  - Often fall back to degraded quality levels depending on resources
  - Want to control quality of different streams
Lottery scheduling [Waldspurger]

- **Issue lottery tickets to processes**
  - Let $p_i$ have $t_i$ tickets, let $T = \sum t_i$
  - 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 quantum gets schedules twice as often
Limitations of lottery scheduling

- **Expected error $O(\sqrt{n_a})$ for $n_a$ allocations**
  - E.g., process A should have had 1/3 of CPU yet after 1 minute has had only 19 seconds

- **Unpredictable latencies**

- **Idea: Apply ideas from weighted fair queuing**
  - Used for scheduling network routing
  - Can achieve similar goals to lottery scheduling
  - But done deterministically
  - Stride scheduling [Waldspurger] was follow-on to lottery scheduling work
Fair Queuing (FQ)

- Which network packet to send next over a link?
- Ideally, would send one bit from each flow
  - In weighted fair queuing (WFQ), more bits from some flows

- Complication: must send whole packets
FQ Algorithm

- Suppose clock ticks each time a bit is transmitted
- Let \( P_i \) = the length of packet \( i \)
- Let \( S_i \) = time when start to transmit packet \( i \)
- Let \( F_i \) = time when finish transmitting packet \( i \)
  \( (F_i = S_i + P_i) \)

- When does router start transmitting packet \( i \)?
  - If arrived before router finished packet \( i - 1 \) from this flow, then immediately after last bit of \( i - 1 \) \( (F_{i-1}) \)
  - If no current packets for this flow, then start transmitting when arrives (call this \( A_i \))

- Thus: \( F_i = \max(F_{i-1}, A_i) + P_i \)

- Note: \( A_i \) advances after time for all non-empty queues to send one bit
FQ Algorithm (cont)

- For multiple flows
  - Calculate $F_i$ for each packet that arrives on each flow
  - Treat all $F_i$s as timestamps
  - Next packet to transmit is one with lowest timestamp

- Not perfect: can’t preempt current packet

- Example:
SMART

- **Goal:** Support multimedia + real-time applications
- **RT interface:** Specify *time constraints & notifications*
  - constraint = deadline + amount of CPU needed before then
  - notification = time + handler (like sig handler)
- **Track “virtual finishing time” of scheduling a process**
  - Like WFQ packet scheduling algorithm
  - But can temporarily “bias” value to capture urgency
- **Assign processes priority to capture importance**
- **Basic algorithm:**
  - Sort processes by <priority, VFT>
  - If first process conventional, just run it
  - Else, consider all schedulable RT processes more important than first conventional one, run most urgent of them
OS support for periodic tasks

```c
struct timeval {
    long tv_sec; /* seconds */
    long tv_usec; /* microseconds */
};

struct itimerval {
    struct timeval it_interval; /* timer interval */
    struct timeval it_value; /* current value */
};
```

```c
#define ITIMER_REAL 0
#define ITIMER_VIRTUAL 1
#define ITIMER_PROF 2
```

```c
int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue);
```

- Delivers SIGALRM, SIGVTALRM, or SIGPROF periodically
  - E.g., use ITIMER_REAL to display frames in mpeg player