IA-32 Mechanisms for Synchronization

- Atomic operations (block any other CPU memory reference)
  - Memory reference to single aligned operand
  - “Locked” instructions
    - Automatic for XCHG (exchange)
    - Can be applied to INC, DEC, ADD, SUB, AND, OR, XOR, and the bit test and modify instructions
  - Automatic locking for certain state updates
- Memory Reference and Instruction Order Serializing instructions (“barriers”)

Linux Kernel Synchronization Mechanisms

- Disable interrupts (local CPU only)
- Disable softirq handler (local CPU only)
- Disable kernel preemption
- Per-CPU variables/data structures
- IA-32 atomic operations and “barriers”
- Spin locks
  - Exclusive
  - Multiple readers ***or*** a single writer
  - Multiple readers ***and*** a single writer
- Read-Copy-Update (a form of lock-free synchronization)
- Semaphores (allow process blocking)
  - Exclusive
  - Multiple readers ***or*** a single writer

Linux Exclusive Spin Lock

Spin lock represented in one byte (splock);
1 is unlocked, <= 0 is locked

```
spin_lock(slp) {

try_lock:(LOCK)DECB slp->splock
         JNS got_it /* 1-1 = 0, sign bit is 0 */
                    /* 0-1 = -1, sign bit is 1 */
do_spin: PAUSE
         CMPB $0, slp->splock
         JLE do_spin
         JMP try_lock
got_it:  ...
}

spin_unlock(slp) {

MOVB $1, slp->splock
}
```


Linux Multiple Readers ***Or*** One Writer

Read/write lock represented in one long (rwlock);
`0x01000000` = unlocked, no readers
`0x00000000` = locked by writer, no readers
[`0x00ffffff`, `0x00000001`] = -n readers

```
read_lock(rwlp) {
try_lock:  (LOCK)SUBL $1, rwlp->rwlock
           JNS got_it
rl_failed:(LOCK)INCL rwlp->rwlock
do_spin:   PAUSE
           CMPl $1, rwlp->rwlock
           JS do_spin
           (LOCK)DECL rwlp->rwlock
           JS rl_failed
got_it:  ...
}
read_unlock(rwlp) {
(LOCK)INCL rwlp->rwplock
}
```

Linux Multiple Readers *Or* One Writer

Read/write lock represented in one long (rwlock);
`0x01000000` = unlocked, no readers
`0x00000000` = locked by writer, no readers
[`0x00ffffff, 0x00000001`] = -n readers

```
write_lock(rwlp) {
try_lock: (LOCK)SUBL $0x01000000, rwlp->rwlock
          JZ got_it
wl_failed:(LOCK)ADDL $0x01000000, rwlp->rwlock
do_spin:  PAUSE
          CMPL $0x01000000, rwlp->rwlock
          JNE do_spin
          (LOCK)SUBL $0x01000000 rwlp->rwlock
          JNZ wl_failed
got_it:  ...
}
write_unlock(rwlp) {
(LOCK)ADDL $0x01000000, rwlp->rwlock
}
```


Linux Multiple Readers *And* One Writer

```
typedef struct {
      unsigned sequence;
      spinlock_t lock;
} seqlock_t;
extern seqlock_t foo;
```

Writer

```
{
 write_seqlock(&foo);
/* CRITICAL REGION FOR WRITE */
 write_sequnlock(&foo);
}

void write_seqlock(seqlock_t *sl)
{
  spin_lock(&sl->lock);
  atomic_inc(&(sl->sequence));
}
void write_sequnlock(seqlock_t *sl)
{
  atomic_inc(&(sl->sequence));
  spin_unlock(&sl->lock);
}
```

RReader(s)
e

```
  unsigned seq;
  do {
      seq = read_seqbegin(&foo);
      /* CRITICAL REGION FOR READ */
  } while (read_seqretry(&foo, seq));

unsigned read_seqbegin(seqlock_t *sl)
{
 unsigned ret = sl->sequence;
 return ret;
}
int read_seqretry(seqlock_t *sl,
                  unsigned iv)
{/*return 1 if iv odd or changed */
 return (iv & 1) |
        (sl->sequence ^ iv);}
```

Allocating CPUs to Processes

- Generically called “Scheduling” but three distinct components:
  - Assigning relative priorities to processes
  - Choosing the next process and CPU pairing
  - Context switching (save/restore state)
- Historically BSD designed for single CPU
  - /sys/kern/sched_4bsd.c default through release 5.1
- Major FreeBSD redesign for MP and MT in release 5
  - /sys/kern/sched_ule.c default in release 5.2

Linux Scheduler Organization

Generic Scheduling

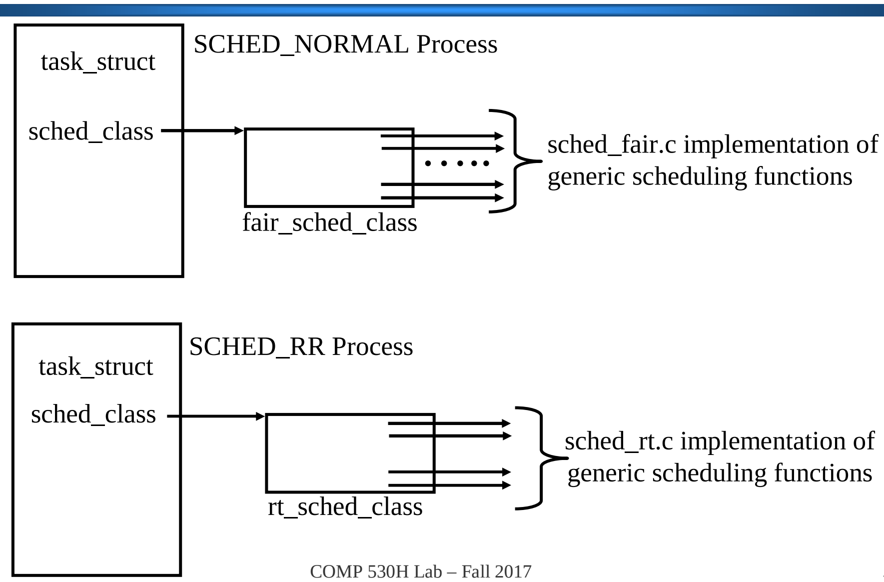
Time Sharing Class (SCHED_NORMAL, SCHED_OTHER) — Priority 100 (high) to 140 (low)

Real Time Class (SCHED_FIFO, SCHED_RR) — Priority 1 (low) to 99 (high)

Scheduler Classes

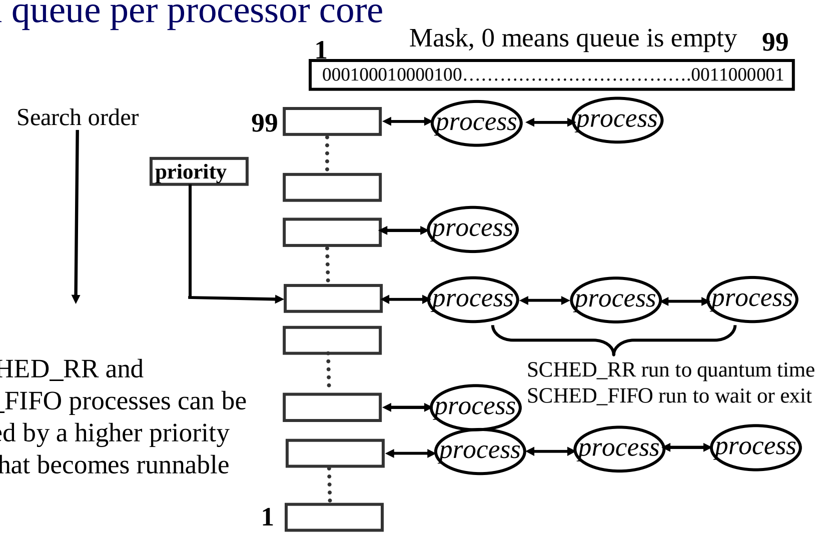
- Each process belongs to exactly one scheduling class identified by sched_class pointer in task_struct
- Generic scheduler code (sched.c) calls scheduling functions for the process through a set of function pointers in a sched_class structure
- Two main entry points to generic scheduler
  - schedule() called from many places in kernel
  - scheduler_tick() called from 1 ms timer interrupts
  - Each type of entry uses different subset of generic functions

Scheduler Class Implementation



Linux Real-Time Implementation

- Run queue per processor core



Both SCHED_RR and SCHED_FIFO processes can be preempted by a higher priority process that becomes runnable

Linux Completely Fair Scheduler (CFS)

- Introduced in 2.6.23, extensively modified since (2.6.24 will be covered here)
- Attempt to emulate ***Generalized Processor Sharing (GPS)***
  - GPS: if ***N*** processes are running, each gets ***1/N*** of CPU ***in parallel*** (a theoretical perfectly fair sharing)
- Key concepts in scheduler/CFS:
  - A run queue for each CPU with load balancing between them
  - Processes executing on CPU(s) accumulate "***virtual***" run time based on their "***nice***" values relative to nice values of other processes
  - The process with the ***smallest*** accumulation of ***virtual*** run time relative to other processes is the next to run after a ***target minimal latency*** (again based on nice values)

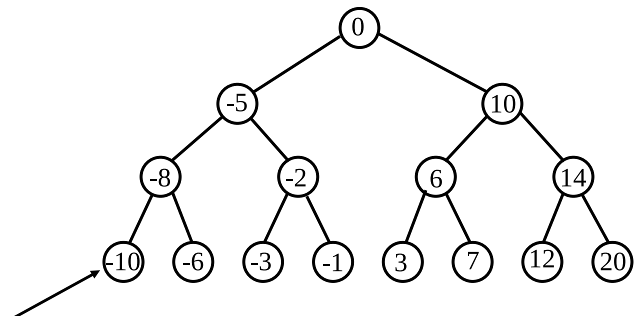
Linux CFS

Run queue is a balanced tree (red-black)

Key for insertion is ***virtual runtime deficit(-)/surplus(+)***

= (process ***virtual*** runtime) – (run tree ***minimum virtual*** runtime)

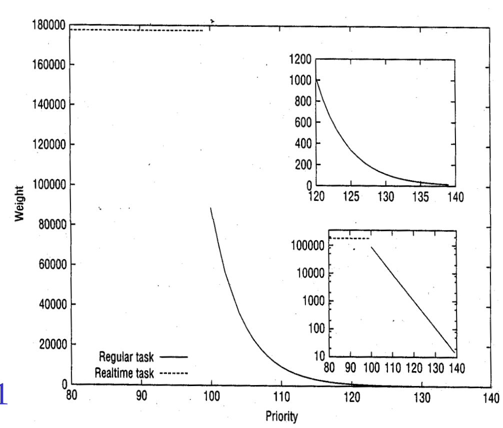
Analogous to “lag” in some scheduling algorithms



Process with largest deficit in ***virtual*** runtime; will run next

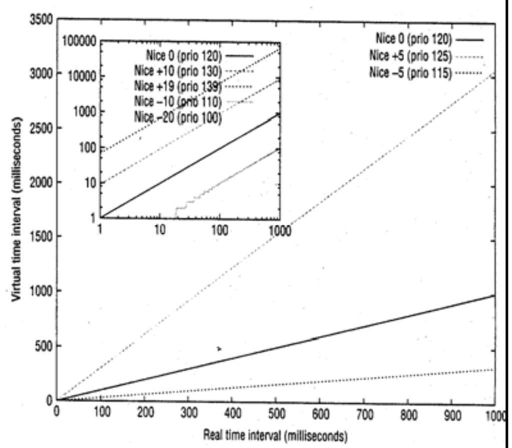
Linux CFS

- Scheduler Parameters:
  - Process “Load” weight as a function of “nice” priority
  - Nice range [-20, 20] is mapped to [100, 140]
    - nice = 0, load = 1024 = ***NICE_0_WEIGHT***
    - nice = -20, load = 88761
    - Nice = 20, load = 15



Linux CFS

- Scheduler Parameters:
  - Process run time scaling (“***virtual time***”) as a function of load weight
    - Ratio of default ***nice*** value to load weight of running process
    - ***exec_weighted = NICE_0_WEIGHT / process_load_weight;***



Linux CFS

- Scheduler Parameters:
  - Target period when each process runs at least once
    - ***period*** = (minimum preemption latency) * (number running processes)
      - = 4 * (number running processes) milliseconds
  - Ideal time slice for a process, given its weight and the total of all process weights on the run tree is:
    - ***ideal_runtime** = **period * (process_load_weight / SUM(process_load_weight))***