Linux Process Blocking

 Processes running in kernel may need to block awaiting the availability of some resource or for some event to happen.
  ✦ semaphore, memory, block from disk, packet from network, etc.
 Processes are blocked by one of the following functions called in kernel code:
  ✦ `sleep_on()`, `interruptible_sleep_on()`
  ✦ `sleep_on_timeout()`, `interruptible_sleep_on_timeout()`
  ✦ `wait_event()`, `wait_event_interruptible()`
  ✦ These functions take a pointer to a wait queue descriptor and possibly a condition to be tested
 Usually leads to a scheduler invocation `(schedule())`

Linux Wait Queues

Wait queue for each resource/event/condition
Queue is list of processes blocked awaiting resource/event/condition
Protected by a queue lock
Linux Process Unblocking (make RUNNABLE and remove from wait queue)

◆ Threads unblocked when kernel makes resource available, event happens, or condition is true
◆ Unblocking may be exclusive or nonexclusive
  ✤ exclusive: selective unblocking of n (typically 1) blocked processes on a wait queue
  ✤ nonexclusive: unblocking of all processes on wait queue
  ✤ Exclusive unblocking avoids “thundering herd” contending for resource that can be held by only one process

Linux Process Unblocking Functions

◆ Multiple forms of base wake_up() function
  ✤ All forms unblock all nonexclusive waiting processes
  ✤ Unblock 1, n, or all exclusive waiting processes
  ✤ Unblock processes in INTERRUPTIBLE state, or also include UNINTERRUPTIBLE processes
  ✤ If priority of any unblocked process is higher than the current process, invoke the scheduler (schedule()), or skip the process priority check.
Linux – Execution Control Paths (nested interrupts, multicore)

Time

PID x

user

CPU 1

PID y

user

kernel

system call (int 0x80, sysenter)

iret

interrupt

fault

CPU 2

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);

- $0x01000000 = \text{unlocked, no readers}$
- $0x00000000 = \text{locked by writer, no readers}$
- $[0x00fffffe, 0x00000001] = -n \text{ readers}$

```c
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);

- $0x01000000 = \text{unlocked, no readers}$
- $0x00000000 = \text{locked by writer, no readers}$
- $[0x00fffffe, 0x00000001] = -n \text{ readers}$

```c
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)DECL rwlp->rwlock
    JNZ wl_failed
    got_it:  ...
}
write_unlock(rwlp) {
(LOCK)ADDL $0x01000000, rwlp->rwlock
}```
typedef struct {
    unsigned sequence;
    spinlock_t lock;
} seqlock_t;

extern seqlock_t foo;

unsigned int seq;

void write_seqlock(seqlock_t *sl) {
    spin_lock(&sl->lock);
    ++sl->sequence;
}

void write_sequnlock(seqlock_t *sl) {
    sl->sequence++;
    spin_unlock(&sl->lock);
}

unsigned read_seqbegin(seqlock_t *sl) {
    unsigned ret = sl->sequence;
    return ret;
}

int read_seqretry(seqlock_t *sl, unsigned iv) {
    unsigned ret = sl->sequence;
    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

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

- task_struct
- sched_class
- sched_fair.c implementation of generic scheduling functions
- fair_sched_class

SCHED_NORMAL Process

- task_struct
- sched_class
- sched_fair.c implementation of generic scheduling functions
- fair_sched_class

SCHED_RR Process

- task_struct
- sched_class
- sched_rt.c implementation of generic scheduling functions
- rt_sched_class

Linux Real-Time Implementation

- Run queue per processor core

- Search order
- priority

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

- Mask, 0 means queue is empty
  99

0001000100010000000000000011000001

SCHED_RR run to quantum time
SCHED_FIFO run to wait or exit
**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 \( \frac{1}{N} \) 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

![Diagram](attachment:image.png)
Scheduler Parameters:

- Process “Load” weight as a function of “nice” priority
- Nice range [-20, 20] is mapped to [100, 140]
  - nice = 0, load = 1024 = \textit{NICE\_0\_WEIGHT}
  - nice = -20, load = 88761
  - Nice = 20, load = 15

Scheduler Parameters:

- Process runtime scaling ("virtual time") as a function of load weight
  - Ratio of default nice value to load weight of running process
  - \textit{exec\_weighted} = \textit{NICE\_0\_WEIGHT} / \textit{process\_load\_weight};
Linux CFS

◆ Scheduler Parameters:
  ✦ Target period when each process runs at least once
      • \textit{period} = (minimum preemption latency) \times (number running processes)
      \textit{\textbullet} = 4 \times (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:
      • \textit{ideal_runtime} = \textit{period} \times \left( \frac{\text{process_load_weight}}{\text{SUM(process_load_weight)}} \right)