Operating Systems: Processes and Threads

keleher

February 19, 2024

Outline

Short-term: Preemptive – 2

- Multi-level Feedback Queue
  - priority of a process depends on its history
  - decreases with accumulated processor time

  - queue 1, 2, $\cdots$, queue N // decreasing priority
  - departure comes from highest-priority non-empty queue
  - arrival coming not from running:
    - joins queue 1
  - arrival coming from running
    - joins queue $\min(i+1, N)$ // i was arrival's previous level

  - To avoid starvation of long processes
    - longer timeslice for lower-priority queues
    - after a process spends a specified time in low-priority queue move it to a higher-priority queue

Lottery Scheduling

- Give each job a specific percentage of CPU. Achieve by:
  - each job has tickets proportional to desired *share*
  - each re-schedule point randomly selects a winning *lottery* ticket

Lottery Scheduling

- Give each job a specific percentage of CPU. Achieve by:
  - each job has tickets proportional to desired *share*
  - each re-schedule point randomly selects a winning *lottery* ticket
- Example: Want A to get twice as much time as B:
  - give A two tickets
  - give B one ticket
  - random choose a ticket at each schedule quantum

Lottery Scheduling

- Give each job a specific percentage of CPU. Achieve by:
  - each job has tickets proportional to desired *share*
  - each re-schedule point randomly selects a winning *lottery* ticket
- Example: Want A to get twice as much time as B:
  - give A two tickets
  - give B one ticket
  - random choose a ticket at each schedule quantum
- Why randomness good?
  - fast: just choose a ticket at random
  - very little state, don't need to track history, etc.
  - avoids nasty corner cases
- Other
  - easily handles different policies: priorities, aging...
  - handles priority *inversion*
    - if low-priority holds lock wanted by high-priority A
    - temporarily give A's tickets to B

Multiprocessor Scheduling

- Set of ready processes is shared
- So scheduling involves
  - get lock on ready queue
    - ensure it is not in a remote processor's cache
  - choose a process (based on its usage of processor, resources, ...)
- Process may acquire affinity to a processor (ie, to its cache)
  - makes sense to respect this affinity when scheduling
- Per-processor ready queues simplifies scheduling, ensures affinity
  - but risk of unfairness and load imbalance
- Could dedicate some processors to long-running processes and others to short/interactive processes

Outline

Interrupt disable/enable: affects only this CPU

```
Disable_Interrupts():                    // abbrv: disable intrpt
  __asm__ "cli"

Enable_Interrupts():                     // abbrv: enable intrpt
  __asm__ "sti"

Begin_Int_Atomic():                      // abbrv: disable intrpt
  ion ← true iff interrupts enabled
  if ion:
    Disable_Interrupts()
  return ion

End_Int_Atomic(ion):                     // abbrv: restore intrpt
  if ion:
    Enable_Interrupts()
```

Spinlocks

- Spinlock in assembly: an int that is 0 iff unlocked

```
Spin_Lock_INTERNAL(x):
  repeat
    busy wait until *x is 0
    set eax to 1
    atomically swap eax and *x
  until eax equals 0
```

```
Spin_Unlock_INTERNAL(x):
  set eax to 0
  atomically swap eax and *x
```

- Spinlock in C: struct {lock, locker, ra, lastlocker}
- Spin_Lock(x): wrapper of assembly fn + update to locker, ra, ...
- Spin_Unlock(x): " " " " " " "
- Ensure interrupts disabled before acquiring a spinlock // Why?
- Restore interrupts after releasing a spinlock

Some spinlock variables

- globalLock // lockKernel(), unlockKernel(); smp.c
- kthreadLock // kthread.c, user.c
- Every *list_t* in `DEFINE_LIST(`*list_t*, *node_t*`)` has a spinlock lock
  - Guards the list in list operations (append, remove, etc)
  - eg, Thread_Queue: s_graveyardQueue.lock, waitQueue.lock
- pidLock // k.thread.c
- kbdQueueLock // keyboard.c
- s_free_space_spin_lock // paging.c
- run_queue_spinlock // sched.c
- mutex->guard // synch.c

Are spinlocks a good idea?

- Project 1 handout describes spinlocks as "quite possibly not a good synchronization tool". Why?

Are spinlocks a good idea?

- Project 1 handout describes spinlocks as "quite possibly not a good synchronization tool". Why?
  - consumes cycles spinning
  - prevents other threads from running
  - ....but does not access memory

Are spinlocks a good idea?

- Project 1 handout describes spinlocks as "quite possibly not a good synchronization tool". Why?
  - consumes cycles spinning
  - prevents other threads from running
  - ....but does not access memory
- Mutex *expects* interrupts to be enabled

Are spinlocks a good idea?

- Project 1 handout describes spinlocks as "quite possibly not a good synchronization tool". Why?
  - consumes cycles spinning
  - prevents other threads from running
  - ....but does not access memory
- Mutex *expects* interrupts to be enabled
  - might need to block

Outline

How GeekOS Handles interrupts

- High level view:
  - assume thread arrived via interrupt (external, trap, exception)
  - construct interrupt state of current thread
  - call the C interrupt handler,

How GeekOS Handles interrupts

- High level view:
  - assume thread arrived via interrupt (external, trap, exception)
  - construct interrupt state of current thread
  - call the C interrupt handler,
  - and then either:
    - resume the current thread

How GeekOS Handles interrupts

- High level view:
  - assume thread arrived via interrupt (external, trap, exception)
  - construct interrupt state of current thread
  - call the C interrupt handler,
  - and then either:
    - resume the current thread
    - switch_to_thread from the run ("ready") queue

How GeekOS Handles interrupts

- High level view:
  - assume thread arrived via interrupt (external, trap, exception)
  - construct interrupt state of current thread
  - call the C interrupt handler,
  - and then either:
    - resume the current thread
    - switch_to_thread from the run ("ready") queue
- Low level view: // in lowlevel.asm
  - push cpu's gp and seg regs // complete interrupt-state
  - call C interrupt handler // with ptr to interrupt-state as arg
  - if not g_preemptionDisabled and g_needReschedule:

    move current thread to runq

    update current thread's state wrt esp, numticks

    get a thread from runq and make it current
  - activate user context (if any) // update ldtr, s_TSS, ...
  - process signal (if any)
  - restore gp and seg regs

Switching a thread

- Switch_To_Thread(thrdptr): // in lowlevel.asm

  // called from Schedule(). interrupts off.
  // using current thread's kernel stack. stack has return addr.
  // current thread struct already in runq or a waitq.
  // save current thread context, activate thread passed as param.

  - change stack content to an intrpt state by adding:
    cs, eflags, fake errorcode/intrpt#, gp and seg regs
  - set threadptr (in arg) as current thread
  - activate user context (if any) // update ldtr, s_TSS, ...
  - process signal (if any)
  - clear APIC interrupt info
  - restore gp and seg regs
  - iret

Scheduling

- Flags checked at every potential switch:
  - g_preemptionDisabled[MAX_CPUS]
  - g_needReschedule[MAX_CPUS]

- Schedule():
  - // current thread voluntarily giving up cpu,
    // eg, Wait(), Mutex_Lock(), Cond_Lock(), Yield().
    // current thread already in runq or a waitq.
  - set g_preemptionDisabled[this cpu] to false
  - runme $\leftarrow$ remove a thread from runq
  - Switch_To_Thread(runme)

- Schedule_And_Unlock(x): // x is a spinlock
  - like Schedule() but unlocks x before Switch_To_Thread(runme)

Outline

- Multiple threads executing concurrently in the same address space
- Threads interact by reading and writing shared memory
- Need to ensure that threads do not "interfere" with each other
- For example, given a linked list X
  - while a thread is adding an item to X, another thread should not read or write X.
  - if thread u blocks when it finds X empty, another thread should not insert an item in between u finding X empty and blocking
- Formalizing "non-interference":

  a code chunk S in a program is atomic if while a thread u is executing S, no other thread can change an intermediate state of u's execution of S.

Synchronization Constructs

- Programming languages usually provide:
  - locks, condition variables, semaphores, ...
- Canonical synchronization problems
  - mutual-exclusion, readers-writers, producer-consumer, ...

Outline

Locks

- Lock operations: acquire and release
- lck ← Lock()   // define a lock
- lck.acq()   // acquire the lock; blocking
  - call only if caller does not hold lck
  - returns only when no other thread holds lck
- lck.rel()   // release the lock; non-blocking
  - call only if caller holds lck
- lck.rel() does not give priority to threads blocked in lck.acq()

Condition variables

- Condition variable operations: wait, signal and signal_all
- A condition variable is associated with a lock
- cv ← Condition(lck) // condition variable associated with lck
- cv.wait() // wait on cv; blocking
  - call only if caller already holds lck
  - atomically release lck and wait on cv
    when awakened: acquire lck and return
- cv.signal() // signal cv; non-blocking
  - call only if caller holds lck
  - wake up a thread (if any) waiting on cv
- cv.signal_all() // wake up all threads waiting on cv
- lck.acq() does not give priority to threads blocked in cv.wait()