Multiple-Processor Scheduling

- **Assumption:** All processors are identical (with identical resources)
  - no advantage in scheduling a thread/process on any specific processor

Three key issues:

1. **The assignment of processes to processors**
   - **how:** static versus dynamic assignment
     - static assignment implies per-processor queues are sufficient
     - dynamic assignment (using a common pool) improves load-balance
   - **what:** master-slave versus peer architectures
     - master-slave architecture: key kernel functions run on a particular processor
       - slaves request service (e.g., I/O) from master
       - simple architecture but master can become a performance bottleneck
     - peer architecture: each processor does *self-scheduling*
       - picks up a processor from the common process pool (requires synchronization)

2. **The use of multiprogramming on individual processors**
   - **uniprocessors:** multiprogramming improves resource utilization
   - **multiprocessors:** less important due to focus on improved performance

3. **The actual dispatching of a process**
   - **process scheduling**
     - **uniprocessors:** priority-based approaches provide sizeable gains over FCFS
     - **multiprocessors:** gains are less clear, so simple policies work well!
       - processes with long CPU bursts are less disruptive (other resources are available)
   - **thread scheduling:** used to exploit true parallelism
     - **load-sharing:** maintain threads in a common pool
     - **dedicated assignment:** one thread per processor for lifetime of program
     - **gang-scheduling:** multiprogrammed processors but all threads belonging to the program are scheduled at the same time across the processors (as a unit)
       - reduces synchronization blocking
       - reduces scheduling overheads
     - **dynamic scheduling:** program (or the OS) can change number of threads
Real-Time Scheduling: Concepts

- Processes have real-time requirements (deadlines)
  - e.g., a video-frame must be processed within certain time
  - growing in importance
    - media-processing on the desktop
    - large-scale use of computers in embedded settings
      - sensors produce data that must be processed and sent to actuators
- Real-time tasks typically considered along two dimensions
  - aperiodic (only one instance) versus periodic (once per period T)
  - hard real-time (strict deadlines) versus soft real-time
    - hard real-time tasks require resource reservation, and
      (typically) specialized hardware and scheduling algorithms
      - earliest-deadline first
      - rate-monotonic scheduling
    - details are beyond the scope of this class (see Prof. Dewar’s Special Topics Class)
- our focus is on supporting soft real-time tasks in a general environment

Soft Real-Time Scheduling

- Most contemporary, general-purpose OSes deal with soft real-time tasks by being as responsive as possible
  - ensure that when a deadline approaches, the task is quickly scheduled
    - minimize latency from arrival of interrupt to start of process execution

Soft Real-Time Scheduling: OS Requirements

- Minimize interrupt processing costs
  - minimization of intervals during which interrupts are disabled
- Minimize dispatch latency
  - preemptive priority scheduling
    - real-time processes have higher priority than non real-time processes
    - priority of real-time processes does not degrade over time
  - current activity must be preemptible
    - unacceptable
      - traditional UNIX approach (waiting for system call completion)
      - preemption at safe points
    - acceptable: entire kernel must be preemptible (e.g., Solaris 2)
      - kernel data structures protected by synchronization mechanisms
    - must cope with the priority inversion problem
      - a solution is priority inheritance
      - process holding the lock inherits priorities of processes waiting for the lock
      - priority reverts to original values when lock is released

Windows NT Scheduler

- Preemptive, priority based
- 32 priority levels
  - higher priority numbers imply higher priority
  - 0-15 are variable priority classes
    - normal processes start off at this level
    - process has a base priority (can take values from 0-15)
    - threads in the process can start at priority = (base_priority + 2)
      - NT Executive raises priorities of I/O-bound threads (max value is 15)
      - NT Execute lowers priorities of CPU-bound threads (min value is base_priority-2)
  - 16-31 are real-time priority classes
    - real-time threads have a fixed priority
    - threads within a particular level processed according to RR
Fair-Share Scheduling

- Problems with priority-based systems
  - priorities are absolute: no guarantees when multiple jobs with same priority
  - no encapsulation and modularity
    - behavior of a system module is unpredictable: a function of absolute priorities assigned to tasks in other modules
- Solution: Fair-share scheduling
  - each job has a share: some measure of its relative importance
    - denotes user’s share of system resources as a fraction of the total usage of those resources
    - e.g., if user A’s share is twice that of user B
    - then, in the long term, A will receive twice as many resources as B
- Traditional implementations
  - keep track of per-process CPU utilization (a running average)
  - reprioritize processes to ensure that everyone is getting their share
  - are slow!

Lottery Scheduling

- A randomized mechanism for efficient proportional-share resource management
  - resource consumption rates of active computations are proportional to the relative shares they are allocated
- Key ideas
  - each process has certain number of lottery tickets (its share)
    - a single physical ticket can represent multiple logical tickets
    - similar to currency notes
  - each allocation is determined by holding a lottery
  - the resource is granted to the process holding the winning ticket

Lottery Scheduling: Concepts

- Lottery Tickets: Resource rights
  - abstract: rights are quantified independent of machine details
  - relative: fraction of resource varies dynamically in proportion to contention for that resource
  - uniform: rights for heterogeneous resources can be homogeneously represented

- Lotteries: A probabilistically fair scheduler
  - expected allocation of resources to processes is proportional to the number of tickets that they hold
  - number of lotteries won by a process has a binomial distribution
    - probability $p$ of winning = $n/T$
    - after $n$ lotteries, $E[w] = np$ and variance = $np(1-p)$
  - number of lotteries to first win has a geometric distribution
    - $E[n] = 1/p$, and variance = $(1-p)/p^2$

Lottery Scheduling: Benefits

- Modular resource management
  - policies of individual system modules can be insulated from other modules
    - processes within the module are assigned relative shares
    - these shares do not change no matter what happens in the rest of the system
  - Lottery tickets are first-class objects (they can be transmitted using IPC)
    - ticket transfers: transfer one process’ share to another
      - e.g., for priority inheritance
    - ticket inflation: adjust relative shares of two processes
      - mutually trusted clients can create additional tickets
    - ticket currencies: different modules can have different interpretations of “value” of a ticket
      - isolate loads across logical trust boundaries
      - an exchange mechanism ensures conversion
    - compensation tickets: when a process uses less than its share
      - the OS can give it a compensation ticket that boosts its share
Lottery Scheduling: Implementation

- Context: Mach 3.0 microkernel, scheduling quantum = 100ms
- Random number generator
  - multiplicative congruential generator \( S' = (A \times S) \mod (2^{31} - 1) \) for \( A = 16807 \)
  - \(-10\) instructions of RISC code
- Lotteries
  - pick a random number from 0..total number of tickets
  - find winner by traversing process list, accumulating a running ticket sum until it reaches the winning value
  - optimizations
    - “move to front” heuristic
    - keep tree of partial sums

Lottery Scheduling: Implementation (contd.)

- Ticket currencies
  - ticket and currency objects
  - currency graph
  - computing values
    - currency: sum value of backing tickets
    - ticket: compute share of currency value
  - example:
    - task 1 funding in base units
      - \( \frac{100}{300} \times 1000 \)
      - \( 333 \) base units
- Kernel interface
  - to create/destroy tickets and currencies
  - to fund/unfund a currency
  - to compute current values of tickets and currencies in base units

Lottery Scheduling: Performance

- Overheads
  - comparable to standard time-sharing scheduler
  - \(-1000\) RISC instructions
- Fairness
  - Dhrystone benchmark
  - two tasks with different ratio of tickets
  - three 60-second runs
- Fairness over time
  - 8-second averages
  - 2 tasks: 2:1 allocation
  - function of scheduling quanta

Lottery Scheduling: Performance (contd.)

- Ticket inflation
  - Monte-Carlo trials
  - three tasks
  - funding based on relative error
- Ticket exchange
  - client-server computation
  - client donates tickets to server on RPC
  - three clients: 8:3:1 allocation
Lottery Scheduling: Performance (contd.)

- Load insulation
  - ticket currencies insulate loads
  - currencies A, B
    - 2:1 funding
  - task A funding 100.A
  - task B1 funding 100.B
  - task B2 joins with funding 100.B

Lottery Scheduling: Other Issues

- Can also be used in various other contexts
  - synchronization resources
    - to avoid the priority inversion problem
  - memory pages
    - use “inverse lotteries”
    - decides which process loses its pages
    - more tickets reduce likelihood of resource revocation
  - extensions to handle multiple resources

Lottery Scheduling: Discussion

- Mechanism for
  - insulating groups of processes from behavior of other processes
    - ticket currencies
  - ensuring relative resource consumption rates for processes within a module

- What can this be used for?

Lecture Summary

- CPU Scheduling (contd.)
  - multiple-processor scheduling
  - real-time and fair-share scheduling
  - case study: lottery scheduling
    - rationale and design
    - evaluation
    - performance
Next Lecture

- Process Synchronization
  - background
  - the critical-section problem
  - synchronization hardware
  - semaphores

Readings
- Silberschatz/Galvin: Sections 6.1 - 6.4