LAST LECTURE

Scheduling Algorithms:
- FIFO
- Shortest job next
- Shortest remaining job next
- Highest Response Rate Next (HRRN)

ROUND-ROBIN

quantum=1:

quantum=4:

Performance of round-robin scheduling:
- Average waiting time: not optimal
- Performance depends heavily on size of time-quantum:
  - too short: overhead for context switch becomes too expensive
  - too large: degenerates to FCFS policy
  - rule of thumb: about 80% of all bursts should be shorter than 1 time quantum
- no starvation

PRIORITIES

- Each thread is associated with a priority
- Basic mechanism to influence scheduler decision:
  - Scheduler will always choose a thread of higher priority over one of lower priority
  - Implemented via multiple FCFS ready queues (one per priority)
- Lower-priority may suffer starvation
  - adapt priority based on thread’s age or execution history
- Priorities can be defined internally or externally
  - internal: e.g., memory requirements, I/O bound vs CPU bound
  - external: e.g., importance of thread, importance of user
Priority queueing:

- Priority 4 (Highest priority)
- Priority 3
- Priority 2
- Priority 1 (Lowest priority)

Feedback scheduling:

- Penalize jobs that have been running longer
- $q = 2^i$: longer time slice for lower priority (reduce starvation)

Priorities influence access to resources, but do not guarantee a certain fraction of the resource (CPU etc)!

Lottery Scheduling

- Process gets “lottery tickets” for various resources
- More lottery tickets imply better access to resource

Advantages:

- Simple
- Highly responsive
- Allows cooperating processes/threads to implement individual scheduling policy (exchange of tickets)
Example (taken from *Embedded Systems Programming*):

Four processes running concurrently:
- Process A: 15% of CPU time
- Process B: 25% of CPU time
- Process C: 5% of CPU time
- Process D: 55% of CPU time

How many tickets should each process get to achieve this?

Number of tickets in proportion to CPU time, e.g., if we have 20 tickets overall:
- Process A: 15% of tickets: 3
- Process B: 25% of tickets: 5
- Process C: 5% of tickets: 1
- Process D: 55% of tickets: 11

**Traditional UNIX Scheduling (SVR3, 4.3 BSD)**

**Objectives:**
- Support for time sharing
- Good response time for interactive users
- Support for low-priority background jobs

**Strategy:**
- Multilevel feedback using round robin within priorities
- Priorities are recomputed once per second
  - Base priority divides all processes into fixed bands of priority levels
  - Priority adjustment capped to keep processes within bands
- Favours I/O-bound over CPU-bound processes

**Note:** UNIX traditionally uses counter-intuitive priority representation (higher value = less priority)

**Bands:**
- Decreasing order of priority
- Swapper
- Block I/O device control
- File manipulation
- Character I/O device control
- User processes

**Advantages:**
- Relatively simple, effective
- Works well for single processor systems

**Disadvantages:**
- Significant overhead in large systems (recomputing priorities)
- Response time not guaranteed
- Non-preemptive kernel: lower priority process in kernel mode can delay high-priority process
**Non-preemptive vs Preemptive Kernel**
- Kernel data structures have to be protected.
- Basically, two ways to solve the problem:
  - Non-preemptive: disable all (most) interrupts while in kernel mode, so no other thread can get into kernel mode while in critical section
    - Priority inversion possible
    - Coarse grained
    - Works only for uniprocessor
  - Preemptive: just lock kernel data structure which is currently modified
    - More fine-grained
    - Introduces additional overhead, can reduce throughput

**Multiprocessor Scheduling**

What kind of systems and applications are there?

**Classification of Multiprocessor Systems:**

- **(a) Tightly coupled multiprocessing**
  - Processors share main memory, controlled by single operating system, called symmetric multi-processor (SMP) system

- **(b) Loosely coupled multiprocessor**
  - Each processor has its own memory and I/O channels
  - Generally called a distributed memory multiprocessor

- **(c) Distributed System**
  - Complete computer systems connected via wide area network
  - Communicate via message passing

**Parallelism**

**Independent parallelism:**
- Separate applications/jobs
- No synchronization
- Parallelism improves throughput, responsiveness
- Parallelism doesn’t affect execution time of (single threaded) programs

**Coarse and very coarse-grained parallelism:**
- Synchronization among processes is infrequent
- Good for loosely coupled multiprocessors
  - Can be ported to multiprocessor with little change
Medium-grained parallelism:
- Parallel processing within a single application
  - Application runs as multithreaded process
  - Threads usually interact frequently
  - Good for SMP systems
  - Unsuitable for loosely-coupled systems

Fine-grained parallelism:
- Highly parallel applications
  - e.g., parallel execution of loop iterations
  - Very frequent synchronisation
  - Works only well on special hardware
    - vector computers, symmetric multithreading (SMT) hardware

Multiprocessor Scheduling:

We discuss:
- Tightly coupled multiprocessing
- Very coarse to medium grained parallelism
- Homogeneous systems (all processors have same specs, access to devices)

Design Issues:
- How to assign processes/threads to the available processors?
- Multiprogramming on individual processors?
- Which scheduling strategy?
- Scheduling dependend processes

Assignment of threads to processors:

- Treat processors as a pooled resource and assign threads to processors on demand
  - Permanently assign threads to a processor
    - Dedicate short-term queue for each processor
      - Low overhead
      - Processor could be idle while another processor has a backlog
    - Dynamically assign process to a processor
      - higher overhead
      - poor locality
      - better load balancing

Who decides which thread runs on which processor?

Master/slave architecture:
- Key kernel functions always run on a particular processor
- Master is responsible for scheduling
- Slave sends service request to the master
  - simple
  - one processor has control of all resources, no synchronisation
  - Failure of master brings down whole system
  - Master can become a performance bottleneck
Peer architecture:
- Operating system can execute on any processor
- Each processor does self-scheduling
- Complicates the operating system
  - Make sure no two processors schedule the same thread
  - Synchronise access to resources
- Proper symmetric multiprocessing

Load is distributed evenly across the processors
- Use global ready queue
  - Threads are not assigned to a particular processor
  - Scheduler picks any ready thread (according to scheduling policy)
  - Actual scheduling policy less important than on uniprocessor
- No centralized scheduler required

Disadvantages of time sharing:
- Central queue needs mutual exclusion
  - Potential race condition when several CPUs are trying to pick a thread from ready queue
  - May be a bottleneck blocking processors
- Preempted threads are unlikely to resume execution on the same processor
  - Cache use is less efficient, bad locality
  - Different threads of same process unlikely to execute in parallel
  - Potentially high intra-process communication latency
**Slide 24**

**Load Sharing: Space Sharing**

Scheduling multiple threads of the same process across multiple CPUs

- Statically assigned to CPUs at creation time (figure) or
- Dynamic assignment using a central server

**Slide 26**

**Gang Scheduling**

Combined time and space sharing:

- Simultaneous scheduling of threads that make up a single process
- Useful for applications where performance severely degrades when any part of the application is not running
  - e.g., often need to synchronize with each other

**Slide 27**

**SMP Support in Modern General Purpose OS’s**

- Solaris 8.0: up to 128
- Linux 2.4: up to 32
- Windows 2000 Data Center: up to 32
- OS/2 Warp: up to 64

**SMP Scheduling in Linux 2.4:**

- Tries to schedule process on same CPU
- If the CPU busy, assigns it to an idle CPU
- Otherwise, checks if process priority allows interrupt on preferred CPU
- Uses spin locks to protect kernel data structures

---

(source http://www.2cpu.com)
**Windows 2000 Scheduling**

- Priority driven, preemptive scheduling system
- If thread with higher priority becomes ready to run, current thread is preempted
- Scheduled at thread granularity
- Priorities: 0 (zero-page thread), 1-15 (variable levels), 16-31 (realtime levels — soft)
- Each thread has a quantum value, clock-interrupt handler deducts 3 from running thread quantum
- Default value of quantum: 6 Windows 2000 Professional, 36 on Windows 2000 Server
- Most wait-operations result in temporary priority boost, favouring I/O-bound threads

**Real-Time Systems**

**What is a real-time system?**

A real-time system is a system whose correctness includes its response time as well as its functional correctness.

**What is a hard real-time system?**

A real-time system with guaranteed worst case response times.

- Hard real-time systems fail if deadlines cannot be met
- Service of soft real-time systems degrades if deadlines cannot be met

**Real-time systems:**

- No clear separation
- System may meet hard deadline of one application, but not of other
- Depending on application, time-scale may vary from microseconds to seconds
- Most systems have some real-time requirements

**Overview:**

- Real time systems
  - Hard and soft real time systems
  - Real time scheduling
  - A closer look at some real time operating systems
Soft Real-time Applications:
- Many multi-media apps
- e.g., DVD or MP3 player
- Many real-time games, networked games

Hard Real-time Applications:
- Control of laboratory experiments
- Embedded devices
- Process control plants
- Robotics
- Air traffic control
- Telecommunications
- Military command and control systems

Hard real-time systems:
- often lack full functionality of modern OS
- secondary memory usually limited or missing
- data stored in short term or read-only memory
- no time sharing

Modern operating systems provide support for soft real-time applications
Hard real-time OS either specially tailored OS, modular systems, or customized version of general purpose OS.

Characteristics of real-time operating systems

Deterministic: How long does it take to acknowledge interrupt?
- Operations are performed at fixed, predetermined times or within predetermined time intervals
- Depends on:
  - response time of system for interrupts
  - capacity of system
- Cannot be fully deterministic when processes are competing for resources
- Requires preemptive kernel

Responsive: How long does it take to service the interrupt?
- Includes amount of time to begin execution of the interrupt
- Includes the amount of time to perform the interrupt

User control: User has much more control compared to ordinary OS’s
- User specifies priority
- Specify paging
- Which processes must always reside in main memory
- Disks algorithms to use
- Rights of processes

Reliability: Failure, loss, degradation of performance may have catastrophic consequences
- Attempt either to correct the problem or minimize its effects while continuing to run
- Most critical, high priority tasks execute
Characteristics of Real-time Operating Systems

General purpose OS objectives like:
- Speed
- Fairness
- Maximising throughput
- Minimising average response time
are not priorities in real time OS’s!

Features of real-time operating systems:
- Fast context switch
- Small size
- Ability to respond to external interrupts quickly
- Predictability of system performance!
- Use of special sequential files that can accumulate data at a fast rate
- Preemptive scheduling based on priority
- Minimization of intervals during which interrupts are disabled
- Delay tasks for fixed amount of time

Real-time scheduling

Preemptive round-robin:

Non-preemptive priority:
Preemption point:
- Current process
- Request from a real-time process
- Real-time process
- Scheduling time
- Wait for next preemption point

Immediate preemptive:
- Current process
- Real-time process
- Scheduling time
- Real-time process preempts current process and executes immediately

Classes of Algorithms:
- Static table-driven
  - Suitable for periodic tasks
  - Input: periodic arrival, ending and execution time
  - Output: schedule that allows all processes to meet requirements (if at all possible)
  - Determines at which points in time a task begins execution
- Static priority-driven preemptive
  - Static analysis determines priorities
  - Traditional priority-driven scheduler is used
- Dynamic planning-based
  - Feasibility to integrate new task is determined dynamically
- Dynamic best effort
  - No feasibility analysis
  - Typically aperiodic, no static analysis possible
  - Does its best, procs that missed deadline aborted

When are periodic events schedulable?
- \( P_i \): period with which event \( i \) occurs
- \( C_i \): CPU time required to handle event \( i \)

A set of events \( e_1 \) to \( e_m \) is schedulable if
\[
\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1
\]

Example:
- Three periodic events with periods of 100, 200, and 500msecs
- Require 50, 30, and 100msec of CPU time

A set of events is schedulable if
\[
\frac{50}{100} + \frac{30}{200} + \frac{100}{500} = 0.5 + 0.15 + 0.2 = 0.85 \leq 1
\]
**Deadline Scheduling**

Current systems often try to provide real-time support by
- starting real time tasks as quickly as possible
- speeding up interrupt handling and task dispatching

Not necessarily appropriate, since
- real-time applications are not concerned with speed but with reliably completing tasks
- priorities alone are not sufficient

### Additional information used:
- **Ready time**
  - sequence of times for periodic tasks, may or may not be known statically
- **Starting deadline**
- **Completion deadline**
- **Processing time**
  - may or may not be known, approximated
- **Resource requirements**
- **Priority**
- **Subtask scheduler**

### Deadline Scheduling

**Earliest deadline first** strategy is provably optimal. It
- minimises number of tasks that miss deadline
- if there is a schedule for a set of tasks, earliest deadline first will find it

**Earliest deadline first**
- can be used for dynamic or static scheduling
- works with starting or completion deadline
- for any given preemption strategy
  - starting deadlines are given: nonpreemptive
  - completion deadline: preemptive

### Two tasks:

**Sensor A:**
- data arrives every 20ms
- processing takes 10ms

**Sensor B:**
- data arrives every 50ms
- processing takes 25ms

**Scheduling decision every 10ms**

<table>
<thead>
<tr>
<th>Task</th>
<th>Arrival Time</th>
<th>Execution Time</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(1)</td>
<td>0</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>A(2)</td>
<td>20</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>A(3)</td>
<td>40</td>
<td>10</td>
<td>60</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>B(1)</td>
<td>0</td>
<td>25</td>
<td>50</td>
</tr>
<tr>
<td>B(2)</td>
<td>50</td>
<td>25</td>
<td>100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Periodic threads with completion deadline:

Arrival times, execution times, and deadlines:

Slide 48

Fixed-priority scheduling:
- A has priority

Slide 49

Earliest-deadline scheduling using completion deadlines:

Slide 50

Rate Monotonic Scheduling

Works for processes which:
- are periodic
- need the same amount of CPU time on each burst
- optimal static scheduling algorithm
- guaranteed to succeed if

\[
\sum_{i=1}^{m} \frac{C_i}{P_i} \leq m \times (2^{\frac{1}{m}} - 1)
\]

for \(m = 1, 10, 100, 1000: 1, 0.7, 0.695, 0.693\)

Works by:
- assigning priorities to threads on the basis of their periods
- highest-priority task is the one with the shortest period

Aperiodic threads with starting deadline:

Slide 49

Requirements:

Arrival times

Slide 51

Periodic task timing diagram:

Slide 51
**Task set with RMS:**

![Diagram showing task set with RMS](image)

**Why use RMS?**

Despite some obvious disadvantages of RMS over EDF, RMS is sometimes used:
- It has a lower overhead
- Simple
- In practice, performance similar
- Greater stability, predictability

**Linux 2.4 Scheduling — Soft Real-time Support**

User assigns static priority to real time processes (1-99), never changed by scheduler.
- Conventional processes have dynamic priority, always lower than real time processes:
  - Sum of base priority and
  - Number of clock ticks left of quantum for current epoch

Scheduling classes:
- `SCHED_FIFO`: First-in-first-out real-time threads
- `SCHED_RR`: Round-robin real-time threads
- `SCHED_OTHER`: Other, non-real-time threads

Within each class multiple priorities may be used.
- Deadlines cannot be specified, no guarantees given.
- Due to non-preemptive kernel, latency can be too high for real-time systems.
Linux scheduling:

(a) Relative thread priorities

D → B → C → A →

(b) Flow with FIFO scheduling

D → B → C → B → C → A →

(c) Flow with RR scheduling

UNIX SVR4 Scheduling

- Highest preference to real-time processes
- Next-highest to kernel-mode processes
- Lowest preference to other user-mode processes
- Real-time processes may block system services

SVR4 dispatch queues:

Process waiting in kernel mode

Process waiting in user mode

Windows 2000 Scheduling

- Priorities organized into two bands or classes
  - Real-time
  - Variable
- Priority-driven preemptive scheduler
- also, no deadlines, no guarantees
Slide 60

Priority

System priorities

User priorities

Zero page thread

Idle thread

Slide 61

Process Priority

Thread’s Base Priority

Thread’s Dynamic Priority

Slide 62

Problem:

- in real life applications, many tasks are not always periodic.
- static priorities may not work

If real time threads run periodically with same length, fixed priority is no problem:

- a: periodic real time thread, highest priority
- b: periodic real time thread
- various different low priority tasks (e.g., user I/O)

Slide 63

But if frequency of high priority task increases temporarily, system may encounter overload:

- system not able to respond
- system may not be able to perform requested service
Example:
Network interface control driver, requirements:
- avoid if possible to drop packets
- definitely avoid overload

If receiver thread get highest priority permanently, system may go into overload if incoming rate exceeds a certain value.
- expected frequency: packet once every 64µs
- CPU time required to process packet: 25µs
- 32-entry ring buffer, max 50% full

Basic Idea: “simulation” of periodic behaviour of thread by assigning
- realtime priority: \( P_r \)
- background priority: \( P_b \)
- execution budget: \( E \)
- replenishment interval: \( R \)

to thread.
- Whenever thread exhausts execution budget, priority is set to background priority \( P_b \)
- When thread blocks after \( n \) units, \( n \) will be added to execution budget \( R \) units after execution started
- When execution budget is incremented, thread priority is reset to \( P_r \)

Example:
- execution budget: 5
- replenishment interval: 13

Thread does not block:
Thread blocks:

- 5
- budget
- replenishment interval
- time

Slide 68

(0) execution starts, 1st replenishment interval starts
(3) thread blocks
(5) continues execution, 2nd replenishment interval starts
(7) budget exhausted
(13) budget set to 3, thread continues execution
(16) budget exhausted
(18) budget set to 2
(19) thread continues execution

Example: Network interface control Driver

- use expected incoming rate and desired max CPU utilisation of thread to compute execution budget and replenishment period
- if no other threads wait for execution, packets can be processed even if load is higher
- otherwise, packets may be dropped

Slide 69

- packet every 64\mu s
- receiver thread 25\mu s/packet
- period: 64\mu s * 16 = 1024\mu s
- execution time: 25\mu s * 16 = 400\mu s
- CPU load caused by receiver thread: 400/1024 = 0.39, about 39%

Slide 70

HARD REAL TIME OS

We look at examples of two types of systems:

- configurable hard real time systems
- system designed as real time OS from the start
- hard real-time variants of general purpose OSs
- try to alleviate shortcomings of OS with respect to real time apps

Slide 71

REAL-TIME SUPPORT IN LINUX

- Scheduling:
  - POSIX SCHED_FIFO, SCHED_RR,
  - ongoing efforts to improve scheduler efficiency

- Virtual Memory:
  - no VM for real-time apps
  - mlock() and mlockall() to switch off paging

- Timer: resolution: 10ms, too coarse grained for real-time apps
**Slide 72**

**High kernel latency in Linux**

Possible solutions:
- **Low Latency Linux**
  - thread in kernel mode yields CPU
  - reduces size of non-preemptable sections
  - used in some real-time variants of Linux
- **Preemptable Linux**
  - kernel data protected using mutexes/spinlocks
- **Lock breaking preemptable Linux**
  - combination of previous two approaches

**Slide 73**

**RTLlinux**

- abstract machine layer between actual hardware and Linux kernel
- takes control of
  - hardware interrupts
  - timer hardware
  - interrupt disable mechanism
- real time scheduler runs with no interference from Linux kernel
- programmer must utilise RTLlinux API for real time applications

**Slide 74**

**QNX**

- Microkernel based architecture
- POSIX standard API
- Modular — can be customised for very small size (e.g., embedded systems) or large systems
- Memory protection for user applications and OS components

**Scheduling:**
- FIFO scheduling
- Round-robin
- Adaptive scheduling
  - thread consumes its timeslice, its priority is reduced by one
  - thread blocks, it immediately comes back to its base priority
- POSIX sporadic scheduling

**Slide 75**

**Windows CE 3.0**

Componentised OS designed for embedded systems with hard real-time support
- handles nested interrupts
- handles priority inversion based on priority inheritance

**Offers**
- guaranteed upper bound on high priority thread scheduling
- guaranteed upper bound on delay for interrupt service routines
Linux scheduling:

(a) Relative thread priorities

(b) Flow with FIFO scheduling

(c) Flow with RR scheduling

UNIX SVR4 Scheduling

- Highest preference to real-time processes
- Next-highest to kernel-mode processes
- Lowest preference to other user-mode processes

SVR4 dispatch queues:

Windows 2000 Scheduling

- Priorities organized into two bands or classes
  - Real-time
  - Variable
- Priority-driven preemptive scheduler

Windows 2000 Scheduling 39