Outline for 9/26

• Objective for today’s lecture:
  Advanced topics in scheduling

• Administrivia
  – Sign up for demos on the demo scheduler
  – Find out where your grader will be holding their demos
  – Do not sign up with same grader twice

Real Time Schedulers

• Real-time schedulers must support regular, periodic execution of tasks (e.g., continuous media).
  – CPU Reservations
    • “I need to execute for X out of every Y units.”
    • Scheduler exercises admission control at reservation time: application must handle failure of a reservation request.
  – Proportional Share
    • “I need \( \frac{1}{n} \) of resources”
  – Time Constraints
    • “Run this before my deadline at time \( T \).”
Assumptions

- Tasks are periodic with constant interval between requests, $T_i$ (request rate $1/T_i$)
- Each task must be completed before the next request for it occurs
- Tasks are independent
- Worst-case run-time for each task is constant (max), $C_i$
- Any non-periodic tasks are special

Task Model
Definitions

- **Deadline** is the time of the next request.
- **Overflow** at time \( t \) if \( t \) is the deadline of an unfulfilled request.
- **Feasible** schedule - for a given set of tasks, a scheduling algorithm produces a schedule so no overflow ever occurs.
- **Critical instant** for a task - time at which a request will have the largest response time.
  - Occurs when the task is requested simultaneously with all tasks of higher priority.

Task Model
Rate Monotonic

- Assign priorities to tasks according to their request rates, independent of run times.
- Optimal in the sense that no other fixed priority assignment rule can schedule a task set which cannot be scheduled by rate monotonic.
- If feasible (fixed) priority assignment exists for some task set, rate monotonic is feasible for that task set.
Earliest Deadline First

- Dynamic algorithm
- Priorities are assigned to tasks according to the deadlines of their current request
- With EDF there is no idle time prior to an overflow
- For a given set of \( m \) tasks, EDF is feasible iff
  \[
  \frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_m}{T_m} \leq 1
  \]
- If a set of tasks can be scheduled by any algorithm, it can be scheduled by EDF
Proportional Share

- Goals: to integrate real-time and non-real-time tasks, to police ill-behaved tasks, to give every process a well-defined share of the processor.
- Each client, $i$, gets a weight $w_i$
- Instantaneous share $f_i(t) = w_i/(\sum w_j)$
- Service time ($f_i$ constant in interval)
  $S_i(t_0, t_1) = f_i(t) \Delta t$
- Set of active clients varies $\Rightarrow f_i$ varies over time

$$S_i(t_0, t_1) = \int_{t_0}^{t_1} f_i(\tau) \, d\tau$$

Common Proportional Share Competitors

- Weighted Round Robin – RR with quantum times equal to share
  RR:  

  WRR:  

- Fair Share – adjustments to priorities to reflect share allocation (compatible with multilevel feedback algorithms)

  20  

  20  

  10  

  Linux
Common Proportional Share Competitors

- Weighted Round Robin – RR with quantum times equal to share
  RR:
  WRR:

- Fair Share – adjustments to priorities to reflect share allocation (compatible with multilevel feedback algorithms)

Linux
Common Proportional Share Competitors

• Fair Queuing
  • Weighted Fair Queuing
  • Stride scheduling
  – VT – Virtual Time advances at a rate proportional to share
    \[ \text{VT}_A(t) = \frac{S_A(t)}{W_A} \]
  – VFT – Virtual Finishing Time: VT a client would have after executing its next time quantum
    \[ \text{VFT} = \frac{2}{3}, \frac{2}{2}, \frac{1}{1} \]
  – WFQ schedules by smallest VFT
    • \( E_A \) never below -1

Lottery Scheduling

• Lottery scheduling [Waldspurger96] is another scheduling technique.
  • Elegant approach to periodic execution, priority, and proportional resource allocation.
  – Give \( W_p \) “lottery tickets” to each process \( p \).
  – \text{GetNextToRun} selects “winning ticket” randomly.
    • If \( S_{W_p} = N \), then each process gets CPU share \( W_p/N \)... probabilistically, and over a sufficiently long time interval.
  – Flexible: tickets are transferable to allow application-level adjustment of CPU shares.
  – Simple, clean, fast.
    • Random choices are often a simple and efficient way to produce the desired overall behavior (probabilistically).
Lottery Scheduling
Waldspurger and Weihl (OSDI 94)

• Goal: responsive control over the relative rates of computation
• Claims:
  – Support for modular resource management
  – Generalizable to diverse resources
  – Efficient implementation of proportional-share resource management: consumption rates of resources by active computations are proportional to relative shares allocated

Basic Idea

• Resource rights are represented by lottery tickets
  – Give $W_p$, “lottery tickets” to each process $p$.
  – abstract, relative (vary dynamically wrt contention), uniform (handle heterogeneity)
  – responsiveness: adjusting relative # tickets gets immediately reflected in next lottery
• At allocation time: hold a lottery; Resource goes to the computation holding the winning ticket.
  – $GetNextToRun$ selects “winning ticket” randomly.
**Fairness**

- Expected allocation is proportional to # tickets held - actual allocation becomes closer over time.
- Number of lotteries won by client
  \[ E[w] = n \cdot p \text{ where } p = \frac{t}{T} \]
- Response time (# lotteries to wait for first win)
  \[ E[n] = \frac{1}{p} \]

<table>
<thead>
<tr>
<th>( w )</th>
<th># wins</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t )</td>
<td># tickets</td>
</tr>
<tr>
<td>( T )</td>
<td>total # tickets</td>
</tr>
<tr>
<td>( n )</td>
<td># lotteries</td>
</tr>
</tbody>
</table>

**Example List-based Lottery**

\[ T = 20 \]

\[
\begin{array}{cccc}
10 & 2 & 5 & 1 & 2 \\
\end{array}
\]

Summing: 10 12 17

Random(0, 19) = 15
Bells and Whistles

- Ticket transfers - objects that can be explicitly passed in messages
  - Can be used to solve priority inversions
- Ticket inflation
  - Create more - used among mutually trusting clients to dynamically adjust ticket allocations
- Currencies - “local” control, exchange rates
- Compensation tickets - to maintain share
  - use only $f$ of quantum, ticket inflated by $1/f$ in next

Kernel Objects

```
<table>
<thead>
<tr>
<th>ticket</th>
<th>Backing tickets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000 base</td>
<td>amount currency</td>
</tr>
<tr>
<td>C_name</td>
<td>Active amount</td>
</tr>
<tr>
<td>300</td>
<td>Issued tickets</td>
</tr>
</tbody>
</table>
```
Example List-based Lottery

\[ T = 3000 \text{ base} \]

\[
\begin{array}{cccc}
10 & \text{task1} & 2 & \text{bob} & 5 & \text{task3} & 1 & \text{bob} \\
	ext{thread1} & \text{alice} & 200 & \text{task2} & 200 & \text{alice} & 200 & \text{bob} \\
\end{array}
\]

Random(0, 2999) = 1500

Exchange rate: 1 bob = 20 base
Compensation

- A holds 400 base, B holds 400 base
- A runs full 100msec quantum, B yields at 20msec
- B uses 1/5 allotted time
  Gets compensation ticket valued at $400/(1/5) = 2000$ base at next lottery

Ticket Transfer

- Synchronous RPC between client and server
- create ticket in client’s currency and send to server to fund it’s currency
- on reply, the transfer ticket is destroyed
Control Scenarios

- Dynamic Control
  Conditionally and dynamically grant tickets
  Adaptability
- Resource abstraction barriers supported by currencies. Insulate tasks.

Other Kinds of Resources

Control relative waiting times for mutex locks.
- Mutex currency funded out of currencies of waiting threads
- Holder gets inheritance ticket in addition to its own funding, passed on to next holder (resulting from lottery) on release.
Scheduling: Beyond "Ordinary" Uniprocessors

- Multiprocessors
  - Co-scheduling and gang scheduling
  - Hungry puppy task scheduling
  - Load balancing
- Networks of Workstations
  - Harvesting Idle Resources - remote execution and process migration
- Laptops and mobile computers
  - Power management to extend battery life, scaling processor speed/voltage to tasks at hand, sleep and idle modes.

RR and System Throughput

On a multiprocessor, RR may improve throughput under light load:

The scenario: three salmon steaks must cook for 5 minutes per side, but there's only room for two steaks on the hibachi.

30 minutes worth of grill time needed: steaks 1, 2, 3 with sides A and B.

FCFS: steaks 1 and 2 for 10 minutes, steak 3 for 10 minutes.

Completes in 20 minutes with grill utilization a measly 75%.
RR and System Throughput

*RR*: 1A and 2A...flip...1B and 3A...flip...2B and 3B.
Completes in three quanta (15 minutes) with 100% utilization.
RR may speed up parallel programs if their inherent parallelism is poorly matched to the real parallelism.
E.g., 17 threads execute for $N$ time units on 16 processors.

---

Multiprocessor Scheduling

What makes the problem different?

- **Workload consists of parallel programs**
  - Multiple processes or threads, synchronized and communicating
  - Latency defined as last piece to finish.
- **Time-sharing and/or Space-sharing**
  (partitioning up the Mp nodes)
  - Both *when* and *where* a process should run
Architectures

Symmetric mp

NUMA

Memory

Node 0

Node 1

Node 2

Node 3

Interconnect

Effect of Workload

Impact of load-balancing on latency

Consider set of processes: 5, 5, 4, 4, 3, 3

3 processors

- If unrelated: (SJF)
  \[
  \text{avg response time} = \frac{3 + 3 + 4 + 7 + 8 + 9}{6} = 5.66
  \]

- If 2 tasks, each 5, 4, 3 (no dependencies)
  \[
  \text{avg response time (SPF)} = \frac{8 + 9}{2} = 8.5
  \]
  \[
  \text{avg response time (LPF)} = \frac{8 + 8}{2} = 8
  \]
**Affinity Scheduling**

- Where (on which node) to run a particular thread during the next time slice?
- Processor’s POV: favor processes which have some residual state locally (e.g. cache)
- What is a useful measure of affinity for deciding this?
  - Least intervening time or intervening activity (number of processes here since “my” last time) *
  - Same place as last time “I” ran.
  - Possible negative effect on load-balance.

**Processor Partitioning**

- Static or Dynamic
- Process Control (Gupta)
  - Vary number of processors available
  - Match number of processes to processors
  - Adjusts # at runtime.
  - Works with task-queue or threads programming model
  - Impact on “working set”
Process Control Claims
Typical speed-up profile

- Lock contention, memory contention, context switching, cache corruption

Outline for 10/1

- Objective for today’s lecture: Advanced topics in scheduling continued
- Administrivia
Co-Scheduling

John Ousterhout (Medusa OS)

- Time-sharing model
- Schedule related threads simultaneously

Why?
How?
  - Local scheduling decisions after some global initialization (Medusa)
  - Centralized (SGI IRIX)

Effect of Workload
Impact of communication and cooperation

Issues:
- context switch
+ common state
- lock contention
+ coordination
**CM*’s Version**

- Matrix S (slices) x P (processors)
- Allocate a new set of processes (task force) to a row with enough empty slots
- Schedule: Round robin through rows of matrix
  - If during a time slice, this processor’s element is empty or not ready, run some other task force’s entry in this column - backward in time (for affinity reasons and purely local “fall-back” decision)

**Design**

- Determining how many processors an application should have
  - Centralized server, with system calls for process status info (user-level) (kernel implementation would be desirable, but...)
- Controlling the number of processes in an application
  - suspend and resume are responsibility of runtime package of application
Networks of Workstations

What makes the problem different?
• Exploiting otherwise “idle” cycles.
• Notion of *ownership* associated with workstation.
• Global truth is harder to come by in wide area context

Harvesting Idle Cycles

• Remote execution on an idle processor in a NOW (network of workstations)
  – Finding the idle machine and starting execution there. Related to load-balancing work.
• Vacating the remote workstation when its user returns and it is no longer idle
  – Process migration
Issues

• Why?
• Which tasks are candidates for remote execution?
• Where to find processing cycles? What does “idle” mean?
• When should a task be moved?
• How?

Motivation for Cycle Sharing

• Load imbalances. Parallel program completion time determined by slowest thread. Speedup limited.
• Utilization. In trend from shared mainframe to networks of workstations → scheduled cycles to statically allocated cycles
  – “Ownership” model
  – Heterogeneity
Which Tasks?

• Explicit submission to a “batch” scheduler (e.g., Condor) or Transparent to user.

• Should be demanding enough to justify overhead of moving elsewhere. Properties?

• Proximity of resources.
  – Example: move query processing to site of database records.
  – Cache affinity

Finding Destination

• Defining “idle” workstations
  – Keyboard/mouse events? CPU load?

• How timely and complete is the load information (given message transit times)?
  – Global view maintained by some central manager with local daemons reporting status.
  – Limited negotiation with a few peers
  – How binding is any offer of free cycles?

• Task requirements must match machine capabilities
When to Move

• At task invocation. Process is created and run at chosen destination.
• Process migration, once task is already running at some node. State must move.
  – For adjusting load balance (generally not done)
  – On arrival of workstation’s owner (vacate, when no longer idle)

How - Negotiation Phase

• Condor example: Central manager with each machine reporting status, properties (e.g. architecture, OS). Regular match of submitted tasks against available resources.
• Decentralized example: select peer and ask if load is below threshold. If agreement to accept work, send task. Otherwise keep asking around (until probe limit reached).
### How - Execution Phase

- **Issue - Execution environment.**
  - File access - possibly without user having account on destination machine or network file system to provide access to user’s files.
  - UIDs?
- **Remote System Calls (Condor)**
  - On original (submitting) machine, run a “shadow” process (runs as user)
  - All system calls done by task at remote site are “caught” and message sent to shadow.

### Remote System Calls

<table>
<thead>
<tr>
<th>Submitting machine</th>
<th>Executing machine</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OS Kernel</strong></td>
<td><strong>OS Kernel</strong></td>
</tr>
<tr>
<td>Shadow Remote syscall code</td>
<td>Remote Job User code</td>
</tr>
<tr>
<td>Regular syscall stubs</td>
<td>Remote syscall stubs</td>
</tr>
</tbody>
</table>
How - Process Migration

Checkpointing current execution state (both for recovery and for migration)
- Generic representation for heterogeneity?
- Condor has a checkpoint file containing register state, memory image, open file descriptors, etc. Checkpoint can be returned to Condor job queue.
- Mach - package up processor state, let memory working set be demand paged into new site.
- Messages in-flight?

Idleness is Powerful

? transition

High power cost X time

Busy

transition

Low power X time cost

Idle
Dynamic Voltage Scaling

- The question: at what clock rate/voltage should the CPU run in the next scheduling interval?

- Voltage scalable processors
  - StrongARM SA-2 (500mW at 600MHz; 40mW at 150MHz)
  - Speedstep Pentium III
  - AMD Mobile K6 Plus
  - Transmeta

- Power is proportional to $V^2 \times F$
- Energy will be affected
  (+) by lower power,
  (-) by increased time

Interval Scheduling
(adjust clock based on past window, no process reordering involved)

- Weiser et. al.
- Algorithms (when):
  - Past
  - $\text{AVG}_n$
- Stepping (how much)
  - One
  - Double
  - Peg – min or max
- Based on unfinished work during previous interval
Implementation of Voltage Scheduling Algorithms

Issues:
• Capturing utilization measure
  – Start with no a priori information about applications and need to dynamically infer / predict behavior (patterns / “deadlines” / constraints?)
  – Idle process or “real” process – usually each quantum is either 100% idle or busy
  – $\text{AVG}_N$: weighted utilization at time $t$
    \[ W_t = \frac{NW_{t-1} + U_{t-1}}{N+1} \]
• Adjusting the clock speed
  – Idea is to set the clock speed sufficiently high to meet deadlines (but deadlines are not explicit in algorithm)

Based on Earliest Deadline First

• Dynamic algorithm
• Priorities are assigned to tasks according to the deadlines of their current request
• With EDF there is no idle time prior to an overflow
• For a given set of $m$ tasks, EDF is feasible iff
  \[ \frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_m}{T_m} \leq 1 \]
• If a set of tasks can be scheduled by any algorithm, it can be scheduled by EDF
lpARM System

- Speed-control register
- Processor cycle ctrs
- System sleep control

Figure 2: lpARM System Block Diagram

Intuition

\[ C_1 = 1 \]
\[ C_2 = 1 \]
Intuition

EventBarrier

- *EventBarrier* has a binary “memory”.
- It has a “broadcast” to notify all waiting threads of an event.
- The broadcast primitive waits until the event is handled.

EventBarrier::Wait()

*If the EventBarrier is not in the signaled state, wait for it.*

EventBarrier::Signal()

*Signal the event, and wait for all waiters/arrivals to respond.*

EventBarrier::Complete()

*Notify EventBarrier that caller’s response to the event is complete.*
*Block until all threads have responded to the event.*
The Moat Problem

- Travelers, knights, and troubadours arrive at the castle.
- The castle guard decides when to lower the bridge to allow the arrivals into the castle.
- If the bridge is down, new arrivals may enter immediately without waiting.
- The guard doesn’t raise the bridge if there are people on it.

- This can be solved easily using EventBarrier.
The Moat Problem with EventBarrier

EventBarrier gate;

/* Called by knights etc. */
void EnterCastle() {
    gate.Wait(); /* wait for gate to open (if necessary) */
    CrossBridge();
    gate.Complete(); /* tell the guard it’s OK to close gate */
}

void GuardThread() {
    while (TRUE) {
        /* twiddle thumbs */
        /* watch for arriving travelers */
        /* decide when to open gate */
        WaitForOrderToOpenGate();
        gate.Signal(); /* open gate, wait for travelers to cross, close gate */
        /* gate is closed */
    }
}

Highway 110 Problem

Highway 110 is a two-lane north-south road that passes across a one-lane bridge. A car can safely enter the bridge if and only if there are no oncoming cars on the bridge.

To prevent accidents, sensors installed at each end of the tunnel notify a controller computer when cars arrive or depart in either direction. The controller uses the sensor input to control signal lights at either end of the bridge.
Highway 110 with EventBarrier

EventBarrier north;
EventBarrier south;

void HeadingNorth {
    north.Wait();
    go across one-lane bridge;
    north.Complete();
}

void HeadingSouth() {
    south.Wait();
    go across one-lane bridge;
    south.Complete();
}

void BridgeScheduler() {
    while (TRUE) {
        north.Signal();
        south.Signal();
        ....
    }
}

Highway 110 with Locks and Condition Variables

OneVehicle(int direc) //direc is either 0 or 1; giving the direction in which the car is to cross
{
    ArriveBridge(direc);
    CrossBridge(direc);
    ExitBridge(direc);
}

ArriveBridge(dir)
{
    bridgeLock->Acquire();
    while (num_on_bridge != 0 &
        dir != direction)
        OKtogo[dir]->Wait();
    direction = dir;
    num_on_bridge++;
    bridgeLock->Release();
}

ExitBridge(dir)
{
    bridgeLock->Acquire();
    num_on_bridge--;
    if (num_on_bridge == 0) {
        direction = !dir;
        Oktogo[direction]->Broadcast();
        bridgeLock->Release();
    }