

## Outline for Today

- Real time scheduling
- Advanced topics in scheduling

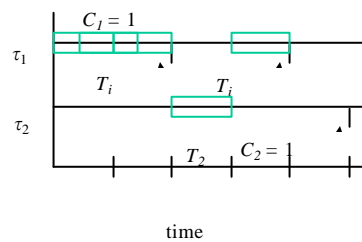
## Real Time Schedulers

- 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  $1/n$  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
- Run-time for each task is constant (max),  $C_i$
- Any non-periodic tasks are special

## Task Model



## Definitions

- **Deadline** is time of next request
- **Overflow** at time  $t$  if  $t$  is deadline of 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 largest response time.
  - Occurs when task is requested simultaneously with all tasks of higher priority

## 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 can not 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  $C_1/T_1 + C_2/T_2 + \dots + C_m/T_m \leq 1$
- 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  $f_i(t) = w_i / (\sum_{j \in A(t)} w_j)$
- 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  
 $S_i(t_0, t_1) = \int_{t_0}^{t_1} f_i(\tau) d\tau$

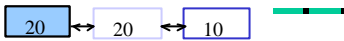
## Common Proportional Share Competitors

- 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)



Linux

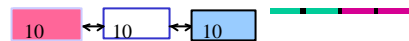
## Common Proportional Share Competitors

- 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

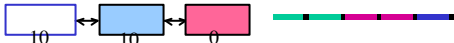
## Common Proportional Share Competitors

- 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

$$VT_A(t) = W_A(t) / S_A$$


- VFT – Virtual Finishing Time: VT a client would have after executing its next time quantum

- WFQ schedules by smallest VFT

- $E_A$  never below -1

$$VFT = 3/3$$

$$VFT = 3/2$$

$$VFT = 2/1$$

## 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  $p$ .
- *GetNextToRun* selects "winning ticket" randomly.
  - If  $\sum W_p = N$ , 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).

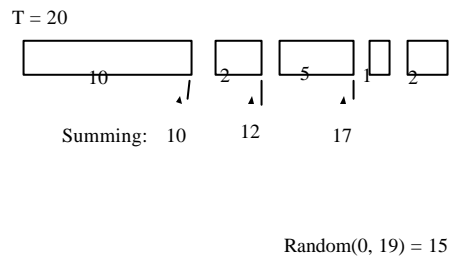
## Basic Idea

- 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  
 $E[w] = n p$  where  $p = t/T$
  - Response time (# lotteries to wait for first win)  
 $E[n] = 1/p$
- |     |                 |
|-----|-----------------|
| $w$ | # wins          |
| $t$ | # tickets       |
| $T$ | total # tickets |
| $n$ | # lotteries     |

## Example List-based Lottery



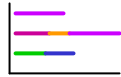
## Beyond "Ordinary" Uniprocessors

- 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.

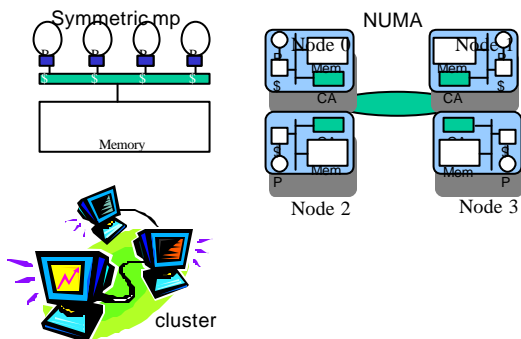
## Multiprocessor Scheduling

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



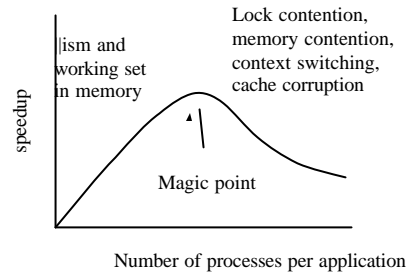
## Affinity Scheduling

- 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



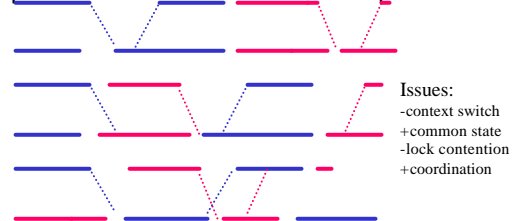
## Co-Scheduling

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



## 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)

## 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

41

## 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)

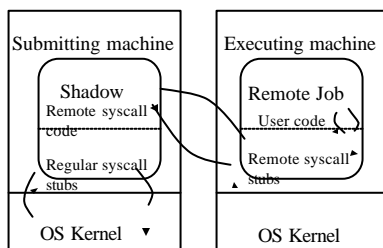
## How - Negotiation Phase

- 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

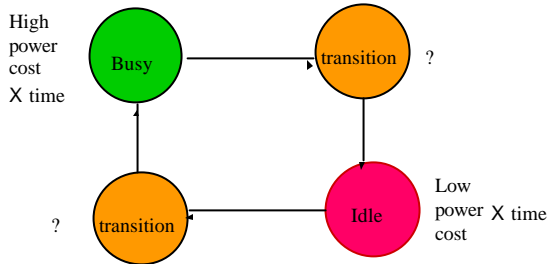


## How - Process Migration

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?

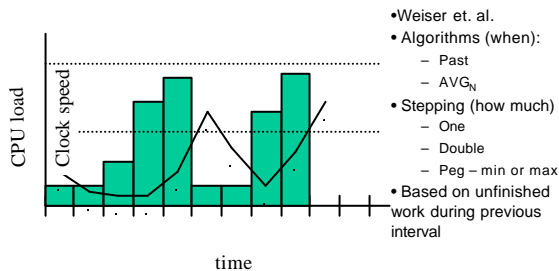
## Idleness is Powerful



## 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)



## Implementation of Voltage Scheduling Algorithms

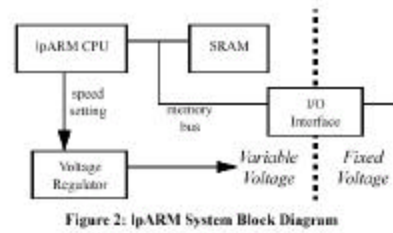
Issues:

- 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
  - $AVG_N$ : weighted utilization at time  $t$   
 $W_t = (NW_{t-1} + U_{t-1}) / (N+1)$
- 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  $C_1/T_1 + C_2/T_2 + \dots + C_m/T_m \leq 1$
- If a set of tasks can be scheduled by any algorithm, it can be scheduled by EDF

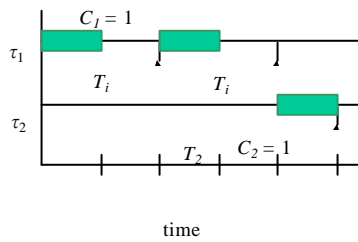
### IpARM System



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

Figure 2: IpARM System Block Diagram

### Intuition



### Intuition

