Multicore Performance #2
- Load Balancing -

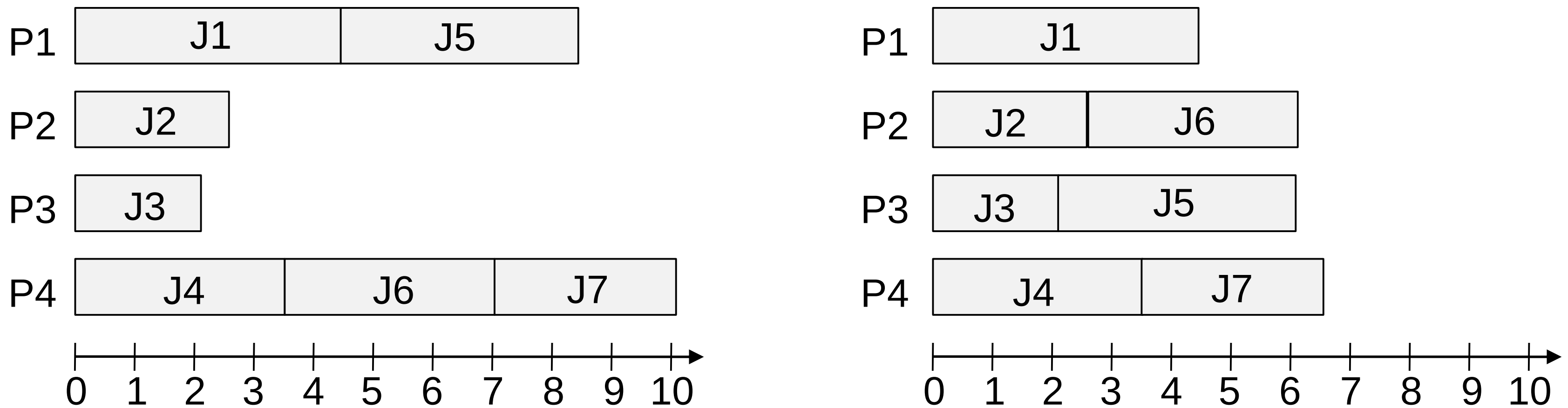
Minsoo Ryu

Department of Computer Science and Engineering
Hanyang University

Outline

Goal of Load Balancing

- **Distribute workload evenly across processors**
  - **To get optimal resource utilization, maximize throughput, and minimize response time**



Unbalanced schedule

Well balanced schedule

Static Balancing vs. Dynamic Balancing

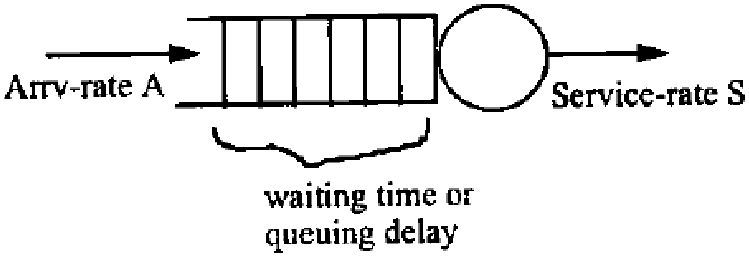
- Static balancing (partitioned sched. w/o migration)
  - Processes are statically assigned to processors during program compilation or loading
  - Works well when there is not much variation in the workload
- Dynamic balancing (partitioned sched. with migration)
  - Running processes are moved to remote processors
  - Works well when loads may vary significantly during runtime
  - But the cost of collecting and maintaining load information and process migration is high
- Adaptive balancing
  - Special type of dynamic load balancing
  - The algorithm may change depending on the system state
  - e.g.) If the system load is very high, it may not even attempt to collect load information

Load Balancing Policies

- Transfer policy
  - Decides whether a processor is eligible for load balancing
  - Usually based on a threshold such as queue length
- Selection policy
  - Decides which task should be moved
- Location policy
  - Decides where to send or receive the task
  - Typical approaches are polling, guessing, or random
- Information policy
  - Decide when (how often) trigger collection of system load information and where to collect them
  - On demand, periodic, or state change driven

Stability Condition

- **Without load balancing**
  - **If A < S, the system is stable**
  - **Otherwise, it is unstable**



- **With load balancing**
  - **If A+LB < S, the system is stable**
    - LB is the overhead due to load balancing
  - **Otherwise, it is unstable**
  - **If the algorithm leads to thrashing, the system is unstable**
    - When a task arrives at a processor 1, it gets transferred to processor 2
    - The task may get transferred to another processor 3
    - It may keep getting transferred from processor to processor

Sender-Initiated Algorithms

- Can be viewed as "work sharing"
- Transfer policy
  - Use a threshold policy on the ready queue length |Q|
  - If a task arrives and makes |Q| > T, the processor becomes a sender
- Selection policy
  - Consider only newly arrived tasks
- Location policy
  - Different algorithms (described on the next slide)
- Information policy
  - Demand driven

Location Policies

- Random LP
  - Choose a remote processor randomly
  - No overhead of collecting information
  - Still, better than no load balancing
- Threshold LP
  - Select remote processors randomly
  - But check if $|Q| < T$
- Shortest queue LP
  - Select *k* processors at random
  - Send to the processor with smallest $|Q|$
  - More overhead than threshold LP, but marginal improvement

Receiver-Initiated Algorithms

- Can be viewed as “work stealing”
- Transfer policy
  - Use a threshold policy on the ready queue length |Q|
  - If a task departs and makes |Q| < T, the processor becomes a receiver
- Selection policy
  - Consider only newly arrived tasks
- Location policy
  - Can choose randomly or based on the queue length |Q|
- Information policy
  - Demand driven

Symmetrically-Initiated Algorithms

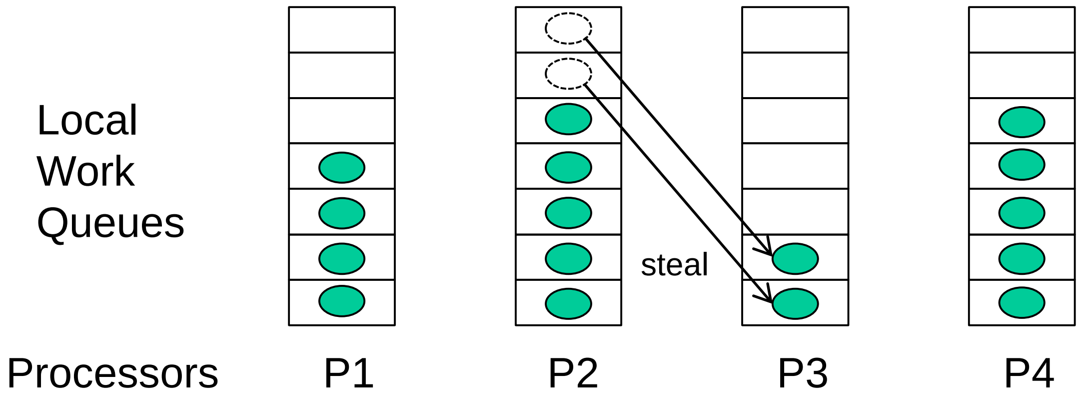
- Combine the previous two algorithms

- Transfer policy
  - Use a double threshold <Lower, Upper>
  - If |Q| > Upper, the processor becomes a sender
  - If |Q| < Lower, the processor becomes a receiver

- Location policy
  - Sender-initiated: broadcasts a “too high” message, and waits for a reply from a receiver
  - Receiver-initiated: broadcasts a “too low” message, and waits for a reply from a sender
  - If no reply has been received within timeout, send a “system load is high/low” message and change the threshold

Work Stealing

- Work stealing is known to be a simple but very effective approach to load balancing
  - Each processor maintains a local work queue
  - If a processor runs out of jobs, it steals from other processor



  - Work stealing is a receiver-initiated approach

Variations of Work Stealing

- Threshold-based work stealing
  - When a processor becomes idle, it randomly chooses another processor and steals a job only if the victim's queue contains more than a threshold number of tasks
- Probabilistic work stealing
  - Whenever a processor accesses its local work queue, it performs a balancing operation with probability inversely proportional to the size of its work queue (with probability 1/m where m is the length of the work queue)

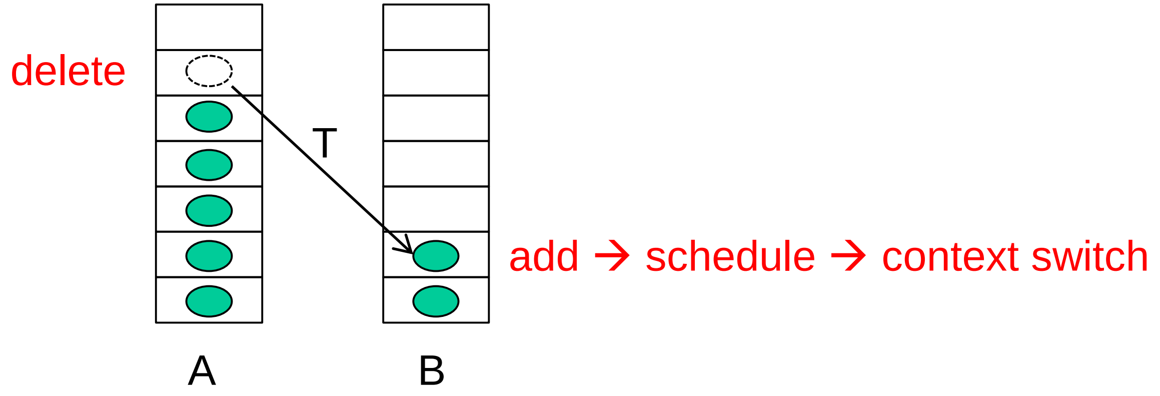
Overhead of Load Balancing

- Information collection
  - Can be done via either shared memory access or inter-processor communication
  - This causes minor overhead when compared to the task migration overhead

- Task migration
  - Scheduling
  - Cold cache effect

Scheduling for Thread Migration

- To move thread T from core A to core B
  - **W need to access and update the kernel's scheduling data structures**
    - Deleting T from A's ready queue
    - Adding T to B's ready queue
  - **The scheduler on B needs to perform scheduling**
    - This would also involve context switching



Cold Cache Effect

- **Thread migration requires transferring program states**
  - **Register state, TLB state, and branch predictor state**

- **The largest overhead is caused by cache states**
  - **A primary mechanism is demand-fetching**
    - Executing the thread on the new core causes demand misses
    - This fills caches slowly (cold cache effect)


Q&A