CSE 3320
Operating Systems
Multiprocessor Scheduling

Jia Rao

Department of Computer Science and Engineering

http://ranger.uta.edu/~jrao

Recap of the Last Class

- Basic scheduling policies on uniprocessors
  - First Come First Serve
  - Shortest Job First
  - Round Robin
  - Priority scheduling
  - Multilevel feedback queue

Time-sharing:
Which thread should be run next ?

Multiprocessor Scheduling

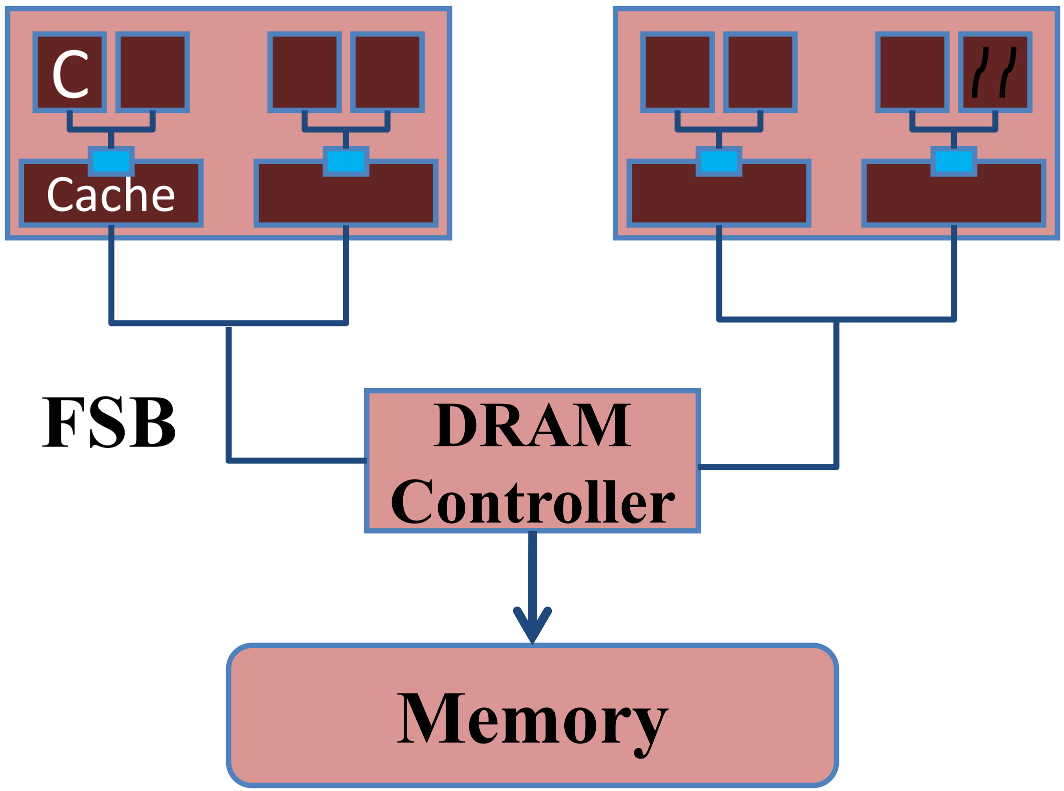
- Two-dimension scheduling
  - Time-sharing on each processor
  - Load-balancing among multiple processors

Which thread to run and where ?

- Several issues
  - Why load balancing ? → take advantage of parallelism
  - Simple time-sharing ? → No, may need to consider a group of thds
  - Are all processors/cores equal ? → No, cache affinity, memory Locality, and cache hotness make them different

Multiprocessor Hardware

- Uniform memory access (UMA)

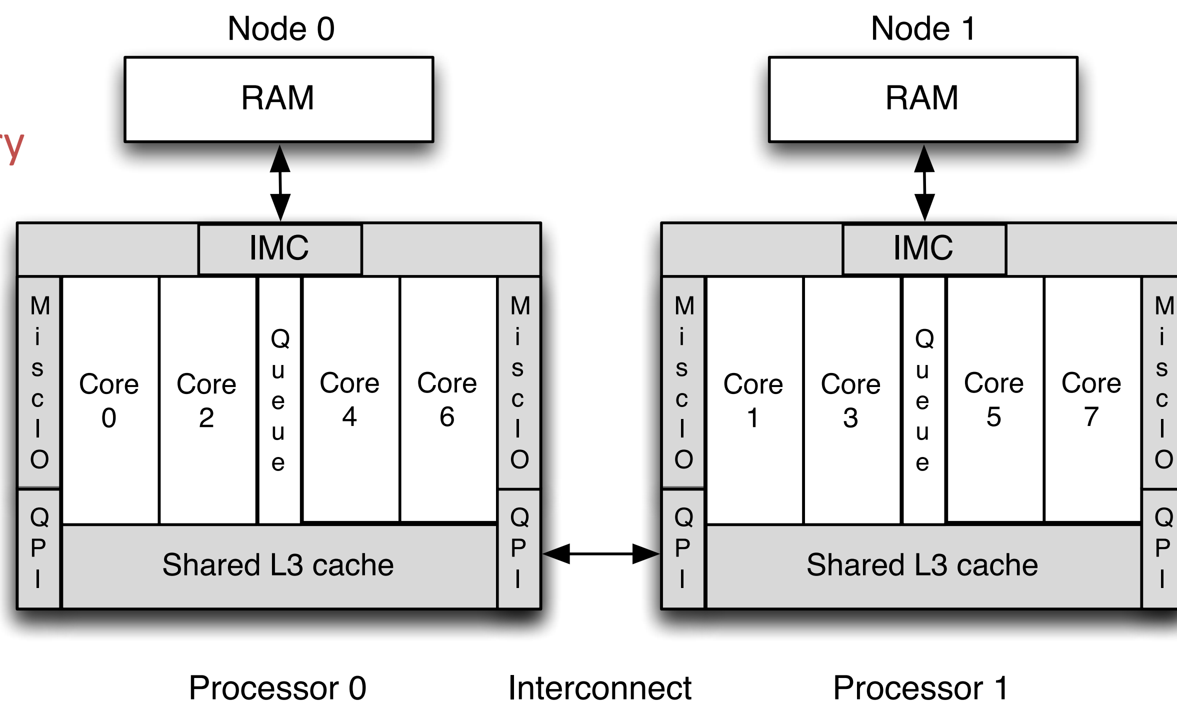


A schematic view of Intel Core 2

Multiprocessor Hardware (cont')

- Non-uniform memory access (NUMA)

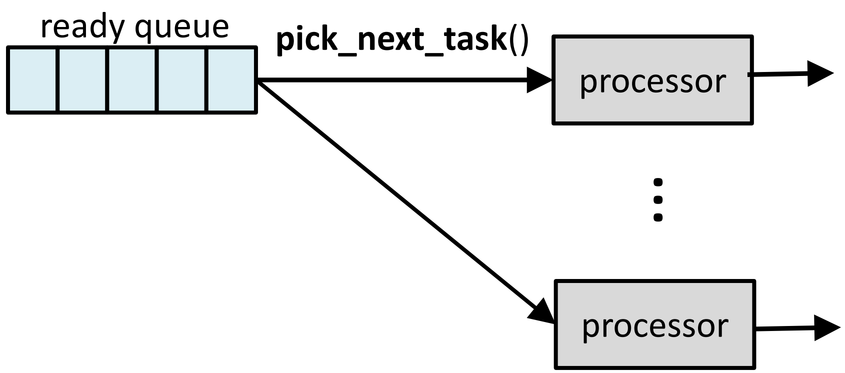
1. Local v.s. remote memory
2. Cache sharing
   1. Constructive
   2. Destructive



A schematic view of Intel Nehalem

Ready Queue Implementation

- A single system-wide ready queue



Pros:

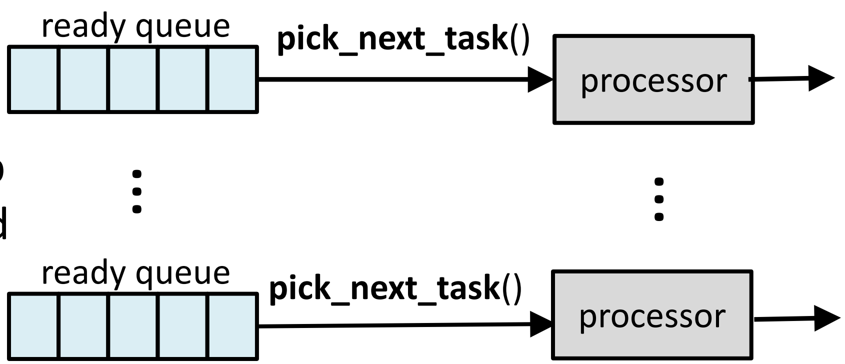
1. Easy to implement
2. Perfect load balancing

Cons:

1. Scalability issues due to centralized synchronization
2. High overhead and low efficiency
    1. Hard to maintain cache hotness

Ready Queue Implementation (cont')

- Per-CPU ready queue



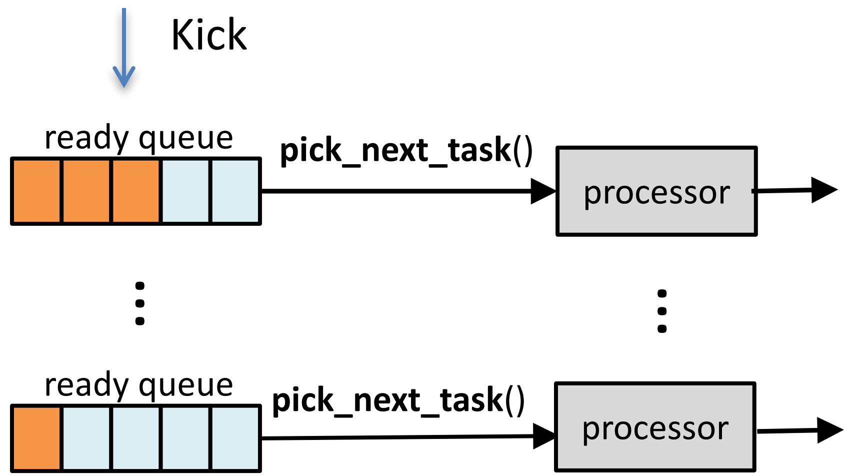
Load balancing: keep queue sizes balanced

Pros:
1. Scalable to many CPUs
2. Easy to maintain cache hotness

Cons:
1. More complex to implement
   1. Push model v.s. pull model
2. Not perfect load balancing → not always balanced

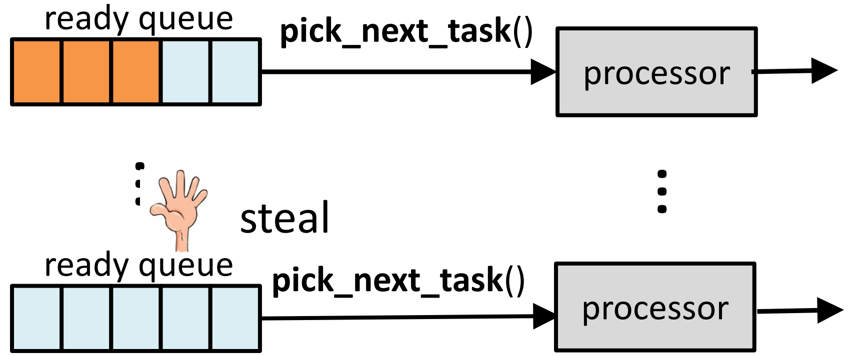
Push Model v.s. Pull Model

- Push model



Every a while, a kernel thread checks load imbalance and move threads

- Pull model



Whenever a queue becomes empty, steal a thread from non-empty queues

Both are widely used

Scheduling Parallel Programs

- A parallel job
  - A collection of processes/threads that cooperate to solve the same problem
  - Scheduling matters in overall job completion time
- Why scheduling matters ?
  - Synchronization on shared data (mutex)
  - Causality between threads (producer-consumer)
  - Synchronization on execution phases (barrier)

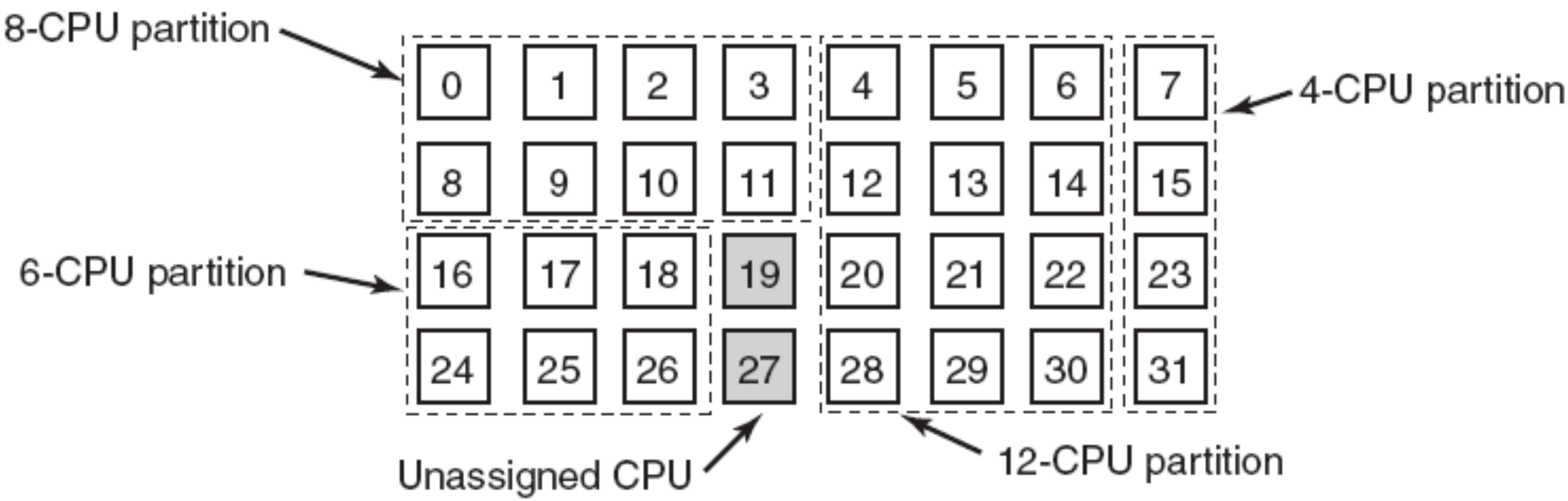
The slowest thread delays the entire job

Space Sharing

- Divide processor into groups
  - Dedicate each group to a parallel job
  - No preemption before job completion

Pros:
1. Highly efficient, low overhead
2. Strong affinity

Cons:
1. Highly inefficient, cycle waste
2. inflexible



Time Sharing: Gang or Co-Scheduling

- Each processor runs threads from multiple jobs
  - Groups of related threads are scheduled as a unit, a gang
  - All CPUs perform context switch together

CPU

Time slot	0	1	2	3	4	5
0	A_0	A_1	A_2	A_3	A_4	A_5
1	B_0	B_1	B_2	C_0	C_1	C_2
2	D_0	D_1	D_2	D_3	D_4	E_0
3	E_1	E_2	E_3	E_4	E_5	E_6
4	A_0	A_1	A_2	A_3	A_4	A_5
5	B_0	B_1	B_2	C_0	C_1	C_2
6	D_0	D_1	D_2	D_3	D_4	E_0
7	E_1	E_2	E_3	E_4	E_5	E_6

Gang scheduling (stricter) > co-scheduling

Summary

- Multiprocessor hardware
- Two implementation of the ready queue
  - A single queue v.s. multiple queues
- Load balancing
  - Push model v.s. Pull model
- Parallel program scheduling
  - Space sharing v.s. time sharing
- Additional practice
  - See the load balancer part in
    - http://www.scribd.com/doc/24111564/Project-Linux-Scheduler-2-6-32
  - See LINUX_SRC/kernel/sched.c
    - Function `load_balance` and `pull_task`