Lecture 4: Process scheduling
Operating Systems – EDA093/DIT401

Vincenzo Gulisano
vincenzo.gulisano@chalmers.se

UNIVERSITY OF GOTHENBURG

What to read (main textbook)

- Chapter 2.4, 8.1.1, 8.1.2, 8.1.4, 10.3.4, 11.4.1

(extra facultative reading: 5.1-5.7, 1.10 from Silberschatz Operating System Concepts)

Objectives

- Get deeper into processes, threads and their scheduling / execution
- Discuss different types of systems (batch/interactive/real-time)
- Discuss challenges of multi-processor/multi-core architectures

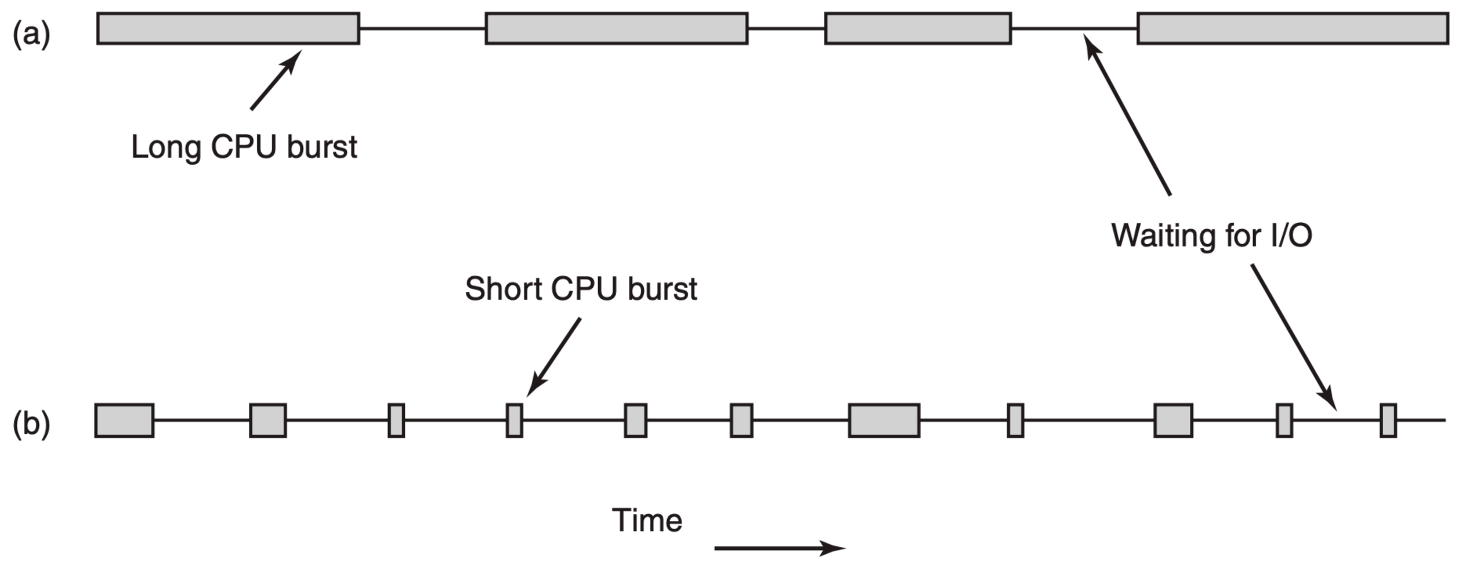
Agenda

- Introduction
- Batch / Interactive / Real-time systems scheduling
- Processes vs. Threads scheduling
- Multiprocessor hardware
  - Why does it complicate the matter?
- Multiprocessor scheduling
  - Time sharing
  - Space sharing
  - Gang scheduling

Agenda

- Introduction
- Batch / Interactive / Real-time systems scheduling
- Processes vs. Threads scheduling
- Multiprocessor hardware
  - Why does it complicate the matter?
- Multiprocessor scheduling
  - Time sharing
  - Space sharing
  - Gang scheduling

Introduction



- 2 types of processes:
  - a) CPU-bound (or compute-bound)
  - b) I/O bound
- Notice: I/O does not mean I/O takes a lot, it means few CPU cycles in-between I/O calls

When can the OS take scheduling decisions?

- Process creation
- Process termination
- I/O request
- I/O interrupt
- Elapsed time

Non-preemptive

preemptive

Agenda

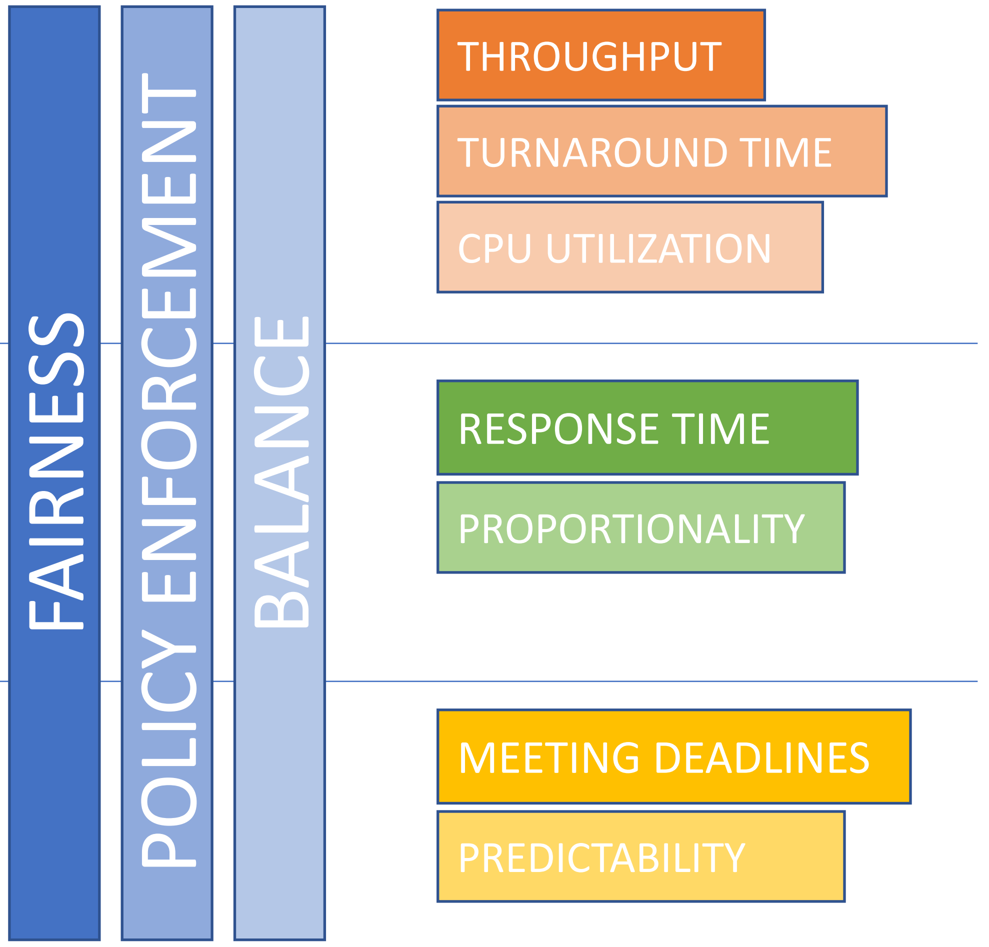
- Introduction
- Batch / Interactive / Real-time systems scheduling
- Processes vs. Threads scheduling
- Multiprocessor hardware
  - Why does it complicate the matter?
- Multiprocessor scheduling
  - Time sharing
  - Space sharing
  - Gang scheduling

Scheduling algorithms – Categories and goals

- **Batch**
  ”business-world” applications, data analysis. Appropriate for non-preemptive

- **Interactive**
  Many users that need responsiveness, requiring preemptive scheduling

- **Real-time**
  for short-lived, short-cycle processes with hard/soft deadlines



… Let’s discuss <u>some</u> scheduling algorithms for <u>some</u> of these categories [read others in book]…

Batch systems – scheduling

Algorithm	+	-
First-Come/First-Serve (non-preemptive)	Easy, fair	Possibly inefficient (esp. for I/O bound processes)
Shortest Job First (non-preemptive)	Optimal for turnaround	Starvation + need to know runtime
Shortest Remaining Time Next (preemptive)	New short jobs get good service	Starvation + need to know runtime

Example of inefficient first-come/first-serve scheduling

- Process 0 (CPU-bound): 1 I/0 every 1 sec of computations, 1000 sec to finish
- Processes 1…1000 (I/O bound): need to perform 1000 I/Os
- FCFS: Processes 1…1000 get to perform 1 I/O every second. Hence, they end in 1000 seconds (>16 minutes)
- Preempting Process 0 every 10 ms, they could complete in 10 seconds…

Interactive Systems - scheduling

- Round-robin ⇦
- Priority scheduling ⇦
- Multiple queues
- Shortest process next
- Guaranteed scheduling
- Lottery scheduling ⇦
- Fair-share scheduling

Round-Robin

- Quantum: time-interval during which the process can run

- Process still running at the end of the quantum? Preempt!
- Simple to implement (keep a list…)

- Challenge: what’s the right quantum length?
    - Too short $\rightarrow$ high overhead
    - Too long $\rightarrow$ responsiveness (e.g., 50th process of a batch scheduled in round-robin with quantum 100ms waits 5 secs to start… what if it was the shortest I/O-bound of the 50 processes???)

Priority scheduling

- Not all processes are equally important, processes with higher priority should be prioritized
- Priorities:
  - Static (by OS or user)
  - Dynamic (by OS or user)
- Priority can be combined with round-robin → priority classes

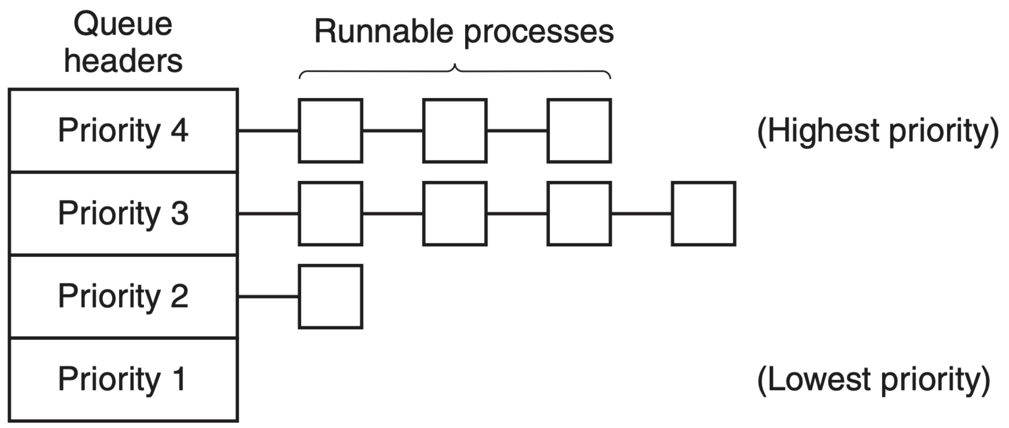


Figure 2-43. A scheduling algorithm with four priority classes.

…In the previous example (FCFS batch) I/O could have higher priority than CPU-bound…

Lottery scheduling

- Alternative to priority scheduling that still gives more resources to some processes rather than others

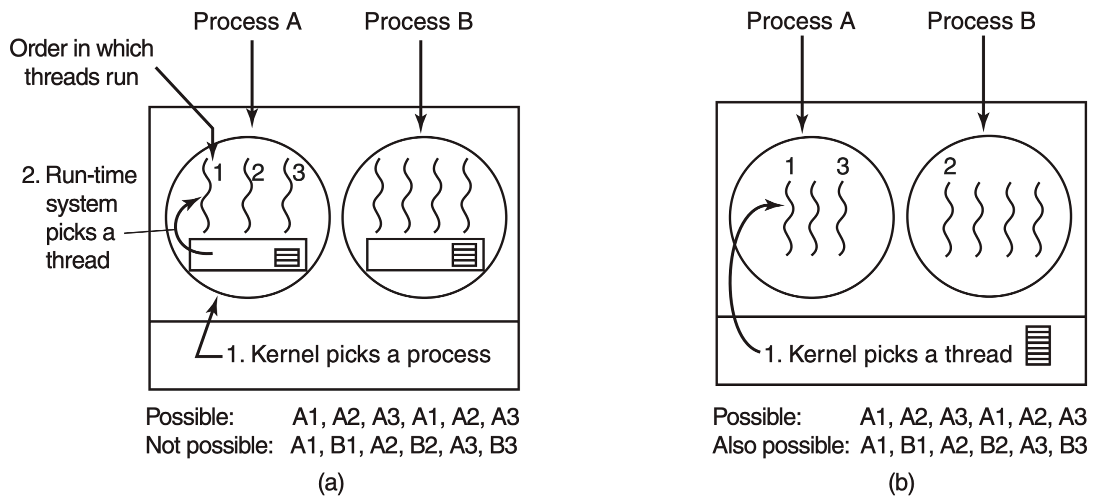
- Processes get “lottery tickets”.
- Next process to run is the one holding the next randomly chose ticket.

- Easier to map portions of resources to give to a process (i.e., portion of tickets to give) than with priority scheduling

Agenda

- Introduction
- Batch / Interactive / Real-time systems scheduling
- Processes vs. Threads scheduling
- Multiprocessor hardware
  - Why does it complicate the matter?
- Multiprocessor scheduling
  - Time sharing
  - Space sharing
  - Gang scheduling

User-level vs. Kernel-level threads



Process A
Process B
Order in which threads run
2. Run-time system picks a thread
1. Kernel picks a process
1 2 3
Possible: A1, A2, A3, A1, A2, A3
Not possible: A1, B1, A2, B2, A3, B3
(a)
Process A
Process B
1 3
2
1. Kernel picks a thread
Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3
(b)

Figure 2-44. (a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).

	User-level	Kernel-level
+	- Inter-quantum thread switch is extremely fast (no real context switch) - Can employ application specific scheduler	Process can keep running even if some of its thread perform I/O
-	A thread blocking on I/O means the entire process does	Thread switch costs more (but OS knows inter-process thread switch might cost more than intra-process one)

Agenda

- Introduction
- Batch / Interactive / Real-time systems scheduling
- Processes vs. Threads scheduling
- Multiprocessor hardware
  - Why does it complicate the matter?
- Multiprocessor scheduling
  - Time sharing
  - Space sharing
  - Gang scheduling

Multiprocessor hardware– Why complicated?

Caches

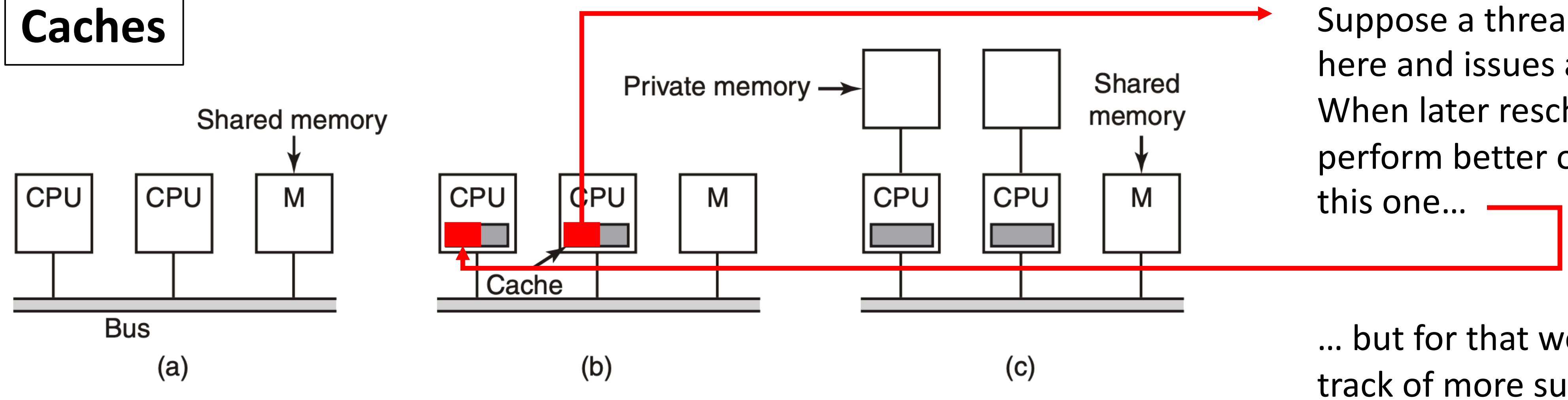


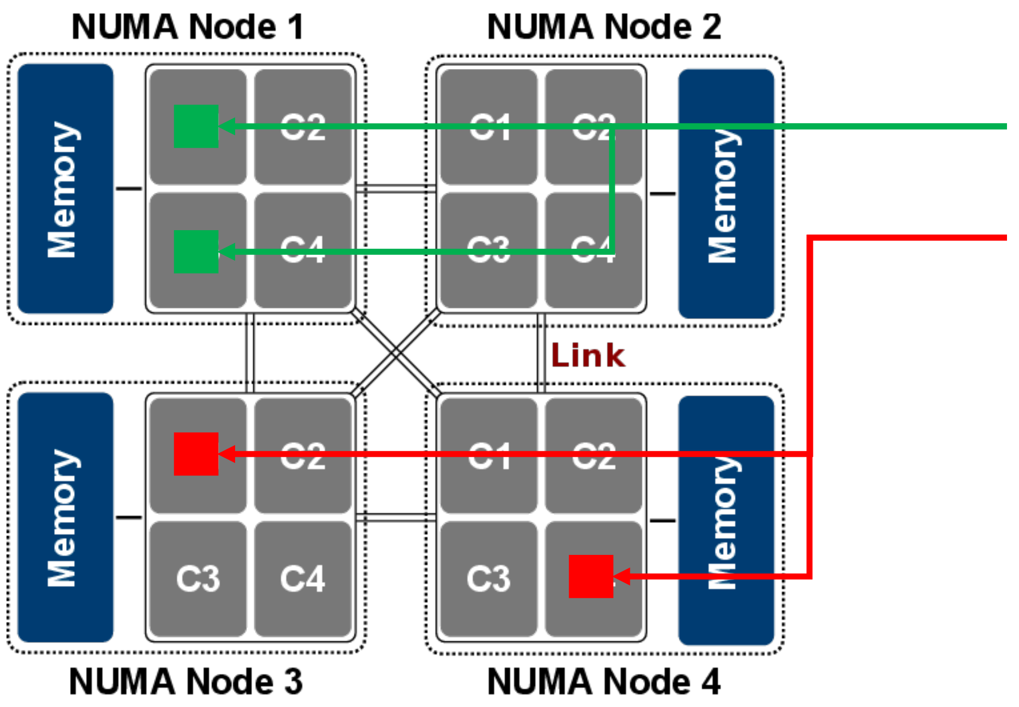
Figure 8-2. Three bus-based multiprocessors. (a) Without caching. (b) With caching. (c) With caching and private memories.

Suppose a thread has some data here and issues an I/O request. When later rescheduled, it might perform better on this CPU than on this one…

… but for that we need to keep track of more such information and make it part of the scheduling process.

Multiprocessor hardware– Why complicated?

NUMA architectures



Suppose two threads (producer/consumer) are scheduled at the same time…

Scheduling on the same socket will perform better than…

Scheduling on two different sockets

To complicate a bit further… how would the OS know 2 threads are producer/consumer?

Multiprocessor hardware– Why complicated?

Where to place the OS itself?

Each CPU its own OS

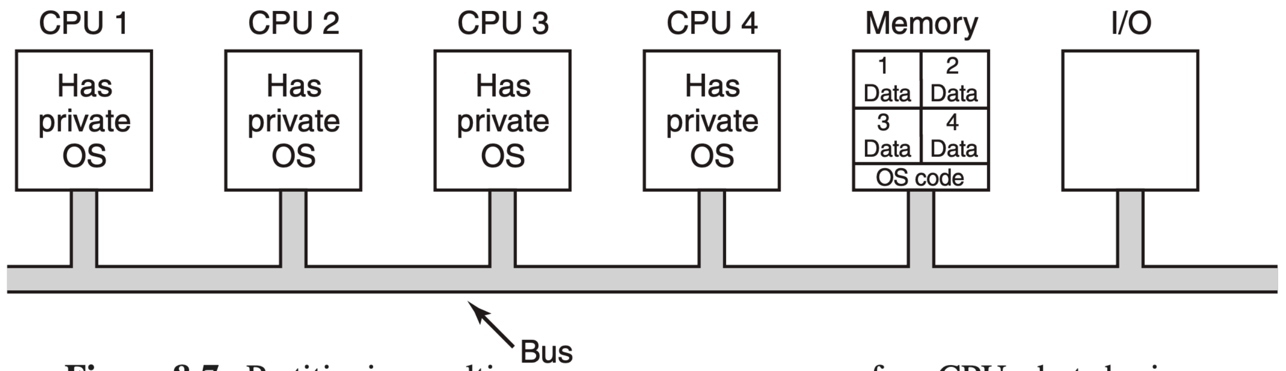


Figure 8-7. Partitioning multiprocessor memory among four CPUs, but sharing a single copy of the operating system code. The boxes marked Data are the operating system's private data for each CPU.

Might still be better than n separate computers

No sharing makes it simple, but also inefficient and possibly useless…

- Load can become imbalanced
- Data can become inconsistent (especially with buffers!)

Multiprocessor hardware– Why complicated?

Where to place the OS itself?

All system calls redirected to the Master CPU

… easy to bottleneck …

Master-Slave

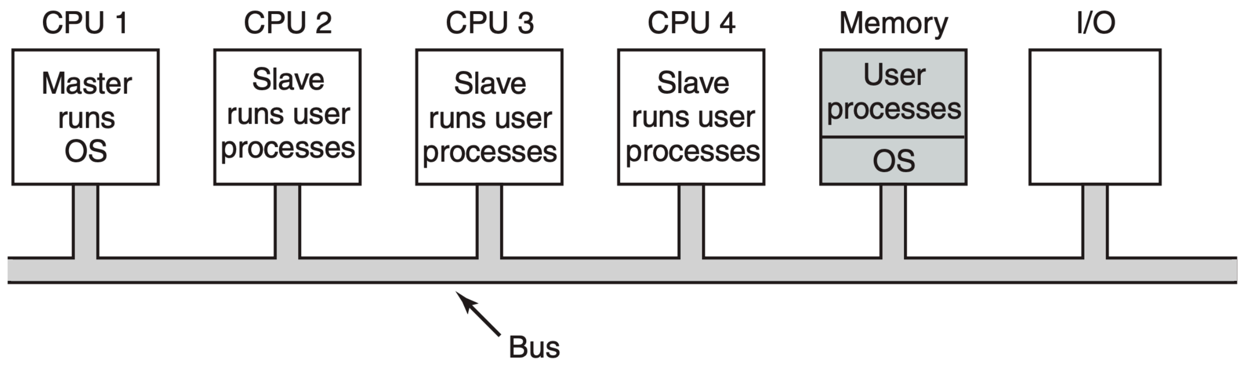

CPU 1
Master runs OS
CPU 2
Slave runs user processes
CPU 3
Slave runs user processes
CPU 4
Slave runs user processes
Memory
User processes
OS
I/O
Bus

Figure 8-8. A master-slave multiprocessor model.

Multiprocessor hardware– Why complicated?

Where to place the OS itself?

Symmetric multiprocessors

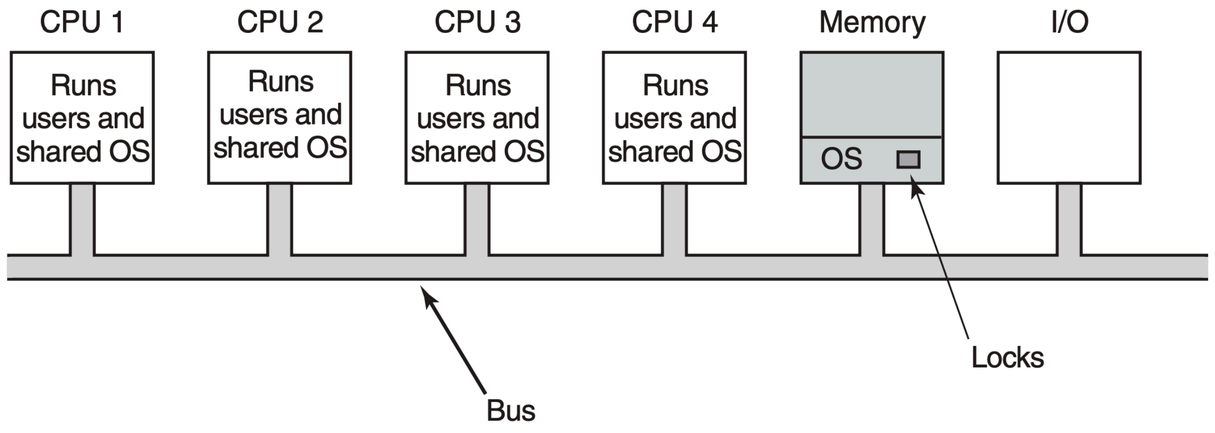


Figure 8-9. The SMP multiprocessor model.

Balances workload / resources

→ CONCURRENT ACCESS TO KERNEL!!!

Such dangerous!!!
Very pain!!!

2 threads could modify the same data structure at the same time.

Big lock? → Then it is basically master-slave

Critical regions / fine-grained parallelism? → better!
…but makes it hard to program (e.g., deadlocks…)

Agenda

- Introduction
- Batch / Interactive / Real-time systems scheduling
- Processes vs. Threads scheduling
- Multiprocessor hardware
  - Why does it complicate the matter?
- Multiprocessor scheduling
  - Time sharing
  - Space sharing
  - Gang scheduling

Time sharing

- Single system-wide data structure (or combination) for all ready threads

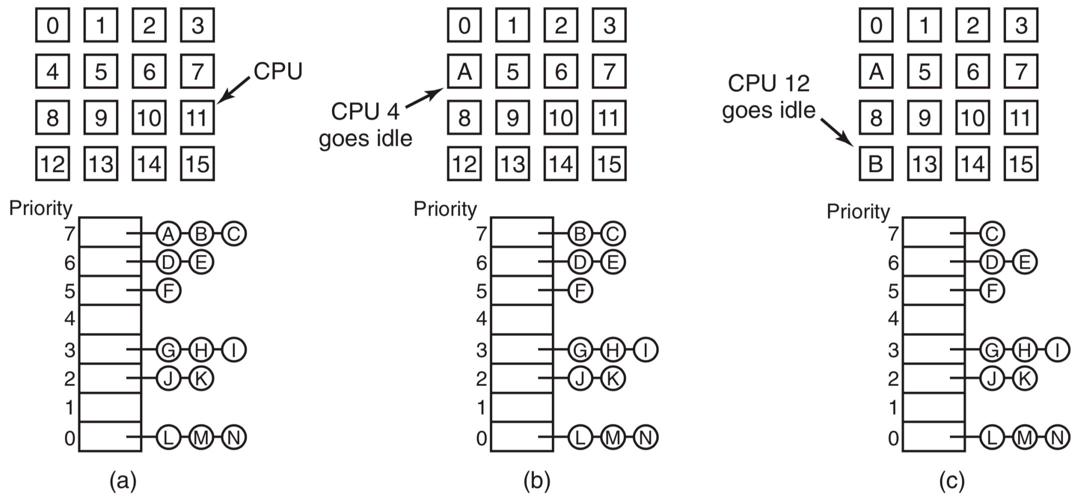


Figure 8-12. Using a single data structure for scheduling a multiprocessor.

+ Automatic load balancing

- Contention might bottleneck the system
- Still suffers from the “affinity problem”

Time sharing – two-level scheduling algorithm

- Each CPU has its collection of threads (assigned at creation time, in e.g. round-robin or least-loaded)
- Idle CPUs can still take threads from other CPUs if needed

- Benefits:
  - Load balancing
  - Cache affinity
  - Less contention

Space sharing

- When a set of related threads (e.g., from the same process) is created, the OS tries to schedule all of them at the same time (if enough CPUs are available).
- Thread issuing I/O still holds the CPU (inefficient…).

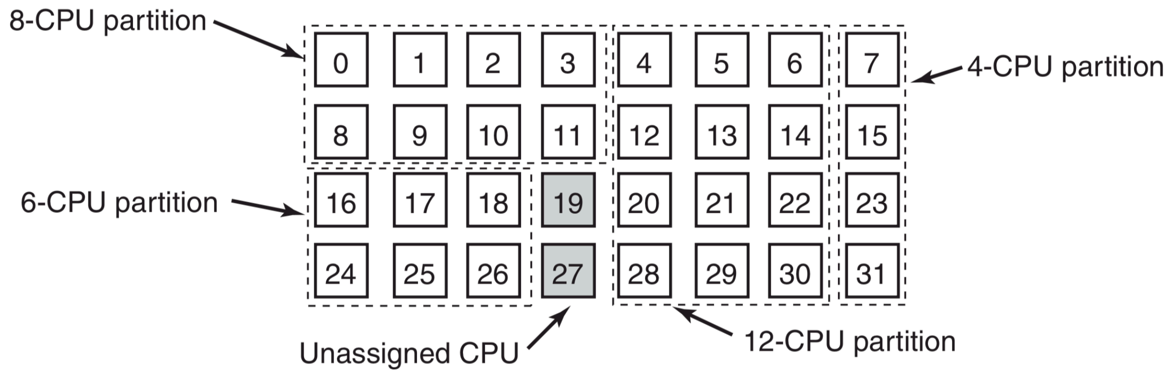
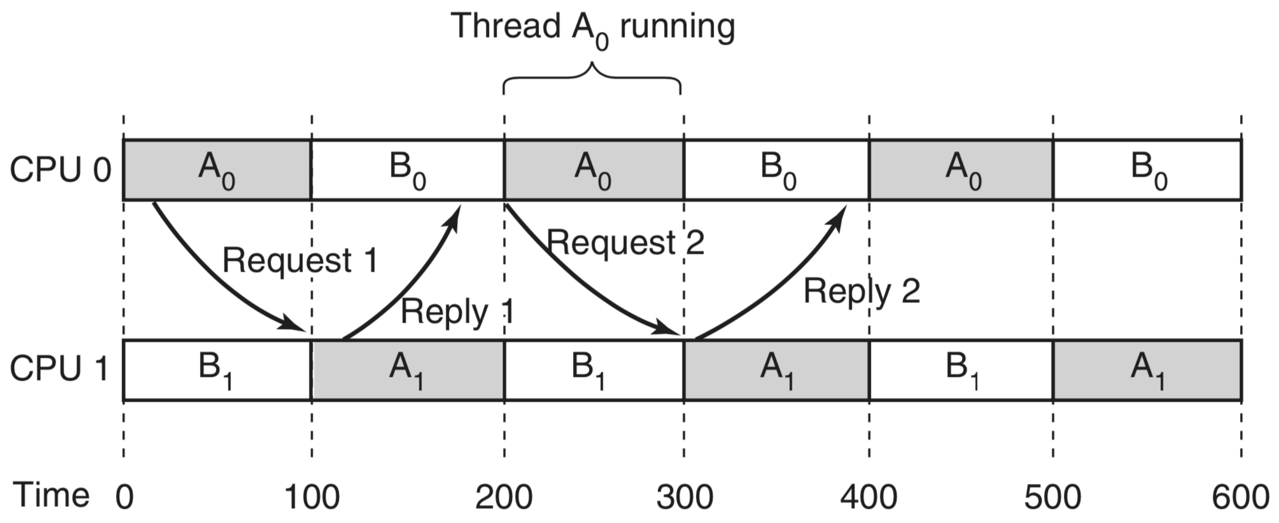


Figure 8-13. A set of 32 CPUs split into four partitions, with two CPUs available.

Gang scheduling

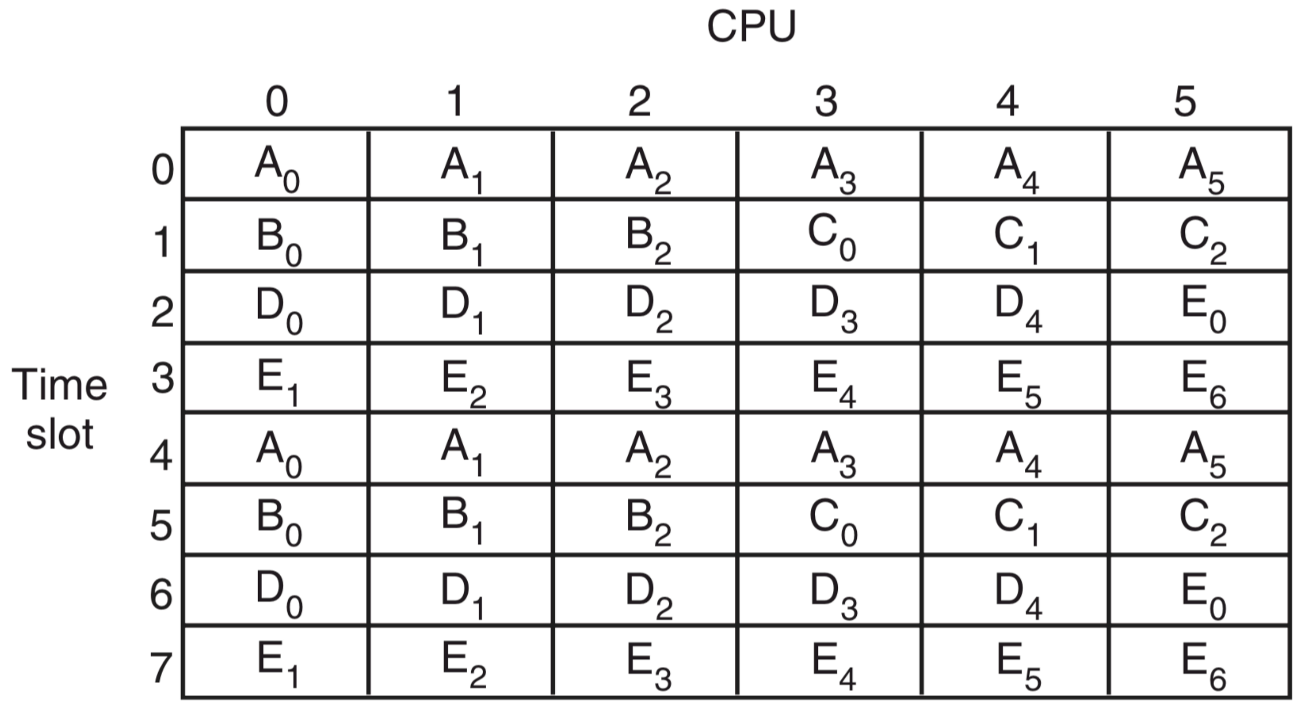
- Schedule both in time and space
- Can prevent problems like the one shown below:



Gang scheduling

- Groups of related threads are scheduled as a gang (with same quantum)
- All gang members run at once
- All gang members start / end their quantum together

Gang scheduling - example



Time slot	CPU 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

Figure 8-15. Gang scheduling.

Thank you for your attention!
Questions?

Feedback / questions:

https://forms.gle/dNkZZ1RE6WwYqzFk6