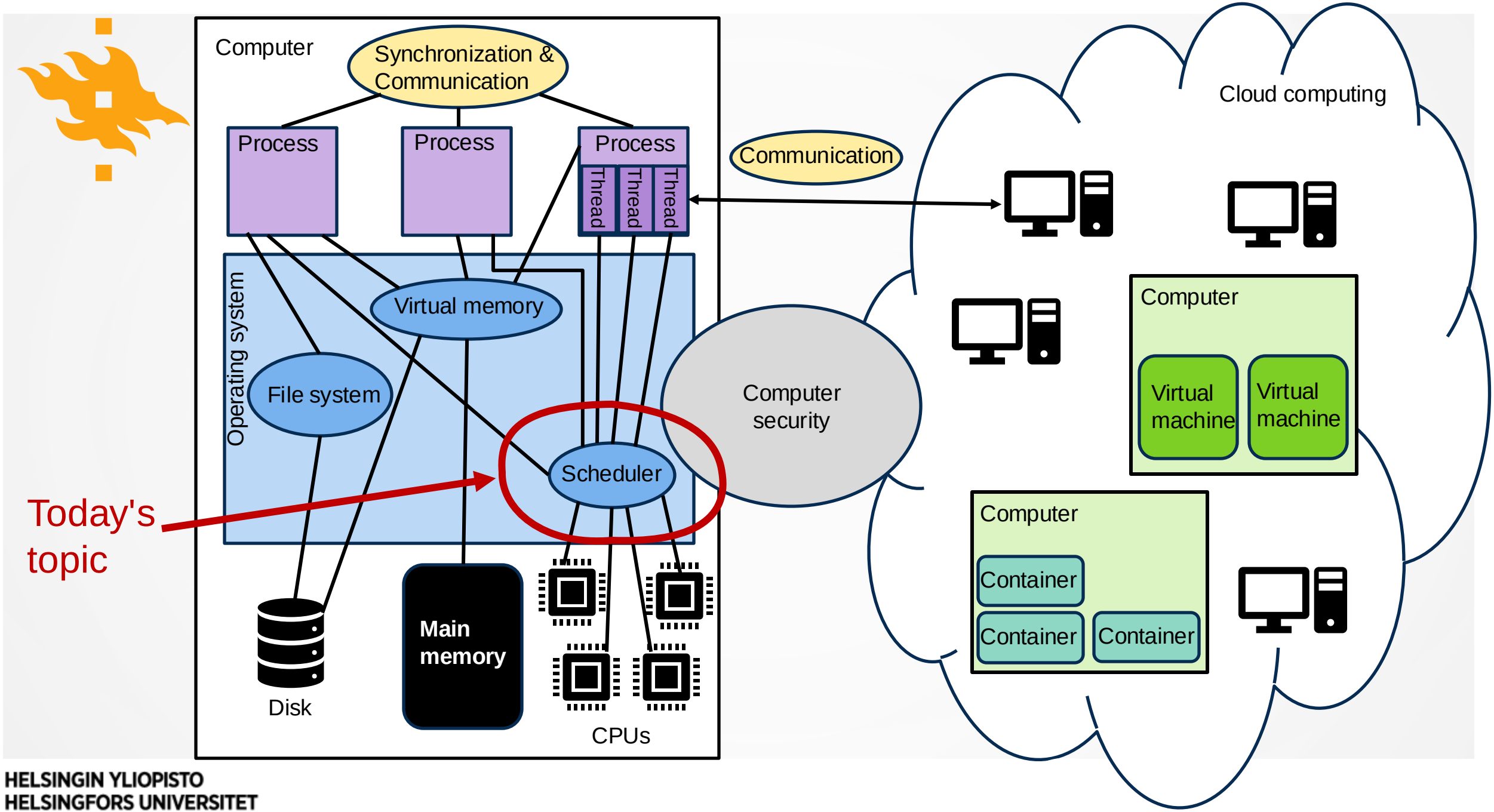
COMPUTING PLATFORMS

Scheduling:
Advanced scheduling algorithms for uniprocessor systems and multiprocessor scheduling


Computer
Synchronization & Communication
Process
Process
Process
Thread
Thread
Thread
Operating system
Virtual memory
File system
Scheduler
Computer security
Today's topic
Main memory
Disk
CPUs
Communication
Cloud computing
Computer
Virtual machine
Virtual machine
Computer
Container
Container
Container

LEARNING OUTCOMES

- After today's lecture, you
  - Are able to describe and apply algorithms for uniprocessor scheduling
  - Know of the design issues in multiprocessor scheduling
  - Are able to describe and apply simple algorithms for multiprocessor scheduling

PRIORITIES AND ADAPTIVITY

- In previous lecture, we looked at simple uniprocessor scheduling algorithms: First-come-first-serve, Shortest job first, Round-Robin
- Are all processes equally important?
- What if we don't know the runtime of the processes in advance? Can we still achieve good scheduling performance?

SCHEDULING ALGORITHMS: PRIORITY SCHEDULING

- Several **Ready**-queues with different priority levels
- Scheduler starts with the highest priority **Ready**-queue (RQ0). If RQ0 is not empty, a process is selected from that queue with some scheduling policy (FCFS, RR, …)
- If RQ0 is empty, the next **Ready**-queue RQ1 is examined, and so on.
- **Problem**: processes in the low priority queues can starve if there is a steady flow of high priority processes

PRIORITY SCHEDULING: EXAMPLE

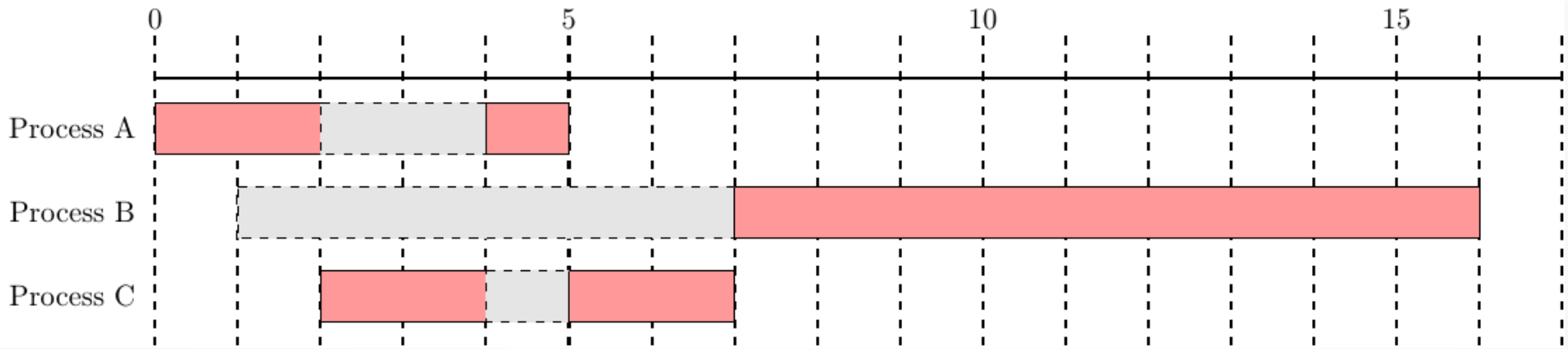
- Assume Round-Robin scheduling with time quantum 2 in both high and low priority queues

Process	Arrival	Runtime	Priority
A	0	3	High
B	1	9	Low
C	2	4	High

PRIORITY SCHEDULING: EXAMPLE

- Assume Round-Robin scheduling with time quantum 2 in both high and low priority queues

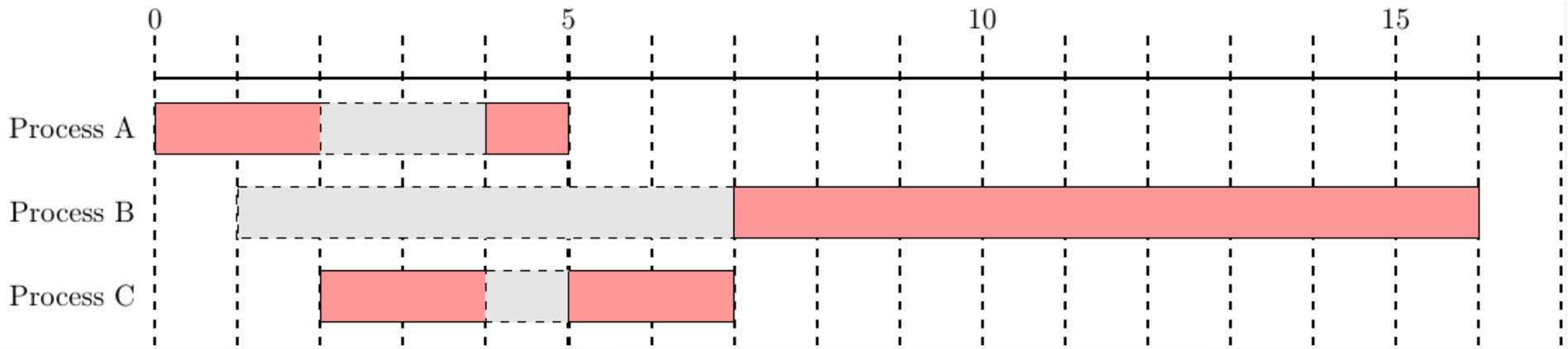
Process	Arrival	Runtime	Priority
A	0	3	High
B	1	9	Low
C	2	4	High



PRIORITY SCHEDULING: EXAMPLE

Process	Arrival	Runtime	Priority
A	0	3	High
B	1	9	Low
C	2	4	High

- Assume Round-Robin scheduling with time quantum 2 in both high and low priority queues



- Average turnaround time: ((5-0)+(16-1)+(7-2))/3 = 8.3
- Average response time: (0+(7-1)+0)/3 = 2

SCHEDULING ALGORITHMS: MULTILEVEL FEEDBACK QUEUE

- Without knowledge of the runtimes of the processes, scheduling algorithms using runtime cannot be used
- By penalizing processes that have been running for a long time we can favor short processes

SCHEDULING ALGORITHMS: MULTILEVEL FEEDBACK QUEUE (MLFQ)

- Rule 1: If Priority(A) > Priority(B), A runs (B doesn't)
- Rule 2: If Priority(A) = Priority(B), A and B run in RR fashion using the quantum of the given priority queue
- Rule 3: When a process arrives, it is placed in the highest priority queue
- Rule 4: When a process has used its allowed time at a given priority level, its priority will be reduced
- Rule 5: After some time period *S*, move all processes to the highest priority queue

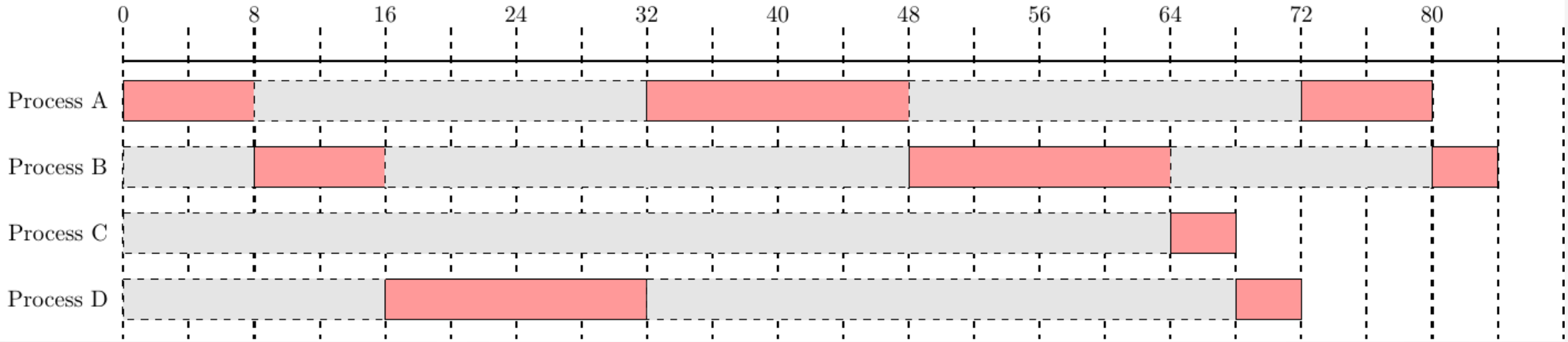
MLFQ: HANDS-ON! (PICK UP PAPER&PEN)

- Rule 1: If Priority(A) > Priority(B), A runs (B doesn't)
- Rule 2: If Priority(A) = Priority(B), A and B run in RR fashion using the quantum of the given priority queue
- Rule 3: When a process arrives, it is placed in the highest priority queue
- Rule 4: When a process has used its allowed time at a given priority level, its priority will be reduced
- Rule 5: After some time period *S*, move all processes to the highest priority queue

- Consider MLFQ with three queues:
  - High priority: scheduled with RR (quantum=8ms), allowed time on this level 8ms
  - Medium priority: scheduled with RR (quantum=16ms), allowed time on this level 16ms
  - Low priority: scheduled with FCFS
- Table below lists the processes, remaining runtimes, and current priorities (A arrived at high priority queue before B)
- How are the processes scheduled from this point on?
- What is the average turnaround time and response time?

PROCESS	RUNTIME	PRIORITY
A	32	high
B	28	high
C	4	low
D	20	medium

MLFQ: HANDS-ON! (PICK UP PAPER&PEN)



- Average turnaround time: (80+84+68+72)/4=76
- Average response time: (0+8+64+16)/4=22

PROCESS	RUNTIME	PRIORITY
A	32	high
B	28	high
C	4	low
D	20	medium

MLFQ: OBSERVATIONS

- Turnaround time of long processes can stretch out alarmingly
- Longer processes can still suffer from starvation
- One possible solution is to promote a process to higher priority queue after it has waited for service in its current queue for a certain amount of time
- Difficult to understand and predict system behavior

SCHEDULING ALGORITHMS: FAIR-SHARE SCHEDULING

- Instead of optimizing turnaround time or response time, the scheduler can guarantee that **each process gets a certain percentage of CPU time**
- Simple version: Lottery scheduling
  - Each process gets a certain number of tickets representing its share of CPU time
  - Scheduler needs to know the total number of tickets
  - Scheduler picks the **winning ticket**, and the process owning that ticket gets to run for the next quantum
  - Example: Process A has tickets 0-24, process B has tickets 25-99
    Scheduler picks ticket 51, process B gets to run

HANDS-ON: LOTTERY SCHEDULING

Lottery scheduling

- Each process gets a certain number of tickets representing its share of CPU time
- Scheduler needs to know the total number of tickets
- Scheduler picks the **winning ticket**, and the process owning that ticket gets to run for the next quantum

- Table below gives two processes and their tickets. Total ticket pool is 0-9 (10 tickets).
- Simulate lottery scheduling for 6 rounds (quantum=2ms)
- Use the last 6 numbers from your student number OR a random number generator to choose the winning ticket
- What proportion of running time did process A get?

PROCESS	TICKETS
A	0,1,2
B	3,4,5,6,7,8,9

FAIR-SHARE SCHEDULING: OBSERVATIONS

- Lottery scheduling leads to **probabilistic correctness** (i.e. desired proportions but no guarantee)
- **Stride scheduling** is a deterministic fair-share scheduling algorithm (see OSTEP for details)

SCHEDULING IN MULTIPROCESSOR / MULTICORE SYSTEMS

- Most desktop computers, laptops etc. are **multiprocessor** systems
- So far, we have looked at uniprocessor scheduling. What changes when there are multiple processors / cores?
  - Several processes / threads can execute **at the same time**
  - **Multiprocessor architecture** (e.g. caches) affects the efficiency of scheduling
  - A process / thread may need to **interact** with other processes / threads
  - Need for **synchronization** (covered later in the course)

CLASSIFICATION OF MULTIPROCESSOR SYSTEMS

- **Loosely coupled or distributed multiprocessor or cluster**: collection of relatively autonomous systems, each processor has its own main memory and IO channels (e.g. cloud systems, covered later in the course)
- **Functionally specialized processors**: e.g., IO processor, GPUs; specialized processors are controlled by the master processor (general purpose processor) and provide services to it
- **Tightly coupled multiprocessor**: a set of processors **sharing main memory** and being under the integrated **control of OS** (our focus today)

SPECIAL CONSIDERATIONS: CACHES

- CPUs have **hardware caches** to speed up access to memory
  - Cache is a **small fast memory**
  - Cache contains copies of **popular** (recently used) data found in main memory
- Two CPUs share the same main memory but each have their own caches (cores can share some caches)
- Caches are based on **temporal and spatial locality**
  - Temporal locality: A recently accessed data item is likely to be accessed again
  - Spatial locality: A program often accesses nearby data items successively
- **Cache affinity**: run a process on the same CPU it has ran on earlier because the cache likely contains relevant data

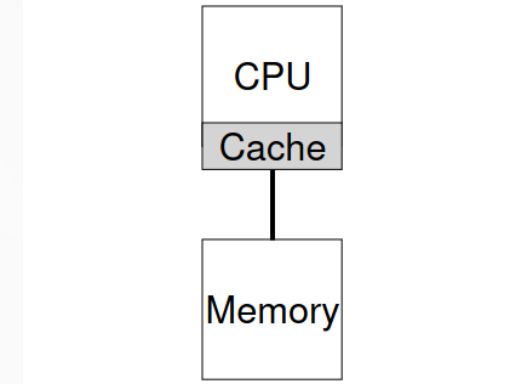


Figure 10.1: **Single CPU With Cache**

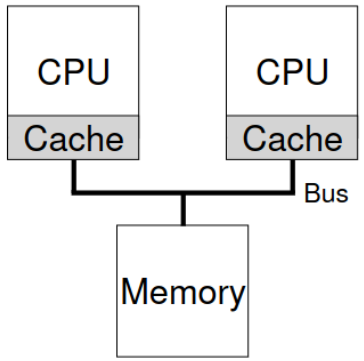


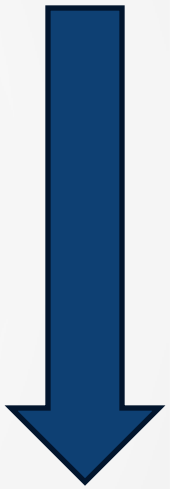
Figure 10.2: **Two CPUs With Caches Sharing Memory**

Figures from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

SPECIAL CONSIDERATIONS: THREAD GRANULARITY

Synchronization interval (#instructions) increases

GRAIN SIZE	EXAMPLE	DESCRIPTION
Fine	Pipelines	Parallelism inherent in a single instruction stream
Medium	Threads	Parallel processing or multitasking in an application
Coarse	Processes	Multiprocessing of concurrent processes in multiprogramming OS
Very coarse	Processes in nodes	Distributed processing across networks
Independent		Multiple unrelated processes



- If processes / threads need to interact frequently, it is advantageous to run them simultaneously

Table adapted from Table 10.1 [Stallings, Operating Systems: Internals and design principles, 9th edition]

ASSIGNMENT OF PROCESSES TO PROCESSORS

- Simplest to treat processors as a pooled resource and assign processes to processors on demand
  - **Static**: process permanently assigned to a processor
  - **Dynamic**: a single scheduling queue or **dynamic load balancing**
- Where does the scheduler run?
  - **Master/slave**: A dedicated processor for kernel, other processors serve user processes
  - **Peer**: Kernel can run on any processor, each processor self-schedules from the pool of processes

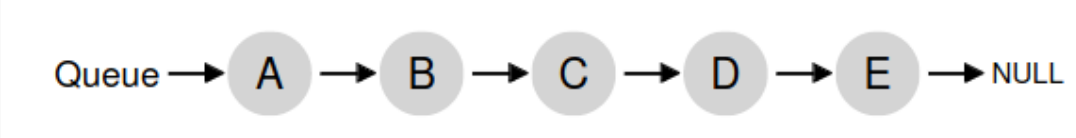
MULTIPROCESSOR SCHEDULING: SINGLE QUEUE

- **Single Ready-queue from which each processor picks processes to run (using e.g. Round-Robin policy)**

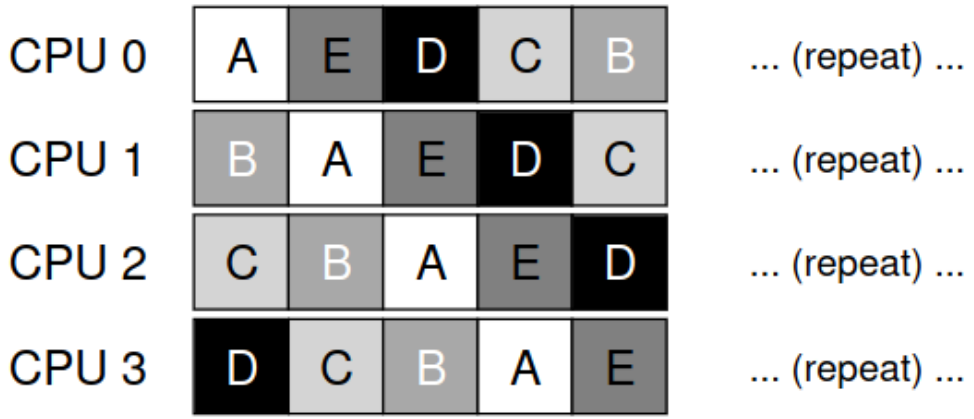
PROS	CONS
• Simple, easy to implement • Load distributed evenly • No centralized scheduler	• Lack of scalability: Need to synchronize access to scheduling queue which induces overhead • Does not preserve cache affinity

SINGLE QUEUE: EXAMPLE

- Consider five processes A, B, C, D, E run on a system with four CPUs
- Scheduling queue:



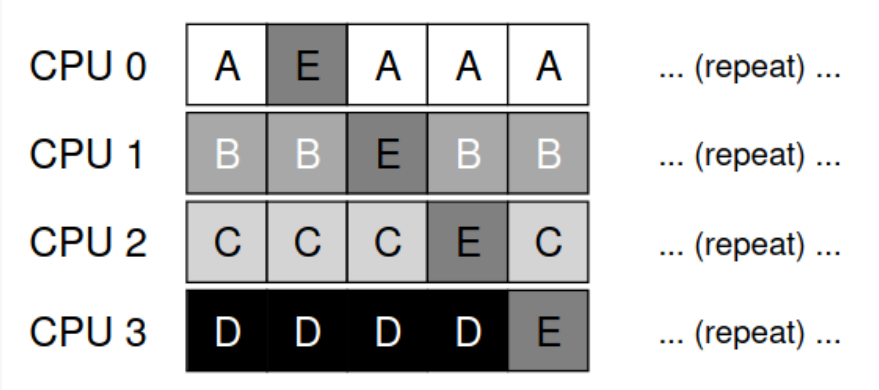
- Scheduling diagram assuming Round-Robin scheduling:



Figures from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

SINGLE QUEUE: PRESERVING CACHE AFFINITY

- Provide cache affinity for some processes
- Other processes moved around to balance load
- Example: Processes A, B, C, D are always on the same processor, process E **migrated** to balance load



- In a later round, migrate another process to provide cache affinity fairness
- Disadvantage: Complex to implement

Figure from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

MULTIPROCESSOR SCHEDULING: MULTIPLE QUEUES (ONE PER PROCESSOR)

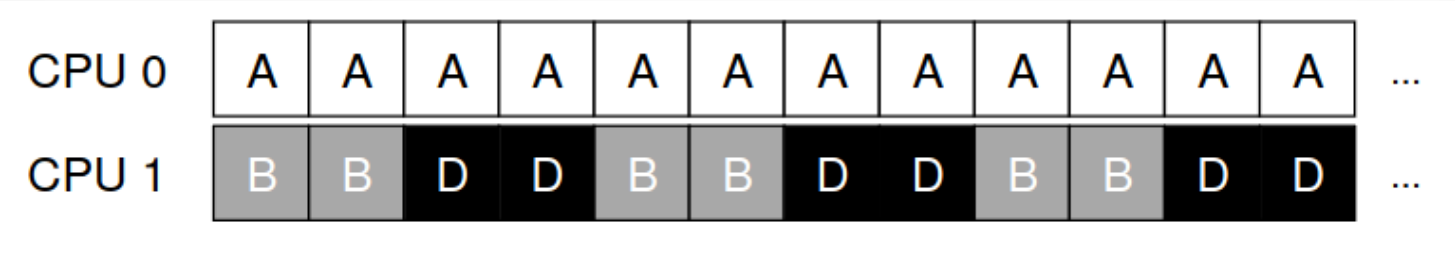
- **Each CPU has its own Ready-queue scheduled with some policy (e.g. Round-Robin)**
- **When a process enters the system it is placed in one queue (randomly, put into the shortest queue,...).**

PROS	CONS
• Scalability • Provides cache affinity	• Load imbalance • Unfairness

MULTIPLE QUEUES: EXAMPLE

- System with two processors. The scheduling queues:

- Scheduling diagram assuming Round-Robin scheduling in each queue:



Figures from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

MULTIPLE QUEUES: LOAD BALANCING

- To balance the load of different CPUs **migrate** processes from one queue to another
- **Work stealing**:
  - A (source) queue with low number of processes will peek into another (target) queue
  - If the target queue is notably fuller, "steal" a process from target queue to source queue
  - How often to peek into other queues?

THREAD SCHEDULING

- Application can be implemented as a set of **cooperating threads that execute concurrently in the same address space**
  - On uniprocessor threads used for program structuring and overlapping IO with processing
  - On multiprocessor true parallelism can be exploited
- Dramatic gains possible but if the threads interact frequently, **small differences** in thread management and **scheduling** can have a **significant impact on performance**

THREAD SCHEDULING: SINGLE QUEUE

- **A global queue of ready threads is maintained and each processor selects a thread to run from that queue**

PROS	CONS
• Load distributed evenly • No centralized scheduler • One of the uniprocessor scheduling policies (FCFS, …) can be used for the global queue	• Mutual exclusion must be enforced on the global queue • Does not preserve cache affinity for threads • All threads in a process unlikely to execute simultaneously, interaction between threads is inefficient

THREAD SCHEDULING: GANG SCHEDULING

- **Simultaneously schedule all threads that make up a single process**

PROS	CONS
• Interaction between threads more efficient • One scheduling decision affects many processors leading to less scheduling overhead	• If a process does not use all processors, wasted processor resources

GANG SCHEDULING: EXAMPLE

- System with four processors
- Assume process A has four threads and process B has one thread
- Gang scheduling with uniform allocation wastes 3/8=37.5% of processor resources

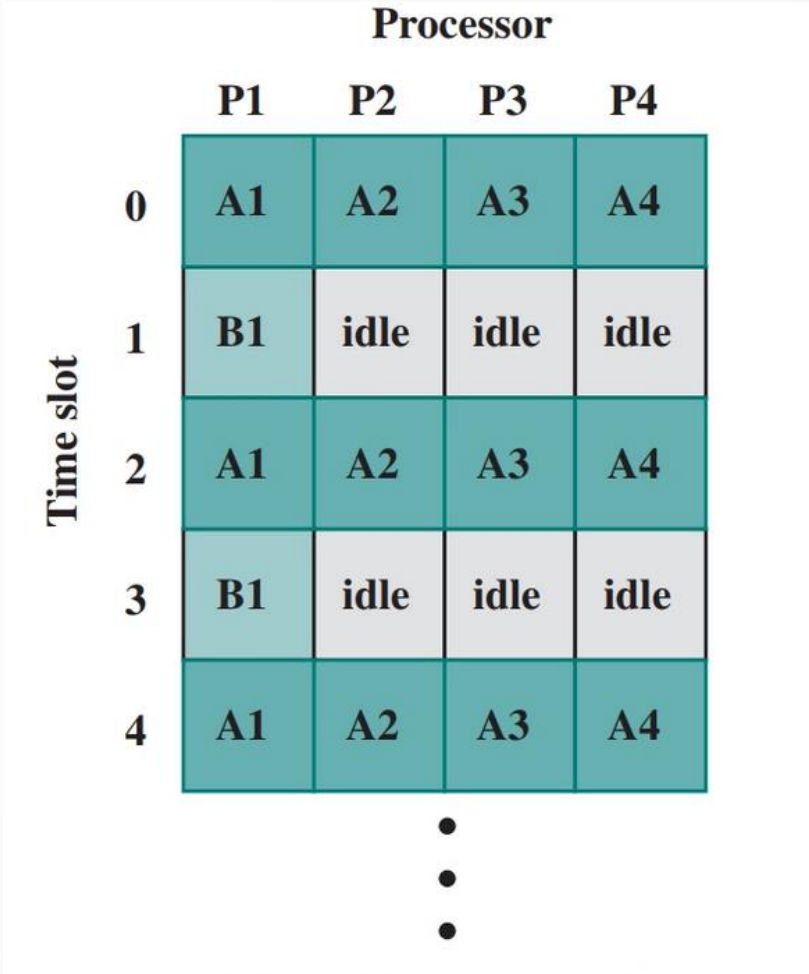


Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

THREAD SCHEDULING: DYNAMIC SCHEDULING

- For some applications, it is possible to provide language and system tools that permit **the number of threads in the process to be altered dynamically**
  - Allows OS to adjust the load to improve utilization
- Both OS and application involved in scheduling decisions
- Outperforms gang scheduling for applications able to take advantage of this approach

SUMMARY

- Scheduling of processes is an important part of an OS
- Uniprocessor scheduling algorithms: FCFS, SJF, RR, MLFQ, Fair-Share Scheduling
- Considerations on multiprocessor systems:
  - Cache affinity
  - Thread interaction
- Multiprocessor scheduling algorithms: Single queue, Multiple queues, Gang scheduling, Dynamic scheduling