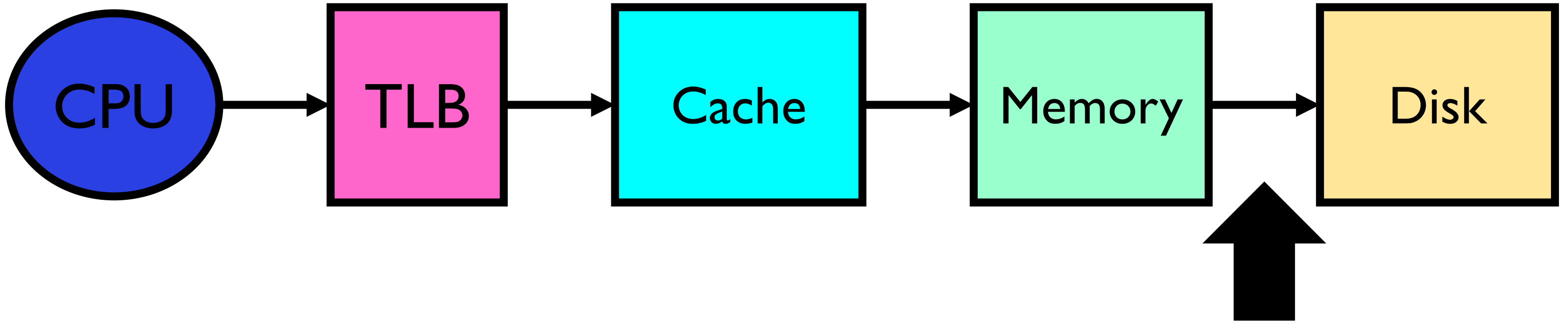
Operating Systems
Lecture 9

Scheduling

Prof. Mengwei Xu

Recap: Cache Hierarchy

- Memory as cache for secondary disk



Recap: Demand Paging (需求分页)

- Modern programs require a lot of physical memory, but they don't use all their memory all of the time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user's code to be in memory
- Solution: use main memory as cache for disk
  - "lazy" memory allocation
- An illusion of infinite memory
  - In-use virtual memory can be bigger than physical memory
  - Combined memory of running processes much larger than physical memory
    - ❑ More programs fit into memory, allowing more concurrency
  - Principle: page table for transparent management

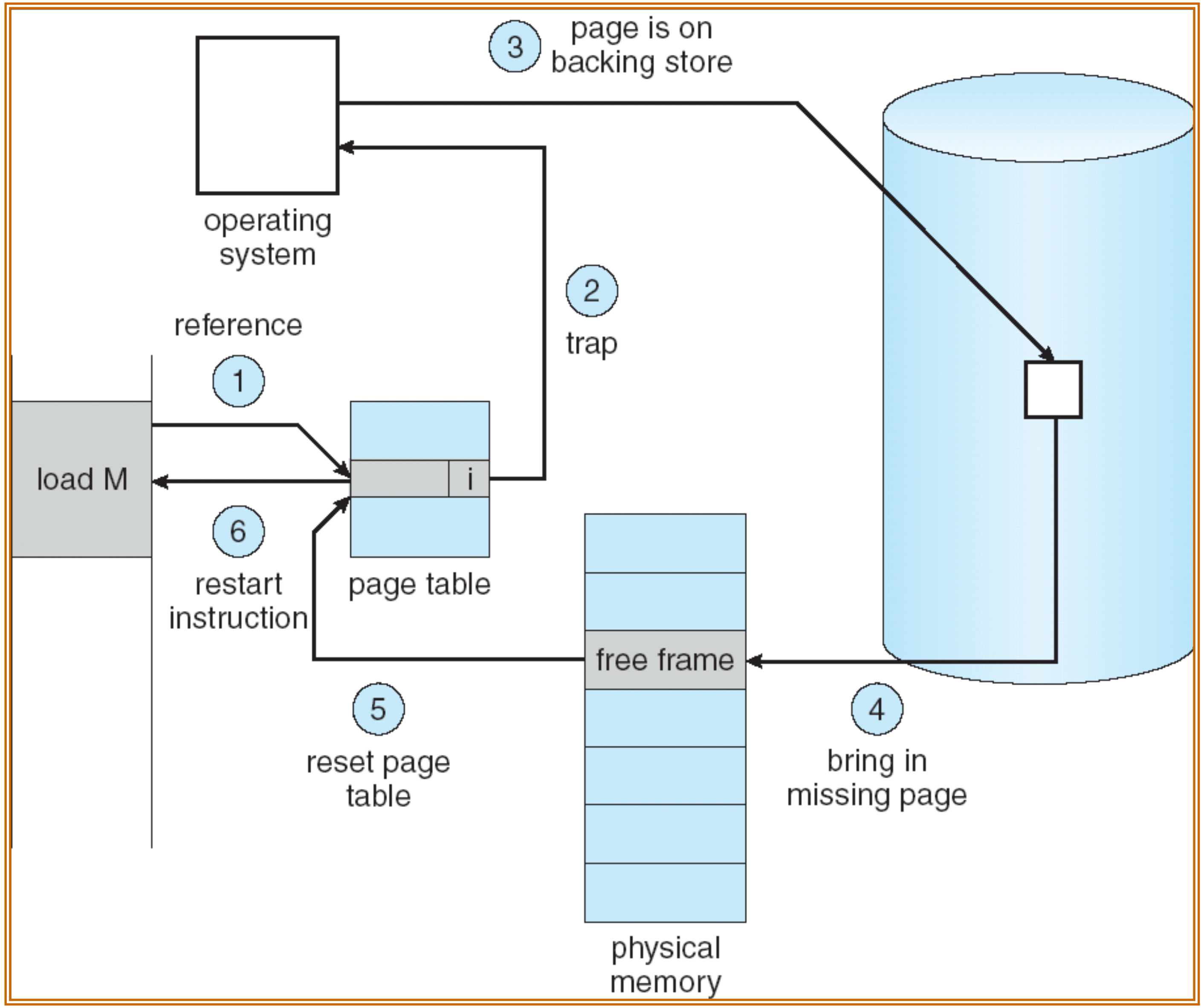
Recap: Demand Paging as Cache

- What is block size?
  - 1 page
- What is organization of this cache (i.e. direct-mapped, set-associative, fully-associative)?
  - Fully associative: arbitrary virtual → physical mapping
- How do we find a page in the cache when look for it?
  - First check TLB, then page-table traversal
- What is page replacement policy? (i.e. LRU, Random…)
  - This requires more explanation… (kinda LRU)
- What happens on a miss?
  - Go to lower level to fill miss (i.e. disk)
- What happens on a write? (write-through, write back)
  - Write-back – need dirty bit!

Recap: Implementation of mmap

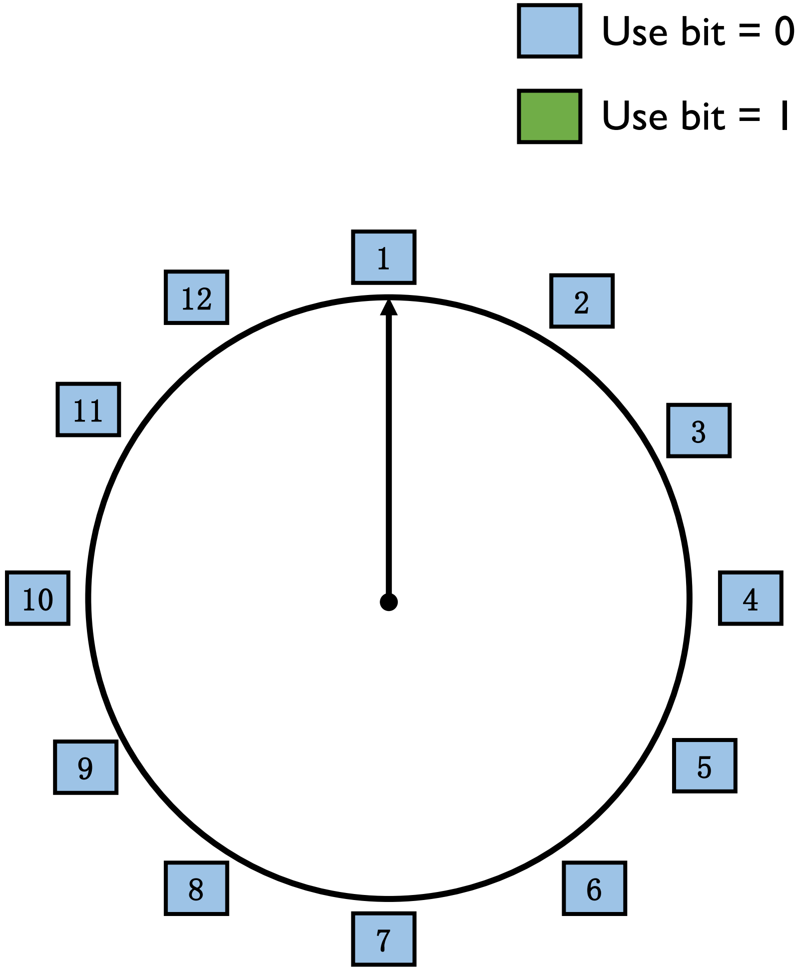
- When program accesses an invalid address
    1. [MMU] TLB miss; full page table lookup
    2. [MMU + OS] Trapping into page fault handler
    3. [OS] Convert virtual address to file offset
    4. [OS] Allocate a new page frame in memory
    5. [OS] Read data from disk to the memory (blocked)
    6. [CPU] Disk interrupt when read completes
    7. [OS] Updating page table by marking the entry as valid
    8. [OS] Resume process
    9. [MMU] TLB miss; full page table lookup
    10. [MMU] TLB update

Recap: Implementation of mmap



Recap: Page Eviction Policy

- Clocking algorithm: approximating LRU
- Implementation with the *use* bit
  - Initialized to 0 in page table
  - Set to 1 whenever there is a page access
- When we need to evict a page, we look at the page under the hand:
  - If its use bit = 1, clear it and move the hand, repeat;
  - If its use bit = 0, evict it



Page reference stream:

Recap: N^{th} Chance Version of Clock Algorithm

- N^{th} chance algorithm: Give page N chances
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks use bit:
    - ❑ 1 → clear use and also clear counter (used in last sweep)
    - ❑ 0 → increment counter; if count=N, replace page
  - Means that clock hand has to sweep by N times without page being used before page is replaced
- How do we pick N?
  - Why pick large N? Better approximation to LRU
    - ❑ If N ~ 1K, really good approximation
  - Why pick small N? More efficient
    - ❑ Otherwise might have to look a long way to find free page
- What about dirty pages?
  - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
  - Common approach:
    - ❑ Clean pages, use N=1
    - ❑ Dirty pages, use N=2 (and write back to disk when N=1)

Recap: Details of Clock Algorithms

- Which bits of a PTE entry are useful to us?
  - Use: Set when page is referenced; cleared by clock algorithm
  - Modified: set when page is modified, cleared when page written to disk
  - Valid: ok for program to reference this page
  - Read-only: ok for program to read page, but not modify
    - For example for catching modifications to code pages!
- Do we really need hardware-supported "modified" bit?
  - No. Can emulate it (BSD Unix) using read-only bit
    - Initially, mark all pages as read-only, even data pages
    - On write, trap to OS. OS sets software "modified" bit, and marks page as read-write.
    - Whenever page comes back in from disk, mark read-only

Scheduling (调度) Concept

- Why we need scheduling? Multitasks and Concurrency.
- Scheduling is only useful when there is not enough resources
- Preemption (抢占) is the basic assumption for fine-grained scheduling
  - Either by timer interrupts or other kinds of interrupts
- Who schedules processes/threads?
  - Mostly by OS.
  - User-level thread libraries schedule the threads by themselves.

Scheduling Policy Goals (1/3)

- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
  - Response time is what the user sees:
    - ❑ Time to echo a keystroke in editor
    - ❑ Time to compile a program
    - ❑ Real-time tasks: Must meet deadlines imposed by World

Scheduling Policy Goals (2/3)

- Minimize Response Time
- Maximize Throughput
  - Maximize operations (or jobs) per second
  - Throughput related to response time, but not identical:
    - ❑ Minimizing response time will lead to more context switching than if you only maximized throughput
  - Two parts to maximizing throughput
    - ❑ Minimize overhead (for example, context-switching)
    - ❑ Efficient use of resources (CPU, disk, memory, etc)

Scheduling Policy Goals (3/3)

- Minimize Response Time
- Maximize Throughput
- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    - Better *average* response time by making system *less* fair

First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS, 先到先服务)
  - Also "First In, First Out" (FIFO, 先进先出) or "Run until done"
    - ❑ In early systems, FCFS meant one program scheduled until done
    - ❑ Now, means keep CPU until thread blocks
- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

  - Suppose processes arrive in the order: P_1 , P_2 , P_3
    The Gantt Chart (甘特图) for the schedule is:

First-Come, First-Served (FCFS) Scheduling

- Example continued:

  - Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
  - Average waiting time: $(0 + 24 + 27)/3 = 17$
  - Average Completion time: $(24 + 27 + 30)/3 = 27$
- *Convoy effect* (护航效应)*:* short process behind long process

First-Come, First-Served (FCFS) Scheduling

- Example continued:
  - Suppose that processes arrive in order: P_2 , P_3 , P_1 Now, we have:

  - Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
  - Average waiting time: $(6 + 0 + 3)/3 = 3$
  - Average Completion time: $(3 + 6 + 30)/3 = 13$
- In second case:
  - Average waiting time is much better (before it was 17)
  - Average completion time is better (before it was 27)
- FIFO Pros and Cons:
  - Simple (+)
  - Short jobs get stuck behind long ones (-)
    - ❑Safeway: Getting milk, always stuck behind cart full of small items

Shortest Job First (SJF) Scheduling

- Shortest Job First (短任务优先) Scheduling
  - Always schedule the job with the shortest remaining time (so sometimes it's also called shortest-remaining-time-first, SRTF)
  - It theoretically minimizes the average response time, why?

Shortest Job First (SJF) Scheduling

- Shortest Job First (短任务优先) Scheduling
  - Always schedule the job with the shortest remaining time (so sometimes it's also called shortest-remaining-time-first, SRTF)
  - It theoretically minimizes the average response time, why?

- Comparison of SRTF with FCFS
  - What if all jobs the same length?
    - ❑ SRTF becomes the same as FCFS (i.e. FCFS is best can do if all jobs the same length)
  - What if jobs have varying length?
    - ❑ SRTF: short jobs not stuck behind long ones

Shortest Job First (SJF) Scheduling

- Shortest Job First (短任务优先) Scheduling
  - Always schedule the job with the shortest <u>remaining</u> time (so sometimes it's also called shortest-remaining-time-first, SRTF)
  - It theoretically minimizes the average response time, why?

- Con#1: starvation (饥饿)
  - If small jobs keep coming, the long jobs will not be served
  - Fairness issue
- Con#2: implementation
  - It's hard to know the task remaining time

Round Robin (RR) Scheduling

- Round Robin (轮询调度) Scheme
  - Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds
  - After quantum expires, the process is preempted and added to the end of the ready queue.
- n processes in ready queue and time quantum is $q \Rightarrow$
  - ❑ Each process gets $1/n$ of the CPU time
  - ❑ In chunks of at most q time units
  - ❑ No process waits more than $(n-1)q$ time units

Round Robin (RR) Scheduling

- Example:

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

  - quantum=20
  - Average waiting time?
  - Average completion time?

Round Robin (RR) Scheduling

- Example:

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

  - The Gantt chart (quantum=20) is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 28 48 68 88 108 112 125 145 153

  - Waiting time for $P_1=(68-20)+(112-88)=72$
    $P_2=(20-0)=20$
    $P_3=(28-0)+(88-48)+(125-108)=85$
    $P_4=(48-0)+(108-68)=88$
  - Average waiting time = $(72+20+85+88)/4=66\frac{1}{4}$
  - Average completion time = $(125+28+153+112)/4 = 104\frac{1}{2}$

Round Robin (RR) Scheduling

- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

Round Robin (RR) Scheduling

- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)
- How do you choose time slice?
  - Too large: Response time suffers
    - What if infinite (∞)? Falls back to FIFO
  - Too small: Throughput suffers

Round Robin (RR) Scheduling

- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)
- How do you choose time slice?
- Actual choices of timeslice:
  - Initially, UNIX timeslice one second:
    - Worked ok when UNIX was used by one or two people.
    - What if three compilations going on? 3 seconds to echo each keystroke!
  - Need to balance short-job performance and long-job throughput:
    - Typical time slice today is between 10ms – 100ms
    - Typical context-switching overhead is 0.1ms – 1ms
    - Roughly 1% overhead due to context-switching

Comparing FCFS and RR

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example:

  10 jobs, each take 100s of CPU time
  RR scheduler quantum of 1s
  All jobs start at the same time

- Completion Times:

Job #	FIFO	RR
1	100	991
2	200	992
…	…	…
9	900	999
10	1000	1000

  - Average response time
    - ❑ Bad when all jobs same length
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
  - Total time for RR longer even for zero-cost switch!

Choice of Time Quantum for RR

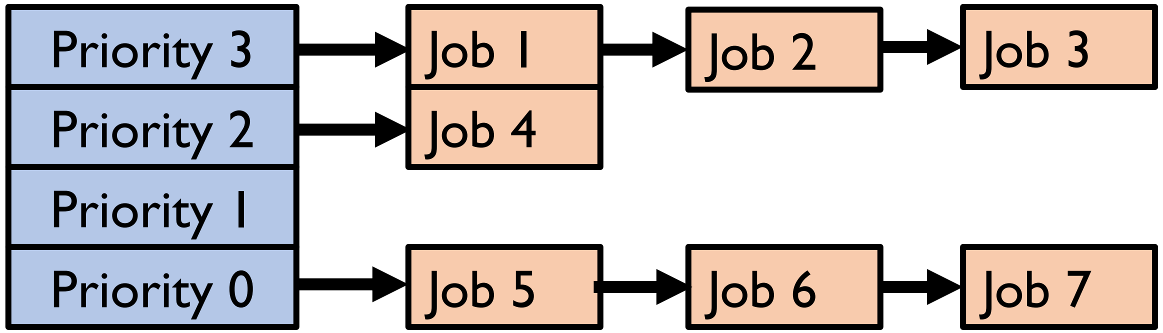
Best FCFS:

P_2 [8]	P_4 [24]	P_1 [53]	P_3 [68]

0 8 32 85 153

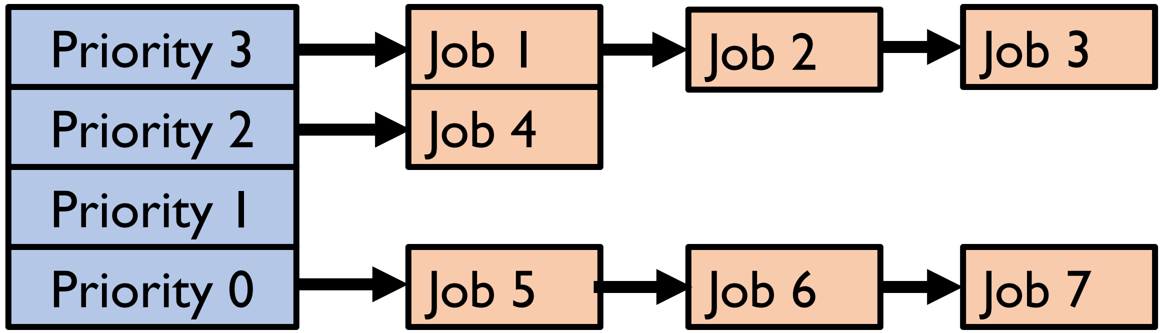
	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	31¼
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	61¼
	Q = 8	80	8	85	56	57¼
	Q = 10	82	10	85	68	61¼
	Q = 20	72	20	85	88	66¼
	Worst FCFS	68	145	0	121	83½
Completion Time	Best FCFS	85	8	153	32	69½
	Q = 1	137	30	153	81	100½
	Q = 5	135	28	153	82	99½
	Q = 8	133	16	153	80	95½
	Q = 10	135	18	153	92	99½
	Q = 20	125	28	153	112	104½
	Worst FCFS	121	153	68	145	121¾

Strict Priority Scheduling



- Strict Priority Scheduling (严格优先级调度)
  - Always execute highest-priority runnable jobs to completion
  - Each queue can be processed in RR with some time-quantum
- Problems:
  - Starvation: Lower priority jobs don't get to run because higher priority jobs
  - Deadlock: Priority Inversion (优先级翻转)
    - ❑ Not strictly a problem with priority scheduling, but happens when low priority task has lock needed by high-priority task

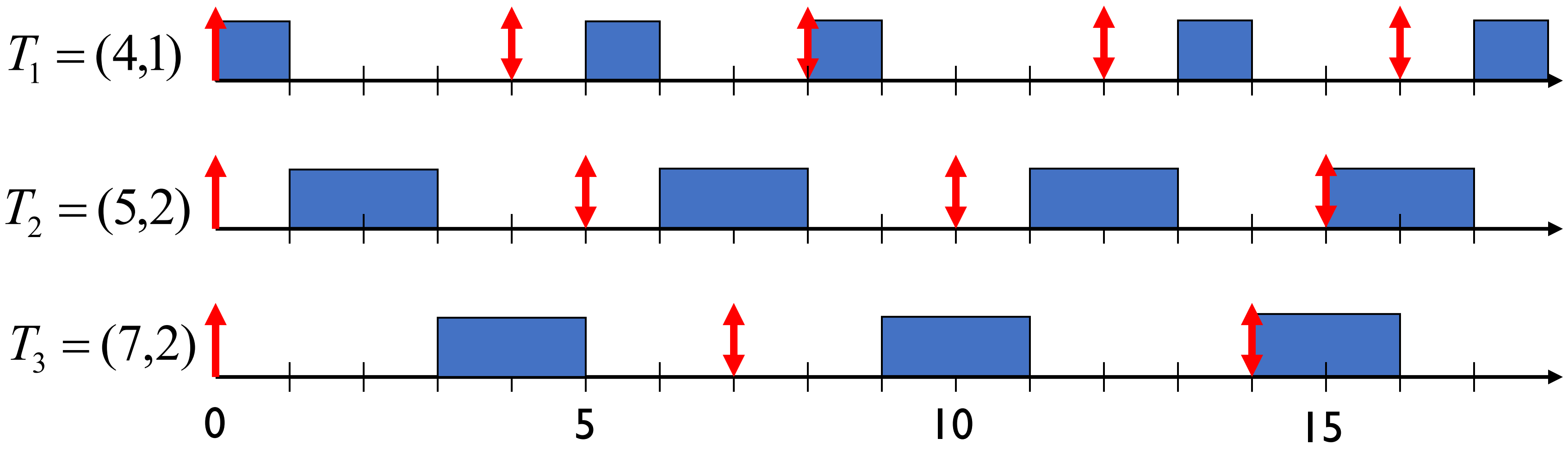
Strict Priority Scheduling



- Strict Priority Scheduling (严格优先级调度)
  - Always execute highest-priority runnable jobs to completion
  - Each queue can be processed in RR with some time-quantum
- Problems:
  - Starvation: Lower priority jobs don't get to run because higher priority jobs
  - Deadlock: Priority Inversion (优先级翻转)
- How to fix? Dynamic priority
  - Dynamic priorities – adjust base-level priority up or down based on heuristics about interactivity, locking, burst behavior, etc…

Earliest Deadline First (EDF)

- Tasks periodic with period P and computation C in each period: (P, C)
- Preemptive priority-based dynamic scheduling
- Each task is assigned a (current) priority based on how close the absolute deadline is
- The scheduler always schedules the active task with the closest absolute deadline



Scheduling Fairness

- What about fairness?
  - Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
    - Long running jobs may never get CPU
    - In Multics, shut down machine, found 10-year-old job
  - Must give long-running jobs a fraction of the CPU even when there are shorter jobs to run
  - Tradeoff: fairness gained by hurting avg response time!

Scheduling Fairness

- How to implement fairness?
  - Could give each queue some fraction of the CPU
    - ❑ What if one long-running job and 100 short-running ones?
    - ❑ Like express lanes in a supermarket—sometimes express lanes get so long, get better service by going into one of the other lines
  - Could increase priority of jobs that don't get service
    - ❑ What is done in some variants of UNIX
    - ❑ This is ad hoc—what rate should you increase priorities?
    - ❑ And, as system gets overloaded, no job gets CPU time, so everyone increases in priority $\Rightarrow$ Interactive jobs suffer

Scheduling Fairness

- If every tasks need the same resource, fairness is easy: RR.
- Yet, tasks may demand different: compute-bound vs. I/O bound
- Max-Min fairness: iteratively maximize the minimum allocation given to a particular process (or threads/users/applications) until all resources are assigned
  - Mostly used in network

Multi-level Feedback Queue (MFQ) Scheduling

- Multi-level Feedback Queue (MFQ, 多级反馈队列调度)
  - Achieves responsiveness (short jobs quickly as SJF), low overhead (minimizing the preemptions and scheduling decision time), and starvation-free (as RR), and fairness (approximately max-min fair share).
    - Yet, it does not perfectly achieve any of these goals.
  - Widely used in commercial OSes such as Windows, MacOS, and Linux.

- Assuming a mix of two kinds of workloads
  1. Interactive tasks (e.g., waiting for user keyboard input): using CPU for a short time, then yield for I/O waiting. Low latency is critical.
  2. CPU-intensive tasks (e.g., compressing files): using CPU for a long period of time. The response time often does not matter much.

Multi-level Feedback Queue (MFQ) Scheduling

- A naïve version of MFQ: maintaining many tasks queues with different priorities, and use following schedule rules.
  - Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
  - Rule 2: If Priority(A) = Priority(B), A & B run in RR.

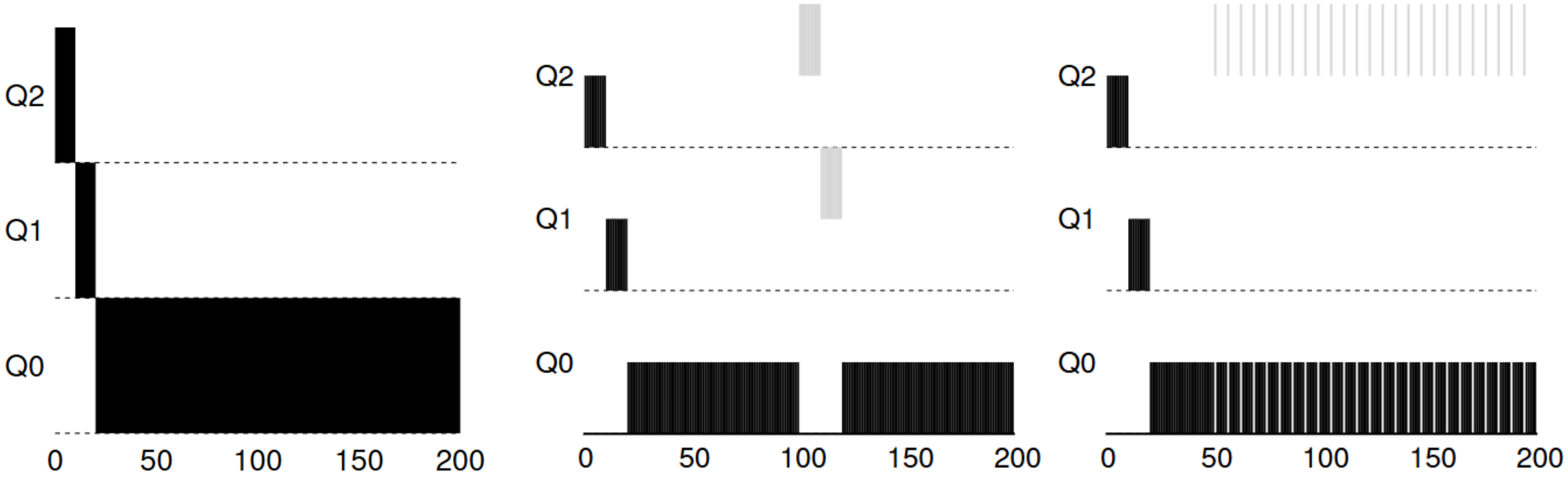
- The key here is how to set the priority.
  - Intuitively, if a job repeatedly relinquishes the CPU while waiting for input from the keyboard, it shall be kept in high priority.
  - Otherwise, if a job uses CPU intensively for long periods of time, its priority shall be reduced.

Multi-level Feedback Queue (MFQ) Scheduling

- A naïve version of MFQ: maintaining many tasks queues with different priorities, and use following schedule rules.
  - Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
  - Rule 2: If Priority(A) = Priority(B), A & B run in RR.
- Our solution: assign a quota for each job at a given priority level, and reduces its priority once the quota is used up.
  - Rule 3: When a job enters the system, it is placed at the highestpriority (the topmost queue).
  - Rule 4a: If a job uses up its allotment while running, its priority isreduced (i.e., it moves down one queue).
  - Rule 4b: If a job gives up the CPU (for example, by performing an I/O operation) before the allotment is up, it stays at the samepriority level (i.e., its allotment is reset).

Multi-level Feedback Queue (MFQ) Scheduling

- A few illustrative examples of our naïve MFQ design.



(a) A single long-running job

(b) A single long-running job and a short-running job

(c) A single long-running job and an interactive job that only uses CPU for 1ms per time then waits for I/O

Multi-level Feedback Queue (MFQ) Scheduling

- There are many issues with this naïve version of MFQ.
  - Starvation: if there are "too many" interactive jobs in the system, they will consume all CPU time, and thus long-running jobs will starve.

Multi-level Feedback Queue (MFQ) Scheduling

- There are many issues with this naïve version of MFQ.
  - Starvation: if there are "too many" interactive jobs in the system, they will consume all CPU time, and thus long-running jobs will starve.
- Solution#1: priority boost
  - Rule 5: After some time period S, move all the jobs in the systemto the topmost queue.

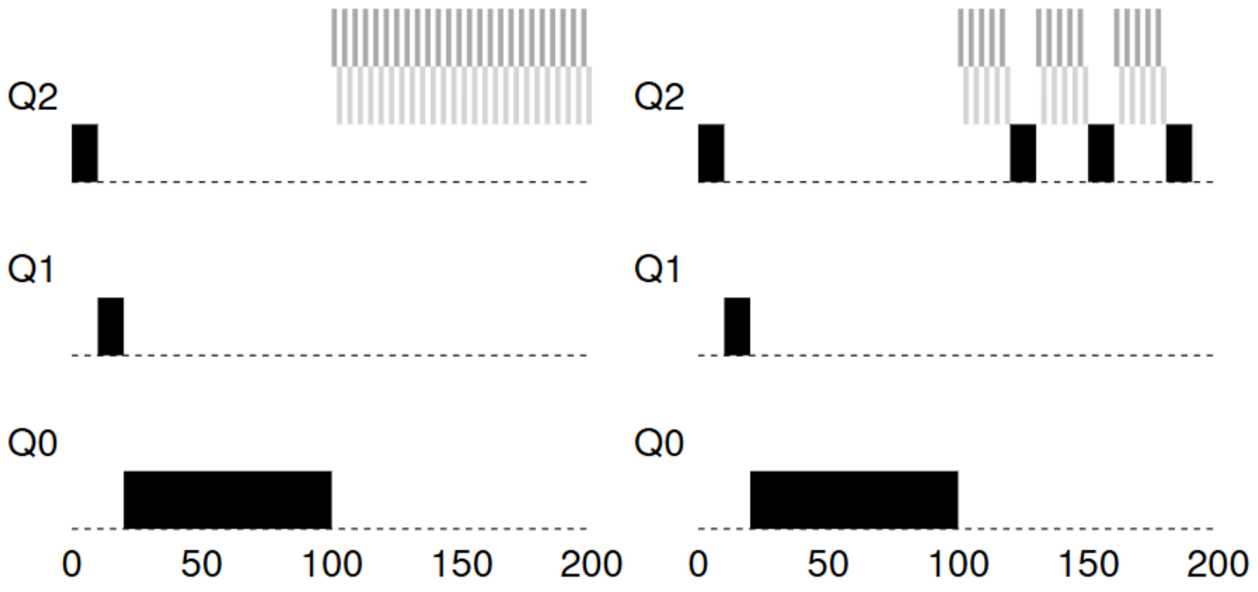


Figure 8.4: **Without (Left) and With (Right) Priority Boost**

Multi-level Feedback Queue (MFQ) Scheduling

- There are many issues with this naïve version of MFQ.
  - Starvation: if there are "too many" interactive jobs in the system, they will consume all CPU time, and thus long-running jobs will starve.
- Solution#1: priority boost
  - Rule 5: After some time period S, move all the jobs in the systemto the topmost queue.
  - S shall be neither too large or too small. Why?

Multi-level Feedback Queue (MFQ) Scheduling

- There are many issues with this naïve version of MFQ.
  - Starvation: if there are "too many" interactive jobs in the system, they will consume all CPU time, and thus long-running jobs will starve.
- Solution#1: priority boost
  - Rule 5: After some time period S, move all the jobs in the systemto the topmost queue.
  - S shall be neither too large or too small. Why?
- Solution#2: time slice across queues
  - each queue gets a certain amount of CPU time
  - e.g., 70% to highest, 20% next, 10% lowest
- More solutions..

Multi-level Feedback Queue (MFQ) Scheduling

- There are many issues with this naïve version of MFQ.
  - Starvation: if there are "too many" interactive jobs in the system, they will consume all CPU time, and thus long-running jobs will starve.
  - Countermeasure: user action that can foil intent of OS designers, e.g., put in a bunch of meaningless I/O to keep job's priority high.
  - How to parameterize the scheduler: how many queues should there be? How big should the time slice be per queue?
  - More..

- Think of possible solutions to them?

Multi-level Feedback Queue (MFQ) Scheduling

- To further extend the MFQ design: Each queue has its own scheduling parameters or even different algorithms!
  - ❑ e.g. foreground – RR, background – FCFS
  - ❑ Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next: 2ms, next: 4ms, etc)

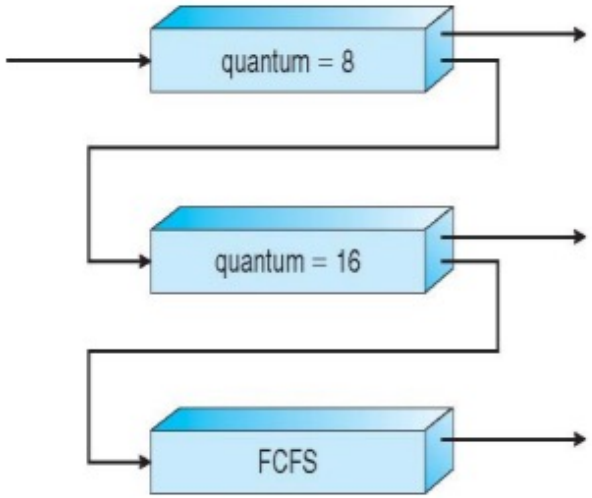
- Adjust each job's priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn't expire, push up one level (or stay at the same one)

Multi-level Feedback Queue (MFQ) Scheduling

- A test

- Assume we have 4 processes in a system with multilevel feedback queue scheduling policy. All the processers arrived ate time 0 and located in the highest level queue in the order of their IDs (1 to 4) a) Calculate the average waiting time and average turnaround time.

Process	Burst Time
P_1	11
P_2	26
P_3	31
P_4	45



b) If a new process P_5 enters the system at time 35 how the gantt chart is going to change?

Fair-share Scheduler

- This type of scheduler aims to guarantee that each job obtain a certain percentage of CPU time.
  - Also known as "proportional-share scheduler.

- Next, we will discuss two types of fair-share scheduler.
  - Lottery scheduling
  - The Linux Completely Fair Scheduler (CFS)

Lottery Scheduling

- Lottery Scheduling
  - Give each job some number of lottery tickets
  - On each time slice, randomly pick a winning ticket
  - On average, CPU time is proportional to number of tickets given to each job (but not deterministically!)
- Assuming there are two jobs: A with 75 tickets, B with 25 tickets
  - Here, B gets run 4 out of 20 time slices (20%).
  - With more tries, B is more likely to get 25% slices.

Here is an example output of a lottery scheduler's winning tickets:

```
63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43  0 49 12
```

Here is the resulting schedule:

```
 A     A  A     A  A  A  A  A  A     A     A  A  A  A  A  A
    B        B                    B     B
```

Lottery Scheduling

- Lottery Scheduling
  - Give each job some number of lottery tickets
  - On each time slice, randomly pick a winning ticket
  - On average, CPU time is proportional to number of tickets given to each job
- How to assign tickets?
  - To approximate SRTF, short running jobs get more, long running jobs get fewer
  - To avoid starvation, every job gets at least one ticket (everyone makes progress)
- Advantage over strict priority scheduling: behaves gracefully as load changes
  - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses

Lottery Scheduling

- Lottery Scheduling Example
  - Assume short jobs get 10 tickets, long jobs get 1 ticket

# short jobs/ # long jobs	% of CPU each short jobs gets	% of CPU each long jobs gets
1/1	??	??
0/2	??	??
2/0	??	??
10/1	??	??
1/10	??	??

Lottery Scheduling

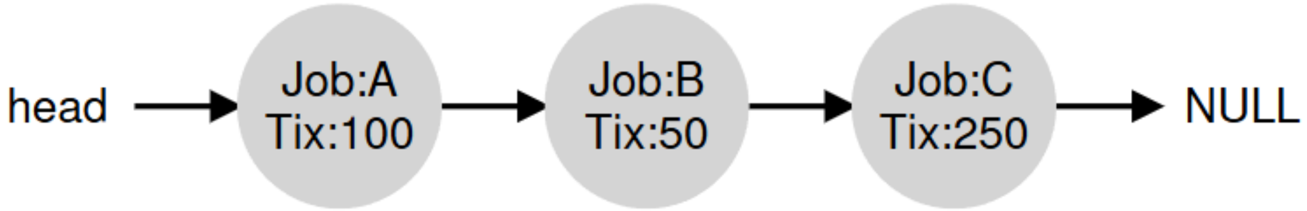
- Lottery Scheduling Example
  - Assume short jobs get 10 tickets, long jobs get 1 ticket

# short jobs/ # long jobs	% of CPU each short jobs gets	% of CPU each long jobs gets
1/1	91%	9%
0/2	N/A	50%
2/0	50%	N/A
10/1	9.9%	0.99%
1/10	50%	5%

Lottery Scheduling

- Implementing lottery scheduling is amazingly easy!
  - One of the important feature of it.
- You only need
  1. A good random number generator
  2. A data structure to track the processes of the system and the total number of tickets

Lottery Scheduling




```
// counter: used to track if we've found the winner yet
int counter = 0;

// winner: call some random number generator to
//         get a value >= 0 and <= (totaltickets - 1)
int winner = getrandom(0, totaltickets);

// current: use this to walk through the list of jobs
node_t *current = head;
while (current) {
    counter = counter + current->tickets;
    if (counter > winner)
        break; // found the winner
    current = current->next;
}
// 'current' is the winner: schedule it...
```

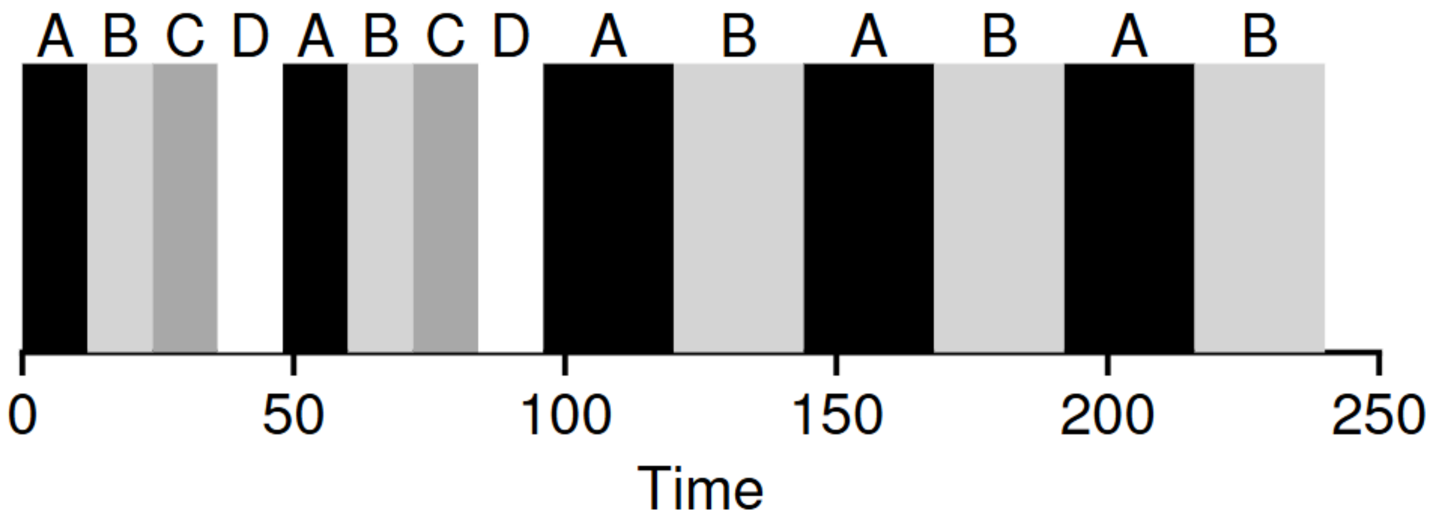
An optimization: organize the list in sorted order

Completely Fair Scheduler (CFS)

- The default Linux scheduler since v2.6.23 (2007).
  - The goal of CFS: to fairly divide a CPU evenly among all competing processes.
- CFS uses a counting-based technique known as **virtual runtime** (vruntime)
  - As each process runs, it accumulates vruntime, e.g., in proportion with the physical (real) time.
  - When a scheduling decision occurs, CFS will pick the process with the lowest vruntime to run next.
- How does CFS know when to stop the running process?
  - The scheduling time slice. Either too large or small. Why?

Completely Fair Scheduler (CFS)

- CFS decides the scheduling interval based on the number of currently running processes.
  - *sched_latency* divided by the number of processes
    - ❑ why?
  - E.g., 48 milliseconds / 4 processes = 12 milliseconds
  - What if there are too many processes? Set a minimal value of time slice: *min_granularity*.



Completely Fair Scheduler (CFS)

- CFS also enables controls over process priority, enabling users to give some processes a higher share of the CPU.
  - Using a UNIX mechanism known as the nice level of a process.
  - Larger nice, lower priority.

```
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */  9548,  7620,  6100,  4904,  3906,
    /*  -5 */  3121,  2501,  1991,  1586,  1277,
    /*   0 */  1024,   820,   655,   526,   423,
    /*   5 */   335,   272,   215,   172,   137,
    /*  10 */   110,    87,    70,    56,    45,
    /*  15 */    36,    29,    23,    18,    15,
};
```

Two jobs: A with nice value of -5, B with nice value of 0. sched_latency is 48ms. What is the time slice of A and B?

$$\mathtt{time\_slice}_k = \frac{\mathtt{weight}_k}{\sum_{i=0}^{n-1} \mathtt{weight}_i} \cdot \mathtt{sched\_latency}$$

Completely Fair Scheduler (CFS)

$$\texttt{vruntime}_i = \texttt{vruntime}_i + \frac{\texttt{weight}_0}{\texttt{weight}_i} \cdot \texttt{runtime}_i$$

weight_0 is the weight of process with default priority (1024)

```
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */  9548,  7620,  6100,  4904,  3906,
    /*  -5 */  3121,  2501,  1991,  1586,  1277,
    /*   0 */  1024,   820,   655,   526,   423,
    /*   5 */   335,   272,   215,   172,   137,
    /*  10 */   110,    87,    70,    56,    45,
    /*  15 */    36,    29,    23,    18,    15,
};
```

$$\texttt{time\_slice}_k = \frac{\texttt{weight}_k}{\sum_{i=0}^{n-1} \texttt{weight}_i} \cdot \texttt{sched\_latency}$$

Completely Fair Scheduler (CFS)

$$\mathtt{vruntime}_i = \mathtt{vruntime}_i + \frac{\mathtt{weight}_0}{\mathtt{weight}_i} \cdot \mathtt{runtime}_i$$

weight_0 is the weight of process with default priority (1024)

How those "magic numbers are determined"?

```
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */  9548,  7620,  6100,  4904,  3906,
    /*  -5 */  3121,  2501,  1991,  1586,  1277,
    /*   0 */  1024,   820,   655,   526,   423,
    /*   5 */   335,   272,   215,   172,   137,
    /*  10 */   110,    87,    70,    56,    45,
    /*  15 */    36,    29,    23,    18,    15,
};
```

$$\mathtt{time\_slice}_k = \frac{\mathtt{weight}_k}{\sum_{i=0}^{n-1} \mathtt{weight}_i} \cdot \mathtt{sched\_latency}$$

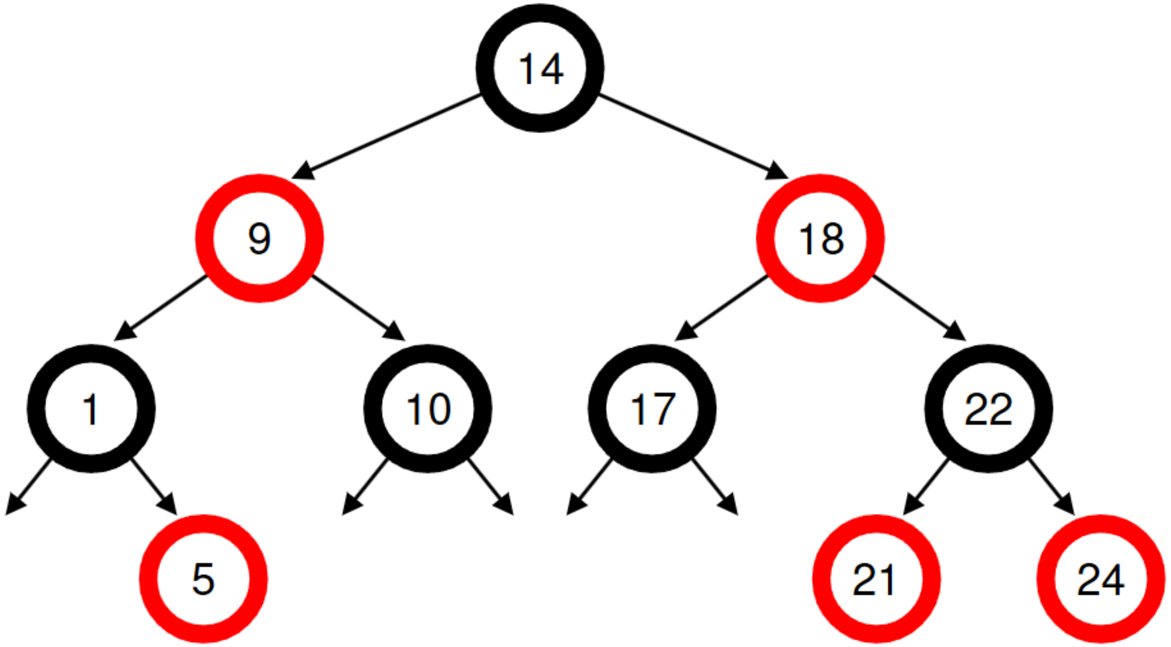
Think about 2 case:
(1) A with nice value 0, B with nice value -5
(2) A with nice value 5, B with nice value 0
Calculate how they will be scheduled

Completely Fair Scheduler (CFS)

- Implementing CFS
  - The ops to be supported: (1) finding the process with lowest vruntime; (2) insert/delete a process
- Approach#1: Ordered List
  - Finding the next job: O(1)
  - Insert/delete: O(n)

Completely Fair Scheduler (CFS)

- Implementing CFS
  - The ops to be supported: (1) finding the process with lowest vruntime; (2) insert/delete a process
- Approach#1: Ordered List
  - Finding the next job: O(1)
  - Insert/delete: O(n)
- Approach#2: Red-Black Tree
  - Finding the next job: O(log n)
  - Insert/delete: O(log n)
    - ❑A node is either red or black
    - ❑The root is black
    - ❑All leaves (NULL) are black
    - ❑Both children of every red node are black
    - ❑Every simple path from root to leaves contains the same number of black nodes.

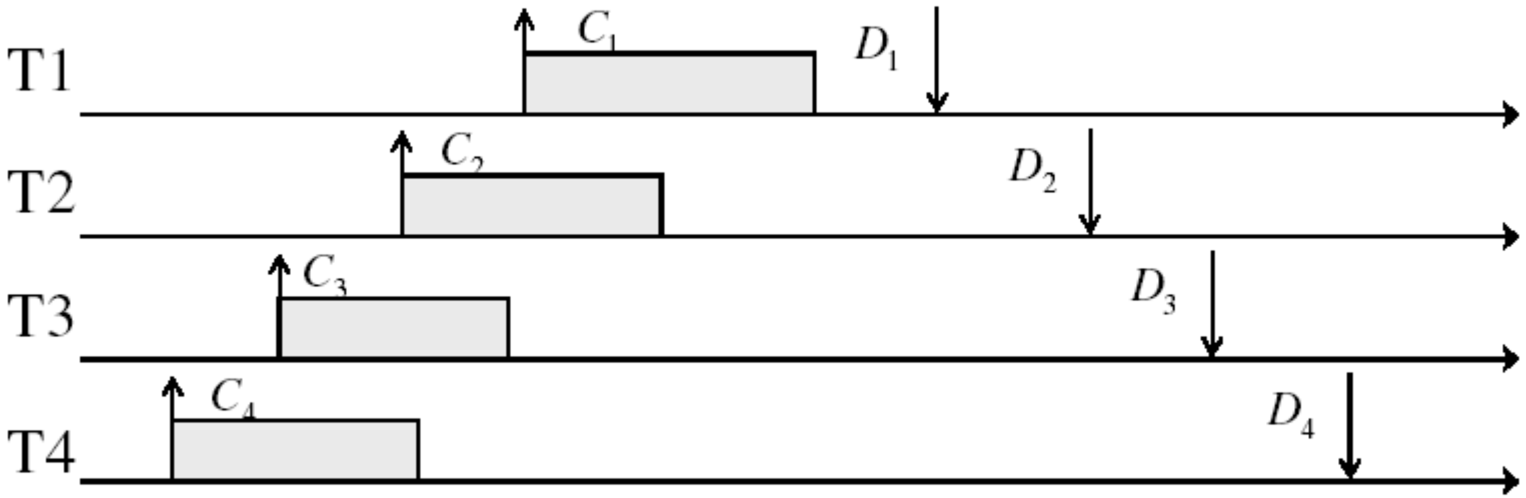


Real-Time Scheduling (RTS)

- Efficiency is important but predictability is essential:
  - We need to predict with confidence worst case response times for systems
  - In RTS, performance guarantees are:
    - ❑ Task- and/or class centric and often ensured a priori
  - In conventional systems, performance is:
    - ❑ System/throughput oriented with post-processing (… wait and see …)
  - Real-time is about enforcing predictability, and does not equal fast computing!!!
- Hard Real-Time
  - *Attempt to meet all deadlines*
  - EDF (Earliest Deadline First), LLF (Least Laxity First), RMS (Rate-Monotonic Scheduling), DM (Deadline Monotonic Scheduling)
- Soft Real-Time
  - *Attempt to meet deadlines with high probability*
  - Minimize miss ratio / maximize completion ratio (firm real-time)
  - Important for multimedia applications
  - CBS (Constant Bandwidth Server)

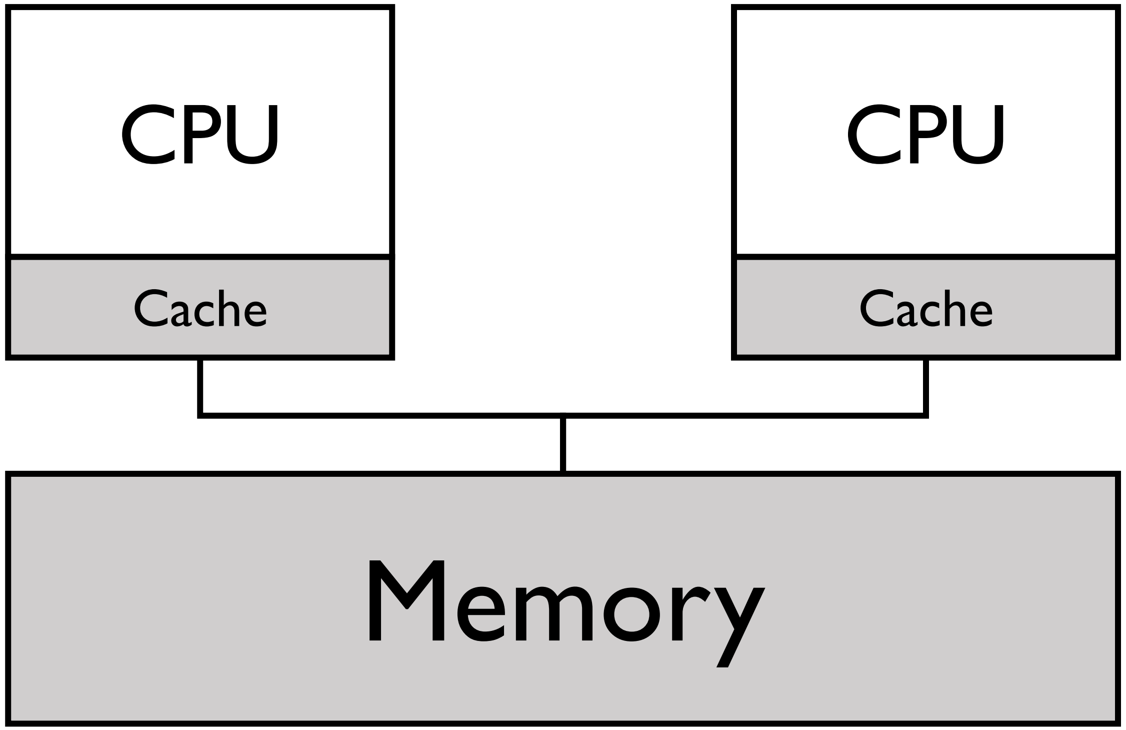
Real-Time Scheduling (RTS)

- Tasks are preemptable, independent with arbitrary arrival (=release) times
- Tasks have deadlines (D) and known computation times (C)
- Example Setup:



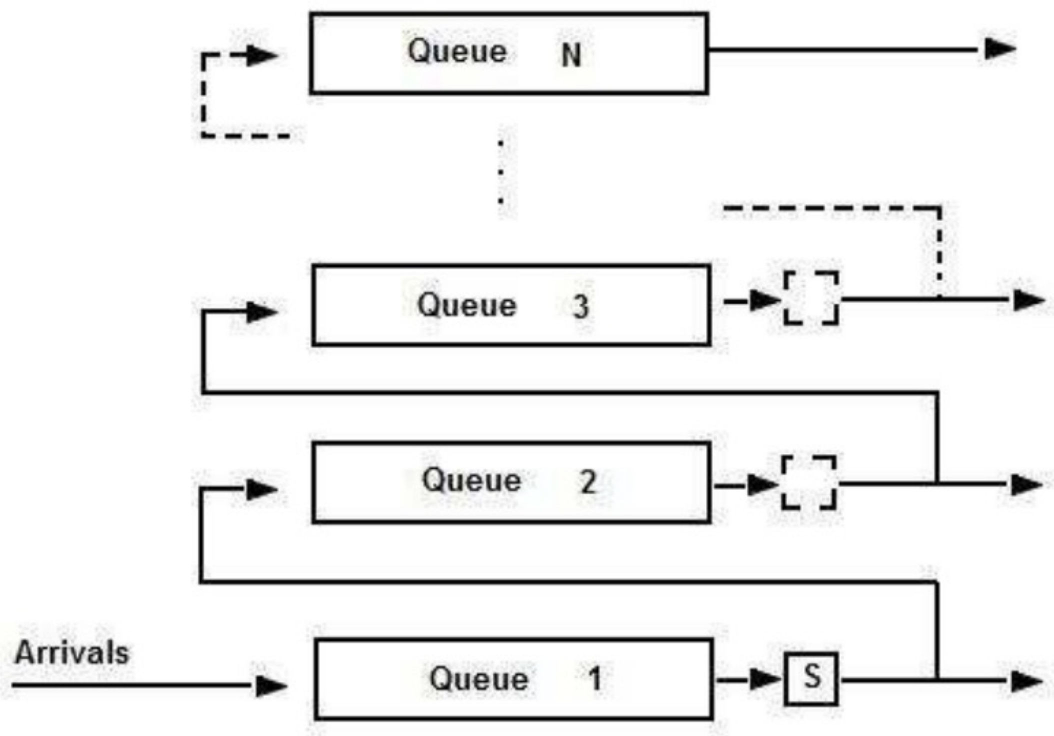
Scheduling on Multiprocessor

- Recall: the cache-memory system, and cache consistency (or coherency) (缓存一致性)



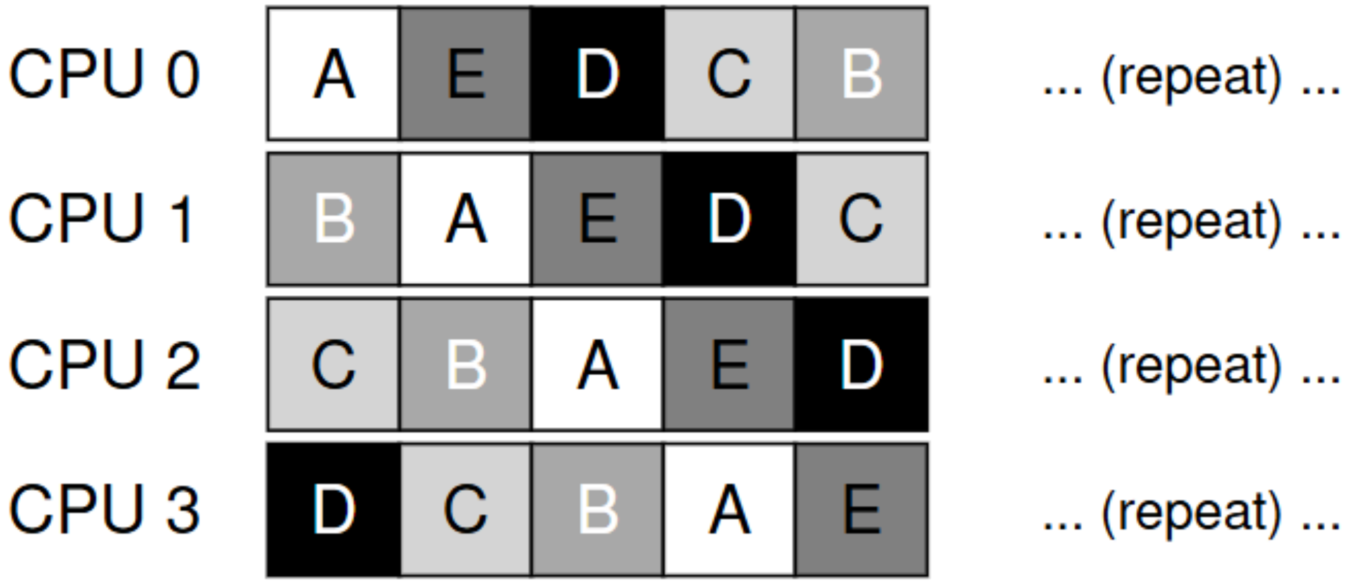
Scheduling on Multiprocessor

- What's wrong with a centralized MFQ?



Scheduling on Multiprocessor

- What's wrong with a centralized MFQ?
  - Contention for the MFQ lock
    - ❑ The lock could become a bottleneck especially with large number of processors
  - Cache coherence overhead
    - ❑ The MFQ data structure will be modified often and cause cache miss when a processor gets its lock to use MFQ



Assuming we have 5 jobs (A, B, C, D, E) running repeatedly in order on 4 CPUs.

Bad cache hit ratio!

Scheduling on Multiprocessor

- What's wrong with a centralized MFQ?
  - Contention for the MFQ lock
    - ❑ The lock could become a bottleneck especially with large number of processors
  - Cache coherence overhead
    - ❑ The MFQ data structure will be modified often and cause cache miss when a processor gets its lock to use MFQ
  - Limited cache reuse
    - ❑ A thread is likely to be scheduled on different processors, so the L1 cache needs to be fetched from the memory again

Scheduling on Multiprocessor

- What's wrong with a centralized MFQ?
  - Contention for the MFQ lock
  - Cache coherence overhead
  - Limited cache reuse
- Modern OSes use per-processor MFQ
- Affinity scheduling (亲和性调度): a thread is always (re)scheduled to the same processor
  - Rebalancing across processors only happens when necessary

How to Evaluate a Scheduling algorithm?

- Deterministic modeling
  - Takes a predetermined workload and compute the performance of each algorithm for that workload
- Queueing models (排队论/模型)
  - Mathematical approach for handling stochastic workloads
  - Commonly used in a variety of fields, including computer science, telecommunications, operations research, and industrial engineering
- Implementation/Simulation:
  - Build system which allows actual algorithms to be run against actual data – most flexible/general

Summary of Scheduling Algorithms

- Round-Robin Scheduling:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs
- Shortest Job First (SJF) / Shortest Remaining Time First (SRTF):
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
  - Pros: Optimal (average response time)
  - Cons: Hard to predict future, Unfair
- Multi-Level Feedback Scheduling:
  - Multiple queues of different priorities and scheduling algorithms
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF

Summary of Scheduling Algorithms

- Lottery Scheduling:
  - Give each thread a priority-dependent number of tokens (short tasks ⇒ more tokens)
- Linux CFS
  - Completely fair across processes (always assign to the one with least running time)
  - Dynamically adjust time slice of each process
  - Using priority (nice level) to control the assignment
- Real-time scheduling
  - Need to meet a deadline, predictability essential
  - Earliest Deadline First (EDF) and Rate Monotonic (RM) scheduling

Summary of Scheduling Algorithms

- This course only covers very basic knowledge of scheduling
  - The schedulers used in real OSes are more complex
  - Choosing a proper schedule depends on many factors: hardware, workloads, etc..
  - Note: almost every hardware resource needs scheduler..
    - GPU, disk, network, etc..
  - Scheduling is common in real-world life
    - Use what you learned to solve them!
    - Example #1: Hospital emergencies?
    - Example #2: Air traffic control?
    - Example #3: Supermarket checkout?
    - Example #4: Print jobs in a printer?
    - Example #5: Control system in a rocket?
    - Example #6: Engine control unit in an automotive application

Homework

- Some simple code about MLFQ. Check out our website.