Operating System: Multiprocessor Scheduling

Sang Ho Choi (shchoi@kw.ac.kr)
School of Computer & Information Engineering
KwangWoon University

Multiprocessor Scheduling

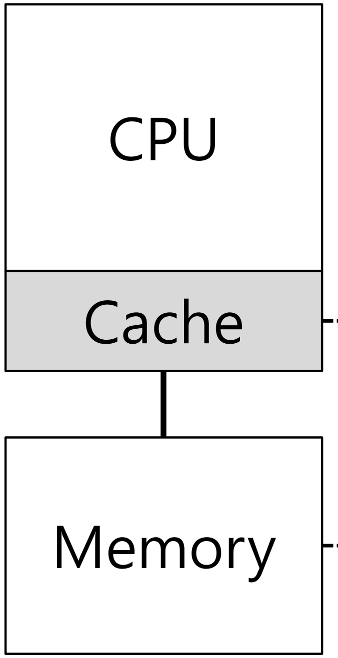
- The rise of the multicore processor is the source of multiprocessor-scheduling proliferation 멀티프로세서의 scheduling을 어떻게 구성할것인가?
  - **Multicore:** Multiple CPU cores are packed onto a single chip

How to schedule jobs on Multiple CPUs?

- Adding more CPUs does not make that single application run faster 단일 어플리케이션의 속도가 빨라지는게 X
  - You'll have to rewrite application to run in parallel, using **threads** 병렬로 스레드를 사용

Single CPU with cache

SRAM



Cache

- Small, fast memories
- Hold copies of popular data that is found in the main memory 많이 사용하는것을 캐싱해둠.
- Utilize *temporal* and *spatial* locality 시간지역성 공간지역성.

Main Memory DRAM

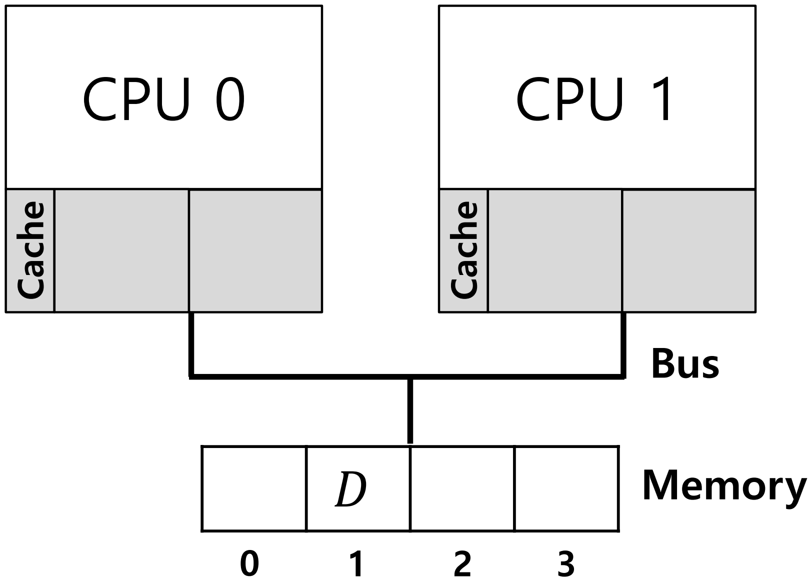
- Holds all of the data
- Access to main memory is slower than cache

By keeping data in cache, the system can make slow memory appear to be a fast one

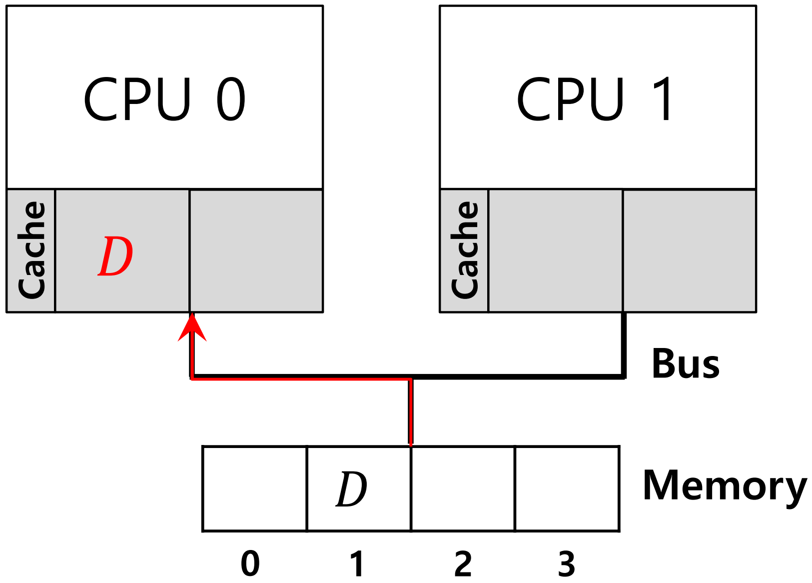
Cache coherence

- Consistency of shared resource data stored in multiple caches

0. Two CPUs with caches sharing memory

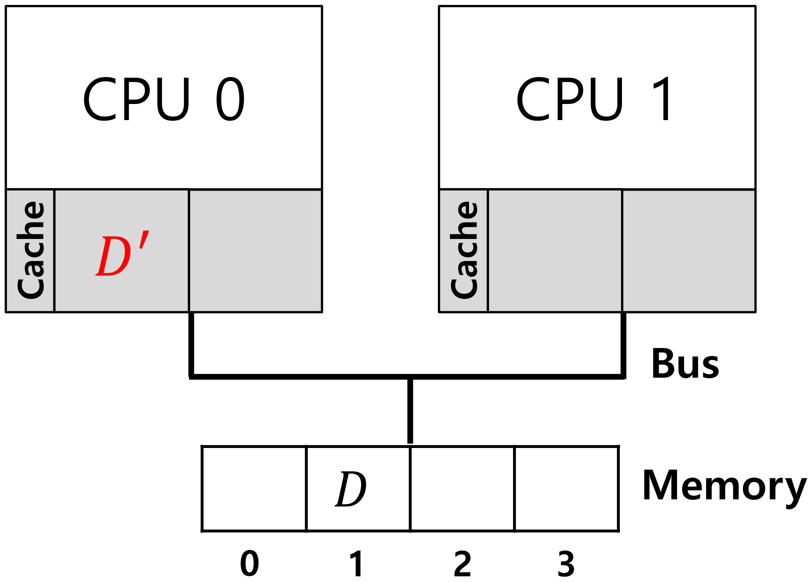


1. CPU0 reads a data at address 1.

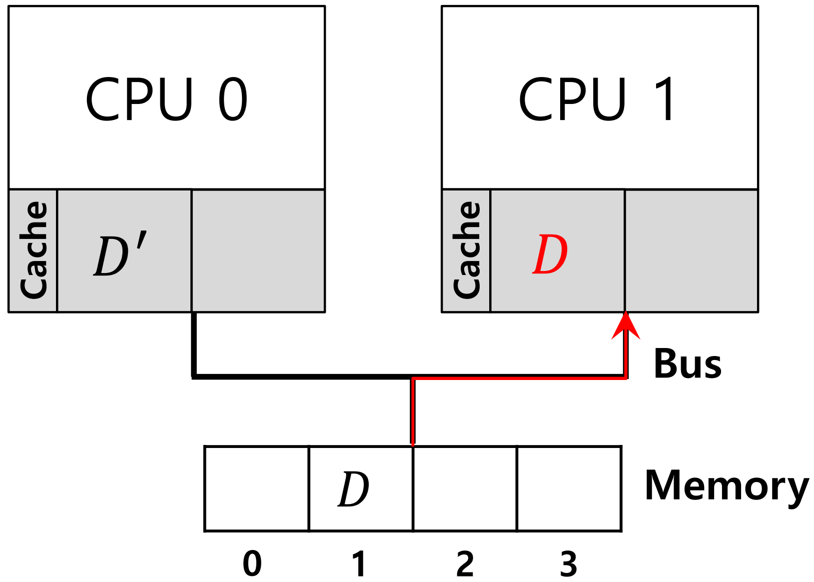


Cache coherence (Cont.)

2. D is updated and CPU1 is scheduled.



3. CPU1 re-reads the value at address A



CPU1 gets the old value D
instead of the correct value D'

Cache coherence solution

- Bus snooping
  - Each cache pays attention to memory updates by **observing the bus**
  - When a CPU sees an update for a data item it holds in its cache, it will notice the change and either <u>invalidate</u> its copy or <u>update</u> it

Don’t forget synchronization

- When accessing shared data across CPUs, mutual exclusion primitives should likely be used to guarantee correctness

```c
typedef struct __Node_t {
        int value;
        struct __Node_t *next;
} Node_t;

int List_Pop() {
        Node_t *tmp = head;          // remember old head ...
        int value = head->value;     // ... and its value
        head = head->next;           // advance head to next pointer
        free(tmp);                   // free old head
        return value;                // return value at head
}
```

Simple List Delete Code

Don’t forget synchronization (Cont.)

- Solution

```
pthread_mtuex_t m;
typedef struct __Node_t {
        int value;
        struct __Node_t *next;
} Node_t;

int List_Pop() {
        lock(&m)
        Node_t *tmp = head;            // remember old head ...
        int value = head->value;       // ... and its value
        head = head->next;             // advance head to next pointer
        free(tmp);                     // free old head
        unlock(&m)
        return value;                  // return value at head
}
```

Simple List Delete Code with lock

CPU 0 CPU 1

process

이렇게하면 ↑ 처럼 구현은 할수 있으나 성능이 ↓ 떨어지니

Cache Affinity

친화성.

프로세스 하나는 CPU 하나가 처리하기 기존에 관련있는 캐시데이터를 재활용 할수 있으니.

- Keep a process on the same CPU if at all possible
  - A process builds up a fair bit of state in the cache of a CPU
  - The next time the process run, it will run faster if some of its state is *already present* in the cache on that CPU

A multiprocessor scheduler should consider cache affinity when making its scheduling decision

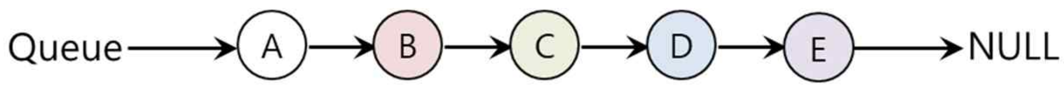
Single queue Multiprocessor Scheduling (SQMS)

- Put all jobs that need to be scheduled into a single queue
  - Each CPU simply picks the next job from the globally shared queue
  - Cons:

경쟁높기. 1) Some form of **locking** have to be inserted → Lack of scalability 확장성.

2) Cache affinity

✓ Ex) Possible job scheduler across CPUs:



CPU0	A	E	D	C	B	... (repeat) ...
CPU1	B	A	E	D	C	... (repeat) ...
CPU2	C	B	A	E	D	... (repeat) ...
CPU3	D	C	B	A	E	... (repeat) ...

Scheduling Example with Cache affinity

캐시친화성을 적용

Queue → A → B → C → D → E → NULL

CPU0	A	E	A	A	A	... (repeat) ...
CPU1	B	B	E	B	B	... (repeat) ...
CPU2	C	C	C	E	C	... (repeat) ...
CPU3	D	D	D	D	E	... (repeat) ...

– Preserving affinity for most
  - ➢ Jobs A through D are not moved across processors
  - ➢ Only job e Migrating from CPU to CPU
– Implementing such a scheme can be **complex**

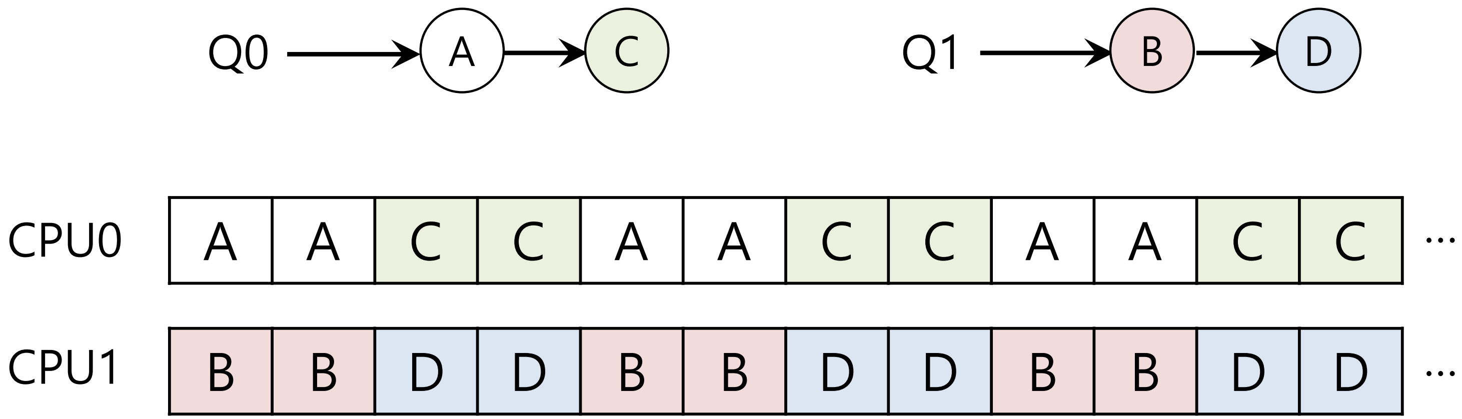
Multi-queue Multiprocessor Scheduling (MQMS)

1Q – 1CPU

- MQMS consists of multiple scheduling queues
  - Each queue will follow a particular scheduling discipline
  - When a job enters the system, it is placed on **exactly one** scheduling queue
  - Avoid the problems of information sharing and synchronization

MQMS Example

- With **round robin**, the system might produce a schedule that looks like this:



MQMS provides more scalability and cache affinity

Load Imbalance issue of MQMS

CPU utilization이 떨어짐.
특정큐에 몰리게되면.

- After job C in Q0 finishes:

Q0 → A   Q1 → B → D

CPU0	A	A	A	A	A	A	A	A	A	A	A	A	…
CPU1	B	B	D	D	B	B	D	D	B	B	D	D	…

A gets twice as much CPU as B and D

- After job A in Q0 finishes:

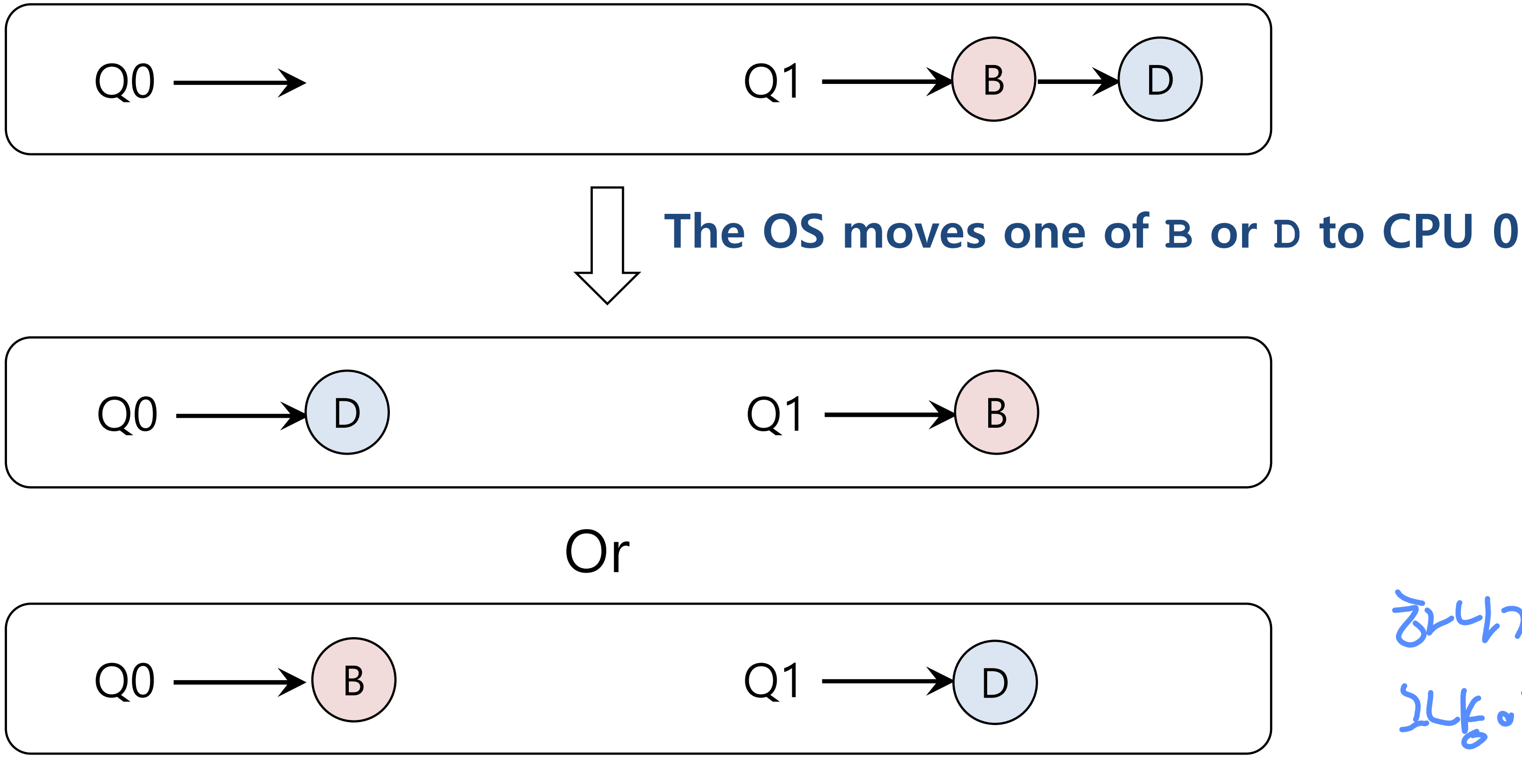
Q0 →   Q1 → B → D

CPU0													…
CPU1	B	B	D	D	B	B	D	D	B	B	D	D	…

CPU0 will be left idle!

How to deal with load imbalance?

- The answer is to move jobs (Migration)
  - Example:



하나가 비면
그냥 이동하면 되는데

How to deal with load imbalance? (Cont.)

- A more tricky case:

애매하게 밸런스가 안맞으면 관련 알고리즘이 필요.

Q0 → A   Q1 → B → D

- A possible migration pattern:
  - Keep switching jobs

CPU0	A	A	A	A	B	A	B	A	B	B	B	B	…
CPU1	B	D	B	D	D	D	D	D	A	D	A	D	…

Migrate `B` to CPU0   **Migrate `A` to CPU1**

Work Stealing

- Move jobs between queues
  - Implementation:
    - ➢ A source queue that is <u>low on jobs</u> is picked

      job의 개수가 가장 적은 큐를 source queue.
    - ➢ The source queue occasionally peeks at another target queue

      소스큐가 다른 큐를 바라보다가
    - ➢ If the target queue is <u>more full than</u> the source queue, the source will "**steal**" one or more jobs from the target queue

      소스큐가 뺏어감
  - Cons:
    - ➢ *High overhead* and trouble *scaling*

steal 때문에 생기는 overhead 나 확장성 문제가 있을 수 있어 설계에 주의해야함.

Linux Multiprocessor Schedulers

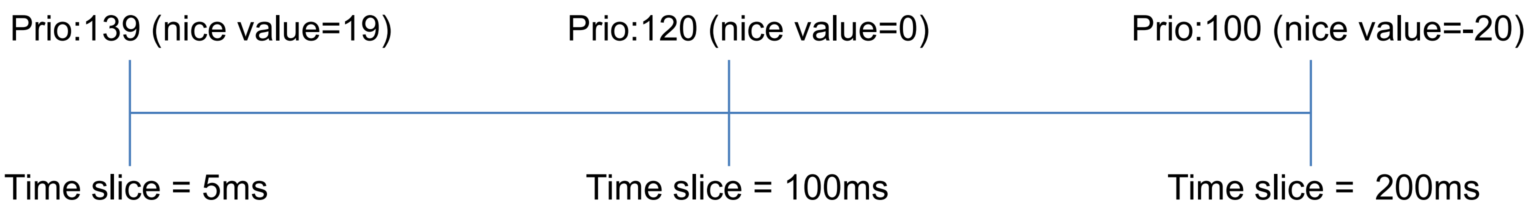
- Time-sharing with time slice
  - Only processes with time slice can execute
  - Time slice is subtracted when timer interrupt occurs
  - When time slice = 0, another process is chosen
  - When all processes have time slice = 0, recalculation is performed

- Real-time with priority
  - Highest priority process always runs first
  - 140 priority levels
    - The lower the value, the higher priority
    - E.g. priority level 110 will have a higher priority than 130

- Linux schedules the highest priority process with time slice

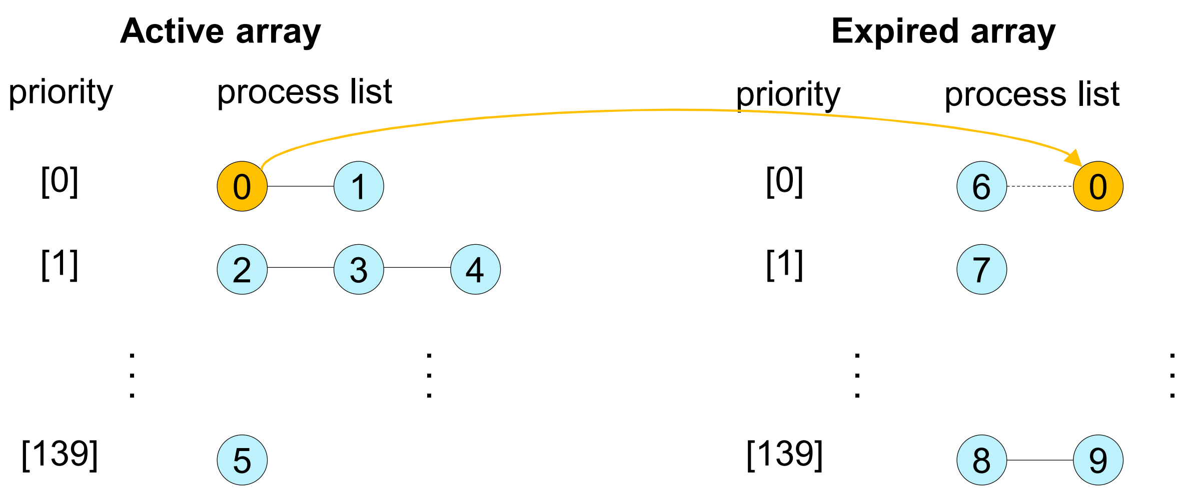
Linux Multiprocessor Schedulers (Cont.)

- O(1) scheduler
  - It selects a highest priority process in a ready queue within a constant amount of time, regardless of how many processes are running
  - It divides the ready queue into multiple queues according to priority
    - ➢ 0~99 for real time processes
    - ➢ 100-139 for non-real time processes
  - Time slice is given based on the priority



Linux Multiprocessor Schedulers (Cont.)

- O(1) scheduler
  - Active array - list of processes with time remaining in their time slices
  - Expired array - list of expired processes
  - The scheduler chooses the process with the highest priority from the active array



  - When the active array is empty, the two arrays are exchanged

모두 종료되면 Active array가 비니까 이제
expired array로 사용하는거지, expired는 Active로 쓰고.

Linux Multiprocessor Schedulers (Cont.)

- CFS (Complete Fair Scheduling) scheduler
  - Since kernel 2.6.23.
  - Proposed to overcome some oddity of O(1) scheduler
  - Time slice based on priority in O(1) scheduler

우선순위 차이는 비슷한데
차이가 별로 안나.

process 1 (prio=120)	process 2 (prio=121)	➔ Time slice = 100ms, 95ms	
process 3 (prio=138)	process 4 (prio=139)	➔ Time slice = 10ms, 5ms	➔ relatively 2 times

차이가 별로 X

2배

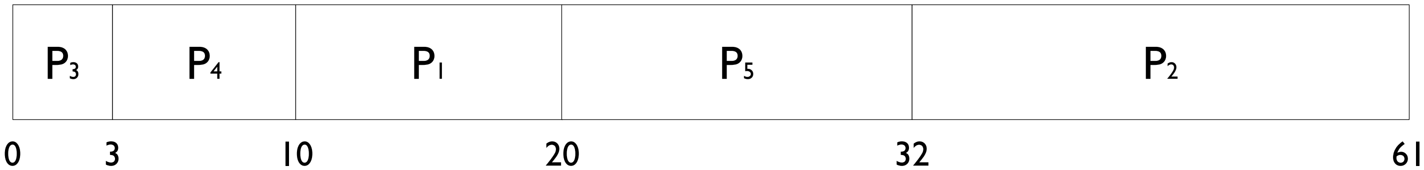
  - Time slice based on weight in CFS scheduler
    - Roughly 1.25 times for each priority level difference

process 5 (prio=n)	process 6 (prio=n+1)	➔ Time slice = 56ms, 44ms	➔ relatively 1.25 times
process 7 (prio=m)	process 8 (prio=m+5)	➔ Time slice = 75ms, 25ms	➔ relatively 3 times

Algorithm Evaluation

- Deterministic modeling
  - takes a particular predetermined workload, and then
  - defines the performance of each algorithm for that workload

  - E.g. SJF with Gantt Chart, when given a set of processes



  - Simple and fast
  - Useful in describing the algorithm and providing examples

Algorithm Evaluation

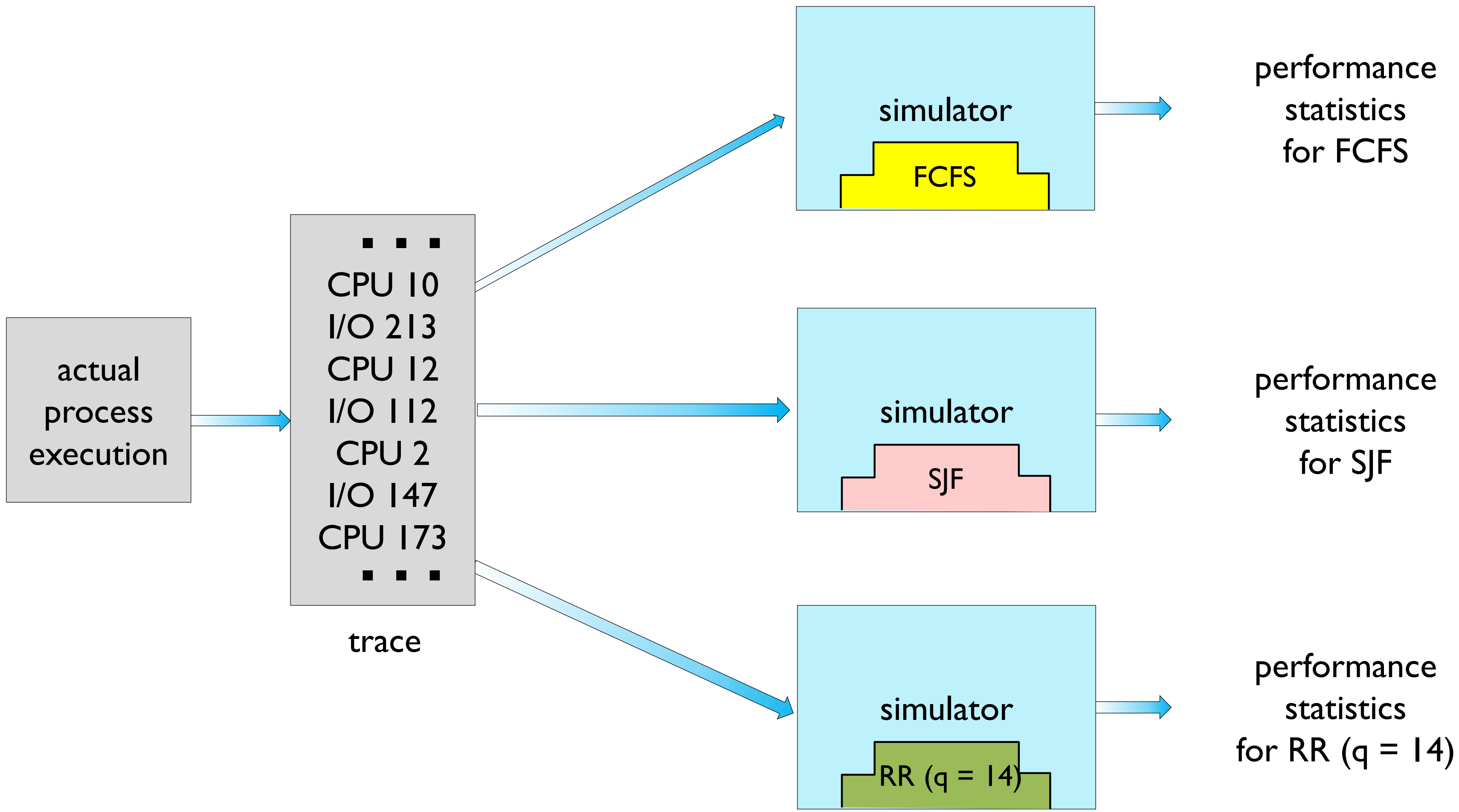
- Implementation
  - Only completely accurate way to evaluate algorithm
  - is very difficult
    - ➢algorithm coding
    - ➢operating system modification

Algorithm Evaluation

- Simulation
  - programs a model of computer system
    - Data structures represent the major components of system
    - E.g. a variable for clock
  - Random number generator is required
    - The number of processes, CPU burst times, arrivals, departures, etc
    - Uniform, exponential, poisson, etc
  - Can be expensive
    - Design, coding, and debugging of simulator is time consuming
    - Large storage space and time may be required

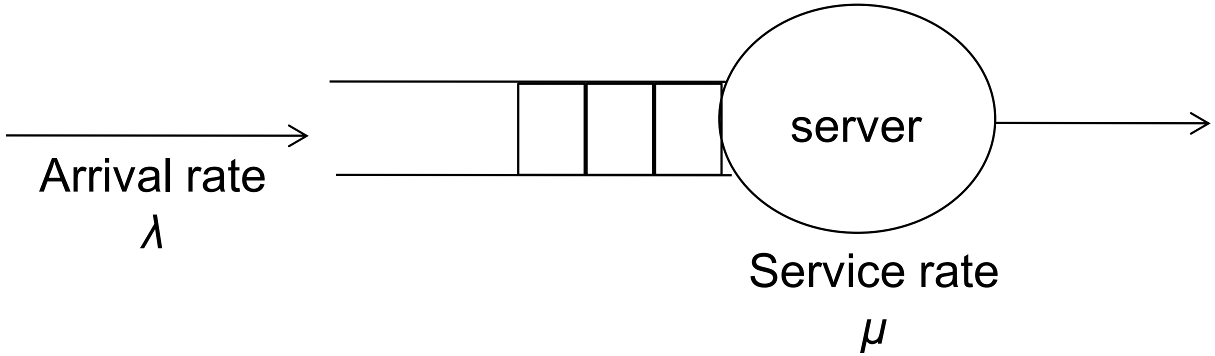
Algorithm Evaluation

- Simulation example



Algorithm Evaluation

- Queueing models
  - with a mathematical formula
  - Simple queueing network model



  - Given arrival rates and service rates,
    - ➢utilization, average queue length, and average waiting time can be derived
  - Limitedly used in some evaluation