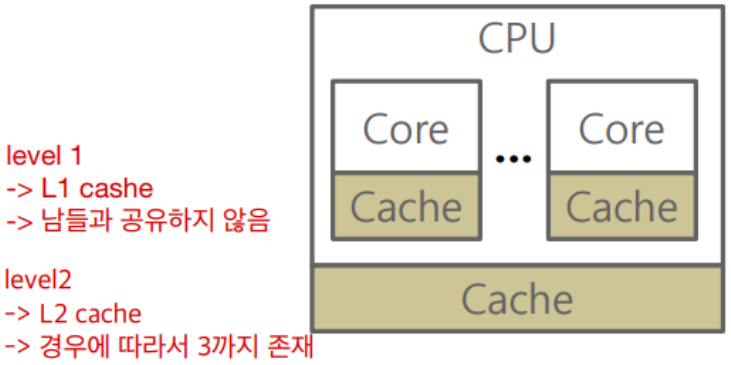
06. Multiprocessor Scheduling

↳CPU가 여러개라면?

▼ Multicore Architecture

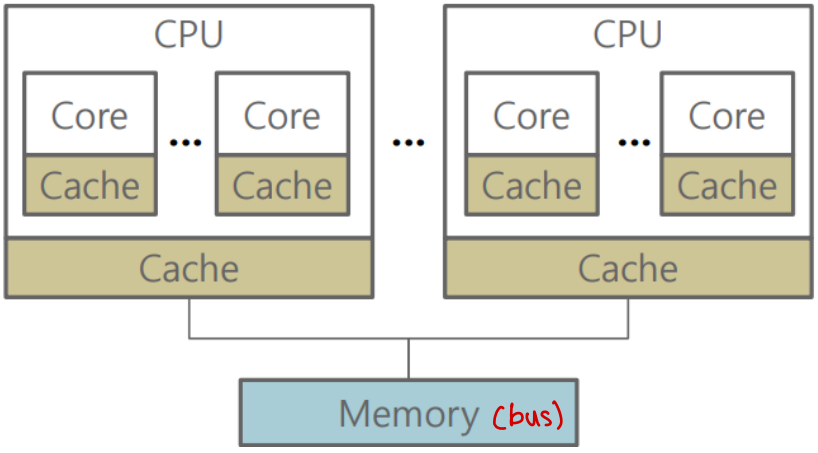
*Single



single CPU

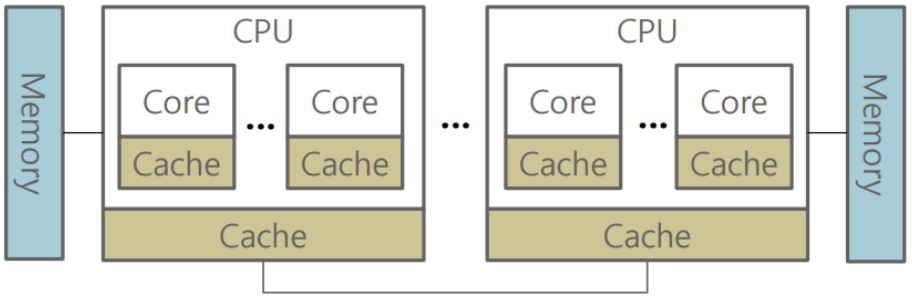
* Multi

- **SMP(symmetric multiprocessor)**



  - cpu와 memory 사이의 package 거리가 모든 cpu에 대해서 동일함
  - 단점
    - memory가 한 곳으로 운용되어 병목 현상 발생 ⇒ numa로 해결

- **NUMA(Non uniform memory access)**



  - cpu마다 가깝게 붙어 있는 메모리가 각각 존재 → 병목 현상 해결
  - memory를 OS가 어디로 잡는가?
    - 응용 program : memory buffer가 필요함. but memory와 거리 구분 불가
    - OS : 거리 파악 후 가까운 걸 주려고 애씀

      (트래픽 분산으로 속도가 빨라지기 위함)

ready queue

▼ Queue Scheduling

cpu : core라고 이해하기

- core 별로 timer, scheduler 존재 → 각각 timer interrupt 발생

⇒ core마다 scheduling 어떻게? → ready queue 개수에 따라 다름

  - 다음에 실행할 process를 ready queue에서만 찾기 때문임

▼ Single-Queue Scheduling (간단하게 생각)

ready queue가 하나인 scheduler (CPU가 여러 개 있다고 하더라도)

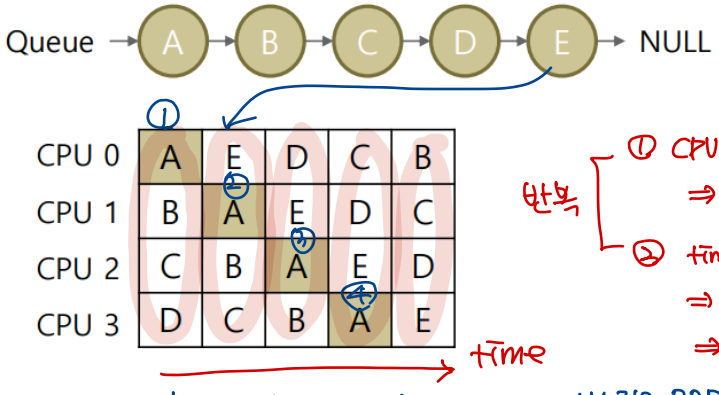
- **Single-queue multiprocessor scheduling (SQMS)**
  - 여태까지 배워온 CPU scheduling algorithm 적용가능
  - 실행할 best n개의 작업 선택 (n = cpu 개수)

달라진 점 → 한꺼번에!!

- 단점
  1. cache affinity (친화도)
     - cache 관점에서 numa architecture에는 별로임

       ex) 기껏 가깝게 가져다 놨는데 cpu에서 쓰는 경우 → single queue bad ☆

       ^ 다른



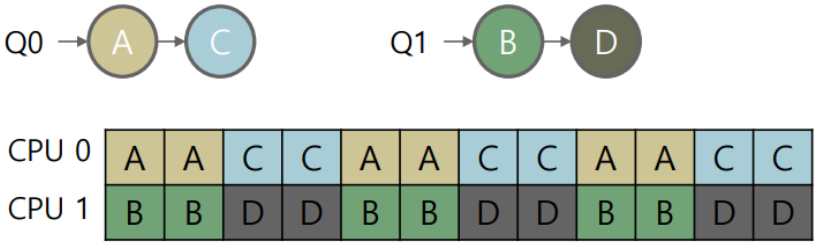
2. Synchronization → Lack of scalability

- 모든 core가 queue를 공유 → race condition 발생
- ready queue에 대해 lock 필요
- 한 scheduler가 하나씩 꺼내갈 때 다른 scheduler lock (Ex. mutex lock)

▼ Multi-Queue Scheduling

core마다 각각의 ready queue가 존재 → synchronoziation 필요 없음

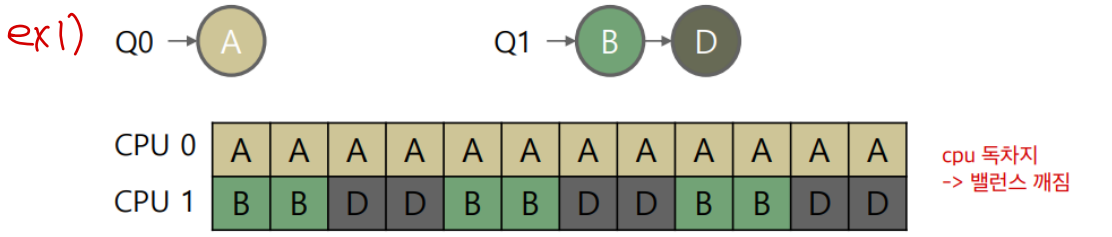
- **Multi-queue multiprocessor scheduling (MQMS)**
  - CPU 당 1개의 ready queue
  - 어떤 process가 도착했을 때 여러 ready queue 중 하나에 넣어줘야 함
    - *random* or *shorter queue*
  - 독립적으로 schedule 되어야 함

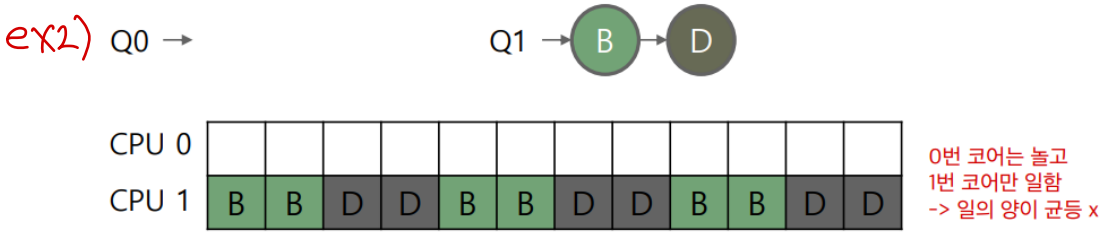


  - OS의 timer interrupt 간격

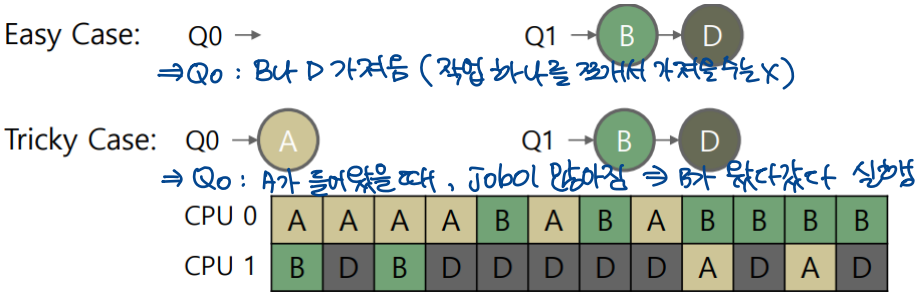
    → core, queue가 많아질수록 전체 시간이 조금 빨라짐

- load imbalance





- 해결책 ⇒ 기본적으로 움직이는 방향이 최선!
  - migration
  - work stealing
    - job이 적은 (src) queue가 다른 (target) queue 훔쳐봄
    - 만약 target queue가 src queue보다 더 많이 차 있다면
      - src queue : target에서 하나 이상의 job 도와줌 → balance load



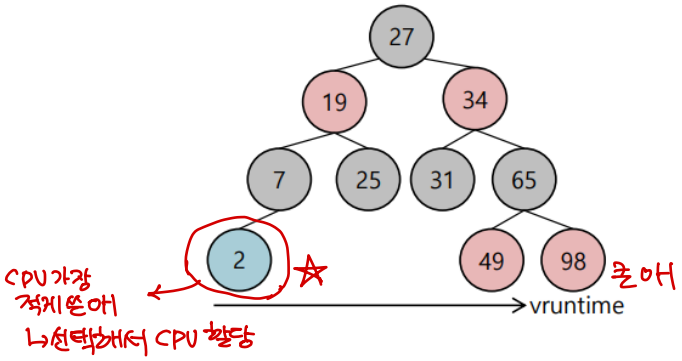
  - 서로 가져오다 보면 어느 쪽에라도 불균형은 생김
  - 한 process가 계속 왔다 갔다 하면서 실행

    → cache affinity BAD

▼ Linux CPU Schedulers → Linux에서 구현된 애들!

▼ Completely Fair Scheduler (CFS) → 우선순위 낮음

⇒ 최대한 cpu 사용 시간(run time)을 공평하게 나눠주려는 scheduler



- ready queue → red black tree *(Weighted fair scheduling)*
  - red black tree : Depth 1 차이로 정렬되어 있는 binary tree
- runtime으로 tree 생성 후 가장 왼쪽 끝에 있는 놈을 고름 ⇒ 제일 작은 놈
- *SCHED_NORMAL*(traditionally called SCHED_OTHER)

- **vruntime(virtual runtime)**
  - 쉬운 task는 각 virtual runtime을 기반으로 red-black tree에 저장함
  - wegihted runtime : 각 process에 대해 nice value(-20~19)를 토대로 계산됨
    - nice value : process마다 cpu 사용 정도를 숫자로 표현하기 위해 가짐
      - OS가 기준을 세우기 위해 가지고 있음
      - nice 할수록 vruntime 크게 가지고 있음

        (내가 사용한 정도보다 더 크게 계산이 되도록)
    - nice value up → *weight_0 / weight_p* 가 1보다 커짐 → 최근 실행 시간 up

$$vruntime = vurnitme + DeltaExec \times \frac{Weight_0}{Weight_p}$$

응용프로그램이 결정하는 값

Nice	$Weight_p$	$Weight_0/Weight_p$
-10	9548	0.107
-5	3121	0.328
0 (default)	1024 ($Weight_0$)	1
5	335	3.057
10	110	9.309

- /proc/<pid>/sched : nice value 확인하는 dir → 저장된 것처럼 보이는 가상 파일 (실제 storage는 X)
  - 현재 process의 pid를 통해 정보 확인 가능

```
root@in9-desktop:/usr/cgroups/cpuset/1# cat /proc/2474/sched
loop (2474, #threads: 1)
-------------------------------------------------------------------
se.exec_start                                :        273032.006506
se.vruntime                                  :         33163.007330
se.sum_exec_runtime                          :         24360.080552
se.nr_migrations                             :                    0
nr_switches                                  :                 4785
nr_voluntary_switches                        :                    0
nr_involuntary_switches                      :                 4785
se.load.weight                               :                 1024
se.avg.load_sum                              :             48066523
se.avg.util_sum                              :             46935823
se.avg.load_avg                              :                 1006
se.avg.util_avg                              :                  983
se.avg.last_update_time                      :       273032006506
policy                                       :                    0
prio    priority                             :                  120
clock-delta                                  :                  150
mm->numa_scan_seq                            :                    0
numa_pages_migrated                          :                    0
numa_preferred_nid                           :                   -1
total_numa_faults                            :                    0
```

(-20 ~ +19)

  - priority = nice + 120
    - CFS: 100 ~ 99
    - 0 ~ 99까지의 우선순위 가지고 있음(작을수록 높음)
      - real-time scheduler를 위해!
  - *renice* command
    - process의 nice value 바꿀 수 있는 command (0 ~ 19)
    - 나의 nice down, 남의 nice up → 이득

    ⇒ 나의 nice 값은 높게만 변경 가능
      - root 권한 가졌을 때 작게 변경 가능 → -1 ~ -20 작게!!

ex) 행렬 연산 많이 하는 process ⇒ CPU bound

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 2714 root      20   0    4204    648    568 R  49.2  0.0   0:29.40 loop
 2474 root      20   0    4204    792    716 R  48.8  0.0   3:06.10 loop
```

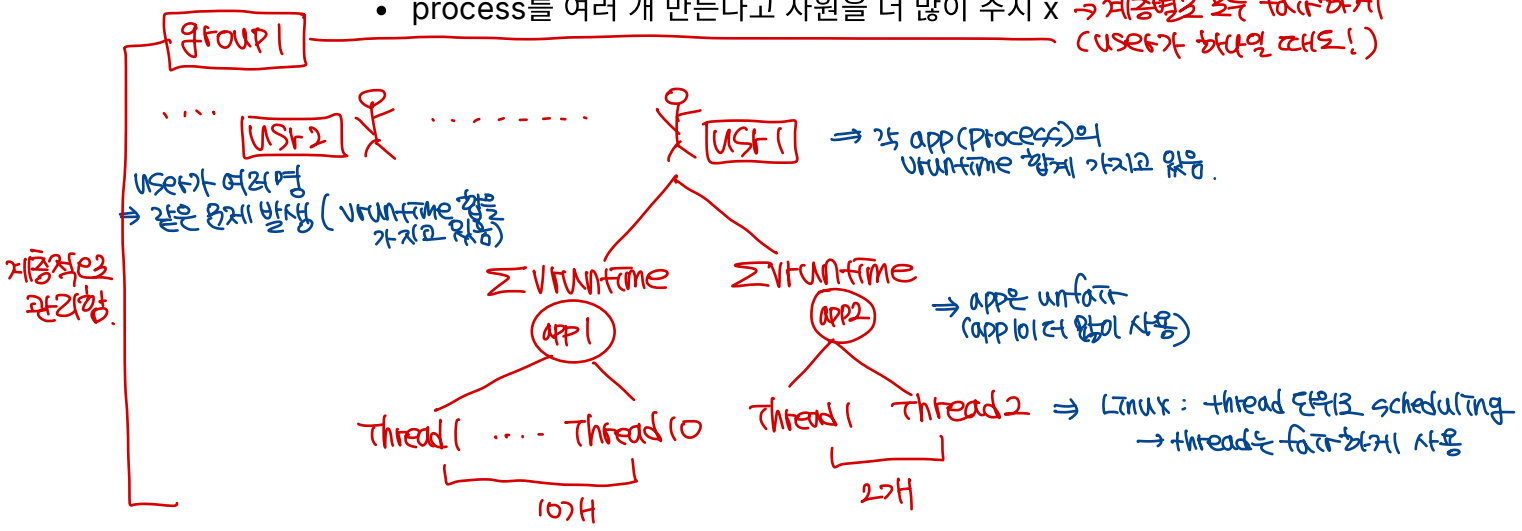
renice 사용해서 nice 값 바꿔본 ver (by root 권한) ⇒ nice value 낮은 process가 CPU 수

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 2474 root      19  -1    4204    792    716 R  53.5  0.0   5:49.98 loop
 2714 root      20   0    4204    648    568 R  42.9  0.0   3:23.21 loop
```

⇒ nice 값 많이 낮출수록 할당 비율 더 차이남.

- **control group (cgroup)** : 동일한 응용 프로그램의 thread가 cgroup 구조로 그룹화
  - 각 계층 별로 fair 할 수 있도록 정보 가지고 있음 (user가 하나일때도)

1. vruntime : 각 thread의 모든 vruntime 합계
2. CFS : cgroup에 알고리즘 적용 → thread 사이의 fair 보장
3. cgroup이 scheduled 되었을 때 가장 낮은 vruntime을 가진 thread가 실행됨
   - group 내의 thread 공정성 보장
   - process를 여러 개 만든다고 자원을 더 많이 주지 x → 계층별로 모두 fair하게 (user가 하나일 때도!)



- Starvation Avoidance
  - CFS : 주어진 시간 내에서 모든 thread를 scheduling → starvation 방지
    - vruntime이 다른 두 thread 실행시간 차이가 preemption period보다 작도록 보장 (ex. 6ms)

- Load Balancing ⇒ multi processor 구조에서 어떻게 scheduling ?
  - Load metric : 낮은 쪽에서 높은 쪽으로 process를 가지고 옴
  - Load of thread : 평균 CPU 사용률 → thread의 우선순위에 따라서 가중치 부여
    - 단위 시간당 CPU 사용량 평균 ↓ 다르게 한 것
  - Load of core : sum of the loads of the threads ⇒ 균등하게 하려고 노력함
  - ⇒ 4ms마다 load balancing → Linux 기준 (최대한 균일하게)
  - ⇒ core : idle에 진입했을 때 periodic load balancer 즉각적으로 호출
  - topology awareness : 아주 정확하게 load balancing 하려고 하지 않음 → 적정선 넘으면 !
    - numar architecture에서 cpu package(numa node) 안에서 하려고 함

⇒ 두 core 사이에 차이 클수록 CFS가 core 균형 맞추기 위해서 Imbalance 더 커지게 됨.
ex) NUMA node가 2개의 system에서 load difference가 발생하는 경우 node 간의 차이가 25%보다 작으면 load balancing X.

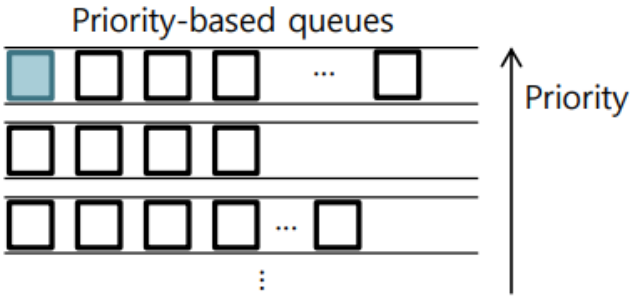
▼ Real-Time Schedulers → 4-2CCH 배움

우선순위 기반 scheduling (우선순위별로 Queue 존재)

⇒ OS가 아니라 응용 SW가 바꿀 수 있음 → 대신 root 권한을 가진 애만 바꿀 수 있음

⇒ 같은 우선 순위에 대해서 fifo로 실행 / time slice마다 바꿔주면서 실행

SCHED_FIFO SCHED_RR



- sched_setattr() ⇒ System Call 사용하여 Process 우선순위 높여주기

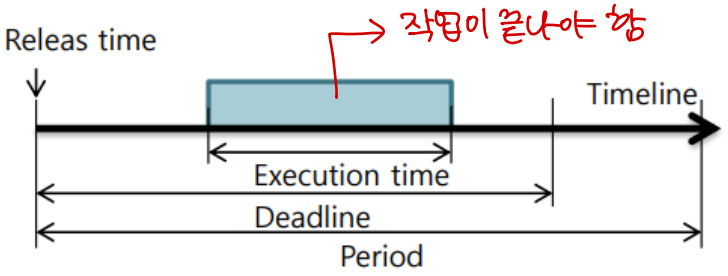
▼ Deadline Scheduler → 우선순위 높음

자동차, 비행기에 들어가는 제어와 관련된 sw 사용(EDF-like periodic scheduler)

→ dead line 제일 짧은 놈부터 실행

→ 정해진 시간마다 일을 하는 period process model에서 사용

⇒ OS : 얼만큼의 period에서 얼만큼 cpu 자원을 할당해줄 건지 사용



- sched_setattr( )
- SCHED_DEADLINE

⇒ root 권한 있어야 두 가지 scheduler 사용 가능

▼ src code → Linux Kernel v5.11.8

- Scheduling classes (/kernel/sched/sched.h)

```
struct sched_class {
  void (*enqueue_task) (struct rq *rq,
  struct task_struct *p,
  int flags);
  void (*dequeue_task) (struct rq *rq,
  struct task_struct *p,
  int flags);
  ...
  struct task_struct *(*pick_next_task)(struct rq *rq);
  ...
  int (*balance)(struct rq *rq, struct task_struct *prev,
  struct rq_flags *rf);
  ...
}
```

```
extern const struct sched_class dl_sched_class;
extern const struct sched_class rt_sched_class;
extern const struct sched_class fair_sched_class;
extern const struct sched_class idle_sched_class;
```

scheduler 구현 → 해당 구조체 모두 채워야 함

- CFS scheduling class (/kernel/sched/fair.c)

```
DEFINE_SCHED_CLASS(fair) = {
  .enqueue_task = enqueue_task_fair,
  .dequeue_task = dequeue_task_fair,
  ...
  .pick_next_task = __pick_next_task_fair,
  ...
  .balance = balance_fair,
  ...
}
```

CFS로 실행하기 위해 초기화된 모습 → 해당 방법에 대한 함수가 연결되어 있음

(ready queue)

- Per-CPU runqueue (/kernel/sched/sched.h)

```
struct rq {
  raw_spinlock_t lock;
  ...
  struct cfs_rq cfs;
  struct rt_rq rt;
  struct dl_rq dl;
  ...
  struct task_struct *curr;
  struct task_struct *idle;
}
```

→ ready queue (run queue)

scheduler

→ current process : 현재 running process

→ idle process의 PCB 가리킴

- CPU scheduler (/kernel/sched/core.c)

```
static void __sched notrace __schedule(bool preempt)
{
  ...
  prev = rq->curr;
  ...
  next = pick_next_task(rq, prev, &rf);
  ...
  rq = context_switch(rq, prev, next, &rf);
  ...
}
```

⇒ 다음에 실행할 process pick

schedule() → __schedule() ⇒ 스케줄링 관련된 내용들이 구현되어 있음

```
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev,
  struct rq_flags *rf)
  {
    ...
    put_prev_task_balance(rq, prev, rf);
    for_each_class(class) {
    p = class->pick_next_task(rq);
    if (p)
    return p;
  }
  ...
}
```

→ schedule class마다 pick_next_task pointer가 존재

⇒ 높은 우선순위에 존재하는 scheduler부터 불러옴.

⇒ 아무도 없게되면 idle process 등장

schedule() → __schedule() → pick_next_task()