Operating System 실습
[7주차]
Process scheduling

한양대학교
HANYANG UNIVERSITY

Data-Centric Systems Lab.

Process switch

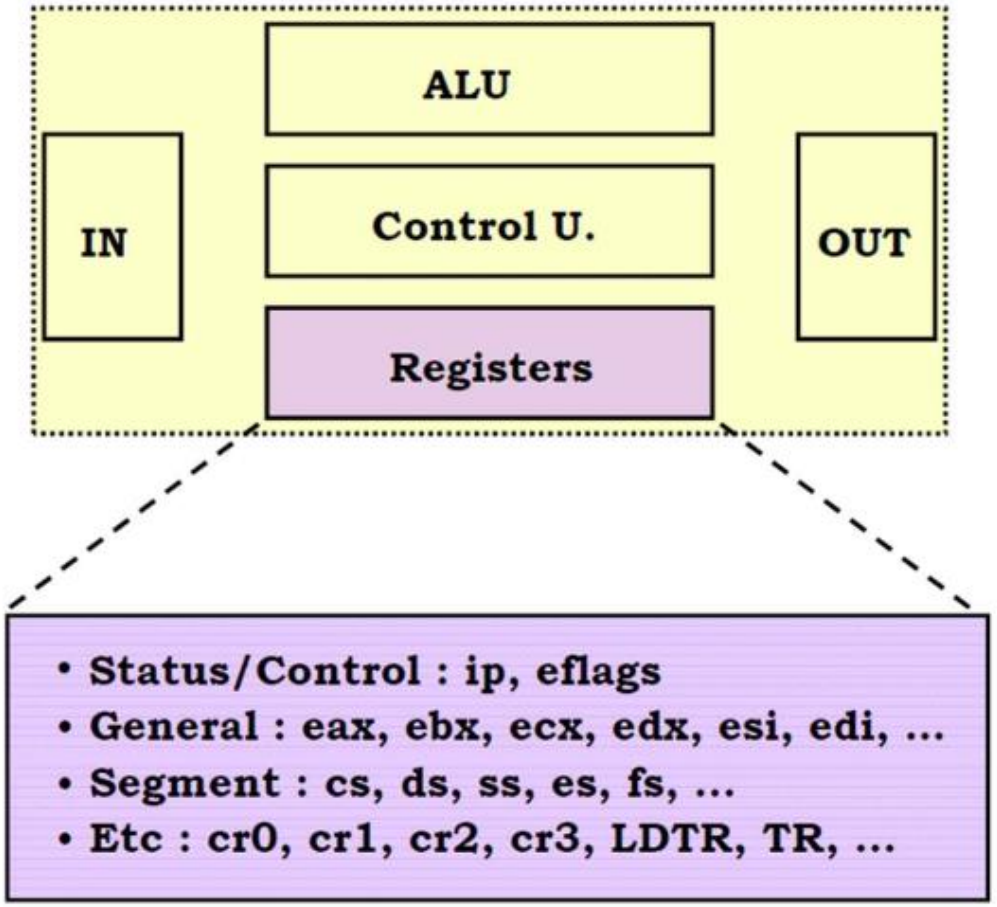
Process switch

- 프로세스 실행을 제어하기 위해 커널은 CPU에서 실행 중인 프로세스의 실행을 멈추고 이전에 멈춘 다른 프로세스의 진행을 재개할 수 있어야 한다

- 프로세스의 실행을 재개하기 전에 레지스터로 다시 복구해야 하는 데이터 집합을 하드웨어 컨텍스트 (Hardware Context)라고 한다

Process switch

- Hardware context

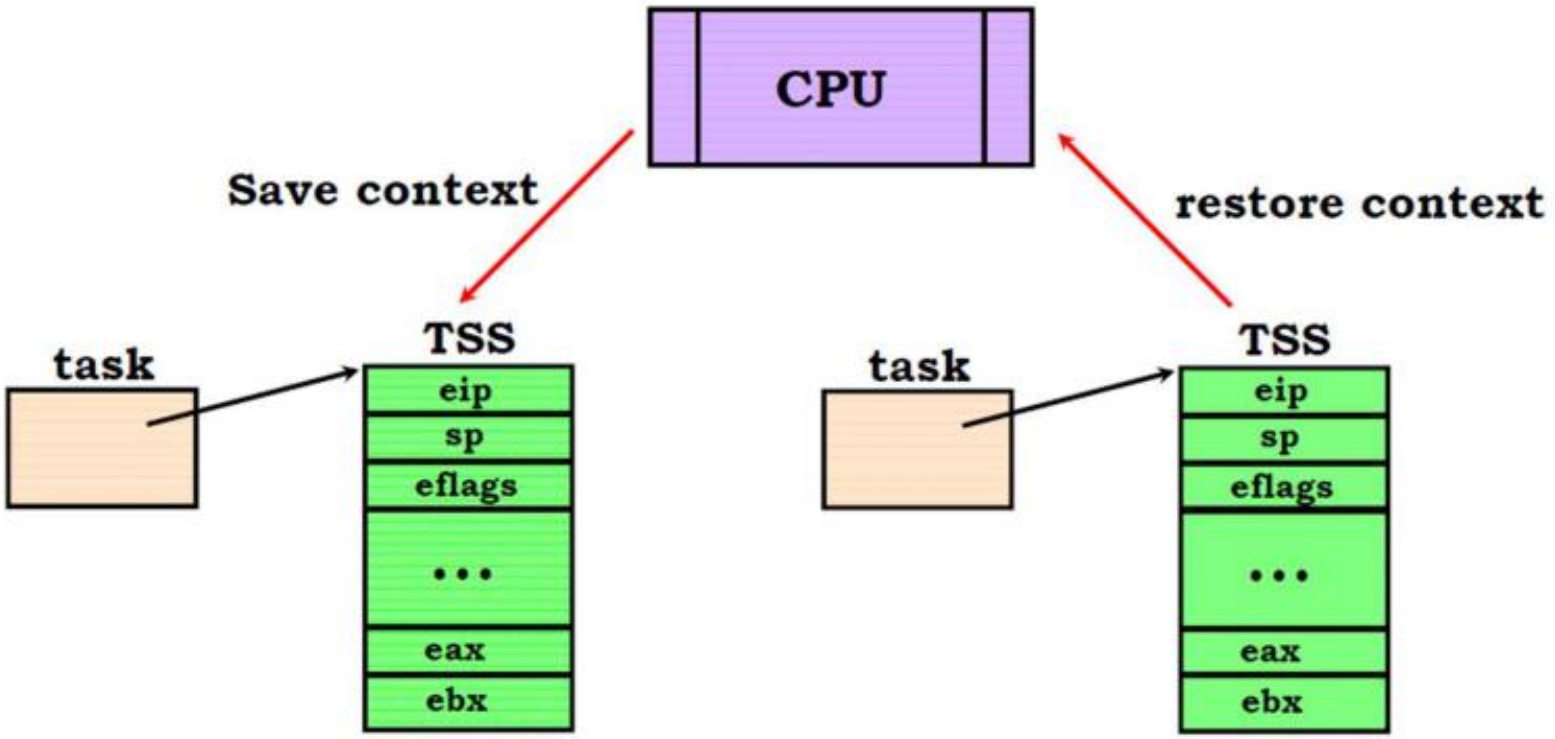


Linux One © 2002

Process switch

- 프로세스의 전환은 매우 자주 일어나므로 하드웨어 컨텍스트를 저장하고 복구하는데 걸리는 시간을 최소화하는 일은 매우 중요하다
- 프로세스 전환은 커널 모드에서만 일어난다
- 모든 레지스터 내용은 프로세스를 전환하기 전에 커널 모드 스택 상에 저장된다

Process switch



Linux One © 2002

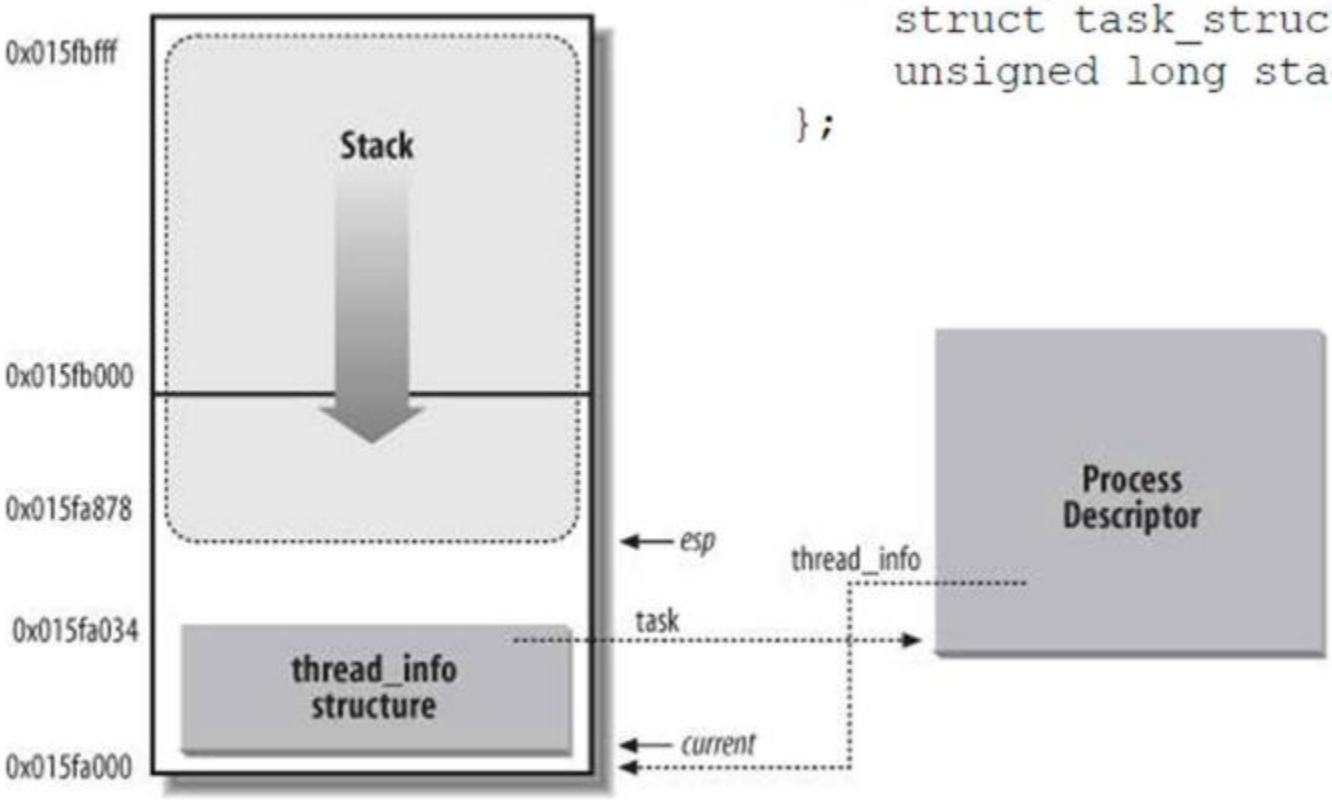
Task State Segment

- TSS (Task State Segment)
  - 인텔 80x86구조에서 하드웨어 컨텍스트를 저장하기 위한 특별한 세그먼트
  - 각 CPU마다 별개의 TSS를 가진다
    * 사용자 모드에서 커널 모드로 전환할 때, TSS에서 커널 모드 스택의 주소를 가져온다

Process switch

- 커널 모드 스택

```
union task_union {
    struct task_struct task;
    unsigned long stack[2048];
};
```



policy

Scheduling Policy

- 스케줄링 알고리즘은 몇 가지 서로 상충되는 목적을 달성해야 한다
  - 프로세스의 반응 시간은 짧아야 한다
  - 백그라운드 작업의 처리량도 좋아야 한다
  - 프로세스의 starvation을 피해야 한다
  - 우선순위가 낮은 프로세스와 높은 프로세스의 조화가 필요하다

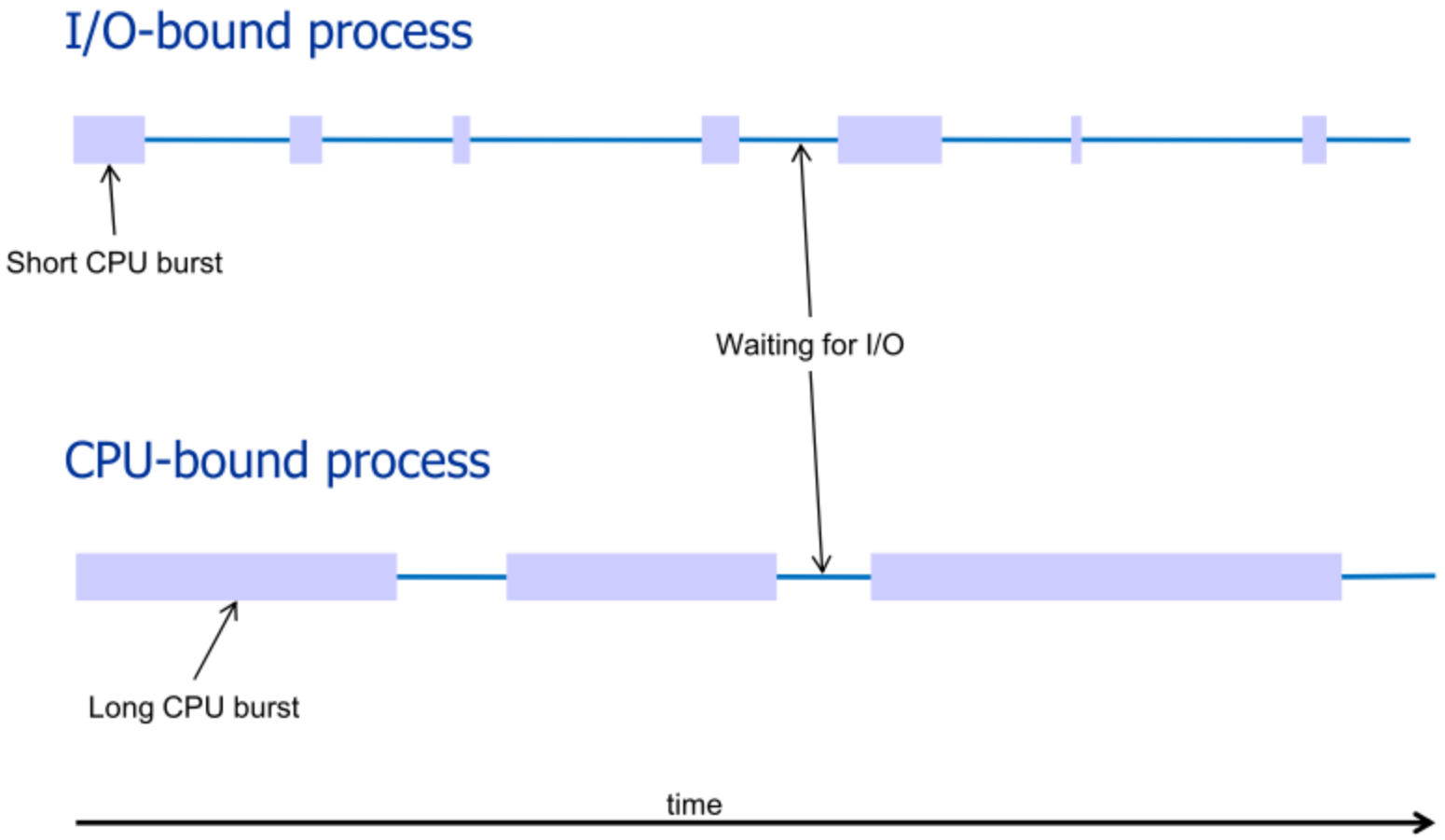
Scheduling Policy

- 리눅스의 스케줄링은 시분할 기법(time sharing technique)을 토대로 한다
  - CPU 시간을 슬라이스(slice)로 쪼개고, 실행 가능한 각 프로세스마다 슬라이스를 하나씩 할당하여 프로세스 여러개를 시간 다중화 (Time Multiplexing)방식으로 실행
  - 현재 실행하고 있는 프로세스가 종료하지 않은 채 프로세스에 부여한 타임 슬라이스, 즉 퀀텀이 만료되면 프로세스 전환이 일어날 수 있다

Scheduling Policy

- 프로세스들은 전통적으로 I/O-Bound와 CPU-Bound로 분류한다
  - I/O-Bound
    * 입출력 장치를 많이 이용하며, 많은 시간을 입출력 작업이 끝나기를 기다리는 데 사용
  - CPU-Bound
    * CPU 시간이 많이 필요한 숫자 계산 프로그램들에서 동작

I/O-Bound vs Processor-Bound Processes



또 다른 분류법

- 상호 작용 프로세스 (Interactive Processes)
  - 끊임없이 사용자와 상호작용. 빠른 반응 시간이 필요. 반응 시간이 늦으면 사용자는 시스템이 정상이 아니라고 생각할 것이다. 전형적인 프로그램으로는 명령 쉘 (command shell), 문서편집기, 그래픽 프로그램
- 일괄 작업 (Batch Processes)
  - 사용자와의 상호 작용을 요구하지 않는다. 프로세스의 반응이 빠르지 않아도 되며, 전형적인 프로그램으로는 컴파일러와 데이터베이스 찾기 엔진, 과학 계산 프로그램
- 실시간 프로세스 (Real-time Processes)
  - 우선순위가 낮은 프로세스가 높은 프로세스를 블록 해서는 안되며, 짧은 반응 시간을 보장하면서 시간 편차를 최소화해야 한다. 전형적인 프로그램은 비디오와 소리 관련 프로세스, 로봇제어기, 센서로 부터 데이터 수집

Scheduling Policy

- 스케줄링 관련 시스템 콜

System call	Description
nice( )	Change the static priority of a conventional process
getpriority( )	Get the maximum static priority of a group of conventional processes
setpriority( )	Set the static priority of a group of conventional processes
sched_getscheduler( )	Get the scheduling policy of a process
sched_setscheduler( )	Set the scheduling policy and the real-time priority of a process
sched_getparam( )	Get the real-time priority of a process
sched_setparam( )	Set the real-time priority of a process
sched_yield( )	Relinquish the processor voluntarily without blocking
sched_get_ priority_min( )	Get the minimum real-time priority value for a policy
sched_get_ priority_max( )	Get the maximum real-time priority value for a policy
sched_rr_get_interval( )	Get the time quantum value for the Round Robin policy
sched_setaffinity( )	Set the CPU affinity mask of a process
sched_getaffinity( )	Get the CPU affinity mask of a process

Process Preemtion

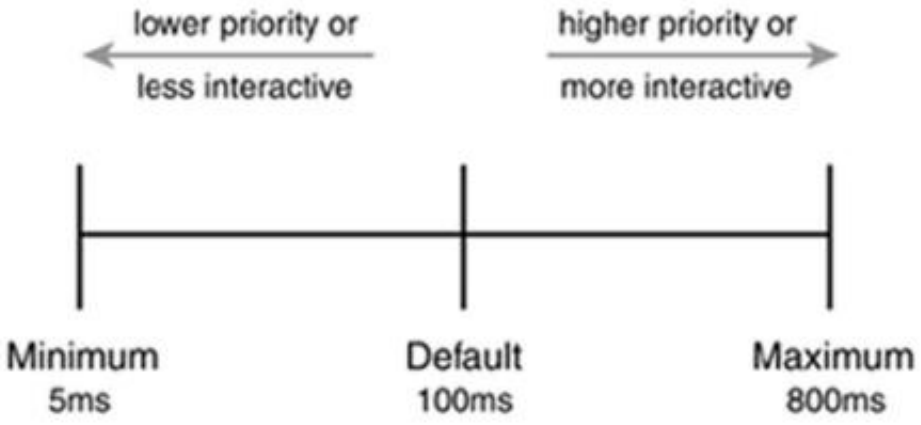
- 리눅스 프로세스는 선점형 (Preemptive)
  - 어떤 프로세스가 TASK_RUNNING 상태가 되면 커널은프로세스의 동적 우선순위가 현재 실행 중인 프로세스의 우선순위보다 높은지 검사하며, 동적 우선순위가 훨씬 높다면 current의 실행을 중단하고 스케줄러를 호출한다
  - 타임 퀀텀을 모두 사용하면, TIF_NEED_RESCHED 플래그를 설정하며, 타이머 인터럽트 핸들러가 끝날 때 스케줄러를 호출

Timeslice

- 타임 퀀텀
  - 시스템 성능에 중요한 영향을 미치며, 너무 길어서도 안되고 너무 짧아서도 안 된다
  - 너무 길면 반응성이 떨어지고, 너무 짧으면 프로세스 전환하는 데만 할당 시간의 대부분을 사용한다
- 리눅스가 채택한 규칙은 시스템의 반응성을 좋게 유지하면서 퀀텀 기간을 가능한 길게 하는 것

Timeslice

Type of Task	Nice Value	Timeslice Duration
Initially created	parent's	half of parent's
Minimum Priority	+19	5ms (MIN_TIMESLICE)
Default Priority	0	100ms (DEF_TIMESLICE)
Maximum Priority	20	800ms (MAX_TIMESLICE)



Process priority

- 전통적인 프로세스 스케줄링
  - 모든 전통적인 프로세스는 자신만의 ‘정적인 우선순위 (static priority)’를 가진다
    * 범위 100(최고)~139(최저)

- 기본 타임 퀀텀

$$\begin{matrix} base\ time\ quantum \\ (\text{in milliseconds}) \end{matrix} = \begin{cases} (140 - static\ priority) \times 20 & \text{if } static\ priority < 120 \\ (140 - static\ priority) \times 5 & \text{if } static\ priority \geq 120 \end{cases}$$

Process priority

- 전통적인 프로세스의 전형적인 우선순위 값

Description	Static priority	Nice value	Base time quantum	Interactivedelta	Sleep time threshold
Highest static priority	100	-20	800 ms	-3	299 ms
High static priority	110	-10	600 ms	-1	499 ms
Default static priority	120	0	100 ms	+2	799 ms
Low static priority	130	+10	50 ms	+4	999 ms
Lowest static priority	139	+19	5 ms	+6	1199 ms

Process priority

- 동적 우선순위와 평균 수면 시간
  - 실행할 새로운 프로세스를 선택할 때 스케줄러가 실제로 사용하는 값
    * Dynamic priority = max(100, min(static_priority – bonus + 5, 139))
    * Bonus : 0 ~ 10, 프로세스의 평균 수면 시간과 연관

      < 5 : penalty

      > 5 : premium

Process priority

- 평균 수면 시간, 보너스, time slice granularity

Average sleep time	Bonus	Granularity
Greater than or equal to 0 but smaller than 100 ms	0	5120
Greater than or equal to 100 ms but smaller than 200 ms	1	2560
Greater than or equal to 200 ms but smaller than 300 ms	2	1280
Greater than or equal to 300 ms but smaller than 400 ms	3	640
Greater than or equal to 400 ms but smaller than 500 ms	4	320
Greater than or equal to 500 ms but smaller than 600 ms	5	160
Greater than or equal to 600 ms but smaller than 700 ms	6	80
Greater than or equal to 700 ms but smaller than 800 ms	7	40
Greater than or equal to 800 ms but smaller than 900 ms	8	20
Greater than or equal to 900 ms but smaller than 1000 ms	9	10
1 second	10	10

Process priority

- 평균 수면 시간은 특정 프로세스가 상호 작용하는지, 또는 일괄 처리하는지를 구분하기 위해 사용
  - 다음 공식을 만족시키면 '상호 작용하는 것'으로 여겨진다

$$\text{dynamic priority} \leq 3 \times \text{static priority} / 4 + 28$$

$$\text{bonus} - 5 \geq \text{static priority} / 4 - 28$$

  - Static priority / 4 - 28은 상호 작용 델타라고 불린다

Active and expired processes

- 활성 프로세스와 만료된 프로세스
  - 활성 프로세스 (Active Processes)
    * 실행 가능한 프로세스들은 아직 자신의 타임 퀀텀을 다 쓰지 않았으며, 따라서 실행될 수 있다
  - 만료된 프로세스 (Expired Processes)
    * 실행 가능한 프로세스들은 자신의 타임 퀀텀을 모두 소비하여서 활동적인 프로세스가 만료될 때까지 실행이 금지된다

Runqueue

- Runqueues
  - 스케줄러의 기본 자료구조
  - kernel/sched.c에 struct runqueue로 정의
  - 한 프로세서의 실행 가능한 프로세스의 목록
  - 한 프로세서당 하나의 runqueue가 존재
  - 실행 가능한 프로세스는 오직 하나의 실행큐에만 포함되어야 함
  - 여러 태스크가 실행큐를 동시에 조작하는 것을 방지하기 위해 스핀락이 사용됨
    * 스핀락 : 임계구역 (critical section)에 진입이 불가능할 때 진입이 가능할 때까지 루프를 돌면서 재시도하는 방식의 락

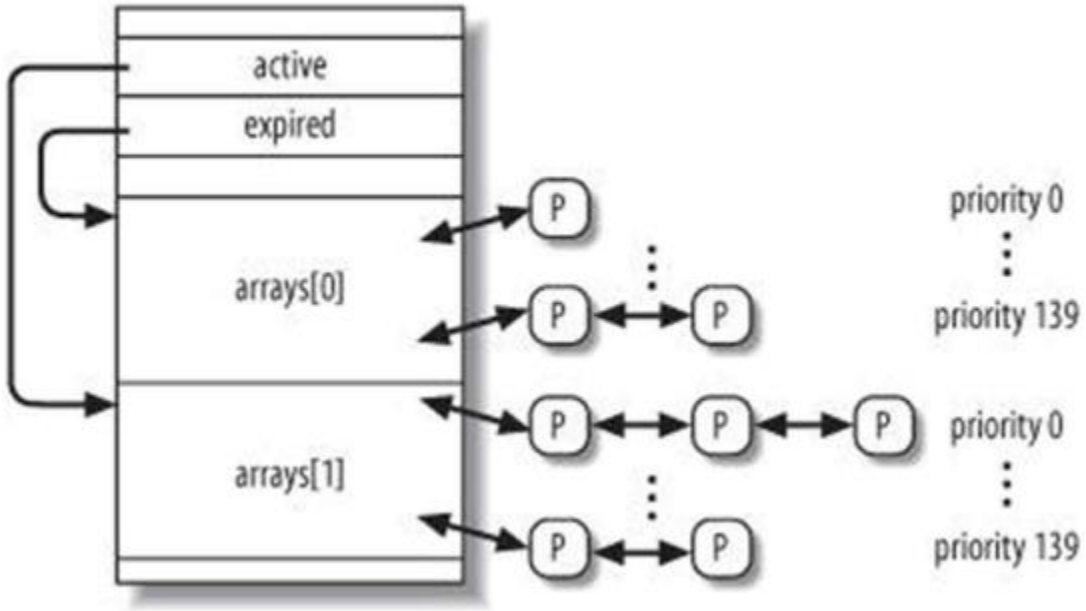
Runqueue

- Runqueue structure

Type	Name	Description
spinlock_t	lock	Spin lock protecting the lists of processes
unsigned long	nr_running	Number of runnable processes in the runqueue lists
unsigned long	cpu_load	CPU load factor based on the average number of processes in the runqueue
unsigned long	nr_switches	Number of process switches performed by the CPU
unsigned long	nr_uninterruptible	Number of processes that were previously in the runqueue lists and are now sleeping in TASK_UNINTERRUPTIBLE state (only fields across all runqueues is meaningful)
unsigned long	expired_timestamp	Insertion time of the eldest process in the expired lists
unsigned long long	timestamp_last_tick	Timestamp value of the last timer interrupt
task_t *	curr	Process descriptor pointer of the currently running process (same as current for the local CPU)
task_t *	idle	Process descriptor pointer of the *swapper* process for this CPU
struct mm_struct *	prev_mm	Used during a process switch to store the address of the memory descriptor of the process being replaced
prio_array_t *	active	Pointer to the lists of active processes
prio_array_t *	expired	Pointer to the lists of expired processes
prio_array_t [2]	arrays	The two sets of active and expired processes
int	best_expired_prio	The best static priority (lowest value) among the expired processes
atomic_t	nr_iowait	Number of processes that were previously in the runqueue lists and are now waiting for a disk I/O operation to complete
struct sched_domain *	sd	Points to the base scheduling domain of this CPU (see the section "Scheduling Domains" later in this chapter)
int	active_balance	Flag set if some process shall be *migrated* from this runqueue to another (runqueue balancing)
int	push_cpu	Not used
task_t *	migration_thread	Process descriptor pointer of the *migration* kernel thread
struct list_head	migration_queue	List of processes to be removed from the runqueue

Runqueue

- 실행 큐 자료 구조와 실행 가능한 프로세스의 집합



Process scheduler

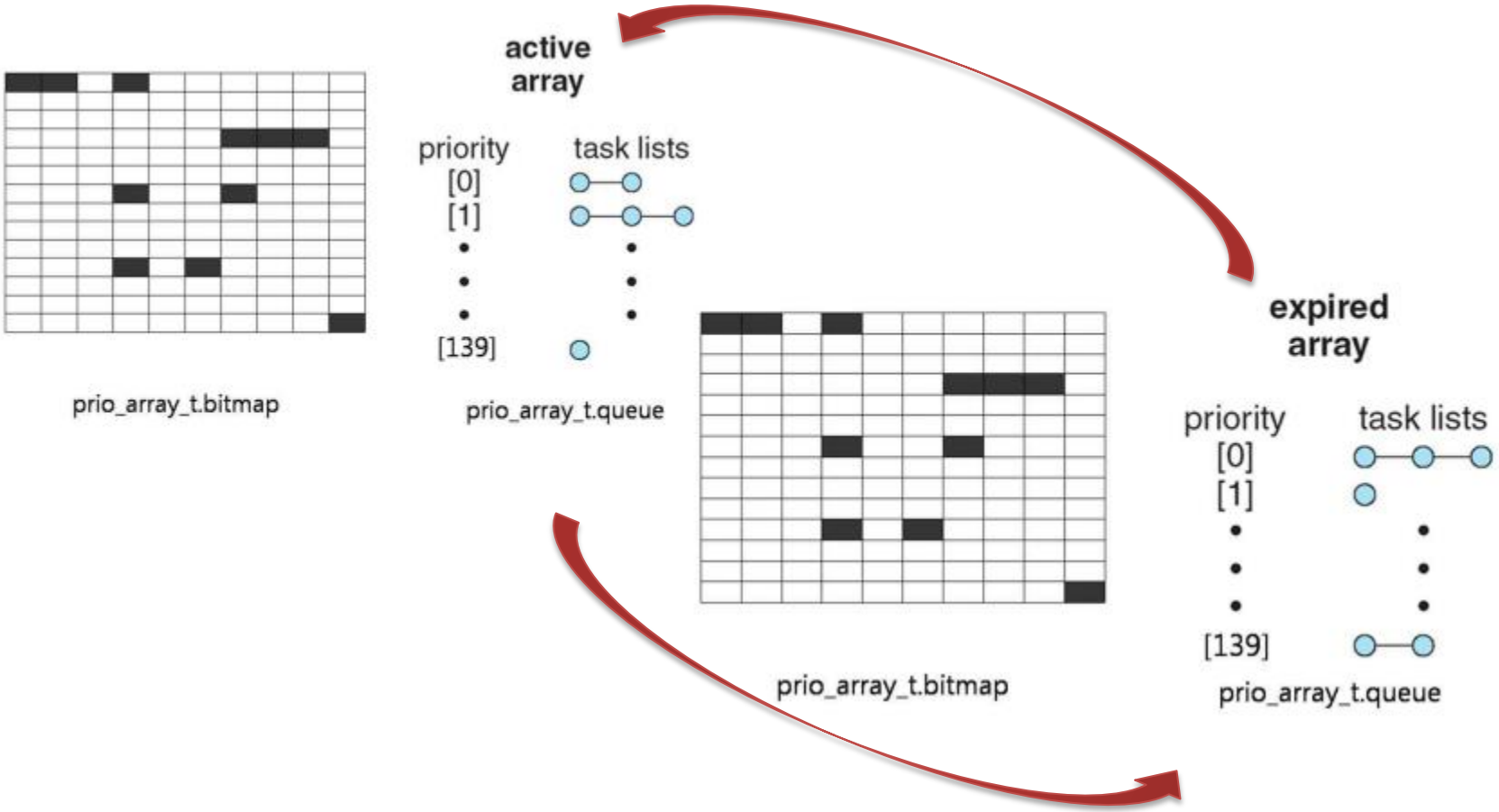
O(1) scheduler

- Kernel 2.5 introduced O(1) scheduler
  - 각각 프로세스는 priority에 따른 time_quantum을 갖는다.
  - Two priority arrays
    * Active array
      runqueue에 있는 타임쿼텀이 남아있는 테스크들
    * Expired array
      runqueue에 있는 타임쿼텀을 다 소비한 테스크들
  - Priority bitmap
    * 실행 가능한 highest-priority 테스크를 효율적으로 찾기위해 사용

O(1) scheduler

- active array로부터 실행 시킬 프로세스를 선택
- 타임 쿼텀이 다 소비하면 expired array로 보내짐
- active array에 프로세스가 없으면 expired array와 swap을 한다.

O(1) scheduler



Process descriptor

- 프로세스 디스크립터 (스케줄링과 연관된 필드)

Type	Name	Description
unsigned long	thread_info->flags	Stores the TIF_NEED_RESCHED flag, which is set if the scheduler must be invoked (see the section "Returning Chapter 4)
unsigned int	thread_info->cpu	Logical number of the CPU owning the runqueue to which the runnable process belongs
unsigned long	state	The current state of the process (see the section "Process State" in Chapter 3)
int	prio	Dynamic priority of the process
int	static_prio	Static priority of the process
struct list_head	run_list	Pointers to the next and previous elements in the runqueue list to which the process belongs
prio_array_t *	array	Pointer to the runqueue's prio_array_t set that includes the process
unsigned long	sleep_avg	Average sleep time of the process
unsigned long long	timestamp	Time of last insertion of the process in the runqueue, or time of last process switch involving the process
unsigned long long	last_ran	Time of last process switch that replaced the process
int	activated	Condition code used when the process is awakened
unsigned long	policy	The scheduling class of the process (SCHED_NORMAL, SCHED_RR, or SCHED_FIFO)
cpumask_t	cpus_allowed	Bit mask of the CPUs that can execute the process
unsigned int	time_slice	Ticks left in the time quantum of the process
unsigned int	first_time_slice	Flag set to 1 if the process never exhausted its time quantum
unsigned long	rt_priority	Real-time priority of the process

O(1) scheduler

- O(1)의 문제점
  - 우선순위가 높은 프로세스(120)는 100ms 마다, 우선 순위가 낮은 프로세스(138)는 10ms 마다 스위칭이 일어난다.
  - 우선순위 120과 121은 100ms와 95ms이기 때문에 차이가 크지 않지만 , 우선 순위가 139와 138은 5ms, 10ms이므로 2배의 차이가 난다.
  - Interactive task를 우선시하는 문제가 있다. 프로세스가 sleep 상태에서 깨어날 때 즉시 실행되게 하므로 expired 큐에 존재하는 프로세스의 실행이 지연된다.

CFS (Completely Fair Scheduler)

- linux 2.6.23 이후, CFS(Completely Fair Scheduler)가 기본 스케줄러로 채택

- CFS의 기본 개념은 프로세스에게 프로세서 시간을 제공할 때 밸런스(공평성)를 유지하는 것

  즉, 프로세스에 공평한 양의 프로세서가 제공되어야 한다.

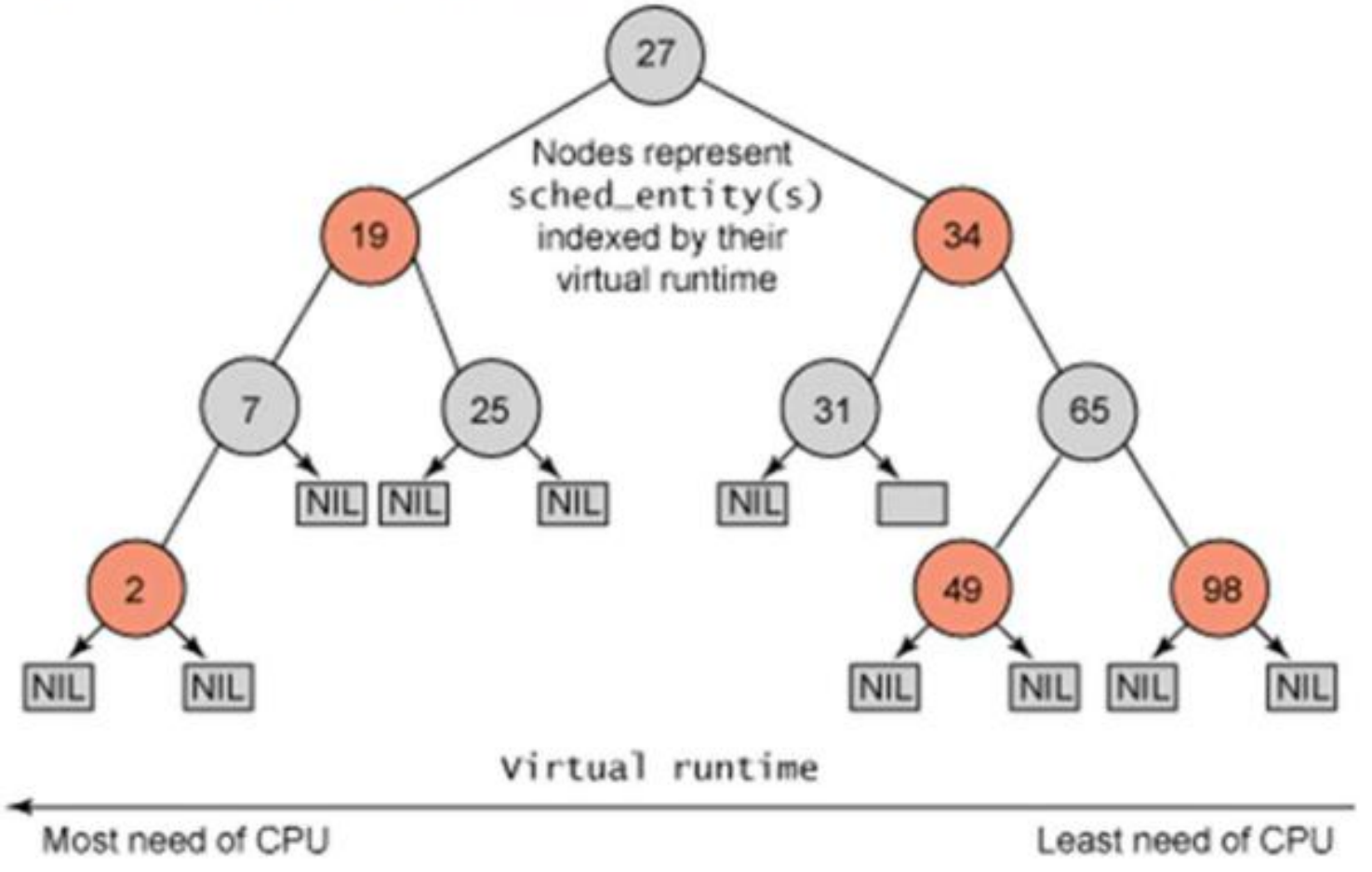
- CFS에서는 밸런스를 결정하기 위해 **가상 런타임**이라는 normalized 또는 weighted된 태스크의 실제 실행된 시간 사용

- CFS는 실행큐 배열을 제거, 모든 프로세스의 실행 상태를 관리하기 위해 Red-Black tree 구조를 사용
  - RB-tree 스스로 밸런스를 조절
  - 트리에 대한 작업이 O(log n)시간 빠르고 효율적으로 삽입, 삭제 가능

CFS (Completely Fair Scheduler)

- 프로세서의 대한 요구가 높은 (가상 런타임이 낮은) 작업부터 차례대로 트리의 왼쪽에 저장

Figure 1. Example of a red-black tree



CFS (Completely Fair Scheduler)

- CFS는 nice값을 태스크가 할당 받을 프로세서의 비율에 대한 weight로 사용 (nice ↓ => weight ↑)

  nice 값 : 프로세스의 실행우선순위를 설정하기 위한 값 (-20 ~ +19)
  낮을 수록 높은 우선순위

- 실행 가능한 태스크들의 weight의 총합을 기준으로 자신의 weight만큼만 비례적으로 timeslice를 할당 받음

$$slice = \frac{task\_weight}{total\_weight} \times period \qquad vruntime = \frac{nice\_0\_load}{task\_weight} \times runtime$$

  - Period는 스케줄러가 모든 태스크들을 수행하는데 걸리는 시간 (default 20ms)
  - nice_0_load : 상수값 1024

CFS (Completely Fair Scheduler)

- Example
  - task_weight 비율이 3:1인 두개의 태스크가 실행
  - 20ms 주기로 컨텍스트 스위칭이 수행된다고 가정
  - 60ms 후 이상적인 경우 45ms, 15ms 씩 수행
  - 실제로 40ms, 20ms씩 수행
  - 다음에는 어느 태스크가 스케줄링 되어야 할까?

  - virtual runtime이 작은 태스크를 수행
  - 실행 시간에 scale factor(nice_0_load/task_weight)를 곱해서 virtual runtime 생성
  - 40ms * 1 < 20ms * 3
  - virtual runtime이 40ms인 태스크가 다음에 스케줄링

CFS (Completely Fair Scheduler)

- Example

Process Number	Nice	Weight	Task Weight/ Total Weight	W0 (nice=0) / Task Weight	Time Slice	vruntime (ms)
1	-10	9548	0.675	0.107	6.75	0.72
2	-5	3121	0.221	0.328	2.21	0.72
3	0	1024	0.072	1	0.72	0.72
4	5	335	0.024	3.057	0.24	0.73
5	10	110	0.008	9.309	0.08	0.74
Total		14138	1.00		10.00	

Homework #2

- 주제 : Process Scheduling Simulator
- 제출물 :
  - 보고서(프로그램 소스, 주석, 실행 결과)
  - 프로그램 소스

- Due date : 2014.05.02 24:00까지
- 강의홈페이지 제출

Homework #2

- Process Scheduling Simulator
  - FCFS Scheduling algorithms
  - SJF Scheduling (Non-Preemptive)
  - Priority Scheduling
  - Round-Robin Scheduling

Homework #2

- Process Scheduling Simulator
  - Millisecond 단위 / context switch 시간은 무시
  - 프로세스 4개 기준
  - 각 스케줄링 알고리즘에 대한 waiting 시간 출력 및 Gantt Chart 출력

```
Gantt Chart
 0        5         10        15        20        25        30
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 3 3 3

Process 1 waiting time = 0 sec
Process 2 waiting time = 24 sec
Process 3 waiting time = 27 sec
Average waiting time = 17 sec
```

Homework #2

- Process Scheduling Simulator
  - 입력 형식 (data file)

Homework #2

- Process Scheduling Simulator
  - struct process

  - 실행 형식
    * ./filename [data filename] [scheduling policy]

```
Gantt Chart
 0        5         10        15        20        25        30
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 3 3 3

Process 1 waiting time = 0 sec
Process 2 waiting time = 24 sec
Process 3 waiting time = 27 sec
Average waiting time = 17 sec
```