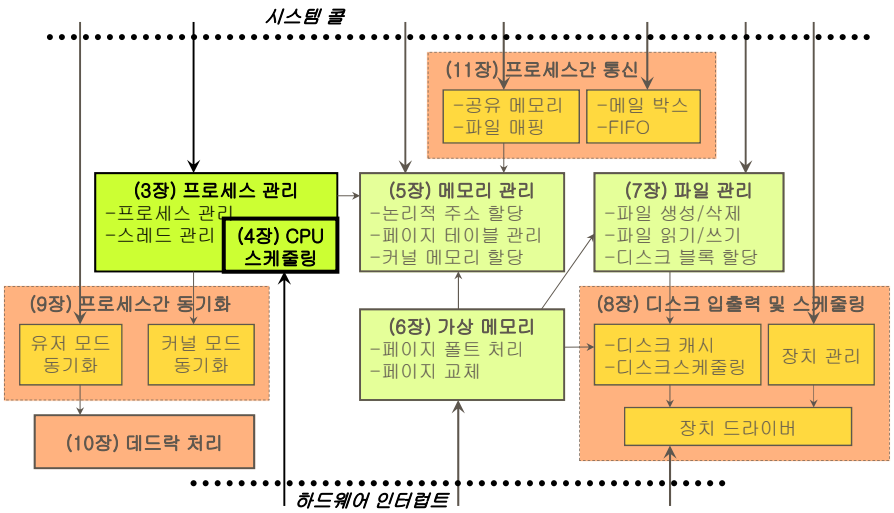
CPU 스케줄링

- ✓프로세스 큐와 큐잉 다이어그램
- ✓단기 스케줄러와 디스패처
- ✓스와핑과 중기 스케줄러
- ✓잡큐와 장기 스케줄러
- ✓선점형 스케줄링과 선점형 커널
- ✓스케줄링의 평가기준
- ✓스케줄링 정책들의 비교
- ✓스케줄링 알고리즘 예
- ✓picoKernel 스케줄링의 구현

관련 운영체제 구성 모듈



이해하고 넘어가야 할 내용들

- ✓ 스케줄링의 의미와 스케줄러의 구현 방법
- ✓ 단기 스케줄러 (CPU 스케줄러), 중기 스케줄러 (스와퍼/페이저) 및 장기 스케줄러의 역할
- ✓ 선점형 스케줄링 및 선점형 커널
- ✓ 스케줄링 결과의 여러 가지 평가 기준
- ✓ 여러 가지 스케줄링 정책들에 대한 이해와 이들의 비교 평가
- ✓ Linux 2.4/2.6, 4.4BSD, Windows, System V 등에서 사용하는 스케줄링 알고리즘들의 비교

CPU의 활용률 향상과 스케줄링

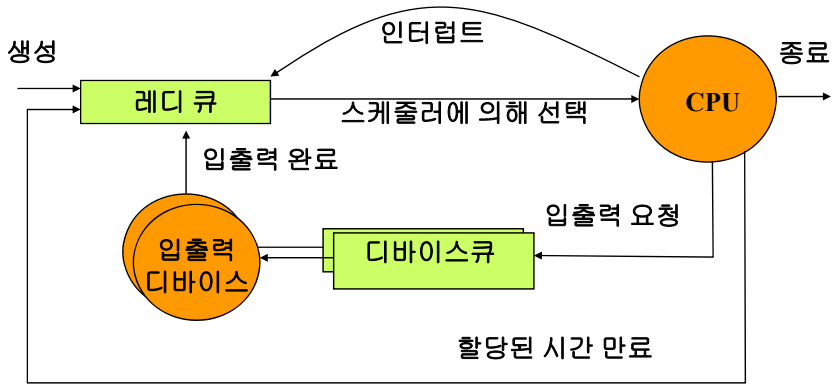
- ✓ **프로세스별 특성**
  - ✓ 하나의 프로세스는 계산작업 단위 (CPU burst)와 입출력작업 단위 (I/O burst)의 반복으로 구성
  - ✓ 계산위주의 프로세스와 입출력 위주의 프로세스가 혼재
- ✓ **CPU 활용률 (utilization) 향상**
  - ✓ 일부 프로세스가 입출력 대기 동안 실행가능 프로세스는 계산 작업 진행
  - ✓ ➔ 전체적으로 CPU 활용률 향상
- ✓ **스케줄링 평가기준**
  - ✓ CPU 활용률 외에도 프로세스의 대기시간, 공평한 실행기회 등 목적에 적절한 기준 적용
  - ✓ 스케줄링 정책은 적용하는 평가기준에 유리한 것 적용

프로세스 큐

- ✓ **프로세스 큐 (queue)**
  - ✓ 프로세스들의 집합을 관리하는 단위
  - ✓ 큐는 FCFS (First-Come First-Served)의 의미이나 실제로는 List, Heap 등의 효율적인 자료 구조 적용
- ✓ **레디 큐**
  - ✓ CPU에 의해 실행되기를 기다리는 프로세스들의 집합
  - ✓ 프로세스들의 상태는 Ready
- ✓ **디바이스 큐**
  - ✓ 디바이스별로 입출력 작업을 위해 대기중인 프로세스들의 집합
  - ✓ 프로세스들의 상태는 Waiting
  - ✓ 입출력 장치는 세마포나 뮤텍스로 매핑하여 처리
- ✓ **CPU 스케줄링**
- ✓ **디바이스 스케줄링**

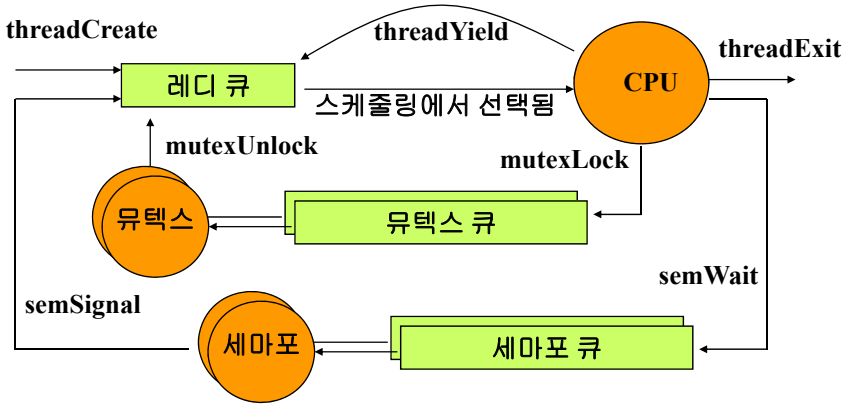
큐잉 (Queuing) 다이어그램

- ✓ **프로세스들이 여러 가지의 큐를 거쳐서 실행이 진행되는 과정 표현**
- ✓ **타이머나 디바이스로부터의 인터럽트 처리결과에 따라서 우선수위가 높은 프로세스가 선점하여 실행될 수도 있음**



picoKernel의 큐잉 다이어그램

- ✓ 각 디바이스는 뮤텍스로 매핑
- ✓ 뮤텍스 및 세마포 별로 별도의 대기큐 사용



스케줄러

- ✓ **스케줄러 (scheduler)**
  - ✓ 단기 스케줄러: CPU 스케줄러로서 수십 밀리초 이내의 짧은 시간 단위로 실행 ➔ 보통 함수 형태로 존재
  - ✓ 중기 스케줄러: 가상메모리와 연관되어 스왑인/아웃대상을 결정 (수백 밀리초 내지 수초 단위로 실행) ➔ 보통 별도의 커널 스레드로 존재
  - ✓ 장기 스케줄러: 현대 운영체제에서는 사용하지 않음
- ✓ **단기 스케줄러 실행 시점**
  - ✓ (1) Running 상태의 프로세스가 다른 상태로 전환
  - ✓ (2) Ready 큐에 프로세스 추가 ➔ 이것이 먼저 실행되어야 할수도
- ✓ **디스패처 (dispatcher)**
  - ✓ 단기 스케줄러에 의해 선택된 프로세스가 CPU에 의해 실제 실행되도록 처리하는 주체
  - ✓ 커널 모드에서 문맥교환 작업을 처리하는 부분이 여기에 해당

선점형 스케줄링

- ✓ **스케줄링 실행시점**
  - ① 실행중인 프로세스가 종료되어 없어질 때 ➔ 자발적
  - ② 실행중인 프로세스가 Waiting 상태로 전환될 때 ➔ 자발적
  - ③ 실행중인 프로세스의 양보할 때 ➔ 자발적
  - ④ 실행중인 프로세스가 할당 시간이 만료될 때 ➔ 강제적
  - ⑤ Waiting 상태의 프로세스가 외부장치의 작업완료 인터럽트에 의해 Ready 상태로 전환될 때 ➔ 강제적
  - ⑥ Waiting 상태의 프로세스가 현재 실행중인 프로세스에 의해 Ready 상태로 전환될 때 ➔ 강제적
  - ⑦ 자식 프로세스가 생성되어 Ready 큐에 등록될 때 ➔ 강제적
- ✓ **선점형 (preemptive) 스케줄링**
  - ✓ 모든 경우에 스케줄링
  - ✓ ← *항상 최우선 순위의 프로세스가 실행되도록 보장함*
  - ✓ → *인터럽트 처리를 정밀하게 해주어야 함 (데이터 일관성 유지)*
- ✓ **비선점형 (non-preemptive) 스케줄링**
  - ✓ 자발적으로 다른 프로세스로 전환해야 하는 경우에만 스케줄링
  - ✓ ← *구현이 쉬움*

스케줄링 작업이 필요한 시점

구 분	발생 원인	현재 프로세스	비선점형 스케줄링	선점형 스케줄링
1. 프로세스 종료	시스템콜 함수호출	계속실행 불가	O	O
2. Waiting 상태로 전환	시스템콜 함수호출	계속실행 불가	O	O
3. 실행 순서 양보	시스템콜 함수호출	자발적으로 레디큐로 이동	O	O
4. 할당시간 만료	타이머 인터럽트	강제로 레디큐로 이동	-	O
5. 외부장치에 의해 Ready 프로세스 추가	디바이스 인터럽트	강제로 레디큐로 이동	-	O
6. 시스템콜에 의해 Ready 프로세스 추가	시스템콜 함수호출	강제로 레디큐로 이동	?	O
7. 프로세스 생성	시스템콜 함수호출	강제로 레디큐로 이동	?	O

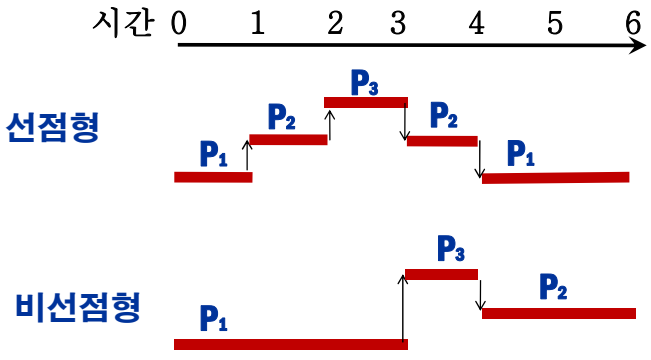
선점형 스케줄링의 구현

- ✓ **선점형 스케줄링**
  - ✓ 타이머 인터럽트에 의한 ④와 장치 하드웨어 인터럽트에 의한 ⑤의 경우에는 인터럽트 처리루틴에서 스케줄링을 실시하고 필요에 따라 문맥교환
  - ➔ 모든 레지스터 저장 필요
  - ➔ 커널 실행중에 문맥교환 발생시 커널 데이터의 일관성이 깨질 수 있으므로 이에 대한 대책 필요
  - ✓ 바람직하지만 구현에 어려움이 있음
- ✓ **구현하기에 단순한 스케줄링**
  - ✓ 실행중인 프로세스가 요청하는 시스템 콜 처리 중에만 스케줄링 (문맥 교환시 일부 레지스터만 저장)
  - ✓ 외부 인터럽트에 의한 스케줄링 제외 (비선점형)
- ✓ **비선점형 커널 (non-preemptive kernel)**
  - ✓ 스케줄링은 유저모드였으면 필요한 모든 경우에 하지만 커널모드였으면 하지 않음 (문맥 교환시 모든 레지스터 저장필요)
  - ← 비교적 쉬운 커널 구현방안

선점형 / 비선점형 스케줄링

Ready 시점: 0:P_1, 1:P_2, 2:P_3

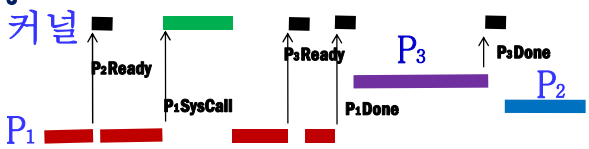
우선순위: **$P_1 < P_2 < P_3$**



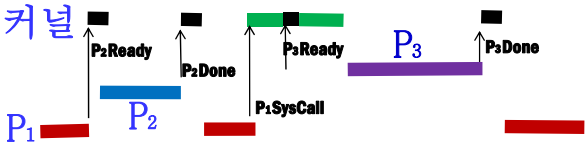
선점형 / 비선점형 커널

우선순위: $P_1 < P_2 < P_3$

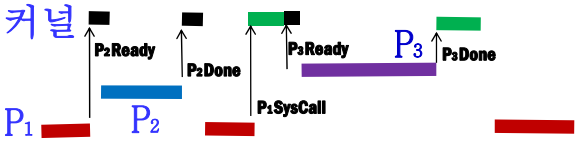
비선점형



선점형
(비선점형 커널)



선점형
(선점형 커널)



스케줄링의 평가 기준

- ✓ **대표적인 평가기준**
  - ✓ CPU 활용률 (utilization)
  - ✓ 응답시간 (response time)
  - ✓ 턴어라운드 시간 (turnaround time)
  - ✓ 대기시간 (waiting time)
  - ✓ 처리율 (throughput)
- ✓ **평가기준의 적용**
  - ✓ 대표 값: 평균값, 최대값, 최소값 등
  - ✓ 목적에 따라 적절한 기준 적용
    - ✓ 대화형 프로세스들 : 짧은 응답시간
    - ✓ 일괄처리 (후면) 프로세스들: CPU 활용률
    - ✓ ...
  - ✓ 가아문제 (starvation)의 발생 여부 검토

스케줄링 정책

- ✓ **스케줄링 정책**
  - ✓ Ready 상태의 프로세스들에 대해 실행 순서를 결정하는 정책
  - ✓ 정책들은 적용하는 평가기준에 따라 유불리에 차이
- ✓ **대표적인 스케줄링 정책들**
  - ✓ FCFS (First-Come First-Served)
  - ✓ SJF (Shortest Job First) 또는 SPN (Shortest Process Next)
  - ✓ SRTF (Shortest Remaining Time First)
  - ✓ 라운드로빈 (Round-Robin 또는 time sliced)
  - ✓ 우선순위 (priority)
  - ✓ Rate Monotonic (실시간 프로세스)
  - ✓ Earliest Deadline First (실시간 프로세스)
  - ✓ *멀티레벨 큐 및 멀티레벨 피드백 큐*
- ✓ **보통 우선순위+라운드로빈+멀티레벨피드백큐**

멀티레벨 피드백 큐

- ✓ **멀티레벨 큐**
  - ✓ 프로세스들의 특성에 따라 여러 레벨로 분리
  - ✓ 높은 레벨의 프로세스들을 우선 선택
- ✓ **멀티레벨 피드백큐**
  - ✓ 상황에 따라서 프로세스들의 레벨을 조정
- ✓ **레벨별로 적절한 스케줄링 정책 적용 가능**
  - ✓ 대화형 프로세스: 레벨을 높게 하고 라운드로빈 적용
  - ✓ 일괄처리(후면) 프로세스: 레벨을 낮게 하고 FCFS 적용
- ✓ **리눅스의 적용 예 (커널2.6 기준)**
  - ✓ 모든 프로세스들에 할당시간을 부여하고 라운드로빈 적용
  - ✓ 높은 레벨의 프로세스들에는 상대적으로 많은 할당시간 부여
  - ✓ 입출력 대기시간이 긴 프로세스: 레벨 상향 조정
  - ✓ 계산시간이 긴 프로세스: 레벨 하향 조정

실시간 프로세스 스케줄링

- ✓ **프로세스의 실행시점이 중요하게 취급되는 특수 목적의 컴퓨터 시스템 (실시간 시스템)에 적용**
  - ✓ 예: 군사용 시스템, 의료/통신 장비, 로봇, 자동차, 공장자동화 시스템 등 각종 임베디드 시스템
- ✓ **주로 주기적으로 실행되는 프로세스들로 구성**
- ✓ **RM (Rate Monotonic) 스케줄링 정책**
  - ✓ 주기가 짧을수록 먼저 선택
  - ✓ 프로세스 별로 주기에 따라 고정 우선순위 부여
- ✓ **EDF (Earliest Deadline First) 스케줄링 정책**
  - ✓ 마감시한이 급할수록 먼저 선택

대화형 프로세스를 위한 정책

- ✓ **대화형 프로세스**
  - ✓ 요구조건: 짧은 응답시간, 기아문제 없게
  - ✓ 짧은 계산단위와 긴 입출력 대기시간의 반복으로 구성
- ✓ **SJF 및 SRTF 정책**
  - ✓ 평균적인 응답시간은 짧지만
  - ✓ 프로세스 별로 응답시간의 편차가 심하고
  - ✓ 실행시간을 미리 예측하는 것이 현실적으로 곤란함
- ✓ **라운드로빈 정책**
  - ✓ 긴 계산시간의 프로세스들에는 응답시간이 길지만
  - ✓ 짧은 계산단위와 긴 입출력 대기시간의 반복으로 구성되는 대화형 프로세스들에는 응답시간이 단축됨

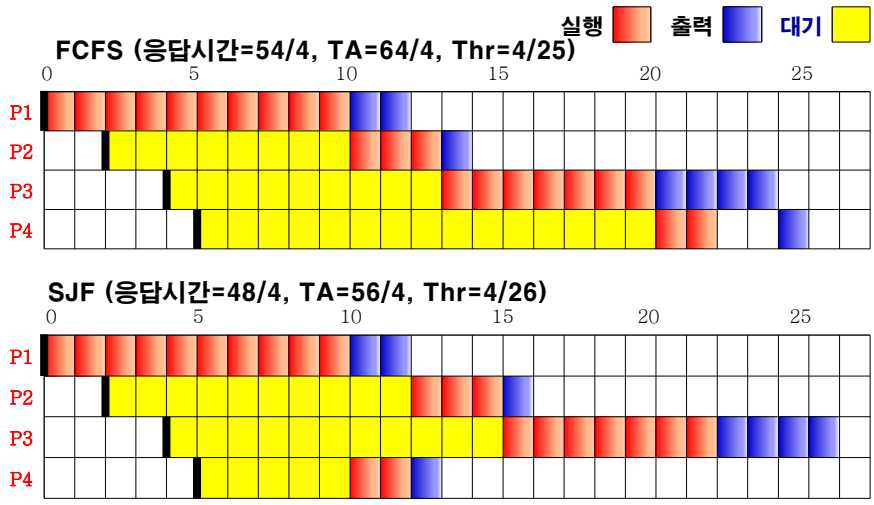
  ← 응답시간은 전부 한번 돌아가는 실행시간 이내임

스케줄링 정책들의 적용 비교

- ✓ **적용대상**
  - ✓ FCFS, SJF, SRTF, RR (slot=1), RR (slot=4), Priority
- ✓ **스케줄링 대상 프로세스 집합**

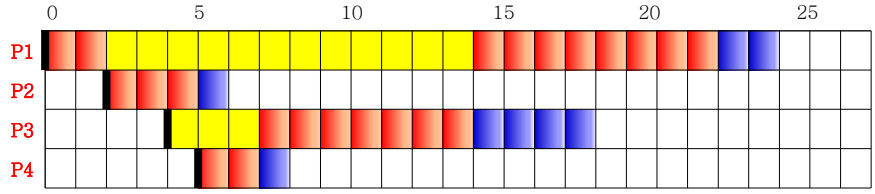
프로세스	P_1	P_2	P_3	P_4	적용대상정책
도착시간	0	2	4	5	모든 스케줄링
실행소요시간	10	3	7	2	모든 스케줄링
출력소요시간	2	1	4	1	모든 스케줄링
우선순위	1	2	3	4	우선순위 스케줄링

스케줄링 결과의 비교

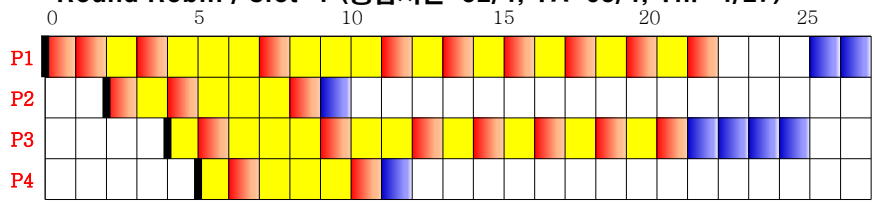


스케줄링 결과의 비교

SRTF (응답시간=37/4, TA=45/4, Thr=4/24)

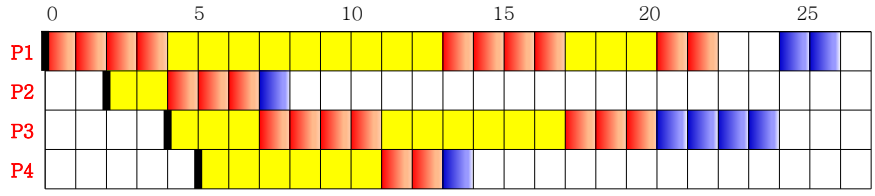


Round Robin / slot=1 (응답시간=52/4, TA=63/4, Thr=4/27)

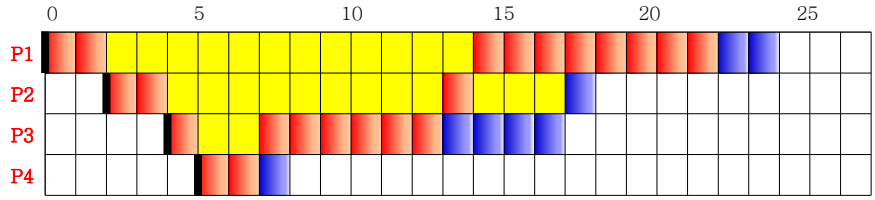


스케줄링 결과의 비교

Round Robin / slot=4 (응답시간=51/4, TA=61/4, Thr=4/26)



Priority (응답시간=45/4, TA=56/4, Thr=4/24)



스케줄링 결과 비교

스케줄링 정책	평균 응답시간	평균 턴어라운드 시간	프로세스 처리율
FCFS	13.50	16.00	0.160
SJF	12.00	14.00	0.154
SRTF	9.25	11.25	0.167
라운드로빈 (할당시간 = 1)	13.00	15.75	0.148
라운드로빈 (할당시간 = 4)	12.75	15.25	0.154
우선순위	11.25	14.00	0.167

스케줄링 정책들의 비교

항목	FCFS	SJF	SRTF	RR	우선순위	실시간 (EDF)
프로세스 선택기준	최장 대기시간	최소 실행시간	최소 남은 실행시간	다음 프로세스	최고 우선순위	최소 마감시한
선점시기	없음	없음	Ready 프로세스 추가	할당시간 만료	Raedy 프로세스 추가	Ready 프로세스 추가
평균 응답시간	☹	☺	☺	😐	☹	☹
대화형 작업의 응답시간	☹	😐	😐	☺	☹	☹
구현 난이도 및 실행 오버헤드	☺	😐 (실행시간예측?)	😐 (실행시간예측?)	☹ (잦은문맥교환)	😐	☹
기아현상	☺	☹	☹	☺	☹	😐

스케줄링 알고리즘 예

- ✓ **유닉스 계열 OS들의 우선순위 설정**
  - ✓ 생성될 때 표준 우선순위로 설정
  - ✓ 우선순위의 변경: nice / setpriority 시스템 콜 함수
  - ✓ 실행과정에서 운영체제가 약간씩 조정
- ✓ **리눅스의 스케줄링**
  - ✓ 커널 버전별로 차이가 많음
  - ✓ 커널 2.4: 라운드로빈 스케줄링을 적용하고 우선순위가 높을수록 긴 할당시간을 부여
  - ✓ 커널 2.6: 멀티레벨피드백큐 정책을 적용한 구현

  ← 실시간 프로세스 지원, 스케줄링 처리 시간 단축

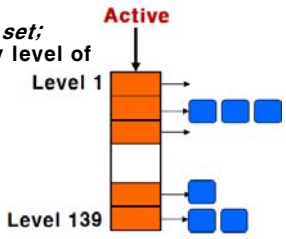
리눅스 2.4 스케줄링 알고리즘

```
linuxSchedule()
{
   // 모든 프로세스가 할당시간 (quantum)을 다 소모하면 다시 할당함
   if (quantum of every ready process is 0) {
      for (each process proc) {
         if (proc->state != STATE_WAITING)
            proc->quantum = proc->priority;
         else
            proc->quantum = proc->priority + proc->quantum / 2;
      }
   }
   // 프로세스 선택은 priority와 quantum이 클수록 먼저 선택
   selected = the first ready process;
   for (each ready process proc with positive quantum) {
      if (proc->priority + proc->quantum
            > selected->priority + selected->quantum)
         selected = proc;
   }
   // 선택된 프로세스로 문맥교환
   if (selected != RunningProcess) {
      RunningProcess = selected;
      context switch to selected;
   }
}
```

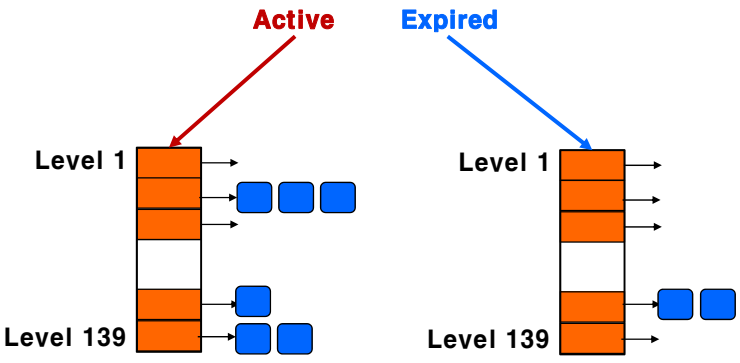
```
timerIsr()          // 타이머 인터럽트 서비스 루틴
{
   RunningProcess->quantum--;
   if (RunningProcess->quantum <= 0)
      reschedFlag = 1;
}

returnToUserMode()
// 커널모드에서 유저모드로 복귀할 때 호출됨
{
   if (reschedFlag != 0) {
      reachedFlag= 0;
      linuxSchedule();
   }
}
```

리눅스 2.6 스케줄링 알고리즘

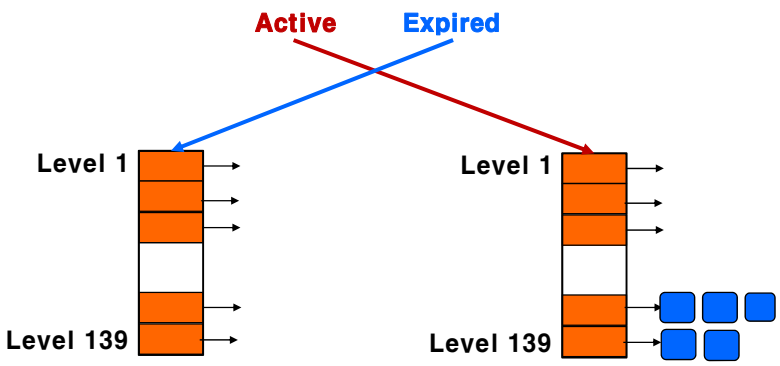
```
linuxSchedule_2_6()
{
   if (active process set is empty)
      exchange active process set and expired process set;
   selected = the first process of the lowest non-empty level of
            active process set;
   if (selected != RunningProcess) {
      RunningProcess = selected;
      context switch to selected;
   }
}
timerIsr_2_6()
{
   RunningProcess->quantum--;
   if (RunningProcess->quantum <= 0) {
      delete RunningProcess from active process set;
      level = calculate dynamic priority;
      add RunningProcess on the level of expired process set;
      RunningProcess->quantum =
         time quantum based on static priority;
      reschedFlag = 1;
   }
}
```



Linux 2.6 Active 와 Expired 교환



Linux 2.6 Active 와 Expired 교환



실시간 프로세스의 지원

- ✓ 대부분의 운영체제에서 지원: 리눅스, 윈도즈, System V 등
- ✓ 프로세스가 생성될 때 지정된 고정된 우선순위 사용 (일반 프로세스들보다 높게 설정)
  - ➔ 멀티레벨피드백 큐를 적용하는 경우 최우선 레벨들을 실시간 프로세스들에 할당
- ✓ 실시간 프로세스들 간에는 우선순위가 높은 프로세스가 먼저 선택됨
- ✓ Linux 2.6: 1 ~ 139 레벨 (낮은 번호 우선)
  - ✓ Level 1~99 : 실시간 프로세스
  - ✓ Level 100~139: 일반 프로세스
- ✓ Windows: 0 ~ 31 레벨 (높은 번호 우선)
  - ✓ Level 16~31 : 실시간 프로세스
  - ✓ Level 0~15: 일반 프로세스

picoKernel의 스케줄링 알고리즘

- ✓ 고정 우선순위 스케줄링 적용
- ✓ 우선순위는 스레드 생성시에 지정
- ✓ Ready 스레드들은 우선순위에 따라 정렬된 리스트 사용
  - ✓ Ready 리스트에 추가시 우선순위에 따른 위치를 찾아서 삽입
  - ✓ 스케줄링은 Ready 리스트의 첫 스레드 선택
- ✓ 하드웨어 인터럽트 처리부분 제외
- ✓ 적용
  - ✓ schedInsertReady(thread)
  - ✓ next = schedRunHighest()
  - ✓ schedDeleteRunning()