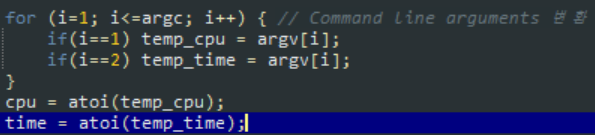
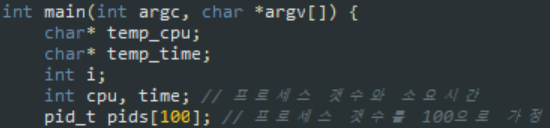
**과제#1**

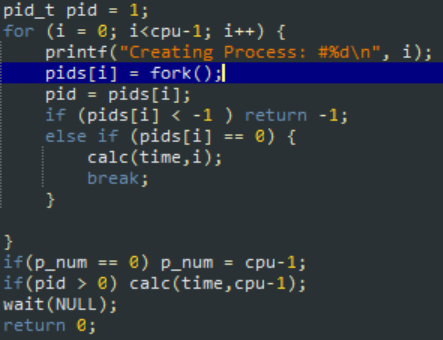
**리눅스**CPU **스케줄러분석**

**201610674 이영훈**

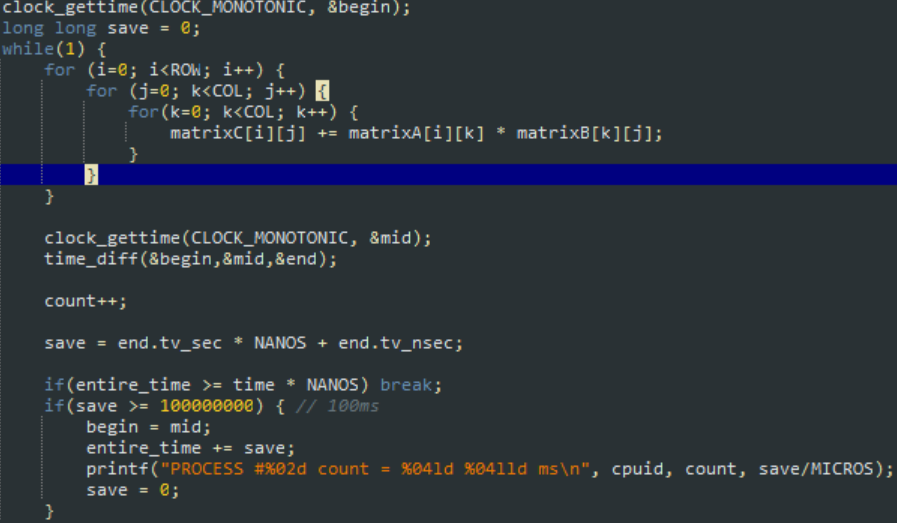
**유저 프로그램 소스 코드 설명**

**1-1**

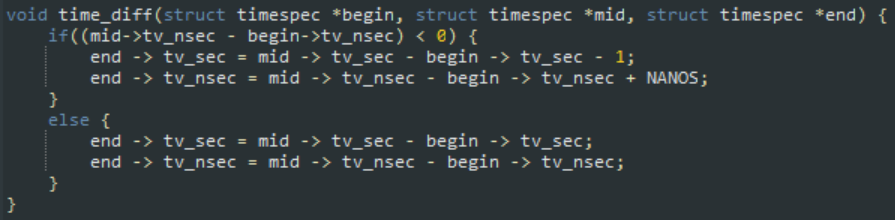


메인 함수에서 fork() 부분

프로세스 5개를 돌린다고 가정하면 부모 프로세스 1개와 자식 프로세스 4개를 만들기 위하여 pid 변수와 pids 배열을 사용하였다. pids로 for문안에서 자식을 구별하고 calc함수를 호출했으며 부모 프로세스는 fork가 끝난 뒤 calc함수를 호출하기 위하여 for문 밖에서 pid 변수로 부모 프로세스를 확인하여 calc함수를 호출하였다. 그 이후 wait으로 부모 프로세스가 먼저 끝난 경우 자식 프로세스를 기다리게 하였다.

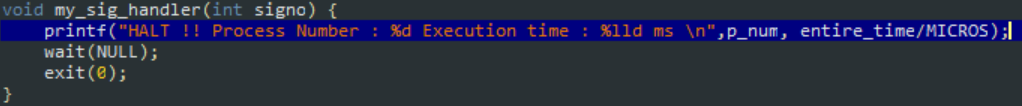
 calc()에서 시간과 연산 수 잼

While문이 계속 돌면서 실제 수행 시간을 더해가며 만약 100ms가 넘게 되면 그 시간을 기준으로 다시 시간을 처음부터 잰다. 그렇게 쌓인 시간은 전체 시간에 더해주어서 정해진 time이 넘어가면 while문 밖으로 나간다.

 monotonic 시간 구하는 함수

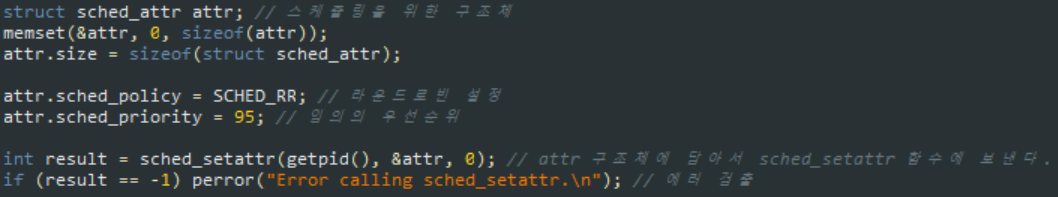
**추가 수행내용 : signal handler**

메인 함수에서 선언

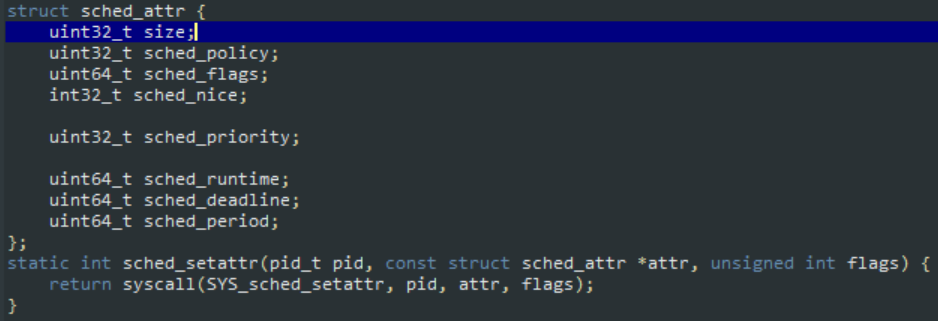


시그널 핸들러 함수를 통해 ctrl+c 한 경우 프로세스 번호와 지금까지 수행시간을 출력 후 wait하고 종료

**1-2**



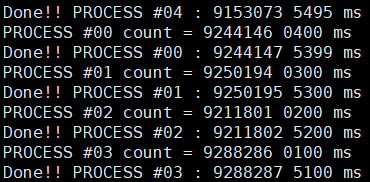
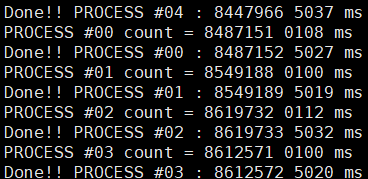
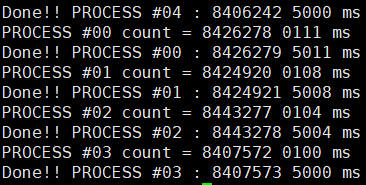
메인 함수에서 sched\_attr 구조체를 할당하여 정책과 우선순위를 정하고 sched\_setattr 함수 호출



아래 sched\_setattr함수에서 받은 매개변수와 함께 시스템콜을 호출한다.

캡쳐화면

Time Slice 1ms 10ms 100ms

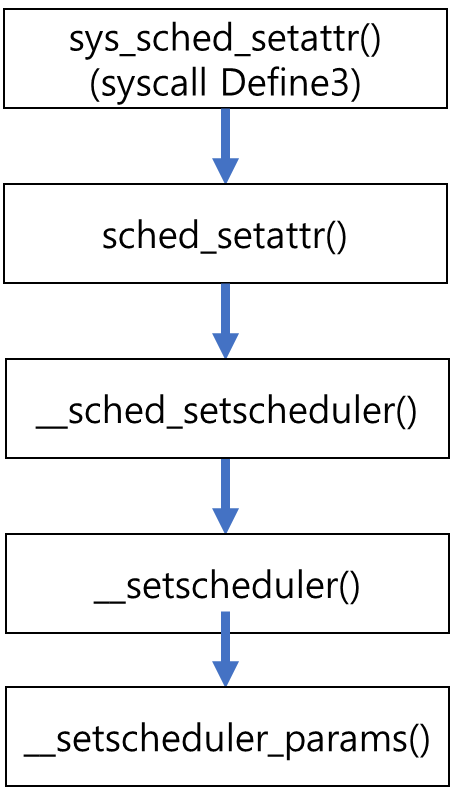


|  |  |  |  |
| --- | --- | --- | --- |
| RR Time slice | **1ms** | **10ms** | **100ms** |
| **Calculations per second**  (total calc / max time) | 1,682.783 | 1,699.487 | 1,741.809 |
| **Baseline=1ms** | 100.00% | 100.99**%** | 103.50% |
| **Baseline=10ms** | 99.01% | 100.00% | 102.49% |

Time slice가 줄어들수록 연산 횟수가 줄어가는 것을 볼 수 있다. 따라서 Time slice가 적을수록 Context Switch가 자주 일어나서 그 만큼의 오버헤드가 발생하여 연산에 영향을 준다는 것을 알 수 있었다.

**1-3**

**Sys\_sched\_setattr()함수 콜체인 그림**



1. Sched\_setattr()과 콜체인 함수들 분석 :

**(코드에 주석으로 설명 기록하여 분석함)**

 include/linux/syscall.h asmlinkage **long** sys\_sched\_setattr()

869 lines

1. asmlinkage **long** sys\_sched\_setattr(pid\_t pid, **struct** sched\_attr \_\_user \*attr, unsigned **int** flags);

프로토타입

kernel/sched/core.c SYSCALL\_DEFINE3(sched\_setattr, pid\_t pid, **struct** sched\_attr \_\_user \* uattr, unsigned **int** flags)

4578 lines

1. SYSCALL\_DEFINE3(sched\_setattr, pid\_t pid, **struct** sched\_attr \_\_user \* uattr, unsigned **int** flags)     // 시스템 호출 핸들러 매크로3
2. {
3. **struct** sched\_attr attr;
4. **struct** task\_struct \*p;
5. **int** retval;
7. **if** (!uattr || pid < 0 || flags)
8. **return** -EINVAL;  // arch/powerpc/boot/stdio.h /\* Invalid argument \*/
10. retval = sched\_copy\_attr(uattr, &attr);  //attr에 uattr 인자로 넘어온 구조체를 복사한다. 정상 수행시 0 리턴
11. **if** (retval)
12. **return** retval;
14. **if** ((**int**)attr.sched\_policy < 0)
15. **return** -EINVAL;
16. // RCU(Read Copy Update)는 읽는 작업이 주로 이루어지는 자료구조를 보호하기 위한 또 하나의 동기화 기법이다. 다른 동기화 방법과는 다르게 락이나 카운터를 사용하지 않는다.
17. rcu\_read\_lock();    //RCU에서 데이터를 참조하기 위한 Critical Section을 시작 했다는 것을 통지한다.
18. retval = -ESRCH;   /\* No such process \*/
19. p = find\_process\_by\_pid(pid); // pid를 받아옴
20. **if** (p != NULL)
21. retval = sched\_setattr(p, &attr);
22. rcu\_read\_unlock();  // RCU에서 데이터의 참조가 끝난것을 통지한다.
24. **return** retval;
25. }

kernel/sched/core.c sched\_copy\_attr() : 유저 영역의 sched\_attr를 커널 영역으로 안전하게 복사해오는 함수

4477 lines

1. **static** **int** sched\_copy\_attr(**struct** sched\_attr \_\_user \*uattr, **struct** sched\_attr \*attr)
2. {
3. u32 size;
4. **int** ret;
6. **if** (!access\_ok(VERIFY\_WRITE, uattr, SCHED\_ATTR\_SIZE\_VER0)) // 유저 프로세스 공간으로의 안전한 접근이 가능한지 확인하는 함수
7. **return** -EFAULT; // /\* Bad address \*/
9. /\* Zero the full structure, so that a short copy will be nice: \*/
10. memset(attr, 0, **sizeof**(\*attr));
12. ret = get\_user(size, &uattr->size);  용도 : 유저 영역의 데이터를 커널 영역으로 복사, 인자의 넘어온 변수의 길이만큼 복사한다.
14. **if** (ret)
15. **return** ret;
17. /\* Bail out on silly large: \*/
18. **if** (size > PAGE\_SIZE) // arch/alpha/include/asm/page.h PAGE\_SIZE 13^2로 고정
19. PAGE\_SIZE보다 큰 경우 복사 해온 데이터를 다시 유저 영역으로 넘긴다
20. **goto** err\_size;  // err\_size:
21. put\_user(**sizeof**(\*attr), &uattr->size); get\_user와 반대로 커널 영역의 데이터를 유저 영역으로 복사 인자의 넘어온 변수의 길이만큼 복사
22. **return** -E2BIG; /\* Argument list too long \*/
24. /\* ABI compatibility quirk: \*/ 호환성문제
25. **if** (!size) size가 0이면 SCHED 구조체 크기로 정해줌
26. size = SCHED\_ATTR\_SIZE\_VER0;  48 /\* sizeof first published struct \*/
28. **if** (size < SCHED\_ATTR\_SIZE\_VER0)  SCHED 구조체 크기보다 작으면 오류가 나므로 **goto**
29. **goto** err\_size;
31. /\*
32. \* If we're handed a bigger struct than we know of,
33. \* ensure all the unknown bits are 0 - i.e. new
34. \* user-space does not rely on any kernel feature
35. \* extensions we dont know about yet.
36. \*/
37. **if** (size > **sizeof**(\*attr)) { 만약 size가 우리가 아는 구조체 크기보다 큰 경우 모르는 모든 비트가 0인지 확인
38. unsigned **char** \_\_user \*addr;
39. unsigned **char** \_\_user \*end;
40. unsigned **char** val;
42. addr = (**void** \_\_user \*)uattr + **sizeof**(\*attr);
43. end  = (**void** \_\_user \*)uattr + size;
45. **for** (; addr < end; addr++) { size와 구조체 크기만큼 get\_user을 통해 비트 확인
46. ret = get\_user(val, addr);
47. **if** (ret)
48. **return** ret;
49. **if** (val)
50. **goto** err\_size;
51. }
52. size = **sizeof**(\*attr);
53. }
55. ret = copy\_from\_user(attr, uattr, size); // get\_user와 다르게 size를 지정하여 유저 영역의 데이터를 커널 영역으로 복사, 정상적 수행됐다면 0을 리턴한다 return 값에는 복사되지 않은 바이트 수가 들어간다.
56. **if** (ret)
57. **return** -EFAULT;  /\* Bad address \*/
59. /\*
60. \* XXX: Do we want to be lenient like existing syscalls; or do we want
61. \* to be strict and return an error on out-of-bounds values?
62. \*/
63. attr->sched\_nice = clamp(attr->sched\_nice, MIN\_NICE, MAX\_NICE);
64. //nice를 MIN과 MAX사이의 값으로 변경되게 해준다.
65. **return** 0;
67. err\_size:
68. put\_user(**sizeof**(\*attr), &uattr->size);
69. **return** -E2BIG;
70. }

arch/alpha/include/asm/uaccess.h get\_user()

lines 59

1. #define get\_user(x, ptr) \  용도 : 유저 영역의 데이터를 커널 영역으로 복사
2. 인자의 넘어온 변수의 길이만큼 복사한다.
3. 성공하면 0 오류는 -EFAULT, 오류 발생시 변수 x가 0으로 설정\*/
4. \_\_get\_user\_check((x), (ptr), **sizeof**(\*(ptr)))

lines 102 \_\_get\_user\_check

1. #define \_\_get\_user\_check(x, ptr, size)              \
2. ({                              \
3. **long** \_\_gu\_err = -EFAULT;                \ /\* Bad address \*/
4. unsigned **long** \_\_gu\_val = 0;             \
5. **const** \_\_typeof\_\_(\*(ptr)) \_\_user \*\_\_gu\_addr = (ptr); \
6. **if** (\_\_access\_ok((unsigned **long**)\_\_gu\_addr, size)) {  \
7. \_\_gu\_err = 0;                   \
8. **switch** (size) {                 \
9. **case** 1: \_\_get\_user\_8(\_\_gu\_addr); **break**;   \
10. **case** 2: \_\_get\_user\_16(\_\_gu\_addr); **break**;  \
11. **case** 4: \_\_get\_user\_32(\_\_gu\_addr); **break**;  \
12. **case** 8: \_\_get\_user\_64(\_\_gu\_addr); **break**;  \
13. **default**: \_\_get\_user\_unknown(); **break**;     \
14. }                       \
15. }                           \
16. (x) = (\_\_force \_\_typeof\_\_(\*(ptr))) \_\_gu\_val;        \
17. \_\_gu\_err;                       \
18. })

lines 57 put\_user

1. #define put\_user(x, ptr) \  용도 : 커널 영역의 데이터를 유저 영역으로 복사
2. 인자의 넘어온 변수의 길이만큼 복사한다.
3. 성공하면 0 오류는 -EFAULT, 오류 발생시 변수 x가 0으로 설정
4. \_\_put\_user\_check((\_\_typeof\_\_(\*(ptr)))(x), (ptr), **sizeof**(\*(ptr)))

lines 198 \_\_put\_user\_check

1. #define \_\_put\_user\_check(x, ptr, size)              \
2. ({                              \
3. **long** \_\_pu\_err = -EFAULT;                \/\* Bad address \*/
4. \_\_typeof\_\_(\*(ptr)) \_\_user \*\_\_pu\_addr = (ptr);       \
5. **if** (\_\_access\_ok((unsigned **long**)\_\_pu\_addr, size)) {  \
6. \_\_pu\_err = 0;                   \
7. **switch** (size) {                 \
8. **case** 1: \_\_put\_user\_8(x, \_\_pu\_addr); **break**;    \
9. **case** 2: \_\_put\_user\_16(x, \_\_pu\_addr); **break**;   \
10. **case** 4: \_\_put\_user\_32(x, \_\_pu\_addr); **break**;   \
11. **case** 8: \_\_put\_user\_64(x, \_\_pu\_addr); **break**;   \
12. **default**: \_\_put\_user\_unknown(); **break**;     \
13. }                       \
14. }                           \
15. \_\_pu\_err;                       \
16. })

include/linux/uaccess.h copy\_from\_user()

lines 144

1. copy\_from\_user(**void** \*to, **const** **void** \_\_user \*from, unsigned **long** n)
2. {
3. **if** (likely(check\_copy\_size(to, n, **false**))) // likely = if 분기문에서 True인 경우가 더 많을 것이라는 정보를 주어 성능 향상 시키는 함수
5. n = \_copy\_from\_user(to, from, n);
6. **return** n;
7. }

include/linux/thread\_info.h check\_copy\_size()

lines 138

1. check\_copy\_size(**const** **void** \*addr, **size\_t** bytes, **bool** is\_source)
2. {
3. **int** sz = \_\_compiletime\_object\_size(addr);
4. **if** (unlikely(sz >= 0 && sz < bytes)) { //likely 와 반대
5. **if** (!\_\_builtin\_constant\_p(bytes))
6. copy\_overflow(sz, bytes);
7. **else** **if** (is\_source)
8. \_\_bad\_copy\_from();
9. **else**
10. \_\_bad\_copy\_to();
11. **return** **false**;
12. }
13. check\_object\_size(addr, bytes, is\_source);
14. **return** **true**;
15. }

include/linux/uaccess.h \_copry\_from\_user()

lines 110

1. #ifdef INLINE\_COPY\_FROM\_USER // 이 옵션을 통해서 유저 어플리케이션에서 사용 될 때 인라인 함수로 제공, 커널에서 호출시에는 라이브러리를 통해 제공
2. **static** **inline** unsigned **long**
3. \_copy\_from\_user(**void** \*to, **const** **void** \_\_user \*from, unsigned **long** n)
4. { //user address(from)에서 kernel address(to)로 nbytes(n)만큼 복사한다.
5. unsigned **long** res = n;
6. might\_fault();
7. **if** (likely(access\_ok(VERIFY\_READ, from, n))) {
8. kasan\_check\_write(to, n);
9. res = raw\_copy\_from\_user(to, from, n);
10. }
11. **if** (unlikely(res))
12. memset(to + (n - res), 0, res);
13. **return** res;
14. } 정상적 수행됐다면 0을 리턴한다 **return** 값에는 복사되지 않은 바이트 수가 들어간다.
15. #else
16. **extern** unsigned **long**
17. \_copy\_from\_user(**void** \*, **const** **void** \_\_user \*, unsigned **long**);
18. #endif

include/linux/rcupdate.h rcu\_read\_lock()

lines 627

1. **static** **inline** **void** rcu\_read\_lock(**void**)  // RCU에서 데이터 참조를 위한 Critical Section을 시작 했다는 것을 통지
2. {
3. \_\_rcu\_read\_lock();
4. \_\_acquire(RCU); // 읽기를 위한 잠금 권한을 획득한다. 읽는 것은 다수의 스레드에서 접근이 가능하므로 다른 CPU에서 rcu\_read\_lock() 함수를 이용하여 중첩해서 사용이 가능하다.
5. rcu\_lock\_acquire(&rcu\_lock\_map);
6. RCU\_LOCKDEP\_WARN(!rcu\_is\_watching(),
7. "rcu\_read\_lock() used illegally while idle");
8. }

lines 80 \_\_rcu\_read\_lock()

1. **static** **inline** **void** \_\_rcu\_read\_lock(**void**)
2. {
3. **if** (IS\_ENABLED(CONFIG\_PREEMPT\_COUNT))
4. preempt\_disable();  // CPU간의 선점을 방지한다.
5. }

lines 679 rcu\_read\_unlock()

1. **static** **inline** **void** rcu\_read\_unlock(**void**)
2. {
3. RCU\_LOCKDEP\_WARN(!rcu\_is\_watching(),
4. "rcu\_read\_unlock() used illegally while idle");
5. \_\_release(RCU);   // \_\_acquire()에서 획득한 잠금을 해제한다.
6. \_\_rcu\_read\_unlock();
7. rcu\_lock\_release(&rcu\_lock\_map); /\* Keep acq info for rls diags. \*/
8. }

lines 86 \_\_rcu\_read\_unlock()

1. **static** **inline** **void** \_\_rcu\_read\_unlock(**void**)
2. {
3. **if** (IS\_ENABLED(CONFIG\_PREEMPT\_COUNT))
4. preempt\_enable(); // \_\_rcu\_read\_lock에서 CPU간의 선점을 방지를 해제한다.
5. }

kernel/sched/core.c

lines 4421 sched\_setattr()

1. **int** sched\_setattr(**struct** task\_struct \*p, **const** **struct** sched\_attr \*attr)
2. {
3. **return** \_\_sched\_setscheduler(p, attr, **true**, **true**);
4. }

lines 4143 \_\_sched\_setscheduler()

1. **static** **int** \_\_sched\_setscheduler(**struct** task\_struct \*p, **const** **struct** sched\_attr \*attr, **bool** user, **bool** pi)
2. {
3. **int** newprio = dl\_policy(attr->sched\_policy) ? MAX\_DL\_PRIO - 1 : // 정책이 deadline인지 확인 MAX\_DL\_PRIO = 0
4. MAX\_RT\_PRIO - 1 - attr->sched\_priority;
5. **int** retval, oldprio, oldpolicy = -1, queued, running;
6. **int** new\_effective\_prio, policy = attr->sched\_policy;
7. **const** **struct** sched\_class \*prev\_class;
8. **struct** rq\_flags rf;
9. **int** reset\_on\_fork;
10. **int** queue\_flags = DEQUEUE\_SAVE | DEQUEUE\_MOVE | DEQUEUE\_NOCLOCK;
11. // 0x02 /\* Matches ENQUEUE\_RESTORE \*/
12. //  0x04 /\* Matches ENQUEUE\_MOVE \*/
13. //      0x08 /\* Matches ENQUEUE\_NOCLOCK \*/
14. **struct** rq \*rq;
16. /\* The pi code expects interrupts enabled \*/
17. BUG\_ON(pi && in\_interrupt()); // in\_interrupt()는 현재 실행 중인 코드가 interrupt context인지 true로 반환해준다. false는 process context
18. recheck:
19. /\* Double check policy once rq lock held: \*/
20. **if** (policy < 0) {
21. reset\_on\_fork = p->sched\_reset\_on\_fork;
22. policy = oldpolicy = p->policy;
23. } **else** {
24. reset\_on\_fork = !!(attr->sched\_flags & SCHED\_FLAG\_RESET\_ON\_FORK);
26. **if** (!valid\_policy(policy)) // idle || fair(normal, batch) || rt(fifo, rr) || dl  인지 확인
27. **return** -EINVAL; // /\* Invalid argument \*/
28. }
30. **if** (attr->sched\_flags & ~(SCHED\_FLAG\_ALL | SCHED\_FLAG\_SUGOV)) // attr->sched\_flags & ~(0x01 | 0x02 | 0x04 | 0x10000000) 즉 XOOOOXXX bit에서 O에 1bit가 들어가게 된 경우 에러 검출
31. //
32. **return** -EINVAL;  /\* Invalid argument \*/
34. /\*
35. \* Valid priorities for SCHED\_FIFO and SCHED\_RR are
36. \* 1..MAX\_USER\_RT\_PRIO-1, valid priority for SCHED\_NORMAL,
37. \* SCHED\_BATCH and SCHED\_IDLE is 0.
38. \*/
39. // mm\_struct인 memory map mm변수  MAX\_RT\_PRIO와 MAX\_USER\_RT\_PRIO는 100으로 매크로 되어있다.
40. **if** ((p->mm && attr->sched\_priority > MAX\_USER\_RT\_PRIO-1) ||
41. (!p->mm && attr->sched\_priority > MAX\_RT\_PRIO-1))      // sched\_RT인 경우 우선순위가 100이 넘는 경우 오류를 검출한다.
42. **return** -EINVAL;  /\* Invalid argument \*/ sched
43. **if** ((dl\_policy(policy) && !\_\_checkparam\_dl(attr)) ||    // policy가 deadline 이면서 deaeline에서 매개변수나 deadline이 0인 경우, SCALE bit, MSB 설정 에러 등 에러 검출
44. (rt\_policy(policy) != (attr->sched\_priority != 0))) // policy가 rt인 경우 우선순위가 0인 경우, rt가 아닌데 우선순위가 있는 경우 에러 검출
45. **return** -EINVAL;  /\* Invalid argument \*/
47. /\*
48. \* Allow unprivileged RT tasks to decrease priority:
49. \*/   우선 순위를 높이고 다른 우선 순위를 설정하도록 허용, (다른 UID) 프로세스 자체적으로 FIFO 및 라운드 로빈 (실시간) 스케줄링 사용 허용, 다른 사람이 사용하는 스케줄링 알고리즘 처리 및 설정 방법.
50. 다른 프로세스에서 CPU 선호도 설정 허용
51. **if** (user && !capable(CAP\_SYS\_NICE)) {  // nice값을 사용하는 지 확인 (fair, rt)
52. **if** (fair\_policy(policy)) { // fair인 경우 nice값이 -20~19 사이인지 체크
53. **if** (attr->sched\_nice < task\_nice(p) &&
54. !can\_nice(p, attr->sched\_nice))
55. **return** -EPERM; /\* Operation not permitted \*/
56. }
58. **if** (rt\_policy(policy)) { // rt인 경우 우선순위 에러 체크
59. unsigned **long** rlim\_rtprio =
60. task\_rlimit(p, RLIMIT\_RTPRIO);
62. /\* Can't set/change the rt policy: \*/
63. **if** (policy != p->policy && !rlim\_rtprio)
64. **return** -EPERM; /\* Operation not permitted \*/
66. /\* Can't increase priority: \*/
67. **if** (attr->sched\_priority > p->rt\_priority &&
68. attr->sched\_priority > rlim\_rtprio)
69. **return** -EPERM; /\* Operation not permitted \*/
70. }
72. /\*
73. \* Can't set/change SCHED\_DEADLINE policy at all for now
74. \* (safest behavior); in the future we would like to allow
75. \* unprivileged DL tasks to increase their relative deadline
76. \* or reduce their runtime (both ways reducing utilization)
77. \*/
78. **if** (dl\_policy(policy)) // 여기서는 deadline 이용 x
79. **return** -EPERM; /\* Operation not permitted \*/
81. /\*
82. \* Treat SCHED\_IDLE as nice 20. Only allow a switch to
83. \* SCHED\_NORMAL if the RLIMIT\_NICE would normally permit it.
84. \*/
85. **if** (idle\_policy(p->policy) && !idle\_policy(policy)) { // idle은 nice 20으로 취급, nice값 에러
86. **if** (!can\_nice(p, task\_nice(p)))
87. **return** -EPERM; /\* Operation not permitted \*/
88. }
90. /\* Can't change other user's priorities: \*/
91. **if** (!check\_same\_owner(p)) // 다른 유저 우선순위 건들기 x
92. **return** -EPERM; /\* Operation not permitted \*/
94. /\* Normal users shall not reset the sched\_reset\_on\_fork flag: \*/
95. **if** (p->sched\_reset\_on\_fork && !reset\_on\_fork) //   sched\_reset\_on\_fork 리셋 x
96. **return** -EPERM; /\* Operation not permitted \*/
97. }
99. **if** (user) {
100. **if** (attr->sched\_flags & SCHED\_FLAG\_SUGOV)
101. **return** -EINVAL; /\* Invalid argument \*/
103. retval = security\_task\_setscheduler(p); // cpuset을 사용할 수 있는지 확인
104. **if** (retval)
105. **return** retval;
106. }
108. /\*
109. \* Make sure no PI-waiters arrive (or leave) while we are
110. \* changing the priority of the task:
111. \*
112. \* To be able to change p->policy safely, the appropriate
113. \* runqueue lock must be held.
114. \*/
115. rq = task\_rq\_lock(p, &rf); // policy을 바꿀 때 PI waiter (pi 란 프로세스설정이 상속이 되냐는 Boolean 변수다) 가 사라지지 않도록 런큐를 잠근다.
116. update\_rq\_clock(rq);
118. /\*
119. \* Changing the policy of the stop threads its a very bad idea:
120. \*/
121. **if** (p == rq->stop) { 스레드 정지정책 바뀌면 에러
122. task\_rq\_unlock(rq, p, &rf);
123. **return** -EINVAL; /\* Invalid argument \*/
124. }
126. /\*
127. \* If not changing anything there's no need to proceed further,
128. \* but store a possible modification of reset\_on\_fork.
129. \*/
130. **if** (unlikely(policy == p->policy)) {
131. **if** (fair\_policy(policy) && attr->sched\_nice != task\_nice(p)) fair 이면서 nice가 변경되었다면 **goto**
132. **goto** change;
133. **if** (rt\_policy(policy) && attr->sched\_priority != p->rt\_priority) rt 이면서 우선순위가 변경되었다면 **goto**
134. **goto** change;
135. **if** (dl\_policy(policy) && dl\_param\_changed(p, attr)) // deadline 이면서 매개변수 변하면 goto
136. **goto** change;
138. p->sched\_reset\_on\_fork = reset\_on\_fork; // 아무것도 바꾸지 않았을 때도 따로 저장해서 관리
139. task\_rq\_unlock(rq, p, &rf);
140. **return** 0;
141. }
142. change:
144. **if** (user) {
145. #ifdef CONFIG\_RT\_GROUP\_SCHED // cgroup의 rt 그룹 스케줄링을 지원하는 커널 옵션
146. /\*
147. \* Do not allow realtime tasks into groups that have no runtime
148. \* assigned.
149. \*/
150. **if** (rt\_bandwidth\_enabled() && rt\_policy(policy) && // 런타임 작업 그룹이 할당되어 있지 않으면서 rt일 경우 에러
151. task\_group(p)->rt\_bandwidth.rt\_runtime == 0 &&
152. !task\_group\_is\_autogroup(task\_group(p))) {
153. task\_rq\_unlock(rq, p, &rf);
154. **return** -EPERM; /\* Operation not permitted \*/
155. }
156. #endif
157. #ifdef CONFIG\_SMP // Symmetric multiprocessing 개별적으로 연산이 돌아감
158. **if** (dl\_bandwidth\_enabled() && dl\_policy(policy) && //dl이면서 대역폭 존재
159. !(attr->sched\_flags & SCHED\_FLAG\_SUGOV)) { // dl 에러 검출x 시
160. cpumask\_t \*span = rq->rd->span;
162. /\*
163. \* Don't allow tasks with an affinity mask smaller than
164. \* the entire root\_domain to become SCHED\_DEADLINE. We
165. \* will also fail if there's no bandwidth available.
166. \*/
167. **if** (!cpumask\_subset(span, &p->cpus\_allowed) || // cpumask < root\_domain이거나
168. rq->rd->dl\_bw.bw == 0) { // 런큐에 대역폭이 없으면 에러
169. task\_rq\_unlock(rq, p, &rf);
170. **return** -EPERM; /\* Operation not permitted \*/
171. }
172. }
173. #endif
174. }
176. /\* Re-check policy now with rq lock held: \*/
177. **if** (unlikely(oldpolicy != -1 && oldpolicy != p->policy)) { // 다시 한번 체크해보기 위한 goto
178. policy = oldpolicy = -1;
179. task\_rq\_unlock(rq, p, &rf);
180. **goto** recheck;
181. }
183. /\*
184. \* If setscheduling to SCHED\_DEADLINE (or changing the parameters
185. \* of a SCHED\_DEADLINE task) we need to check if enough bandwidth
186. \* is available.
187. \*/
188. **if** ((dl\_policy(policy) || dl\_task(p)) && sched\_dl\_overflow(p, policy, attr)) { //deadline으로 바꾸거나 매개변수를 바꾸는 경우에 충분한 대역폭이 있나 확인하기
189. task\_rq\_unlock(rq, p, &rf);
190. **return** -EBUSY;
191. }
193. p->sched\_reset\_on\_fork = reset\_on\_fork; // 현재 스케쥴링과 우선순위를 이전 변수에 넣는다.
194. oldprio = p->prio;
196. **if** (pi) {
197. /\*
198. \* Take priority boosted tasks into account. If the new
199. \* effective priority is unchanged, we just store the new
200. \* normal parameters and do not touch the scheduler class and
201. \* the runqueue. This will be done when the task deboost
202. \* itself.
203. \*/
204. new\_effective\_prio = rt\_effective\_prio(p, newprio); // rt의 새 우선순위 할당
205. **if** (new\_effective\_prio == oldprio) // 전 우선순위와 같다면 그대로 유지
206. queue\_flags &= ~DEQUEUE\_MOVE;
207. }
209. queued = task\_on\_rq\_queued(p); // p가 런큐에 큐 되어 있나
210. running = task\_current(rq, p); // 현재 런큐에 올라가있나
211. **if** (queued)
212. dequeue\_task(rq, p, queue\_flags); 프로세스가 더이상 실행 가능할 상태가 아닐 때
213. **if** (running)
214. put\_prev\_task(rq, p); 실행중인 테스크를 다시 queue에 넣는다
216. prev\_class = p->sched\_class; 현재 클래스를 prev에 저장
217. \_\_setscheduler(rq, p, attr, pi); 스케쥴러를 설정한다.
219. **if** (queued) {
220. /\*
221. \* We enqueue to tail when the priority of a task is
222. \* increased (user space view).
223. \*/
224. **if** (oldprio < p->prio) // 바뀐 우선순위가 더 큰 경우 큐에서 자리를 바꿔준다.
225. queue\_flags |= ENQUEUE\_HEAD;
227. enqueue\_task(rq, p, queue\_flags); // 프로세스가 실행가능한 상태로 들어간다.
228. }
229. **if** (running)
230. set\_curr\_task(rq, p); 테스크의 스케쥴링 클래스나 태스크 그룹을 바꿀때
232. check\_class\_changed(rq, p, prev\_class, oldprio); // 전 클래스와 전 우선순위로 현재의 클래스와 우선순위를 비교하여 제대로 바뀌었는지 확인
234. /\* Avoid rq from going away on us: \*/
235. preempt\_disable(); //선점 지연 비활성화
236. task\_rq\_unlock(rq, p, &rf); // policy을 바꿨으니 잠금을 해제한다.
238. **if** (pi)
239. rt\_mutex\_adjust\_pi(p); // 우선순위 설정을 한 경우에 pi체인을 다시 확인함
241. /\* Run balance callbacks after we've adjusted the PI chain: \*/
242. balance\_callback(rq); // rq lock을 acquire 실패시 rq lock과 선점을 다른 프로세스가 가지고 있을시에 rq에서 탈취함. 이 때 선점 지연 비활성화 상태다.
243. preempt\_enable(); //선점 지연 활성화
245. **return** 0;
246. }

kernel/sched/sched.h inline으로 정의된 policy

1. **static** **inline** **int** idle\_policy(**int** policy)
2. {
3. **return** policy == SCHED\_IDLE;
4. }
5. **static** **inline** **int** fair\_policy(**int** policy)
6. {
7. **return** policy == SCHED\_NORMAL || policy == SCHED\_BATCH;
8. }
10. **static** **inline** **int** rt\_policy(**int** policy)
11. {
12. **return** policy == SCHED\_FIFO || policy == SCHED\_RR;
13. }
15. **static** **inline** **int** dl\_policy(**int** policy)
16. {
17. **return** policy == SCHED\_DEADLINE;
18. }
19. **static** **inline** **bool** valid\_policy(**int** policy)
20. {
21. **return** idle\_policy(policy) || fair\_policy(policy) ||
22. rt\_policy(policy) || dl\_policy(policy);
23. }

kernel/sched/core.c \_\_setscheduler()

lines 4106

1. **static** **void** \_\_setscheduler(**struct** rq \*rq, **struct** task\_struct \*p,
2. **const** **struct** sched\_attr \*attr, **bool** keep\_boost)
3. {
4. \_\_setscheduler\_params(p, attr);
6. /\*
7. \* Keep a potential priority boosting if called from
8. \* sched\_setscheduler().
9. \*/
10. p->prio = normal\_prio(p); // task의 우선순위를 각 정책에 맞게 맞춰주고
11. **if** (keep\_boost) // pi가 true인 경우 즉 향상된 우선순위를 제공한 경우
12. p->prio = rt\_effective\_prio(p, p->prio);
14. **if** (dl\_prio(p->prio)) // 각각 설정한 스케쥴링 정책에 맞게 클래스도 변경한다.
15. p->sched\_class = &dl\_sched\_class;
16. **else** **if** (rt\_prio(p->prio))
17. p->sched\_class = &rt\_sched\_class;
18. **else**
19. p->sched\_class = &fair\_sched\_class;
20. }

lines 4080 \_\_setscheduler\_params()

1. **static** **void** \_\_setscheduler\_params(**struct** task\_struct \*p,
2. **const** **struct** sched\_attr \*attr)
3. {
4. **int** policy = attr->sched\_policy; // policy에 받아온 policy를 넣는다
6. **if** (policy == SETPARAM\_POLICY) // setparam\_policy은 -1로 -1 인 경우에는 함수가 변경하지 않도록 해줌
7. policy = p->policy;
9. p->policy = policy; // task의 policy를 현재 policy로 바꿔주고
11. **if** (dl\_policy(policy))  // dl인경우 dl에 맞는 매개변수(runtime, deadline, period, flags, bw, density)를 맞춰준다
12. \_\_setparam\_dl(p, attr);
13. **else** **if** (fair\_policy(policy)) // fair인 경우 현재 nice에서 default prio를 더해주는데 이 때 default prio는 120값을 가진다 (nice 범위 -20~19 = 40, MAX\_RT\_PRIO = 100, 100 + 20) 이 것을 static\_prio에 넣어준다.
14. p->static\_prio = NICE\_TO\_PRIO(attr->sched\_nice);
16. /\*
17. \* \_\_sched\_setscheduler() ensures attr->sched\_priority == 0 when
18. \* !rt\_policy. Always setting this ensures that things like
19. \* getparam()/getattr() don't report silly values for !rt tasks.
20. \*/
21. p->rt\_priority = attr->sched\_priority;
22. p->normal\_prio = normal\_prio(p); // policy가 무엇이냐에 따라서 각각 최대 우선순위에서 1을 뺀 값을 넣어준다
23. set\_load\_weight(p, **true**); // load\_weight란 공정한 cpu시간을 할당하기 위하여 우선순위의 비율을 정한 것이다.  위에서 하지 않은 정책들인 Idle과 other인 경우에 필요한 load\_weight을 설정해준다.
24. }

**1-3 2) 스케쥴링 클래스 분석 (rt클래스)**

**kernel/sched/sched.h**

**lines 1558**

1. **extern** **const** **struct** sched\_class stop\_sched\_class;
2. **extern** **const** **struct** sched\_class dl\_sched\_class;
3. **extern** **const** **struct** sched\_class rt\_sched\_class;
4. **extern** **const** **struct** sched\_class fair\_sched\_class;
5. **extern** **const** **struct** sched\_class idle\_sched\_class;

**5개의 클래스 정의**

**Lines 1476 struct sched\_class**

1. **struct** sched\_class {
2. **const** **struct** sched\_class \*next;  // 우선순위 순서 stop -> dl -> rt -> fair -> idle (for\_each\_class()에서 돔)
3. **void** (\*enqueue\_task) (**struct** rq \*rq, **struct** task\_struct \*p, **int** flags); // 프로세스가 실행 가능한 상태로 진입
4. **void** (\*dequeue\_task) (**struct** rq \*rq, **struct** task\_struct \*p, **int** flags); // 프로세스가 더 이상 실행 가능한 상태가 아닐때
5. **void** (\*yield\_task)   (**struct** rq \*rq); // 프로세스가 스스로 yield() 시스템콜을 실행했을 때
6. **bool** (\*yield\_to\_task)(**struct** rq \*rq, **struct** task\_struct \*p, **bool** preempt); 현재 프로세서를 다른 스레드에 양보하거나 스레드를 프로세서쪽으로 가속시킴
8. **void** (\*check\_preempt\_curr)(**struct** rq \*rq, **struct** task\_struct \*p, **int** flags); // 현재 실행 중인 프로세스를 선점할 수 있는지 검사
10. /\*
11. \* It is the responsibility of the pick\_next\_task() method that will
12. \* return the next task to call put\_prev\_task() on the @prev task or
13. \* something equivalent.
14. \*
15. \* May return RETRY\_TASK when it finds a higher prio class has runnable
16. \* tasks.
17. \*/
18. **struct** task\_struct \* (\*pick\_next\_task)(**struct** rq \*rq, **struct** task\_struct \*prev,  **struct** rq\_flags \*rf); //실행할 다음 프로세스를 선택
19. **void** (\*put\_prev\_task)(**struct** rq \*rq, **struct** task\_struct \*p); // 실행중인 태스크를 다시 내부 자료구조에 큐잉
21. #ifdef CONFIG\_SMP // SMP설정일 때 (symmetric multiprocessing)
22. **int**  (\*select\_task\_rq)(**struct** task\_struct \*p, **int** task\_cpu, **int** sd\_flag, **int** flags); //  wake 또는 fork 밸런싱인 경우에만 가장 낮은 우선 순위부터 요청한 태스크의 우선순위 범위 이내에서 동작할 수 있는 cpu를 찾아 선택한다.
23. **void** (\*migrate\_task\_rq)(**struct** task\_struct \*p);
25. **void** (\*task\_woken)(**struct** rq \*this\_rq, **struct** task\_struct \*task); // task가 실행중이지 않고 조정하지 않는다면 밀어낸다
27. **void** (\*set\_cpus\_allowed)(**struct** task\_struct \*p, **const** **struct** cpumask \*newmask); // 요청 태스크가 운영될 수 있는 cpu들을 지정한다
29. **void** (\*rq\_online)(**struct** rq \*rq); // rq 온라인 설정
30. **void** (\*rq\_offline)(**struct** rq \*rq); // rq 오프라인 설정
31. #endif
33. **void** (\*set\_curr\_task)(**struct** rq \*rq);  //태스크의 스케줄링 클래스나 태스크 그룹을 바꿀때
34. **void** (\*task\_tick)(**struct** rq \*rq, **struct** task\_struct \*p, **int** queued); // 타이머 틱 함수가 호출
35. **void** (\*task\_fork)(**struct** task\_struct \*p); // cfs인 경우 fork 된 새 child task의 vruntime을 결정해준다.
36. **void** (\*task\_dead)(**struct** task\_struct \*p); // dl인 경우 total\_bw 속 dl\_bw 감소후 dl 타이머를 중지시켜준다.
38. /\*
39. \* The switched\_from() call is allowed to drop rq->lock, therefore we
40. \* cannot assume the switched\_from/switched\_to pair is serliazed by
41. \* rq->lock. They are however serialized by p->pi\_lock.
42. \*/
43. **void** (\*switched\_from)(**struct** rq \*this\_rq, **struct** task\_struct \*task); // from으로부터 policy 교체, 큐에 있는 마지막 작업이면 pull을 통해 다른 작업을 불러온다.
44. **void** (\*switched\_to)  (**struct** rq \*this\_rq, **struct** task\_struct \*task); // to로 policy 교체
45. **void** (\*prio\_changed) (**struct** rq \*this\_rq, **struct** task\_struct \*task, **int** oldprio); //  task의 priority가 같은 type 내에서 바뀔 때 호출
47. unsigned **int** (\*get\_rr\_interval)(**struct** rq \*rq, **struct** task\_struct \*task); // rt클래스에서 사용 rr인 경우 인터벌을 리턴하고 fifo인 경우 0 리턴
49. **void** (\*update\_curr)(**struct** rq \*rq);// 현재 런타임을 업데이트한다.
51. #define TASK\_SET\_GROUP      0
52. #define TASK\_MOVE\_GROUP     1
54. #ifdef CONFIG\_FAIR\_GROUP\_SCHED // cgroup의 cfs 그룹 스케줄링을 지원하는 커널 옵션
56. **void** (\*task\_change\_group)(**struct** task\_struct \*p, **int** type);
57. #endif
58. };

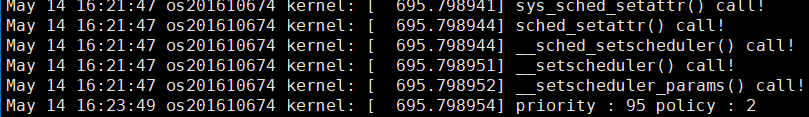
**그 중 rt클래스의 구조체를 분석해보았다.**

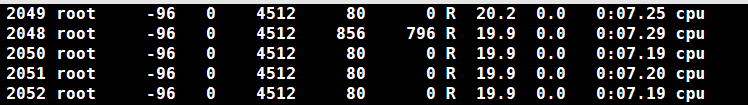
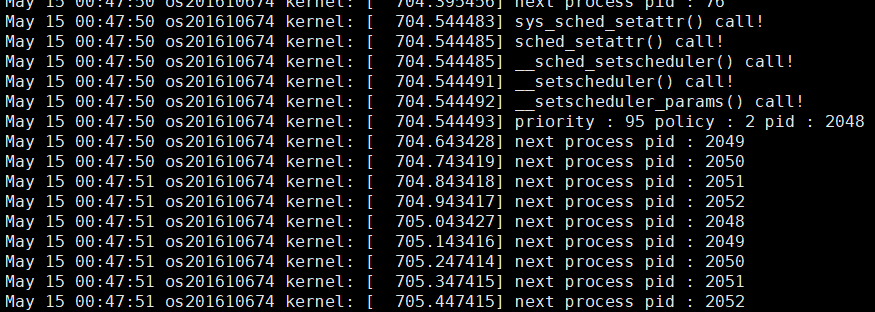
**kernel/sched/rt.c rt\_sched\_class**

**lines 2366**

1. **const** **struct** sched\_class rt\_sched\_class = {
2. .next           = &fair\_sched\_class, // 우선순위 순서 stop -> dl -> rt -> fair -> idle (for\_each\_class)
3. .enqueue\_task       = enqueue\_task\_rt, // 프로세스가 실행 가능한 상태로 진입
4. .dequeue\_task       = dequeue\_task\_rt, //프로세스가 더 이상 실행 가능한 상태가 아닐때
5. .yield\_task     = yield\_task\_rt, //프로세스가 스스로 yield() 시스템콜을 실행했을 때
7. .check\_preempt\_curr = check\_preempt\_curr\_rt,//현재 실행 중인 프로세스를 선점(preempt)할 수 있는지 검사
9. .pick\_next\_task     = pick\_next\_task\_rt,//실행할 다음 프로세스를 선택
10. .put\_prev\_task      = put\_prev\_task\_rt,//실행중인 태스크를 다시 내부 자료구조에 큐잉
12. #ifdef CONFIG\_SMP
13. .select\_task\_rq     = select\_task\_rq\_rt, // wake 또는 fork 밸런싱인 경우에만 가장 낮은 우선 순위부터 요청한 태스크의 우선순위 범위 이내에서 동작할 수 있는 cpu를 찾아 선택한다.
15. .set\_cpus\_allowed       = set\_cpus\_allowed\_common, //요청 태스크가 운영될 수 있는 cpu들을 지정한다
16. .rq\_online              = rq\_online\_rt, // rq 온라인 설정
17. .rq\_offline             = rq\_offline\_rt, // rq 오프라인 설정
18. .task\_woken     = task\_woken\_rt, // task가 실행중이지 않고 조정하지 않는다면 밀어낸다
19. .switched\_from      = switched\_from\_rt, // rt에서 클래스 바꿈
20. #endif
22. .set\_curr\_task          = set\_curr\_task\_rt, //태스크의 스케줄링 클래스나 태스크 그룹을 바꿀때
23. .task\_tick      = task\_tick\_rt, // 타이머 틱 함수가 호출
25. .get\_rr\_interval    = get\_rr\_interval\_rt, // rr인 경우 인터벌 리턴, fifo인 경우 0 리턴
27. .prio\_changed       = prio\_changed\_rt, // 우선순위 교체
28. .switched\_to        = switched\_to\_rt, // rt클래스로 switch함
30. .update\_curr        = update\_curr\_rt, // 현재 런타임을 업데이트한다.
31. };

**1-3 3) Printk()을 통한 콜체인 확인과 스케쥴링 클래스 (rt-rr) 분석 (캡쳐화면)**



1-3 1) 그림과 같이 총 5개의 함수 콜체인이 일어나는 것을 알 수 있었다. 그리고 유저 어플리케이션에서 넣은 우선순위와 policy가 제대로 전달받았는지 확인하기 위해 출력해본 결과 제대로 전달 되었다. 

1-3 2)에서 분석한 스케쥴링 클래스 중 RT클래스에서 다음 프로세스를 정확히 찾아가는지 확인하기 위해 pick\_next\_task\_rt() 함수에 다음 프로세스의 pid를 출력하게 하고 유저 어플리케이션을 프로세스 5개로 수행해보았다. 그 결과 유저 어플리케이션은 RT클래스 중 라운드 로빈으로 설정 되고나서 프로세스 5개가 돌아가면서 수행되는 것을 볼 수 있었다.

* + **특히 어려웠던 점 및 해결 방안**

제일 먼저 1-1 유저 어플리케이션에서 각각 프로세스들의 시간을 구할 때 애를 먹었다. Get\_time()함수를 처음 써보면서 time\_spec 구조체를 공부하고 그 차이를 구할 때 앞쪽 시간의 nsec이 음수가 될 때 예외처리를 해줘야 하는 것을 생각하기 어려웠다. 인터넷에서 찾아본 함수를 인용하여 어플리케이션을 완성 시켰다.

그리고 1-3에서 sched\_setattr()을 분석하는 부분에서 양이 상당히 방대했으며 꽤나 많은 함수에 들어가서 여러 동작을 취하는데 그 동작 하나하나 쫓아가면서 기능을 알아 보았는데 어려웠다. 특히 pi변수는 느닷없이 등장했으나 제대로 된 설명을 찾기가 어려워서 이해가 안되었다. bootlin에 있는 주석으로는 확실히 설명이 부족하여 대부분 구글링을 하며 여러 설명들을 찾아보며 이해하였다. 이렇게 하나하나 분석해가니 더디지만 확실히 이해를 할 수 있었다. 전부 분석을 하며 공부를 같이 하니 콜 체인에 printk()을 커널에 넣어주니 한 번에 원하는 답을 얻어 낼 수 있었다.

처음 스케쥴링 클래스가 아닌 schedule() 함수를 분석하면서 스케쥴링되는 과정을 먼저 알게 되고 스케쥴링 클래스를 분석해보았다. 처음에는 생각을 잘못하여 함수를 먼저 분석하게 되었는데 이후 클래스를 보니까 구조체에서 어떤 부분이 어떤 역할을 하는지에 대해 쉽게 이해할 수 있었다.

마지막으로 커널을 수정하고 컴파일하면서 리눅스에 대해 조금 더 알게 된 거 같다. 항상 윈도우 운영체제에서 코딩만 해오다가 이번 과제를 처음에 직면했을 때는 마냥 어려워 보였지만 계속 공부하면서 추상적으로만 이해했던 개념들을 제대로 이해한 것 같다.