과제 #2

- ~~Simple~~ Linux CPU Scheduler

제출 기한: 6/30 (화) 23:59

제출 방법: IEILMS “과제 #2”

질의 응답 프로토콜

1. IEILMS Q/A 게시판 확인
    1. PLEASE!! (과제 1 종료 이후 조회 수가 거의 학생 수와 비슷!!)

2. 인터넷 (구글) 검색
    1. 문제가 생겼을 때의 error message 를 문장 혹은 키워드로 구글 검색
    2. 한글 자료는 많이 없으므로, 영어 사용

3. IEILMS Q/A 질문답변 게시판에 질문 올리기
    1. JCLOUD 인스턴스 포트 번호 쓰기 (19xxx)
    2. 가능한 자세한 정보를 제공할 것 (명령 수행 및 결과 화면 등)

* 코드 등 과제 결과물이 노출되어야 하는 경우, 반드시 메일을 사용할 것
(문제가 있는 질문 글은 임의로 삭제할 수 있음)

Tip. 컴파일 시, make clean 사용

- Make clean
  - 이전 컴파일에 생성된 오브젝트 파일 등의 내용을 모두 제거함
    - Configuration 파일 등은 그대로 남아있음
  - 이후 처음부터 새롭게 컴파일 수행하게 됨
    - 대신 시간은 오래 걸림
- 왜 make clean 을 써야 하나?
  - 헤더 파일, 심볼 테이블 등이 변경되는 경우,
    기존 컴파일된 오브젝트 파일, 링크된 파일들과 충돌이 발생하여,
    커널 빌드는 정상 수행되지만,
    재부팅 후 커널이 정상적으로 동작하지 않을 수 있음.
  - 이러한 경우를 방지하기 위해 커널 코드에 문제가 없는데도 오류가 발생하면
    make clean 을 수행하고, 다시 빌드하는 것이 문제를 해결할 수 있음
- make clean; make -j 24 && make -j 24 modules_install install

Tip. 커널 설치 확인

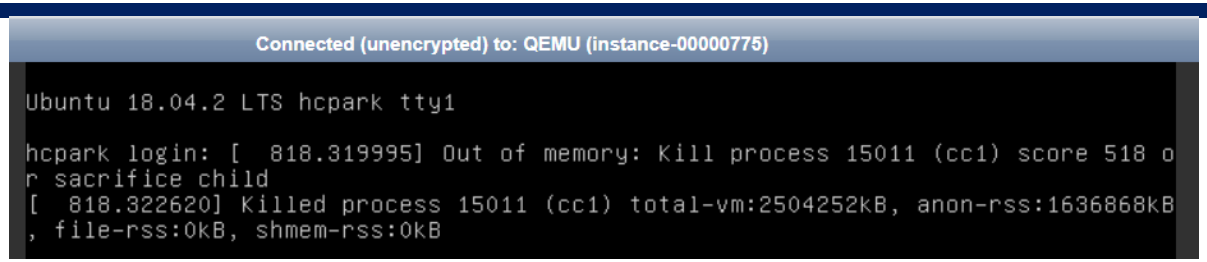
- 재컴파일하고 재설치된 커널은 /boot 디렉토리로 복사됨
- 이번에 컴파일한 커널이 잘 설치되었는지 알려면?

```
root@hcpark:~# ls -al /boot/vmlinuz-4.18.20-hcpark
-rw-r--r-- 1 root root 8386432 May 29 13:20 /boot/vmlinuz-4.18.20-hcpark
root@hcpark:~# date
Fri May 29 13:24:19 KST 2020
root@hcpark:~#
```

- 재부팅 후, 커널 이름 확인
  - hostnamectl or uname

```
root@hcpark:~# hostnamectl
   Static hostname: hcpark
         Icon name: computer-vm
           Chassis: vm
        Machine ID: 32cd22818fee491ab370e3a29dbdb4f7
           Boot ID: 0c03f891a8ea4342a809ac9ff9601279
    Virtualization: kvm
  Operating System: Ubuntu 18.04.2 LTS
            Kernel: Linux 4.18.20-hcpark
      Architecture: x86-64
root@hcpark:~# uname -r
4.18.20-hcpark
root@hcpark:~#
```

Tip. Swap 생성하기



- 간혹 drivers 컴파일 과정에서 에러나는 이유: 메모리 부족
- Swap 공간: 메모리 부족 시, 물리 메모리 내용을 임시로 스토리지 장치의 swap 영역에 저장함으로써 물리 메모리 공간 확보
- Commands: 루트 디렉토리에 swap 영역으로 사용할 2GB 파일을 생성하고, 스왑 파일로 설정한 후, 스왑 영역으로 지정함
  - $ sudo -s
  - # fallocate -l 2147483648 /swap (소문자 L)
  - # chmod 600 /swap
  - # mkswap /swap
  - # swapon /swap
- 확인: free -m
- 재부팅 시 다시 수행

```
root@hcpark:~# free -m
              total        used        free      shared  buff/cache   available
Mem:           2999         147        1562           0        1288        2673
Swap:          2047           0        2047
```

Tip. printk()를 쓸때 주의할 점.

- 커널 내에서 printk()를 찍다보면 문제가 발생할 수 있습니다.
- 아주 빈번하게 호출되는 루틴 안에서 썼다가 화면에 출력하는 일을 계속 하느라 시스템이 멈춰버리는게 대표적입니다.
- 이러한 경우만 주의하면 별 문제는 없지만, 반드시 주요한 파일들은 sz 명령 등을 이용해 본인이 백업해두시기 바랍니다.

Tip. J-Cloud 에서 재부팅하는 법

- JCloud 에서 인스턴스 콘솔을 띄워둔 상태에서, 또다른 Jcloud 탭 또는 창을 하나 더 띄워서 인스턴스를 hard reboot 시키십시오.
  - soft reboot 은 잘 동작하지 않습니다
- 그리고 인스턴스 콘솔 창을 새로 고침하면 부트 메뉴를 확인할 수 있습니다.
  - 미리 ppt 에서 처럼 부트 메뉴가 뜨게 설정해둬야겠지요?

- 기본 커널 등을 이용해 정상 부팅 시키고 수정 작업을 진행하십시오.

과제 학습 목표 및 내용

- 학습 목표

  2-1. 리눅스 CPU 스케줄러 구조를 학습한다.

  - LKM (Loadable Kernel Module)을 사용해본다.

  2-2. 세 가지 스케줄링 알고리즘들을 구현한다.

  - 병렬로 동작하는 프로세스들에 대해 다양한 알고리즘을 적용하여 알고리즘 간의 차이점을 파악한다.
  - 직접 만든 스케줄러를 이용해 Context switching overhead를 측정해본다

- 내용
  - 리눅스 CPU 스케줄러 구현
    - 2-1. 리눅스 CPU 스케줄링 구조 수정
    - 2-2. 스케줄링 알고리즘 구현 및 context switching overhead 분석
      - firm deadline: 6/30(화) 23:59 (지각 제출 없음)

과제 제출 내용

- 보고서와 소스코드를 압축(zip)해서 1개 파일로 제출
- 보고서: 학번.pdf (* PDF 아니면 채점 안 함!!!! (NO PDF, NO SCORE!))
  - 표지: 제목, 학번, 이름 (1장)
  - 2-1: 커널 수정 내용 및 수정 이유 설명, 결과 화면 캡처 등
    - 소스코드 포함: mysched.c, Makefile
  - 2-2: 각 스케줄링 기법의 구현에 대한 설명과 결과 캡처, 결과 분석 내용
    - 캡처 화면: 과제 목표를 달성했음을 알 수 있는 화면으로 구성
    - 소스 코드: 수정한 부분, 핵심적인 부분만 포함할 것. 주석 필수. 캡처 무방.
    - Context switching overhead 분석 내용을 3페이지 이내로 작성
  - 특히 어려웠던 점 및 해결 방법 (1장 이내)
- 소스코드
  - 수정한 소스 코드 모두 하나의 tar 파일로 압축해서 제출
  - 각 파일의 용도를 기술한 readme.txt 작성해서 최상위 디렉토리에 포함시킬 것

순서

2-1. 리눅스 CPU 스케줄링 구조 수정

2-2. 스케줄링 알고리즘 구현 & Context switching overhead 분석

2-1.
리눅스 CPU 스케줄링 구조 수정

2-1. 과제 내용

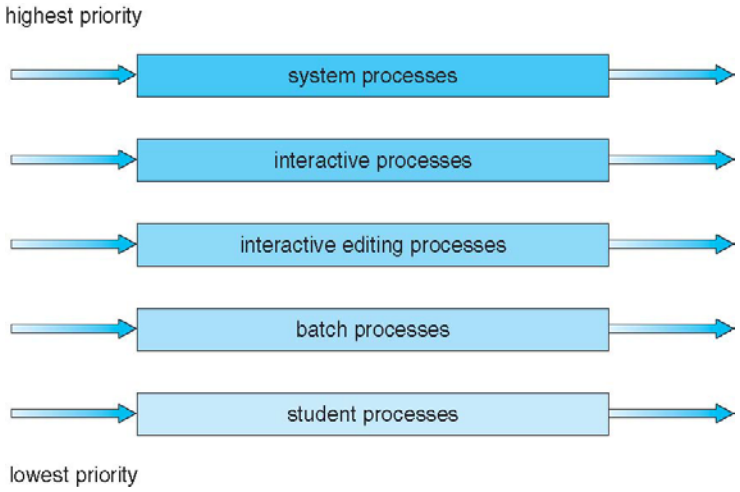
- 기존 리눅스 스케줄링 구조를 수정
  - 새로운 스케줄링 클래스를 추가
  - 새로운 스케줄링 알고리즘을 모듈 형태로 구현하도록 수정
- 이제 커널 코드를 진짜 수정해보자!
  - 커널 코드 보는 방법
    - 방대한 코드에서 내가 필요한 부분을 파악할 수 있어야 함
    - 우선 구조부터 파악하고, 세부 내용을 추적해 감
    - 커널 수정은 대부분 커널 코드를 이해하는 과정
  - 유용한 도구
    - 책: “최신의” 리눅스 커널 책
      - Understanding Linux kernel, Linux kernel development 등
    - Web-based cross reference tool: lxr, Elixir Cross Referencer

2-1. 과제 내용

1. 새로운 스케줄링 클래스를 추가
   - mysched_class
     - **Fair 와 idle class 사이에 새로운 스케줄링 클래스를 추가**
     - 실제 스케줄링을 관리하는 코드: 모듈에서 구현
     - **커널 내에서 수정할 부분: 모듈과 연계하여 동작할 수 있도록 기존 코드 수정**
       - 커널 컴파일, 설치, 재부팅 필요 <- 이 단계를 최소한으로 줄이는 것이 중요
2. 모듈 컴파일 및 실제 스케줄링 기법 구현
   - mysched.ko
     - 2-1 에서는 모듈 예제 소스 및 Makefile 제공. 각자 본인이 수정한 커널 내용에 맞추어 변경 후 사용.
     - 2-2 에서 실제 FIFO, Round Robin, Weighted Round Robin 등의 알고리즘 구현
   - 수행 확인
     - Mysched.ko 모듈을 올리고, cpu.c 를 수정해서 돌린 다음, 정상 종료 되면 성공
     - Mysched.ko 모듈 내의 mysched_class 로 cpu 프로세스를 관리하는 것

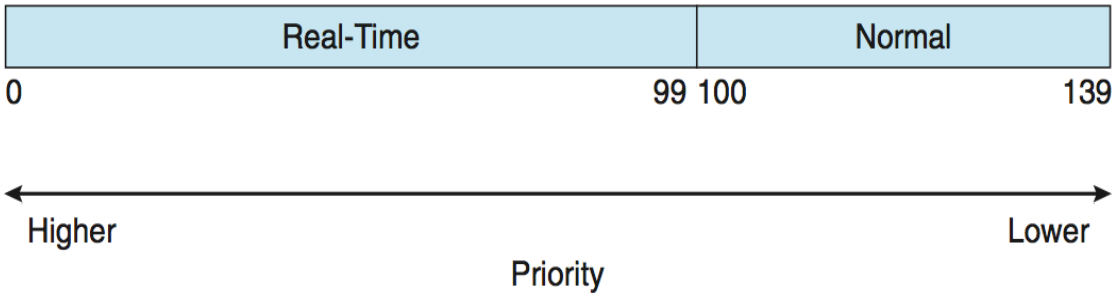
기존 리눅스 구조

- Priority-based multilevel queue



linux/kernel/sched/sched.h

```
1302  #define sched_class_highest (&stop_sched_class)
1303  #define for_each_class(class) \
1304     for (class = sched_class_highest; class; class = class->next)
1305
1306  extern const struct sched_class stop_sched_class;
1307  extern const struct sched_class dl_sched_class;
1308  extern const struct sched_class rt_sched_class;
1309  extern const struct sched_class fair_sched_class;
1310  extern const struct sched_class idle_sched_class;
1311
1312
```



(과제1) 커널: sched_setattr() call chain

```
04541: /**
04542:  * sys_sched_setattr - same as above, but with extended sched_attr
04543:  * @pid: the pid in question.
04544:  * @uattr: structure containing the extended parameters.
04545:  * @flags: for future extension.
04546:  */
04547: SYSCALL_DEFINE3(sched_setattr, pid_t, pid, struct sched_attr __user *, uattr,
04548:                 unsigned int, flags)
04549: {
04550:     struct sched_attr attr;
04551:     struct task_struct *p;
04552:     int retval;
04553:
04554:     if (!uattr || pid < 0 || flags)
04555:         return -EINVAL;
04556:
04557:     retval = sched_copy_attr(uattr, &attr);
04558:     if (retval)
04559:         return retval;
04560:
04561:     if ((int)attr.sched_policy < 0)
04562:         return -EINVAL;
04563:
04564:     rcu_read_lock();
04565:     retval = -ESRCH;
04566:     p = find_process_by_pid(pid);
04567:     if (p != NULL)
04568:         retval = sched_setattr(p, &attr);
04569:     rcu_read_unlock();
04570:
04571:     return retval;
04572: } ? end SYSCALL_DEFINE3 ?
04573:
```

(과제1) 커널: sched_setattr() call chain

```
04377: /**
04378:  * sched_setscheduler - change the scheduling policy and/or RT priority of a thread.
04379:  * @p: the task in question.
04380:  * @policy: new policy.
04381:  * @param: structure containing the new RT priority.
04382:  *
04383:  * Return: 0 on success. An error code otherwise.
04384:  *
04385:  * NOTE that the task may be already dead.
04386:  */
04387: int sched_setscheduler(struct task_struct *p, int policy,
04388:                        const struct sched_param *param)
04389: {
04390:     return _sched_setscheduler(p, policy, param, true);
04391: }
04392: EXPORT_SYMBOL_GPL(sched_setscheduler);
04393:
04394: int sched_setattr(struct task_struct *p, const struct sched_attr *attr)
04395: {
04396:     return __sched_setscheduler(p, attr, true, true);
04397: }
04398: EXPORT_SYMBOL_GPL(sched_setattr);
04399:
```

(과제1) 커널: sched_setattr() call chain

```
04359: static int _sched_setscheduler(struct task_struct *p, int policy,
04360:                       const struct sched_param *param, bool check)
04361: {
04362:     struct sched_attr attr = {
04363:         .sched_policy   = policy,
04364:         .sched_priority = param->sched_priority,
04365:         .sched_nice     = PRIO_TO_NICE(p->static_prio),
04366:     };
04367:
04368:     /* Fixup the legacy SCHED_RESET_ON_FORK hack. */
04369:     if ((policy != SETPARAM_POLICY) && (policy & SCHED_RESET_ON_FORK)) {
04370:         attr.sched_flags |= SCHED_FLAG_RESET_ON_FORK;
04371:         policy &= ~SCHED_RESET_ON_FORK;
04372:         attr.sched_policy = policy;
04373:     }
04374:
04375:     return __sched_setscheduler(p, &attr, check, true);
04376: }
```

```
04120: static int __sched_setscheduler(struct task_struct *p,
04121:                       const struct sched_attr *attr,
04122:                       bool user, bool pi)
```

```
04319:
04320:     queued = task_on_rq_queued(p);
04321:     running = task_current(rq, p);
04322:     if (queued)
04323:         dequeue_task(rq, p, queue_flags);
04324:     if (running)
04325:         put_prev_task(rq, p);
04326:
04327:     prev_class = p->sched_class;
04328:     __setscheduler(rq, p, attr, pi);
04329:
04330:     if (queued) {
04331:         /*
```

(과제1) 커널: sched_setattr() call chain

```
/* Actually do priority change: must hold pi & rq lock. */
static void __setscheduler(struct rq *rq, struct task_struct *p,
                           const struct sched_attr *attr, bool keep_boost)
{
        __setscheduler_params(p, attr);

        /*
         * Keep a potential priority boosting if called from
         * sched_setscheduler().
         */
        p->prio = normal_prio(p);
        if (keep_boost)
                p->prio = rt_effective_prio(p, p->prio);

        if (dl_prio(p->prio))
                p->sched_class = &dl_sched_class;
        else if (rt_prio(p->prio))
                p->sched_class = &rt_sched_class;
        else
                p->sched_class = &fair_sched_class;
}
```

(과제1) 커널: sched_setattr() call chain

```
static void __setscheduler_params(struct task_struct *p,
                const struct sched_attr *attr)
{
        int policy = attr->sched_policy;

        if (policy == SETPARAM_POLICY)
                policy = p->policy;

        p->policy = policy;

        if (dl_policy(policy))
                __setparam_dl(p, attr);
        else if (fair_policy(policy))
                p->static_prio = NICE_TO_PRIO(attr->sched_nice);

        /*
         * __sched_setscheduler() ensures attr->sched_priority == 0 when
         * !rt_policy. Always setting this ensures that things like
         * getparam()/getattr() don't report silly values for !rt tasks.
         */
        p->rt_priority = attr->sched_priority;
        p->normal_prio = normal_prio(p);
        set_load_weight(p, true);
}
```

리눅스 스케줄러 코드

- 실제 리눅스 스케줄링을 수행하는 코드

linux/kernel/sched/core.c

```
03472: asmlinkage __visible void __sched schedule(void)
03473: {
03474:     struct task_struct *tsk = current;
03475:
03476:     sched_submit_work(tsk);
03477:     do {
03478:         preempt_disable();
03479:         __schedule(false);
03480:         sched_preempt_enable_no_resched();
03481:     } while (need_resched());
03482: }
03483: EXPORT_SYMBOL(schedule);
```

```
03304: /*
03305:  * __schedule() is the main scheduler function.
03306:  *
03307:  * The main means of driving the scheduler and thus entering this function are:
03308:  *
03309:  *   1. Explicit blocking: mutex, semaphore, waitqueue, etc.
03310:  *
03311:  *   2. TIF_NEED_RESCHED flag is checked on interrupt and userspace return
03312:  *      paths. For example, see arch/x86/entry_64.S.
03313:  *
03314:  *      To drive preemption between tasks, the scheduler sets the flag in timer
03315:  *      interrupt handler scheduler_tick().
03316:  *
03317:  *   3. Wakeups don't really cause entry into schedule(). They add a
03318:  *      task to the run-queue and that's it.
03319:  *
03320:  *      Now, if the new task added to the run-queue preempts the current
03321:  *      task, then the wakeup sets TIF_NEED_RESCHED and schedule() gets
03322:  *      called on the nearest possible occasion:
03323:  *
03324:  *       - If the kernel is preemptible (CONFIG_PREEMPT=y):
03325:  *
03326:  *         - in syscall or exception context, at the next outmost
03327:  *           preempt_enable(). (this might be as soon as the wake_up()'s
03328:  *           spin_unlock()! )
03329:  *
03330:  *         - in IRQ context, return from interrupt-handler to
03331:  *           preemptible context
03332:  *
03333:  *       - If the kernel is not preemptible (CONFIG_PREEMPT is not set)
03334:  *         then at the next:
03335:  *
03336:  *          - cond_resched() call
03337:  *          - explicit schedule() call
03338:  *          - return from syscall or exception to user-space
03339:  *          - return from interrupt-handler to user-space
03340:  *
03341:  * WARNING: must be called with preemption disabled!
03342:  */
03343: static void __sched notrace __schedule(bool preempt)
```

리눅스 스케줄러 코드

linux/kernel/sched/core.c

```
03403:
03404:      if (task_on_rq_queued(prev))
03405:          update_rq_clock(rq);
03406:
03407:      next = pick_next_task(rq, prev, &rf);
03408:      clear_tsk_need_resched(prev);
03409:      clear_preempt_need_resched();
03410:
03411:      if (likely(prev ! = next)) {
03412:          rq->nr_switches++;
03413:          rq->curr = next;
03414:          ++*switch_count;
03415:
03416:          trace_sched_switch(preempt, prev, next);
03417:
03418:          /* Also unlocks the rq: */
03419:          rq = context_switch(rq, prev, next, &rf);
03420:      } else {
03421:          rq->clock_update_flags &= ~(RQCF_ACT_SKIP|RQC
03422:          rq_unpin_lock(rq, &rf);
03423:          raw_spin_unlock_irq(&rq->lock);
03424:      }
03425:
03426:      balance_callback(rq);
03427: } ? end __schedule ?
```

```
03265: /*
03266:  * Pick up the highest-prio task:
03267:  */
03268: static inline struct task_struct *
03269: pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_fla
03270: {
03271:      const struct sched_class *class;
03272:      struct task_struct *p;
03273:
03274:      /*
03275:       * Optimization: we know that if all tasks are in
03276:       * the fair class we can call that function directly:
03277:       */
03278:      if (likely(rq->nr_running == rq->cfs.h_nr_running)) {
03279:          p = fair_sched_class.pick_next_task(rq, prev, rf);
03280:          if (unlikely(p == RETRY_TASK))
03281:              goto ↓again;
03282:
03283:          /* Assumes fair_sched_class->next == idle_sched_class */
03284:          if (unlikely(! p))
03285:              p = idle_sched_class.pick_next_task(rq, prev, rf);
03286:
03287:          return p;
03288:      }
03289:
03290: again:
03291:      for_each_class(class) {
03292:          p = class->pick_next_task(rq, prev, rf);
03293:          if (p) {
03294:              if (unlikely(p == RETRY_TASK))
03295:                  goto ↑again;
03296:              return p;
03297:          }
03298:      }
03299:
03300:      /* The idle class should always have a runnable task: */
03301:      BUG();
03302: } ? end pick_next_task ?
```

리눅스 스케줄러 코드: context switch (참고)

```
02828: /*
02829:  * context_switch - switch to the new MM and the new thread's register state.
02830:  */
02831: static __always_inline struct rq *
02832: context_switch(struct rq *rq, struct task_struct *prev,
02833:                struct task_struct *next, struct rq_flags *rf)
02834: {
02835:     struct mm_struct *mm, *oldmm;
02836:
02837:     prepare_task_switch(rq, prev, next);
02838:
02839:     mm = next->mm;
02840:     oldmm = prev->active_mm;
02841:     /*
02842:      * For paravirt, this is coupled with an exit in switch_to to
02843:      * combine the page table reload and the switch backend into
02844:      * one hypercall.
02845:      */
02846:     arch_start_context_switch(prev);
02847:
02848:     if (!mm) {
02849:         next->active_mm = oldmm;
02850:         atomic_inc(&oldmm->mm_count);
02851:         enter_lazy_tlb(oldmm, next);
02852:     } else
02853:         switch_mm_irqs_off(oldmm, mm, next);
02854:
02855:     if (!prev->mm) {
02856:         prev->active_mm = NULL;
02857:         rq->prev_mm = oldmm;
02858:     }
02859:
02860:     rq->clock_update_flags &= ~(RQCF_ACT_SKIP|RQCF_REQ_SKIP);
02861:
02862:     /*
02863:      * Since the runqueue lock will be released by the next
02864:      * task (which is an invalid locking op but in the case
02865:      * of the scheduler it's an obvious special-case), so we
02866:      * do an early lockdep release here:
02867:      */
02868:     rq_unpin_lock(rq, rf);
02869:     spin_release(&rq->lock.dep_map, 1, _THIS_IP_);
02870:
02871:     /* Here we just switch the register state and the stack. */
02872:     switch_to(prev, next, prev);
02873:     barrier();
02874:
02875:     return finish_task_switch(prev);
02876: } ? end context_switch ?
```

linux/kernel/sched/core.c

Arch/x86/include/asm/switch_to.h

```
00068: #define switch_to(prev, next, last)                     \
00069: do {                                                     \
00070:     prepare_switch_to(prev, next);                       \
00071:                                                          \
00072:     ((last) = __switch_to_asm((prev), (next)));          \
00073: } while (0)
00074:
```

Arch/x86/entry/entry_64.s

```
ENTRY(__switch_to_asm)
    /*
     * Save callee-saved registers
     * This must match the order in inactive_task_frame
     */
    pushq   %rbp
    pushq   %rbx
    pushq   %r12
    pushq   %r13
    pushq   %r14
    pushq   %r15

    /* switch stack */
    movq    %rsp, TASK_threadsp(%rdi)
    movq    TASK_threadsp(%rsi), %rsp

#ifdef CONFIG_CC_STACKPROTECTOR
    movq    TASK_stack_canary(%rsi), %rbx
    movq    %rbx, PER_CPU_VAR(irq_stack_union)+stack_canary_offset
#endif

    /* restore callee-saved registers */
    popq    %r15
    popq    %r14
    popq    %r13
    popq    %r12
    popq    %rbx
    popq    %rbp

    jmp __switch_to
END(__switch_to_asm)
```

리눅스 스케줄링 클래스의 구성

- 리눅스는 각 스케줄러 클래스 별로 서로 다른 알고리즘을 사용할 수 있는 framework 을 제공함

linux/kernel/sched/sched.h

```
01328: struct sched_class {
01329:     const struct sched_class *next;
01330:
01331:     void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
01332:     void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
01333:     void (*yield_task) (struct rq *rq);
01334:     bool (*yield_to_task) (struct rq *rq, struct task_struct *p, bool preempt);
01335:
01336:     void (*check_preempt_curr) (struct rq *rq, struct task_struct *p, int flags);
01337:
01338:     /*
01339:      * It is the responsibility of the pick_next_task() method that will
01340:      * return the next task to call put_prev_task() on the @prev task or
01341:      * something equivalent.
01342:      *
01343:      * May return RETRY_TASK when it finds a higher prio class has runnable
01344:      * tasks.
01345:      */
01346:     struct task_struct * (*pick_next_task) (struct rq *rq,
01347:                                             struct task_struct *prev,
01348:                                             struct rq_flags *rf);
01349:     void (*put_prev_task) (struct rq *rq, struct task_struct *p);
01350:
```

스케줄러 내에서 클래스 함수를 호출하는 예

```
03265: /*
03266:  * Pick up the highest-prio task:
03267:  */
03268: static inline struct task_struct *
03269: pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_fla
03270: {
03271:     const struct sched_class *class;
03272:     struct task_struct *p;
03273:
03274:     /*
03275:      * Optimization: we know that if all tasks are in
03276:      * the fair class we can call that function directly:
03277:      */
03278:     if (likely(rq->nr_running == rq->cfs.h_nr_running)) {
03279:         p = fair_sched_class.pick_next_task(rq, prev, rf);
03280:         if (unlikely(p == RETRY_TASK))
03281:             goto ↓again;
03282:
03283:         /* Assumes fair_sched_class->next == idle_sched_class */
03284:         if (unlikely(!p))
03285:             p = idle_sched_class.pick_next_task(rq, prev, rf);
03286:
03287:         return p;
03288:     }
03289:
03290: again:
03291:     for_each_class(class) {
03292:         p = class->pick_next_task(rq, prev, rf);
03293:         if (p) {
03294:             if (unlikely(p == RETRY_TASK))
03295:                 goto ↑again;
03296:             return p;
03297:         }
03298:     }
03299:
03300:     /* The idle class should always have a runnable task: */
03301:     BUG();
03302: } ? end pick_next_task ?
```

Fair class 예제: CFS scheduler 사용

linux/kernel/sched/core.c

```
09440:
09441: /*
09442:  * All the scheduling class methods:
09443:  */
09444: const struct sched_class fair_sched_class = {
09445:     .next               = &idle_sched_class,
09446:     .enqueue_task       = enqueue_task_fair,
09447:     .dequeue_task       = dequeue_task_fair,
09448:     .yield_task         = yield_task_fair,
09449:     .yield_to_task      = yield_to_task_fair,
09450:
09451:     .check_preempt_curr = check_preempt_wakeup,
09452:
09453:     .pick_next_task     = pick_next_task_fair,
09454:     .put_prev_task      = put_prev_task_fair,
09455:
09456: #ifdef CONFIG_SMP
09457:     .select_task_rq     = select_task_rq_fair,
09458:     .migrate_task_rq    = migrate_task_rq_fair,
09459:
09460:     .rq_online          = rq_online_fair,
09461:     .rq_offline         = rq_offline_fair,
09462:
09463:     .task_dead          = task_dead_fair,
09464:     .set_cpus_allowed   = set_cpus_allowed_common,
09465: #endif
09466:
09467:     .set_curr_task      = set_curr_task_fair,
09468:     .task_tick          = task_tick_fair,
09469:     .task_fork          = task_fork_fair,
09470:
09471:     .prio_changed       = prio_changed_fair,
09472:     .switched_from      = switched_from_fair,
09473:     .switched_to        = switched_to_fair,
09474:
09475:     .get_rr_interval    = get_rr_interval_fair,
09476:
09477:     .update_curr        = update_curr_fair,
09478:
09479: #ifdef CONFIG_FAIR_GROUP_SCHED
09480:     .task_change_group  = task_change_group_fair,
09481: #endif
09482: };
09483:
```

```
04751:
04752: /*
04753:  * The enqueue_task method is called before nr_running is
04754:  * increased. Here we update the fair scheduling stats and
04755:  * then put the task into the rbtree:
04756:  */
04757: static void
04758: enqueue_task_fair(struct rq *rq, struct task_struct *p, int flags)
04759: {
04760:     struct cfs_rq *cfs_rq;
04761:     struct sched_entity *se = &p->se;
04762:
04763:     /*
04764:      * If in_iowait is set, the code below may not trigger any cpufreq
04765:      * utilization updates, so do it here explicitly with the IOWAIT flag
04766:      * passed.
04767:      */
```

```
06248:
06249: static struct task_struct *
06250: pick_next_task_fair(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
06251: {
06252:     struct cfs_rq *cfs_rq = &rq->cfs;
06253:     struct sched_entity *se;
06254:     struct task_struct *p;
06255:     int new_tasks;
06256:
06257: again:
06258: #ifdef CONFIG_FAIR_GROUP_SCHED
06259:     if (!cfs_rq->nr_running)
06260:         goto idle;
06261:
06262:     if (prev->sched_class != &fair_sched_class)
06263:         goto simple;
06264:
06265:     /*
06266:      * Because of the set_next_buddy() in dequeue_task_fair() it is rather
06267:      * likely that a next task is from the same cgroup as the current.
06268:      *
06269:      * Therefore attempt to avoid putting and setting the entire cgroup
06270:      * hierarchy, only change the part that actually changes.
06271:      */
06272:
06273:     do {
06274:         struct sched_entity *curr = cfs_rq->curr;
```

LKM: Loadable Kernel Module

- 커널 코드를 수정할 때 문제점: 시간!!
  - OS 커널의 경우, 수정한 내용에 대한 확인이 너무 오래 걸림
  - Development cycle: 컴파일, 설치, 재부팅, 실행
- LKM: 커널이 수행되고 있는 중에, 커널 코드를 추가할 수 있는 방법
  - 새로운 코드가 커널 영역 메모리에 삽입되고, 수행 가능
  - 고민할 점 (2-1 과제의 핵심!!)
    - 모듈 컴파일 시, 기존 커널의 심볼에 어떻게 접근할 수 있나?
    - 모듈로 로딩된 코드를 기존 커널에서 어떻게 연계할 것인가?
- 관련 명령
  - 모듈 삽입 : insmod mysched.ko (모듈 파일명 입력)
  - 모듈 제거 : rmmod mysched (이때는 모듈 이름만)
  - lsmod: 현재 삽입된 모듈 리스트 확인
  - 모듈의 삽입, 제거 여부는 printk() 를 이용해 출력 후, dmesg로 확인

LKM: Loadable Kernel Module

- 모듈의 기본 형태

```
#include <linux/module.h>
#include <linux/init.h>

MODULE_LICENSE( "GPL" );
MODULE_AUTHOR( "Module Author" );
MODULE_DESCRIPTION( "Module Description" );

static int __init mod_entry_func( void )
{
  return 0;
}

static void __exit mod_exit_func( void )
{
  return;
}

module_init( mod_entry_func );
module_exit( mod_exit_func );
```

Module macros

Module contructor / destructor

Entry / exit macros

LKM: Loadable Kernel Module

- 모듈 컴파일 방법
  - Makefile 이용
  - “make” 수행
- 결과물
  - .ko 파일

```
obj-m += mysched.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

```
root@hcpark-VirtualBox:~/proj2/mysched# make clean; make
make -C /lib/modules/4.10.0hcpark_mysched/build M=/home/hcpark/proj2/mysched clean
make[1]: Entering directory '/home/hcpark/v4.10'
  CLEAN   /home/hcpark/proj2/mysched/.tmp_versions
  CLEAN   /home/hcpark/proj2/mysched/Module.symvers
make[1]: Leaving directory '/home/hcpark/v4.10'
make -C /lib/modules/4.10.0hcpark_mysched/build M=/home/hcpark/proj2/mysched modules
make[1]: Entering directory '/home/hcpark/v4.10'
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/hcpark/proj2/mysched/mysched.mod.o
  LD [M]  /home/hcpark/proj2/mysched/mysched.ko
make[1]: Leaving directory '/home/hcpark/v4.10'
root@hcpark-VirtualBox:~/proj2/mysched#
```

LKM: mysched.c

- 제공하는 모듈: mysched.c
  - 하나의 프로세스만 관리하는 very simple scheduler
  - 동작: Global scheduler가 pick_next_task() 를 수행하면 관리 중인 프로세스를 리턴함

```c
static struct task_struct *
pick_next_task_fifo(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
    //printk(KERN_DEBUG "MYMOD: pick_next_task_fifo CALLED\n");

    if(task!=NULL) {
        put_prev_task(rq, prev);
        //printk(KERN_DEBUG "MYMOD: pick_next_task_fifo return task p\n");
        return task;
    }

    return NULL;
}

static void
enqueue_task_fifo(struct rq *rq, struct task_struct *p, int flags)
{
    printk(KERN_INFO "MYMOD: enqueue_task_fifo CALLED task = %p\n", p);
    add_nr_running(rq, 1);

    task = p;
}

static void
dequeue_task_fifo(struct rq *rq, struct task_struct *p, int flags)
{
    printk(KERN_INFO "MYMOD: dequeue_task_fifo CALLED\n");
    sub_nr_running(rq, 1);

    task = NULL;
}
```

```c
const struct sched_class my_sched_class = {
    .next           = &idle_sched_class,

    .enqueue_task       = enqueue_task_fifo,
    .dequeue_task       = dequeue_task_fifo,
    .yield_task     = yield_task_fifo,

    .check_preempt_curr = check_preempt_curr_fifo,

    .pick_next_task     = pick_next_task_fifo,
    .put_prev_task      = put_prev_task_fifo,


    .set_curr_task          = set_curr_task_fifo,
    .task_tick      = task_tick_fifo,

    .get_rr_interval    = get_rr_interval_fifo,

    .prio_changed       = prio_changed_fifo,
    .switched_to        = switched_to_fifo,
    .update_curr        = update_curr_fifo,
};
```

LKM: mysched.c

- 수정할 부분
  - init_mysched(), exit_mysched() : 모듈의 적재 및 해제 시 수행
  - myclass(): 임의로 넣어둔 함수. 각자의 구현에 따라 사용하지 않아도 좋고, 수정해서 활용해도 됨.
  - (2-1에서는) 타 함수는 수정하지 말 것. 전역 변수 등은 필요시 추가, 수정 가능

```
void myclass (struct task_struct *p) {
        printk(KERN_INFO "MYMOD: myclass CALLED\n");
}

static int __init init_mysched(void)
{
    const struct sched_class *class;

    for_each_class(class)
        printk(KERN_INFO "INIT_MOD: class = %p\n", class);

    return 0;

}

static void __exit exit_mysched(void)
{
    const struct sched_class *class;

    for_each_class(class)
        printk(KERN_INFO "EXIT_MOD: class = %p\n", class);

    return;

}
```

LKM: mysched.c

- 작업 위치:
  - /usr/src/linux-source-4.18.0/linux-source-4.18.0/kernel/sched/
  - 기존 커널 소스에 존재하는 디렉토리
  - 하위에 mysched 디렉토리를 생성하고, 제공된 mysched.c 을 해당 디렉토리에 copy
  - Makefile은 직접 작성

```
root@hcpark:/usr/src/linux-source-4.18.0/linux-source-4.18.0/kernel/sched/mysched# ls -al
total 16
drwxr-xr-x 2 root root 4096 May 28 22:27 .
drwxr-xr-x 3 root root 4096 May 28 22:26 ..
-rw-r--r-- 1 root root  199 May 28 22:26 Makefile
-rwxr--r-- 1 root root 3409 May 28 22:26 mysched.c
```

- 수정
  - 본인이 수정한 커널 내용에 맞춰 모듈 소스 수정
- 컴파일
  - 위 작업 디렉토리에서 make 수행

```
1  obj-m += mysched.o
2
3  KBUILD_EXTRA_SYMBOLS += /proc/kallsyms
4
5  all:
6      make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
7
8  clean:
9      make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
10
11
```

- 모듈 적재 및 해제
  - insmod mysched.ko
  - rmmod mysched
  - (컴파일 및 적재 시, 동작 중인 커널 버전이 모듈 컴파일한 커널 버전과 동일해야 함)

HINT: 커널-LKM 연계 방법

- Multi-queue 스케줄링 방법
  - 상위 클래스의 큐부터 순회하며,
    큐 내의 프로세스들을 각 큐의 스케줄링 정책에 따라 수행함
  - 즉, 순회를 하는 과정에서 fair 다음에 idle 이 아니라, 새로 추가한 mysched 를 확인하도록 해야 함
- 그러나 커널 코드에는 mysched class가 없음
  - Mysched class는 LKM에만 존재함
  - 그렇다면, 순회를 하는 코드에서는,
    LKM이 load 되었을 때와, load 되지 않았을 때를 미리 구분하여 코드가 작성되어 있어야 함.
    - 어떻게 LKM Load 여부를 판단할 수 있을까? <- 첫 번째 해결 과제
  - sched_setattr() 등을 이용해 스케줄링 정책을 변경할 때도 유사한 문제 발생
    - 기존 코드에는 없고, 모듈이 적재되었을 때만 새로운 스케줄링 클래스로 포함 <- 두 번째
  - 사용자가 변경 요청한 클래스의 validity 를 검사하는 코드가 존재함
    - 그 코드는 내 스케줄러를 모르니까, 수정해주어야 함 <- 세 번째
  - Fair class 의 경우, const 로 선언되어 수정할 수 없음
    - 그러나 모듈 적재 여부에 따라 클래스 순서를 변경하기 위해서는 수정이 불가피함. <- 네 번째
  - 모듈: 모듈 삽입 시, fair 와 idle 사이에 새로운 mysched_class 삽입하고, 모듈 제거 시, 원복 <- 다섯 번째
  - (+) 새로운 클래스를 구분하기 위한 번호 지정해줄 것 (예. 7번 스케줄링 클래스. 기존 클래스들 번호 참조)

모듈에서 커널 심볼 접근

- EXPORT_SYMBOL
  - 모듈에서 접근할 수 있도록, 커널 심볼을 외부에 노출시켜줌
  - Export 되지 않은 심볼을 모듈에서 접근하면?
    - 컴파일 시 경고 발생, Insmod 실패

```
make -C /lib/modules/4.10.0hcpark_mysched/build M=/home/hcpark/proj2/mysched modules
make[1]: Entering directory '/home/hcpark/v4.10'
  Building modules, stage 2.
  MODPOST 1 modules
WARNING: "stop_sched_class" [/home/hcpark/proj2/mysched/mysched.ko] undefined!
make[1]: Leaving directory '/home/hcpark/v4.10'
root@hcpark-VirtualBox:~/proj2/mysched#
```

```
#if CONFIG_MYSCHED
void (*set_class_my) (struct task_struct *p);

EXPORT_SYMBOL(set_class_my);

EXPORT_SYMBOL(idle_sched_class);
EXPORT_SYMBOL(fair_sched_class);
EXPORT_SYMBOL(rt_sched_class);
EXPORT_SYMBOL(resched_curr);
#endif
```

* stop_sched_class, dl_sched_class는 core.c 에 선언되지 않기 때문에, 선언된 파일을 찾아 선언문 이후에 EXPORT_SYMBOL()을 사용해야 함

모듈에서 커널 심볼 접근

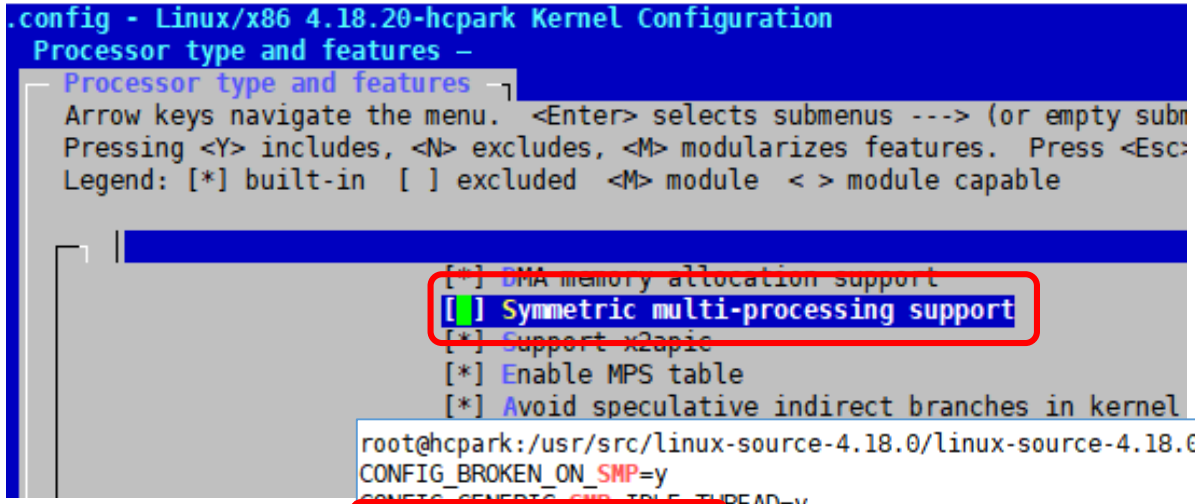
```
WARNING: "idle_sched_class" [/usr/src/linux-source-4.18.0/linux-source-4.18.0/kernel/sched/mysched/mysched.ko] undefined!
WARNING: "stop_sched_class" [/usr/src/linux-source-4.18.0/linux-source-4.18.0/kernel/sched/mysched/mysched.ko] undefined!
  CC      /usr/src/linux-source-4.18.0/linux-source-4.18.0/kernel/sched/mysched/mysched.mod.o
  LD [M]  /usr/src/linux-source-4.18.0/linux-source-4.18.0/kernel/sched/mysched/mysched.ko
make[1]: Leaving directory '/usr/src/linux-source-4.18.0/linux-source-4.18.0'
root@hcpark:/usr/src/linux-source-4.18.0/linux-source-4.18.0/kernel/sched/mysched# insmod mysched.ko
insmod: ERROR: could not insert module mysched.ko: Unknown symbol in module
root@hcpark:/usr/src/linux-source-4.18.0/linux-source-4.18.0/kernel/sched/mysched# dmesg | tail -n 5
[1190742.374581] QNX4 filesystem 0.2.3 registered.
[1856184.710553] mysched: loading out-of-tree module taints kernel.
[1856184.711058] mysched: module verification failed: signature and/or required key missing - tainting kernel
[1856184.711514] mysched: Unknown symbol idle_sched_class (err 0)
[1856184.711604] mysched: Unknown symbol stop_sched_class (err 0)
root@hcpark:/usr/src/linux-source-4.18.0/linux-source-4.18.0/kernel/sched/mysched#
```

- 모듈 컴파일 시, undefined symbol 경고가 나오는 경우
  - 현재 모듈 내에서는 모르는 symbol 이므로, 링크 과정에서 실체와 bind가 되어야 함
    - 커널 내에서 해당 심볼 들에 대해 EXPORT_SYMBOL 선언을 해주어야 함
    - 커널 수정, 재컴파일, 재설치, 재부팅 필요
  - Insmod 에서, unknown symbol 에러가 나오면 EXPORT_SYMBOL 적용이 제대로 안된 것
    - Dmesg 로 어떤 심볼에서 에러가 났는지 알 수 있음

모듈 적재 성공 후, No-SMP 커널 설정

- 커널 모듈 적재가 성공한 다음, 수행 확인은 SMP support 를 끄고 사용할 것
  - 1개 코어만 사용. 동기화 등 복잡한 문제를 제거하기 위해.
  - Makefile 을 수정해서 기존 커널과 구분: 상황에 따라 선택적으로 사용
- make menuconfig -> processor type and features
  - Space bar 를 이용하여 선택 해제할 것
  - 반드시 저장하고 종료
  - 설정 확인 방법: .config 파일에서 CONFIG_SMP 를 확인
  - 커널 전체 재컴파일: make clean; make -j 24 && make -j 24 modules_install install
  - 재부팅 후, lscpu 수행하여 CPU 개수 확인

```
# SPDX-License-Identifier: GPL-2.0
VERSION = 4
PATCHLEVEL = 18
SUBLEVEL = 20
EXTRAVERSION = -hcpark-nosmp
NAME = Merciless Moray
```




```
ubuntu@hcpark:~$ lscpu
Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
CPU(s):                  1
On-line CPU(s) list:     0
```

```
root@hcpark:/usr/src/linux-source-4.18.0/linux-source-4.18.0# cat .config | grep SMP
CONFIG_BROKEN_ON_SMP=y
CONFIG_GENERIC_SMP_IDLE_THREAD=y
# CONFIG_SMP is not set
CONFIG_SCSI_SAS_HOST_SMP=y
CONFIG_VIDEO_VP27SMPX=m
root@hcpark:/usr/src/linux-source-4.18.0/linux-source-4.18.0#
```

수행 내용 확인

- No-SMP 커널로 부팅, mysched.ko 로딩
- 과제 1에서 작성한 cpu.c 프로그램의 수정
  - RTRR 대신 새롭게 추가한 스케줄링 클래스로 변경 요청하도록 수정
  - attr.sched_policy 필드에 SCHED_RR 대신 새로운 스케줄링 클래스 번호 기입
    - 예. SCHED_MYCLASS (7)
  - attr.sched_priority = 0; <- 우선 순위도 0으로 변경
- CPU 프로그램 수행
  - 정상 종료된다면 성공한 것
  - 디버깅
    - 새로운 스케줄링 클래스로 변경되는지 여부 확인 (printk() and dmesg)
    - Mysched 모듈의 코드에서 printk() 구문 출력 여부 확인
- Tip
  - 문제가 생기면 즉각 시스템이 중단되어, dmesg 를 확인할 수 없음
  - tail -f /var/log/kern.log
    - 커널 메시지를 즉각 확인 가능함
    - 작업 세션과 별도로 세션을 열어놓고 진행

결과 예시

```
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /usr/src/linux-source-4.18.0/linux-source-4.18.0/kernel/sched/mysched/mysched.mod.o
  LD [M]  /usr/src/linux-source-4.18.0/linux-source-4.18.0/kernel/sched/mysched/mysched.ko
make[1]: Leaving directory '/usr/src/linux-source-4.18.0/linux-source-4.18.0'
root@hcpark:/usr/src/linux-source-4.18.0/linux-source-4.18.0/kernel/sched/mysched# insmod mysched.ko
root@hcpark:/usr/src/linux-source-4.18.0/linux-source-4.18.0/kernel/sched/mysched# dmesg
```

```
May 30 00:38:15 hcpark kernel: [  190.932248] HCPARK: sched_setattr called
May 30 00:38:15 hcpark kernel: [  190.932257] MYMOD: myclass CALLED
May 30 00:38:15 hcpark kernel: [  190.932263] MYMOD: enqueue_task_fifo CALLED task = 00000000d1546c2e
May 30 00:38:15 hcpark kernel: [  190.932264] MYMOD: set_curr_task_fifo CALLED
May 30 00:38:15 hcpark kernel: [  190.932267] MYMOD: switched_to_fifo CALLED new = 00000000d1546c2e
May 30 00:38:16 hcpark kernel: [  191.938045] MYMOD: dequeue_task_fifo CALLED
May 30 00:38:16 hcpark kernel: [  191.938047] MYMOD: enqueue_task_fifo CALLED task = 00000000d1546c2e
May 30 00:38:16 hcpark kernel: [  191.938048] MYMOD: set_curr_task_fifo CALLED
May 30 00:38:16 hcpark kernel: [  191.938054] MYMOD: dequeue_task_fifo CALLED
May 30 00:39:08 hcpark kernel: [  244.203566] EXIT_MOD: class = 0000000060834378
May 30 00:39:08 hcpark kernel: [  244.203568] EXIT_MOD: class = 00000000d9bcfdf4
May 30 00:39:08 hcpark kernel: [  244.203568] EXIT_MOD: class = 0000000038e5f4a9
May 30 00:39:08 hcpark kernel: [  244.203569] EXIT_MOD: class = 00000000c77fdf8c
```

```
root@hcpark:~/proj2# gcc -o cpu cpu.c
root@hcpark:~/proj2# ./cpu 1 1
** START: Processes = 01 Time = 01 s
PROCESS #00 count = 0125 100 ms
PROCESS #00 count = 0257 100 ms
PROCESS #00 count = 0390 100 ms
PROCESS #00 count = 0523 100 ms
PROCESS #00 count = 0681 100 ms
PROCESS #00 count = 0844 100 ms
PROCESS #00 count = 1006 100 ms
PROCESS #00 count = 1167 100 ms
PROCESS #00 count = 1329 100 ms
PROCESS #00 count = 1491 100 ms
DONE!! PROCESS #00 : 001491 1003 ms
root@hcpark:~/proj2#
```

2-2.
스케줄링 알고리즘 구현 &

Context switching overhead 분석

과제 학습 목표 및 내용

- 학습 목표

  2-1. 리눅스 CPU 스케줄러 구조를 학습한다.

  - LKM (Loadable Kernel Module)을 사용해본다.

  2-2. 세 가지 스케줄링 알고리즘들을 구현한다.

  - 병렬로 동작하는 프로세스들에 대해 다양한 알고리즘을 적용하여 알고리즘 간의 차이점을 파악한다.
  - 직접 만든 스케줄러를 이용해 Context switching overhead를 측정해본다

- 내용
  - 리눅스 CPU 스케줄러 구현
    - 2-1. 리눅스 CPU 스케줄링 구조 수정
    - 2-2. 스케줄링 알고리즘 구현 및 context switching overhead 분석
      - firm deadline: 6/30(화) 23:59 (지각 제출 없음)

2-2. 과제 내용

- 간단한 스케줄링 기법 세 가지를 리눅스 커널에 구현
  - FIFO
  - Round robin
  - Weighted round robin : priority-based proportional sharing

  ✓ 2-1 에서 작성한 모듈을 기반으로 각각 분리된 파일에 알고리즘 구현
    - fifo.c, rr.c, wrr.c
    - 제출 내용: 정상 동작 여부를 확인할 수 있도록 결과 화면 캡처
    - 2-1의 수행 시, 함수들이 어떤 순서로 호출되면서 스케줄링이 동작하는지 분석
- 본인이 작성한 알고리즘을 토대로 context switching 오버헤드를 분석
  - 과제 1을 참고하고, 측정 방식은 자유
  - 정확하게 측정할 수 있는 방식을 고민해볼 것
- cpu.c 프로그램 수정 필요
  - sched_setattr()을 fork() 이후, 각 프로세스가 각각 수행시키도록 해야 함

FIFO

- 단순 FIFO 방식
  - 앞서 들어온 프로세스가 모두 종료되어야 새로운 프로세스가 수행 시작됨
- 커널 list_head 자료 구조를 활용해서 클래스 내부 run queue 를 관리할 것
  - Include/linux/list.h
  - sched/rt.c 참고
    - struct sched_rt_entity *rt_se;
  - kmalloc() 사용: 커널 내 동적메모리 할당

```
01385: struct sched_rt_entity {
01386:     struct list_head run_list;
01387:     unsigned long timeout;
01388:     unsigned long watchdog_stamp;
01389:     unsigned int time_slice;
01390:     unsigned short on_rq;
01391:     unsigned short on_list;
01392:
01393:     struct sched_rt_entity *back;
01394: #ifdef CONFIG_RT_GROUP_SCHED
01395:     struct sched_rt_entity  *parent;
01396:     /* rq on which this entity is (to be) queued: */
01397:     struct rt_rq        *rt_rq;
01398:     /* rq "owned" by this entity/group: */
01399:     struct rt_rq        *my_q;
01400: #endif
01401: };
```

- Hint
  - Jcloud: Instance 메뉴에서 Hard-reboot
  - tail -f /var/log/syslog
    - Dmesg 하지 않고 즉각 커널 메시지 확인할 수 있음
    - 에러가 있는 경우, 커널이 다운되기 전까지 출력한 메시지를 볼 수 있음
  - Queue 의 전체 상태를 dump 하면서 진행해야 정확한 동작을 알 수 있음
  - rmmod mysched && make clean; make && insmod mysched.ko && dmesg -c

수행 예

```
root@hcpark:~/proj2# PROCESS #00 count = 0542 100 ms
DONE!! PROCESS #00 : 000542 1005 ms
./cpu 2 1
** START: Processes = 02 Time = 01 s
        Creating Process: #0
PROCESS #01 count = 0123 100 ms
PROCESS #01 count = 0255 100 ms
PROCESS #01 count = 0370 100 ms
PROCESS #01 count = 0502 100 ms
PROCESS #01 count = 0634 100 ms
PROCESS #01 count = 0767 100 ms
PROCESS #01 count = 0929 100 ms
PROCESS #01 count = 1091 100 ms
PROCESS #01 count = 1254 100 ms
PROCESS #01 count = 1417 100 ms
DONE!! PROCESS #01 : 001417 1003 ms
root@hcpark:~/proj2# PROCESS #00 count = 0160 100 ms
PROCESS #00 count = 0321 100 ms
PROCESS #00 count = 0482 100 ms
PROCESS #00 count = 0640 100 ms
PROCESS #00 count = 0793 100 ms
PROCESS #00 count = 0955 100 ms
PROCESS #00 count = 1114 100 ms
PROCESS #00 count = 1271 100 ms
PROCESS #00 count = 1431 100 ms
PROCESS #00 count = 1588 100 ms
DONE!! PROCESS #00 : 001588 1003 ms
```

```
root@hcpark:~/proj2# ./cpu 3 1
** START: Processes = 03 Time = 01 s
        Creating Process: #1
        Creating Process: #0
PROCESS #02 count = 0117 100 ms
PROCESS #02 count = 0248 100 ms
PROCESS #02 count = 0380 100 ms
PROCESS #02 count = 0513 100 ms
PROCESS #02 count = 0674 100 ms
PROCESS #02 count = 0836 100 ms
PROCESS #02 count = 0993 100 ms
PROCESS #02 count = 1155 100 ms
PROCESS #02 count = 1317 100 ms
PROCESS #02 count = 1479 100 ms
DONE!! PROCESS #02 : 001479 1003 ms
root@hcpark:~/proj2# PROCESS #01 count = 0157 100 ms
PROCESS #01 count = 0319 100 ms
PROCESS #01 count = 0480 100 ms
PROCESS #01 count = 0639 100 ms
PROCESS #01 count = 0797 100 ms
PROCESS #01 count = 0955 100 ms
PROCESS #01 count = 1113 100 ms
PROCESS #01 count = 1271 100 ms
PROCESS #01 count = 1432 100 ms
PROCESS #01 count = 1584 100 ms
DONE!! PROCESS #01 : 001584 1002 ms
PROCESS #00 count = 0160 100 ms
PROCESS #00 count = 0322 100 ms
PROCESS #00 count = 0483 100 ms
PROCESS #00 count = 0644 100 ms
PROCESS #00 count = 0806 100 ms
PROCESS #00 count = 0966 100 ms
PROCESS #00 count = 1126 100 ms
PROCESS #00 count = 1286 100 ms
PROCESS #00 count = 1446 100 ms
PROCESS #00 count = 1605 100 ms
DONE!! PROCESS #00 : 001605 1002 ms
```

Round robin

- Time quantum = 4ms
  - 리눅스에서 기본 설정된 time quantum 은 4ms (250HZ)
  - 4ms 마다 스케줄러의 task_tick() 함수를 호출함
    - Pick_next_task()는 언제 호출이 되는가?
- Time slice는 두 가지로 설정:200ms, 500ms
  - RR에서 하나의 프로세스가 한 라운드에 수행할 수 있는 전체 시간
- 결과: 설정에 따른 변화를 확인 가능하도록 출력함
- Hint
  - sched/core.c : resched_curr() 함수

```
root@hcpark:/usr/src/linux-source-4.18.0/linux-source-4.18.0# cat .config | grep HZ
CONFIG_NO_HZ_COMMON=y
# CONFIG_HZ_PERIODIC is not set
CONFIG_NO_HZ_IDLE=y
# CONFIG_NO_HZ_FULL is not set
CONFIG_NO_HZ=y
# CONFIG_HZ_100 is not set
CONFIG_HZ_250=y
# CONFIG_HZ_300 is not set
# CONFIG_HZ_1000 is not set
CONFIG_HZ=250
CONFIG_MACHZ_WDT=m
```

```
00476: /*
00477:  * resched_curr -  mark rq's current task 'to be rescheduled now'.
00478:  *
00479:  * On UP this means the setting of the need_resched flag, on SMP it
00480:  * might also involve a cross-CPU call to trigger the scheduler on
00481:  * the target CPU.
00482:  */
00483: void resched_curr(struct rq *rq)
00484: {
00485:         struct task_struct *curr = rq->curr;
00486:         int cpu;
00487:
```

RR 핵심 코드 부분 예시

- Task_tick_fifo()
  - 매 Timer Tick 마다 수행되는 핵심 코드
  - 구현에 따라 코드는 달라질 수 있음
  - 참고하여 다른 코드 부분을 유추하고 자신의 코드를 구상해볼 것.
  - DEF_SLICE 125 = 500 ms (1 tick = 4ms)

```
#define MY_DEF_SLICE (125)
```

```
static void task_tick_fifo(struct rq *rq, struct task_struct *curr, int queued)
{
        //do not use printk() in this function !!
        //printk(KERN_DEBUG "MYMOD: task_tick_fifo CALLED\n");

        if(--curr->rt.time_slice)
                return;

        curr->rt.time_slice = MY_DEF_SLICE;

        list_del_init(&curr->rt.run_list);
        list_add_tail(&curr->rt.run_list, head);
        resched_curr(rq);
}
```

Round robin

- 200ms와 500ms의 결과
  - 이때 context switching은 각각 몇 번 일어났을까?
  - 500ms 수행 동안 5번 출력이 일어나지 않은 이유는?

```
root@hcpark:~/proj2# ./cpu 2 1
** START: Processes = 02 Time = 01 s
        Creating Process: #0
PROCESS #01 count = 0112 100 ms
PROCESS #01 count = 0243 100 ms
PROCESS #00 count = 0001 201 ms
PROCESS #00 count = 0133 100 ms
PROCESS #01 count = 0252 202 ms
PROCESS #01 count = 0414 100 ms
PROCESS #00 count = 0259 295 ms
PROCESS #00 count = 0421 100 ms
PROCESS #01 count = 0574 298 ms
PROCESS #01 count = 0737 100 ms
PROCESS #00 count = 0582 299 ms
PROCESS #00 count = 0744 100 ms
DONE!! PROCESS #00 : 000744 1097 ms
PROCESS #01 count = 0899 200 ms
DONE!! PROCESS #01 : 000899 1104 ms
```

```
root@hcpark:~/proj2# ./cpu 2 1
** START: Processes = 02 Time = 01 s
        Creating Process: #0
PROCESS #01 count = 0082 100 ms
PROCESS #01 count = 0173 100 ms
PROCESS #01 count = 0264 100 ms
PROCESS #01 count = 0355 100 ms
PROCESS #00 count = 0001 498 ms
PROCESS #00 count = 0110 100 ms
PROCESS #00 count = 0214 105 ms
PROCESS #00 count = 0324 100 ms
PROCESS #00 count = 0435 100 ms
PROCESS #01 count = 0465 600 ms
DONE!! PROCESS #01 : 000465 1003 ms
root@hcpark:~/proj2# PROCESS #00 count = 0542 100 ms
DONE!! PROCESS #00 : 000542 1005 ms
```

WRR: priority-based proportional sharing

- Priority set = {0,1,2,3,4};
  - 낮은 숫자일수록 높은 우선순위
  - 먼저 mysched_class로 변경된 순서대로 우선순위 부여할 것
- Proportional sharing
  - Time slice = {50, 40, 30, 20, 10}; //unit: quantum, thus 200 ms ~ 40 ms
    - = 10 * (5-priority)
- Round robin
  - 재스케줄링 조건 (re-scheduling)
    - Time quantum = 4ms마다 재스케줄링
    - 현재 실행중인 process가 time slice를 모두 소진했을 때
  - 현재 남아있는 time slice가 많은 순서대로 스케줄링
  - 모든 프로세스가 time slice를 소진한 경우, 새로운 round 시작

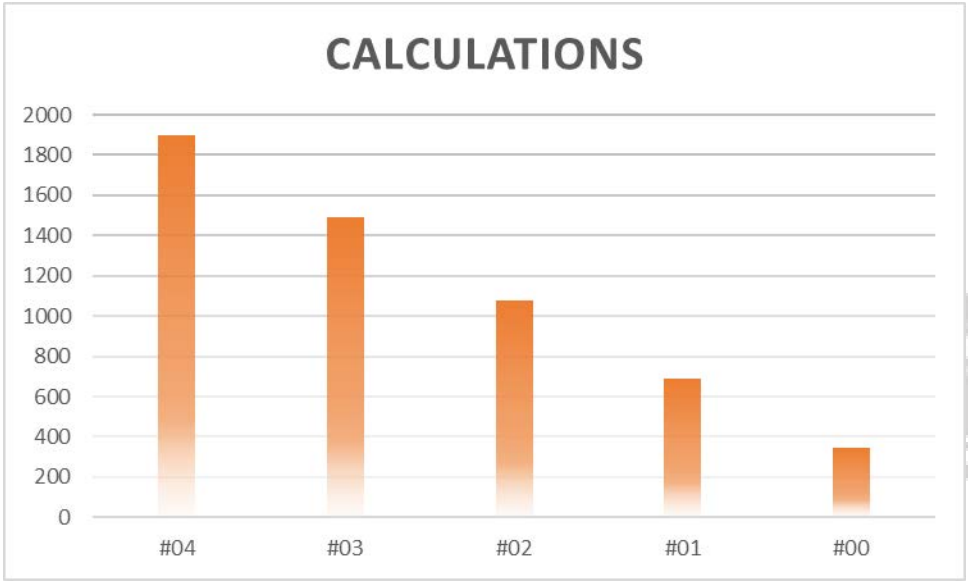
WRR: priority-based proportional sharing

```
root@hcpark:~/proj2# ./cpu 5 5
** START: Processes = 05 Time = 05 s
        Creating Process: #3
        Creating Process: #2
        Creating Process: #1
        Creating Process: #0
```

- 결과는 어떻게 보여주어야 할까?

Process	Calculations	비율
#04	1896	5.17
#03	1491	4.07
#02	1076	2.94
#01	691	1.89
#00	344	0.94
SUM	5498	15.00

```
DONE!! PROCESS #02 : 001076 5024 ms
PROCESS #01 count = 0691 271 ms
DONE!! PROCESS #01 : 000691 5028 ms
PROCESS #03 count = 1491 103 ms
DONE!! PROCESS #03 : 001491 5079 ms
PROCESS #00 count = 0344 387 ms
DONE!! PROCESS #00 : 000344 5086 ms
PROCESS #04 count = 1896 100 ms
DONE!! PROCESS #04 : 001896 5100 ms
root@hcpark:~/proj2#
```



Context switching overhead 분석

- 스케줄링 알고리즘 간의 비교 분석
  - User process들의 수행 시간은 언제나 동일하더라도
  - 스케줄링 알고리즘에 따라 전체 수행 시간이 다를 수 있음
  - 이를 통해 context switching overhead를 대략적으로 유추할 수 있음
    - 한계: 가상 머신이므로 정확한 실제 값을 알기는 어려움

- Note
  - Fair class 아래에 있다는 것에 유의할 것
    - Rt, fair class의 다른 프로세스들에 의해 선점당할 수 있음
    - 이런 경우 계산하지 못하는 context switch 가 있을 수 있음
  - printk()가 영향을 줄 수 있으므로 모두 끄고 수행

Appendix 1:

Fork()를 이용한 병렬 프로그램

Fork() 시스템 콜

- $ man fork

```
FORK(2)                          Linux Programmer's Manual                          FORK(2)

NAME
       fork - create a child process

SYNOPSIS
       #include <unistd.h>

       pid_t fork(void);

DESCRIPTION
       fork()  creates  a  new process by duplicating the calling process.  The new process is referred to as the
       child process.  The calling process is referred to as the parent process.

       The child process and the parent process run in separate memory spaces.  At the time of fork() both memory
       spaces  have  the  same  content.  Memory writes, file mappings (mmap(2)), and unmappings (munmap(2)) per-
       formed by one of the processes do not affect the other.
```

- 구글에서 example 검색해볼 것

Fork() 를 이용해 2개의 프로세스를 수행한 예

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

수행 결과 예제

- 기본 리눅스 CFS를 사용 시, 대체로 평등한 결과
- VM의 Core 개수에 따라 약간 다를 수 있음
  - 변경하여 관찰해볼 것
- $ top
  - 실시간으로 프로세스 별 자원 사용 현황을 볼 수 있음
  - 수행 후, ‘s’ 입력하고 ‘1’: 1초 마다 update
  - 수행 후, ‘1’ 입력: core 별 현황 보기

```
CPU #0 progress: count = 169000
CPU #1 progress: count = 168000
CPU #2 progress: count = 170000
CPU #0 progress: count = 170000
CPU #1 progress: count = 169000
CPU #2 progress: count = 171000
^CCPU #02 : 171111
CPU #01 : 169415
CPU #00 : 170620
root@hcpark-VirtualBox:~/proj2#
```

```
Tasks: 211 total,   4 running, 207 sleeping,   0 stopped,   0 zombie
%Cpu0  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :  0.0 us,  0.3 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  4046436 total,  2254620 free,   708888 used,  1082928 buff/cache
KiB Swap:  4192252 total,  4192252 free,        0 used.  3043540 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 6254 root      20   0    4704   1168    972 R  99.7  0.0   0:43.85 cpu
 6255 root      20   0    4704   1132    952 R  99.7  0.0   0:43.85 cpu
 6256 root      20   0    4704   1132    952 R  99.7  0.0   0:43.81 cpu
 3765 hcpark    20   0   97496   4508   3556 S   0.3  0.1   0:00.29 sshd
 5680 root      20   0       0      0      0 S   0.3  0.0   0:00.07 kworker/u8:0
    1 root      20   0  185468   6088   3988 S   0.0  0.2   0:04.37 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kthreadd
    4 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
    6 root      20   0       0      0      0 S   0.0  0.0   0:00.01 ksoftirqd/0
    7 root      20   0       0      0      0 S   0.0  0.0   0:00.21 rcu_sched
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
```

```
while(1)
{

    int i, j, k;

    for(i = 0 ; i < ROW ; i++)
    {
        for(j = 0 ; j < COL ; j++)
        {
            for(k = 0 ; k < COL ; k++)
            {
                matrixC[i][j] += matrixA[i][k] * matrixB[k][j];
            }
        }
        count++;
    }
    //count++;

    if(count%1000 == 0) {
        printf("CPU #%01d progress: count = %02d\n", cpuid, count);
    }

}
```

Appendix 2:

list 자료 구조

Intro.

- List_head 구조체
  - Double linked list 자료 구조를 구현하기 위한 구조체
  - 그 외에도 다양한 형태의 자료 구조 구현을 위해 사용될 수 있도록 유연한 구조로 디자인됨

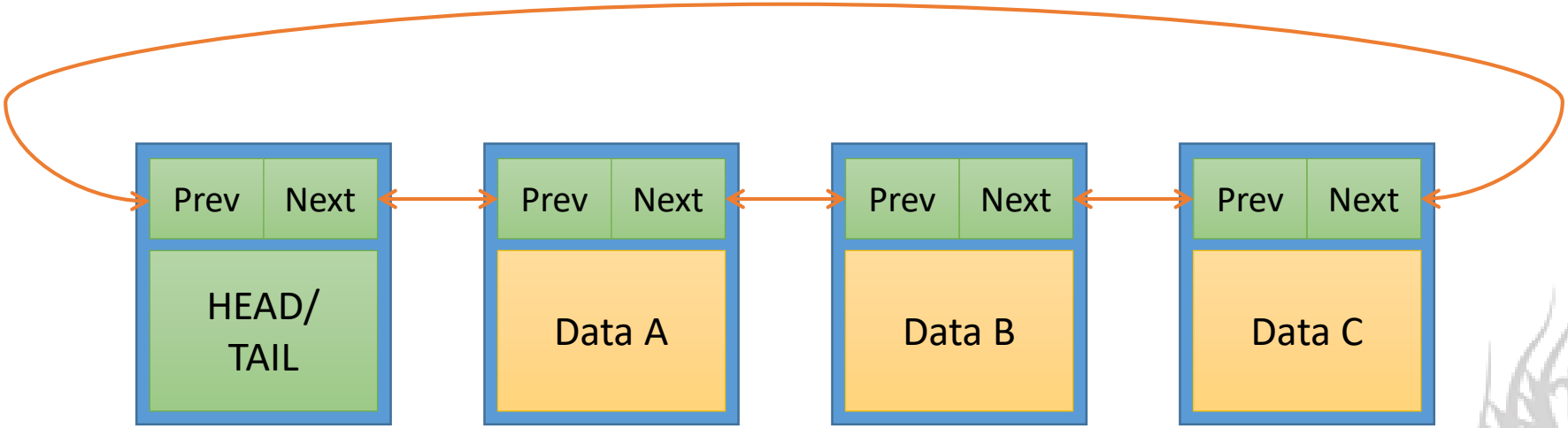
  - Definition

```
struct list_head {
        struct list_head *next, *prev;
};
```

  - 두 개의 list_head 구조체에 대한 포인터 저장

일반적인 Double Linked list 연결 방식

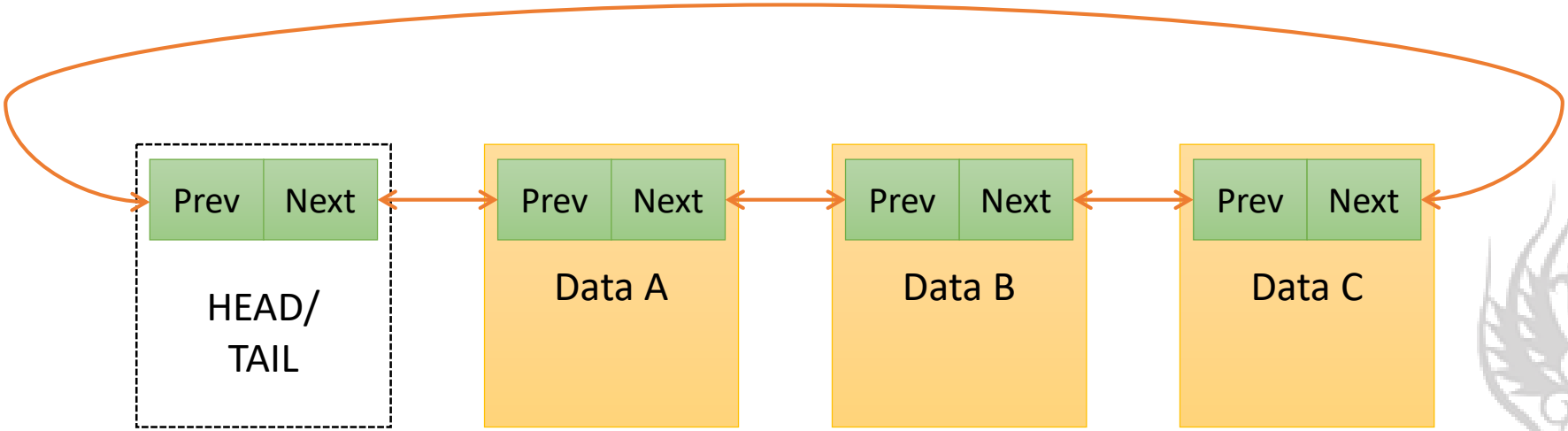
- Container 형태
  - 연결을 위한 container 구조체를 만들고,
  - 링크와 적재할 데이터를 해당 구조체에 탑재



List_head 구조체의 연결 방식

- Data structure 내부에 list_head 필드를 삽입

```
struct example {
        …
        struct list_head my_list;
        …
}
```

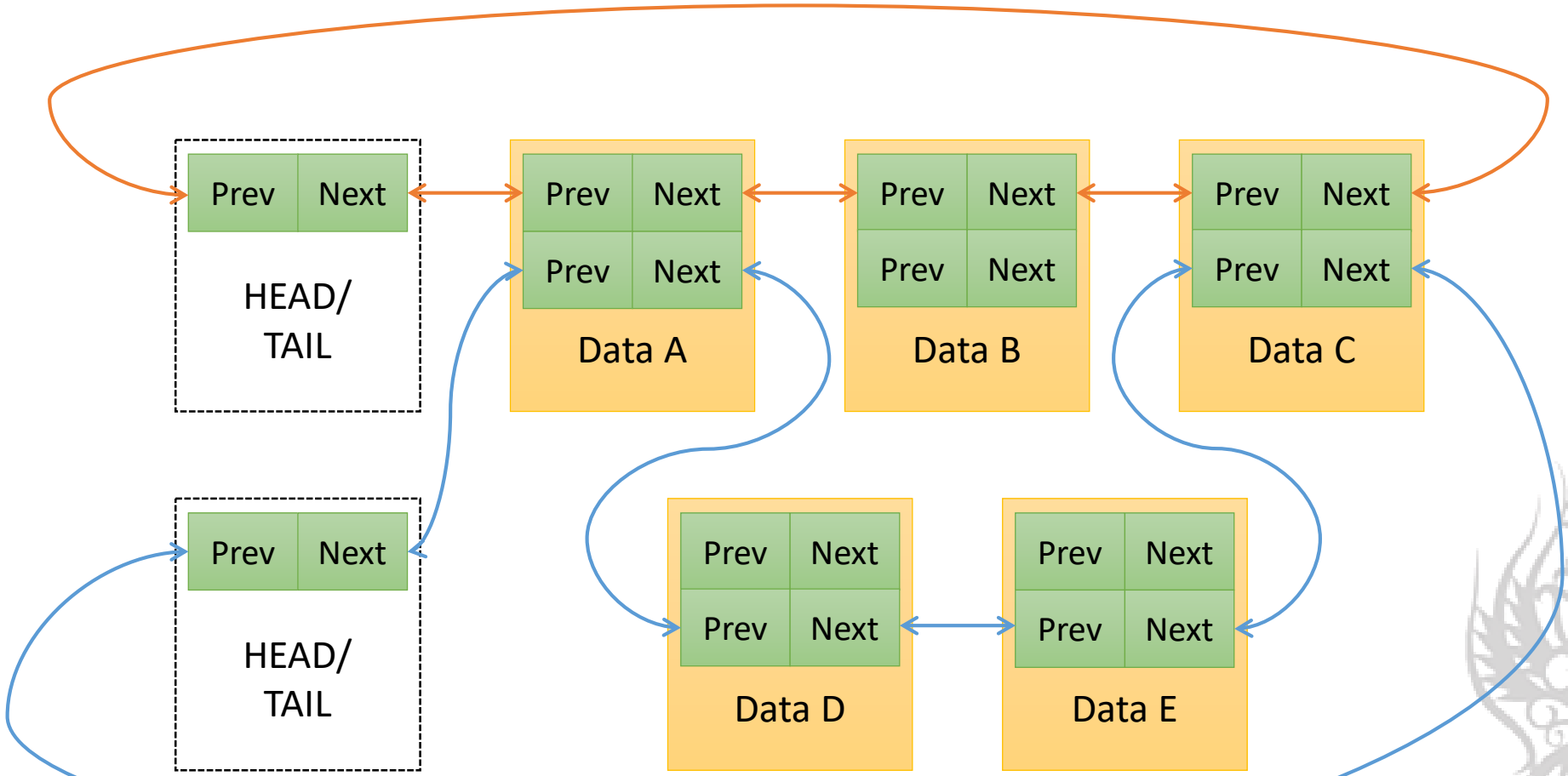


Pros. and Cons.

- Pros.
  - 간단한 형태의 구조체 하나로 다양한 자료 구조를 표현 가능
  - 커널 메모리의 절약
    - 연결을 위해 또다른 자료 구조를 할당할 필요가 없음
    - 두 개 이상의 연결을 필요로 하는 구조체를 효율적으로 구현
      - 하나의 구조체가 여러 연결 리스트에 포함되어야 하는 경우, list_head 필드를 여러 개 할당하여 구현할 수 있음
      - 다양한 컴포넌트에서 데이터를 공유하는 커널의 특성 상, 이러한 구조체가 대단히 많음
- Cons.
  - Readability의 하락
    - 직관적인 구현 방식은 아님

List_head 구조체의 연결 방식 #2

- 하나의 object를 두 개의 리스트에 포함시켜 관리할 수 있음

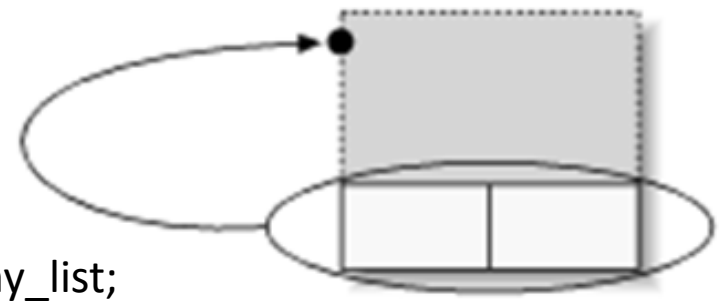


How to get “Object” from the list_head?

```
358  /**
359   * list_entry - get the struct for this entry
360   * @ptr:    the &struct list_head pointer.
361   * @type:   the type of the struct this is embedded in.
362   * @member: the name of the list_head within the struct.
363   */
364  #define list_entry(ptr, type, member) \
365      container_of(ptr, type, member)
```

- Get the “struct” for the entry
  - #define list_entry(ptr, type, member)
    - @ptr: 특정 list_head 구조체의 포인터.
    - @type: list_head ptr을 가진, 찾고자 하는 구조체의 이름
    - @member: struct type 내에서 struct list_head의 이름
  - Example for struct example

```
struct example {
        ...
        struct list_head my_list;
        ...
}
```



  - struct list_head *curr_list = curr_example->my_list;
  - struct example *next

    = list_entry(curr_list ->next, struct example, my_list);

Services: to manipulate the list

- Linux/include/linux/list.h
- Initialization
  - static inline void INIT_LIST_HEAD
    (struct list_head *list)
- Empty or not (1: empty, 0: not empty)
  - static inline int list_empty
    (const struct list_head *head)
- Add entry to list
  - static inline void list_add
    (struct list_head *new, struct list_head *head)
  - static inline void list_add_tail
    (struct list_head *new, struct list_head *head)

Services: to manipulate the list

- Delete entry from list
  - static inline void list_del
    (struct list_head *entry)
  - static inline void list_del_init
    (struct list_head *entry)
- Etc.
  - static inline void list_replace
  - static inline void list_move
  - static inline void list_splice
  - static inline void list_cut_position

Services: to search the list

- Find structure from entry
  - #define list_entry(ptr, type, member)

- List 순회
  - #define list_for_each(pos, head)

    - @pos: Loop 내부에서 사용할 struct list_head*
    - @head: 순회할 대상 List의 Head

    - (for loop을 생성하는 매크로)