1. 개요

기존 xv6의 스케쥴러는 time quantum(이하 TQ)이 1 tick인 가장 단순한 형태의 Round Robin 기반으로 구현되어 있다. 과제의 첫 번째 요구 사항은 xv6의 기존 스케쥴러를 포함하여 스케쥴링이 어떻게 동작하는지 분석하는 것이다. 두 번째는 MLFQ(Multi-Level Feedback Queue)와 유사한 형태의 SSU Scheduler 스케쥴러를 구현한다. 마지막으로, 두 스케쥴러의 성능을 비교한다. 성능의 측정 항목은 Turn Around Time과 Response Time이다. 성능 비교에서 다양한 상황을 가정하기 위해, 프로세스의 우선순위와 종료 틱을 임의로 설정할 수 있는 시스템 콜을 추가하고, 성능 비교를 위한 별도의 실행 파일을 만든다.
기존 스케쥴러 분석과 성능 비교에 관한 내용은 보고서의 가장 마지막 부분에 있다.

2. 설계

시스템 콜 추가는 user.h, usys.S, syscall.h, syscall.c, sysproc.c의 파일을 수정한다. 시스템 콜 추가에 관한 내용은 사소(trivial)하므로 생략한다.
SSU Scheduler 구현을 위해 Proc Tick, Priority 등의 필드를 proc 구조체에 추가하며, 기타 성능 측정 등을 위한 필드도 추가하였다. 스케쥴러는 proc.c에서 schedule 함수를 수정하였다. Round Robin 기반 스케쥴러를 제거하고, 실제 Queue를 기반으로 동작하는 MLFQ와 유사한 형태의 스케쥴러를 구현하였다. 또한, trap.c의 내용을 수정하여 TQ가 30 tick이 되도록 하였다.
마지막으로 성능 측정을 위한 실행 파일을 구현한다. 또한, EVAL 환경 변수가 설정되면 성능 측정을 자동화된 절차로 여러 번 수행한다.

2-1. 소스코드별 함수 개요

2-1-1. scheduler_test.c

추가한 시스템 콜 set_sched_info를 사용하여 스케쥴러의 성능을 측정하기 위한 프로그램 scheduler_test를 구현한 소스 파일

2-1-1-1. int main(int argc, char *argv[])

과제에 명시된 그대로 작성하였다.

2-1-1-2. void scheduler_func()

해당 함수는 자식 프로세스를 PNUM개만큼 생성한 후, 각각 자식 프로세스가 set_sched_info 시스템 콜을 호출하여 스케쥴러 성능 측정에 활용되도록 한다. EVAL 환경 변수가 실행하면 자동화된 절차로 여러 번 이 작업을 실시한다.

2-1-2. proc.c

추가한 시스템 콜의 동작 부분을 작성한 소스 파일

2-1-2-1. void scheduler(void)

SSU Scheduler를 구현하였다.

2-1-2-2. 큐(연결 리스트 기반) 관련 함수

- struct node *create_node()
- void free_node(struct node *node)
- void linked_list_push_back(struct linked_list *container, struct node *node)
- struct node *linked_list_shift(struct linked_list *container)

2-1-2-3. static int min_priority()

현재 스케쥴러의 큐에 있는 프로세스 중 가장 작은 우선 순위 값을 반환한다. 큐에 있는 프로세스가 없으면 0을 반환한다.

2-1-3. trap.c

TQ, 종료 틱 등과 관련된 사항을 구현하기 위해 수정한 소스 파일

2-1-3-1. void trap(struct trapframe *tf)

타이머 인터럽트(Timer Interrupt)가 발생하면, proc_tick, tq_tick(할당 받은 시간)과 종료 틱(이하 exit_tick)

등을 관리하며 각각 1씩 증가하거나 감소 시켜준다. exit_tick이 0이 되면 프로세스를 종료하고, tq_tick이 0이 되면 양보(yield)한다.

2-2. 함수 호출 그래프

2-2-1. scheduler_test

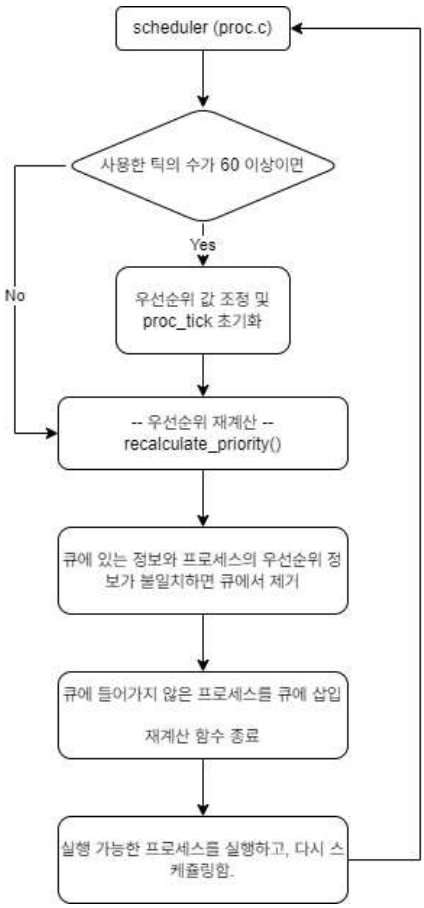
main/scheduler_func (scheduler_test.c) -> fork (system call) -> set_sched_info (system call in child process) -> (infinite loop... in child process) / killed (child process is killed by exit tick in trap.c) -> wait (parent process)

2-2-2. 스케쥴러 (xv6 내부)

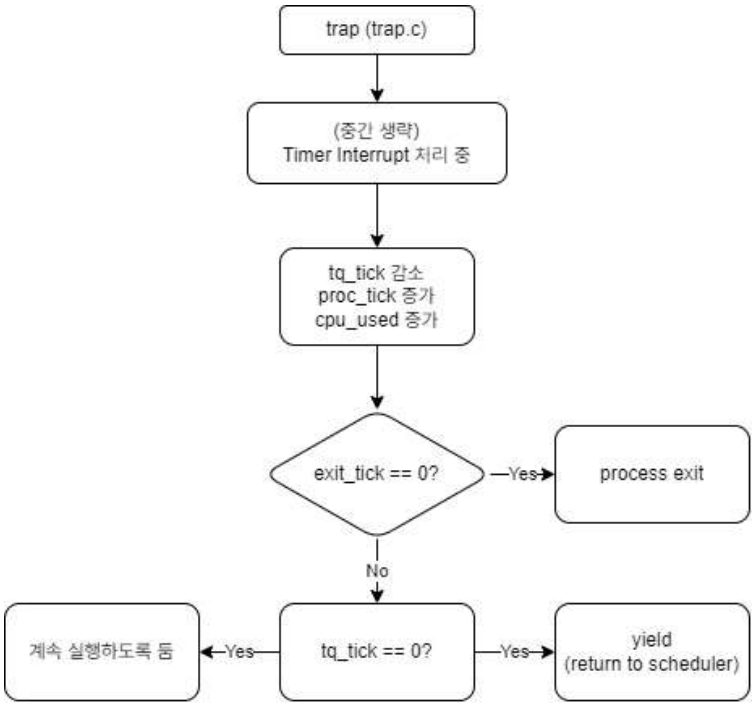
scheduler (proc.c) -> (스케쥴러에서 선택한 프로세스와 문맥 전환) -> (프로세스의 CPU 반환) -> yield, exit, sleep 함수 중 하나 (proc.c) -> sched (proc.c) -> (스케쥴러로의 문맥 전환) -> scheduler (proc.c) -> (무한 반복..)

2-3. 순서도

2-3-1. proc.c 순서도 (scheduler 처리)



2-3-2. trap.c 순서도 (schedule 관련 처리)



3. 실행결과 (scheduler_test)

3-1. DEBUG Enabled

스케쥴링에서 프로세스를 선택했을 때, 그 프로세스의 상태(우선순위, proc_tick 등..)을 출력한다. 명세에 제시된 priority_tick은 proc_tick + lazy_proc_tick을 계산하면 된다. (옛날 명세로 구현해 놨어서 그렇습니다.)
turn around tick은 arrival한 이후로 종료되기까지의 걸린 tick이며, resp tick은 set_sched_info 시스템 콜이 호출된 이후에, 다시 최초로 스케쥴링 되기까지의 걸린 tick(response tick)이다.

```
start scheduler_test
set_sched_info() pid = 4
PID : 4, priority : 1, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 30 ticks
set_sched_info() pid = 5
PID : 5, priority : 10, proc_tick : 29 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 29 ticks
set_sched_info() pid = 6
PID : 6, priority : 11, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 30 ticks
PID : 4, priority : 4, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 60 ticks
PID : 4, priority : 7, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 90 ticks
PID : 4, priority : 7, proc_tick : 20 ticks, lazy_proc_tick : 30 ticks, total_cpu_usage : 110 ticks
PID : 4 terminated (turn around tick : 169, resp tick : 0)
PID : 5, priority : 12, proc_tick : 10 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 39 ticks
PID : 6, priority : 14, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 60 ticks
PID : 6 terminated (turn around tick : 209, resp tick : 59)
PID : 5, priority : 13, proc_tick : 21 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 60 ticks
PID : 5 terminated (turn around tick : 230, resp tick : 30)
end of scheduler_test
$
```

예제 1 실행 결과

```
start scheduler_test
set_sched_info() pid = 4
PID : 4, priority : 1, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 30 ticks
set_sched_info() pid = 5
PID : 5, priority : 22, proc_tick : 27 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 27 ticks
set_sched_info() pid = 6
PID : 6, priority : 11, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 30 ticks
PID : 4, priority : 4, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 60 ticks
PID : 4, priority : 7, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 90 ticks
PID : 4, priority : 7, proc_tick : 20 ticks, lazy_proc_tick : 30 ticks, total_cpu_usage : 110 ticks
PID : 4 terminated (turn around tick : 167, resp tick : 0)
PID : 6, priority : 14, proc_tick : 10 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 40 ticks
PID : 6, priority : 15, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 70 ticks
PID : 6, priority : 15, proc_tick : 30 ticks, lazy_proc_tick : 30 ticks, total_cpu_usage : 100 ticks
PID : 6, priority : 21, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 130 ticks
PID : 6, priority : 21, proc_tick : 30 ticks, lazy_proc_tick : 30 ticks, total_cpu_usage : 160 ticks
PID : 5, priority : 24, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 57 ticks
PID : 6, priority : 27, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 190 ticks
PID : 5, priority : 27, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 87 ticks
PID : 5, priority : 27, proc_tick : 30 ticks, lazy_proc_tick : 30 ticks, total_cpu_usage : 117 ticks
PID : 6, priority : 30, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 220 ticks
PID : 6, priority : 30, proc_tick : 30 ticks, lazy_proc_tick : 30 ticks, total_cpu_usage : 250 ticks
PID : 6 terminated (turn around tick : 477, resp tick : 57)
PID : 5, priority : 33, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 147 ticks
PID : 5, priority : 33, proc_tick : 30 ticks, lazy_proc_tick : 30 ticks, total_cpu_usage : 177 ticks
PID : 5, priority : 39, proc_tick : 23 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 200 ticks
PID : 5 terminated (turn around tick : 560, resp tick : 30)
end of scheduler_test
$
```

예제 2 실행 결과

```
start scheduler_test
set_sched_info() pid = 4
PID : 4, priority : 1, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 30 ticks
set_sched_info() pid = 5
PID : 5, priority : 22, proc_tick : 29 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 29 ticks
set_sched_info() pid = 6
PID : 6, priority : 34, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 30 ticks
PID : 4, priority : 4, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 60 ticks
PID : 4, priority : 7, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 90 ticks
PID : 4, priority : 7, proc_tick : 30 ticks, lazy_proc_tick : 30 ticks, total_cpu_usage : 120 ticks
PID : 4, priority : 13, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 150 ticks
PID : 4, priority : 13, proc_tick : 30 ticks, lazy_proc_tick : 30 ticks, total_cpu_usage : 180 ticks
PID : 4, priority : 19, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 210 ticks
PID : 4, priority : 19, proc_tick : 30 ticks, lazy_proc_tick : 30 ticks, total_cpu_usage : 240 ticks
PID : 5, priority : 24, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 59 ticks
PID : 4, priority : 25, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 270 ticks
PID : 5, priority : 27, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 89 ticks
PID : 5, priority : 27, proc_tick : 30 ticks, lazy_proc_tick : 30 ticks, total_cpu_usage : 119 ticks
PID : 4, priority : 28, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 300 ticks
PID : 4 terminated (turn around tick : 449, resp tick : 0)
PID : 5, priority : 33, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 149 ticks
PID : 6, priority : 37, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 60 ticks
PID : 5, priority : 36, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 179 ticks
PID : 5, priority : 39, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 209 ticks
PID : 5, priority : 39, proc_tick : 30 ticks, lazy_proc_tick : 30 ticks, total_cpu_usage : 239 ticks
PID : 6, priority : 40, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 90 ticks
PID : 6, priority : 40, proc_tick : 30 ticks, lazy_proc_tick : 30 ticks, total_cpu_usage : 120 ticks
PID : 5, priority : 45, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 269 ticks
PID : 6, priority : 46, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 150 ticks
```

```
PID : 5, priority : 48, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 299 ticks
PID : 6, priority : 49, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 180 ticks
PID : 5, priority : 51, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 329 ticks
PID : 5, priority : 51, proc_tick : 30 ticks, lazy_proc_tick : 30 ticks, total_cpu_usage : 359 ticks
PID : 6, priority : 52, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 210 ticks
PID : 6, priority : 52, proc_tick : 30 ticks, lazy_proc_tick : 30 ticks, total_cpu_usage : 240 ticks
PID : 5, priority : 57, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 389 ticks
PID : 6, priority : 58, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 270 ticks
PID : 5, priority : 60, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 419 ticks
PID : 6, priority : 61, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 300 ticks
PID : 5, priority : 63, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 449 ticks
PID : 5, priority : 63, proc_tick : 30 ticks, lazy_proc_tick : 30 ticks, total_cpu_usage : 479 ticks
PID : 6, priority : 64, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 330 ticks
PID : 6, priority : 64, proc_tick : 30 ticks, lazy_proc_tick : 30 ticks, total_cpu_usage : 360 ticks
PID : 5, priority : 69, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 509 ticks
PID : 6, priority : 70, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 390 ticks
PID : 5, priority : 72, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 539 ticks
PID : 6, priority : 73, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 420 ticks
PID : 5, priority : 75, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 569 ticks
PID : 5, priority : 75, proc_tick : 30 ticks, lazy_proc_tick : 30 ticks, total_cpu_usage : 599 ticks
PID : 6, priority : 76, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 450 ticks
PID : 6, priority : 76, proc_tick : 30 ticks, lazy_proc_tick : 30 ticks, total_cpu_usage : 480 ticks
PID : 5, priority : 81, proc_tick : 1 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 600 ticks
PID : 5 terminated (turn around tick : 1380, resp tick : 30)
PID : 6, priority : 82, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 510 ticks
PID : 6, priority : 82, proc_tick : 29 ticks, lazy_proc_tick : 30 ticks, total_cpu_usage : 539 ticks
PID : 6, priority : 87, proc_tick : 30 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 569 ticks
PID : 6, priority : 87, proc_tick : 30 ticks, lazy_proc_tick : 30 ticks, total_cpu_usage : 599 ticks
PID : 6, priority : 93, proc_tick : 1 ticks, lazy_proc_tick : 0 ticks, total_cpu_usage : 600 ticks
PID : 6 terminated (turn around tick : 1500, resp tick : 59)
end of scheduler_test
$
```

예제 3 실행 결과

3-2. DEBUG Disabled

```
start scheduler_test
set_sched_info() pid = 4
set_sched_info() pid = 5
set_sched_info() pid = 6
PID : 4 terminated (turn around tick : 169, resp tick : 0)
PID : 6 terminated (turn around tick : 208, resp tick : 58)
PID : 5 terminated (turn around tick : 231, resp tick : 30)
end of scheduler_test
```

예제 1 실행 결과

```
start scheduler_test
set_sched_info() pid = 4
set_sched_info() pid = 5
set_sched_info() pid = 6
PID : 4 terminated (turn around tick : 169, resp tick : 0)
PID : 6 terminated (turn around tick : 479, resp tick : 59)
PID : 5 terminated (turn around tick : 560, resp tick : 30)
end of scheduler_test
$
```

예제 2 실행 결과

```
start scheduler_test
set_sched_info() pid = 4
set_sched_info() pid = 5
set_sched_info() pid = 6
PID : 4 terminated (turn around tick : 447, resp tick : 0)
PID : 5 terminated (turn around tick : 1380, resp tick : 30)
PID : 6 terminated (turn around tick : 1500, resp tick : 57)
end of scheduler_test
$
```

예제 3 실행 결과

3-3. [추가 기능] EVAL 모드

evaluation의 약자로, 이 기호가 선언되면 scheduler_test는 자동화된 절차(10회 실시)를 통해서 프로세스 그룹 여러 개를 계속 생성하여 스케쥴러의 성능을 측정한다. 통계 활용에 편하도록 엑셀에 복사-붙여넣기 가능한 형식으로 출력한다. 또한, 프로세스 종료, 시스템 콜 호출 등을 알리는 출력은 모두 비활성화된다.

```
start scheduler_test
turn around time    response time
28  0
252 58
390 28
turn around time    response time
168 58
200 28
217 0
turn around time    response time
89  0
267 30
437 51
turn around time    response time
206 30
531 34
537 0
```

프로세스 개수 = 3, 시행 횟수 = 10

4. 소스코드 (주석 포함)

4-1. scheduler_test.c

```
#include "types.h"
#include "stat.h"
#include "user.h"
#ifdef EVAL
#define TC 10
#define PNUM 3
int arr[TC][PNUM][2] =
{{{81,28},{90,168},{13,194},},{{91,45},{63,62},{10,110},},{{28,38},{55,199},{95,200},},{{96,194},{16,146},{97,197
},},{{95,22},{49,160},{80,60},},{{15,1},{42,23},{91,128},},{{79,176},{95,101},{65,160},},{{4,73},{85,43},{93,137},},
{{68,80},{76,149},{74,95},},{{39,85},{65,35},{17,61},},};
#else
#define PNUM 3
int arr[PNUM][2] = {
    {1, 300},
    {22, 600},
    {34, 600},
};
#endif
void scheduler_func() {
  printf(1, "start scheduler_test\n");
#ifdef EVAL
  for (int tc =0; tc < TC; tc++) {
    printf(1, "turn around time\tresponse time\n");
#endif
  for (int i =0; i < PNUM; i++) {
    int pid = fork();
    if (!pid) {
#ifdef EVAL
      set_sche_info(arr[tc][i][0], arr[tc][i][1]);
#else
      set_sche_info(arr[i][0], arr[i][1]);
#endif
      while (1);
    }
  }
  for (int i =0; i < PNUM; i++) wait();
#ifdef EVAL
  }
#endif
  printf(1, "end of scheduler_test\n");
}
int main() {
  scheduler_func();
  exit();
}
```

4-2. proc.c (연결 리스트 관련)

```
struct node {
  struct proc *proc;
  struct node *next, *prev;
};
struct linked_list {
  struct node *front, *back;
};
#define NULL (0)
struct node _node[NPROC *2];
int _use[NPROC *2];
struct node *create_node() {
  for (int i =0; i < NPROC *2; i++) {
    if (!_use[i]) {
      _use[i] =1;
      memset(_node + i, 0, sizeof(struct node));
      return _node + i;
    }
  }
  return 0;
}
void free_node(struct node *node) {
  _use[node - _node] =0;
  node->prev = node->next =NULL;
}
void linked_list_push_back(struct linked_list *container, struct node *node) {
  node->prev = node->next =NULL;
  if (container->front ==NULL) {
    container->front = container->back = node;
  } else {
    node->prev = container->back;
    container->back->next = node;
    container->back = node;
  }
}
struct node *linked_list_shift(struct linked_list *container) {
  if (container->front ==NULL) return NULL;
  struct node *ret = container->front;
  container->front = container->front->next;
  if (container->front ==NULL) container->back =NULL;
  else container->front->prev =NULL;
  ret->next = ret->prev =NULL;
  return ret;
}
```

4-3. proc.c (스케쥴러 관련)

```
// ptable must be held.
static int min_priority() {
  struct proc *p;
  int min_priority =100;
  for (p = ptable.proc; p <&ptable.proc[NPROC]; p++)
    if (p->state == RUNNABLE) min_priority = min(min_priority, p->priority);
  if (min_priority ==100) min_priority =0;
  return min_priority;
}

void recalculate_priority(struct linked_list queue[25]) {
  struct proc *p;
  int exist_process[NPROC] = {0};
  for (int i =0; i <25; i++) {
    struct node *curr =queue[i].front;
    struct linked_list list = {0};
    while (curr) {
      struct node *tmp = curr->next;
      p = curr->proc;
      if (p->state != RUNNABLE || i != p->priority /4) {
        free_node(curr);
      } else {
        if (exist_process[p - ptable.proc]) panic("scheduler process duplicated");
        exist_process[p - ptable.proc] =1;
        linked_list_push_back(&list, curr);
//          if (p->for_test)
//              cprintf("rechk: %d %d %d %d %d\n", p->pid, p->state, p->resp_ok, p->priority,
p->turn_arnd_tick);
      }
      curr = tmp;
    }
    queue[i].front = list.front;
    queue[i].back = list.back;
  }
  for (int i =0; i < NPROC; i++) {
    p = ptable.proc + i;
    if (!exist_process[i] && p->state == RUNNABLE) {
      struct node *node = create_node();
      node->proc = p;
      linked_list_push_back(&queue[p->priority /4], node);
    }
  }
}

void
scheduler(void) {
```

```
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc =0;
  struct linked_list queue[25] = {0};
  // Enable interrupts on this processor.
  sti();
  for (int t =0;;) {
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    // ptable.proc에 있는 process struct address는 바뀌지 않는다.
    // UNUSED에서 풀린 process 혹은 UNUSED가 된 process는 큐에서 어떻게 해야 할까?
    // 1. 큐 돌면서 프로세스 정보와 우선순위 정보가 일치하지 않거나, RUNNABLE 상태가 아니면 큐에서 제거함
    // 2. 프로세스가 있어야 할 큐에 없으면 큐에 넣음
    // 3. 최상위 우선순위 레벨의 큐에서 프로세스 하나 뽑음
    // update priority
    if (t >=60) {
      for (p = ptable.proc; p <&ptable.proc[NPROC]; p++) {
        if (p->state != RUNNABLE) continue;
        p->priority = min(99, p->priority + (p->proc_tick + p->lazy_proc_tick) /10);
        p->lazy_proc_tick = p->proc_tick =0;
      }
      t =0;
    }
    recalculate_priority(queue);
    for (int i =0; i <25; i++) {
      if (queue[i].front) {
        p =queue[i].front->proc;
        p->lazy_proc_tick += p->proc_tick;
        p->proc_tick =0;
        p->tq_tick = min(30, 60 - t);
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;
        swtch(&c->scheduler, p->context);
        switchkvm();
#ifdef DEBUG
        if (p->for_test)
          cprintf("PID : %d, priority : %d, proc_tick : %d ticks, lazy_proc_tick : %d ticks, total_cpu_usage : %d ticks\n", p->pid, p->priority, p->proc_tick, p->lazy_proc_tick, p->cpu_used);
#endif
        if (p->proc_tick && p->for_test) {
          p->resp_ok =1;
        }
        t += p->proc_tick;
        linked_list_push_back(&queue[i], linked_list_shift(&queue[i]));
        if (p->for_test) {
```

```
            for (struct proc *j = ptable.proc; j < ptable.proc + NPROC; j++)
              if (j->state == RUNNABLE || j == p) {
                j->turn_arnd_tick += p->proc_tick;
                if (!j->resp_ok) j->resp_tick += p->proc_tick;
              }
            if (p->exit_tick ==0) {
#ifdef EVAL
              cprintf("%d\t%d\n", myproc()->turn_arnd_tick, myproc()->resp_tick);
#else
              cprintf("PID : %d terminated (turn around tick : %d, resp tick : %d)\n",
myproc()->pid, myproc()->turn_arnd_tick, myproc()->resp_tick);
#endif
            }
          }
          c->proc =0;
          break;
        }
      }
      release(&ptable.lock);
    }
}
```

4-4. proc.h (수정한 부분만) - struct proc

```
int priority;
int proc_tick;
int tq_tick;
int lazy_proc_tick;
int cpu_used;

int exit_tick;
int for_test;
int resp_ok;
int resp_tick;
int turn_arnd_tick;
```

4-5. sysproc.c - 시스템 콜 구현부

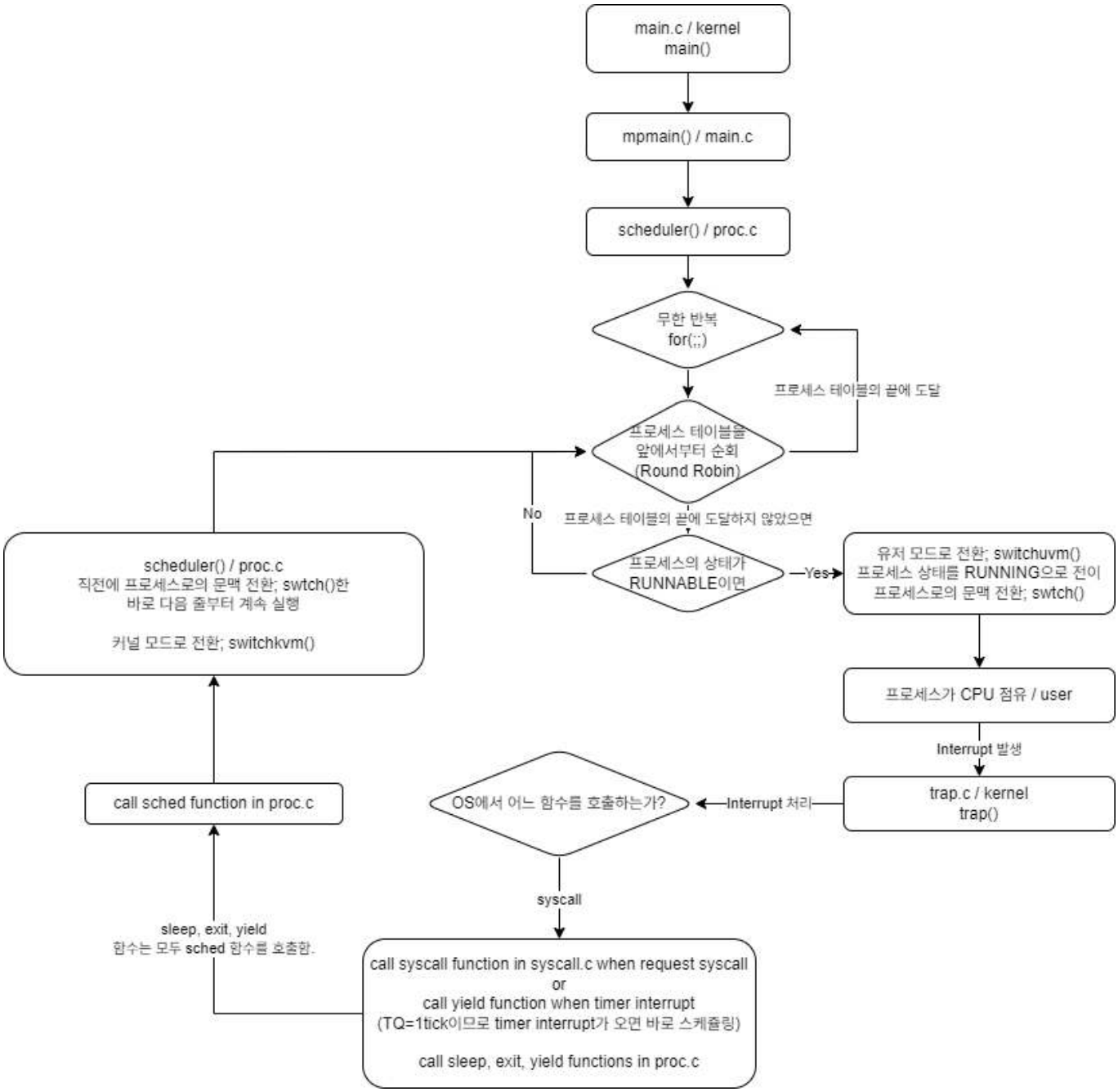
```
int
sys_set_sche_info(void)
{
  int priority, end_tick;
  if (argint(0, &priority) <0) return -1;
  if (argint(1, &end_tick) <0) return -1;
#ifndef EVAL
  cprintf("set_sched_info() pid = %d\n", myproc()->pid);
#endif
  myproc()->priority = priority;
```

```
  myproc()->exit_tick = end_tick;
  myproc()->for_test =1;
  return 0;
}
```

4-6. trap.c - trap 함수 수정 (수정한 부분만 / Timer Interrupt 부분)

```
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER) {
  myproc()->tq_tick--;
  myproc()->proc_tick++;
  myproc()->cpu_used++;
  if (--myproc()->exit_tick ==0) {
    exit();
  }
  if (myproc()->proc_tick >=30 || myproc()->tq_tick <=0) {
    yield();
  }
}
```

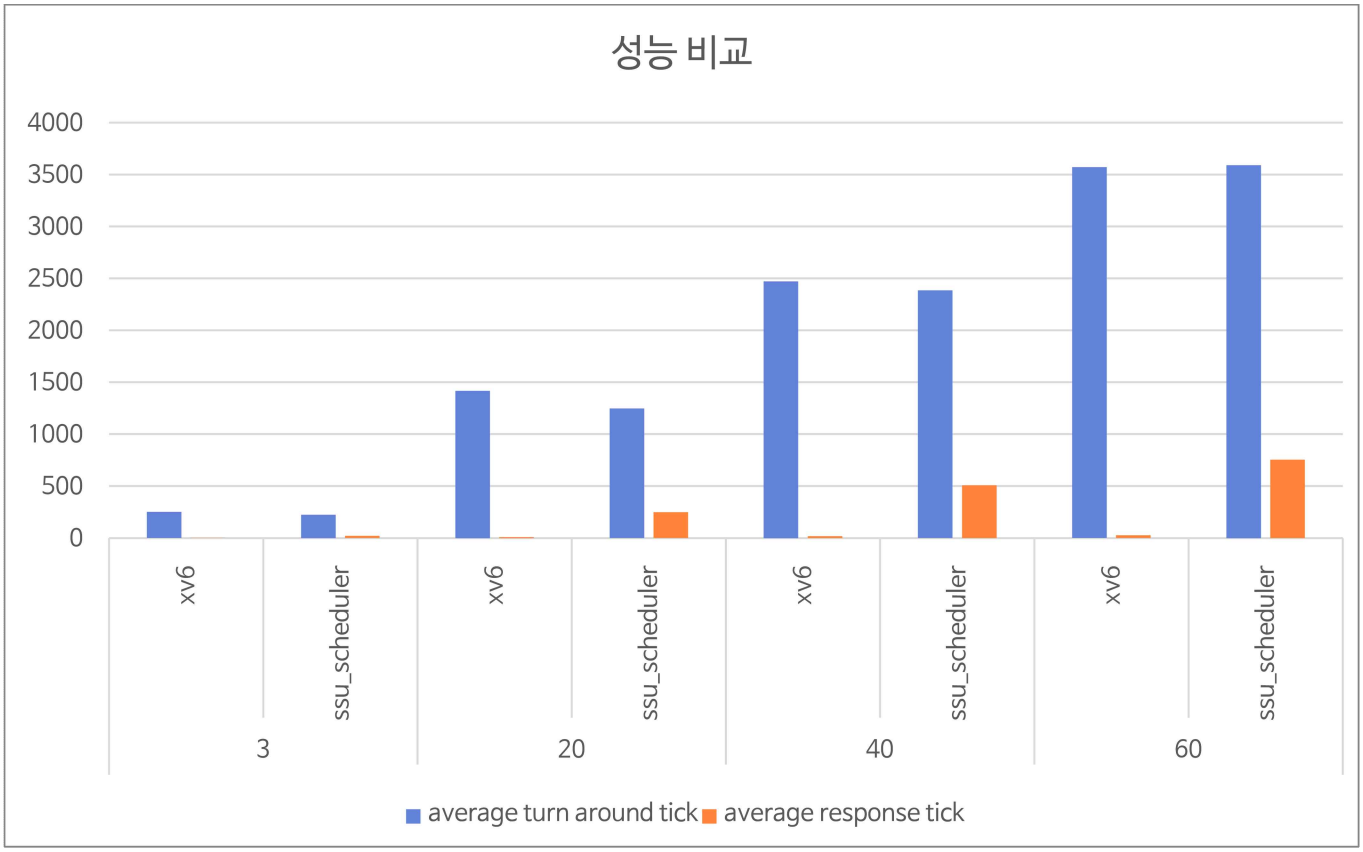
5. 기존 xv6 스케쥴러 분석



스케쥴링 과정에서 사용된 주요 함수 분석 (개요)

- scheduler() [proc.c] : 프로세스 테이블을 무한히 순회하면서 스케쥴링과 문맥 전환을 하는 함수이다.
- switchuvm(), switchkvm() : 각각 유저 모드 또는 커널 모드로 전환하는 함수이다.
- swtch() : 레지스터의 스택 정보를 바꾸어서 문맥 전환을 하는 함수이다.
- trap() [trap.c] : Interrupt가 왔을 때 처리하는 함수이다.
- sleep(), exit() [proc.c] : wait, sleep 시스템 콜이나, 프로세스의 (강제)종료 등으로 호출되는 함수이다.
- yield() [proc.c] : Timer Interrupt가 오면 다음 프로세스로 스케쥴링하기 위해서 **양보**하는 함수이다.
- sched() [proc.c], sleep, exit, yield 함수는 모두 sched 함수를 호출하며, 이 함수는 다시 스케쥴러로 문맥 전환을 한다.

6. 기존 스케쥴러와 ssu_scheduler와의 성능 및 기능 비교



기존 xv6 스케쥴러는 Round Ronbin으로 구현되어 응답 시간(response tick)이 매우 낮지만, 그에 비해 ssu_scheduler는 MLFQ로 구현되어 평균 응답 시간이 xv6에 비해 매우 높게 측정된다. 한편, 평균 around time의 경우에는 한번에 실행하는 프로세스의 개수에 따라 다른 양상을 보였다.
프로세스 개수가 3, 20, 40개에서는 기존 스케쥴러에 비해 ssu_scheduler가 조금 빠르게 동작했으나, 60개 단위로 실행할 때는 ssu_scheduler의 turn around time이 더 오래 걸렸다. 따라서, 유의미한 성능의 향상을 판단하기 위해서는 추가적인 테스트가 필요할 것으로 보인다. 하지만, 프로세스 테이블의 길이가 64개이므로 이 이상 테스트를 해보기 어렵다.

	기아 발생	응답 시간	유리한 프로세스	turn around time	스케쥴러 동작 시간 (연산량)
xv6 스케쥴러	x	낮음	완전히 공정함	상대적으로 비교하기 어려움	매우 빠름. best O(1) worst O(N)
ssu_scheduler	x	높음	짧은 프로세스가 비교적 유리함		always O(N) note that big constants