컴퓨터학부 20202904 문준혁

0. 개요

xv6 운영체제의 프로세스 스케줄링 메커니즘을 분석하고 확장하는 작업을 수행하게 됩니다. 우선, xv6에서 사용되는 기본 스케줄링 알고리즘인 단순 라운드 로빈 방식에 대해 깊이 있게 이해하고 분석한 후, 새로운 스케줄링 시스템을 구현하는 것이 목표입니다. 새로 구현할 스케줄러는 동적 우선순위 조정이 가능한 다단계 피드백 큐(Multi-level Feedback Queue, MLFQ) 스케줄러로, CPU 집중형 작업과 I/O 집중형 작업 간의 공정성을 개선하고 전체 시스템 성능을 최적화하는 것을 목적으로 합니다.

1. 기존 xv6 스케줄러 분석

- scheduler() 함수 분석

scheduler 함수는 각 CPU가 독립적으로 실행하며, 프로세스 테이블을 순회하여 RUNNABLE 상태인 프로세스를 찾고 실행하는 역할을 함. 스케줄러는 무한 루프 속에서 실행 가능한 프로세스를 찾아내며, RUNNING 상태로 전환한 후 CPU를 할당.

```
// 프로세스 테이블에 있는 프로세스를 찾고 실행하는 xv6 스케줄러의 메인 함수
void
scheduler(void)
{
  struct proc *p; // 현재 스케줄링할 프로세스를 가리킴
  struct cpu *c = mycpu(); // 현재 CPU의 구조체 포인터
  c->proc = 0;  // CPU에 현재 실행 중인 프로세스가 없음을 나타냄

  for(;;){  // 스케줄러는 무한 루프
    // Enable interrupts on this processor.
    sti();  // 인터럽트를 활성화하여 외부 이벤트를 처리 가능하게 함

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);  // 프로세스 테이블에 대한 접근을 보호하기 위해 락을 획득
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){  // 프로세스 테이블을 순회
      if(p->state != RUNNABLE)  // 현재 프로세스가 실행 가능한 상태가 아니면 건너뜀
        continue;

      // 선택된 프로세스를 실행 상태로 변경
      c->proc = p;  // CPU에 선택된 프로세스를 등록
      switchuvm(p);  // 프로세스의 메모리 공간을 사용자 모드로 전환
      p->state = RUNNING;  // 프로세스의 상태를 실행 중으로 변경

      // 컨텍스트 스위칭: CPU 상태를 스케줄러에서 프로세스로 전환
      swtch(&(c->scheduler), p->context);
      switchkvm();  // 커널 모드 메모리로 다시 전환 -> 여기서 trap()로 갔다옴 (interrupt)

      // 현재 실행된 프로세스는 종료되었거나, 다른 이유로 중단됨
      // c->proc가 다시 0으로 초기화됨 (즉, 더 이상 실행 중인 프로세스가 없음)
      c->proc = 0;
    }
    release(&ptable.lock);  // 프로세스 테이블 락을 해제하여 다른 프로세스도 접근 가능하게 함
  }
}
```

- **sched() 함수 분석**
  sched() 함수는 프로세스가 더 이상 실행되지 않고 CPU 제어권을 스케줄러로 넘겨야 할 때 호출. 이 함수는 ptable.lock을 유지한 상태에서 컨텍스트 스위칭을 수행.

```
// 스케줄러로 제어를 넘기는 함수. 프로세스는 실행을 멈추고 스케줄러가 다른 프로세스를 선택할 수 있도록 함.
void
sched(void)
{
  int intena;
  struct proc *p = myproc(); // 현재 실행 중인 프로세스를 가져옴

  if(!holding(&ptable.lock)) // ptable.lock을 유지하지 않으면 패닉
    panic("sched ptable.lock");

  if(mycpu()->ncli != 1) // 락이 적절히 관리되지 않으면 패닉
    panic("sched locks");

  if(p->state == RUNNING) // 실행 중인 프로세스가 여전히 RUNNING 상태면 패닉
    panic("sched running");

  if(readeflags()&FL_IF) // 인터럽트가 활성화된 상태에서 스케줄링이 호출되면 패닉
    panic("sched interruptible");

  intena = mycpu()->intena; // 인터럽트 상태를 저장
  swtch(&p->context, mycpu()->scheduler); // 현재 프로세스의 컨텍스트를 저장하고, 스케줄러로 전환
  mycpu()->intena = intena; // 인터럽트 상태 복구
}
```

- **yield() 함수 분석**
yield 함수는 프로세스가 스스로 CPU 사용을 포기할 때 호출. 현재 프로세스를 RUNNABLE 상태로 바꾸고 sched()를 호출하여 스케줄러로 제어를 넘김.

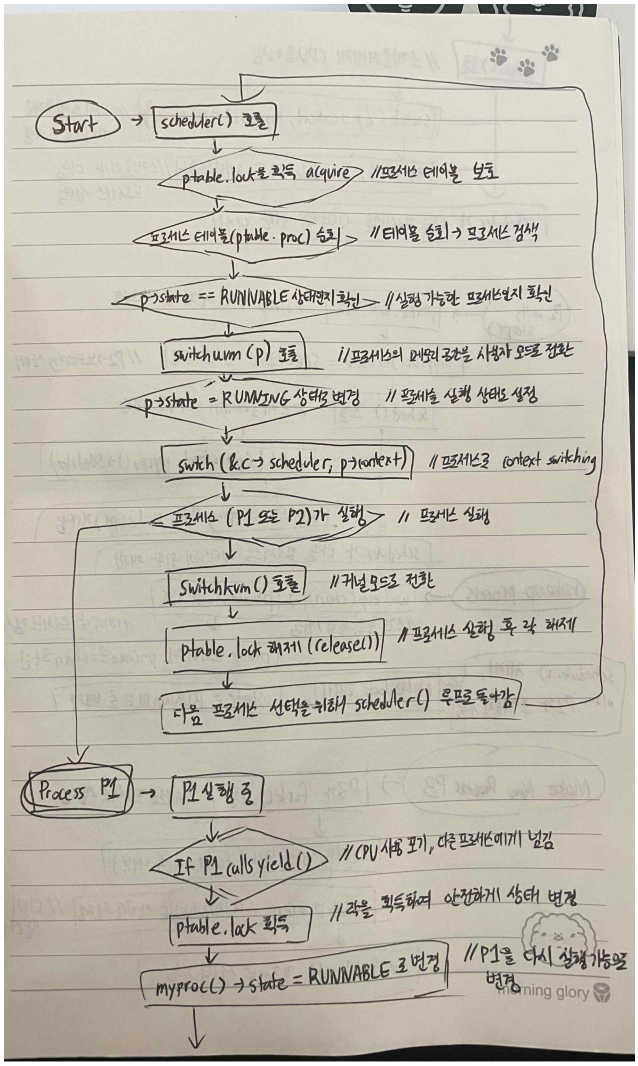
```
// Give up the CPU for one scheduling round.
// 프로세스가 한 스케줄링 라운드 동안 CPU 사용을 포기하고 다른 프로세스에게 CPU를 넘기기 위한 함수
void
yield(void)
{
  acquire(&ptable.lock);  // 프로세스 테이블에 대한 락을 획득, DOC: yieldlock
  myproc()->state = RUNNABLE;  // 현재 프로세스의 상태를 RUNNABLE로 변경
  sched();  // 스케줄러로 전환하여 다른 프로세스가 실행되도록 함
  release(&ptable.lock);  // 프로세스 테이블 락을 해제
}
```

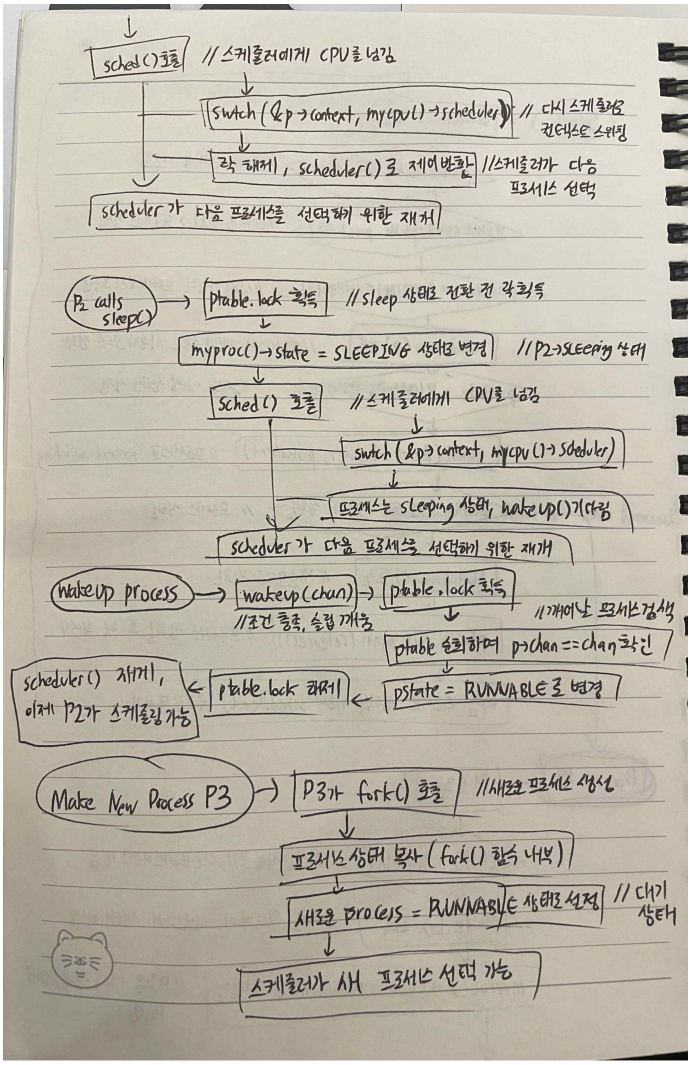
- **swtch() 함수 분석 (swtch.S)**
swtch()는 어셈블리로 구현되어 있으며, 실제로 프로세스의 컨텍스트(스택 및 레지스터)를 저장하고 복구하는 역할을 함.

- 전체적인 xv6 scheduler()

xv6 스케줄러는 CPU의 타임 쉐어링을 구현하기 위해 설계된 핵심 함수로, 컨텍스트 스위칭을 통해 커널 모드와 유저 모드 사이에서 프로세스 전환을 처리한다. 스케줄러는 무한 루프 형태로 동작하며, 실행 가능한 프로세스를 찾아 실행하는 역할을 한다. 프로세스가 타이머 인터럽트와 같은 이유로 중단되면 trap.c의 trap() 함수가 호출되고, 타이머 인터럽트가 발생하면 yield()가 실행된다. 이후 sched() 함수가 호출되어 vm.c의 swtch() 함수를 통해 현재 프로세스의 컨텍스트를 저장하고, 새롭게 실행할 프로세스의 컨텍스트로 전환한다.
이후, switchkvm() 함수로 커널 메모리 영역을 다시 적재한 후 스케줄러가 계속 동작하며, RUNNABLE 상태인 프로세스를 ptable에서 찾아 실행한다. xv6의 기본 스케줄링 방식은 1 tick마다 실행 가능한 프로세스를 선택하는 Round Robin 방식과 비슷하지만, 프로세스가 ptable에 삽입되는 순서는 고정되지 않으므로 완전히 순차적이지는 않다. 스케줄링은 프로세스가 종료하거나 대기 상태에 들어갈 때, 혹은 타이머 인터럽트가 발생할 때 이루어지며, exit(), sleep(), yield()와 같은 함수들이 sched()를 호출하여 스케줄러가 작동하게 된다.

- 기존 xv6 함수 콜 그래프 (순서도)





2. 상세설계

- 구현한 함수 프로토타입

void increment_io_wait_time(struct proc *current_proc)
- 받은 프로세스를 제외한 다른 SLEEPING하는 프로세스들의 io_wait_time 시간을 증가해주는 함수

void increment_cpu_wait_time(struct proc *current_proc)
- 받은 프로세스를 제외한 다른 RUNNABLE한 프로세스들의 cpu_wait 시간을 증가해주는 함수

void add_to_queue(int level, struct proc *p)
- 프로세스를 큐에 추가해주는 함수

void remove_from_queue(int level, struct proc *p)
- 프로세스를 큐에서 제거해주는 함수

void aging_processes(void)
- 각 프로세스를 age 시켜주는 함수

void sort_queue_by_io_wait_time(int level)
- 큐들을 io_wait_time으로 우선 sort, 만약 같으면 queue_entry_time이 늦는 순으로 sort

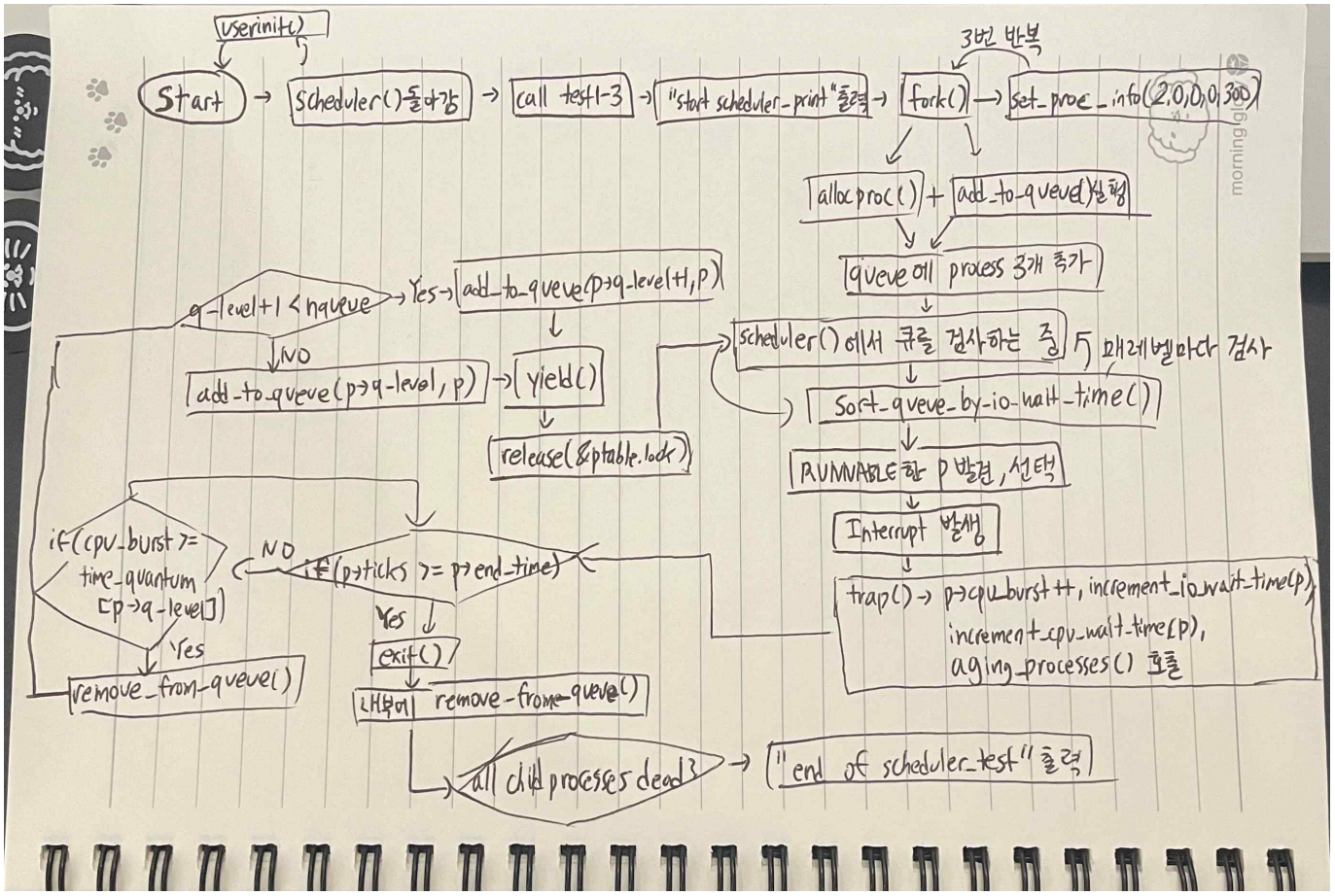
int sys_set_proc_info(void)
- 시스템 콜 함수 -> 테스트 프로그램에서 set_proc_info(q_level, cpu_burst, cpu_wait_time, io_wait_time, end_time)를 넘겨받음

- 원래 함수 수정한 내용 (가장 주요한 부분, 소스코드는 보고서 맨 뒤에)

trap.c에서 변경된 부분:
cpu_burst 증가: p->cpu_burst가 매 틱마다 증가하도록 추가
aging_processes() 호출: 프로세스들이 오래 대기할 경우 우선순위를 높여주는 에이징 함수가 호출됩니다.
타임 퀀텀에 따른 큐 이동: 각 프로세스의 cpu_burst가 큐의 타임 퀀텀을 초과하면, 해당 프로세스는 하위 큐로 이동되며, 그 과정에서 대기 시간 및 I/O 시간이 초기화됩니다.
프로세스 종료 로직: 프로세스의 total_ticks가 end_time에 도달하면 프로세스가 종료되도록 설정되었고, 이에 대한 디버그 메시지가 추가되었습니다.

scheduler() 함수에서 변경된 부분:
큐 우선순위 정렬: 각 우선순위 큐를 I/O 대기 시간을 기준으로 정렬하는 기능이 추가되었습니다.
스케줄링 로직: 기존의 xv6와 동일하게, 실행 가능한 프로세스를 선택하여 CPU에 할당한 후, switchuvm 및 swtch 함수를 통해 프로세스 간 컨텍스트 스위칭을 수행합니다. 다만, 프로세스가 실행을 완료한 후에는 해당 프로세스를 다시 초기화(c->proc = 0, p = 0)하는 코드가 포함되었습니다.

- 구현한 내용과 기 구현된 함수 간의 호출 그래프 (test1-3 예제로 만든 호출 그래프)



3. 결과

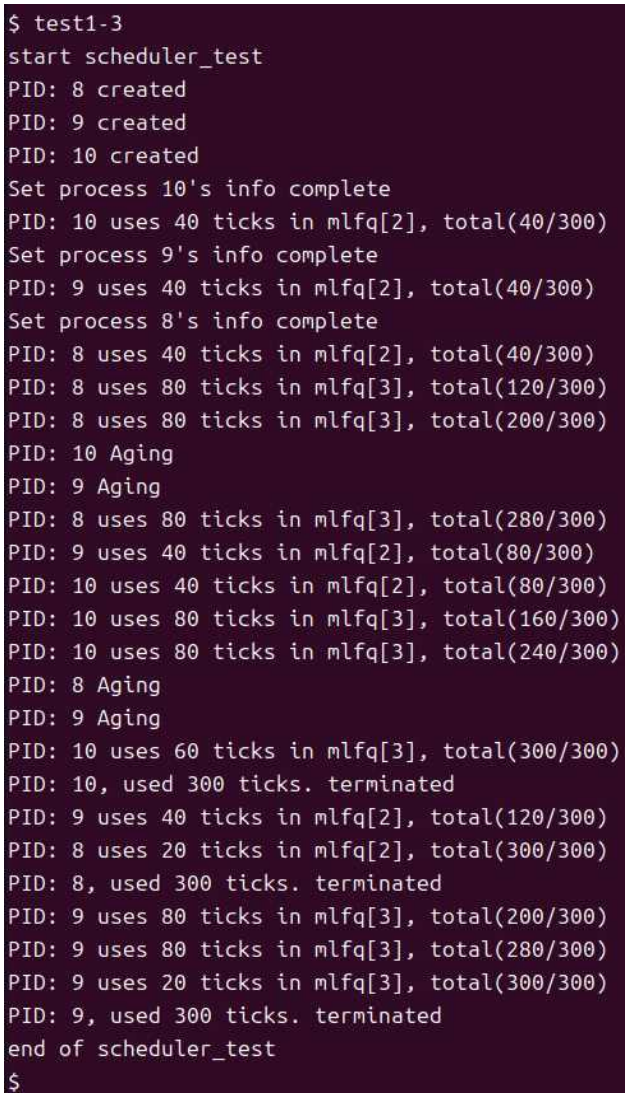
$ test1-1

```
$ test1-1
start scheduler_test
PID: 4 created
Set process 4's info complete
PID: 4 uses 10 ticks in mlfq[0], total(10/500)
PID: 4 uses 20 ticks in mlfq[1], total(30/500)
PID: 4 uses 40 ticks in mlfq[2], total(70/500)
PID: 4 uses 80 ticks in mlfq[3], total(150/500)
PID: 4 uses 80 ticks in mlfq[3], total(230/500)
PID: 4 uses 80 ticks in mlfq[3], total(310/500)
PID: 4 uses 80 ticks in mlfq[3], total(390/500)
PID: 4 uses 80 ticks in mlfq[3], total(470/500)
PID: 4 uses 30 ticks in mlfq[3], total(500/500)
PID: 4, used 500 ticks. terminated
end of scheduler_test
```

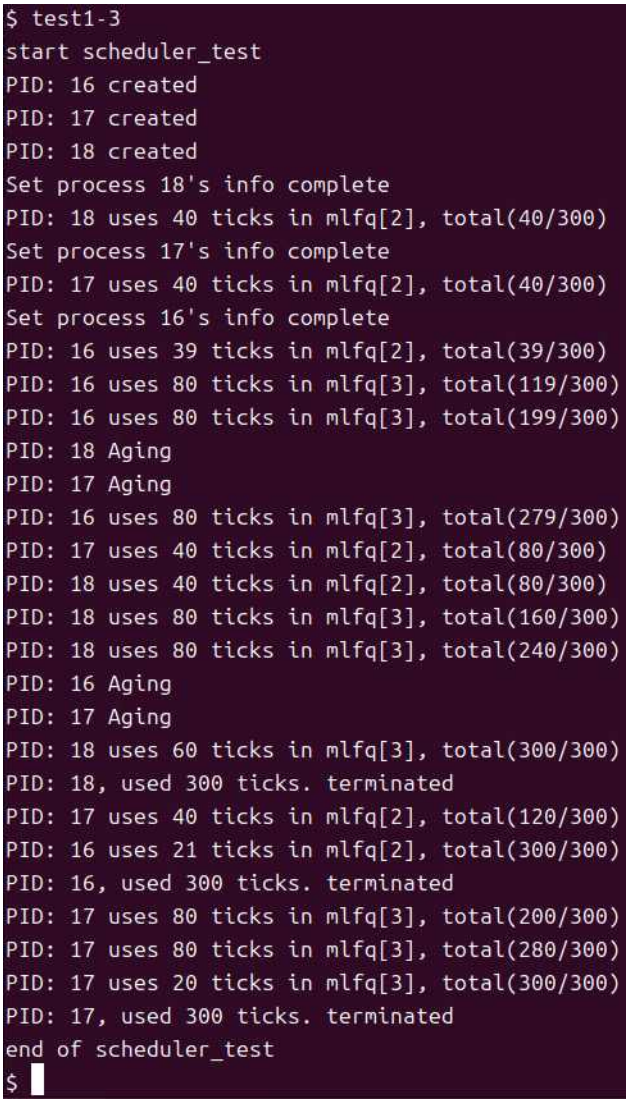
$ test1-2

```
$ test1-2
start scheduler_test
PID: 6 created
Set process 6's info complete
PID: 6 uses 20 ticks in mlfq[1], total(20/500)
PID: 6 uses 40 ticks in mlfq[2], total(60/500)
PID: 6 uses 80 ticks in mlfq[3], total(140/500)
PID: 6 uses 80 ticks in mlfq[3], total(220/500)
PID: 6 uses 80 ticks in mlfq[3], total(300/500)
PID: 6 uses 80 ticks in mlfq[3], total(380/500)
PID: 6 uses 80 ticks in mlfq[3], total(460/500)
PID: 6 uses 40 ticks in mlfq[3], total(500/500)
PID: 6, used 500 ticks. terminated
end of scheduler_test
```

$ test1-3



완전 정상



틱 1씩 정도 차이 날 때

4. 기존 xv6 스케줄러와 SSU 스케줄러의 기능 및 성능 비교 분석

기능 차이점:

- 큐 시스템:

xv6 스케줄러: 간단한 라운드 로빈 방식을 사용하여 각 실행 가능한 프로세스를 순차적으로 선택

SSU 스케줄러: **4단계의 우선순위 큐(Multi-Level Feedback Queue, MLFQ)**를 사용하여 프로세스를 관리합니다. 프로세스는 CPU 버스트 시간과 에이징 메커니즘에 따라 큐 사이를 동적으로 이동합니다. CPU를 많이 사용하는 프로세스는 낮은 우선순위 큐로 이동되고, 오래 기다린 프로세스는 상위 큐로 승격됩니다.

성능 비교

성능을 비교하기 위해 기존 xv6 스케줄러가 SSU 스케줄러의 test1-3과 비슷한 환경인 300 tick의 프로세스 3개를 주고 다 끝날때까지의 각 시작 틱, 끝 틱과, 그리고 arrival time을 출력결과를 통해 계산했습니다.

```
$ scheduler_test
Start of test_xv6_scheduler
PID 4 forked at tick 185
PID: 4 used 1 ticks.
PID: 4 used 2 ticks.
PID 5 forked at tick 188
PID: 4 used 3 ticks.
PID: 5 used 1 ticks.
PID 6 forked at tick 190
PID: 4 used 4 ticks.
PID: 5 used 2 ticks.
PID: 6 used 1 ticks.
PID: 4 used 5 ticks.
PID: 5 used 3 ticks.
PID: 6 used 2 ticks.
PID: 4 used 6 ticks.
PID: 5 used 4 ticks.
PID: 6 used 3 ticks.
PID: 4 used 7 ticks.
```

```
PID: 4 used 300 ticks.
PID: 4, used 300 ticks. terminated
PID 4 terminated at tick 1174
PID: 5 used 298 ticks.
PID: 6 used 297 ticks.
PID: 5 used 299 ticks.
PID: 6 used 298 ticks.
PID: 5 used 300 ticks.
PID: 5, used 300 ticks. terminated
PID 5 terminated at tick 1180
PID: 6 used 299 ticks.
PID: 6 used 300 ticks.
PID: 6, used 300 ticks. terminated
PID 6 terminated at tick 1183
End of test_xv6_scheduler
PID 3 terminated at tick 1186
$
```

xv6 결과 -> PID 4, 5, 6 순으로 돌아가는 것을 볼 수 있으므로 Round Robin 방식으로 작동하는 것을 볼 수 있다. PID 4, 5, 6의 Turn Around Time과 Response Time을 분석해보면

PID	Arrival Time = Fork Time	Terminated at	Start Time	Turnaround Time	Response Time
4	185	1174	185	989	0
5	188	1180	189	992	1
6	190	1183	192	993	2

그러면 평균 Turnaround Time은 991.3 ticks, 평균 Response Time은 1.5 ticks.

이제 SSU 스케줄러를 make analyze=1 qemu를 통해 test1-3을 돌려 출력한 결과는

```
$ test1-3
start scheduler_test
PID: 4 created
PID 4 forked at tick 255
PID: 5 created
PID 5 forked at tick 256
PID: 6 created
PID 6 forked at tick 257
Set process 6's info complete
PID: 6 uses 40 ticks in mlfq[2], total(40/300)
```

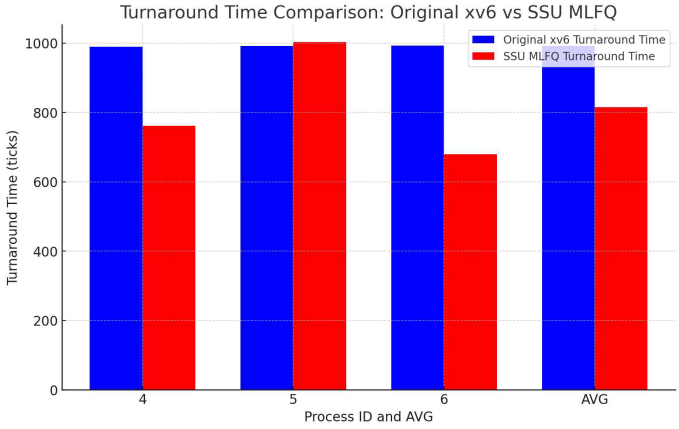
```
PID: 5 Aging
PID 6 terminated at tick 937
PID: 5 uses 39 ticks in mlfq[2], total(118/300)
PID 4 terminated at tick 1017
PID: 5 uses 80 ticks in mlfq[3], total(198/300)
PID: 5 uses 80 ticks in mlfq[3], total(278/300)
PID 5 terminated at tick 1259
end of scheduler_test
PID 3 terminated at tick 1263
$
```

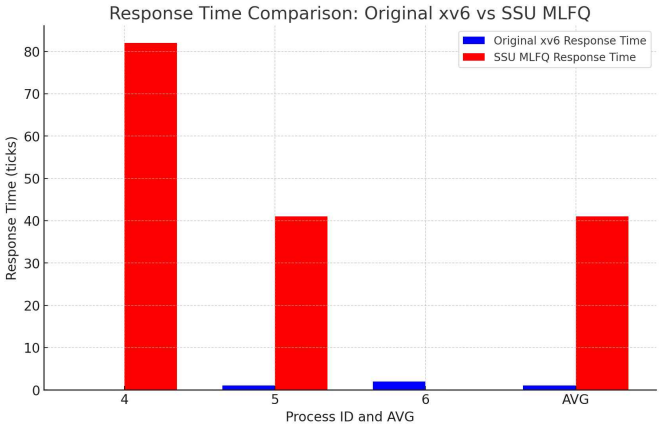
SSU 결과 -> 테이블로 만들어보면

PID	Arrival Time = Fork Time	Terminated at	Start Time	Turnaround Time	Response Time
4	255	1017	337	762	82
5	256	1259	297	1003	41
6	257	937	257	680	0

그러면 평균 Turnaround Time은 815.0 ticks, 평균 Response Time은 41.0 ticks.

각 xv6과 SSU의 Turn Around Time과 Response Time을 그래프로 만들면





요약:

일단 세 개의 자식을 fork하고 time을 300씩 줬을 때를 기준으로 분석을 해보면, SSU가 평균적인 Turn Around Time이 더 적었고, xv6가 response time 부분에서는 훨씬 적은 방면이 보인다. 이것은 SSU가 다양한 작업 부하를 더 효율적으로 처리하여 전체적인 프로세스 완료 시간(Turnaround Time)을 줄이는 데 유리함을 나타냅니다. 반면에 xv6는 응답 시간(Response Time)이 짧아 빠른 첫 반응이 필요한 시스템에서 더 적합하다는 것을 보여줍니다 (물론 현재는 xv6 쪽은 time slice을 1틱마다 돌게 해서 이를 바꾸면 다르게 작동할 수도 있음).

5. 주석 달린 소스코드 (수정한 소스코드 부분만)

proc.c

```
int time_quantum[NQUEUE] = {10, 20, 40, 80};  //각 큐의 Time Quantum
struct proc *queue[NQUEUE][NPROC];  //Queue is managed as arrays

void increment_io_wait_time(struct proc *current_proc) {
    // Loop through all queues and increment io_wait_time for sleeping processes
    for (int level = 0; level < NQUEUE; level++) {
        for (int i = 0; i < NPROC; i++) {
            struct proc *p = queue[level][i];
            if (p != 0 && p != current_proc && p->state == SLEEPING) {
                p->io_wait_time += 1;
            }
        }
    }
}

void increment_cpu_wait_time(struct proc *current_proc) {
    // Loop through all queues and increment cpu_wait for runnable processes, excluding the
current process
    for (int level = 0; level < NQUEUE; level++) {
        for (int i = 0; i < NPROC; i++) {
            struct proc *p = queue[level][i];
            if (p != 0 && p != current_proc && p->state == RUNNABLE) {
                p->cpu_wait += 1;
            }
        }
    }
}
```

```
int globalcount = 0;

//Add processes to queue
void add_to_queue(int level, struct proc *p) {
  for(int i = 0; i < NPROC; i++) {
    if(queue[level][i] == 0) {
      globalcount++;
      //cprintf("PID :%d added at level : %d\n", p->pid, level);
      queue[level][i] = p;
      p->q_level = level;
      p->queue_entry_time = globalcount;
      break;
    }
  }
}

//Remove process from queue
void remove_from_queue(int level, struct proc *p)
{
  for(int i = 0; i < NPROC; i++)
  {
    if(queue[level][i] == p)
    {
      //cprintf("PID: %d removed from level: %d\n", p->pid, level);
      queue[level][i] = 0;
      break;
    }
  }
}

static struct proc* allocproc(void) {
  ...
  // Initialize MLFQ-related fields
  p->q_level = 3;            // Start at the highest priority queue
  p->cpu_burst = 0;          // Initialize CPU burst time to 0
  p->cpu_wait = 0;           // Initialize CPU wait time to 0
  p->io_wait_time = 0;       // Initialize I/O wait time to 0
  p->end_time = -1;// Default end time -> very big number.

  return p;
}

void userinit(void) {
  ...
  add_to_queue(p->q_level, p); // Add to queue at end of user_init
}

int fork(void) {
```

```
  ...
  //If not shell process set q_level to 0
  if(pid > 2 && strncmp(np->name, "sh", 2))
  {
    np->q_level = 0;
  }

  // Add to queue
  add_to_queue(np->q_level, np);

  if(pid > 2 && strncmp(np->name, "sh", 2))
  {
    cprintf("PID: %d created\n", pid);
  }

  return pid;
}

void exit(void)
{
  ...
  // Remove process from queue that was killed
  remove_from_queue(curproc->q_level, curproc);
  ...
}

// Function to age processes in the MLFQ
void aging_processes(void) {
    int aging_threshold = 250;

    // Aging mechanism: Increase priority of processes waiting long enough
    for (int level = 1; level < NQUEUE; level++) {
        for (int i = 0; i < NPROC; i++) {
            struct proc *p = queue[level][i];
            // If process is runnable and if waiting time is bigger than aiging t.h.
            if (p && p->state == RUNNABLE && p->cpu_wait >= aging_threshold) {
                // Move the process to a higher priority queue (lower q_level)
                remove_from_queue(p->q_level, p);
                p->q_level -= 1;
                add_to_queue(p->q_level, p);
                p->cpu_wait = 0;  // Reset wait time after aging
                cprintf("PID: %d Aging\n", p->pid);
            }
        }
    }
}

// Function to sort queues by io_wait_time
// Uses bubble sort -> should change later
```

```
void sort_queue_by_io_wait_time(int level)
{
  for(int i = 0; i < NPROC - 1; i++)
  {
    for(int j = 0; j < NPROC - i - 1; j++)
    {
      struct proc *p1 = queue[level][j];
      struct proc *p2 = queue[level][j + 1];

      if(p1 != 0 && p2 != 0)
      {
        // Sort by io_wait_time (larger value has higher priority)
        if(p1->io_wait_time < p2->io_wait_time)
        {
          struct proc *temp = queue[level][j];
          queue[level][j] = queue[level][j + 1];
          queue[level][j + 1] = temp;
        }
        // If io_wait_time is the same, sort by arrival time (later arrival gets higher priority)
        else if (p1->io_wait_time == p2->io_wait_time && p1->queue_entry_time <
p2->queue_entry_time) {
          struct proc *temp = queue[level][j];
          queue[level][j] = queue[level][j + 1];
          queue[level][j + 1] = temp;
        }
        // If io_wait_time is the same, and entry time is same, later pid gets prio.
        else if (p1->io_wait_time == p2->io_wait_time && p1->queue_entry_time ==
p2->queue_entry_time && p1->pid < p2->pid) {
          struct proc *temp = queue[level][j];
          queue[level][j] = queue[level][j + 1];
          queue[level][j + 1] = temp;
        }
      }
    }
  }
}

void scheduler(void)
{  struct proc *p;
  struct cpu *c = mycpu();
  int found = 0;
  int end_ticks = 0;
  c->proc = 0;
  for(;;)
  {
    sti(); // Enable interrupts on this processor
    acquire(&ptable.lock);  // Lock process table
    found = 0; // Reset found flag
```

```
    // Check each priority queue in order
    for(int level = 0; level < NQUEUE; level++)
    {
      // Sort the queue by io_wait_time (larger first) and then by arrival time (newer first)
      sort_queue_by_io_wait_time(level);
      for(int i = 0; i < NPROC; i++)
      {
        p = queue[level][i]; // Select process from current level queue
        if(p != 0 && p->state == RUNNABLE) // Check if process is runnable
        {
          // Set process to run on the CPU
          c->proc = p;
          switchuvm(p); // Set process memory space
          p->state = RUNNING;
          p->start_ticks = end_ticks; // Set start time to end time from last process

          swtch(&(c->scheduler), p->context);  // Context switch

          switchkvm(); // Restore kernel memory space

          // Go to trap.c and come back
          c->proc = 0;
          p=0;
          found = 1; // A process was found and run
          break; // Stop searching queues
        }
      }
      if(found == 1) break;
    }
    // Set end_ticks after return from context switch
    end_ticks = ticks;
    p = 0;
    c->proc = 0;
    release(&ptable.lock);  // Unlock process table
  }
}
```

trap.c

```
void trap(struct trapframe *tf)
{
  ...
  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check nlock.
   if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
  {
      struct proc *p = myproc();
      p->cpu_burst += 1;
      increment_io_wait_time(p);
      increment_cpu_wait_time(p);
```

```
    aging_processes();

    int time_quantum[NQUEUE] = {10, 20, 40, 80};  //각 큐의 Time Quantum
    // Terminate process if it reaches end_time
    if(p->total_ticks >= p->end_time && p->end_time != -1)
    {
      int ticks_used = p->cpu_burst;
      if (p->total_ticks + ticks_used >= p->end_time) {
        ticks_used = p->end_time - p->total_ticks;
      }
      #if defined(DEBUG) || defined(ANALYZE)
      if(p->pid > 2 && strncmp(p->name, "sh", 2))
      {
        cprintf("PID: %d used %d ticks. terminated\n", p->pid, p->end_time);
      }
      #endif
      exit();
    }

    // If ticks already passed time quantum -> print accordingly
    if(ticks - p->start_ticks >= time_quantum[p->q_level])
    {
      // Reset burst and track total ticks used
      int ticks_used = p->cpu_burst;
      if (p->total_ticks + ticks_used >= p->end_time && p->end_time != -1) {
        ticks_used = p->end_time - p->total_ticks;

        #ifdef DEBUG
        if(p->pid > 2 && strncmp(p->name, "sh", 2))
        {
          cprintf("PID: %d uses %d ticks in mlfq[%d], total(%d/%d)\n", p->pid, ticks_used,
p->q_level, p->end_time, p->end_time);
          cprintf("PID: %d, used %d ticks. terminated\n", p->pid, p->end_time);
        }
        #endif
        exit();
      }
      p->total_ticks += ticks_used;

      #if defined(DEBUG) || defined(ANALYZE)
      if(p->pid > 2 && strncmp(p->name, "sh", 2))
      {
        cprintf("PID: %d uses %d ticks in mlfq[%d], total(%d/%d)\n", p->pid, ticks_used, p->q_level,
p->total_ticks, p->end_time);
      }
      #endif
      // Remove from current queue
      remove_from_queue(p->q_level, p);
```

```
        // Move to the next lower queue (unless it's already in the lowest)
        if(p->q_level + 1 < NQUEUE)
        {
          p->cpu_wait = 0;
          p->io_wait_time = 0;
          add_to_queue(p->q_level + 1, p);
        }
        else
        {
          add_to_queue(p->q_level, p);
        }

        p->cpu_burst = 0;
        yield();
      }
  }
  ...
}
```

proc.h

```
#define NQUEUE 4  //우선순위 큐 개수

void increment_io_wait_time(struct proc *current_proc);
void increment_cpu_wait_time(struct proc *current_proc);
void add_to_queue(int level, struct proc *p);
void remove_from_queue(int level, struct proc *p);
void aging_processes(void);

struct proc {
  ...
  int q_level;           // 프로세스가 속한 큐의 레벨 (0 ~ 3)
  int cpu_burst;         // 현재 큐에서 사용 중인 CPU 시간
  int cpu_wait;          // 큐에서 대기 중인 시간
  int io_wait_time;      // I/O 대기 시간
  int end_time;          // 프로세스의 총 CPU 사용 할당량

  int total_ticks;       // Total ticks used
  int queue_entry_time;  // Track when the process entered its current queue
  uint start_ticks;  // Start ticks of process

  uint entry_time; // For analyzing results vs xv6
};
```

syscall.c

```
extern int sys_set_proc_info(void);

static int (*syscalls[])(void) = {
  ...
[SYS_set_proc_info] sys_set_proc_info,
```

```
};
```

syscall.h

```
#define SYS_set_proc_info 22
```

sysproc.c

```
int sys_set_proc_info(void) {
    int q_level, cpu_burst, cpu_wait_time, io_wait_time, end_time;
    //cprintf("Set process info start\n");
    if (argint(0, &q_level) < 0 ||
        argint(1, &cpu_burst) < 0 ||
        argint(2, &cpu_wait_time) < 0 ||
        argint(3, &io_wait_time) < 0 ||
        argint(4, &end_time) < 0)
        return -1;
    struct proc *p = myproc();

    if(p)
    {
      // Remove process from its current queue
      remove_from_queue(p->q_level, p);
      // Set new process parameters
      p->q_level = q_level;
      p->end_time = end_time;  // Ensure the process has the correct end_time
      p->cpu_burst = cpu_burst;     // Init CPU burst time
      p->cpu_wait = cpu_wait_time;      // Init CPU wait time
      p->io_wait_time = io_wait_time; // Init I/O wait time
      p->total_ticks = 0;  // Init total ticks used by the process
      // Add process to the correct queue
      add_to_queue(p->q_level, p);
      cprintf("Set process %d's info complete\n", p->pid);
    }
    return 0;
}
```

test1-1.c

```
#include "types.h"
#include "stat.h"
#include "user.h"

#define ITERATIONS 10000000

int main(void) {
    int pid;
    printf(1, "start scheduler_test\n");

    if ((pid = fork()) == 0) {
        // Set process information: start in queue 0 with 500 total ticks allowed
        set_proc_info(0, 0, 0, 0, 500);
```

```
        while(1);
        exit();
    }

    wait();  // Wait for the child process to finish
    printf(1, "end of scheduler_test\n");
    exit();
}
```

test1-2.c

```
#include "types.h"
#include "stat.h"
#include "user.h"

#define ITERATIONS 10000000

int main(void) {
    int pid;
    printf(1, "start scheduler_test\n");

    if ((pid = fork()) == 0) {
        // Set process information: start in queue 1 with 500 total ticks allowed
        set_proc_info(1, 0, 0, 0, 500);

        while(1);
        exit();
    }

    wait();  // Wait for the child process to finish
    printf(1, "end of scheduler_test\n");
    exit();
}
```

test1-3.c

```
#include "types.h"
#include "stat.h"
#include "user.h"

#define ITERATIONS 10000000

int main(void) {
    printf(1, "start scheduler_test\n");

    for (int i = 0; i < 3; i++) {
        if (fork() == 0) {
            set_proc_info(2, 0, 0, 0, 300);

            while(1);
```

```
            exit();
        }
    }

    // Parent process waits for all children to finish
    for (int i = 0; i < 3; i++) {
        wait();
    }

    printf(1, "end of scheduler_test\n");
    exit();
}
```

user.h

```
int set_proc_info(int q_level, int cpu_burst, int cpu_wait_time, int io_wait_time, int end_time);
```

usys.S

```
SYSCALL(set_proc_info)
```

Makefile

```
UPROGS=\
        _test1-1\
        _test1-2\
        _test1-3\

ifeq ($(debug), 1)
CFLAGS += -DDEBUG
endif
ifeq ($(analyze), 1)
CFLAGS += -DANALYZE
endif

EXTRA=\
        ...
        test1-1.c test1-2.c test1-3.c\
        ...

CPUS := 1
```