Project 02

2018008068 김재형

목차

1️⃣ 과제 명세

기존 xv6 코드를 수정하고 성능을 개선시킨다는 점에서 과제 명세는 다음과 같습니다.

세부 명세 및 구현사항은 Implementation 에 서술하겠습니다.

1. MLFQ 스케줄러 구현

👉 L0, L1, L2, L3 Multilevel스케줄러 구현

👉 L0,L1,L2 는 FSFS(First come First served) , L3 는 Priority queue 구현

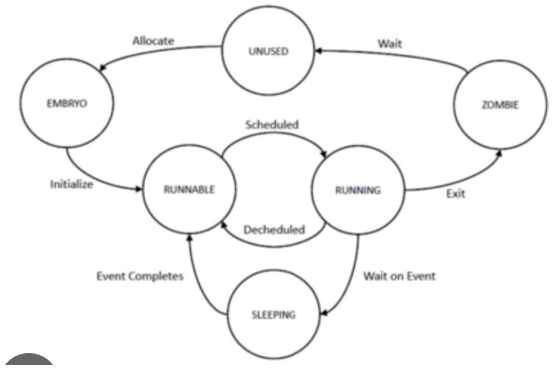
2. MoQ 스케줄러 구현

👉독점적으로 실행할수있는 FCFS(First come First served) 큐 구현

🔎 기존 Xv6 코드 분석

일단 기존 xv6 코드부터 살펴봐야 합니다. xv6가 기본적으로 사용하는 scheduler 는 Round-robin 방식으로 스케줄링을 진행합니다. 1 tick (약 10ms) 마다 timer Interrupt 가 발생하면, yield() 함수가 호출되어 현재 실행중이었던 프로세스의 상태를 RUNNING 에서 RUNNABLE 로 전환합니다. 그리고 scheduler에서 context switching 을 발생하여 다음 프로세스를 선택하는 과정을 진행합니다.

Process 의 State 변화표



Trap(Timer Interrupt) → yield → sched

● 1tick 마다 timer Interrupt 하는 부분 (trap.c)

```
// 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)
  yield();
```

● yield 함수가 호출되는 부분 (proc.c)

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

● context switching 을 하는 부분[process → scheduler] (proc.c)

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

굳이 잘동작하고 있는 Round-Robin 방식을 변경해야하는 의문이 듭니다.

다음과 같은 이유로 Round-Robin 방식을 MLFQ + MoQ 방식으로 바꿔야 한다고 생각합니다.

1. cpu bound process vs io bound process 구별

io bound process는 사용자가 입력을 안할시 할일이 없지만, cpu를 자주 조금씩 사용해야하는반면, cpu bound process는 한번 cpu를 잡으면 최대한 오래쓸려고 하는 프로세입니다.

즉, User 편의적인 운영체제를 만들기 위해서는 io bound process 를 빠르게 반응할수있도록 작성해야 합니다.

2. 프로세스별 우선순위 구별

기존 프로세스에 존재하지 않는 우선순위를 추가하여 우선순위가 높은 프로세스 먼저 끝날수있도록 프로세스 스케줄링을 할수 있습니다.

3. 독점적인 Process User 가 우선적으로 설정

때로 특정한 프로세스를 독점적으로 사용하게 스케줄링 하기위해서는 MoQ의 설계및 구현이 필요합니다.

2️⃣ Design 선택 및 구현Process 설계

✅ Design 선택

MLFQ 를 설계할시, 다음과 같은 3가지 선택지가 있으며, 장단점 및 TradeOff 를 분석해본 이후 최종적으로 ☑️ **3번[기존 Array 에 링크드 리스트 멤버변수를 추가]** 구현하였습니다.

1. 직접 큐를 설계(자료구조 추가)하여서 MLFQ 구현
2. 기존 Array 에 멤버변수를 추가하되 알고리즘을 변경하여 MLFQ 처럼 구현
3. ☑️ **기존 Array 에 링크드 리스트 멤버변수를 추가하여, MLFQ 구현**

🆚 1,2,3 번의 Trade off 및 장단점은 다음과 같습니다.

1. 직접 Queue를(자료구조 추가) 하여서 MLFQ 구현

실제로 구현하였으나, 메모리 할당 관련한 부분에서 확신이 없었으며, 자료구조 하나를 더 추가해야 하므로, 동시성 및 병렬성 (추후 CPU 가 늘어났을 경우를 대비하여) Risk 가 존재하였기에 구현한 이후 다시 RollBack 하였습니다.

구현하였던 queue.c 의 일부

```
#include "types.h"
#include "param.h"
#include "proc.h"
#include "queue.h"

// 큐의 노드를 위한 구조체
typedef struct queue_node {
  struct proc* process; // 프로세스 포인터
  struct queue_node* next;
} queue_node;

// 큐 구조체
typedef struct {
  queue_node* head;
  queue_node* tail;
} queue;

// 큐 초기화
void queue_init(queue* q) {
  q->head = q->tail = NULL;
}

// 큐에 프로세스 추가
```

```
void queue_push(queue* q, struct proc* p) {
  queue_node* new_node = (queue_node*)kalloc(); // xv6에서 메모리 할당을 위해 kalloc() 사용
  new_node->process = p;
  new_node->next = NULL;
  if (q->tail == NULL) {
    q->head = q->tail = new_node;
  } else {
    q->tail->next = new_node;
    q->tail = new_node;
  }
}

// 큐에서 프로세스 제거
struct proc* queue_pop(queue* q) {
  if (q->head == NULL) return NULL;

  queue_node* temp = q->head;
  struct proc* p = temp->process;
  q->head = q->head->next;

  if (q->head == NULL) {
    q->tail = NULL;
  }

  kfree((char*)temp); // xv6에서 메모리 해제를 위해 kfree() 사용
  return p;
}
```

Xv6에서는 메모리 할당을 위해서

kalloc(); 을사용한다고 하였으며, 이는 조사한 결과 페이지 단위 할당방식으로 알았습니다.

kalloc 에 대한 확신과 Ptable 구조체와 queue 구조체를 동시에 관리하기에 유지보수적인 측면에서 복잡해질것으로 예상하여 1번을 선택하였으나 3번으로 옮겼습니다.

하지만, 장점역시 명백히 존재하였습니다. 큐를 실제로 구현하였기에 기존 Array 의 프로세스를 찾기위한 O(N) 시간복잡도에서 O(1) 시간복잡도로 줄일수 있었습니다.

2. 기존 Array 에 멤버변수를 추가하되 알고리즘을 변경하여 MLFQ 처럼 구현

큐를 따로 구현하지 않고, 알고리즘을 통하여 중첩 for 문을 사용하여 MLFQ 처럼 작동하도록 하는 방식입니다. 위 방식은 자료구조를 따로 구현할 필요가 없어 간단하다고 생각하였으나 ,

큐 구현이 아니므로 매번 Scheduler가 특정 Level 에 있는 Process를 탐색할때마다 O(N) 의 시간복잡도가 걸리므로 **시간복잡도 측면에서 단점**이 명확하므로 해당 구현방식은 선택하지 않았습니다.

☑️ 3. 기존 Array 에 링크드 리스트 멤버변수를 추가

1번과 같이 큐를 구현하는것은 동일하나 실제 자료구조를 구현하는것이 기존 자료구조에
next 멤버변수와 , level 별 head를 추가함으로써 linked list queue를 구현하는 방식입니다.

해당 방식은 자료구조를 추가하는 overhead를 가지고있지도 않으며, 큐 레벨별로 Search 시 시간복잡도 역시 O(1) 로 구현할수 있기에 해당 3번 방식을 선택하기로 하였습니다.

✅ 구현 방향 설계

한번에 MLFQ 와 MOQ 를 설계한이후 테스트를 하였을시, 테스트의 복잡성과 어디에서 에러가 났는지 찾기 어려울수도 있으므로, 다음과 같은 Bottom - Up 설계방식을 거쳤습니다.

0. 자료구조에 필요한 변수 추가
1. MLFQ 의 L0 큐 구현 (FCFS Test)
2. L0,L1,L2,L3 큐 구현 (L3 의 Priority 큐 구현 및 테스트)
3. MoQ 구현 및 테스트

✅ 디자인 철학

다음과 같은 디자인 철학 아래에서 과제를 수행하였습니다.

1. Queue 안에는 RUNNABLE 한 State 한 Process 만을 가질수있다. (다른 State 의 Process 허용x)
2. Queue 의 Push 와 Pop 은 최대한 Scheduler 알고리즘 안에서 구현한다.

3️⃣ Implement

Implement는 다음과 같은 순서대로 설명할 예정입니다.

1. 기존 Ptable, Proc 멤버변수 추가
2. Queue 알고리즘 구현
3. Priority Queue 알고리즘 구현
4. Time Quantum 계산 알고리즘 구현
5. Boosting 알고리즘 구현
6. MoQ 알고리즘 구현
7. Main Scheduler 함수 구현
8. 기타 System Call 구현

1. 기존 Ptable, Proc 멤버변수 추가

● Ptable 멤버변수 추가 (proc.c)

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
  struct proc* heads[5];
  struct proc* tail[5];
  int numproc[5];
  int is_monopoly;
} ptable;
```

변수명	설명
struct proc* heads[5]	L0 ~ MoQ 의 Head Pointer
struct proc* tail[5];	L0 ~MoQ의 Tail Pointer
int numproc[5]	L0 ~ MoQ의 각 큐에 들어가있는 RUNNABLE STATE 의 갯수정보
int is_monopoly	현재 Monopoly(독점) 정책이 시행되고 있는지 확인하는 변수

● Proc 구조체에 멤버변수 추가 (proc.h)

```
// Per-process state
struct proc {
....
....
....
....
  int priority; // 프로세스의 우선순위
  int level;  // 프로세스의 레벨
  int passed_time; // 경과시간
  struct proc* next; // 링크드 리스트 구현으로 인한 다음 프로세스 포인터
};
```

변수명	설명
int priority	L3 큐에서 의미있는 프로세스 별 우선순위
int level	해당 프로세스가 L0,L1,L2,L3,MoQ 에 있는지 알려주는 변수
int passed_time	해당 프로세스가 Time Quantum 이 얼마나 지났는지 알려주는 변수
struct proc* next	L0~MoQ Linked List에서 다음 Pointer를 가리키는 변수

● allocproc 함수에서 변수 초기화 코드 추가 (proc.c)

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  p->next = NULL;
  p->priority = 0;
  p->passed_time = 0;
  p->level = 0;
```

allocproc 함수는 프로세스 할당하는 함수로

1. fork() : 프로세스 생성하는 시스템콜
2. userinit() : 초기 프로세스

에서 실행됩니다. 해당 함수에서 멤버변수 초기화를 설정하였습니다.

2. Queue 알고리즘 구현 (Not prioirty Queue)

● Queue Pop 구현

```
struct proc* queue_pop(int level){
  struct proc* tmp_proc = ptable.heads[level];
  ptable.heads[level] = ptable.heads[level]->next;
  if(ptable.heads[level] == NULL) ptable.tail[level] = NULL;
  ptable.numproc[level]--;
  tmp_proc->next=NULL;
  return tmp_proc;
}
```

L0, L1, L2, L3, MoQ 모두 Head Pointer를 Head→Next로 설정함으로써 Queue_pop 을 구현하였습니다. 길이가 1개일시 pop 처리등 예외처리 조건 역시 구현하였습니다.

단, queue_pop은 ptable값을 수정하는 것이므로, Ptable.lock 이 보장되어야 합니다. (Multi CPU 환경)

● Queue Push 구현

L0, L1, L2 큐의 구현은 다음과 같이 하였습니다. (Priority Queue 구현 부분은 제외하였습니다.)

```
void queue_push(struct proc* p){
  int tmp_level = p->level;
  if(tmp_level != 3){
    if(ptable.numproc[tmp_level] == 0) {
      ptable.heads[tmp_level] = ptable.tail[tmp_level] = p;
    }
    else{
      ptable.tail[tmp_level]->next = p;
      ptable.tail[tmp_level] = p;
    }
    p->next = NULL;
  }
  else if(tmp_level==3)// Priority Queue 구현부분에서 설명할예정입니다
  }
  ptable.numproc[tmp_level]++;
}
```

일반적인 Queue_push 와 같이 Tail 값에 새로운 Pointer 를 추가하는 것으로 설정하였습니다.

ptable.tail[tmp_level]→next = p; 가 주요알고리즘이며, Queue 의 길이가 0혹은 1일때의 예외처리 역시 하였습니다 위에서 언급한 Queue_push, Queue_pop 은 lock으로 제어되어야 함은 다시한번 강조드립니다.

● 기본함수에서의 Queue_push 사용사례

기존 xv6에 Queue_push 를 구현하였으니, fork(), userinit(), 그리고 Wakeup 에 Queue_push 를 추가하였습니다.

> *"RUNNABLE 상태의 Process 는 반드시 Queue 에 들어가 있어야한다."*

1. fork()에서 Queue_push 추가

기존 fork() 함수에서 np→state = RUNNABLE 로 설정한이후 L0 queue에 집어넣었습니다.

```
    np->state = RUNNABLE;
  queue_push(np);
  release(&ptable.lock);

  return pid;
```

2. Userinit() 에서 Queue_push 추가

Userinit 을 하는 과정에서 shell process를 실행시키는 User process 역시 QUeue 에 집어넣어야하므로 userinit 에 집어넣었습니다. 여기서 초기 monopoly 상태는 0이므로 독점 관련 변수 역시 초기화 하였습니다.

```
  acquire(&ptable.lock);
  ptable.is_monopoly = 0;
  p->state = RUNNABLE;
  queue_push(p);
```

```
  release(&ptable.lock);
```

3. Wakeup()에서 Queue_push 추가

디자인 철학에서 언급하였듯이, RUNNABLE STATE 가 아닌 Process (Sleeping Process)가 Wakeup 될시 다시 큐에 들어와야 합니다. 이때를 위해서 Wakeup 함수에 추가하였습니다.

```
static void
wakeup1(void *chan)
{
  struct proc *p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan){
      p->state = RUNNABLE;
      queue_push(p);
    }
}
```

● 기본함수에서의 Queue_pop 사용사례

Scheduler 에서 Pop을 사용하였으므로 이후 Scheduler 구현 Implementation 부분에서 자세히 설명할 예정입니다.

3. Priority Queue 알고리즘 구현

● Queue_push 구현 [proc.c]

```
  if(tmp_level != 3){
        //일발 큐에서의 push 과정
  }
  else if(tmp_level==3){
    if (ptable.numproc[tmp_level] == 0) {
      ptable.heads[tmp_level] = ptable.tail[tmp_level] = p;
      p->next = NULL; //이거 안해서 에러난듯 ㅇㅇ
    }
    else{
      struct proc* curr = ptable.heads[tmp_level];
      struct proc* prev = NULL;

      while (curr != NULL && curr->priority >= p->priority){
        prev = curr;
        curr = curr->next;
      }

      if(prev == NULL){
        // 리스트 맨앞에 삽입
        p->next = ptable.heads[tmp_level];
        ptable.heads[tmp_level] = p;
      }
      else{
        // 중간 또는 끝에 삽입
        // cprintf("여기임 리스트 중간또는 끝에 삽입 %d\n", p->pid);
        p->next = curr;
        prev->next = p;
```

```
        // 끝에 삽입할때,
        if(curr == NULL){
          ptable.tail[tmp_level] = p;
        }
      }

    }
  }
```

Prioiry Queue는 proc→priority 가 큰순으로 정렬되어야 하므로, 해당 알고리즘을 직접 구현하였습니다. 길이가 0일시 삽입하는 과정은 기존과 동일하지만, 길이가 1이상일시에는

prev , curr 변수를 선언한 이후에 process 가 들어갈 자리를 찾은 이후에 연결하는 알고리즘을 따로 구현하였습니다.

1. 길이가 0일시 삽입
2. 길이가 1이상일시 삽입
   - 링크드 리스트 맨앞에 삽입
   - 링크드 리스트 중간또는 끝에 삽입
   - 링크드 리스트 끝에 삽입할시 ptable.tail[3] 설정

과 같은 구조로 알고리즘 구현을 하였습니다.

● setprioirty구현 [proc.c]

```
int
setpriority(int pid, int priority)
{
  if(priority<0 || priority>10) return -2;
  int find = -1;

  acquire(&ptable.lock);
  for(struct proc* p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->priority = priority;
      find = 0;
      break;
    }
  }

  struct proc* prev = NULL;
  for(struct proc* p=ptable.heads[3] ; p!= NULL ; p=p->next){
    if(p->pid == pid){
      if (prev == NULL) queue_pop(3);
      else{
        if(p==ptable.tail[3]){
          ptable.tail[3] = prev;
          prev->next = NULL;
        }
        else prev->next = p->next;
      }
      p->next = NULL;
      queue_push(p);
    }
```

```
    prev = p;
  }

  release(&ptable.lock);
  return find;
}
```

해당 함수는 특정 pid의 prioirty 를 설정하는 함수입니다.

Ptable lock 이 걸린상태에서 실행되어야하며, L0,L1,L2,MoQ 에서는 Prioirty 가 의미가 없기에 순서 변함이 없으나 **L3큐에서 prioirty 가 변동이있을시 해당 process의 위치가 변함**으로 인하여 ,기존 process를 삭제한이후 다시 집어넣는 과정을 반복하였습니다.

4. Time Quantum 알고리즘 구현

기존에는 앞서 설명하였듯이,

● 해당 프로세스가 RUNNING 상태고

● TImer Interrupt 가 동작할시

yield 프로세스를 호출하여 스케줄러로 양보하였지만, 이제는 큐 레벨별로 다르게 구현하여야 하므로 다음과같이 코드를 작성하였습니다.

● degrade 구현 [proc.c 에서 구현]

```
int time_quantum[4] = {2,4,6,8};
int degrade(){
  int ret = 0;
  acquire(&ptable.lock);
  if(ptable.is_monopoly){
    release(&ptable.lock);
    return 0;
  }

  struct proc *p = myproc();
  p->passed_time++;
  if(p->passed_time == time_quantum[p->level]){
    ret = 1;
    if(p->level == 0){
      if(p->pid % 2) p->level = 1;
      else p->level =2;
    }
    else if(p->level == 1 || p->level ==2) p->level = 3;
    else if(p->level == 3){
      if(p->priority > 0) p->priority--;
    }
    p->passed_time = 0;
  }

  release(&ptable.lock);
  return ret;
}
```

우선적으로, 독점상태일때는 degrade 가 일어나면 안되므로 선검사를 진행하였으며,

ptable의 무결성이 보장되어야 하므로 lock 설정을 하였습니다.

프로세스가 RUNNING 상태일때이면서 ,timer interrupt 설정될시 해당 process의 Passed_time 을 1씩 증가시켰습니다.
time_quantum 배열로 검사를 비교하였으며

만일, 큐 레벨이 증가하였을시 passed_time 은 초기화하였으며 ,level 역시 변화시켰습니다.

● L0 큐에서 passed_time=2일시 , pid 의 홀짝여부에 따라 L1,L2 큐로 이동

● L1,L2 큐에서 passed_time = 4,6 일시 L3로 공통적으로 이동

● L3 큐에서는 passed_time = 8 일시 L3 ptable[3].tail→next pointer 참조하여 이동

● degrade 사용사례[trap.c]

```
  if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0 + IRQ_TIMER){
    int result = degrade();
    if(result) yield();
  }
```

만약 degrade 의 반환값이 1이라면, process의 상태가 time_quantum을 모두 채웠다는것이므로 , 스케줄러에게 양보하게 되며,
만일 time_quantum을 사용하지 않았다면 그냥 pass 하는 방식으로 사용하였습니다.

5. Boosting 알고리즘 구현

Boosting 알고리즘은 다음과같습니다.

● time tick % 100 == 0 일때마다 모든 Process 의 level =0, passed_time = 0 으로 초기화합니다.

L3 큐에서 Boosting 알고리즘을 구현하는 이유는 아래와 같이 서술하였습니다..

기존 Priority Queue 는 우선순위를 기준으로 스케줄링하기에 Prioirty 가 낮은 Process 일경우

큐에서 스케줄링이 되지않은 “Starvation” 문제가 생겨날수 있습니다. 이를 방지하기 위해서

Boosting 알고리즘을 과제 명세대로 구현하였습니다.

● boost 구현 [proc.c 에서 구현]

```
void boost(){
  struct proc* tmp_proc;
  acquire(&ptable.lock);
  if(ptable.is_monopoly==1){
    release(&ptable.lock);
    return;
  }
  for(struct proc* p= ptable.proc ; p < &ptable.proc[NPROC]; p++){
    if(p->level != 4 && p != NULL){
      p->level = 0;
      p->passed_time = 0;
    }
  }

  while(ptable.numproc[1]){
    tmp_proc = queue_pop(1);
    queue_push(tmp_proc);
  }

  while(ptable.numproc[2]){
    tmp_proc = queue_pop(2);
    queue_push(tmp_proc);
```

```
  }

  while(ptable.numproc[3]){
    tmp_proc = queue_pop(3);
    queue_push(tmp_proc);
  }
  release(&ptable.lock);
}
```

만일 monopoly 정책으로 인하여 Monopoly queue 로 이동하였을경우, boosting 의 영향을 받지 않게 하였으며, 현재 monopoly 정책이 시행중일경우 boosting 이 시행되지 않도록 구현하였습니다.

구현 자체는 간단합니다.

모든 ptable을 돌면서 level ≠ 4 일시 level 과 passed_time 을 모두 0으로 초기화 하였습니다.

● boost 사용사례 [proc.c 에서 구현]

```
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      if(!(ticks % 100)) boost();
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
```

timer interrrupt 가 실행될시 tick 값이 100의 배수일때마다 ,boosting 알고리즘 을 구현하였습니다.

6. MoQ 알고리즘 구현

MoQ 알고리즘은 기존 L0,L1,L2 와 같이 FCFS(First Come First Served) 와 같으므로 다를바가없지만,

시스템 콜로 MLFQ에서 정책 MoQ로 바꾸면서 시행하게 되므로 MoQ 관련 시스템 콜 구현과 관련하여 설명하겠습니다.

● monopolize 구현 [proc.c 에서 구현]

```
void monopolize(void){
  acquire(&ptable.lock);
  ptable.is_monopoly = 1;
  release(&ptable.lock);
}
```

monopolize 함수는 기존 MLFQ 정책에서 독점적으로 해당 큐를 사용하겠다는 정책설정입니다.

MoQ 진입을 위한 필수적인 시스템 콜입니다.

● unmonopolize 구현 [proc.c 에서 구현]

```
void unmonopolize(void){
  acquire(&ptable.lock);
  acquire(&tickslock);
  ptable.is_monopoly = 0;
  ticks = 0;
  release(&tickslock);
  release(&ptable.lock);
}
```

Unmonopolize 는 MoQ의 스케줄러 실행이 완료되었으면, 자동적으로 MLFQ 정책으로 변환되도록 실행을 시키는 시스템 콜입니다. MoQ 정책에서 빠져나오기 위한 필수적인 시스템 콜입니다.

● setmonopoly 구현 [proc.c 에서 구현]

```
int setmonopoly(int pid, int password){
  struct proc* now_proc = myproc();
  int find = -1;
  if(password !=2018008068) return -2;
  acquire(&ptable.lock);
  int before_level;
  if(now_proc->pid == pid){
    now_proc->level = 4;
    find = -4;
  }
  else{
    for(struct proc* p = ptable.proc; p < &ptable.proc[NPROC]; p++){

      if(p->pid == pid){
        before_level = p->level;
        if(p->level ==4) find = -3;
        else{
          p->level = 4;
          queue_push(p);
          find = ptable.numproc[4];
          if(p->state == RUNNABLE) pop_specific_pid(before_level, pid);
        }
        break;
      }
    }
  }
```

setmonopoly 함수는 특정 pid 를 MoQ로 이동시키는 시스템콜입니다.

다음과 같은 알고리즘 순서로 구현하였습니다.

1. 저의학번 (2018008068) 과 일치하지 않으면 -2를 return 합니다.
2. 만일 지금 실행중인 Process 를 MoQ로 이동하고자 한다면 -4를 return 합니다.
3. 만약 해당 Pid 를 찾지 못한다면 -1 을 Return 합니다.
4. 해당 Pid를 찾았을시 해당 Proess 의 레벨이 속해있는 큐에들어가서 queue_pop 을진행합니다.

● pop_specific_pid 구현 [proc.c 에서 구현]

```
void pop_specific_pid(int lev,int pid){
  struct proc* current = ptable.heads[lev];
  struct proc* previous = NULL;

  while (current != NULL && current->pid != pid) {
      previous = current;
      current = current->next;
  }

  // 만일 길이가 1일시..
  if (previous == NULL) {
      ptable.heads[lev] = current->next;
      if (ptable.heads[lev] == NULL) ptable.tail[lev] = NULL;
  } else {
      previous->next = current->next;
      if (current->next == NULL) ptable.tail[lev] = previous;
  }

  // 프로세스 수 감소
  current->next = NULL;
  ptable.numproc[lev]--;
}
```

특정 큐에 들어가서 pid 와 동일한 process를 찾았을시 해당 queue 를 pop 하는 함수입니다.

setmonopoly를 위하여 설정하였습니다.

● MoQ 알고리즘 적용사례

1. MoQ 알고리즘 적용시 boost, degrade 는 일어나지 않도록 설정하였습니다.
2. Sleep 상태에서 기존 xv6는 yield는 스케줄러에게 양보를 하였지만, MoQ 정책도중에서는 양보하지 않았습니다.

7. Scheduler 알고리즘 구현

Scheduler 의 알고리즘 명세는 다음과같습니다.

0. Scheduler Queue 안에는 Runnable 한 Process 만 들어갈수 있다는 철학위에서 실행됩니다.
1. Monopoly 정책이 시행되는지 우선적으로 확인합니다.
2. 각 Process 가 실행이 완료되었을시, L0 에 새로운 프로세스가 들어왔을수 있으므로 ,확인을 하며, 만일 들어왔다면 L0 부터 다시 실행을 합니다.

● scheduler 알고리즘 구현 코드[proc.c 구현]

```
void change_context(struct proc* p1,struct cpu* c1){
  c1->proc = p1;
  switchuvm(p1);
  p1->state = RUNNING;
  swtch(&(c1->scheduler), p1->context);
  switchkvm();
  c1->proc = 0;
}
```

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;
  for(;;){
    sti();
    acquire(&ptable.lock);
queue_search_start:
    if(!ptable.is_monopoly){
      for(int i =0 ; i<=3 ; i++){
        while(ptable.numproc[i]){
          p = queue_pop(i);
          change_context(p, c);
          if(p->state == RUNNABLE) queue_push(p);
          goto queue_search_start;
        }
      }
    }
    else{
      while(1){
          p = queue_pop(4);
          change_context(p, c);
        }
        if(ptable.numproc[4] ==0) ptable.is_monopoly = 0;
        goto queue_search_start;
      }
    }
    release(&ptable.lock);
  }
}
```

주요 알고리즘 코드는 다음과 같습니다.

```
          p = queue_pop(i);
          change_context(p, c);
          if(p->state == RUNNABLE) queue_push(p);
          goto queue_search_start;
```

큐에서 queue_pop으로 프로세스 pointer를 가져온 이후,

프로세스를 실행합니다. 프로세스는 실행도중 완료될수 있고 (ZOMBIE State 로 변환), Sleeping 상태로 변할수도 있습니다. (SLEEPING STATE) , 또한 interrupt 로 인한 다른 큐로 들어갈수도 있습니다(레벨이 변할수도 있습니다.)

즉, 이는 모두 queue_push 로 구현하였으며, 이는 앞서 설명드린, **boost, degrade를 모두 반영한 결과**입니다. 추가로 queue_search_start 로 인하여, 만일 L3 큐를 실행하여도 실행도중, L0 큐에 프로세스 들어왔을시 L0부터 실행할수있도록 스케줄링 알고리즘을 구현합니다.

8. 기타 System Call 구현

앞서 언급하지않은 시스템 콜은 다음과같이 구현하였습니다.

● sys_yield[sys_proc.c]

```
int
sys_yield(void)
{
  yield();
  return 0;
}
```

기존에 시스템콜로는 yield 가 구현되어있지 않았기에 wrapper function을 구현하였습니다.

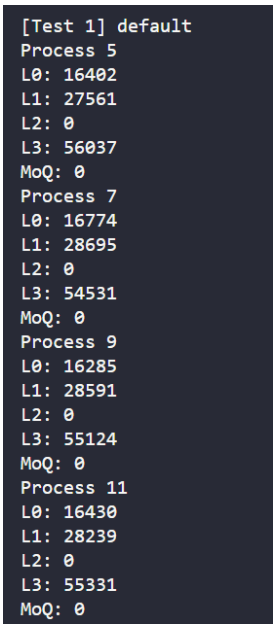
● getlev 구현[proc.c]

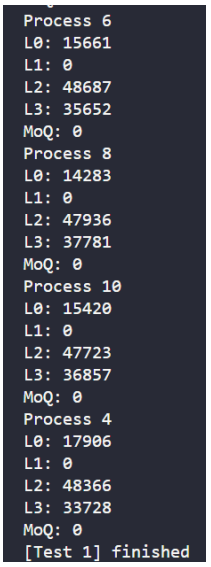
```
int
getlev(void)
{
  struct proc *p = myproc();
  if(p->level == 4) return 99;
  else return p->level;
}
```

Process의 level 을 반환합니다. 하지만 MoQ의 경우 99를 반환합니다. (mlfq_test를 위해)

4️⃣ Result

● Test 1 (MLFQ Test)

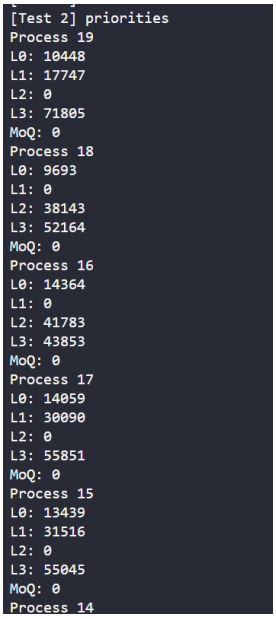


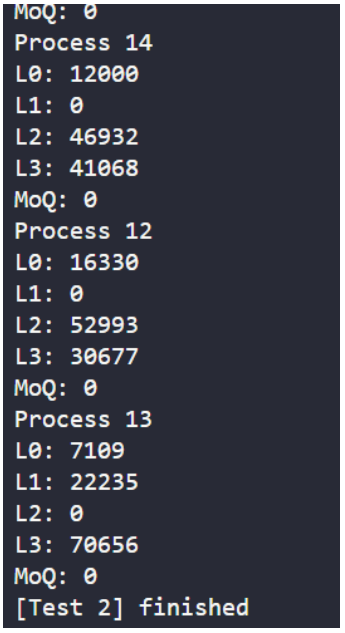


MLFQ 스케줄링 알고리즘 경우 L1 큐를 L2 큐보다 먼저 보도록 설계되어있으며,

L1 큐의 경우 홀수 pid를 가지는 프로세스를 담으므로 홀수 Pid 가 먼저끝나는 것이 정상적인 출력 결과이며, 다음과 같이 정상적으로 출력되는 것을 확인할수있습니다.

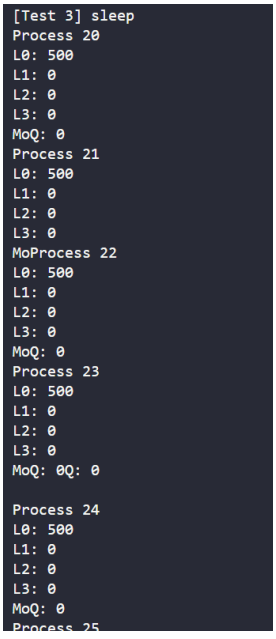
● Test 2 (Priority Test)

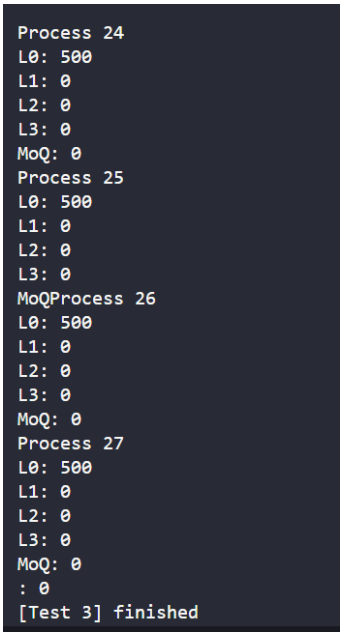




Test 2 의 경우 Pid 가 큰 Process 일수록 Priority 가 높게 설정되어있으므로 Priority 가 큰순서대로 출력되는것이 정상적인 결과이며 다음과같은 경향성(높은 pid가 먼저 끝남)을 확인할수 있습니다.

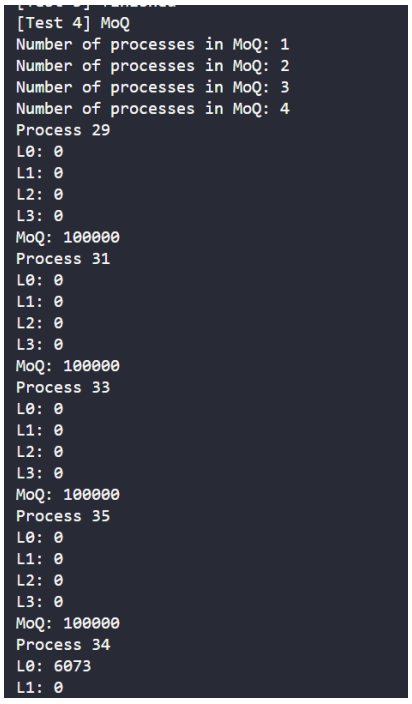
● Test 3 (Sleeping Test)

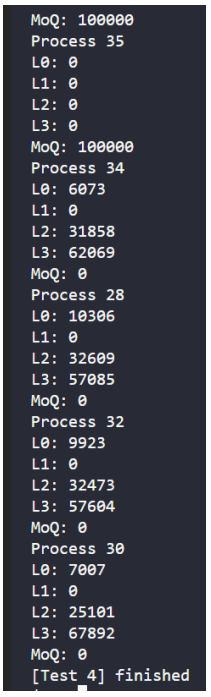




Teset 3의 경우 L0에서 머무르기에 L0에서 500이 출력되는것을 확인할수있습니다. 따라서 Pid 가 작은 Process 가 먼저 끝나는 경향성을 확인할수 있습니다.

● Test 4 (MoQ Test)





Pid 가 작은 Process 가 먼저 MoQ에 들어가므로, Pid 가 작은 Process 가 먼저 종료되는 것을 확인할수 있습니다. 위의 스크린샷에서는 29,31,33, 35 번 Process가 먼저 종료되는것을 확인할수있으며,

이후 ,34, 28, 32, 30 순으로 짝수번 Process는 그다음 종료되는것을 확인할수있습니다.

5️⃣ Trouble Shooting

Xv6의 경우 운영체제이기에 정상적인 디버깅 방식이 안된다는 것을 확인하였으며,

다음과같이 중간중간 Print를 할수있는 함수 와 코드를 구현하였습니다.

● printf_queue_info 구현[proc.c]

```
void print_queue_info() {
  struct proc *current_proc;
  for (int i = 0; i < 5; i++) {  // 각 레벨별로 순회
      if (ptable.numproc[i]) cprintf("Level %d processes:\n", i);
      current_proc = ptable.heads[i];
      while (current_proc != NULL) {
          // UNUSED 상태가 아닌 프로세스 정보 출력
          if (current_proc->state != UNUSED) {
              cprintf(" PID: %d,Plevel: %d, Name: %s, State: %s\n", current_proc->pid,cu
          }
          current_proc = current_proc->next;  // 다음 노드로 이동
      }
  }
  return;
}
```

각 큐별로 존재하는 Process의 상태를 출력하는 함수입니다.

● print_proc_info함수구현[proc.c]

```
void print_proc_info() {
    struct proc *tmp_p;
    cprintf("Current processes in the system:\n");
    for(tmp_p = ptable.proc; tmp_p < &ptable.proc[NPROC]; tmp_p++) {
        if (tmp_p->state != UNUSED) {  // UNUSED 상태가 아닌 프로세스만 출력
            cprintf("PID: %d, Name: %s, State: %s Level : %d priority : %d\n", tmp_p->pi
        }
    }
}
```

에러가 생길때마다 위의 두함수를 이용하여, 중간중간, print 하였습니다.

⚠ Issue 1 : Link List Self Pointing Issue

[Log Print]

Level 0 processes:
Level 1 processes:

PID: 5, Name: mlfq_test, State: RUNNABLE
PID: 11, Name: mlfq_test, State: RUNNABLE
PID: 9, Name: mlfq_test, State: RUNNABLE
Level 2 processes:
PID: 6, Name: mlfq_test, State: RUNNABLE
PID: 10, Name: mlfq_test, State: RUNNABLE
PID: 4, Name: mlfq_test, State: RUNNABLE
PID: 8, Name: mlfq_test, State: RUNNABLE
Level 3 processes:
Level 4 processes:

(267) [scheduler→ process] pid :5, pname:mlfq_test, level : 1
(267) [process→scheduler] pid : 5 pname: mlfq_test

(271) pid : 5,pname : mlfq_test [1→3] 로 큐이동

Level 0 processes:
Level 1 processes:
PID: 11, Name: mlfq_test, State: RUNNABLE
PID: 9, Name: mlfq_test, State: RUNNABLE
Level 2 processes:
PID: 6, Name: mlfq_test, State: RUNNABLE
PID: 10, Name: mlfq_test, State: RUNNABLE
PID: 4, Name: mlfq_test, State: RUNNABLE
PID: 8, Name: mlfq_test, State: RUNNABLE
Level 3 processes:
PID: 7, Name: mlfq_test, State: RUNNABLE

PID: 5, Name: mlfq_test, State: RUNNING

PID: 11, Name: mlfq_test, State: RUNNABLE
PID: 9, Name: mlfq_test, State: RUNNABLE

다음과 같이 L1 에 있던 Pid 5인 Queue 가 L3로 이동을 할시 기존에 연결을 하고있던 5→11→9에서 5만 밖으로 빠져나와야 하지만, 5→11→ 9 가 그대로 연결이 지속되어 L1, L3 큐에 11,9Process 가 중복되는 이슈가 있었습니다. 해당 이슈는 다음과같이

```
  if (ptable.numproc[tmp_level] == 0) {
    ptable.heads[tmp_level] = ptable.tail[tmp_level] = p;
    p->next = NULL;
  }
```

Queue_push 함수에 p→next =NULL 로 설정을 하니 문제가 사라졌습니다.

⚠ Issue 2 : MoQ 알고리즘 이슈

```
for(struct proc* p= ptable.proc ; p < &ptable.proc[NPROC]; p++){
  if(p->level != 4 && p != NULL){
    p->level = 0;
    p->passed_time = 0;
  }
}
```

MoQ의 Setmonopoly 이후 Boost 함수가 호출될시 , setmonopoly 를 통해 설정된 level4가 다시 level0 으로 재조정되는 이슈가 있었기에 boost 함수가 호출시, setmonopoly 가 적용되어 level 4로 조정된 process의 경우 boost 의 영향을 받지 않도록 설계하였습니다.

```
if(p->level !=4){
  p->chan = chan;
  p->state = SLEEPING;
 // cprintf("sleep sched 호출 process 이름은 %s\n", p->name);
  sched();
}
```

MoQ 의 경우 현재 Process가 Sleep을 할시 Queue_pop을 한 이후 Queue_push 를 할지 ,sleep을 지속시킨상태로 끝날때까지 기다릴지 고민을 하였으나 ,MoQ 의 정책에 맞추어 Sleep 을 지속시킨 상태로 끝날때까지 지속하도록 하였습니다.