REPORT

- 제 목 : Project2 Wiki
- 수강과목 : 202410HY13065_운영체제
- 담당 교수: 강수용 교수님
- 학 과 : 컴퓨터소프트웨어학부
- 학 번 : 2019089270
- 이 름 : 김주호
- 제출일자: 2024.04.21

Project2 wiki

Outline

CPU Scheduling

- MLFQ(Multi-Level Feedback Queue) with Priority Boosting구현
  - L0~L2: Round Robin Scheduling
  - L3: Priority Queue Scheduling
- MoQ(Monopoly Queue) 구현
  - FCFS Scheduling

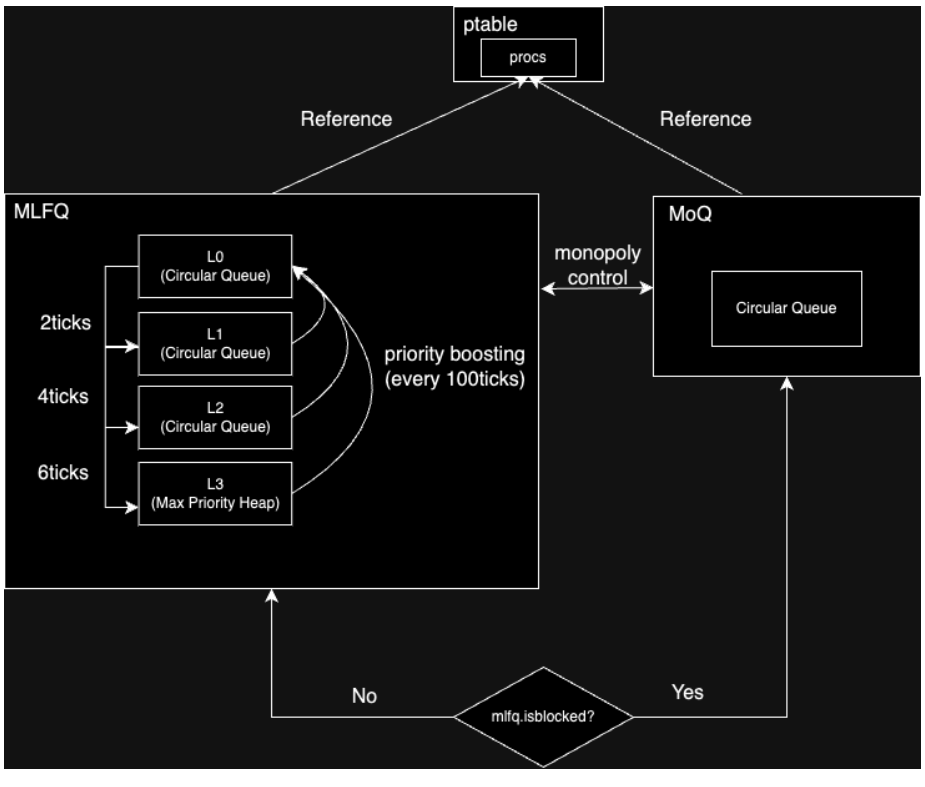
System call

cpu 스케쥴러를 활용하기 위한 system call의 구현

- void yield(void)
  - 자신이 점유한 cpu를 양보한다.
- int getlev(void)
  - 프로세스가 속한 큐의 레벨을 반환, MOQ인 경우 99 반환
- int setpriority(int pid, int priority)
  - 특정 pid를 가지는 프로스세의 priority 설정
  - 성공시 0
  - 실패시
    - pid가 존재하지 않는 경우 -1
    - priority가 0~10의 정수가 아닌경우 -2 반환
- int setmonopoly(int pid, int password)
  - 특정 pid를 가지는 프로세스를 MoQ로 이동하고 password로 학번을 입력받는다
  - password가 일치시
    - 종료되지 않는 MoQ내의 프로세스의 개수를 반환
  - 실패시
    - 존재하지 않는 프로세스의 pid인 경우 -1
    - 암호가 일치하지 않는 경우 -2
    - 이미 MoQ에 존재하는 경우 -3
    - 자기 자신을 MoQ로 이동시키려 하는 경우 -4
- void monopolize(void)
  - mlfq 스케쥴링을 중지하고 MoQ 스케쥴링을 시작합니다
- void unmonopolize(void)
  - MoQ스케쥴링을 중지하고 mlfq 스케쥴링으로 돌아갑니다.

Design

CPU Scheduling



MLFQ

L0~L3가 모두 동일한 구조체를 활용하기 위해서 Queue를 배열로 구현할 것이다.

- L0~L2: 배열로 queue를 구현하면 큐가 가리키는 위치가 점점 커지는 문제가 발생하므로 circular queue를 활용할 것이다.
- L3: 우선순위를 고려하기 위해서 priority를 pivot으로 활용한 max heap를 활용하여 구현할 것이다.

MoQ

기존 MLFQ의 큐 구조는 그대로 활용하되 FCFS가 성립하도록 큐의 동작을 구현할 것이다.

Monopoly Control

setmonoploy와 unmonopolize 함수가 실행될때 MLFQ와 MoQ를 동시에 컨트롤하는 함수를 제공하여 상태를 제어할 수 있도록 할 것이다.

Scheduling Control

ptable을 따로 변경하지 않고 MLFQ와 MoQ모두 ptable의 proc 구조체를 참조하는 방식으로 구현하여 최대한 기존의 형태를 유지하는 방향으로 설계하였으며 scheduling 진입시 MLFQ가 block이 되었는지 확인하여 MLFQ와 MoQ 스케쥴링이 선택될 수 있도록 구현할 것이다.

Implement

Structure

MLFQ와 MoQ의 구현을 위한 구조체들의 변경사항은 아래와 같습니다.

MLFQ

- MLFQ Structure

```
//proc.h
typedef struct proc_queue{
  struct proc* elements[NPROC+1];
  int front;
  int end;
  int size;
} proc_queue;


struct ptable_t{
  struct spinlock lock;
  struct proc proc[NPROC];
};
```

```
struct mlfq_t{
  struct proc_queue queues[4];
  struct spinlock lock;
  int time_quantum[4];
  int ticks;
  int is_blocked;
} ;
```

- MLFQ Method
  - class의 method를 쓰는 것처럼 필요한 함수들을 구현하여queue의 배열의 직접적으로 접근하는 것을 최대한 제한하였습니다.

```
//proc.c
//---- MLFQ ----

void mlfq_block(){
  mlfq.is_blocked=1;
}

void mlfq_unblock(){
  mlfq.is_blocked=0;
  mlfq.ticks=0;
}

int
mlfq_is_full(int level){
  return mlfq.queues[level].size==NPROC;
}

int
mlfq_is_empty(int level){
  return mlfq.queues[level].size==0;
}

//해당 레벨의 큐에 프로세스를 삽입
//성공시 0, 실패시 -1 반환
int
mlfq_enqueue(struct proc* p, int priority, int level){
  if(mlfq_is_full(level)){
    return -1;
  }
  if(level<3){
    mlfq.queues[level].end = (mlfq.queues[level].end + 1) % (NPROC+1);
    mlfq.queues[level].elements[mlfq.queues[level].end]=p;
  }else{
    //우선순위큐
    int idx=mlfq.queues[level].size;
    while((idx!=0) && (mlfq.queues[level].elements[(idx-1)/2]->priority<priority)){
      mlfq.queues[level].elements[idx]=mlfq.queues[level].elements[(idx-1)/2];
      idx=(idx-1)/2;
    }
    mlfq.queues[level].elements[idx]=p;
  }
  mlfq.queues[level].size++;
  return 0;
}
```

```
//해당 레벨의 큐에서 프로세스의 주소를 반환
//실패시 0 반환
struct proc*
mlfq_dequeue(int level){
  struct proc* ret=0;
  if(mlfq_is_empty(level)){
    return ret;
  }
  if(level<3){
    mlfq.queues[level].front = (mlfq.queues[level].front + 1) % (NPROC+1);
    ret=mlfq.queues[level].elements[mlfq.queues[level].front];
  }else{
    //우선순위 큐
    int parent=0, child=1;
    ret=mlfq.queues[level].elements[0];
    struct proc* last=mlfq.queues[level].elements[mlfq.queues[level].size-1];
    while(child<=mlfq.queues[level].size-1){
      if(child<mlfq.queues[level].size-1 && mlfq.queues[level].elements[child]->priority<mlfq
        child++;
      }
      if(last->priority>=mlfq.queues[level].elements[child]->priority){
        break;
      }
      mlfq.queues[level].elements[parent]=mlfq.queues[level].elements[child];
      parent=child;
      child=child*2+1;
    }
    mlfq.queues[level].elements[parent]=last;
  }
  mlfq.queues[level].size--;
  return ret;
}

//pid에 해당하는 프로세스를 queue에서 제거하고 해당 주소를 반환한다.
//존재하지 않는 경우 0 반환
struct proc*
mlfq_delete(int pid){
  int lvl=0;
  //MLFQ에서 해당 프로세스가 어느 큐에 있는지 찾기
  int flag=0;
  struct proc* target=0;
  for(lvl=0;lvl<4;lvl++){
    if(lvl<3){
      int i=(mlfq.queues[lvl].front+1)%(NPROC+1);
      while(1){
        if(mlfq.queues[lvl].elements[i]->pid==pid){
          target=mlfq.queues[lvl].elements[i];
          flag=1;
          break;
        }
        if(i==mlfq.queues[lvl].end) break;
        i=(i+1)%(NPROC+1);
      }
    }else{
      for(int i=0;i<mlfq.queues[lvl].size;i++){
        if(mlfq.queues[lvl].elements[i]->pid==pid){
          target=mlfq.queues[lvl].elements[i];
          flag=1;
```

```
        break;
      }
    }
  }
  if(flag) break;
}
//프로세스가 존재하지 않는 경우
if(target==0) return 0;

//MLFQ에서 해당 프로세스 제거
struct proc* temp[NPROC];
int count=0;
while(!mlfq_is_empty(lvl)){
  struct proc* cur=mlfq_dequeue(lvl);
  if(cur->pid==pid){
    continue;
  }
  temp[count++]=cur;
  }
  for(int i=0;i<count;i++){
    mlfq_enqueue(temp[i], temp[i]->priority, temp[i]->level);
  }
  return target;
}

//해당 레벨의 큐에서 가장 앞의 프로세스의 주소를 반환
struct proc* mlfq_front(int lvl){
  if(mlfq_is_empty(lvl)){
    return 0;
  }
  if(lvl<3){
    return mlfq.queues[lvl].elements[(mlfq.queues[lvl].front+1)%(NPROC+1)];
  }else{
    return mlfq.queues[lvl].elements[0];
  }
}


//우선순위 큐를 다시 만든다.
//ptable lock이 걸려있어야 함
void
mlfq_remake_priority_queue(){
  struct proc* temp[NPROC];
  int count=0;
  while(!mlfq_is_empty(3)){
    temp[count++]=mlfq_dequeue(3);
  }
  for(int i=0;i<count;i++){
    mlfq_enqueue(temp[i], temp[i]->priority, 3);
  }
}
```

MoQ

- MoQ Structure
  - MLFQ의 proc_queue를 그대로 계승하여 사용하였습니다.

```
//proc.c
struct proc_queue moq;//Monopoly Queue
```

- MoQ Control
  - MLFQ와 유사하게 최대한 class의 method을 사용하는 것처럼 각각 함수들을 만들어 배열의 직접적인 접근을 최소화하는 방식으로 구현하였습니다.

```
//proc.c
int
moq_is_full(){
  return moq.size==NPROC;
}

int
moq_is_empty(){
  return moq.size==0;
}

int moq_is_have_proc(int pid){
  if(moq_is_empty()) return 0;
  int i=(moq.front+1)%(NPROC+1);
  while(1){
    if(moq.elements[i]->pid==pid){
      return 1;
    }
    if(i==moq.end) break;
    i=(i+1)%(NPROC+1);
  }
  return 0;
}

struct proc*
moq_find_active_proc(int* active_cnt){
  if(moq_is_empty()) return 0;
  int i=(moq.front+1)%(NPROC+1);
  while(1){
    if(moq.elements[i]->state!=ZOMBIE){
      (*active_cnt)++;
    }
    if(moq.elements[i]->state==RUNNABLE || moq.elements[i]->state==RUNNING){
      return moq.elements[i];
    }
    if(i==moq.end) break;
    i=(i+1)%(NPROC+1);
  }
  return 0;
}

int moq_enqueue(struct proc* p){
  if(moq_is_full()){
    return -1;
  }
  moq.end=(moq.end+1)%(NPROC+1);
  moq.elements[moq.end]=p;
  moq.size++;
  return 0;
```

```
}

struct proc*
moq_dequeue(){
  if(moq_is_empty()){
    return 0;
  }
  moq.front=(moq.front+1)%(NPROC+1);
  struct proc* ret=moq.elements[moq.front];
  moq.size--;
  return ret;
}

void
moq_delete(int pid){
  struct proc* temp[NPROC];
  int count=0;
  while(!moq_is_empty()){
    struct proc* cur=moq_dequeue();
    if(cur->pid==pid) continue;
    temp[count++]=cur;
  }
  for(int i=0;i<count;i++){
    moq_enqueue(temp[i]);
  }
}
```

proc

- system call의 반환 값과 MLFQ의 동작을 처리하기 위해서 proc 구조체에 level, time_quantum, priority 값을 추가하였습니다.

```
//proc.h
// Per-process state
struct proc {
    ...
  int level;                  // queue level
  uint time_quantum;           // time quantum
  int priority;               // priority
  char name[16];              // Process name (debugging)
};
```

Scheduling

스케쥴링을 위해 추가 및 변경된 함수에 대한 구현 사항은 아래와 같습니다.

- void proc_queue_init(void)
  - main.c에 보면 int main(void)에 시작할때 구동되는 함수들을 모아두었는데 이때 실행되는 pinit()에서 MLFQ와 MoQ의 초기화를 진행합니다.

```
//proc.c
void
proc_queue_init(void){
  //mlfq init
  int len=sizeof(mlfq.queues)/sizeof(mlfq.queues[0]);
  for(int i = 0; i < len; i++){
    memset(mlfq.queues[i].elements, 0, sizeof(struct proc*)*(NPROC+1));
```

```
    mlfq.queues[i].front=0;
    mlfq.queues[i].end=0;
    mlfq.queues[i].size=0;
    mlfq.time_quantum[i] = 2*i+2;
  }
  mlfq.ticks=0;
  mlfq.is_blocked=0;

  //moq init
  memset(moq.elements, 0, sizeof(struct proc*)*(NPROC+1));
  moq.front=0;
  moq.end=0;
  moq.size=0;
}

void
pinit(void)
{
  initlock(&ptable.lock, "ptable");
  proc_queue_init();
}
```

- static struct proc* allocproc(void)
  - 프로세스를 할당하는 함수들인 fork, userinit과 같은 함수들을 보면 allocproc 함수를 이용하는 것을 알 수 있다. 이 과정에서 ptable에 프로세스를 삽입하는데 이때 MLFQ에도 프로세스의 참조값을 넣어주어 큐에 프로세스가 진입되도록 하였습니다.

```
//proc.c
static struct proc* allocproc(){
...
if(mlfq_enqueue(p, 0, 0)==-1){ //L0 queue에 삽입이 되지 않는 경우 프로세스를 생성하지 않음
    release(&ptable.lock);
    return 0;
  }
...
}
```

- void scheduler(void)
  - 이번 프로젝트의 가장 핵심적인 부분으로 크게 프로세스를 찾는 부분, 타임 퀀텀을 계산하여 MLFQ를 조정하는 부분으로 구성이 되어 있습니다.
    - 프로세스를 찾는 부분
      - 현재 스케쥴링이 MLFQ, MoQ 중 어느 큐에서 진행되는지에 따라서 프로세스를 가져옵니다.

```
//proc.c
void scheduler(void){
...
if(!mlfq.is_blocked){
      p=find_runnable_proc();
    }else{
        int active_cnt=0;
      p=moq_find_active_proc(&active_cnt);
      if(active_cnt==0){ //독점적으로 수행할 프로세스가 없는 경우
        //unmonopolize
        mlfq_unblock();
      }
    }
```

```
...
}
```

- struct proc* find_runnable_proc(void)
  - MLFQ에서 runnable한 process를 찾는 함수로 현재 수행할 프로세스를 queue의 가장 앞에 두고 이외의 프로세스는 그 순서를 유지합니다.

```
//RUNNABLE 프로세스를 queue에서 찾아 반환
//실패시 0 반환
struct proc* find_runnable_proc(){
  struct proc* ret=0;
  int flag=0;

  for(int lvl=0; lvl<4; lvl++){
    if(mlfq_is_empty(lvl)){
      continue;
    }
      //큐를 순회하면서 첫번째로 발견되는 RUNNABLE 프로세스를 찾음
      int size=mlfq.queues[lvl].size;
      struct proc* temp[size];
      int count=0;
      //수행할 프로세스를 찾고 큐의 가장 앞에 있도록 처리
      //이외의 프로세스는 순서를 그대로 유지
      while(!mlfq_is_empty(lvl)){
        struct proc* cur=mlfq_dequeue(lvl);
        if(cur->state==RUNNABLE && !flag){
          ret=cur;
          flag=1;
        }else{
          temp[count++]=cur;
        }
      }
      if(ret) mlfq_enqueue(ret, ret->priority, ret->level);
      for(int i=0; i<count; i++){
        mlfq_enqueue(temp[i], temp[i]->priority, temp[i]->level);
      }
      if(flag) break;
  }
  if(flag==0) ret=0; //없으면 더미 포인터 반환
  return ret;
}
```

- struct proc* moq_find_active_proc(int* active_cnt)
  - MoQ의 경우 들어온 순서대로 실행이 되어야하므로 먼저 들어온 프로세스가 sleep 중인 상태가 아니라면, 먼저 들어온 프로세스가 종료될때까지 계속 스케쥴링이 수행됩니다.

```
struct proc*
moq_find_active_proc(int* active_cnt){
  if(moq_is_empty()) return 0;
  int i=(moq.front+1)%(NPROC+1);
  while(1){
    if(moq.elements[i]->state!=ZOMBIE){
      (*active_cnt)++;
    }
    if(moq.elements[i]->state==RUNNABLE || moq.elements[i]->state==RUNNING){
      return moq.elements[i];
```

```
    }
    if(i==moq.end) break;
    i=(i+1)%(NPROC+1);
  }
  return 0;
}
```

- MLFQ를 조정하는 부분
  - 바꿀 프로세스가 있는 경우에만 해당 부분으로 진입합니다. 또한 MoQ는 따로 큐를 조정해줄 필요가 없으므로 MLFQ만 조정이 됩니다.
  - tick 컨트롤
    - 프로세스가 RUNNING → RUNNABLE로 변경이 되었을때만 즉, 정상적으로 종료가 되었을때만 time_quantum을 상승시키며 MLFQ의 global tick의 경우 starvation을 해결하기 위해 존재하는 것이므로 MLFQ의 프로세스가 변경이 될 때마다 1tick을 상승시킵니다.
  - 프로세스가 종료되면 큐의 가장 뒤에 삽입되는데 우선순위큐의 경우 priority만을 생각하는 max heap인데 priority를 변경하는 상황을 고려해야하므로 항상 맨 앞에 있음을 보장하기 어렵습니다. 따라서 해당 프로세스를 찾을때 까지 반복하여 큐 뒤에 집어넣는 코드가 포함되어 있습니다.

```
//proc.c
void scheduler(void){
...
if(p!=0 && !(mlfq.is_blocked && (curproc!=0 && curproc->pid==p->pid))){
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();
      //cprintf("ticks:%d\n", mlfq.ticks);


      if(!mlfq.is_blocked){
        //MLFQ 스케쥴링
        if(p->state==RUNNABLE){ //올바르게 process가 종료됨
          p->time_quantum++;
        }
        //priority boosting
      if(++mlfq.ticks==100){
        mlfq.ticks=0;
        struct proc* temp[NPROC];
        int count=0;
        for(int lvl=1; lvl<4;lvl++){
          while(!mlfq_is_empty(lvl)){
            temp[count++]=mlfq_dequeue(lvl);
          }
        }
        for(int i=0;i<count;i++){
          temp[i]->level=0;
          temp[i]->time_quantum=0;
          mlfq_enqueue(temp[i], temp[i]->priority, 0);
        }
      }
      //time_quantum이 time_quantum을 넘으면 우선순위를 낮추거나 레벨을 낮추어 큐에 삽입하고 새로
```

```
          else if((p->time_quantum)>=mlfq.time_quantum[p->level]){
            int lvl=p->level;
            if(p->level==0){
              if(p->pid%2!=0){
                p->level=1;
              }else{
                p->level=2;
              }
            }else{
              if(p->level==3){
                p->priority=p->priority>0 ? p->priority-1 : 0;
              }
              p->level=3;
            }
            p->time_quantum=0;

            //프로세스가 수행을 완료한 경우 다시 큐에 삽입
            //RR인 경우 1번만 수행하면 되지만
            //PQ인 경우 큐를 순회하면서 찾아야함
            struct proc* temp[NPROC];
            int count=0;
            do{
              temp[count++]=mlfq_dequeue(lvl);
              if(temp[count-1]->pid==p->pid){
                break;
              }
            }while(1);
            for(int i=0;i<count-1;i++){
              mlfq_enqueue(temp[i], temp[i]->priority, lvl);
            }
            mlfq_enqueue(temp[count-1], temp[count-1]->priority, temp[count-1]->level);
          }
        }

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
      }
...
}
```

- int wait(void)
  - 프로세스가 종료되어 ZOMBIE 상태가 되면 ptable에서 프로세스를 삭제하는 과정이 wait 함수에서 수행되므로 이때 MLFQ와 MoQ를 확인하여 종료된 프로세스를 제거하는 과정이 필요합니다.

```
//proc.c
int wait(void){
...
if(p->state == ZOMBIE){
        // Found one.
        //MLFQ에서 삭제하거나 MOQ에서 삭제
        if(mlfq_delete(p->pid)!=0){
          moq_delete(p->pid);
        }
...
}
```

System call

system call의 경우 직접적으로 추가한 system call이므로 ex_syscall.c이라는 파일을 추가하여 작성하였습니다. 이를 위해서 proc.c의 구현들을 참조할 수 있도록 proc.h를 변경하였습니다.

```
//proc.h
...
typedef struct proc_queue{
  struct proc* elements[NPROC+1];
  int front;
  int end;
  int size;
} proc_queue;


struct ptable_t{
  struct spinlock lock;
  struct proc proc[NPROC];
};

struct mlfq_t{
  struct proc_queue queues[4];
  struct spinlock lock;
  int time_quantum[4];
  int ticks;
  int is_blocked;
} ;


extern struct ptable_t ptable;
extern struct mlfq_t mlfq; //Multi-Level Feedback Queue

extern void mlfq_remake_priority_queue();
extern int moq_is_have_proc(int);
extern void proc_queue_init(void);
extern int move_mlfq_to_moq(int);
extern int move_moq_to_mlfq();
extern void mlfq_block(void);
extern void mlfq_unblock(void);
extern void print_current_state_p(struct proc* p);
```

- void yield(void)

```
//ex_syscall.c
// 자신이 점유한 cpu를 양보합니다.
void sys_yield(void)
{
  yield();
}
```

- int getlev(void)

```
//ex_syscall.c
int
getlev(void){
    return myproc()->level;
}
```

```
//현재 프로세스가 속한 큐의 레벨 반환
int sys_getlev(void)
{
  return getlev();
}
```

- int setpriority(void)
  - 프로세스 객체의 수정이 요구되므로 ptable의 lock을 잡고 수행되며 우선순위가 변경되었을 경우 max heap을 재생성하여 우선순위를 유지할 수 있도록합니다.

```
//ex_syscall.c
int setpriority(int pid, int priority){
  if(priority<0 || priority>10) return -2;
  struct proc* p;
  int ret=-1;
  acquire(&ptable.lock);
  //MLFQ에서 해당 프로세스의 우선순위를 변경
  for(int i = 0; i < NPROC; i++){
      p = &ptable.proc[i];
      if(p->pid == pid){
          p->priority = priority;
          if(p->level==3){
            mlfq_remake_priority_queue();
          }
          ret = 0;
      }
  }
  release(&ptable.lock);
  return ret;
}

//프로세스의 우선순위를 설정
int sys_setpriority(void)
{
  int pid,priority;
  if(argint(0, &pid) < 0 || argint(1, &priority) < 0)
    return -1;
  return setpriority(pid, priority);
}
```

```
//proc.c
//우선순위 큐를 다시 만든다.
//ptable lock이 걸려있어야 함
void
mlfq_remake_priority_queue(){
  struct proc* temp[NPROC];
  int count=0;
  while(!mlfq_is_empty(3)){
    temp[count++]=mlfq_dequeue(3);
  }
  for(int i=0;i<count;i++){
    mlfq_enqueue(temp[i], temp[i]->priority, 3);
  }
}
```

- int setmonopoly(int pid, int password)

- 명세에 맞게 에러는 처리하고 해당 pid를 가진 프로세스를 moq로 이동합니다. 이때 스케쥴링이 실행되면 안되므로 ptable에 대한 lock을 가지고 수행합니다.
- 또한 반환시 zombie가 아닌 프로세스의 개수를 반환합니다.

```
//ex_syscall.c
int setmonopoly(int pid, int password){
  if(password!=PW) return -2;
  if(moq_is_have_proc(pid)) return -3;
  if(myproc()->pid==pid) return -4;
  acquire(&ptable.lock);
  int ret=move_mlfq_to_moq(pid);
  release(&ptable.lock);
  return ret;
}

//프로세스를 MoQ로 이동
int sys_setmonopoly(void)
{
  int pid,password;
  if(argint(0, &pid) < 0 || argint(1, &password) < 0)
    return -1;
  return setmonopoly(pid, password);
}

//proc.c
//MLFQ에 있는 pid를 가진 프로세스를 MOQ로 옮김
//실패시 -1 반환
int move_mlfq_to_moq(int pid){
  if(moq_is_full() || moq_is_have_proc(pid)) return -1;
  struct proc* target=mlfq_delete(pid);
  if(target==0){
    return -1;
  }

  //MOQ에 삽입
  target->level=99;
  moq_enqueue(target);
  int i=(moq.front+1)%(NPROC+1);
  int active_cnt=0;
  while(1){
    if(moq.elements[i]->state!=ZOMBIE){
      active_cnt++;
    }
    if(i==moq.end) break;
    i=(i+1)%(NPROC+1);
  }
  return active_cnt;
}
```

- void monopolize()
  - mlfq에 block을 걸어 moq가 시작되도록합니다. 이때 스케쥴링이 되지 않도록 lock을 수행합니다. 완료되면 바로 다음 프로세스가 수행될 수 있도록 yield를 수행합니다.

```
//ex_syscall.c
//MoQ의 프로세스가 CPU를 독점하도록한다.
void monopolize(void){
  int pid=myproc()->pid;
```

```
  acquire(&ptable.lock);
  mlfq_block();
  if(moq_is_have_proc(pid)){
    release(&ptable.lock);
    return; //이미 실행되고 있는 경우
  }
  release(&ptable.lock);
  yield();
}

//프로세스가 CPU를 독점하도록 설정
int sys_monopolize(void)
{
  monopolize();
  return 0;
}
```

- void unmonopolize()
  - mlfq에 block을 해제하고 moq의 남은 프로세스들은 MLFQ의 입장에서 새로 들어오는 프로세스이므로 L0에 삽입됩니다. 이때 스케쥴링이 되지 않도록 lock을 잡고 진행하며 완료되면 바로 다음 프로세스가 스케쥴링이 될 수 있도록 yield를 수행합니다.

```
//ex_syscall.c
//MoQ의 프로세스가 CPU 독점을 해제하고 MLFQ로 돌아간다.
void unmonopolize(void){
  acquire(&ptable.lock);
  mlfq_unblock();
  move_moq_to_mlfq();
  release(&ptable.lock);
  yield();
}


//프로세스가 CPU 독점을 해제하고 MLFQ로 돌아간다.
int sys_unmonopolize(void)
{
  unmonopolize();
  return 0;
}

//proc.c
//MOQ에 있던 프로세스를 MLFQ로 이동한다.
//이때 L0 queue로 이동한다.
int move_moq_to_mlfq(){
  if(mlfq_is_full(0)) return -1;
  while(!moq_is_empty()){
    struct proc* target=moq_dequeue();
    target->level=0;
    mlfq_enqueue(target, target->priority, 0);
  }
  return 0;
}
```

Debugging

MLFQ와 MoQ의 정상작동 여부의 확인을 위해서 디버깅용으로 추가된 코드들이 존재합니다.

- 특히 권한을 가지고 있는 유저(PW)의 경우 현재 프로세스를 유저단에서 확인할 수 있는 디버깅용 코드가 존재합니다.

```
//proc.c
void print_current_state_p(struct proc* pcur){
  if(pcur!=0){
      cprintf("Current Process: (pid:%d, time_quantum:%d)\n", pcur->pid, pcur->time_quantum);
  }
  if(!mlfq.is_blocked){
    cprintf("RR Queue: \n");
    for(int lvl=0;lvl<3;lvl++){
      cprintf("Level %d: ", lvl);
      if(!mlfq_is_empty(lvl)){
        int cur=(mlfq.queues[lvl].front+1)%(NPROC+1);
        while(1){
          cprintf("(pid:%d, time_quantum:%d, state:%d) ", mlfq.queues[lvl].elements[cur]->pid, m
          if(cur==mlfq.queues[lvl].end){
            break;
          }
          cur=(cur+1)%(NPROC+1);
        }
      }
      cprintf("(size:%d)\n", mlfq.queues[lvl].size);
    }
    cprintf("Priority Queue: ");
    for(int i=0;i<mlfq.queues[3].size;i++){
      cprintf("(pid:%d, time_quantum:%d, state:%d, priority:%d) ", mlfq.queues[3].elements[i]->p
    }
    cprintf("(size:%d)\n", mlfq.queues[3].size);
  }else{
    cprintf("Monopoly Queue: ");
    if(!moq_is_empty()){
        int i=(moq.front+1)%(NPROC+1);
        while(1){
          cprintf("(pid:%d, time_quantum:%d, state:%d) ", moq.elements[i]->pid, moq.elements[i]-
          if(i==moq.end) break;
          i=(i+1)%(NPROC+1);
        }
    }
    cprintf("(size:%d)\n", moq.size);
  }
}

void print_process(struct proc* p){
  if(p==0){
    cprintf("0 Process!\n");
  }else{
      cprintf("pid: %d, name: %s, state: %d, level: %d, time_quantum: %d, priority: %d, pgdir:%0
  }
}

//ex_syscall.c
void printcurrentstate(int pw){
  if(pw!=PW) return;
  acquire(&ptable.lock);
  print_current_state_p(0);
  release(&ptable.lock);
}
```

```
int sys_printcurrentstate(void){
  int password;
  if(argint(0, &password) < 0)
    return -1;
  printcurrentstate(password);
  return 0;
}
```

Result

os 이미지 생성

아래의 명령어를 순서대로 실행하면 됩니다.

- make clean
- make
- make fs.img

Test

Given Test Case

- [Test 1] default
- [Test 2] Priority

```
[Test 1] default
Process 7
L0: 6078
L1: 10519
L2: 0
L3: 33403
MoQ: 0
Process 5
L0: 7966
L1: 11840
L2: 0
L3: 30194
MoQ: 0
Process 8
L0: 8329
L1: 0
L2: 21426
L3: 20245
MoQ: 0
Process 10
L0: 7998
L1: 0
L2: 24637
L3: 17365
MoQ: 0
Process 11
L0: 8039
L1: 15805
L2: 0
L3: 26156
MoQ: 0
Process 9
L0: 9778
L1: 20017
L2: 0
L3: 20205
MoQ: 0
Process 4
L0: 10171
L1: 0
L2: 28646
L3: 11183
MoQ: 0
Process 6
L0: 10131
L1: 0
L2: 27626
L3: 12243
MoQ: 0
[Test 1] finished
```

```
[Test 2] priorities
Process 19
L0: 5898
L1: 11943
L2: 0
L3: 32159
MoQ: 0
Process 18
L0: 7905
L1: 0
L2: 17395
L3: 24700
MoQ: 0
Process 17
L0: 8152
L1: 16462
L2: 0
L3: 25386
MoQ: 0
Process 16
L0: 7323
L1: 0
L2: 21616
L3: 21061
MoQ: 0
Process 14
L0: 10162
L1: 0
L2: 28921
L3: 10917
MoQ: 0
Process 15
L0: 10056
L1: 20604
L2: 0
L3: 19340
MoQ: 0
Process 12
L0: 10394
L1: 0
L2: 30262
L3: 9344
MoQ: 0
Process 13
L0: 10215
L1: 20307
L2: 0
L3: 19478
MoQ: 0
[Test 2] finished
```

전반적으로 우선순위가 높은 프로세스가 종료됨을 알 수 있다.

[Test 4] MoQ

전반적으로 홀수가 더 빨리 끝나는 것을 알 수 있다.

[Test 3] Sleep

```
[Test 3] sleep
Process 20
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 21
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 22
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: Process 23
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
0
Process 25
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 24
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 26
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
Process 27
L0: 500
L1: 0
L2: 0
L3: 0
MoQ: 0
[Test 3] finished
```

전반적으로 pid가 작은 프로세스가 먼저 끝나는 것을 알 수 있다.

```
[Test 4] MoQ
Number of processes in MoQ: 1
Number of processes in MoQ: 2
Number of processes in MoQ: 3
Number of processes in MoQ: 4
Process 29
L0: 0
L1: 0
L2: 0
L3: 0
MoQ: 50000
Process 31
L0: 0
L1: 0
L2: 0
L3: 0
MoQ: 50000
Process 33
L0: 0
L1: 0
L2: 0
L3: 0
MoQ: 50000
Process 35
L0: 0
L1: 0
L2: 0
L3: 0
MoQ: 50000
Process 28
L0: 2938
L1: 0
L2: 6013
L3: 41049
MoQ: 0
Process 34
L0: 4114
L1: 0
L2: 11654
L3: 34232
MoQ: 0
Process 32
L0: 4045
L1: 0
L2: 11681
L3: 34274
MoQ: 0
Process 30
L0: 6071
L1: 0
L2: 14576
L3: 29353
MoQ: 0
[Test 4] finished
```

홀수번째 그리고 pid가 작은 프로세스가 먼재 끝나는 것을 알 수 있다.

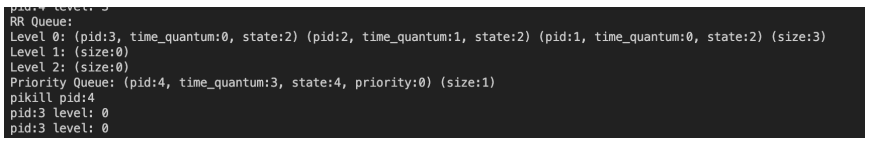
Personal Case

나머지 테스트 케이스는 모두 비슷하지만 unmonopolize를 묵시적으로 표기하는 프로세스에 대한 케이스가 있고 강제로 프로세스를 kill했을 때 로직이 정상적으로 작동하는지 확인하는 케이스가 있다. 모든 출력마다 현재 큐의 상태를 확인하여 테스트를 진행하였다.

```
//test_program.c
...
 case '3':
      printf(1, "[Test 3] monopolize Default\n");
      //독점모드가 잘 작동하는지 확인
      if(pid==parent){
        printf(1, "parent:%d\n", pid);
        int child=1;
        int ret=setmonopoly(child, PASSWORD);
        if(ret!=0){
          printf(1, "Monopoly Failed\n");
        }
        monopolize();
        printf(1,"monopolize\n");
        //child가 종료되고 실행되는 부분
        for(int i=0;i<SMALL_NUM_LOOP;i++){
          int x=getlev();
          printf(1, "level: %d\n", x);
```

```
        printcurrentstate(PASSWORD);
      }
    }else{
      printf(1, "child:%d\n", pid);
      for(i=0;i<MED_NUM_LOOP;i++){
        int x=getlev();
        printf(1, "level: %d\n", x);
        printcurrentstate(PASSWORD);
      }
      printf(1,"unmonopolize\n");
      unmonopolize();
    }
    exit_children();
    printf(1, "[Test 3] finished\n");
    break;
case '5':
    printf(1, "[Test 5] kill test\n");
    //child를 강제로 kill했을때 스케쥴러가 정상적으로 작동하는지 체크한다.
    pid=fork_children(1);
    if(pid==parent){
      sleep(10);
      int child=pid+1;
      kill(child);
      printf(1,"kill pid:%d\n", child);
      for(int i=0;i<SMALL_NUM_LOOP;i++){
        int x=getlev();
        printf(1, "pid:%d level: %d\n", pid, x);
      }
    }else{
      for(int i=0;i<MED_NUM_LOOP;i++){
        int x=getlev();
        printf(1, "pid:%d level: %d\n", pid, x);
        printcurrentstate(PASSWORD);
      }
      exit();
    }
    exit_children();
    printf(1, "[Test 5] finished\n");
...
```

- Result
  - case 5



- 실행 방법
  - $ test_program <number>
    - 1~5까지 가능하다.

Analysis

[Test 1] default에서 pid에 따른 순서가 실행할때마다 다른 케이스에 비해서 영향을 크게 받는데 왜 그런지 분석을 해보았다.

- 가장 큰 이유는 priority boosting이다.

```
MLFQ test start
[Test 1] default
priority boosting!
priority boosting!
priority boosting!
Process 7
L0: 3950
L1: 7672
L2: 0
L3: 38378
MoQ: 0
priority boosting!
Process 5
L0: 6687
L1: 12388
L2: 0
L3: 30925
MoQ: 0
Process 11
L0: 5554
L1: 10870
L2: 0
L3: 33576
MoQ: 0
priority boosting!
Process 4
L0: 9592
L1: 0
L2: 23145
L3: 17263
MoQ: 0
Process 10
L0: 10277
L1: 0
L2: 21292
L3: 18431
MoQ: 0
Process 9
L0: 7205
L1: 17291
L2: 0
L3: 25504
MoQ: 0
priority boosting!
Process 8
L0: 8547
L1: 0
L2: 29620
L3: 11833
MoQ: 0
Process 6
L0: 9175
L1: 0
L2: 26539
L3: 14286
MoQ: 0
[Test 1] finished
```

- 위의 테스트는 50000번의 loop 과정에서 priority boosting이 발생할때 마다 커널에서 cprintf(”priority boosting!\n”);를 실행한 결과인데 짧은 과정에서도 priority boosting이 다수 발생함을 알 수 있다.
    - priority boosting이 발생하게 되면 구현 로직상 L1 → L2 → L3 순서대로 다시 L0로 삽입이 되면서 time_quantum이 0이 되는데 이 과정에서 pid의 우선순위를 무시하는 경우를 다음과 같이 생각할 수 있다
        - 이미 홀수 pid를 가진 프로세스가 L3, 짝수 pid를 가진 프로세스가 L2에 존재하는 경우
        - L0로 올라온 프로세스가 time_quantum을 한 번만 쓰고 하위 레벨로 내려가기전에 priority boosting에 의해 다시 time_quantum이 0이 되는 경우
    - 우리의 스케쥴링은 1tick마다 실행되지만 loop문은 매우빠르게 수행이 되므로 위의 케이스가 순서에 영향을 주면 결과에 상당한 영향을 주게된다.

Trouble Shooting/Thinking

1. **Queue 구현 로직 선정**
    a. 기존에 MLFQ를 구현할때 linked list를 이용하려고 해서 malloc & free를 사용하려고 했으나 <stdlib.h>을 사용할 경우 exit function의 중복 선언 오류가 발생하고 <malloc.h>만 가져오는 경우 경로를 찾을 수 없는 오류가 발생하여서 memset과 같이 내장 함수가 존재하는지 확인해보았다. 그 결과 umalloc.c에 존재하는 함수 임을 알 수 있었는데 이는 유저가 호출하는 함수이므로 이를 커널에서 가져와서 쓰는게 맞는지 의문이 들어서 사용하지 못하고 아직 메모리 계열은 배우지 않아서 malloc을 구현하는 것도 너무 어려운 과정이라고 생각하여 배열을 이용하여 구현을 수행하였다.
2. **RUNNABLE한 프로세스를 찾을때 큐의 이동 문제**
    a. 처음에 구현당시 runnable한 프로세스를 찾으면서 만났던 프로세스들은 다시 뒤로 넘기는 식으로 구현을 하였는데 그때 당시에는 sleep인 프로세스니까 크게 문제가 안된다고 생각했으나 큐 내부에서도 starvation 문제가 발생할 수 있어 priority boosting과 합쳐서 매우 이상한 결과가 도출되었다. 따라서 현재는 순서를 유지하는 식으로 구현을 변경하여 문제를 해결하였다.
3. **global tick**
    a. 현재 MLFQ global tick의 경우 state가 sleep 상태가 반복이 되면 올라가지 않는다. 이때 starvation 문제는 특정 프로세스가 스케쥴링이 되지 않아서 발생하는 문제이기 때문에 아예 스케쥴링이 안되는 상황에서는 신경쓰지 않아도 될 문제라고 생각하여 위와 같이 구현하였지만 이에 대한 이렇게 구현해도 되는지에 대한 고민이 더 필요해보인다.