Operating System – Project1 Wiki

2019073181 심기호

Design

명세에서 요구하는 조건에 대해서 어떻게 구현할 계획인지, 어떤 자료구조와 알고리즘이 필요한지, 자신만의 디자인을 서술합니다.

0. process 구조체

-proc 구조체에 쓴 시간을 저장할 timequantum, 현재 큐 레벨을 저장할 level, priority를 저장할 priority 변수들을 추가로 만들어서 접근할 것이다.

1. MLFQ 구현방식

-실제로 3개의 큐를 구현하고, 큐에 해당하는 enqueue, dequeue 기능를 구현할 것이다. 큐에는 ptable의 proc 배열의 인덱스를 담을 것이다. 그래서 인덱스를 뽑아서 ptable의 proc배열의 뽑은 인덱스로 프로세스를 접근할 것이다. 프로세스가 생성되면 L0큐에 enqueue하고, scheduler()내부에서 MLFQ에서 큐가 뽑히는 방식은 pick_in_mlfq()함수를 통해서 L0에서 runnable한 프로세스를 찾고, 없다면 L1에서 runnable한 프로세스를 찾고, 없다면 L2에서 priority순으로 FCFS방식으로 runnable한 프로세스를 찾을 것이다. pick_in_mlfq()함수를 통해서 모든 큐를 순회했는데도 runnable한 프로세스가 없다면 scheduler()함수 내부에서 계속 for문을 도는 방식으로 구현할 것이다.

2. queue간의 이동방식 구현

프로세스들이 각 큐에서 돌다가 timequantum이 2n+4가 된다면 하위 큐로 내려가야 하는데, 이 구현은 scheduler()함수에서 pick_in_mlfq()함수에서 뽑고 그 프로세스로 context switch하기 전에 timequantum을 1증가시키고, 그 증가시킨 프로세스의 timequantum이 2n+4보다 작은지, 크거나 같은지를 비교해서 queue를 내려주거나 그대로 유지할 것이다. 만약 기존에 L2였다면 priority도 감소시켜주고, 감소했는데 0미만이였다면 다시 0으로 계속 고정할 것이다. 프로세스가 context switch를 통해 cpu를 받는 건 확실하기 때문에 아예 받기 전에 queue간의 이동을 구현하는 게 어떨까라는 생각에 이렇게 구현하기로 했다.

3. global tick과 priority boosting에 관한 구현

Trap.c에서 쓰이는 ticks 변수를 proc.c에서도 그대로 사용하기 위해 extern 변수로 proc.c에서도 사용했고, trap.c에서 ticks++하는 파트 뒤에 tick이 100이 되었는지 확인하고 바로 priorityboosting을 해주면 어떨까 라는 생각을 했다. Priorityboosting은 따로 함수를 만들어서 L1에 있는 프로세스를 다 빼서 다시 L0큐로 다시 넣어주고, L2에 있는 프로세스를 모두 빼서 L0로 넣어주고, L0에 있는 프로세스들을 모두 순회하면서 timequantum과 priority를 다시 초기화 해주는 방식으로 구현할 것이다.

4. schedulerLock/Unlock에 관한 구현

-먼저 schedulerLock이 걸리면 pick_in_mlfq()함수가 돌면 안되기 때문에 isLocked 라는 불리언의 역할을 하는 정수 변수를 하나 만들어서 초기값은 0으로 지정할 것이고, Lock이 호출되면 그 함수 안에서 isLocked 변수를 1로 만들고, Unlock이 호출되면 isLocked변수를 0으로 만들 것이다. schedulerLock이 systemcall로도, 인터럽트로도 불려야 한다고 생각해서 systemcall로 호출하는 부분은 usys.S, syscall.h, Makefile 등 호출해줘야 하는 부분에서 선언해주었고, 인터럽트로 불리는 부분은 trap.c에서 tf->trapno에서 case로 나누어서 호출해주었다.

5. SystemCall에 관한 구현

-proc.c에 원래 함수들을 만들고, sysproc.c에 wrapper function을 만들어준다. 그리고 defs.h, Usys.S 등 정의해줘야 할 적절한 파일들에 함수들을 정의해준다.

Implement

Proc 구조체 : proc구조체에서 level, priority, timequantum 변수를 만들어주었다. 현재 프로세스가 있는 큐의 레벨과 우선순위와 사용한 시간을 표시하기 위함이다.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  int priority;                // priority
  int timequantum;             // timequantum
  int level;                   // cur Q level
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

struct proc

Queue 구현 : 처음엔 큐를 아예 구현하지 않고, proc 구조체에 level변수를 만들어서 level을 저장해놓고, ptable을 순회하면서 레벨을 확인하고 가장 낮은 레벨이면서 runnable인 process를 뽑는 형식으로 scheduler을 구현하려고 했다.

이 방식대로 한다면 queue를 구현하지 않고 ptable을 레벨 수 만큼, 즉 64*3번 정도만 순회하면서 process를 scheduling하기 때문에 구현이 쉬울 것 같았다.

그런데 priorityboosting 상황에서 L0큐로 모든 프로세스가 재조정되는 상황에서 구현하기 어려울 것 같았다. L0큐로 갈 때 L1큐 먼저, L1에서도 먼저 들어온 프로세스 먼저, L2큐에서도 먼저 들어온 프로세스를 L0로 넣어줘야 하는데, 배열에서는 누가 먼저 들어왔는지를 따지면서 sorting작업도 필요할 것 같아서 ptable을 순회하는 방식으로는 하지 않기로 하였다.

그래서 MLFQ 말 그대로 Q3개를 직접 구현해서 process가 생겨나면 enqueue해주고, 뽑을 때에는 dequeue 를 해서 구현하는 것이 더 나아보였다. 큐에는 프로세스의 주소보다는 ptable.proc배열에서 프로세스가 위치한 인덱스를 넣는 방식으로 했다.

```
1  typedef struct queue{
2    int rear;
3    int front;
4    int index_list[64];
5    int size;
6  }Queue;
7
8  Queue mlfq[3];
9
10 int enqueue(int level, int index);
11 int dequeue(int level);
12 int is_full(int level);
13 int is_empty(int level);
14 void qinit();
```

queue.h

아무래도 주소값을 쓰게 되면 직접 접근하는 거라 훨씬 편리할 것 같지만, 포인터를 많이 쓰는 방식으로 하다보니 분명 실수가 많이 나올 것 같았다. 큐는 일반적인 큐가 아니라 circular queue를 통해서 64개 크기의 배열을 가진 queue를 구현했다. Queue.h에 새롭게 정의를 했고 queue는 int front, int rear, int index_list[64], int size라는 멤버변수를 가진다. 각각 큐의 맨 앞, 맨 뒤, ptable의 인덱스를 저장하는 배열, 그리고 현재 배열에 들어있는 프로세스의 개수를 나타내는 변수로 이루어져있다. 그리고 enqueue(int level, int index), dequeue(int level), isEmpty(int level), isFull(int level), void

qinit()으로 총 4개의 큐에 넣고, 빼고, 비어있나 확인하고, 가득찼나 확인하고, 초기화하는 함수들을 만들었다.

```c
#include "types.h"
#include "defs.h"
#include "queue.h"

int enqueue(int level, int index)
{
  mlfq[level].rear = (mlfq[level].rear + 1) % 64;   //  rear 하나 증가시키기
  mlfq[level].index_list[mlfq[level].rear] = index; //  list의 rear에 data 추가
  mlfq[level].size++;
  return 1;
}

/*    큐 데이터 꺼내기   */
int dequeue(int level)
{
  mlfq[level].front = (mlfq[level].front + 1) % 64;
  mlfq[level].size--;
  return mlfq[level].index_list[mlfq[level].front];
}

/*    공백 상태인지 여부    */
int is_empty(int level)
{
  if (mlfq[level].front == mlfq[level].rear)
    return 1;
  else
    return 0;
}

/*    포화 상태인지 여부    */
int is_full(int level)
{
  if (mlfq[level].size==64)
    return 1;
  else
    return 0;
}

void qinit()
{
  for (int i = 0; i < 3; i++)
  {
    mlfq[i].front = 0;
    mlfq[i].rear = 0;
    mlfq[i].size = 0;
  }
}
```

Queue.c

1. 각 큐는 각각 다른 timequantum을 가집니다.(timequantum<2*n+4)

-큐에서 뽑힌 프로세스가 cpu를 받기 전에 미리 timequnatum을 증가시켜준다. 그 후 p->level * 2 + 4와 비교를 해서 작다면, 원래 레벨로 다시 enqueue해준다. 2n+4보다 클리는 없겠지만, 혹시 모를 에러를 위해서 크거나 같을 때로 분기를 나누었고, 크거나 같다면 timequantum을 다시 0으로 만들고, 기존의 레벨이 L0나 L1이였다면, p->level+1인 레벨로 enqueue해주고, p->level도 하나 올려준다. 그리고 만약 p->level==2이였다면 레벨 변동 없이 L2로 enqueue해주고, priority를 하나 감소시켜준다. 감소시켰는데 0보다 작다면 원래 0이였다는 것이므로, 다시 0으로 설정해준다. 이 과정들을 통해 위 명세를 구현했다.

```
p->timequantum++; // tq 증가
    if (p->timequantum < (p->level * 2 + 4)) //tq가 2n+4보다 작다면
    {
      enqueue(p->level, pickedProcessIndex); // 원래 레벨로 뽑힌 인덱스 그대로 다시 넣어줌
    }
    else if (p->timequantum >= (p->level * 2 + 4)) // 2n+4보다 크다면
    {
      p->timequantum = 0; // 큐레벨에 상관없이 timequantum = 0 초기화
      if (p->level != 2)  // L0나 L1이라면
      {
        enqueue(p->level + 1, pickedProcessIndex); // 큐 레벨 하나 내려보내야함.
        p->level++; //level도 하나 증가
      }
      else if (p->level == 2) // L2레벨이라면
      {
        enqueue(2, pickedProcessIndex); //L2에 넣어준다.
        p->priority--;       // PRIORITY감소시키고
        if (p->priority < 0) // 0보다 작다면
        {
          p->priority = 0; // PRIORITY는 0으로 유지
        }
      }
    }
```

timequantum과 level 비교

2. 처음 실행된 프로세스는 가장 높은 레벨의 큐(L0)에 들어갑니다

-모든 프로세스가 탄생하는 allocproc()함수에서 enqueue(0,p-ptable.proc)을 통해서 처음 탄생하는 프로세스는 allocproc()에서 바로 L0에 넣어주었다.

```
115   // Set up new context to start executing at forkret,
116   // which returns to trapret.
117   sp -= 4;
118   *(uint *)sp = (uint)trapret;
119
120   sp -= sizeof *p->context;
121   p->context = (struct context *)sp;
122   memset(p->context, 0, sizeof *p->context);
123   p->context->eip = (uint)forkret;
124   enqueue(0, p - ptable.proc);//L0로 넣어줍니다.
125   release(&ptable.lock);
126   return p;
127 }
```

3. L0큐와 L1큐는 기본Round-Robin 정책을 따릅니다. 스케줄러는 기본적으로 L0큐의 RUNNABLE한 process를 스케줄링합니다 .L0의 RUNNABLE한 프로세스가 없을 경우, L1의 process를 스케줄링합니다.L1 큐의 RUNNABLE한 프로세스가 없을 경우, L2의 process를 스케줄링합니다.

-MLFQ라는 큐를 만든 후 pick_in_mlfq()라는 함수를 만들었다. 그 함수에서는 L0큐의 사이즈만큼 dequeue(0)를 해서 RUNNABLE한 프로세스를 찾는다. 만약 있다면 바로 그 큐에서 나온 인덱스를 리턴하고, L0큐의 사이즈만큼 순회했을 때 없다면 L1큐의 사이즈만큼 dequeue(1)를 하고 똑같이 RUNNABLE한 프로세스를 찾는다

```
int pick_in_mlfq()
{ // 큐에서 MLFQ 를 통해 process의 !INDEX!를 뽑는다.
  // L0 큐
  int proc_index = -1; // 뽑힌 index
  int sz = mlfq[0].size;
  for (int i = 0; i < sz; i++)//L0의 사이즈만큼
  {
    proc_index = dequeue(0); // dequeue의 파라미터는 level 즉 dequeue(0)은 L0큐에서 뽑겠다는 소리.
    if (ptable.proc[proc_index].state == RUNNABLE) //상태가 runnable이면
    {
      ptable.proc[proc_index].level = 0; //뽑힐테니까 level을 0로 설정해주고
      return proc_index; //그 인덱스 리턴
    }
    enqueue(0, proc_index); //runnable이 아니면 다시 큐에 넣어준다.
  }

  // L1 큐
  sz = mlfq[1].size;
  for (int i = 0; i < sz; i++) //L1의 사이즈만큼
  {
    proc_index = dequeue(1); // dequeue의 파라미터는 level 즉 dequeue(0)은 L0큐에서 뽑겠다는 소리.
    if (ptable.proc[proc_index].state == RUNNABLE)//runnable이면
    {
      ptable.proc[proc_index].level = 1; //레벨 다시 1로 세팅
      return proc_index;//인덱스 리턴
    }
    enqueue(1, proc_index); //runnable이 아니면 다시 L1에넣는다.
  }
```

pick_in_mlfq()에서 L0, L1 큐에서 뽑는 과정

4. L2큐는 priority 스케줄링을 합니다.우선순위를 설정할 수 있는 시스템콜인 setPriority() 시스템콜이 추가되어야 합니다. Priority는 프로세스가 처음 실행될 때 3으로 설정되며, setPriority() 시스템콜을 통해 0~3 사이의값을 설정 할 수 있습니다. Priority값이 작을수록 우선순위가 높습니다. 우선순위가 같은 프로세스끼리는 FCFS로 스케줄링 됩니다.

우선순위는 L2 큐에서만 스케줄링에 영향을 줍니다.L2 큐에서 실행된 프로세스가 L2에서의 timequantum을 모두 사용한 경우, 해당 프로세스의 priority 값이 1감소하고 time quantum은 초기화됩니다. Priority 값이 0일 경우, 더 이상 값을 감소시키지않고 0으로 유지합니다.

-Priority는 처음 allocproc()함수에서 p->priority = 0으로 초기화된다. 그리고 다시 pick_in_mlfq()함수에서 L0, L1까지 봤는데도 없다면 L2의 큐 사이즈만큼 찾기 시작하는데 먼저 priority 순으로 찾아야 하기 때문에 minPriority라는 변수를 만들고 4로 세팅해준다. 그리고 runnable하면서 p->priority가 minPriority보다 작은 process를 찾으면 minPriority를 그 프로세스의 priority로 바꿔서 L2 안의 가장 작은 priority로 갱신을 한다. 그런 다음 다시 L2를 순회하면서 minPriority이면서 가장 '먼저' 찾은 프로세스의 인덱스를 리턴해준다.

```
// L2 큐
  sz = mlfq[2].size;//L2의 사이즈만큼
  int minPriority = 4;//가장 작은 priority를 뽑을 예정. max값인 3보다 큰 4로 세팅
  for (int i = 0; i < mlfq[2].size; i++) // 현재 L2큐에서 RUNNABLE하면서 priority가 가장 낮은 애를 찾는 과정
  {
    proc_index = dequeue(2);//뽑고
    if (ptable.proc[proc_index].state == RUNNABLE && ptable.proc[proc_index].priority < minPriority)
    {                               // RUNNABLE하고 현재 minspriority더 낮은 애들
      minPriority = ptable.proc[proc_index].priority; // 현재 L2에서 가장 낮은 priority
    }
    enqueue(2, proc_index); //아니면 다시 넣는다.
  }
  for (int i = 0; i < sz; i++) //다시 L2의 사이즈만큼
  {
    proc_index = dequeue(2); //뽑고
    if (ptable.proc[proc_index].state == RUNNABLE) //runnable인 process중에서
    {
      if (ptable.proc[proc_index].priority == minPriority) //아까 뽑은 minPriority로 처음 나오는 process를
      {
        ptable.proc[proc_index].level = 2; //레벨 2로 다시 세팅하고
        return proc_index; //proc_index 리턴
      }
    }
    enqueue(2, proc_index); //
  }
  return proc_index;
}
```

pick_in_mlfq()에서 L2

Timequantum을 모두 사용한 경우, priority값을 감소시켜주고, timequantum은 다시 0으로 초기화 해주는 부분은 scheduler()내부에서 프로세스를 뽑고 timequantum과 level*2+4를 비교해주는 부분에서 구현을 해주었다.

5. Priority boosting

Global tick이100 ticks가 될 때마다 모든 프로세스들은 L0 큐로 재조정됩니다.

Priority boosting이될 때, 모든 프로세스들의 priority 값은 3으로 재설정됩니다.

Priorityboosting이될 때, 모든 프로세스들의 time quantum은 초기화 됩니다.

-Global tick이 100일 될때에 trap.c의 ticks++하는 부분 바로 뒤에서 tick이 100이 넘어가는지 확인을 하고 그 부분에서 바로 Priority_boosting()호출했다. Priority_boosting함수는 먼저 L1에 있는 모든 프로세스를 다시 L0로 넣어주고, L2에 있는 모든 프로세스를 L0로 넣어주었다. 그 후 L0 큐를 순회하면서 timequantum을 0으로, level도 0으로, priority도 다시 3으로 초기화 해주었다.

```
case T_IRQ0 + IRQ_TIMER:
  if (cpuid() == 0)
  {
    acquire(&tickslock);
    ticks++;
    if (ticks >= 100)
    {
      isLocked = 0;        // global tick 100되면 풀어줘야함.
      ticks = 0;           // tick 초기화
      priority_boosting(); // priority boosting호출
    }
    wakeup(&ticks);
    release(&tickslock);
  }
```

global tick이 100이 될 때

```
void priority_boosting() // PRIORITY BOOSTING
{
  struct proc *p = myproc();
  int index;
  int sz = mlfq[1].size;
  for (int i = 0; i < sz; i++) // L1 큐 사이즈만큼
  {
    index = dequeue(1); // L1에서 빼서
    enqueue(0, index);  // L0에 넣는다.
  }
  sz = mlfq[2].size;
  for (int i = 0; i < sz; i++) // L2 큐 사이즈만큼
  {
    index = dequeue(2); // L2에서 빼서
    enqueue(0, index);  // L0에 넣는다.
  }
  sz = mlfq[0].size; //L0의 사이즈만큼
  for (int i = 0; i < sz; i++)
  {
    index = dequeue(0); //빼서 인덱스를 찾고
    p = &ptable.proc[index]; //process할당하고
    p->timequantum = 0; // timequnatum 0으로 할당하고
    p->level = 0; //level도 다시 0으로 넣고
    p->priority = 3; // priority도 다시 3으로 세팅
    enqueue(0, index); // 모든 과정 거친 후 L0에 넣는다.
  }
  return;
}
```

Priority_boosting()에서는 L1의 사이즈만큼 L0에 넣고, L2의 사이즈만큼 L0에 넣는다.
그리고 L0의 사이즈만큼 순회하면서 timequantum을 다시 0으로, priority도 3으로, level도 0으로 재조정한다. 그리고 다시 L0에 넣어준다.

6. MLFQ 스케줄러에 의해 스케줄링되는 프로세스보다 항상 우선적으로 처리되어야 하는 프로세스가 있을 수 있습니다. 이를 위해 스케줄러 lock/unlock을 가능하게 하는 schedulerLock(), schedulerUnlock() 시스템콜이 구현되어야 합니다. 스케줄러를 lock하는 프로세스가 존재할 경우 MLFQ 스케줄러는 동작하지 않고, 해당프로세스가 최우선적으로 스케줄링되어야 합니다.

```
23    static void wakeup1(void *chan);
24    int pick_in_mlfq();
25    void priority_boosting();
26    int isLocked = 0;            // SCHEDULER의 LOCK 유무
27    int pickedProcessIndex = 0; // queue에서 뽑은 process
28    int noSuchPid = 1;           // 그런 PID 또 없습니다~
29    extern uint ticks;
30    void printProcessInfo(struct proc *p);
31    void pinit(void)
```

int isLocked

-먼저 isLocked라는 변수를 proc.c에 선언을 해주고

```
void scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;
  for (;;)
  {
    // Enable interrupts on this processor.
    sti();
    acquire(&ptable.lock);
    if (isLocked == 0) // 잠기지 않았다면
    {
      pickedProcessIndex = pick_in_mlfq();
      if (ptable.proc[pickedProcessIndex].state != RUNNABLE)
      {
        release(&ptable.lock); // 다시 for문 돌아와야 하니까 re
        continue;
      }
    }
```

if (isLocked==0)으로 구분

-scheduler()함수에서 isLocked의 유무를 확인해서 pick_in_mlfq()함수를 호출하도록 한다. 그리고 schedulerLock()함수가 암호가 맞아서 호출이 된다면 안에서 isLocked를 1로 바꿔주고, schedulerUnlock이 호출이되거나, priority_boosting이 된다면 isLocked를 0으로 바꿔준다.

(schedulerLock과 Unlock의 코드는 아래에 있습니다.)

7. schedulerLock() 시스템콜은 '프로세스가 우선적으로 처리되어야할 자격을 증명'하기 위한 암호를 인자로 받습니다. 암호는 자신의 학번으로 합니다. 성공적으로 실행되었다면, global tick은 priority boosting 없이 0으로 초기화됩니다. schedulerLock() 시스템콜은 기존에 존재하는 프로세스만 호출할 수 있습니다.

```
1  void schedulerLock(int password)
2  {
3    struct proc *p = myproc();
4    acquire(&ptable.lock);
5    if (myproc()->state != RUNNING || isLocked == 1) // state가 RUNNING이 아니거나, 먼저 걸려있었거나
6    {
7      release(&ptable.lock);
8      return;
9    }
10   if (password == 2019073181) // 비밀번호도 걸려있어야 하고 락도 안걸려있나 확인해야함.
11   {
12     if (cpuid() == 0)
13     {
14       acquire(&tickslock);
15       release(&ptable.lock);
16       ticks = 0;
17       wakeup(&ticks);
18       release(&tickslock);
19       acquire(&ptable.lock);
20     }
21     isLocked = 1;
22     release(&ptable.lock);
23     return;
24   }
25   else
26   {
27     cprintf("password error cur process's pid is %d, timequantum is %d, Queue level is %d\n", p->pid, p->timequantum, p->level);
28     release(&ptable.lock);
29     //kill(myproc()->pid);
30     exit();
31   }
32 }
```

schedulerLock()

schedulerLock함수는 현재 돌고있는 프로세스에서 호출되어야 하므로, state가 RUNNING이지 않은 상황이거나, isLocked==1로 먼저 잠겨있는 상황에서 또 잠구려고 하면 다시 return을 한다.
그 후 password를 확인 후 맞다면 tickslock을 걸고 ticks를 0으로 초기화하고, isLocked변수를 1로 만들고, 리턴한다. 이대로 리턴한다고 하면, scheduler내부에서는 pick_in_mlfq()함수가 돌아가지 않기 때문에 원래 뽑혔던 pickedProcessIndex 변수로 계속 process가 유지될 것이다.

8. schedulerUnlock() 시스템콜은 '우선적으로 처리되어야할 프로세스자격을 해제'하기위한 암호를 인자로 받습니다. 암호는 자신의 학번으로 합니다. 성공적으로 실행되었다면, 기존의MLFQ 스케줄러로 돌아갑니다.

해당 프로세스는 L0 큐의 제일 앞으로 이동하고, priority는 3으로 설정합니다. 해당프로세스의 time quantum은 초기화됩니다.

-schedulerUnlock함수에서도 RUNNING이 아니거나 isLocked가 0이 아니면 리턴한다. 또한 password를 확인하고 맞다면 해당 프로세스를 L0로 넣어주고, level과 timequantum과 priority를 초기화해주고 isLocked를 0으로 만들어준다.

```
1  void schedulerUnlock(int password)
2  {
3    struct proc *p = myproc();
4    acquire(&ptable.lock);
5    if (myproc()->state != RUNNING ||isLocked == 0)
6    {
7      release(&ptable.lock);
8      return;
9    }
10
11   if (password == 2019073181) // 비밀번호도 맞아야 하고 락도 걸려있어야 한다.
12   {
13     enqueue(0, p - ptable.proc);
14     p->level = 0;
15     p->timequantum = 0;
16     p->priority = 3;
17     isLocked = 0; // scheduler 풀어줘야함.
18     release(&ptable.lock);
19     return;
20   }
21   else
22   {
23     cprintf("password error cur process's pid is %d, timequantum is %d, Queue level is %d\n", p->pid, p->timequantum, p->level);
24     release(&ptable.lock);
25     // kill(myproc()->pid);
26     exit();
27   }
28 }
```

schedulerUnlock()

9. 두 시스템콜 호출시, 암호가 일치하지 않으면 해당프로세스를 강제로 종료합니다. 이때프로세스의pid, time quantum, 현재 위치한큐의 level을 출력합니다.

-schedulerLock과 schedulerUnlock함수에서 password가 일치하지 않으면 현재 프로스의 pid, timequantum과 level을 출력하고 ptable lock을 해제하고, exit()함수로 강제종료한다.

10. 두 시스템콜은 인터럽트를 통해 실행될 수 있어야 합니다.129번 인터럽트 호출시, schedulerLock() 시스템콜을 실행합니다. 130번 인터럽트 호출시, schedulerUnlock() 시스템콜을 실행합니다

-tvinit()함수에서 SETGATE함수를 통해서 권한을 DPL_USER로 낮춰주어서 시스템콜로 인터럽트를 호출할 수 있게 해주었다. 그리고 trap함수에서 tf->trapno가 129와 130인 경우로 나누어서 schedulerLock함수와 schedulerUnlock()함수를 호출하게 했다.

```
void tvinit(void)
{
  int i;

  for (i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE << 3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE << 3, vectors[T_SYSCALL], DPL_USER);
  SETGATE(idt[129], 1, SEG_KCODE << 3, vectors[129], DPL_USER);
  SETGATE(idt[130], 1, SEG_KCODE << 3, vectors[130], DPL_USER);

  initlock(&tickslock, "time");
}

void idtinit(void)
{
  lidt(idt, sizeof(idt));
}

// PAGEBREAK: 41
void trap(struct trapframe *tf)
{
  if (tf->trapno == T_SYSCALL)
  {
    if (myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if (myproc()->killed)
      exit();
    return;
  }
  switch (tf->trapno)
  {
  case 129:
    schedulerLock(2019073181);
    break;
  case 130:
    schedulerUnlock(2019073181);
    break;
```

schedulerLock/Unlock in Trap.c

11. SYSTEM CALL 구현

```
1  int getLevel(void)
2  {
3    return myproc()->level;
4  }
5
6  void setPriority(int pid, int priority)
7  {
8    struct proc *p;
9    acquire(&ptable.lock);
10   for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
11   {
12     if (p->pid == pid)
13     {
14       p->priority = priority;
15     }
16   }
17   release(&ptable.lock);
18 }
```

getLevel과 setPriority함수를 proc.c에 구현해주었고, setPriority는 ptable을 순회하면서 해당 pid가 나왔을 때 priority를 변경해주는 식으로 구현했다. Yield는 원래 구현되어있어서 캡쳐는 하지 않았고, schedulerLock과 Unlock은 위에 캡쳐해놓았다. 처음에 schedulerLock/Unlock을 어떻게 구현해야 하는지 고민이 많았다. 학번값을 인자로 주려면 시스템콜로 구현해야하고, 인터럽트로 구현하게 되면 인자값을 주지 못해서 어떻게 하나 고민하던 차에, 인터럽트로 구현할 때 고정으로 맞는 학번으로 주면 되겠구나 라고 생각을 하고 두가지로 구현했다.

```c
int sys_yield(void) {
  yield();
  return 1;
}

int sys_getLevel(void) {
  return getLevel();
}

int sys_setPriority(void) {
  int pid, priority;
  if(argint(0, &pid)<0){
    return -1;
  }
  if(argint(1, &priority)<0||!(priority>=0 && priority<=3)){//인자를 받아오는데 실패했거나, 0과 3사이의 값이 아닌경우
    return -1;
  }
  setPriority(pid, priority);
  return 1;
}

int sys_schedulerLock() {
  int password;
  if (argint(0,&password)<0) {
    return -1;
  }
  schedulerLock(password);
  return 1;
}

int sys_schedulerUnlock() {
  int password;
  if (argint(0,&password)<0) {
    return -1;
  }
  schedulerUnlock(password);
  return 1;
}
```

sysproc.c에 구현한 wrapper function들이다. Sys_yield와 sys_getLevel은 그대로 호출만 해주었고, setPriority에서는 인자를 받아오는데 실패한 경우는 -1을 리턴해주었고, priority가 0과 3사이가 아닌 경우에도 -1을 리턴하는 식으로 예외처리를 해주었다. 또한 schedulerLock/Unlock도 argint로 password를 잘 받아오는지 확인했다.

xv6-public > C user.h > ...

```c
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int myfunction(char*);
void yield(void);
int getLevel(void);
void setPriority(int, int);
void schedulerLock(int);
void schedulerUnlock(int);
```

```
195   int             getLevel(void);
196   void            setPriority(int pid, int priority);
197
198   void            mycall(void);
199
200   //queue.c
201   int            enqueue(int, int);
202   int             dequeue(int);
203   int             is_full(int);
204   int             is_empty(int);
205   void            qinit(void);
206   void            printQinfo(int);
207
208
209   void            schedulerLock(int);
210   void            schedulerUnlock(int);
211   void            priority_boosting(void);
```

User.h와 defs.h에도 선언을 해주어서 사용자 프로그램에서 호출을 할 수 있게 해주었다.

```
107   extern int sys_yield(void);
108   extern int sys_getLevel(void);
109   extern int sys_setPriority(void);
110   extern int sys_schedulerLock(void);
111   extern int sys_schedulerUnlock(void);
```

```
136   [SYS_yield] sys_yield,
137   [SYS_getLevel] sys_getLevel,
138   [SYS_setPriority] sys_setPriority,
139   [SYS_schedulerLock] sys_schedulerLock,
140   [SYS_schedulerUnlock] sys_schedulerUnlock,
```

```
33    SYSCALL(yield)
34    SYSCALL(getLevel)
35    SYSCALL(setPriority)
36    SYSCALL(schedulerLock)
37    SYSCALL(schedulerUnlock)
```

각각 syscall.h과 usys.S에도 선언을 해주었다.

Result

컴파일 및 실행과정

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
```

Command에 make, make fs.img, ./bootxv6.sh 를 연속적으로 입력하여 부팅하였다.

```
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16480
echo           2 4 15336
forktest       2 5 9640
grep           2 6 18696
init           2 7 15916
kill           2 8 15364
ln             2 9 15216
ls             2 10 17848
mkdir          2 11 15460
rm             2 12 15440
sh             2 13 28080
stressfs       2 14 16352
usertests      2 15 67456
wc             2 16 17216
zombie         2 17 15028
my_userapp     2 18 15308
testcode       2 19 18460
console        3 20 0
```

Ls를 입력했을 때의 화면이다. init프로세스와 shell프로세스가 잘 동작하는 모양이다.

```
$ testcode
MLFQ test start
[Test 1] default
Process 5
L0: 8605
L1: 17260
L2: 74135
L3: 0
L4: 0
Process 6
L0: 8793
L1: 16963
L2: 74244
L3: 0
L4: 0
Process 7
L0: 12171
L1: 25281
L2: 62548
L3: 0
L4: 0
Process 8
L0: 12500
L1: 25384
L2: 62116
L3: 0
L4: 0
[Test 1] finished
done
$
```

Piazza에 올라온 testcode를 해본 결과 화면이다.

프로세스가 L0, L1, L2큐에 생성된 비율이 1 : 2 : 7 정도 되는걸로 보아 타임퀀텀이 가장 짧은 L0와 그 다음으로 짧은 L1, 그리고 가장 많이 존재할 수 밖에 없는 L2에 잘 들어간 것으로 보이고, priority_boosting과 timequantum계산이 정상적으로 작동한 것 같다.

setPriority Test

```
case '2':
  printf(1, "[Test 2] priorities\n");
  pid = fork_children2();

  if (pid != parent)
  {
    for (i = 0; i < NUM_LOOP; i++)
    {
      int x = getLevel();
      if (x < 0 || x > 2)
      {
        printf(1, "Wrong level: %d\n", x);
        exit();
      }
      count[x]++;
    }
    printf(1, "Process %d\n", pid);
    for (i = 0; i < MAX_LEVEL; i++)
      printf(1, "Process %d , L%d: %d\n", pid, i, count[i]);

  }
  exit_children();
  printf(1, "[Test 2] finished\n");
  break;
```

Set Priority를 테스트하는 코드이다. Fork_children2를 호출하면 그 안에서 setPriority가 호출된다.

```
$ testcode 2
MLFQ test start
[Test 2] priorities
before sleep pid: 4, lv: 0
before sleep pid: 5, lv: 0
after sleep pid: 4, lv: 0
after sleep pid: 5, lv: 0
before sleep pid: 6, lv: 0
after sleep pid: 6, lv: 0
before sleep pid: 7, lv: 0
after sleep pid: 7, lv: 0
Process 4
Process 4 , L0: 6006
Process 4 , L1: 11200
Process 4 , L2: 82794
Process 5
Process 5 , L0: 9228
Process 5 , L1: 23895
Process 5 , L2: 66877
Process 6
Process 6 , L0: 8139
Process 6 , L1: 17248
Process 6 , L2: 74613
Process 7
Process 7 , L0: 7101
Process 7 , L1: 16234
Process 7 , L2: 76665
[Test 2] finished
done
```

Testcode2는 잘 호출되고, 프로세스의 큐 비율은 1 : 2 : 7 정도 나왔다.

Yield Test

```
case '3':
  printf(1, "[Test 3] yield\n");
  pid = fork_children2();

  if (pid != parent)
  {
    for (i = 0; i < NUM_YIELD; i++)
    {
      int x = getLevel();
      if (x < 0 || x > 2)
      {
        printf(1, "Wrong level: %d\n", x);
        exit();
      }
      count[x]++;
      if (pid % 2 == 0) yield();
    }
    printf(1, "Process %d\n", pid);
    for (i = 0; i < MAX_LEVEL; i++)
      printf(1, "Process %d , L%d: %d\n", pid, i, count[i]);

  }
  exit_children();
  printf(1, "[Test 3] finished\n");
  break;
```

Yield systemcall을 호출하는 코드이다. Pid가 짝수인 프로세스들은 yield를 호출한다.

```
$ testcode 3
MLFQ test start
[Test 3] yield
before sleep pid: 9, lv: 0
before sleep pid: 10, lv: 0
after sleep pid: 10, lv: 0
after sleep pid: 9, lv: 0
before sleep pid: 11, lv: 0
Process 9
Process 9 , L0: 3240
Process 9 , L1: 11982
Process 9 , L2: 4778
after sleep pid: 11, lv: 0
before sleep pid: 12, lv: 0
Process 11
Process 11 , L0: 5037
Process 11 , L1: 13278
Process 11 , L2: 1685
Process 10
Process 10 , L0: 12
Process 10 , L1: 24
Process 10 , L2: 19964
after sleep pid: 12, lv: 0
Process 12
Process 12 , L0: 11
Process 12 , L1: 24
Process 12 , L2: 19965
[Test 3] finished
done
```

Yield test code를 돌린 결과이다. 짝수 프로세스 10과 프로세스 12는 확연히 L0,L1의 비율이 적고, L2의 비율이 높게 나온 것을 확인할 수 있다.

SetpriorityTest

```
case '6':
  printf(1, "[Test 6] setPriority return value\n");
  child = fork();

  if (child == 0)
  {
    // int r;
    int grandson;
    sleep(10);
    grandson = fork();
    if (grandson == 0)
    {
      setPriority(getpid() - 2, 0);
      setPriority(getpid() - 3, 0);
    }
    else
    {
      setPriority(grandson, 0);
      setPriority(getpid() + 1, 0);
    }
    sleep(20);
    wait();
  }
  else
  {
    int child2 = fork();
    sleep(20);
    if (child2 == 0)
      sleep(10);
    else
    {
      setPriority(child, -1);
      setPriority(child, 11);
      setPriority(child, 10);
      setPriority(child + 1, 10);
      setPriority(child + 2, 10);
      setPriority(parent, 5);
    }
  }

  exit_children();
  printf(1, "done\n");
  printf(1, "[Test 6] finished\n");
  break;
```

setPriority systemcall이 호출이 잘 되는지 test하는 코드이다. Pid를 주고, priority에 랜덤값을 줘서 잘 호출되는지 test한다.

```
$ testcode 6
MLFQ test start
[Test 6] setPriority return value
done
[Test 6] finished
done
```

setPriority testcode의 결과이다. 오류 없이 잘 수행된 결과이다.

schedulerLock& Unlock Test

```
case '7':
  printf(1, "[Test 7] schedulerLock & Unlock Test\n");
  child = fork();

  if (child == 0) // child
  {
    int grandson;
    printf(1, "Before Lock Process : %d \n", getpid());
    schedulerLock(PASSWORD);
    printf(1, "After Lock Process : %d \n", getpid());
    grandson = fork();
    if (grandson == 0) // gradnsond
    {
      printf(1, "GRANDSON : %d\n", getpid());
      printf(1, "Before try Unlock Process : %d \n", getpid());
      schedulerUnlock(PASSWORD);
      printf(1, "After try Unlock Process : %d \n", getpid());
      schedulerLock(PASSWORD);

    }
    else // child
    {
      printf(1, " CHILD : %d\n", getpid());
      printf(1, "Before try Unlock Process : %d \n", getpid());
      schedulerUnlock(PASSWORD);
      printf(1, "After try Unlock Process : %d \n", getpid());
      printf(1, "Process : %d still work \n", getpid());
    }
    sleep(20);
    wait();
  }
  else // parent
  {
    printf(1, "Else\n");

    int child2 = fork();
    sleep(20);
    if (child2 == 0) {//child2
      printf(1, " CHILD2 : %d\n", getpid());

      printf(1, "Before try Lock Process : %d \n", getpid());

      schedulerLock(PASSWORD);
      printf(1, "After try Lock Process : %d \n", getpid());
      sleep(2);
      for (int i = 0;i < NUM_SLEEP * 1000000;i++) {
        if (i % 100000000 == 0) {
          printf(1, "spend time for priority boosting while running\n");
        }
      }
      printf(1, "Before try Lock Again Process : %d \n", getpid());

      schedulerLock(PASSWORD);
      printf(1, "After try Lock Again Process : %d \n", getpid());
    }
    else
    {
      printf(1, " PARENT : %d\n", getpid());
      schedulerLock(PASSWORD);
      printf(1, "test before for sleep priority Boost\n");
      sleep(100);
      printf(1, "test after for sleep priority Boost\n");

    }
  }
  printf(1, "Process : %d\n", getpid());

  exit_children();
  printf(1, "Process : %d\n", getpid());
  printf(1, "[Test 7] finished\n");

  break;
```

schedulerLock과 Unlock을 테스트하는 코드이다.

```
$ testcode 7
MLFQ test start
[Test 7] schedulerLock & Unlock Test
Else
Before Lock Process : 28
After Lock Process : 28
 CHILD : 28
Before try Unlock Process : 28
Af PARENT : 27
test before for sleep priority Boost
test after for sleep priority Boost
Process : 27
```

schedulerLock & Unlock testcode의 결과이다. schedulerLock과 Unlock이 잘 되는 중이다가, Process의 번호를 출력하고 어떠한 출력도 되지 않는다. 아무래도 fork된 어떤 프로세스가 zombie가 되고 부모 프로세스가 회수해주지 못한 모습으로 보인다.

Trouble Shooting

1. 처음에 디버깅을 하면서 cprint문을 코드 내에 많이 넣어서 디버깅을 했는데, 콘솔 출력 자체가 하나의 프로세스라서 내가 원했던 결과가 잘 나오지 않았다. 특히 tick에 대한 정보를 cprint로 출력할 때 tick이 출력하면서도 변하게 되어서 내 예측과 달리 나오는 것에 대해서 난감했던 적이 많았다. 그래서 tick이 너무 빨리 지나가기 때문에 어느정도의 오차를 허용해야 함을 깨달아서 정확한 값을 분기로 하지 않고 크거나 같다 아니면 어느 범위 사이 이런식으로 오차를 생각해서 tick을 잡고 디버깅을 했다.

2. 그리고 pick_in_mlfq함수에서 뽑은 인덱스를 실제로 상태를 보니까 SLEEPING인 적이 있었다. 나는 분명히 pick_in_mlfq함수에서는 RUNNABLE한 프로세스를 뽑기로 했는데, 나온 프로세스의 상태가 SLEEPING인 것에 대해 상당히 놀랐었다. 하지만, 함수를 다시 천천히 보니까, L2까지도 지나간다면 SLEEPING인 상태가 나올 수 있었고, 그 상황이 sh 프로세스가 실행되고 입력을 기다리는 상황임을 알게 되었다. 그럴 때에는 어떤 프로세스가 나오는지 RR방식의 태초의 xv6코드를 보니 뽑을 프로세스가 없다면 다시 for문으로 돌아가서 무한루프를 돎을 깨닫게 되었다. 그래서 SLEEPING한 프로세스가 나온다면 다시 for문으로 올려보냈다.

3. 또한 과제를 시작하고 초반쯤에 hardware에서 부팅이 멈춘 적이 있었는데 디버깅을 할 수 없는 상황이라 정말 막막했다. 그래서 userinit()이 불리기 전에 에러가 난 것으로 판단해서 천천히 찾다가 mycpu()에 진입해서 코드의 순서가 섞여 있음을 발견했다. 내 코드는 mycpu()에 진입하자마자 panic을 호출하게 되어있었다. 언제 코드 순서가 바뀌어 있었는지 몰르겠지만, 그것을 원래의 위치로 돌리자마자 부팅이 완료되었다.

4. schedulerLock과 Unlock을 잘 구현을 했다고 생각했는데, 테스트코드에서 잘 동작하지 않았다. 그래서 Lock과 Unlock을 호출한 프로세스의 상태를 RUNNING인지 아닌지 확인하고, isLocked이 이중으로 걸리는지 풀리는지도 확인해봤고, acquire, release가 잘되었는지도 확인하고, 처음에는 kill(myproc()->pid)를 했었는데, exit()으로도 바꾸어보았는데, 잘 돌아가지 않는 것 같다…