

P1 Wiki: 2020028586 이혜민

Design

[Multi-level feedback queue](#)

[xv6 scheduler design overview](#)

[Design of MLFQ scheduler in xv6](#)

[Queue를 어떻게 구현할 것인가](#)

[System call](#)

[yield](#)

[getLevel](#)

[setPriority](#)

[Scheduler Lock, Unlock](#)

[동작 방식과 관련하여](#)

[각 함수의 구현](#)

[예외적인 케이스에 대한 디자인](#)

Implement

[MLFQ scheduler](#)

[proc 구조체 변경 및 상수 추가](#)

[MLFQ 자료구조와 Queue handling function 추가](#)

[MLFQ scheduling 알고리즘 구현](#)

[System calls](#)

[Scheduler Lock & Unlock](#)

[구현](#)

[호출](#)

[보조적인 기능](#)

Result

[컴파일 및 실행](#)

[MLFQ Scheduler](#)

[Required system call & interrupts](#)

[system call23: yield](#)

[system call24: getLevel](#)

[system call25: setPriority](#)

[system call26, 27 & interrupt 129, 130: schedulerLock, schedulerUnlock](#)

[SchedulerLock](#)

Trouble shooting

[xv6 구조 파악](#)

[디자인 관련](#)

[Queue 구현 아이디어 후보](#)

[MLFQ scheduling 구현 아이디어 후보](#)

[테스트 케이스 생성](#)

[에러](#)

[debugging](#)

[unexpected trap 14](#)

[하나의 프로세스는 1tick만 실행되던 문제](#)

[queue 내부에서 프로세스가 1tick마다 Round-Robin되지 않음](#)

[mycpu called with interrupts enabled](#)

[ubuntu system error](#)

[IDE 사용과 관련하여](#)

Design



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

Multi-level feedback queue

xv6 scheduler design overview

원래의 xv6는 Round-Robin scheduler를 이용합니다.

main.c의 `mpmain`에서는 일반적인 CPU의 setup code를 수행합니다. 여기서 proc.c의 `scheduler()`가 호출되면서, 프로세스가 실행되기 시작합니다.

scheduler 내부에서는 `for(;;)`를 통해 계속해서 무한 루프를 돌게됩니다. 무한 루프 안에서 ptable을 순회하면서, RUNNABLE한 프로세스가 발견되면 해당 프로세스의 context와 scheduler를 교환하면서 프로세스를 실행합니다.

```
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
        continue;
```

```

// 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;
switchvm(p);
p->state = RUNNING;

swtch(&(c->scheduler), p->context);
switchkvm();

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

```

이때, xv6에서는 1tick마다 timer interrupt가 발생합니다. 그러면 trap.c에서 해당 interrupt number를 통해 `ticks`를 1 증가시키고, process가 RUNNING 중이라면, `yield`를 통해 실행중이던 process의 state를 RUNNABLE로 전환하고, 다시 `sched`를 통해 scheduler에게 권한을 줘서(swtch를 통해 context를 switch하여 mycpu의 scheduler가 실행되도록 합니다) scheduling이 일어나게 됩니다. 이때, scheduler 내부에서는 이전에 실행하던 것에 이어서 실행이 되게 됩니다. 그 다음 RUNNABLE한 프로세스를 선택하여 다시 context switch를 통해 실행합니다. 이 과정이 반복됩니다.

```

// 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();

```

결과적으로, 1tick(RR의 time quantum)마다 timer interrupt가 발생하면 yield하여 실행중이던 프로세스를 다시 ready queue에 넣고, 그 다음 프로세스를 1tick만큼 실행하는 ... 이러한 과정을 반복합니다. 이로써 결과적으로 ptable을 순회하며, Round Robin 스케줄링이 일어납니다.

여기까지가 기존의 xv6 scheduler 동작에 대한 대략적인 설명입니다.

Design of MLFQ scheduler in xv6

MLFQ(Multi-Level Feedback Queue) 스케줄러는 명세 기준 세 개의 레벨로 구성되어 있으며, 각 레벨에서는 다른 스케줄링 알고리즘인 Round Robin과 Priority scheduling이 사용됩니다. L0와 L1에서는 Round Robin 방식이, L2에서는 Priority 방식이 이용되며, 같은 Priority를 가지는 경우에는 FCFS(First-Come, First-Served)로 스케줄링이 이루어집니다. 처음 scheduler는 CPU를 할당해줄 프로세스를 스케줄링하기 위하여 L0, 즉 가장 작은 level의 Queue부터 순회하면서 각 level의 스케줄링 정책에 따라 프로세스를 선택합니다.

이를 위해, 프로세스를 저장하고 있을 3개의 Queue의 구현이 필요합니다. xv6에서 maximum number of processes는 64이므로, 각 Queue는 `proc*[64]` 형태의 배열로 구성됩니다.

qtable은 qtable에 대한 데이터 동기화를 위한 lock과, 3개 level의 Queue(`queue[3]`)를 가지고 있습니다.

각 프로세스는 n-level의 Queue에서 최대 $2n + 4$ 의 time quantum을 가집니다. time quantum을 초과하면, 한단계 높은 level의 queue로 이동합니다. 이 미 L2, 즉 가장 높은 레벨의 Queue에 있는 상태라면, Priority를 하나 감소시켜서 starvation을 방지합니다(다만 이는 아직 완벽한 해결책은 아닙니다). 이 방법을 사용하여도 여전히 starvation이 일어날 수 있기에, global tick이 100이 되면 L1, L2에 있던 프로세스를 다시 L0에 enqueue, 즉 priority boosting이 일어나게 합니다. 그리고 실행 시간을 초기화 시킵니다.

이를 위해, process의 메타 데이터를 저장하는 `proc` 구조체에 `priority`, `arrivedTime`, `execTime`, `qLevel` 멤버 변수를 추가합니다.

scheduler가 동작하면, 우선 이전에 진행되던 것에 이어서 진행이 되게 되므로, 무한 루프 안의 qLevel을 순회하는 루프 안에서 현재의 qLevel보다 작은 qLevel에 새로 들어온 process가 있는지 확인합니다. 있다면, 더 낮은 level의 프로세스를 선택해서 실행해야 할 것입니다. 이러한 방법을 통해 L0, 실행할 수 있는 프로세스가 없으면 L1, L1에도 없으면 L2의 프로세스가 실행되도록 할 수 있습니다. 즉, qLevel에 의거하여 선택가능해집니다.

L0, L1에서 Round-Robin이 진행되도록 하기 위해, 우선 스케줄러는 L0(L0가 없다면 L1)에서 dequeue한다고 가정했을 때 실행하게 되는 프로세스를 잡아서 실행합니다. 1tick이 지나 timer interrupt가 발생했을 때, yield로 진입하면 이전에 실행하던 프로세스를 dequeue하여 다시 queue의 끝으로 enqueue합니다. 그후, scheduler가 실행되면, 아까 실행됐던 프로세스는 queue의 맨 뒤에 가있게 되고, 그다음 프로세스를 실행하게 됩니다.

또한, 적합한 프로세스를 뽑는 과정에서, 만약 time quantum을 넘긴 프로세스가 있다면 그 프로세스는 dequeue되어 queue level을 조정하거나 priority를 조정하는, 해당 queue level에 맞는 정책을 적용해줍니다. 그 후 다시 적절한 프로세스를 찾게 됩니다.

global tick이 100이 되어 priority boosting이 일어나면, L0에 있던 것은 그대로 L0에 있고, L1에 있던 프로세스를 먼저 L0에 enqueue해주고, L2에 있던 프로세스를 순서대로 L0에 enqueue합니다.



L2의 time quantum과 관련하여, L2에서 스케줄링 되어야하는 때는 계속해서 가장 낮은 priority를 가지고 있는게 선택되고, 그 중에서도 FCFS에 의해 가장 일찍 온 프로세스만이 지속적으로 선택되게 됩니다. 단, 완료가 보장되는 것은 아니며, timer interrupt에 걸리면 또 새롭게 scheduling 할 것을 찾다가 더 낮은 priority 및 arrivedTime을 갖는 것이 있다면 해당 프로세스를 스케줄링합니다(즉, preemptive). 이때 발생할 수 있는 Starvation은 Priority boosting을 통해 해결합니다.
Piazza와 일부 다른 부분에 대하여 채점에 참고 부탁드립니다.)

Queue를 어떻게 구현할 것인가

Queue 자료 구조를 구현하는 방법으로 2가지 후보를 생각하였었습니다.

1. linked list로 구현하는 방법

하지만, xv6에서는 `malloc` 을 사용할 수 없습니다. `kmalloc` 은 page 단위로, 4kb씩 할당하기 때문에 필요 이상의 크기를 할당하게 되므로, 자원 낭비가 심해질 것입니다.

2. xv6에서는 ptable의 최대 길이가 64개로 제한됩니다. 즉, 64개의 `struct proc*` 데이터를 가지고 있을 수 있습니다. 따라서, length가 64인 배열을 이용하여 한 level의 queue를 만들 수 있습니다.

동적 할당되는 linked list 대신 크기가 고정적인 2번 구현 방법을 채택하였습니다. enqueue를 하면 현재 rear를 증가시키고, 가장 최근에 빈 곳에 process를 넣습니다. dequeue를 하게 되면, 0번째 배열에서 process 정보를 뺀 뒤, rear를 감소시키고, 배열의 원소를 하나씩 앞당깁니다.

또한, queue와 관련된 동작을 할때(enqueue, dequeue, delete, frontenqueue 등)는, qLevel input이 적절하지 않거나, 해당 qLevel의 프로세스가 64개를 초과하거나, RUNNABLE하지 않은 process를 enqueue하는 등의 동작이 제한됩니다.

System call

yield

기존의 yield와 유사하게 진행하되, timer interrupt가 발생할 때마다 각 queue 안에서 round되도록 dequeue와 enqueue가 이루어지는 것을 추가합니다.

yield를 호출한다는 것은, 다음 프로세스에게 프로세서를 양보하고, 자기 자신은 자기가 원래 속하던 queue의 맨 뒤로 다시 가겠다는 것을 의미합니다.

getLevel

process 구조체에 `qLevel` 을 저장해둡니다. enqueue될 때, 항상 process의 qLevel도 알맞게 변경시켜줌으로써 올바른 qLevel을 가지고 있을 수 있습니다. 그리고 getLevel system call이 호출되면, proc에 저장된 `qLevel` 을 반환하게 합니다.

setPriority

ptable을 순회하면서 특정한 input pid에 해당하는 process를 찾고, 해당 process의 priority를 input 값으로 변경시켜줍니다.

Scheduler Lock, Unlock



time quantum과 관련된 내용이 나오기 전에 lock을 이미 구현하여서, schedulerLock은 1tick 씩 100번 할당 받는 방식이 아니라 100tick 동안 CPU를 차지하는 방식으로 구현되었습니다. Piazza와 일부 다른 부분에 대하여 채점에 참고 부탁드립니다.)

명세서에서 “MLFQ 스케줄러에 의해 스케줄링 되는 프로세스보다 항상 우선적으로 처리되어야 하는 프로세스가 있을 수 있습니다.”와, “스케줄러를 lock하는 프로세스가 존재할 경우 MLFQ 스케줄러는 동작하지 않고, 해당 프로세스가 최우선적으로 스케줄링 되어야 합니다.”를 봤을 때, MLFQ 스케줄러가 동작하지 않는다는 말이 MLFQ에 의해 스케줄링되는 프로세스는 무시되고 lock을 건 프로세스만을 동작한다는 뜻으로 받아들였기 때문에 이와 같이 디자인하였습니다. 즉, MLFQ 알고리즘에 의해서는 후순위가 된다고 해도, 이와는 전혀 상관없이 lock을 걸었기 때문에 무슨 일이 있어도 최우선적으로 실행되어야 한다는 것으로 받아들여 이를 토대로 디자인하였습니다. 즉, ‘최우선적으로 스케줄링되어야 하기 때문에’ 마치 L0 queue의 0번째에 고정되어 있는 것처럼 작동합니다.

또한, 다음의 Piazza 질의응답을 바탕으로 디자인하였습니다.

#1

예를 들어서 process A가 schedulerLock()이라는 system call을 호출한다고 하면

1. scheduler에 의해서 process A가 CPU를 잡게 되고
 2. schedulerLock에 의해서 A가 계속해서 CPU를 잡게 되므로
 3. 다른 process가 schedulerUnlock()이라는 system call을 호출하고 싶어도 할 수 없다.
- 와 관련한 답변을 바탕으로 디자인하였습니다.

#2

과제 명세서에 있는 "schedulerLock() 시스템 콜을 호출한 프로세스는 특정 조건을 만족할 경우 MLFQ로 돌아갑니다." 라는 조건은 해당 조건들을 만족하지 않으면 MLFQ가 작동하지 않고, 해당 프로세스만 계속 작동함을 의미한다.

동작 방식과 관련하여

1. scheduler 전체에 lock을 걸어주고, schedulerLock에서는 lock을 acquire하며, schedulerUnlock에서는 lock을 release하는 방법
2. scheduler 함수는 그대로 동작하되, 플래그를 통해 lock이 걸린 프로세스가 있는지 확인하는 방법.

2번의 방법으로 구현하였습니다. schedulerLock 함수에서 scheduler에 대한 lock을 acquire 해두고, schedulerUnlock 함수에서 scheduler에 대한 lock을 release하는 새로운 방식의 도입보다는 xv6 scheduler의 틀에 맞도록 sched를 통해 권한을 받는 기존 scheduler 함수에서 lock을 처리하게 합니다.

lock process만을 따로 저장하는 변수를 하나 뒤서, lock이 된 프로세스가 있다면 해당 process를 lock process에 저장해둡니다. lock process가 null(0)이 아니라면, scheduler에서는 항상 우선적으로 scheduling이 됩니다. schedulerLock은 오직 하나의 프로세스에서만 호출할 수 있기 때문입니다.

user는, `schedulerLock(password)` 나 `schedulerUnlock(password)` 시스템 콜을 호출하여 scheduler lock 및 unlock 기능을 이용할 수 있습니다. 단, 이때 올바른 password(2020028586)를 입력하여야만 lock이 동작하고, 틀린 비밀번호를 입력하게 되면 lock system call을 호출한 process는 kill됩니다.

각 함수의 구현

1. schedulerLock

schedulerLock을 호출하면, lock된 process가 이미 있는지 확인하고, 없다면 scheduler lock을 거는 로직이 실행됩니다. xv6의 scheduler에서는 lock을 건 프로세스가 있는지 확인하여 있다면 최우선적으로 스케줄링합니다. 성공하면 global tick을 0으로 초기화합니다.

2. schedulerUnlock

global tick이 100이 되거나, 명시적으로 호출되는 경우 실행되는 로직입니다. lock된 프로세스가 있다면, 해당 lock 플래그를 제거해주고, 다시 MLFQ scheduling이 일어날 수 있도록 변경됩니다.

3. scheduler에서의 변경 사항

lock된 프로세스가 있는지 항상 최우선적으로 확인해줍니다.

다만, 이렇게 플래그를 확인하는게 아니라, lock이 실행되면 인터럽트가 걸리고, 다른 스케줄러 함수를 만들어서 그 스케줄러가 실행되는 방법도 생각해 보았었는데, 예상치 못한 에러를 겪으면서 해당 방법으로는 구현하지 않았습니다.

예외적인 케이스에 대한 디자인

1. schedulerLock을 통해 우선적으로 스케줄링 되고 있는 상황에서 yield를 호출하는 경우, yield를 한다는 것 자체가 다음 프로세스에게 프로세서를 양보, 즉 더이상 우선적으로 스케줄링 되지 않도록 하겠다는 의미이므로, 다음과 같은 두가지 처리 방식 케이스를 생각해볼 수 있습니다.

- a. schedulerLock을 통해 우선적으로 실행되던 프로세스에서 yield를 호출하면, schedulerUnlock이 되면서 mlfq scheduling이 다시 동작하고, mlfq scheduler에 따라 다른 프로세스에게 yield됨.
- b. schedulerLock을 통해 실행되던 프로세스에서 호출되는 yield는 여전히 다음 프로세스로 schedulerLock된 프로세스를 실행하므로 yield의 호출 의미가 없음

좀더 근본적인 디자인 이유는 다음과 같습니다. yield를 호출하는 경우, 해당 Process는 CPU를 Scheduler한테 양보하게 되고, scheduler는 다음에 스케줄링할 프로세스를 찾습니다. 이때 다음 Process란, Scheduler가 다음으로 Scheduling해주는 Process를 의미하게 되는데, schedulerLock이 걸려 있으면 mlfq scheduler는 동작하지 않으며(과제 명세서) 앞서 언급한 디자인에 따라 schedulerLock이 걸린 프로세스가 계속해서 CPU를 잡고 있게 되므로, yield를 통해 다음 프로세스에게 양보를 해줬자 다음 스케줄링 되는 코드역시 lock된 프로세스이므로 yield를 호출해도 실행되는 프로세스에는 변화가 없을 것입니다.

더불어, yield는 별다른 암호 없이도 호출이 가능한 반면, schedulerLock이나 schedulerUnlock은 암호가 맞아야 실행되는 일종의 좀더 특권을 가진 함수라고 생각되었습니다. 그렇기에, 암호 입력도 안했는데 yield를 호출할 때 schedulerUnlock이 호출되는 것은 본래라면 암호까지 받아서 실행해야하는 schedulerUnlock의 취지에 적합하지 않다고 생각했습니다.

2. schedulerLock을 통해 우선적으로 처리되던 프로세스에서 또 schedulerLock을 호출하는 경우, 후반부의 schedulerLock은 무시됩니다.

3. schedulerLock이 걸려있던 프로세스가 unlock 호출 전에 sleep하거나 exit 하는 경우, 더이상 해당 lock된 프로세스는 CPU를 잡지 않으므로 (RUNNABLE하지 않으므로), CPU를 양보하게 됩니다. 이때, 먼저 unlock 로직이 실행됩니다.

4. schedulerLock된 프로세스가 없는데 schedulerUnlock을 호출하면, 아무런 일도 벌어지지 않습니다. 즉, 그냥 schedulerUnlock함수가 도중 종료됩니다.

5. PASSWORD가 틀린 경우

user program에서 schedulerLock 혹은 schedulerUnlock을 호출했는데 password가 틀린 경우, `[scheduler (un)lock] Wrong Password!` 라는 경고 문구와 함께 해당 system call을 호출한 process의 pid, time quantum(execution된 시간), process가 속한 queue level을 출력하고(과제 명세) kill됩니다.

Implement



본인이 새롭게 구현하거나 수정한 부분에 대해서 무엇이 기존과 어떻게 다른지, 해당 코드가 무엇을 목적으로 하는지에 대한 설명을 구체적으로 서술합니다.

MLFQ scheduler

proc 구조체 변경 및 상수 추가

1. proc 구조체 멤버 변수 추가

앞서 Design에서 설명한 바와 같이, 다음의 MLFQ 알고리즘을 위하여 멤버 변수를 추가하였습니다.

```
// member variables for mlfq
int priority;           // priority of process
int arrivedTime;       // arrived time
int execTime;          // time passed after execution
enum queueLevel qLevel; // queue level
```

추후 `schedulerLock` 을 구현하기 위한 멤버 변수 또한 추가합니다.

```
enum locked isLock;
```

2. 필요한 상수 추가

```
enum queueLevel { TOP = 0, MIDDLE, BOTTOM };
enum locked { UNLOCKED = 0, LOCKED };

#define TIME_QUANTUM(LEVEL) (2 * LEVEL) + 4
```

```
#define MAXQLEVEL    3 // max level of multi level queue
#define MAXPRIORITY  4 // max priority of process
```

MLFQ 자료구조와 Queue handling function 추가

- MLFQ 자료구조

```
struct mlfQueue {
    // RUNNABLE한 프로세스만 저장하는 큐
    int rear;
    struct proc* procsQueue[NPROC];
};
```

```
struct
{
    struct spinlock lock;
    struct mlfQueue mlfQueue[MAXQLEVEL];
} qtable;
```

enqueue를 하면 현재 rear를 증가시키고, 가장 최근에 빈 곳에 process를 넣습니다. dequeue를 하게 되면, 0번째 배열에서 process 정보를 뺀 뒤, rear를 감소시키고, 배열의 원소를 하나씩 앞당깁니다.

또한, qtable에서는 정보를 동기화하기 위한 lock이 쓰이며, MAXQLEVEL만큼의 queue가 생성됩니다(여기서는 3-level의 queue array를 가짐).

- queue 자료 구조를 handling하기 위한 함수들을 구현했습니다.

```
// qtable의 lock을 initialize
void      qinit(void);

// queue에서 특정 pid를 가진 프로세스를 찾는다
int      findFromQueueByPid(int pid, struct qLocation* qLoc);

// `qLevel`의 level의 queue에 enqueue
int      MLFQenqueue(struct proc* p, int qLevel);

// 추후 schedulerUnlock이 되고 나서 queue의 맨 앞에 enqueue하기 위한 특수한 함수
int      MLFQfrontEnqueue(struct proc* p, int qLevel);

// 추후 schedulerLock에서 lock된 프로세스를 queue 내부에서 manage하기 위한 특수한 함수
int      MLFQdeleteByPid(int pid);

// `qLevel`의 level의 queue에 dequeue
struct proc*      MLFQdequeue(int qLevel);
```

```
struct proc* MLFQfirstProc(int qLevel);
```

- `MLFQfirstProc`은 queue에서 적합한 process가 선택되어 실행되는 순간은 scheduler에서이지만, 실제로 dequeue 및 enqueue되어야하는 타이밍은 yield될 때이기 때문에, 이를 위해 구현된 함수입니다. L0, L1 기준으로는 queue의 0번째 process를 리턴하고, L2에서는 우선은 priority가 높은 것, 그리고 그중에서는 arrivedTime이 짧은 것을 선택하여 리턴합니다.

```
for (int iterPrior = 0; iterPrior < MAXPRIORITY && dequeueIndex == -1; iterPrior++)
{
    minArrivedTime = qtable.mlfQueue[BOTTOM].procsQueue[0]->arrivedTime;
    for (int i = 0; i < qtable.mlfQueue[BOTTOM].rear; i++)
    {
        // 1. priority가 높은 것 우선
        // 2. FCFS: arrivedTime이 짧은 것 우선
        if (qtable.mlfQueue[BOTTOM].procsQueue[i]->priority == iterPrior && qtable.mlfQueue[BOTTOM].procsQueue[i]->arrivedTime < minArrivedTime)
        {
            dequeueIndex = i;
            break;
        };
    }
    if (minArrivedTime == qtable.mlfQueue[BOTTOM].procsQueue[0]->arrivedTime)
        dequeueIndex = 0;
}
```


- 이와 관련하여, dequeue를 할 때도 L0, L1에서는 흔히 생각하듯 queue의 가장 0번째에 위치한 process를 제거하고, L2에서 MLFQfirstProc가 예외처리 되듯 dequeue에서도 MLFQfirstProc과 동일한 순서로 process를 제거합니다. 물론 L2에 enqueue 될때마다 정렬하는 방법을 쓸 수도 있지만, 저는 dequeue 및 firstProc을 확인할 때마다 주의하는 방식으로 이렇게 구현하였습니다. 이는 enqueue할 때마다 정렬하게 되면, 나중에 priorityBoosting을 할 때 Queue에 들어온 순서대로 넣지 못하고, 섞일 수 있기 때문입니다.

MLFQ scheduling 알고리즘 구현

- 추가된 함수

```
// scheduler에서 적합한 프로세스 선택
struct proc* schedulerChooseProcess(int qLevel);

// RUN하기에 valid한 프로세스인지 검사
int isValidProcess(struct proc* p);

// 1tick이 증가하고 timer interrupt가 발생할 때마다
// 실행중인 프로세스의 execution time을 증가시켜줌
void increaseExecTime(struct proc* p);
void priorityBoosting(void);
```

- void scheduler(void)

```
acquire(&ptable.lock);

for (int qLevel = 0; qLevel < MAXQLEVEL; qLevel++) // ㉑
{
    targetProc = 0;

    // check for higher level queue has new arrived RUNNABLE process
    for (int prev = 0; prev <= qLevel; prev++) // ㉒
    {
        targetProc = schedulerChooseProcess(prev);

        if (isValidProcess(targetProc))
        {
            qLevel = prev;
            break;
        }
    }
}

if (!isValidProcess(targetProc) || targetProc->execTime >= TIME_QUANTUM(qLevel)) // ㉓
    continue;

// ㉔
// 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 = targetProc;
switchuvm(targetProc);
targetProc->state = RUNNING;

swtch(&(c->scheduler), targetProc->context);
switchkvm();

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

release(&ptable.lock);
```

㉑ 루프를 통해 qLevel = 0부터 3까지 순회합니다. 만약, 루프를 도는 도중에 interrupt가 걸리고 나중에 다시 sched로부터 context switch를 받아 실행 되면, 이전에 돌던 루프에 이어서 돌게 됩니다. 하지만, 이런 경우 qLevel이 0이 아닌 1이나 2인 것부터 확인을 하게 될 수도 있는데, 그 사이 L0에 새로운 process가 왔을 수도 있습니다(이 경우 L0에 있는 것부터 스케줄링해야함). 따라서, 이러한 문제를 해소하고자, ㉒에서 추가적인 루프를 또 돕니다. 이를 통해 현재 qLevel보다 낮은 queue에 새로운 process가 그새 왔는지 확인합니다.

㉒ 루프 안에서는, schedulerChooseProcess를 통해 적합한 프로세스를 뽑게 됩니다(추후 이 함수에 대해 설명하도록 하겠습니다). 적절한 값을 반환받았다면, ㉔ 이후의 context switch를 진행, 선택된 프로세스를 실행합니다.

- struct proc* schedulerChooseProcess(int qLevel);

해당 함수가 특정 qLevel에서 MLFQ 스케줄링 알고리즘에 기반하여 적합한 Process를 뽑아내는 핵심 함수입니다.

```
struct proc *targetProc = MLFQfirstProc(qLevel);
```

위 코드를 통해 qLevel에서 가장 0번째, 혹은 L2라면 가장 우선적인 priority 및 FCFS에 의거하여 뽑아진 process가 RUN하기에 적합한지 검사합니다.

```
if (targetProc->state != RUNNABLE || targetProc->killed == 1) // ㉑
...
if (targetProc->execTime >= TIME_QUANTUM(qLevel)) // ㉒
```

state가 RUNNABLE한지, kill되지는 않았는지, execution time(실행된 시간)이 time quantum을 넘지는 않는지의 조건을 확인합니다. 만약 적합하지 않은 것이 firstProc으로 뽑혀 나왔다면, dequeue를 통해 제거하거나(㉠), 또한 만약 time quantum을 초과하였다면, 다음 queue로 넘겨주기 혹은 priority 상승을 시켜줍니다(㉡).

적합한 프로세스라면, 루프를 빠져나와서 해당 프로세스를 return해줍니다.

- void yield(void)

```
increaseExecTime(myproc());
...
// 실행되던 프로세스를 다시 RUNNING에서 RUNNABLE로 바꿔줌
myproc()->state = RUNNABLE;

if (myproc()->isLock == UNLOCKED)
{
    MLFQdequeue(myproc()->qLevel);
    MLFQenqueue(myproc(), myproc()->qLevel);
}
sched();
```

디자인 단계에서 언급했듯, timer interrupt에 의해 yield되는 상황에서는 exectime을 1 증가시켜야하고, system call 차원에서 yield될 때는 yield라는 action이 1tick을 소모하는 것이므로 exectime을 1 증가시켜야합니다. 따라서, 어느 쪽이든 exectime을 증가시켜야하고, 그래서 증가시키게 됩니다.

프로세스가 잘 스케줄링되어 실행되다가, 1tick이 지나 timer interrupt가 발생하면 yield를 통해 다음 프로세스에게 양보합니다.

- void priorityBoosting(void)
 - global tick이 100tick이 되면, starvation을 막기 위해 priority boosting이 발생합니다.

```
if (ticks % 100 == 0 && ticks != 0)
{
    ...
    priorityBoosting();
}
```

```
void priorityBoosting(void)
{
    acquire(&ptable.lock);

    // TOP queue에 있는 process의 exectime과 priority 초기화
    int TopRear = qtable.mlfQueue[TOP].rear;
    for (int iter = 0; iter < TopRear; iter++)
    {
        struct proc *p = qtable.mlfQueue[TOP].procsQueue[iter];
        if (p != 0)
        {
            p->execTime = 0;
            p->priority = MAXPRIORITY - 1;
        }
    }

    // MIDDLE, BOTTOM에 있는 프로세스 초기화 후 TOP에 enqueue
    for (int qLevel = MIDDLE; qLevel < MAXQLEVEL; qLevel++)
    {
        struct proc *p = MLFQdequeue(qLevel);

        while (p != 0)
        {
            p->execTime = 0;
            p->priority = MAXPRIORITY - 1;
            MLFQenqueue(p, TOP);
            p = MLFQdequeue(qLevel);
        }
    }

    release(&ptable.lock);

    // priority boosting에 따른 tick 초기화
    acquire(&tickslock);
    ticks = 0;
    release(&tickslock);
}
```

priority boosting이 발생하면, L1, L2에 있던 프로세스는 L0로 승격되고, 모든 프로세스의 priority와 execution time이 초기화된다(각각 3과 0). L0 - L1 - L2 순으로 L0에 다시 배치되므로, 우선 L0에 있던 프로세스를

System calls

각 System call은 커널 코드 내의 필요한 부분에 로직이 구현되어있으며, sysproc.c에 각각의 wrapper function이 정의되어 있습니다. System call number 설정, user.h에서의 선언, usys.S의 등록 등이 완료되어있습니다.

system call23: yield(void)	system call24: getLevel(void)	system call25: setPriority(pid, priority)
----------------------------	-------------------------------	---

```

acquire(&ptable.lock); // DOC: yield1
myproc()->state = RUNNABLE; // 실행되던 프로세스
// You, 2 hours ago • p1: implement mlfq
if (myproc()->isLock == UNLOCKED)
{
    MLFQdequeue(myproc()->qLevel);
    MLFQenqueue(myproc(), myproc()->qLevel);
}
sched();
release(&ptable.lock);

```

`sys_yield` 를 통해 wrapper되어 호출됩니다.
`sys_yield`는 단순히 `yield`를 호출하기만 합니다.

`ptable`의 `lock`을 얻고, 현재 실행되던 프로세스의 상태를 `RUNNABLE`로 바꿔줍니다. 이후 `lock`된 프로세스가 아니라면(`lock`된 프로세스는 `enqueue`, `dequeue`하는 과정이 무의미) `Queue`에서 제거한뒤, 맨 뒤에 다시 넣어주게 됩니다.
(L0, L1에서 1tick씩 Round Robin을 하게됨)

```

int getLevel(void)
{
    return myproc()->qLevel;
}

```

`sys_getLevel` 을 통해 wrapper됩니다.
`sys_getLevel`은 단순히 `getLevel`을 호출하기만 합니다.
현재 `running`중인 프로세스가 속한 `queue level`을 반환합니다.

```

if (priority < 0 || priority > MAXPRIORITY)
    return; // priority can be 0~3 value
acquire(&ptable.lock);

int found = 0;
for (int qLevel = 0; qLevel < MAXQLEVEL && !found; qLevel++)
{
    for (int i = 0; i < qtable.mlfQueue[qLevel].rear; i++)
    {
        if (qtable.mlfQueue[qLevel].procsQueue[i]->pid == pid)
        {
            qtable.mlfQueue[qLevel].procsQueue[i]->priority = priority;
            found = 1;
            break;
        }
    }
}

release(&ptable.lock);

```

`sys_setPriority` 를 통해 wrapper됩니다.
`ptable`에서 일치하는 `pid`를 찾고, 해당 `process`의 `priority`를 `input priority`로 변경해줍니다.

Scheduler Lock & Unlock

구현

- 도입부: 두 함수 `user program`에서 사용 가능한 `system call` 및 `interrupt`이며, 비밀번호를 `input`으로 받습니다.

```

argint(0, &password);
if(password != SLPASSWORD) {
    struct proc* p = myproc();
    cprintf("[scheduler lock] Wrong Password\n");
    cprintf("pid: %d, time quantum: %d, level of queue: %d\n\n", p->pid, p->execTime, p->qLevel);
    kill(p->pid);
    return -1;
};

```

비밀번호가 틀리면 프로세스는 `kill`되게 됩니다. `SLPASSWORD` 는 `param.h`에 저장되어있고, `2020028586`입니다.

- `ltable`

```

struct
{
    struct spinlock lock;
    struct proc *proc;
} ltable;

```

```
initlock(&ltable.lock, "ltable");
```

`main`에서 함수를 통해 호출되는 `spinlock` 초기화

- `schedulerLock`

```

struct proc *curproc = myproc();
...
// 이미 lock된 프로세스가 있으면 그냥 종료한다.
if (ltable.proc != 0)
    return;

MLFQdeleteByPid(pid); // lock된 프로세스는 MLFQ에서 빠져나와서
                      // ltable에 존재하다가 unlock되면 MLFQ로 돌아간다.

acquire(&ptable.lock);
curproc->isLock = LOCKED; // 현재 프로세스를 LOCKED로 바꿔줌
curproc->state = RUNNABLE; // sched 호출을 위해 RUNNABLE로 바꿔줌
release(&ptable.lock);

acquire(&ltable.lock);
ltable.proc = curproc; // ltable의 proc를 설정
release(&ltable.lock);

// lock 성공, initialize global tick to 0
acquire(&tickslock);
ticks = 0;
release(&tickslock);

acquire(&ptable.lock);
sched();
release(&ptable.lock);

```

`ltable`에서는 `lock`된 프로세스와, `spinlock`을 멤버 변수로 가지고 있습니다. 따라서, 별도의 `boolean` 변수 없이 `ltable`의 `proc`이 0인지 아닌지 판단하여 스케줄러에서는 `lock`된 프로세스가 있는건지 아닌지 판단할 수 있습니다.

`curproc` 은 schedulerLock을 호출한 프로세스(myproc)입니다. 잘못된 비밀번호를 입력하면 kill되며, 이미 lock된 process가 있는 경우에는 lock 호출이 무시됩니다.

무사히 lock 로직 내로 들어오게 되면, curproc의 `isLock` 을 `LOCKED` 로 바꿔주고, sched 호출을 위해 상태를 RUNNABLE로 바꿔줍니다. ltable의 proc을 curproc으로 바꿔주어서 lock된 프로세스가 있음을 나타내줌과 동시에 한번에 lock된 프로세스를 접근할 수 있게 해줍니다.

schedulerLock에 성공했으면, global tick을 0으로 설정해줍니다. 이후, 다시 스케줄링을 하기 위해 `sched()` 를 호출합니다.

- schedulerUnlock 및 schedulerLockDone

```
struct proc *lockproc = ltable.proc;

acquire(&ltable.lock);
ltable.proc = 0;
release(&ltable.lock);

...

acquire(&ptable.lock);
lockproc->execTime = 0;
lockproc->priority = MAXPRIORITY - 1;
lockproc->isLock = UNLOCKED;

enum procstate curState = RUNNABLE;
if (lockproc->state != RUNNING || lockproc->state != RUNNABLE)
    curState = lockproc->state;
lockproc->state = curState;

if (!isExit)
    MLFQfrontEnqueue(lockproc, TOP);
release(&ptable.lock);
```

schedulerUnlock 로직에서는, ltable.proc을 0으로 만들어줌으로써 lock된 프로세스가 없다는 것을 나타낼 수 있습니다.

또한, 기존에 lock되어있던 proc은 실행 시간, priority를 각각 0과 3으로 초기화하고, unlock 상태로 만들며, 상태를 RUNNABLE 혹은 RUNNING이 아니었다면(sleeping이나 zombie) 그 상태로 바꿔줍니다.

또한, schedulerLock되어 실행되던 프로세스가 100 tick을 실행하기 전에 unlock 호출없이 exit하거나 sleep 해버리는 경우, 자동으로 unlock이 호출되는데, 이때는 queue에 삽입해줄 필요가 없으므로 `isExit` 플래그를 통해 확인해서 L0의 가장 앞에 enqueue 해줍니다,

- scheduler(void)

```
if (ltable.proc != 0)
{
    if (ltable.proc->state != RUNNABLE)
    {
        schedulerLockDone(0);
        continue;
    }

    // locked process exists
    targetProc = ltable.proc;
    c->proc = targetProc;
    switchvm(targetProc);

    acquire(&ptable.lock);
    targetProc->state = RUNNING;

    swtch(&(c->scheduler), targetProc->context);
    switchkvm();

    // Process is done running for now.
    // It should have changed its p->state before coming back.
    c->proc = 0;
    release(&ptable.lock);
}
else ...
```

ltable.proc이 0이 아니라면, 즉 lock된 프로세스가 있다면, 해당 프로세스를 실행해줍니다.

호출

- System call은 각각 26, 27 넘버입니다.

각 System call은 커널 코드 내의 필요한 부분에 로직이 구현되어있으며, sysproc.c에 각각의 wrapper function이 정의되어 있습니다. System call number 설정, user.h에서의 선언, usys.S의 등록 등이 완료되어있습니다.

- Interrupt 넘버는 각각 129, 130입니다.

```
// scheduler interrupt system call
#define T_SCHEDULER_LOCK    129
#define T_SCHEDULER_UNLOCK 130
```

```
// interrupt를 user가 호출할 수 있도록 권한을 부여해줍니다.
SETGATE(idt[T_SCHEDULER_LOCK], 1, SEG_KCODE << 3, vectors[T_SCHEDULER_LOCK], DPL_USER);
SETGATE(idt[T_SCHEDULER_UNLOCK], 1, SEG_KCODE << 3, vectors[T_SCHEDULER_UNLOCK], DPL_USER);
...
// schedulerLock과 schedulerUnlock을 interrupt를 통해 호출합니다.
case T_SCHEDULER_LOCK:
    if(myproc()->killed)
        break;
    schedulerLock(SLPASSWORD);
    break;
case T_SCHEDULER_UNLOCK:
    if(myproc()->killed)
        break;
    schedulerUnlock(SLPASSWORD);
    break;
...
```

보조적인 기능

- printProcess: process의 정보를 log 형태로 cprintf해줍니다. 커널 코드입니다.
- printProcessInfo: 현재 process의 정보를 log 형태로 콘솔에 출력해줍니다. system call입니다.

Result



컴파일 및 실행 과정과, 해당 명세에서 요구한 부분이 정상적으로 동작하는 실행 결과를 첨부하고, 이에 대한 동작 과정에 대해 설명합니다.

컴파일 및 실행

편의를 위해 bootxv6.sh에 코드를 좀 더 추가하였습니다.

```
#!/bin/bash
make clean
make CPUS=1
make fs.img
qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m 512
```

따라서 `./bootxv6.sh` 만 입력해도 바로 바뀐 코드를 make하여 실행이 됩니다. 다만 명령어를 하나씩 모두 입력하여 실행해도 무방합니다.

또한, test를 위한 user program은 실습 때처럼 Makefile에 추가해줍니다.

코드는 프로젝트 폴더의 master 브랜치에 별다른 폴더 구분 없이 바로 push 되어있습니다.

MLFQ Scheduler

- 단순한 반복문을 도는 하나의 프로그램으로 테스트해보았습니다.

```
int loop = 11000000;
for(int i = 0; i < loop; i++){
    for(int j = 0; j < 1000000; j++){
        for(int k = 0; k < 100000; k++){
            //getpid();
            //getLevel();
        }
    }
    if(i % 1000000 == 0) printProcessInfo();
}
exit();
```

위와 같이 단순히 루프를 많이 도는 간단한 프로그램으로 테스트를 해보겠습니다.

```
$ myapp
[user program log] pid: 3, qLevel: 0, state: 4, arrivedTime: 3, execTime: 0, priority: 3, isLock: 0
[user program log] pid: 3, qLevel: 0, state: 4, arrivedTime: 3, execTime: 1, priority: 3, isLock: 0
[user program log] pid: 3, qLevel: 0, state: 4, arrivedTime: 3, execTime: 1, priority: 3, isLock: 0
[user program log] pid: 3, qLevel: 0, state: 4, arrivedTime: 3, execTime: 1, priority: 3, isLock: 0
[user program log] pid: 3, qLevel: 0, state: 4, arrivedTime: 3, execTime: 2, priority: 3, isLock: 0
[user program log] pid: 3, qLevel: 0, state: 4, arrivedTime: 3, execTime: 2, priority: 3, isLock: 0
[user program log] pid: 3, qLevel: 0, state: 4, arrivedTime: 3, execTime: 2, priority: 3, isLock: 0
[user program log] pid: 3, qLevel: 0, state: 4, arrivedTime: 3, execTime: 3, priority: 3, isLock: 0
[user program log] pid: 3, qLevel: 0, state: 4, arrivedTime: 3, execTime: 3, priority: 3, isLock: 0
[user program log] pid: 3, qLevel: 0, state: 4, arrivedTime: 3, execTime: 3, priority: 3, isLock: 0
[user program log] pid: 3, qLevel: 1, state: 4, arrivedTime: 3, execTime: 0, priority: 3, isLock: 0
```

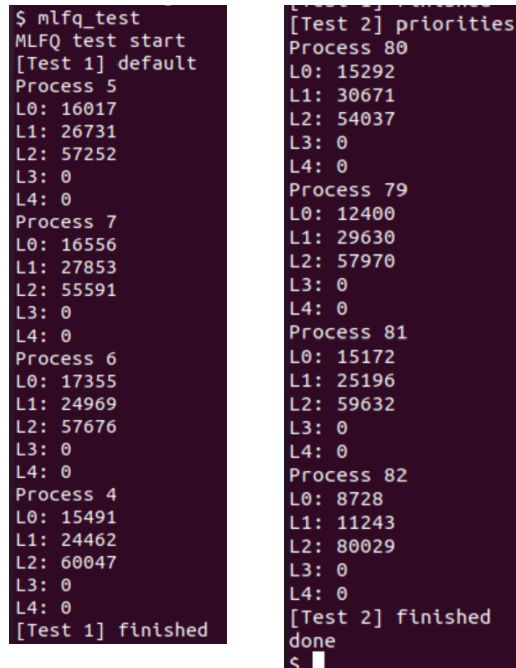
처음에 qLevel이 0인 queue에 enqueue되어 execution time이 time quantum인 $2 * 0 + 4$ 를 넘기 전까지 실행됩니다. 그러다가 time quantum을 넘는 4가 됐을 때, qLevel이 1인 queue에 enqueue되어 실행됩니다.

- mlfq_test.c의 실행 결과

- Test1은 조교님께서 제공해주신 mlfq_test의 코드와 같습니다.

fork_children을 통해 NUM_THREAD만큼 프로세스를 생성합니다. 각 프로세스에서, NUM_LOOP만큼 루프를 돌게 되는데, 이때 이 프로세스가 어느 레벨에서 실행되는지의 비율을 확인해볼 수 있습니다. L0에서의 time quantum이 가장 짧으므로 L0에서 실행되는 시간 비율이 가장 짧고, 그다음이 L1, 그다음이 L2 순으로 실행되는 시간 비율이 짧을 것입니다($L0 < L1 < L2$). 그러나, 100tick마다 priority boosting이 일어나기 때문에, L0에서 프로세스가 실행되는 시간이 극단적으로 짧지는 않으며, 어느정도의 오차 비율 안에서 실행됩니다.

- Test2의 priorities는 Test1과 다른 부분은 다 같고, fork_children2로 실행해본 결과입니다.



```
$ mlfq_test
MLFQ test start
[Test 1] default
Process 5
L0: 16017
L1: 26731
L2: 57252
L3: 0
L4: 0
Process 7
L0: 16556
L1: 27853
L2: 55591
L3: 0
L4: 0
Process 6
L0: 17355
L1: 24969
L2: 57676
L3: 0
L4: 0
Process 4
L0: 15491
L1: 24462
L2: 60047
L3: 0
L4: 0
[Test 1] finished

[Test 2] priorities
Process 80
L0: 15292
L1: 30671
L2: 54037
L3: 0
L4: 0
Process 79
L0: 12400
L1: 29630
L2: 57970
L3: 0
L4: 0
Process 81
L0: 15172
L1: 25196
L2: 59632
L3: 0
L4: 0
Process 82
L0: 8728
L1: 11243
L2: 80029
L3: 0
L4: 0
[Test 2] finished
done
$
```

Required system call & interrupts

system call23: yield

`sys_yield`를 통해 wrapper되어 호출됩니다.

ptable의 lock을 얻고, 현재 실행되던 프로세스의 상태를 RUNNABLE로 바꿔줍니다. 이후 lock된 프로세스가 아니라면(lock된 프로세스는 enqueue, dequeue하는 과정이 무의미) Queue에서 제거한뒤, 맨 뒤에 다시 넣어주게 됩니다. (L0, L1에서 1tick씩 Round Robin을 하게됨)

system call24: getLevel

`sys_getLevel`을 통해 wrapper됩니다.

현재 프로세스가 속한 queue level을 반환합니다.

system call25: setPriority

`sys_setPriority`를 통해 wrapper됩니다. 특정 pid의 priority를 input priority로 바꿔줍니다.

```
pid1 = fork();

if(pid1 == 0){
    // child process
    printf(1, "@@ yield to other process\n");
    yield();

    printf(1, "@@ before changing priority");
    printProcessInfo();

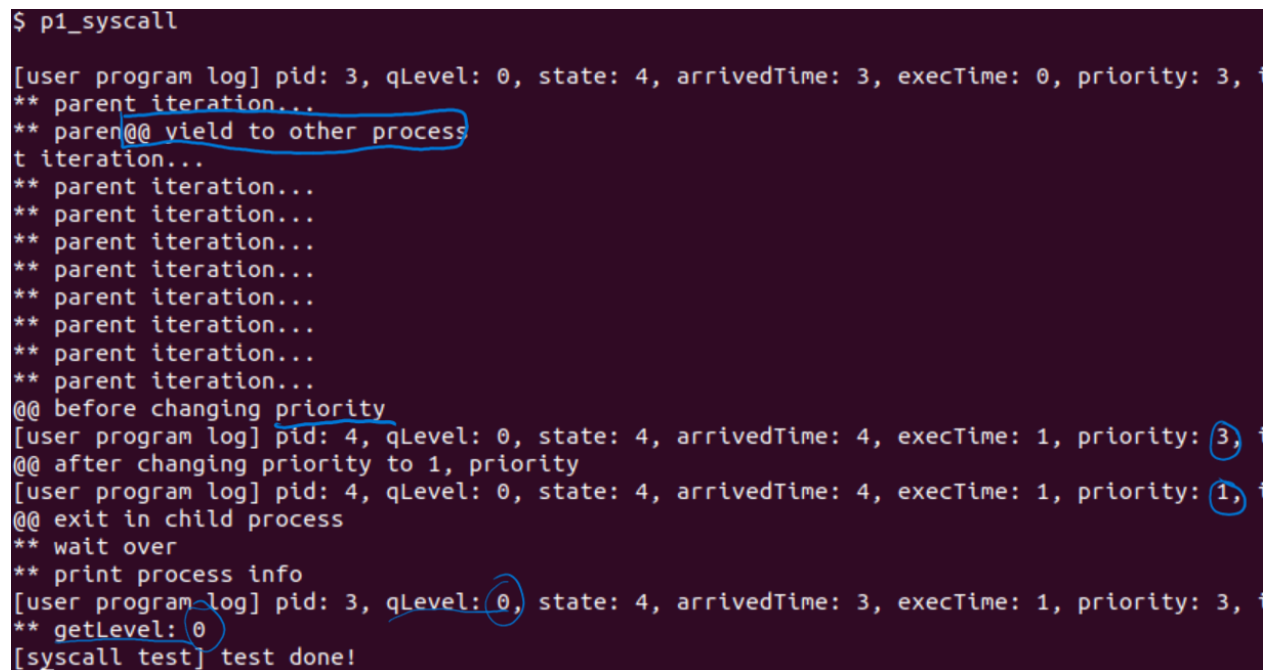
    setPriority(getpid(), 1);

    printf(1, "@@ after changing priority to 1,");
    printProcessInfo();

    printf(1, "@@ exit in child process\n");
    exit();
}
else {
    // parent process
    for(int i = 0; i < 1000; i++){
        if(i % 100 == 0)
            printf(1, "** parent iteration...\n");
    }
    wait();
    printf(1, "** wait over\n");
    printf(1, "** print process info ");
    printProcessInfo();
    printf(1, "** getLevel: %d\n", getLevel());
}

printf(1, "[syscall test] test done!\n");
exit();
```

p1_syscall.c의 일부



```
$ p1_syscall

[user program log] pid: 3, qLevel: 0, state: 4, arrivedTime: 3, execTime: 0, priority: 3, f
** parent iteration...
** parent@@ yield to other process
t iteration...
** parent iteration...
** parent iteration...
** parent iteration...
** parent iteration...
** parent iteration...
** parent iteration...
** parent iteration...
@@ before changing priority
[user program log] pid: 4, qLevel: 0, state: 4, arrivedTime: 4, execTime: 1, priority: 3, f
@@ after changing priority to 1, priority
[user program log] pid: 4, qLevel: 0, state: 4, arrivedTime: 4, execTime: 1, priority: 1, f
@@ exit in child process
** wait over
** print process info
[user program log] pid: 3, qLevel: 0, state: 4, arrivedTime: 3, execTime: 1, priority: 3, f
** getLevel: 0
[syscall test] test done!
```

테스트 결과: 사진에서 @@ 접두어는 child process에서 찍힌 로그, ** 접두어는 parent process에서 찍힌 로그입니다.

1. yield: @@ yield to other process 이후 system call yield 가 호출되면, parent iteration을 호출하던 코드가 이어서 곧장 시작되는 것을 확인할 수 있습니다.
2. getLevel: parent의 ** getLevel: 에서, printProcessInfo 를 통해 확인한 queue Level의 정보와, getLevel 을 통해 반환받은 queue level이 같음을 확인할 수 있습니다.
3. setPriority: child process에서 setPriority 가 호출되기 전 후로 priority가 변한 것을 확인할 수 있습니다.

system call26, 27 & interrupt 129, 130: schedulerLock, schedulerUnlock

schedulerLock과 schedulerUnlock은 system call로도 호출이 가능합니다. sys_ 형태의 wrapper function에서는 정수형 인자를 받아서 password(학번)와 비교해 일치하면 schedulerLock 및 schedulerUnlock 로직을 실행합니다. user.h와 usys.S에 정의되어, user program에서 사용할 수 있습니다.

이 두가지 항목은 각각 interrupt 129(lock), interrupt 130(unlock)으로도 호출할 수 있습니다. 실습 때 인터럽트를 추가한 것처럼, interrupt number를 추가하고, 유저 권한을 부여하고, 해당 트랩 넘버에 대한 handling을 추가하였습니다.

시스템 콜 자체는 잘 호출이 되며, 자세한 schedulerLock의 로직 테스트 결과는 아래에서 설명하겠습니다.

SchedulerLock

저는 앞서 [callout](#)에서 서술한 바와 같이, schedulerLock을 걸면, schedulerUnlock을 호출하거나 global tick이 100이 되기 전까지는 schedulerLock을 호출한 프로세스가 CPU를 독차지하여 MLFQ scheduler에 의한 scheduling을 무시하고 실행되도록 구현하였습니다.

```
if(pid1 == 0){
    schedulerLock(pw); // ㉑
    printf(1, "\n@@ locked process id: %d\n", getpid());

    for(int iter = 0; iter < 103; iter++){
        printf(1, "@@ iteration in child lock... iter: %d\n", iter);
    }

    printf(1, "@@ slept for 50 in child locked process...\n");
    sleep(10); // ㉒
    schedulerUnlock(pw); // ㉓
    printf(1, "@@ scheduler unlocked!\n"); // ㉔
    for(int iter = 0; iter < 100; iter++){
        printf(1, "@@ loop in child, iter: %d\n", iter);
        sleep(1);
    }
    exit();
}

else {
    printf(1, "\n** processs outside of locked process again: %d\n", getpid());
    printf(1, "*** parent is going to run loop for 100 times...\n");
    for(int iter = 0; iter < 100; iter++){
        printf(1, "*** loop in parent, iter: %d\n", iter);
        sleep(1);
    }

    wait();
    printf(1, "*** slept is over in parent, exit in parent process\n");
}
```

위와 같은 코드를 통해 테스트해보았습니다. 사진에서 @@ 접두어는 child process에서 찍힌 로그, ** 접두어는 parent process에서 찍힌 로그입니다. [~~~ log] 접두어는 커널 코드에서 cprintf로 찍힌 로그입니다.

1. lock을 걸었을 때의 결과(위의 코드에서 ㉑, ㉒, ㉓를 활성화)

```
$ p1_locktest
[user program log] pid: 83, qLevel: 0, state: 4, arrivedTime

** processs outside of locked process again: 83
** parent is going to run loop for 100 times...
** loop in parent, iter: 0

@@ locked process id: 84
@@ iteration in child lock... iter: 0
@@ iteration in child lock... iter: 1
@@ iteration in child lock... iter: 2
@@ iteration in child lock... iter: 3
@@ iteration in child lock... iter: 4
@@ iteration in child lock... iter: 5
@@ iteration in child lock... iter: 6
@@ iteration in child lock... iter: 7
@@ iteration in child lock... iter: 8
@@ iteration in child lock... iter: 9
@@ iteration in child lock... iter: 10
@@ iteration in child lock... iter: 11
@@ iteration in child lock... iter: 12
@@ iteration in child lock... iter: 13
@@ iteration in child lock... iter: 14
@@ iteration in child lock... iter: 15
@@ iteration in child lock... iter: 16
@@ iteration in child lock... iter: 17
@@ iteration in child lock... iter: 18
@@ iteration in child lock... iter: 19
@@ iteration in child lock... iter: 20
@@ iteration in child lock... iter: 21
@@ iteration in child lock... iter: 22
```

lock이 걸린 순간부터 child process(schedulerLock을 호출한 프로세스)만 실행된다.

```

@@ iteration in child lock... iter: 97
@@ iteration in child lock... iter: 98
@@ iteration in child lock... iter: 99
@@ iteration in child lock... iter: 100
@@ iteration in child lock... iter: 101
@@ iteration in child lock... iter: 102
@@ slept for 50 in child locked process...
scheduler unlocked!

[unlock - ltable.proc log] process is NULL!

[unlock - lockproc log] pid: 4, qLevel: 0, state: 4, arrivedTime: 9, execTime: 0, priority: 3, isLock: 0
** loop in parent, iter: 1
** loop in parent, iter: 2
** loop in parent, iter: 3
** loop in parent, iter: 4
** loop in parent, iter: 5
** loop in parent, iter: 6
** loop in parent, iter: 7
** loop in parent, iter: 8
** loop in parent, iter: 9
** loop in parent, iter: 10
scheduler unlock called
no locked process now...
@@ scheduler unlocked!
@@ loop in child, iter: 0
** loop in parent, iter: 11
@@ loop in child, iter: 1
** loop in parent, iter: 12
@@ loop in child, iter: 2
** loop in parent, iter: 13
@@ loop in child, iter: 3
** loop in parent, iter: 14
@@ loop in child, iter: 4

```

child process가 sleep하면서 커널 내부적으로 unlock을 호출, 부모와 혼재되어 출력됨 (실제로는 scheduler unlocked! ~ [unlock - lockproc log] line의 출력문은 삭제 되었기 때문에 출력되지 않습니다)

캡처 사진처럼, fork로 생성된 child process에 lock을 걸면(물론 실제로 lock이 걸리고 sched가 호출되기 전까지, printf가 실행됨으로 인해 console output에는 약간의 오차가 있습니다), lock된 프로세스의 루프만 독립적으로 실행되는 것을 확인할 수 있습니다.

그러다가 schedulerLock된 프로세스에서 ㉔, 즉 sleep을 호출하면, 더이상 프로세스가 RUNNABLE이 아니게 되므로 schedulerUnlock이 호출됩니다. lock했던 프로세스와 같은 프로세스가 unlock됨을 확인할 수 있습니다.

그 이후로는 child가 sleep(10)을 하는 동안, parent에서 한번 iter 변수를 출력하고 sleep(1)을 하는 루프가 실행되므로, 10번 실행됩니다. 이후 child가 깨어나서 schedulerUnlock을 명시적으로 호출하는데, 디자인에서 언급된 바와 같이 lock된 프로세스가 없으므로 아무런 일을 하지 않습니다.

2. lock을 걸지 않았을 때(위의 코드에서 ㉔, ㉕, ㉖를 주석 처리)

```

$ p2_locktest
Creating process id: 3

** processs outside of locked process again
@@ locked process id: 4
@@ iteration in child lock... iter: 0
@@ iteration in child lock... iter: 1
@@ iteration in cn: 3
** parent is going to run loop for 100 times)...
** loop in parent, iter: 0
child lock... iter: 2
@@ iteration in child lock... iter: 3
@@ iteration in child lock... iter: 4
** loop in parent, iter: 1
child lock... iter: 4
@@ iteration in child lock... iter: 5
@@ iteration in child lock... iter: 6
** loop in parent, iter: 2
@@ iteration in child lock... iter: 7
@@ iteration in child lock... iter: 8
@@ iteration in child lock... iter: 9
** loop in parent, iter: 3
@@ iteration in child lock... iter: 10
@@ iteration in child lock... iter: 11
@@ iteration in child lock... iter: 12
@@ iteration in child lock... iter: 13
** loop in parent, iter: 4
@@ iteration in child lock... iter: 14
@@ iteration in child lock... iter: 15
** loop in parent, iter: 5
@@ iteration in child lock... iter: 16
@@ iteration in child lock... iter: 17
@@ iteration in child lock... iter: 18
** loop in parent, iter: 6
@@ iteration in child lock... iter: 19
@@ iteration in child lock... iter: 20
@@ iteration in child lock... iter: 21
** loop in parent, iter: 7
@@ iteration in child lock... iter: 22

```

child의 loop와 parent의 loop가 혼재되어 출력되는 것을 볼 수 있습니다.

3. pw는 기본적으로 유저 프로그램 안에서 제 학번으로 설정되어있습니다. 하지만, 코드 상으로 학번을 수정할 수 있으며, command line argument가 기본적으로 설정되어있는 첫번째 인자를 학번으로 받습니다.

```

int pw = 2020028586;

if(argc > 1){
    pw = atoi(argv[1]);
    printf(1, "if statement runs, input pw: %d\n", pw);
}

```



```

$ p2_locktest 1234
Creating process id: 3
if statement runs, input pw: 1234

** processs outs[scheduler lock] Wrong Password
pid: 4, time quantum: 0, level of queue: 0

ide of locked process again: 3
** parent is going to run loop for 100 times)...
** loop in parent, iter: 0
** loop in parent, iter: 1
** loop in parent, iter: 2
** loop in parent, iter: 3
** loop in parent, iter: 4
** loop in parent, iter: 5
** loop in parent, iter: 6
** loop in parent, iter: 7
** loop in parent, iter: 8
** loop in parent, iter: 9
** loop in parent, iter: 10
** loop in parent, iter: 11
** loop in parent, iter: 12
** loop in parent, iter: 13
** loop in parent, iter: 14
** loop in parent, iter: 15
** loop in parent, iter: 16
** loop in parent, iter: 17
** loop in parent, iter: 18
** loop in parent, iter: 19
** loop in parent, iter: 20

```

Password가 틀리면, pid, time quantum, level of queue가 출력되고, schedulerLock을 호출했던 child process는 kill됩니다.

4. interrupt를 통해서도 호출 가능합니다.

```

int main(int argc, char *argv[]){
    __asm__("int $129"); // schedulerLock
    printf(1, "locked in userprogram by using trap");
    __asm__("int $130"); // schedulerUnlock
    sleep(300);
    exit();
}

```

Trouble shooting



과제를 수행하면서 마주하였던 문제와 이에 대한 해결 과정을 서술합니다. 혹여 문제를 해결하지 못하였다면 어떤 문제였고 어떻게 해결하려 하였는지에 대해서 서술합니다.

enqueue, dequeue 등의 method를 완성할 때마다 따로 작은 프로그램을 만들어서 의도한 대로 잘 동작하는지 확인해보았었습니다. 이를 위해 올바른 시나리오와 그에 따른 테스트 케이스를 직접 생각해내야 했었습니다.

하지만, 실제로 작성한 코드가 xv6 전체에서 잘 동작하는 것과 작은 단위의 테스트는 역시 조금 거리가 있었습니다.

xv6 구조 파악

scheduler를 짜기에는 아직 xv6의 전체적인 구조에 대한 이해가 부족했었습니다. 이를테면, scheduler는 계속해서 루프를 이어서 돌고, sched를 통해 context switch되어 실행되는 등의 부분에 대한 이해가 부족해서, 왜 이어서 실행되는지 등과 관련된 간단한 문제조차 쉽사리 고치지 못했었습니다.

그래서 MIT의 xv6 book을 찾아서 공부해보았었고, 이론 수업 및 조교님들께서 해주셨던 실습 수업을 다시 복습하면서 xv6가 어떻게 실행되는 것인지, 스케줄러가 어떤 원리로 작동하는 것인지(timer interrupt가 걸릴 때마다 yield 함으로써 Round-Robin을 사용하는 것처럼 되는 등) 공부하였습니다.

6.1810 / Fall 2022

 <https://pdos.csail.mit.edu/6.828/2022/>

디자인 관련

Queue 구현 아이디어 후보

ptable을 그대로 쓰는 방법을 생각했었습니다. Queue를 실제로 구현하지는 않지만, MLFQ 알고리즘에 기반하여 프로세스를 선택할 수 있도록 보조적인 함수를 만들고자 하였었습니다. 즉, 실제로 Queue의 자료 구조가 존재하지 않지만, ‘*마치 queue 자료 구조를 사용하는 것처럼*’ 동작하게 만들려고 하였습니다. 기존의 xv6의 scheduling을 비롯한 process와 관련된 동작들이 ptable기반으로 모두 짜여져 있기 때문에 안정성이 높을 것이라는 추측에서 이루어진 일이었습니다. 하지만, ptable의 중간까지 실행하다가 앞의 프로세스가 무작위로 kill되고, 재할당되는 경우 등의 여러 edge 케이스를 생각할 수 있었고, 이에 대비하기가 힘들어질 것이며, Queue를 직접 구현하여 사용하는 것이 오히려 MLFQ 알고리즘을 적용하기에는 더 적합할 것이라는 생각이 들어 해당 후보는 제외되었습니다.

MLFQ scheduling 구현 아이디어 후보

1. 초기에는 queue 자료 구조를 만들지 않고, ptable을 그대로 쓰되 ‘*마치 queue처럼 동작하도록 하는*’ 방법을 이용하려고 했었습니다. 당시 생각했던 방법은 다음과 같습니다.

- `lastProc`이라는 `struct proc*` 변수를 두고, 해당 변수에 마지막으로(latest) 실행되었던 process를 저장합니다.

- b. `scheduler` 함수가 권한을 받아서 이어서 실행되면, queue level을 0부터 2까지 증가시키는 for문 안에서, 가장 마지막에 실행되고 있었던 `lastProc` 을 확인합니다.
 - time quantum을 초과하지 않았다면, 그대로 이어서 실행합니다.
 - 만약 time quantum을 초과했다면, level이 높은 queue로 이동시키거나, priority를 감소시킵니다. 이후 현재 순회하고 있는 queue level과 맞는 ptable의 process를 찾아서 해당 프로세스를 실행합니다.

2. 하지만, 이 경우 `trap 14` 에러가 나면서 아예 shell조차 구동되지 않았었습니다. 디버깅을 해보다가, 여러 edge case가 많다는 것을 깨닫고, queue를 직접 구현하는 것이 나올 것 같다는 판단을 내렸습니다.

테스트 케이스 생성

mlfq scheduler나 schedulerLock을 테스트하는 케이스가 유효한지 논리적으로 검증하는 과정에서 애로 사항을 겪었습니다.

에러

debugging

- using `cprintf`
평소에 ubuntu, vim, gdb를 많이 사용해보지 않아서, 처음에는 `cprintf` 로 디버깅을 시도했었습니다. 하지만, 매번 `cprintf`를 코드 사이사이에 추가하고, 다시 빌드하고, 또한 중간중간 새롭게 보고 싶은 값이 있을 때 혹은 플래그를 변경해서 보고 싶을 때 멈췄다가 다시 해야되는 번거로움이 있었습니다.
- using gdb
따라서 xv6에서 gdb를 사용하는 방법을 찾아보았습니다. `make qemu-nox-gdb CPUS=1` 이라는 옵션을 통해 xv6를 빌드하고, 터미널 창을 하나 더 열어서 `gdb kernel` 을 통해 원하는 지점에 브레이크 포인트를 걸거나, 행 단위로 실행하고, 변수 값을 조회하거나 설정하는 등의 디버깅을 할 수 있었습니다.

unexpected trap 14

처음 마주했던 문제는 다음과 같습니다.

```
unexpected trap 14 from cpu 0 eip 80105de6 (cr2=0x10)
lapicid 0: panic: trap
80105f52 80105b6c 80105b6c 8010301f 8010316c 0 0 0 0
```

proc.c의 scheduler가 문제인건지, 아니면 trap.c 혹은 기타 system call을 추가하는 과정에서 문제가 생긴건지 구별하기 위해서, 각각의 부분에 대해 기존 코드로 교체하고 어디가 문제인지 찾아보고자 했었습니다. 원인은 proc.c의 기존 scheduler인 Round Robin에서 제가 구현한 MLFQ로 교체한 것이었습니다.

하지만 로직적으로 왜 이 문제가 발생하는지 디버깅을 해보고자 `cprintf` 를 scheduler에서 무한 루프를 도는 for(;;) 전후로 출력해보았었습니다. 하지만, `sti()` 자체에 접근하지 못하고 trap이 발생하였었습니다.

trap 14는 xv6에서 page fault를 나타낸다고 하여, gdb를 통해 디버깅해보고자 하였었습니다. 하지만, gdb를 통해 브레이크 포인트를 걸어가며 천천히 실행하자 trap 14에 걸리지 않고 잘 실행됐었습니다. 그래서 디버깅하기가 더 까다롭기도 했고, 왜 gdb로는 trap이 재현되지 않는건지 궁금했었습니다. 구글링을 통해 다음의 답변을 얻을 수 있었습니다.

This could be happening because gdb debugger is slowing down the execution of your code which gives your system more time to load pages into memory before they are needed. When you run your code without gdb debugger, it runs faster and may cause page faults because pages are not loaded into memory before they are needed.

You can try running your code with gdb debugger and using the command ``set pagination off`` to turn off pagination which may help you see more information about what is happening when a page fault occurs.

다만 trap 14 자체는 다양한 원인에 의해 발생하는 것이기 때문에 정확한 원인을 찾았다기 보다는 어느 line에서 trap 14가 발생한건지 찾을 수 있었습니다.

하나의 프로세스는 1tick만 실행되던 문제

프로세스를 MLFQ에서 dequeue하여 RUNNABLE하면 CPU를 할당해주고 해당 프로세스를 실행한 뒤, 다시 queue에 enqueue 해줘야합니다.

그러나 enqueue를 해주지 않았기 때문에 오직 1tick만 실행된 채로 queue를 빠져나간 뒤 돌아오지 않았고, 실행할 프로세스가 없어서 xv6를 빌드하면 프로그램이 멈추는 문제가 발생했었습니다.

queue 내부에서 프로세스가 1tick마다 Round-Robin되지 않음

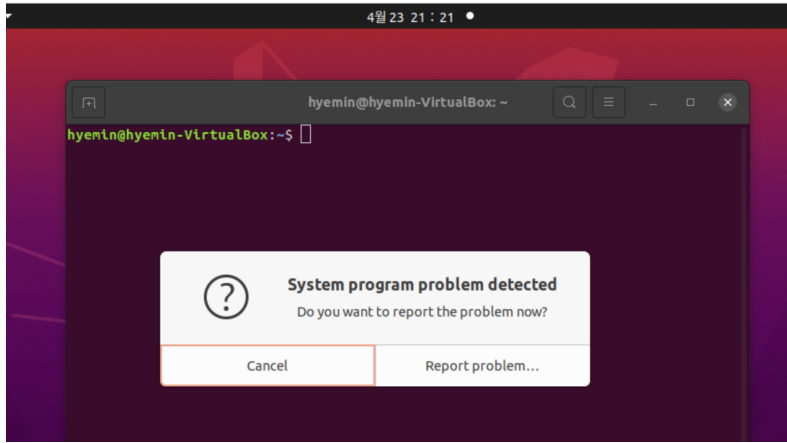
실행하면 높은 확률로 xv6가 버벅이고 터미널이 얼어버리는 현상이 발생했었습니다. 원래는 scheduler가 sched를 통해 실행되면, scheduler 안에서 dequeue를 하고, scheduler가 끝날때쯤 (즉, $c \rightarrow \text{proc} = 0$ 을 실행한뒤) enqueue를 했었습니다. 그런데, scheduler가 끝날 때 enqueue하는게 아니라 타이머 인터럽트가 발생하고 yield를 할 때 queue 내부의 process를 dequeue하거나 enqueue하는 동작을 해야 정상적으로 동작하였었습니다. scheduler 내부에서 실행하다가 sleep 되거나 exit 되면 어차피 schedulerChooseProcess의 동작에 의해 새로운 적합한 프로세스를 찾을 수 있으므로, tick에 따라 yield가 될 때 실질적인 enqueue 및 dequeue를 해줘야 tick에 맞게 동작할 수 있었습니다. 즉, Running중이던 프로세스가 RUNNABLE로 전환될 때 dequeue 및 enqueue를 해줘야했었습니다.

mycpu called with interrupts enabled

proc.c의 schedulerLock에서 cpuid를 호출해서 생긴 문제였습니다.

tick = 0로 해주는 코드를 trap.c에서 그대로 가져와서 사용했었기 때문인데, 어차피 cpu는 하나이기 때문에 mycpu의 확인 없이 tick을 초기화해주어도 괜찮았습니다.

ubuntu system error



ubuntu를 정상적으로 종료 및 실행해도, 계속해서 `system program problem is detected`라는 경고 팝업이 뜨면서 ubuntu가 얼어버리는 현상이 발생했었습니다. 다양한 원인이 있기에, 구글링을 해보았으나, 아직 적절한 해결 방법을 찾지 못하였습니다.

IDE 사용과 관련하여

virtual box 자체에 vscode를 설치해서 쓰려고 했으나 느리거나 해상도 문제가 있었고, CLI만으로 작업하는 터미널 방식이나 vim이 익숙하지 않았었습니다.

그래서 project01을 위한 브랜치를 따로 만들어서, 코드 작성시에는 windows 환경의 vscode에서 작업을 하고, 브랜치에 push한 뒤, 작은 테스트만 virtual box의 ubuntu 환경에서 진행하였습니다.

최종적으로 완성한 과제는 master branch에 한개의 커밋으로 squash merge 하였으나, `project01` 및 `p1` 접두어가 붙은 브랜치에서 커밋 내역을 보실 수 있습니다. 다만 master가 아닌 브랜치에서는 windows에서의 커밋으로 인하여 제 이름으로 커밋된 내역이 있습니다.