Project2_WIKI

▼ xv6 상세분석

Locks and conditional variables

- xv6에서의 Locks(i.e, spinlocks)는 xchg atomic instruction에 의하여 수행이된다. acquire(struct spinlock *lk) 함수는 모든 interrupts를 disable상태로 만들며 release(struct spinlock *lk)함수를 사용하여 enable로 바꾸어 준다.
- xv6는 sleep이라는 함수와 wakeup이라는 함수를 제공하는데 이들은 conditional variable을 사용하는 wait , signal 함수와 동일한 역할을 수행한다. sleep, wakeup 함수는 atomically하게 작동하는 것을 보장해주는 lock을 반드시 획득한 상태로 작동시켜야한다. sleep(chan, lock)에서 **chan**이라는 것은 Opaque Type 이며 lock은 sleep, wakeup을 호출하는 모든 코드에서 사용되는 어떤 lock이던 상관없다. Sleep function은 process로 부터 받은 lock을 release시키고 process의 state를 변경시키고 scheduler를 invoke하기 위하여 ptable.lock을 획득한다. ptable.lock을 두 번 acquire할 수 있기 때문에 lock의 종류가 ptable.lock인지 아닌지 확인하는 과정을 거친다.
- 일단 한번 sleep function이 scheduler를 호출하게 되면 context switch가 발생한다. Process가 깨워지고 Schedueler에 의해 다시 호출되면 이 sleep fucntion이 scheduler를 호출한 다음 line부터 시작한다. sleep 함수는 caller가 제공한 lock이나 ptable.lock중 하나를 보유하고 있기 때문에 다른 프로세스가 sleep()을 수행하는 동안 wakeup 시킬수 없다.
- lock을 가지고 있는 동안 조건을 만족시키는 어떤 프로세스가 wakeup(chan)을 호출하게 될 것이다. wakeup call은 모든 waiting process들을 RUNNABLE상태로 만들며 이것은 pthreads API의 broadcast function과 동일한 역할을 수행한다. 주목해야할 점은 wakeup call이 실제로 프로세스를 깨우는 context switch가 아니라 RUNNABLE상태로 state를 바꿔주는 과정이며, 이렇게 RUNNABLE로 바뀐 프로세스들은 나중에 scheduler에 의하여 CPU를 차지한 후에 코드를 계속해서 수행하게 된다.
- xv6는 sleep과 wakeup 함수를 userspace process들에 export하지 않는다. 하지만 코드내의 일부분에서 내부적으로 사용할 수 있게 만들어 준다. 예를 들면 pipe의 구현에 있어서 Binding buffer를 사용할 때 생산자-소비자 문제에 명확하게 매핑되고 이 때 lock과 conditional variable을 통하여 동기화를 진행한다.

Scheduler and context switching

- xv6는 다음과 같은 process state를 사용한다 : UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE. 또한 proc 구조체들은 모두 linked list에 저장되어 있다.
- 모든 CPU는 scheduler thread를 가지고 있으며 이는 scheduler function을 수행하고 이 function은 loops를 영원히 수행한다. 이 scheduler function의 작업은 process list를 탐색하여 RUNNABLE 상태의 프로세스를 찾고 RUNNING상태로 바꾼후 process를 swich해주는 작업이다.
- Process list에서 일어나는 모든 작업은 ptable.lock에 의해 안전성이 보장되며 따라서 이 자료구조에 존재하는 값을 변경하고 싶은 모든 프로세스들은 lock을 획득한 상태로 작업을 수행해야 한다. 만약 lock을 획득하지 않은 상태로 작업을 수행한다면 두개의 CPU가 하나의 process를 RUNNABLE상태로 변환할 수 있으며 이는 시스템 오류를 유발한다.
- 모든 프로세스들은 Context라는 structure를 자신의 stack에 가지고 있으며 이 구조체에는 **EDI**(Extended Destination Index), **ESI**(Extended Source Index), **EBX**(Extended Base Register), **EBP**(Extended Base Pointer-스택 프레임 시작주소), **EIP**(PC라고도 불리우는 Instruction pointer- 다음에 실행이 어디서 재개될지 알 수 있다) 이렇게 5개의 Context switch에 필요

한 CPU register값을 저장한다. 현재 RUNNING상태인 P1 프로세스에서 P2라는 프로세스로 Switch하기 위해서 P1의 레지스터 값들을 스택에 저장하고 P2가 한 번 수행됐었던 프로세스라면 자신의 context를 reload하여 사용한다.

- swtch fucntion은 두 개의 context를 switching해주는 역할을 수행한다. register 값들을 old stack에 push하는 과정은 새로운 stack에서 register값들을 복원하는 과정과 유사하다. Context structuer가 'swtch'에 의해 Push 되지 않는 유일한 순간은 프로세스가 처음으로 작성될 때이다. 모든 새로운 프로세스들에 대하여 'allocproc'은 new kernel stack에 새로운 context를 작성한다. 이것은 나중에 proecess가 처음 swtch에 의해서 수행될 때 CPU register에 load된다.
  모든 새로운 프로세스들은 forkret에서 처음 수행되며 코드는 다음과 같다.

```
2850 // A fork child’s very first scheduling by scheduler()
2851 // will swtch here. "Return" to user space.
2852 void
2853 forkret(void)
2854 {
2855 static int first = 1;
2856 // Still holding ptable.lock from scheduler.
2857 release(&ptable.lock);
2858
2859 if (first) {
2860 // Some initialization functions must be run in the context
2861 // of a regular process (e.g., they call sleep), and thus cannot
2862 // be run from main().
2863 first = 0;
2864 iinit(ROOTDEV);
2865 initlog(ROOTDEV);
2866 }
2867
2868 // Return to "caller", actually trapret (see allocproc).
2869 }
```

그 결과 allocproc은 이 주소를 EIP(PC)에 저장한다. 이렇게 처음 생성된 프로세스의 경우를 제외하고 EIP에는 항상 'swtch'가 발생된 시점의 주소가 담겨있으며 다시 스케줄링을 할당받을 때 그 주소로 부터 시작된다.

- xv6에서 scheduler는 고유의 stack에서 분리된 스레드로 작동한다. 그러므로 context switch는 running process의 kernel mode로 부터 scheduler thread로 발생한다. 그리고 scheduler thread에서 다시 새로운 프로세스로 context switching이 일어나는 것이다. 그 결과, 'swtch' function은 두 번 호출된다. 기존 프로세스에서 한번, 스케줄러에서 한번. 오직 예외는 처음 실행되는 프로세스 이다.

- cpu를 포기하려는 프로세스에서 'sched'를 호출한다. 이 함수는 context switch를 유발하며 나중에 프로세스를 다시 전환하여 돌아오면 실행을 'sched'에서 재개한다. 그러므로 sched를 호출하는 것은 프로세스의 실행을 일시적으로 중단시키게 된다. 그렇다면 프로세스가 CPU를 포기하는 순간은 언제일까?
  1) timer interrupt가 발생할 때 프로세스가 오래 실행된 것으로 간주되며 Trap함수는 yield를 호출하고 이는 다시 sched를 호출한다.
  2) 프로세스가 exit를 사용하여 종료될 때 프로세스는 sched를 호출하여 CPU를 포기해야 한다.
  3) 프로세스가 event 발생을 막고 sleep해야 할때 sched를 호출하여 CPU를 포기한다. sched는 단순히 조건을 체크하고, swtch를 호출하여 스케줄러 스레드로 전환한다.

- sched를 호출한 function들은 항상 ptable.lock을 확보한 상태여야 하며 이 lock은 context switch동안 계속 잡고있어야 한다. P1에서 P2로 변경된다고 하면 P1에서 scheduler로 넘어갈 때 lock을 hold하고 scheduler에서 P2로 넘어간 후에 sched에서 빠져나오면 lock을 release한다.

- 모든 interrupt들은 lock이 hold되어있는 동안 disabled된다. 만약 스케줄러가 아무런 프로세스가 작동하고 있지 않음을 확인한다면 스케줄러는 주기적으로 ptable.lock을 release해준 후 다시 RUNNABLE한 프로세스가 있는지 없는 지 확인한다.

1. 분석 및 구현

1-1. RR-Scheduling : XV6 Default

Process Table 자료구조가 존재하며 이것은 lock이라는 변수와 proc이라는 Array를 가지고 있다. lock은 멀티 프로세서를 사용하는 환경에서 Race condition을 막기 위한 변수이고 ptable은 프로세스들은 main에서 pinit() 이라는 함수에 의해 초기화 된다.

timer interrupt가 발생하면 아래 코드가 실행된다.

```
// trap.c의 trap()일부
if(myproc() && myproc()->state==RUNNING &&
  tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();

// proc.c의 일부
struct proc * myproc(void){
  struct cpu *c;
  struct proc *p;
  pushcli();
  c=mycpu();
  p= c-> proc;
  popcli();
  return p;
}

void
yield(void){
  acquire(&ptable.lock); /// DOC : yield lock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

interrupt가 발생했을 때 현재 CPU의 프로세스를 myproc()을 통하여 얻어내고 1.프로세스가 존재하며, 2. 프로세스의 상태가 RUNNING이고, 3.Trapframe의 trapno, 즉 interrupt의 종류가 Time interrupt일 때 yield()를 수행한다. yield()는 현재 수행중인 프로세스의 상태를 Runnable로 변경해주고 sched()를 호출한다.
++pushcli와 popcli는 cli, sti와 비슷한 역할을 수행하며 cli는 CLear Interrupt의 약자이고 sti는 SeT Interrupt의 역할이며 각각 인터럽트를 disable, enable하는 역할을 수행한다.

```
//proc.c
void
sched(void){
  int intena;
  struct proc *p = myproc();
  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state==RUNNING)
    panic("sched running");
  if(readeflags() & FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

sched는 기본적으로 현재 프로세스의 context와 새로운 프로세스의 context를 switch해주는 것이며 위에서는 현재 프로세스(old)와 CPU scheduler의 context를 switch해주고 있다.
++ ncli는 pushcli의 nesting된 Depth를 의미하고 cpu의 intena는 pushcli전에 interrupt가 enabled인 상태인지 확인하는 것이다.

```
//proc.c
void
scheduler(void){
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc =0; //scheduler

  for(;;){
    //Enable interrupts on this processor.
    sti();
    //Loop over process table looking for process tor run
    acquire(&ptable.lock);
    for(p=ptable.proc; p< &ptable.proc[NPROC];p++){
      if(p->state != RUNNABLE)
        continue;
      c->proc = p;
      switchuvm(p);
      p->state =RUNNING;
      swtch(&(c->scheduler), p->context);
      switchkvm();
      c->proc = 0;
    }
    release(&ptable.lock)
  }
}
```

scheduler()는 실제 프로세스를 스케줄링하는 function이며 process의 state가 RUNNABLE인 프로세스를 골라 cpu→scheduler(old) 와 process의 context를 switch한다. 이러면 해당 프로세스가 멈췄던 부분부터 다시 시작할 수 있게 되는 것이다.

즉 요약하면, Timer interrupt가 발생할 때 마다 현재 프로세스와 CPU scheduler의 context가 switch되어 스케줄링이 시작이되고 다음 프로세스를 찾아 다시 CPU scheduler와 context switch가 일ㄹ어나며 스케줄링이 진행되는 것이다. xv6의 timer interrupt의 경우 1tick(10ms)마다 발생한다.

1-2. FCFS

요구사항:
1. 먼저 fork()된 프로세스가 먼저 스케줄링이 되어야 한다.
2. 기본적으로 스케줄링 된 프로세스는 종료되기 전까지는 switch-out되지 않는다.
3. 그러나 만일 프로세스가 스케줄링 된 이후 200ticks가 지날 때까지 종료되거나 SLEEPING상태로 전환되지 않으면 강제 종료시켜야 한다.
4. 만약 실행 중인 프로세스가 SLEEPING 상태로 전환되면 (IO, Sleep 등) 다음으로 생성된 프로세스가 스케줄링되며, 만약 sleep상태에 있던 프로세스가 깨어날 경우 해당 프로세스가 다시 스케줄링 된다.

구현사항:
0. SCHED_POLICY = FCFS_SCHED로 MAKE하였을 때 작동할 수 있는 처리를 한 후 void sched_FCFS(void) function을 만든다.

```
void
scheduler(void)
{
#ifdef RR_SCHED
```

```
  sched_RR();

#elif FCFS_SCHED
  sched_FCFS();

#elif MULTILEVEL_SCHED
  sched_MLQ();
#elif MLFQ_SCHED
  sched_MLFQ();
#endif
  while(1);
}
```

1. proc.h의 struct proc에 uint allocated_time을 만들어 CPU를 얼마나 차지하고 있었는지를 계산한다.

```
//proc.h 의 struct proc의 일부
char name[16];               // Process name (debugging)
uint allocated_tme;           // the time for cpu running
};
```

3. **void sched_FCFS(void)**를 짠다. 기본적인 아이디어는 ptable을 순차적으로 탐색하여 가장 작은 pid를 가진 프로세스를 찾는다면 allocated_time을 설정해주고 해당 프로세스가 swtch를 이용하여 CPU를 차지할 수 있도록 만들면서 FCFS를 수행할 수 있게 만든다.

```
void
sched_FCFS(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  struct proc *fcfs_proc;
  c->proc =0;
  for(;;){
    sti();

    fcfs_proc=0;
    acquire(&ptable.lock);
    for(p=ptable.proc; p< &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      if(fcfs_proc==0)
        fcfs_proc=p;
      else{
        if(p->pid<fcfs_proc->pid)
          fcfs_proc=p;
      }
    }
    if(fcfs_proc!=0){
        fcfs_proc->allocated_time=ticks;
        c->proc=fcfs_proc;
        switchuvm(fcfs_proc);
        fcfs_proc->state=RUNNING;
        swtch(&(c->scheduler),fcfs_proc->context);
        switchkvm();
        c->proc=0;
      }

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

4. 두번째 요구사항인 스케줄링 된 프로세스가 종료되기 전까지 switch되지 않고 200ticks가 넘을 경우 프로세스를 강제 종료시키기 위하여 다음과 같이 코드를 구성한다. tick에서 현재 프로세스의 allocated_time을 빼 현재 프로세스가 얼마나 CPU를 차지하고 있었는지 확인할 수 있다.

```
//trap.c의 trap의 일부
#elif FCFS_SCHED
  // if the SCHED_POLICY is FCFS_SCHED,
//current process doesn't yield CPU to some other process
  // but if process run more than 200ticks than kill the process
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno==T_IRQ0+IRQ_TIMER && (ticks - myproc()->allocated_time) >= 200){
    cprintf("200ticks over... kill %d process\n",myproc()->pid);
    kill(myproc()->pid);
  }
```

-실행결과-

p2_fcfs_test.c를 수행한 결과 yield()시스템콜을 정의하지 않아 make를 할 수 없어 yield() system call을 만들어 주었다. 다음은 실행 결과이다.

p2_fcfs_test.c에서의 테스트는 다음과 같이 진행된다.

1.sleep과 yield없이 진행

2.yield()를 호출하며 진행

3.sleep()하며 진행

4.Infinite loop테스트

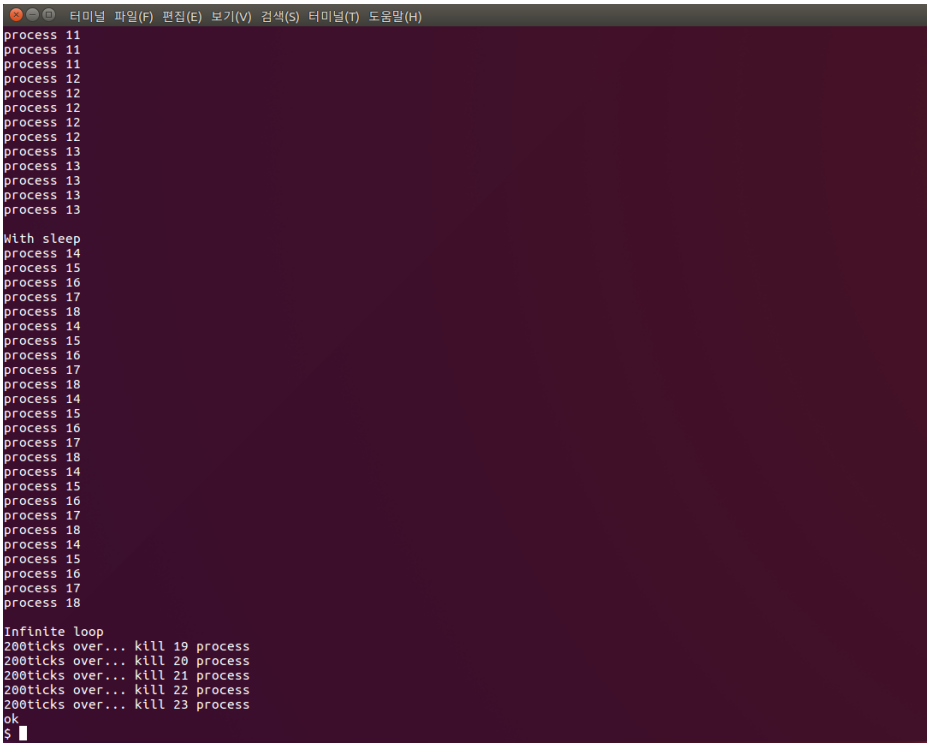
1번의 경우 프로세스 순서대로 진행되어 pid순서대로 종료되고

2번의 경우 yield()를 호출하여도 계속 pid가 낮은 프로세스가 CPU를 차지하여 먼저 종료됨을 알 수 있고

3번의 경우 sleep으로 인해 순차적으로 진행되는 모습이며

4번의 경우 200ticks가 넘어가면 kill되는 모습니다.

모두 정상적으로 FCFS로 작동함을 알 수 있다.



1-3. Multilevel Queue

요구사항:

- 스케줄러는 2개의 큐로 이루어 진다. pid가 짝수인 프로세스들은 RR방식으로 스케줄링 되며 pid가 홀수인 프로세스들은 FCFS방식으로 스케줄링 된다.

- RR큐가 FCFS큐보다 우선순위가 높으며, 이는 항상 pid가 짝수인 프로세스가 먼저 실행됨을 의미한다.
- FCFS 큐는 먼저 생성된 프로세스를 우선적으로 처리한다.
- SLEEPING상태에 있는 프로세스는 무시한다. 즉 RR큐에 프로세스가 존재해도 모두 RUNNABLE상태가 아니라면 FCFS큐를 확인해야 한다.

구현:
1. 스케줄러는 2개의 큐로 이루어진다 하였으나.. 코드에서 홀수 짝수로 나누어 바로 스케줄링을 진행하는 것이 proc구조체에 새로운 queue를 추가하여 탐색을 돌리는 것보다 빠르다고 생각해 다음과 같은 void sched_MLQ(void)함수를 proc.c 내에 작성하였다. 상세한 내용은 주석을 참고하길 바란다.

```
void
sched_MLQ(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc =0;
  int check_even_proc;
  struct proc * odd_proc;

  for(;;){
    sti();

    acquire(&ptable.lock);
    // set odd_proc and check_even_proc 0
    odd_proc =0;
    check_even_proc =0;
    for(p=ptable.proc; p< &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE){
        continue;
      }
      // 이 파트는 홀수 pid를 가진 프로세스를 FCFS수행을 하기 위해 저장해두는 과정이다.
      // 저장만 해두는 이유는 홀수 pid가 저장된 큐의 우선순위가 짝수보다 낮기때문이다.
      if(p->pid%2==1){
        if(!odd_proc || (p->pid < odd_proc->pid)){
          odd_proc = p;
        }
        continue;
      }
      // 이 부분은 짝수 pid를 가진 프로세스를 처리하기 위한 부분이다.
      // write this part as same way of shced_RR (xv6 default)
      // 다른건 다 같지만 마지막에 check_even_proc을 1로 하여 홀수 pid를 실행하지 않게 만든다.
      c->proc = p;
      switchuvm(p);
      p->state=RUNNING;
      swtch(&(c->scheduler),p->context);
      switchkvm();
      c->proc =0;
      check_even_proc=1;
    }
    //when we got out of the loop, we check check_even_proc is 0 or 1
    //실행할 짝수 프로세스가 없음을 확인하고 실행할 홀수 프로세스가 있다면 스케줄링 해준다.
    if(check_even_proc==0 && odd_proc !=0){
      odd_proc->allocated_time=ticks;
      c->proc = odd_proc;
      switchuvm(odd_proc);
      odd_proc->state=RUNNING;
      swtch(&(c->scheduler), odd_proc->context);
      switchkvm();
      c->proc=0;
    }
    release(&ptable.lock);
  }
}
```

2. odd pid를 가진 프로세스는 FCFS로 진행되기 때문에 trap.c에서 동일하게 pid가 홀수일 경우 200tick이 넘어갈 경우 프로세스가 kill되도록 만들었다.

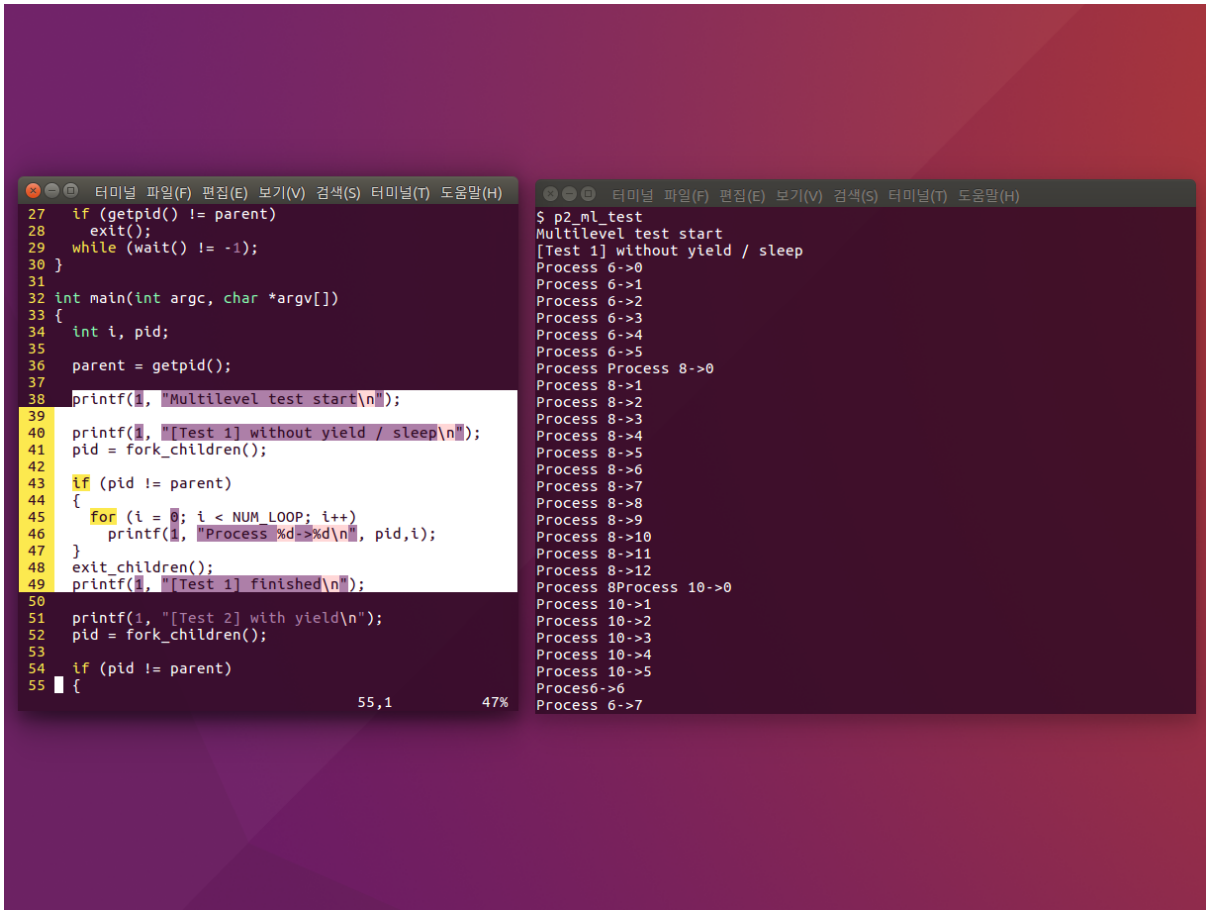
```
//trap.c일부
#elif MULTILEVEL_SCHED
  // if the SCHED_POLICY is MULTILEVEL_SCHED,
// 만약 pid가 짝수라면 RR방식이니 yield()하고
// 홀수라면 FCFS로 계속 선점하게 하되 200tick이 넘어갔을 때 프로세스를 kill한다.
// 이렇게 되면 다음 tick에 trap에서 kill 프로세스를 없애고 다음 스케줄링이 진행된다.
  if(myproc() && ((myproc()->pid)%2==0) && myproc()->state == RUNNING &&
     tf->trapno==T_IRQ0+IRQ_TIMER ){
    yield();
  }else if(myproc() && (myproc()->pid)%2==1){
    if(myproc()->state==RUNNING && tf->trapno==T_IRQ0+IRQ_TIMER &&
     (ticks - myproc()->allocated_time) >= 200){
      cprintf("200ticks over... kill %d process\n", myproc()->pid);
      kill(myproc()->pid);
    }
  }
```

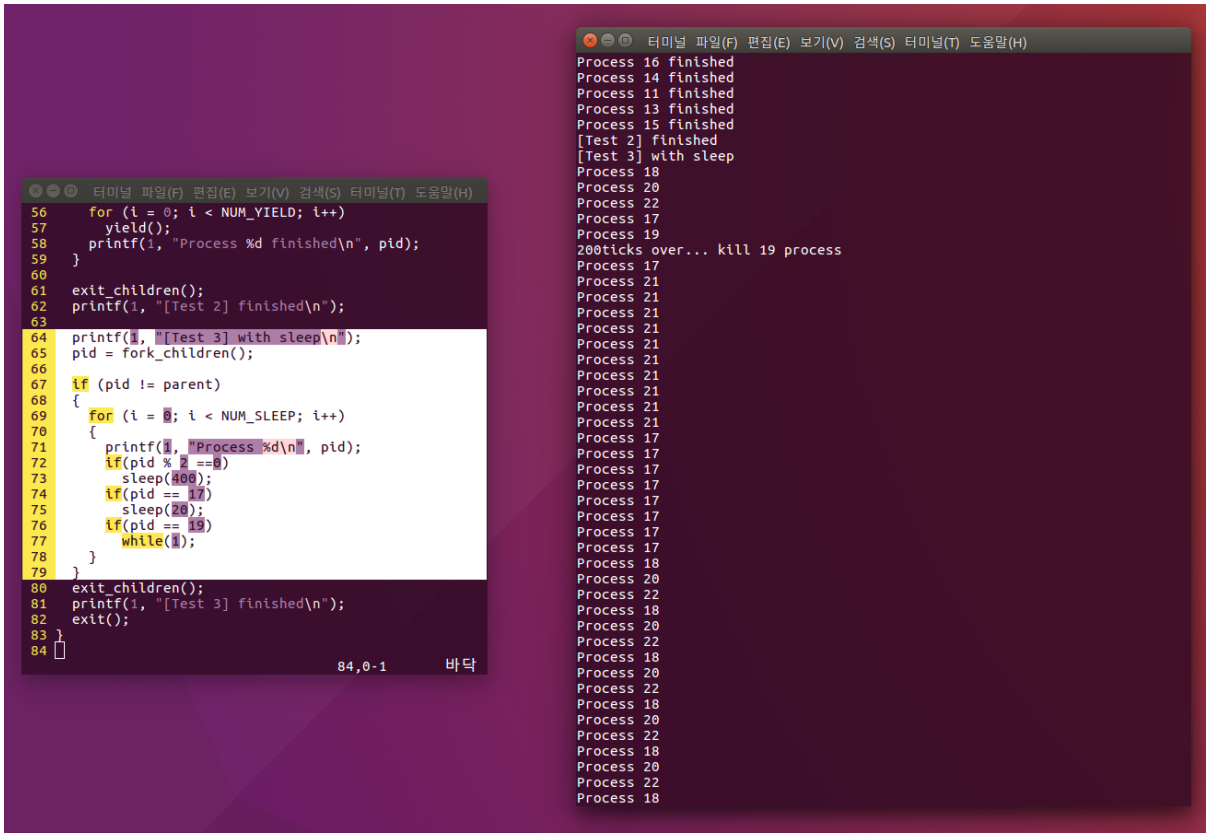
-실행결과-

다음은 p2_ml_test.c파일을 실행한 결과이다. 문제없이 스케줄링이 진행되었다.

1.yield와 sleep없이 진행하였을 경우 첫번째 이미지에서 볼 수 있듯이 짝수 프로세스가 RR방식으로 수행된 후에 홀수 프로세스가 FCFS방식으로 수행되며

2.두번째 사진은 짝수프로세스를 전부 sleep시켜둔 후에 17번 프로세스를 sleep시키고 19번 프로세스를 계속 돌리는 코드로 변경하여 테스트 하였을 때 우선순위가 높은 RR방식의 짝수 pid가 자신의 pid를 출력후 바로 sleep에 들어가고 FCFS를 수행하는 홀수 pid프로세스는 17번이 sleep에 들어가며 그결과 19번 프로세스가 CPU를 차지하고 있다가 200tick이 지나 kill되고 홀수 프로세스들이 실행되는 도중에 짝수 프로세스들이 깨어나 RR방식으로 스케줄링 되고있는 모습이다.





1-4. MLFQ(Multi Level Feedback Queue)

요구사항 :
- 2-level feedback queue + disabling /enabling preemption
- L0, L1 두 queue로 이루어져 있으며 L0의 우선순위가 더 높다.
- 각 queue는 각각 다른 time quantum을 가진다.
- 처음 실행된 프로세스는 모두 L0에 들어간다.
- Starvation을 막기 위해 priority boosting이 구현되어야 한다.

- L0 는 기본적으로 RR정책을 수행한다. 그리고 L0큐의 모든 프로세스가 SLEEPING상태가 되면 L1의 프로세스를 수행한다.
- L1 큐는 priority scheduling을 한다. 우선순위를 설정할 수 있는 시스템 콜인 setpriority()를 추가해야 하며 프로세스의 처음 priority는 0이며 0~10사이의 값을 설정할 수 있다. 값이 클수록 우선순위가 높다.
- 우선순위가 같을 경우 FCFS로 우선순위가 정해진다.
- Prioirty 는 L1에서만 유효하고 L0에서는 아무런 역할을 하지 않는다.

- 각 레벨에서 주어진 시간동안 실행후 switch된다.
- L0에서 프로세스가 time quantum을 모두 썼을 때 L1으로 간다.
- L1에서 quantum을 모두 쓸 경우 priority값이 1 감소하며 0이 최소이다.
- L0 : 4ticks ㅣL1 : 8ticks

- MLFQ스케줄링을 무시하고 프로세서를 독점하는 monopolize 시스템 콜을 만들며 자신의 학번을 암호로 받아 실행하고 만약 암호가 일치하지 않으면 프로세스를 강제 종료한다.
- Monopolize가 수행될때는 MLFQ가 일어나지 않는다.
- monopolize를 다시 한번 호출 하면 기존의 MLFQ로 돌아가며 해당 프로세스는 L0에 Priority =0 인상태로 배정된다. 여기서도 암호가 틀리면 종료되어야 한다.

구현사항 :
1. 요구사항을 읽어보고 proc 구조체에 어떤 값이 더 필요할까 고민해본 결과, int priority(프로세스의 Priority check를 위함), char lev(L0 queue에 속해있는지 L1 queue에 속해있는지 확인하기 위한 변수), char check_monopolize(monopolize가 선택되었는지 체크하기 위한 변수)를 만들어 넣었다.
또한 이 변수 값들의 초기화는 allocproc의 found: 아랫부분에서 진행하였다. ptable.lock을 해제하기 전에 모든 값을 0(처음에 만들어진 프로세스는 L0에 priority 0인 상태로 독점 권한 없이 배정되기 때문..

2. 그 다음 요구사항에서 구현하라고 한 System call들, getlev, setpriority, monopolize를 구현하였다. 구현에 이용한 파일들과 코드는 다음과 같다. 코드의 설명은 주석에 자세히 적어두었다. 또한 어디에 추가해야 하는지는 cscope를 사용하여 getpid가 정의된 부분들을 참고하였다.

```
//defs.h
int             setpriority(int, int);
int             getlev(void);
void            monopolize(int);

//proc.c
//pid와 일치하는 process를 찾아 주어진 priority로 설정한다.
//만약 pid와 일치하는 process가 없거나 자신과 자신의 자식프로세스 이외의
//프로세스의 priority를 변경하려고 할 경우 명세서의 요구대로 -1을 반환한다.
// priority값이 0이상 10이하의 수 임은 sys_setpriority가 보장해준다.
// 만약 성공정으로 값을 설정했을 경우 0을 반환한다.
int
setpriority(int pid, int priority)
{
  struct proc * p;
  struct proc * sp=0;
```

```
  acquire(&ptable.lock);
  for(p=ptable.proc; p<&ptable.proc[NPROC];p++){
    if(p->pid==pid){
      sp=p;
      break;
    }
  }
  //if there is no process have argumnet pid return -1
  if(sp==0){
    release(&ptable.lock);
    return -1;
  }
//이 부분에서 프로세스가 나 혹은 내 자식 프로세스인지 아닌지 확인한다.
  if(sp->parent != myproc() && sp != myproc()){
    cprintf("this is not my child process\n");
    release(&ptable.lock);
    return -1;
  }
  // if success to set priority return 0;
  sp->priority = priority;
  release(&ptable.lock);
  return 0;
}

//MLFQ 스케줄링 과정에서 L0에 속해있는지 L1에 속해있는지 확인후 값을 반환한다.
//이 값은 0또는 1으로 proc구조체의 lev값에 저장되어있기에 이 값을 그대로 돌려준다.
int
getlev(void)
{
  return myproc()->lev;
}

//monopolize는 password값을 받아서 일치할 경우 독점권한을 부여하는 함수이다.
// 처음에 password값을 문자열로 받았으나 학번은 항상 0으로 시작하지 않기 때문에
// int 자료형을 사용하였다.
// 기본적인 동작구조는 check_monopolize 값이 0일 때 즉 독점권한이 없을 때
// 독점권한을 부여하고 비밀번호가 틀렸을 때 프로세스를 죽인다.
// 반대로 값이 1이라면 독점권한을 해제하고 L0큐에 priority 값이 0인 채로 집어넣는다.
// 이때도 비밀번호가 틀릴경우 프로세스를 죽인다.
void
monopolize(int password)
{
  acquire(&ptable.lock);
  if(myproc()->check_monopolize==0){
    if(password == 2015005169){
      myproc()->check_monopolize=1;
    }else{
      cprintf("password is wrong. Kill %d process\n",myproc()->pid);
//이부분은 처음에 kill(myproc()->pid)로 구현하였다가 수정하였다.
      myproc()->killed = 1;
      // Wake process from sleep if necessary.
      if(myproc()->state == SLEEPING)
        myproc()->state = RUNNABLE;
    }
//이부분은 처음에 kill(myproc()->pid)로 구현하였다가 수정하였다.
  }else if(myproc()->check_monopolize==1){
    if(password==2015005169){
      myproc()->check_monopolize=0;
      myproc()->lev=0;
      myproc()->priority=0;
    }else{
      myproc()->check_monopolize=0;
      cprintf("password is wrong. Kill %d process\n",myproc()->pid);
//이부분은 처음에 kill(myproc()->pid)로 구현하였다가 수정하였다.
      myproc()->killed = 1;
      // Wake process from sleep if necessary.
      if(myproc()->state == SLEEPING)
        myproc()->state = RUNNABLE;
//이부분은 처음에 kill(myproc()->pid)로 구현하였다가 수정하였다.
    }
  }
  release(&ptable.lock);
```

```
}


//syscall.c
extern int sys_getlev(void);
extern int sys_setpriority(void);
extern int sys_monopolize(void);
...
[SYS_getlev]  sys_getlev,
[SYS_setpriority]   sys_setpriority,
[SYS_monopolize]    sys_monopolize,


//syscall.h
#define SYS_getlev 25
#define SYS_setpriority 26
#define SYS_monopolize 27


//sysproc.c
int
sys_getlev(void){
  return getlev();
}

//아까 요구사항에 있던 0이상 10이하의 값인지는 이곳에서 확인해 주었다
//0미만이거나 10초과의 값이 들어올 경우 -2를 반환한다.
//이부분에서 argint()를 사용하는데 실습 수업이 큰 도움이 되었다.
int
sys_setpriority(void){
  int pid, priority;
  argint(0,&pid);
  argint(1,&priority);
  //if priority is not 0~10 then return -2
  if(priority<0 || priority>10)
    return -2;
  return setpriority(pid,priority);
}

int
sys_monopolize(void){
  int password;
  argint(0,&password);
  monopolize(password);
  return 0;
}

//user.h
int getlev(void);
int setpriority(int, int);
void monopolize(int);

//usys.h
SYSCALL(getlev)
SYSCALL(setpriority)
SYSCALL(monopolize)
```

3. 그 다음은 스케줄러 함수 sched_MLFQ를 구현하였다.

```
void
sched_MLFQ(void)
{
  struct proc *p;
  struct cpu *c =mycpu();
  c->proc=0;
  int check_L0_proc;
  struct proc *l1_proc;
  for(;;){
    sti();
```

```
    acquire(&ptable.lock);
//L0에 프로세스가 몇 개 있는지확인하기 위해 check_L0_proc을 0으로 초기화하고
//L1 큐의 priority scheduling을 위해 l1_proc을 0으로 초기화한다.
    check_L0_proc=0;
    l1_proc=0;
    //check how many L0 processes exists.
    for(p=ptable.proc;p<&ptable.proc[NPROC];p++){
      if((p->lev==0) &&(p->state==RUNNABLE)){
        check_L0_proc++;
      }
    }
    //if there is more than 1 L0 process do RR
//만약 L0프로세스가 존재한다면 ptable을 탐색하여 RUNNABLE이 아니거나 L1프로세스를 skip하고
//해당하는 프로세스들만(L0의 RUNNABLE프로세스들) RR방식으로 스케줄링을 진행하였다.
    if(check_L0_proc>0){
      for(p=ptable.proc;p<&ptable.proc[NPROC];p++){
        if((p->state != RUNNABLE) || (p->lev==1))
          continue;
 //time quantum check를 위해 allocacted_time에 ticks를 저장한다.
        p->allocated_time=ticks;
        c->proc =p;
        switchuvm(p);
        p->state=RUNNING;

        swtch(&(c->scheduler), p->context);
        switchkvm();
        c->proc=0;
      }
    }else{
      // 이 부분은 check_L0_proc값이 0일때이다.
      // 즉 L1 queue에 해당하는 프로세스들이 여기서 스케줄링된다.
      for(p=ptable.proc;p<&ptable.proc[NPROC];p++){
        if((p->state != RUNNABLE)|| (p->lev==0))
          continue;
        if(l1_proc==0){
          //if l1_proc==0, it means p is first L1 process
          l1_proc=p;
        }else{
          //priority값이 더 큰 프로세스를 l1_proc에 저장한다.
          if(l1_proc->priority < p->priority){
            l1_proc=p;
          }else if(l1_proc->priority == p->priority){
            // if two processes have same priority
            // FCFS is applied
            if(p->pid < l1_proc->pid)
              l1_proc=p;
          }
        }
      }
      //i루프를 빠져나오게 되면 priority값이 가장 높은 프로세스가 선택이된다.
//그 다음은 swtch과정을 거쳐 스케줄링을 진행한다.
      if(l1_proc!=0){
        l1_proc->allocated_time=ticks;
        c->proc=l1_proc;
        switchuvm(l1_proc);
        l1_proc->state=RUNNING;
        swtch(&(c->scheduler),l1_proc->context);
        switchkvm();
        c->proc=0;
      }
    }
    release(&ptable.lock);
  }
}
```

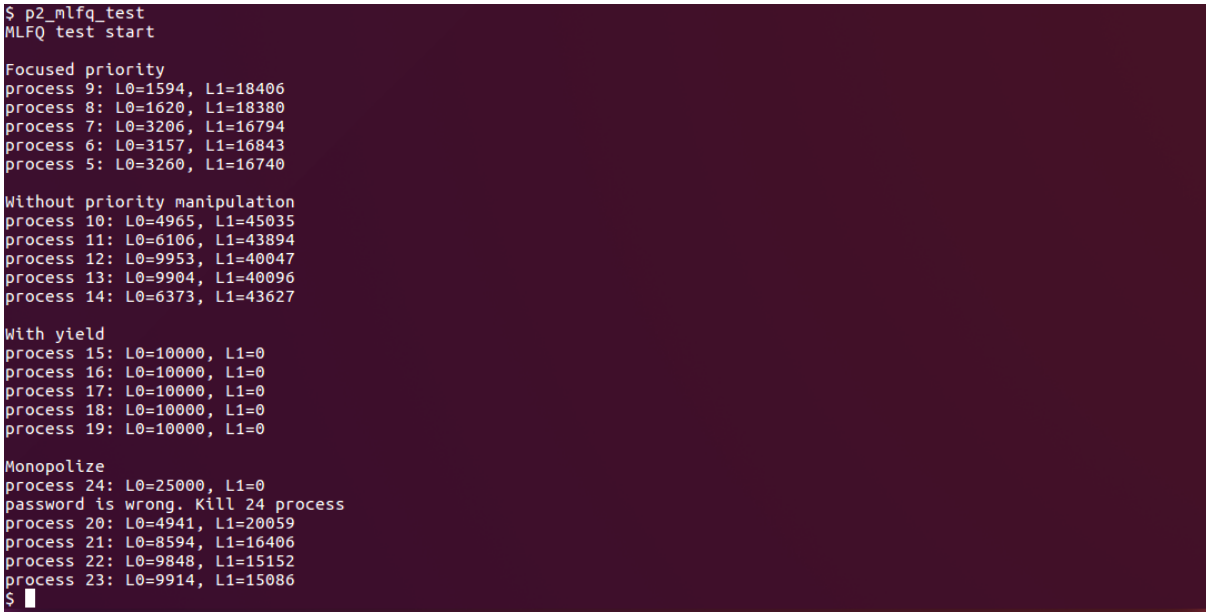
4. 스케줄링 과정에 있어서 L0와 L1에 적절한 timer interrupt과정이 일어날 수 있도록 trap.c코드를 작성해준다. 자세한 사항은 주석으로 달았다. 이 trap.c에서 사용하는 함수들(chagelev, changepriority, priorityboost)은 process의 값을 바꾸기 때문에 ptable.lock을 획득해야하여 proc.c에 구현하였다.

```
#elif MLFQ_SCHED
  //우선은 L0큐에 존재하는 프로세스들을 위한 time interrupt처리이다.
  //만약 프로세스가 RUNNING상태이고 L0큐에 속하며, 독점상태가 이니고 FCFS에서와 같이
//allocated_time을 사용하여 계산한 값이 주어진 time quantum인 4ticks 이상이 되었을 때
// changelev()를 통해 0에서 1로 옮겨주고 cpu를 양보한다.
  if(myproc()&&myproc()->state==RUNNING && myproc()->lev==0 &&
     myproc()->check_monopolize!=1 && (ticks-myproc()->allocated_time >=4)){
    changelev();
    yield();
  }
  //이부분은 L1의 priority scheduling을 위한 부분이다.
  //만약 프로세스의 상태가 RUNNING이고 L1에 속해있고 독점상태가 아니면서
//allocated_time을 이용하여 계산한 값이 8ticks를 넘어갔다면
//changepriority를 이용하여 우선순위를 낮추고 cpu를 양보한다.
  if(myproc()&&myproc()->state==RUNNING && myproc()->lev==1 &&
     myproc()->check_monopolize!=1 && (ticks-myproc()->allocated_time >=8)){
    changepriority();
    yield();
  }
//starvation을 막기위해 요구사항에 200ticks마다 priority boost를 하라고 하였기 때문에
//proc.c에 구현한 priorityboost()를 호출한다.
  if(ticks %200==0)
    priorityboost();
  #endif
```

```
//defs.h
void            changelev(void);
void            changepriority(void);
void            priorityboost(void);

//proc.c
void
changelev(void)
{
  acquire(&ptable.lock);
  if(myproc()->lev==0)
    myproc()->lev=1;
  else
    myproc()->lev=0;
  release(&ptable.lock);
  return;
}

void
changepriority(void)
{
  acquire(&ptable.lock);
  if(myproc()->priority>0)
    myproc()->priority--;
  release(&ptable.lock);
  return;
}

//In requirements every 200ticks all processes move to L0 and set priority0
void
priorityboost(void)
{
  struct proc *p;
  acquire(&ptable.lock);
  for(p=ptable.proc;p<&ptable.proc[NPROC];p++){
  //it is already have ptable.lock
// so not use changelev and changeprioirty function but change directly
    p->lev=0;
    p->priority=0;
  }
  release(&ptable.lock);
  return;
}
```

-실행결과-

mlfq test는 1.priority를 부여한 방식, 2.priority 조작 없이 진행한 방식 3.yield()를 사용한 방식, 4.마지막 child process에 monopolize를 적용한 방식으로 진행되었으며
1. 코드상 pid값이 더 큰 프로세스가 더 높은 priority를 가졌기 때문에 pid가 큰 프로세스가 먼저 끝나게 된것을 확인할 수 있다.
2. 우선순위를 지정하지 않았을 경우 모두 priority는 0이며 이 경우 L1에서 FCFS로 pid가 작은 프로세스가 먼저 스케줄링 되기 때문에 pid가 작은 프로세스가 먼저 끝나게 되는 것을 확인할 수 있다.
3. yield()를 사용할 경우 L0에 계속 남아 작업이 일어나기 때문에 모두 L0에서 수행됨을 확인할 수 있다. 다만 이 때 모두 비슷하게 작업을 진행하기 때문에 동시에 프로세스들의 작업이 끝났다는 점이 1,2,4와 달랐다.
4. 마지막 child process가 독점으로 먼저 L0에서 작업을 끝내고 나머지 프로세스들이 스케줄링을 통해 작업을 마치는 것을 확인할 수 있다.



컴파일 &Trouble Shooting

<Complie>: 아무것도 손대지 않은 상태에서 $ make clean후를 전제로 함

```
//기본 RR방식
$ make
$ make fs.img
$ qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m 512
//FCFS test
// $ make SCHED_POLICY=FCFS_SCHED CPUS=1 도 가능 밑에도 모두 테스트결과 이상없음
$ make SCHED_POLICY=FCFS_SCHED
$ make fs.img
$ qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m 512
//MLQ test
$ make SCHED_POLICY=MULTILEVEL_SCHED
$ make fs.img
$ qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m 512
//MLFQ test
$ make SCHED_POLICY=MLFQ_SCHED
$ make fs.img
$ qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m 512
```

<FCFS>

1. void scheduler(void)에 처음에 #endif이후에 아무런 값이 없었는데 그랬더니 컴파일 도중 에러가 발생하였다. 그 이유는 defs.h 에 scheduler가 void scheduler(void) __attribute__((noreturn)); 으로 선언되어있었으며 이 의미는 이 함수를 실행하였을 때 무한루프가 존재하며 빠져나오지 않는다는 것을 의미였으며 따라서 scheduler 내부에 무한루프가 존재하지 않아 에러가 발생하는 것이었다.
이는 마지막에 while(1);을 추가하여 해결하였다.

2. FCFS 스케줄링을 위하여 sched_FCFS를 구현할 때 컴파일에는 문제가 없었으나 xv6를 실행시 아래와 같은 에러를 내며 멈추었다.

```
if(fcfs_proc!=0){
        fcfs_proc->allocated_time=ticks;
        c->proc=fcfs_proc;
        switchuvm(fcfs_proc);
        fcfs_proc->state=RUNNING;
        swtch(&(c->scheduler),fcfs_proc->context);
        switchkvm();
        c->proc=0;
      }
```

원인은 처음 구성시 sched_FCFS에서 위 코드부분을 if(fcfs_proc !=0)으로 감싸지 않았기 때문이다. fcfs_proc에 할당 된게 없는데 swtch시키면서 에러가 발생하였다. 즉 0일때 swtch가 되지 않게 만들었더니 해결되었다.

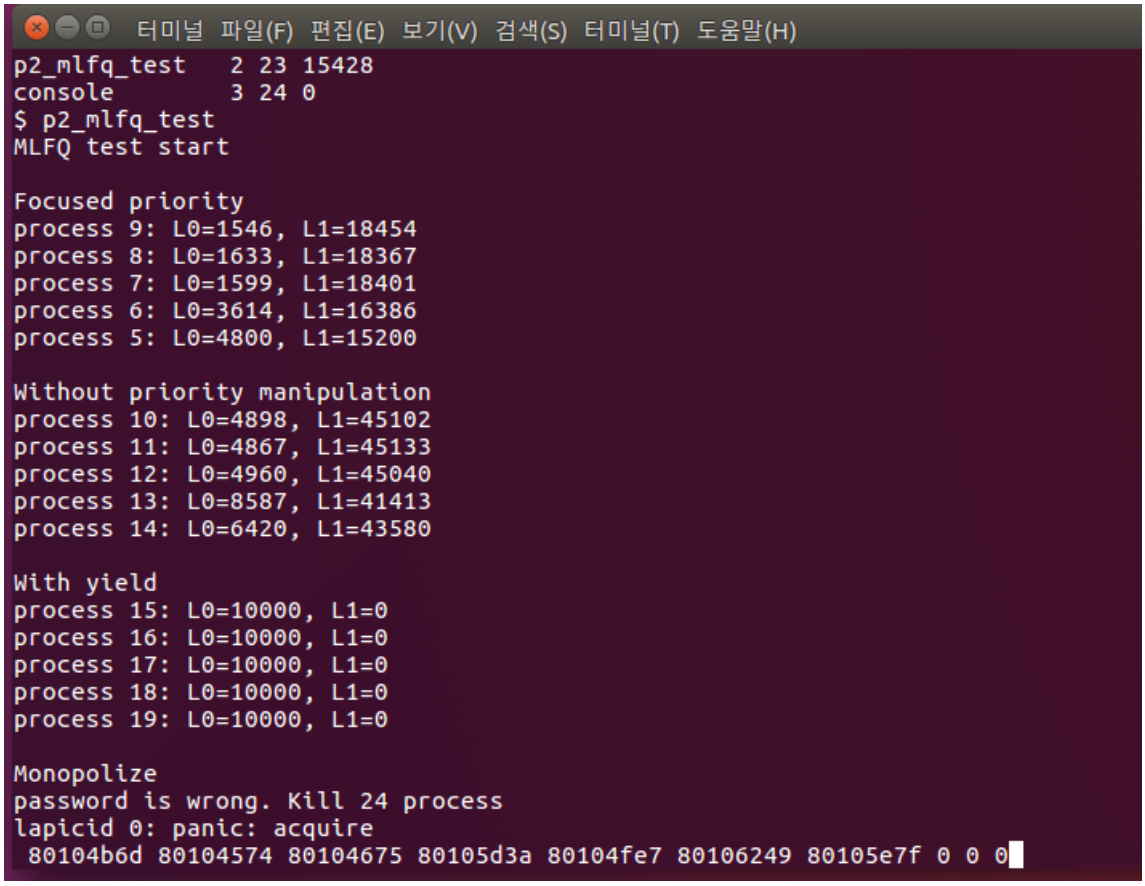
```
nakevin96@Cowl:~/xv6-public$ qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m 512
WARNING: Image format was not specified for 'fs.img' and probing guessed raw.
         Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
WARNING: Image format was not specified for 'xv6.img' and probing guessed raw.
         Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
unexpected trap 14 from cpu 0 eip 80103bad (cr2=0x7c)
lapicid 0: panic: trap
 8010628a 80105e4f 80103c0b 80102e2f 80102f6f 0 0 0 0 0
```

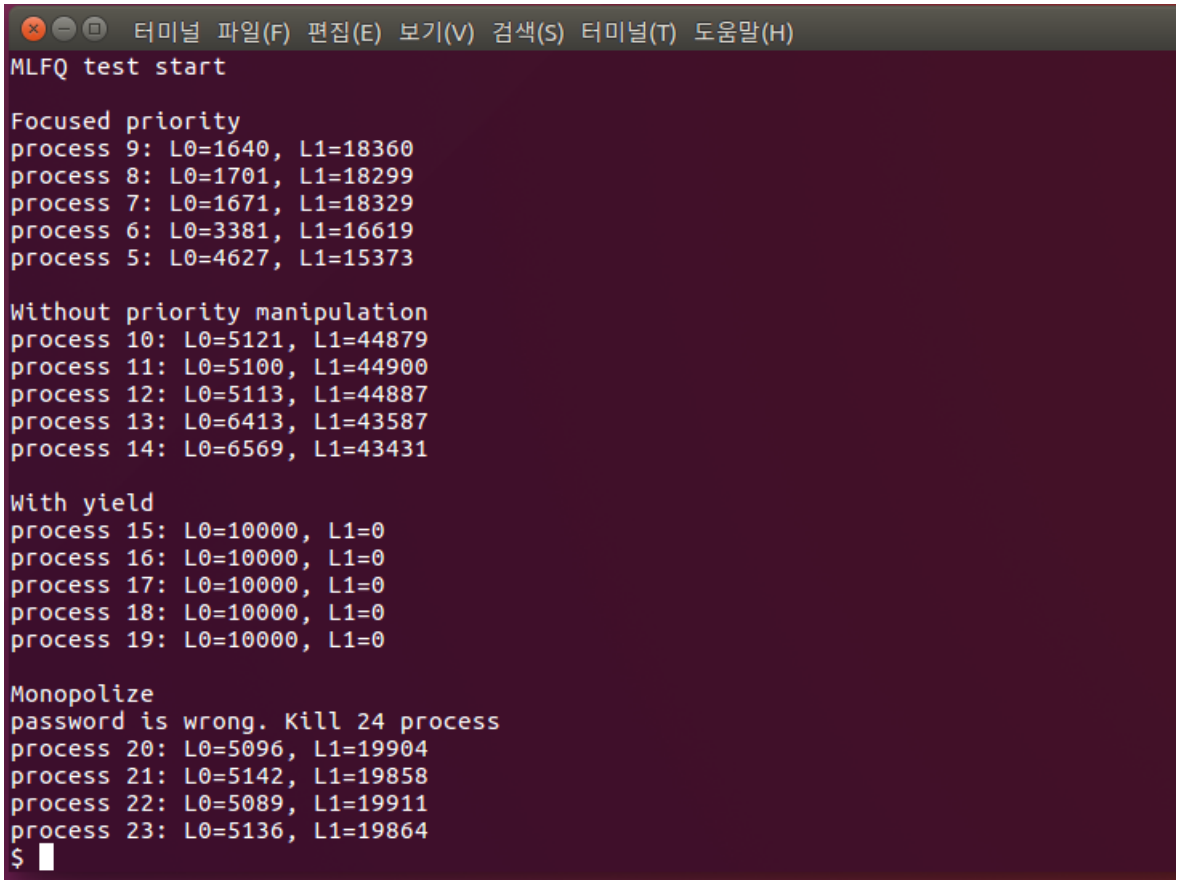
<MLFQ>

1. 컴파일시에 계속하여 changelev, changepriority, priorityboost함수 부분에 에러(implicit declaration of function)가 발생하였다. 해당 에러는 함수를 명시적으로 선언하지 않은 채로 사용했다는 의미였으며 처음에는 자료형을 잘못 명시하였거나 매개변수를 잘못 쓴줄 알고 해맸다. 문제는 함수선언을 proc.c 상단에 하고 defs.h에 선언을 안해주어 참조가 불가능했기 때문이었다. 따라서 proc.c에 적었던 함수 선언을 defs.h에 옮기며 문제가 해결되었다.

2. monopolize함수를 구현하는데 처음에 비밀번며호가 틀렸을때 kill(myproc()→pid)로 구현하였더니 다음과 같은 에러가 발생하였다. lapicid가 interrupt에 관련이 되어있고 panic : acquire라고 나와있어 ptable.lock을 두번 acquire시도 하여서 그런가 하고 proc.c의 kill의 acquire(&ptable.lock) release(&ptable.lock)사이의 코드를 가져와 대체하였더니 문제없이 작동하게 되었다.

사실 코드를 짜면서 겪는 어려움보다. 운영체제의 동작과정을 완벽하게 이해하지 못하여 proc.c, trap.c 부분을 파악하는게 정말 오랜시간이 걸렸다. 따라서 xv6를 분석하는데 오랜 시간이 걸렸고 그 때 정리한 내용을 상단에 기술해두었다.


터미널 파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
p2_mlfq_test 2 23 15428
console 3 24 0
$ p2_mlfq_test
MLFQ test start
Focused priority
process 9: L0=1546, L1=18454
process 8: L0=1633, L1=18367
process 7: L0=1599, L1=18401
process 6: L0=3614, L1=16386
process 5: L0=4800, L1=15200
Without priority manipulation
process 10: L0=4898, L1=45102
process 11: L0=4867, L1=45133
process 12: L0=4960, L1=45040
process 13: L0=8587, L1=41413
process 14: L0=6420, L1=43580
With yield
process 15: L0=10000, L1=0
process 16: L0=10000, L1=0
process 17: L0=10000, L1=0
process 18: L0=10000, L1=0
process 19: L0=10000, L1=0
Monopolize
password is wrong. Kill 24 process
lapicid 0: panic: acquire
80104b6d 80104574 80104675 80105d3a 80104fe7 80106249 80105e7f 0 0 0


터미널 파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
MLFQ test start
Focused priority
process 9: L0=1640, L1=18360
process 8: L0=1701, L1=18299
process 7: L0=1671, L1=18329
process 6: L0=3381, L1=16619
process 5: L0=4627, L1=15373
Without priority manipulation
process 10: L0=5121, L1=44879
process 11: L0=5100, L1=44900
process 12: L0=5113, L1=44887
process 13: L0=6413, L1=43587
process 14: L0=6569, L1=43431
With yield
process 15: L0=10000, L1=0
process 16: L0=10000, L1=0
process 17: L0=10000, L1=0
process 18: L0=10000, L1=0
process 19: L0=10000, L1=0
Monopolize
password is wrong. Kill 24 process
process 20: L0=5096, L1=19904
process 21: L0=5142, L1=19858
process 22: L0=5089, L1=19911
process 23: L0=5136, L1=19864
$