20192392 컴퓨터학부 김한승

1. 개요

본 과제를 통해서 XV6 상에 프로세스를 관리(프로세스의 생애) 및 기본 스케쥴러 와 새로운 스케쥴러 구현을 통한 스케쥴링 기법에 대해서 이해하였습니다.

기존 스케쥴러는 RR(Round- Robin) 방식이며, 상세 설계에서 기존 XV6 스케쥴러에 대해 콜 그래프 와 함께 상세하게 설명하겠습니다. 또한, 새로운 스케쥴러는 프로세스의 우선순위 및 프로세스가 마지막으로 우선순위를 조정한 후 지난 시간(priority\_tick) 을 통해 우선순위를 재 계산합니다.

기존 스케쥴러에 time quantum은 1tick 이었으며, 이를 새로운 스케쥴러에서는 30tick으로 설정하였습니다.

우선순위가 높은 프로세스가 우선적으로 수행되기 때문에 starvation(기아) 현상이 발생할 수 있으며,

우선순위 값은 0 ~ 99 사이의 값을 갖습니다. 이를 통해 우선순위가 재 계산 될떄 99 이상의 값으로 계산 된다면 최댓값인 99를 유지 하도록 하였습니다. 이때, pid == 1 과 pid == 2 인 IDLE 프로세스의 경우에는 우선순위의 초기값이 99입니다.

또한, 프로세스가 SLEEPING 상태에서 RUNNABLE 상태가 되는 wakeup(깨움) 이 되더라도 IDLE 프로세스는 우선순위를 99로 유지합니다. 그 외 프로세스는 wakeup시, 우선순위를 0으로 조정하였습니다.

새로 만든 스케쥴러는 런큐를 통해 프로세스들을 관리하며, 런큐는 인덱스마다 큐 구조체를 가집니다.

큐 구조체는 head 와 tail을 가진 링크드 리스트 이며, 프로세스들의 링크드 리스트를 관리하고 있습니다.

런큐의 인덱스는 0 ~ 24 총 25개의 인덱스를 가지며 현재 프로세스의 priority / 4를 인덱스로 해당 큐 구조체의 링크드 리스트로 들어가게됩니다.

이때, 1.프로세스가 wakeup 되어서 우선순위가 조정 될 경우, 2.프로세스들의 총 CPU 사용시간 주기 60tick 마다 우선순위가 재조정 될 경우 3.set\_sche\_info()를 통한 우선순위 조정과 같이 기존 런큐에 프로세스가 있을 때 우선순위가 변하 하는 경우 기존 우선순위에 해당하는 인덱스에서 해당 프로세스를 링크드리스트에서 제거한 후, 새로운 인덱스의 링크드리스트로 재 배치합니다.

새롭게 런큐에 들어오는 프로세스들 또한 자신의 우선순위에 맞는 인덱스의 링크드 리스트에 추가하였습니다.

링크드 리스트에 새로운 인덱스의 재 배치 하는 것 외에 프로세스가 종료될 경우에는 런 큐에서 완전히 제거하였습니다.

exit() 이후 ZOMBIE 상태가 되는 프로세스의 경우에는 런큐에서 제외하였습니다.

상세 설계의 목차로 2-1.기존 XV6 스케쥴러에 대한 분석 2-2. SSU scheduler 분석에 대한 설계 2-3. set\_sche\_info() 시스템 콜 추가 및 설계 2-4. 다음에 실행 될 프로세스 선정 과정에 디버깅 기능을 추가하는 과정 설계 2-5에서 1,2,3,4 를 활용하여 XV6 기존 스케쥴러(Round Robin) 스케쥴러와 SSU 스케쥴러의 성능을 비교 분석하였습니다.

기타 참고사항인 CPU 코어 개수 1개 제한 또한 Makefile 수정을 통해 수정하였습니다.

2. 상세 설계

2-1. 기존 XV6 스케쥴러에 대한 분석 (1번 요소)

2-1에서 분석을 위해 사용하는 코드는 기존 XV6에 스케쥴러 변경을 위해 코드 수정을 하기 전 원본 코드입니다.

2-1-1. 스케쥴러 scheduler() 호출 과정

Scheduler() 함수 즉, Round Robin 방식으로 다음 실행될 RUNNABLE한 프로세스를 찾아서 mycp()->scheduler와 문맥교환을 하는 함수(다음 번 실행될 프로세스를 선택) 스케쥴링 하는 함수를 2-1-2 에서 분석하기 전 scheduler() 함수가 어떤 함수에서 호출되는지 콜 그래프를 통해 설명하겠습니다.

운영 체제(OS) 또는 자원관리자라고 불리는 이 프로그램 또한 SoftWare(SW) 입니다. main() 함수에서 실행을 시작하는데

main() 함수에서 쉘 프로세스(pid == 2)를 호출하기 전까지 main()에서 여러 함수를 호출하는데 그 중에 mpmain() 함수를 주목하여야 합니다. 해당 함수 mpmain() 함수는 scheduler() 함수를 호출합니다.

텍스트, 스크린샷, 폰트, 문서이(가) 표시된 사진

자동 생성된 설명

2-1-2. scheduler() 함수

스케쥴러 교체 이후, 명세서에서 요구하는 출력문 2번 : context switch 시점에 해당 합니다.

여기서 주목해야 할 점은 main() 함수내에서 여러 함수를 호출하고 마지막으로 호출한 mpmain() 함수가 호출한 scheduler() 함수는 다음번 실행할 프로세스를 스케쥴링하는 함수입니다. 스케쥴링 정책은 기존 XV6에서는 RoundRobin 방식 scheduler() 함수에 초반 부분을 보면 for(;;) // 무한 루프로 감싸져있는 것을 볼 수 있습니다.

이로 통해 XV6 커널 프로세스는 scheduler() 함수를 통해 지속해서 스케쥴링을 담당합니다.

  for(;;){

    // Enable interrupts on this processor.

    sti();

    // Loop over process table looking for process to run.

    acquire(&ptable.lock);

    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;

      switchuvm(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;

    }

    release(&ptable.lock);

  }

해당 for 문내에서 스케쥴러 프로세스의 인터럽트를 허용하고 sti()

ptable은 프로세스들의 정보를 가진 프로세스 테이블입니다. 새로 구현하는 스케줄러에서는 해당 테이블이 아닌

런큐를 통해 다음 실행할 프로세스를 선택합니다.

ptable에 대한 lock을 acquire() 한 후, ptable을 순차 순회 하며 RUNNABLE 한 상태에 있는 프로세스를 찾고, 이를 c->scheduler, p->context 와 문맥교환을 합니다.

이 과정을 통해 다음 번 실행될 프로세스에 대한 준비가 끝났습니다.

그 후, ptable에 대한 lock을 release()합니다.

이렇게 코드를 통해 XV6의 기존 스케쥴러 정책은 Round Robin 스케쥴링 방식입니다. 프로세스들 사이에 우선순위를 두지 않으며, Time Quantum(Time Slicing)을 통해 다른 프로세스들과 공정하게 기회를 부여 받고, 문맥교환이 많이 이루어지지만, 그만큼 응답 시간이 짧은 장점이 있는 스케쥴링 방식입니다.

2-1-3. 프로세스의 생애

기존 XV6 스케쥴링을 위한 scheduler() 함수가 어떠한 순서로 호출되었고, 해당 스케쥴링을 하는 함수 scheduler() 에 대해서 확인을 하였습니다. 그러면 언제 실제로 스케쥴러가 선택한 실행 가능한 프로세스와 문맥교환이 이루어지는지 보기전

proc.h내에 프로세스들의 상태에 대한 구조체를 확인하겠습니다.

  enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

UNUSED 상태는 초기 세팅이 완료 되지 않은 프로세스의 상태입니다.

proc.c 내에 allocproc()을 통해 초기 변수 세팅등 프로세스에 대한 초기화가 이루어진 상태를 EMBRYO 상태입니다.

SLEEPING 상태는 I/O(BLOCKED) 된 상태입니다. wakeup() 함수를 통해 깨움 이벤트가 발생하지 않으면 해당 상태를 유지합니다.

RUNNABLE 상태는 스케쥴러가 선택 가능한(실행 가능한) 상태입니다. 해당 상태에 프로세스를 scheduler() 함수가 선택하여서 현재 실행중인 RUNNING 상태의 프로세스와 교체를 할 수 있습니다.

RUNNING 상태는 현재 수행중인 프로세스 상태입니다. 현재 프로세스가 여러 이유에 따라 스케쥴러가 선택한 스케쥴링된 프로세스와 교체 될 수 있습니다.(2-1-5~) 에서 해당 상황에 대해서 설명하겠습니다.

ZOMBIE 상태는 현재 프로세스가 종료된 상태입니다. 프로세스가 종료되는 exit() 함수에 대해서 2-1-4 에서 설명하겠습니다.

2-1-4. sched() vs scheduler()

proc.c 내부 sched() 함수와 scheduler() 함수 모두 swtch() 함수를 동일하게 호출하는 것으로 보이는데 두 함수 모두 스케쥴링에 관여를 하는 것 아닌가? 라는 잠시 고민을 하여서 분석하였습니다.

scheduler() 함수는 스케쥴링 정책에 따라 실행 가능한 프로세스(RUNNABLE) 한 프로세스 들 가운데 골라 다음 번 실행할 프로세스를 스케쥴링합니다.

struct cpu \*c = mycpu();

struct proc \*p; //ptable 순회를 통해 RUNNABLE 상태인 프로세스를 찾으면 초기화

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

switchkvm();

sched() 함수는 “현재 실행중인 프로세스” 와 “스케쥴링 하여 다음 실행할 프로세스(scheduler() 함수에서 고른 프로세스)”를 교체합니다.

struct proc \*p = myproc();

swtch(&p->context, mycpu()->scheduler);

즉 해당 과제에서 명세서에서 scheduler() 함수에 새로 구현한 SSU 스케쥴러를 통해 실행할 프로세스를 고르는 부분(스케쥴러 부분)을 코드 수정 후, 2번 상황 Context Switch 직전 상황을 cprintf()를 통해 출력합니다.

다음번 실행될 프로세스를 고르는 함수는 scheduler() 함수

스케쥴러 함수 scheduler()를 통해 선택된 프로세스 (mycpu()->scheduler) 와 현재 수행중인 프로세스를 교체하는 함수는 sched() 함수입니다.

커널 프로세스에서 scheduler() 함수 내 for(;;) 를 통해 다음 실행될 프로세스를 결정하고,

타이머 인터럽트 , 프로세스 종료(exit()) , 프로세스 sleep(sleep())에 경우마다 sched() 함수를 호출하여 RUNNING 상태의 프로세스를 문맥 교환합니다.

2-1-5. sched() 함수 호출 1 : 프로세스 종료 exit()

스케쥴러 교체 이후, 명세서에서 요구하는 출력문 3번 :프로세스 종료 시점에 해당하므로 추가로 분석하였습니다

프로세스가 종료될 상황이 되면(killed 비트가 1이 된 경우등 trap.c exit() 호출), 사용자 측에서 exit() 함수 사용을 통해 프로세스 종료 포함.

해당 프로세스의 ofile을 모두 닫습니다. 부모 프로세스가 있을수 있으니, wakeup1(curproc->parent); 를 통해 꺠우고

그 후, 종료될 자식 프로세스들의 부모를 initproc으로 변경 후, wakeup1(initproc) 현재 프로세스의 자식 프로세스들을 initproc에게 넘깁니다.

현재 종료시킬 프로세스의 부모 / 자식 프로세스에 관한 처리를 완료 한 이후 ZOMBIE 상태로 변환 한 후 sched() 함수를 호출합니다.

void

exit(void)

{

  struct proc \*curproc = myproc();

  struct proc \*p;

  int fd;

  if(curproc == initproc)

    panic("init exiting");

  // Close all open files.

  for(fd = 0; fd < NOFILE; fd++){

    if(curproc->ofile[fd]){

      fileclose(curproc->ofile[fd]);

      curproc->ofile[fd] = 0;

    }

  }

  begin\_op();

  iput(curproc->cwd);

  end\_op();

  curproc->cwd = 0;

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().

  wakeup1(curproc->parent);

  // Pass abandoned children to init.

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

    if(p->parent == curproc){

      p->parent = initproc;

      if(p->state == ZOMBIE)

        wakeup1(initproc);

    }

  }

  // Jump into the scheduler, never to return.

  curproc->state = ZOMBIE;

  sched();

  panic("zombie exit");

}

2-1-6. sched() 함수 호출 2 : Timer interrupt

스케쥴러 교체 이후, 명세서에서 요구하는 출력문 1번 : 타임 퀀텀 종료 시점에 해당하므로 추가로 분석하였습니다

타이머 인터럽트란 TIME QUANTUM(time slicing)이 끝난 후, 인터럽트가 걸린 것입니다. 스케쥴링 정책에 따른 다음 실행할 프로세스(스케쥴링 된 프로세스)와 문맥 교환을 합니다. 이 부분은 trap.c trap() 함수에서 해당 부분을 확인 합니다.

타이머 인터럽트 시 yield() 함수를 호출합니다.

// 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();

proc.c 내 yield() 함수는

// Give up the CPU for one scheduling round.

void

yield(void)

{

  acquire(&ptable.lock);  //DOC: yieldlock

  myproc()->state = RUNNABLE;

  sched();

  release(&ptable.lock);

}

ptable 에 대한 lock을 acquire 한 후

현재 실행 중인 프로세스를 RUNNABLE 한 상태로 변경 후 , sched() 함수를 호출합니다.

그 후, ptable에 대한 lock을 release 합니다.

proc.c 내 sched() 함수는 스케쥴러가 선택한 스케쥴링 한 프로세스인 mycpu()->scheduler와 현재 실행 중인 프로세스를

문맥 교환하여서 성공적으로 다음 실행될 프로세스를 실행합니다.

// Enter scheduler.  Must hold only ptable.lock

// and have changed proc->state. Saves and restores

// intena because intena is a property of this

// kernel thread, not this CPU. It should

// be proc->intena and proc->ncli, but that would

// break in the few places where a lock is held but

// there's no process.

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;

}

텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명

switch.S 내 swtch()

# Context switch

#

#   void swtch(struct context \*\*old, struct context \*new);

#

# Save the current registers on the stack, creating

# a struct context, and save its address in \*old.

# Switch stacks to new and pop previously-saved registers.

.globl swtch

swtch:

  movl 4(%esp), %eax

  movl 8(%esp), %edx

  # Save old callee-saved registers

  pushl %ebp

  pushl %ebx

  pushl %esi

  pushl %edi

  # Switch stacks

  movl %esp, (%eax)

  movl %edx, %esp

  # Load new callee-saved registers

  popl %edi

  popl %esi

  popl %ebx

  popl %ebp

  ret

2-1-7. sched() 함수 호출 3 : sleep()

현재 프로세스를 SLEEPING 상태로 바꾼 후, wakeup() 함수 사용시 프로세스를 다시 RUNNABLE 상태로 깨우기 위해 쓸

p->chan 변수를 초기화 합니다. 그 후, sched() 함수를 통해 현재 프로세스를 다음 실행할 프로세스와 교체합니다.

void

sleep(void \*chan, struct spinlock \*lk)

{

  struct proc \*p = myproc();

  if(p == 0)

    panic("sleep");

  if(lk == 0)

    panic("sleep without lk");

  if(lk != &ptable.lock){  //DOC: sleeplock0

    acquire(&ptable.lock);  //DOC: sleeplock1

    release(lk);

  }

  // Go to sleep.

  p->chan = chan;

  p->state = SLEEPING;

  sched();

  // Tidy up.

  p->chan = 0;

  // Reacquire original lock.

  if(lk != &ptable.lock){  //DOC: sleeplock2

    release(&ptable.lock);

    acquire(lk);

  }

}

2-2. SSU\_SCHEDULER 구현 (2번 요소)

2-2-1. proc 구조체 추가 변수, 큐 구조체, proc.c 내 allocproc()

// 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

  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)

  // 설계 과제 3을 위한 추가 변수

  int priority;                 // 프로세스의 우선순위

  int proc\_tick;                // 현재 프로세스가 차지한 CPU tick수

  int cpu\_used;                 // 총 CPU 이용 시간

  int priority\_tick;            // 마지막으로 우선순위 재조정 한 후 tick

  struct proc\* next;

  int is\_time\_set;              // set\_sche\_info() 초기 설정된 프로세스 인가?

  int remaining\_ticks;          // 남은 tick

};

// run queue 구현을 위한 큐 구조체 해당 큐를 통해 프로세스의 링크드 리스트를 구현합니다.

// 각 큐는 프로세스를 가르킬 head와 tail이 있습니다.

typedef struct queue{

    struct proc\* head;

    struct proc\* tail;

}queue;

proc.c 내 runqueues 런큐 배열

queue runqueues[RUN\_QUEUE\_SIZE]; // 링크드리스트를 관리하는 큐의 배열

RUN\_QUEUE\_SIZE 는 길이 25를 뜻하며, queue 배열을 통해 인덱스마다 우선순위에 해당하는 프로세스들의 링크드리스트를 갖습니다.

명세서에 명시되지 않은 추가 proc 구조체 내 변수로는 링크드 리스트 구현을 위해 만든 struct proc\* next; 다른 프로세스와의 링크드 리스트 연결을 위한 변수와 is\_time\_set 과 remaining\_ticks는 set\_sche\_info()를 통해 초기화 할 변수 입니다.

is\_time\_set은 set\_sche\_info()를 통해 우선순위와 남은 실행 시간이 설정되 있는가를 나타내는 값입니다.

allocproc() 함수내 반환할 proc 구조체 변수 값을 초기화 하는 이유는

userinit() 과 fork() 와 같이 새롭게 프로세스를 만드는 함수들에서

allocproc() 함수를 호출하여 allocproc()함수가 UNUSED 상태에서 EMBRYO 상태로 만들어 반환한 프로세스를

RUNNABLE 상태로 만들어 scheduler() 함수가 fork() , userinit() 후 만들어진 프로세스를 스케쥴링 할 수 있도록 하기 때문입니다.

allocproc() 함수내 proc 구조체 변수 초기화를 위해 추가한 코드

우선순위에 경우 1과 2 IDLE 프로세스의 경우에는 초기 우선순위가 0이며,

그 외 프로세스에 경우에는

런 큐에 1,2 프로세스 외의 프로세스가 없다면 0으로 초기화 하며

런 큐에 1,2 프로세스 외의 프로세스가 있다면 프로세스들 중 최소 우선순위 값으로 초기화 합니다.

  if(p->pid == 1 ||p->pid == 2)

    p->priority = 99;

  else

  {

    int i,queue\_min = 0;

    int find = 0;

    for(i = 0; i < RUN\_QUEUE\_SIZE; i++)

    {

      if((queue\_min = get\_least\_priority\_value(&runqueues[i])) < 0)

      continue;

      else

      {

        p->priority = queue\_min;

        find = 1;

        // 현재 큐에서 우선순위 값을 얻었다면 다음

        // 다음 큐보단 값이 작을 것이 자명하기 때문에 break

       break;

      }

    }

    if(!find)

      p->priority = 0; // PID 1 , 2 외에 값을 찾지 못함 기본값인 0 설정

  }

  p->proc\_tick = 0;

  p->cpu\_used  = 0;

  p->is\_time\_set = 0;

  p->remaining\_ticks =0;

  p->priority\_tick = 0;

  p->next = NULL;

2-2-2. 런큐 관련 함수 프로토타입

2-2-2-1. proc.c 내 calculate\_new\_priority()

void calculate\_new\_priority()

해당 함수의 프로토타입은 defs.h내에 선언을 하고, trap.c 내에서 관리하는 tick 변수(프로세스들이 cpu를 차지한 시간) 이 60tick이 될때마다 모든 프로세스들의 우선순위를 재 계산하기 위해 만들었습니다.

2-2-2-2. node.c 내 push\_proc\_back()

void   push\_proc\_back(queue\* q, struct proc\* p);     // queue tail에 삽입

큐 포인터를 넘겨주어서 런큐에 struct proc\* p가 들어갈 인덱스의 큐를 첫번째 인자로 넣어줍니다.

ex ) push\_proc\_back(&runqueues[p->priority/4], p);

삽입하는 큐에 프로세스가 없다면 head 와 tail 모두 p를 가리키게 하며

큐에 프로세스가 있다면 tail에 있는 프로세스가 p를 가리키게 하며 tail을 옮겨 p를 가리키게 합니다.

결과적으로 해당하는 큐(링크드 리스트)에 tail에 p를 연결합니다.

2-2-2-3. node.c 내 get\_priority\_proc()

struct proc\*  get\_priority\_proc(queue\* q);           // queue 에서 가장 우선순위 높은 node 뽑기(우선순위 값 최소)

현재 인자로 주어진 큐에 우선순위 값이 최소(우선순위가 가장 높은) RUNNABLE 한 노드를 반환 받는 함수입니다.

만약, 큐에 프로세스 구조체가 없거나 RUNNABLE한 프로세스 구조체가 없다면 NULL을 리턴합니다.

ex ) scheduling() 함수에서 가장 우선순위 값이 낮은(우선순위가 가장 높은) 프로세스를 반환 받을 때 0~3 우선순위를 가진 큐(링크드 리스트를) 가장 먼저 순회합니다. 차례로 runqueues[0](0~3 우선순위를 갖는 프로세스 링크드리스트), runqueues[1](4~7 우선순위를 갖는 프로세스 링크드리스트)를 탐색하는데 만약 작은 우선순위 값을 갖는 큐에서 RUNNABLE한 프로세스를 찾는다면 그 뒤 더 우선순위 값이 큰 프로세스들을 갖는 큐를 탐색할 필요가 없습니다.

get\_prirority\_proc() 함수에서는 하나의 큐를 head ~ tail 까지 탐색하여서 가장 우선순위 값이 작은 RUNNABLE한 큐를 찾습니다.

2-2-2-4. node.c 내 pop\_proc()

struct proc\*  pop\_proc(queue\* q, struct proc\* p);    // queue 에서 프로세스 노드 제거

큐에서 해당하는 프로세스를 제거하는(링크드 리스트 연결을 끊는) 함수입니다. 프로세스의 우선순위가 변경되거나 프로세스가 ZOMBIE 상태가 되어 종료될 때 런큐의 해당 큐에서 프로세스를 제거해야합니다.

인자로 넘겨준 프로세스 구조체 p와 pid 값이 같은 프로세스를 찾을 경우 연결을 끊은 후에 포인터를 반환받습니다.

2-2-2-5. node.c 내 find\_proc()

int    find\_proc(queue\* q, struct proc\* p);   // queue 에서 해당하는 프로세스 노드 찾기

큐에서 해당하는 프로세스가 존재하는지 여부를 반환하는 함수입니다. 인자로 받은 큐를 head ~ tail 까지 탐색하며 pid 값이 같은 프로세스를 찾을 경우 1을 리턴 받으며 그 외 찾을 수 없는 경우 -1을 리턴 합니다.

2-2-2-6. node.c 내 print\_queue()

void   print\_queue(queue\* q);                  // 큐에 있는 모든 노드 출력

해당하는 큐에 링크드리스트로 연결된 프로세스들을 출력하기 위한 함수입니다. 큐에 head ~ tail을 순차 탐색하며 링크드 리스트로 연결된 프로세스들의 정보를 출력합니다.

2-2-2-7. node.c 내 init\_queue()

void   init\_queue(queue\* q);                  // queue 초기화

전역 변수로 선언되었던 queue runqueues[RUN\_QUEUE\_SIZE] 를 처음 쓰기 전 모든 배열의 원소 큐(링크드 리스트) 를 초기화 하기위해 만든 함수입니다. 해당 함수를 userinit() 에서 사용하며, scheduler() 함수가 호출 되기 전, 모든 큐(링크드 리스트)를 사용하려면 head / tail 링크드 리스트와 관련된 큐 포인터에 값을 NULL로 초기화 해주어야 합니다.

2-2-2-8. node.c 내 get\_least\_priority\_value()

int    get\_least\_priority\_value(queue\* q);    // 큐에 노드가 없을 경우 -1, 그 외 큐에서 가장 작은 우선순위 값 반환

해당 함수는 proc.c 내 allocproc() 함수 내에서 pid 1,2 이외에 프로세스가 존재할 경우 해당 프로세스들 중 가장 우선순위 값이 작은 값을 반환 받아 초기화 하기 위에 만들었습니다.

0~ 24 런큐 인덱스를 가진 런큐(큐 배열)의 큐를 하나씩 넣고 -1이 이외에 값을 찾으면 그 뒤로 다음 큐 탐색을 멈추고 해당 값으로 초기화 시켰습니다. 해당 함수내에서 pid == 1 과 pid ==2 값을 가진 프로세스의 우선순위를 반환하는 일은 없습니다.

2-2-3. TIME QUANTUM 30 수정 및 ticks 60마다 우선순위 재 계산

  // 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)

  // 타임 슬라이싱 (TIMEQUANTUM == 30)

  if(myproc()->proc\_tick == TIMEQUANTUM)

  {

    cprintf("PID : %d, priority : %d, proc\_tick : %d ticks, total\_cpu\_usage : %d ticks (1)\n",myproc()->pid,myproc()->priority,myproc()->proc\_tick,myproc()->cpu\_used);

    myproc()->proc\_tick = 0;

    yield();

  }

trap.c 내 trap() 함수 중 우선순위 재 계산 함수 호출

  // 60 tick 마다 priority 재 계산

  if((cpu\_used\_ticks % 60) == 0)

    calculate\_new\_priority();

2-2-4. Makefile CPU 코어 1개 수정

ifndef CPUS

CPUS := 1 # 설계과제 명세서에 의해 단일 CPU

endif

2-2-5. scheduler() 함수내 SSU\_SCHEDULER 구현

#elif SSU\_SCHEDULER

  struct proc\* p;

  int index = 0;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++, index++)

  {

      if(p->state != RUNNABLE)

        continue;

    ptable\_index = index;

    // RUNNABLE인 프로세스가 런큐에 없다면 삽입한다.

    if(find\_proc(&runqueues[p->priority/4],p) < 0)

      push\_proc\_back(&runqueues[p->priority/4], p);

  }

  int i;

  for(i = 0; i < RUN\_QUEUE\_SIZE; i++)

  {

    // runqueue 에서 가장 우선순위 값이 작은(우선순위가 높은) 프로세스 선택한다.

    if((p = get\_priority\_proc(&runqueues[i])) == NULL)

      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;

    switchuvm(p);

    p->state = RUNNING;

    cprintf("PID : %d, priority : %d, proc\_tick : %d ticks, total\_cpu\_used : %d ticks (2)\n",p->pid, p->priority, p->proc\_tick, p->cpu\_used);

    #ifdef DEBUG

      if(p)

        cprintf("PID : %d, NAME : SSU\_SCHEDULER.\n",p->pid);

    #endif

    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;

    break;

  }

#else

#endif

도표, 텍스트, 평면도, 라인이(가) 표시된 사진

자동 생성된 설명

2-2-6. 프로세스 종료시 pop\_proc()

exit() 함수내 ZOMBIE 상태로 프로세스 상태를 초기화 후, 런큐에서 해당 프로세스를 pop\_proc()합니다.

  // Jump into the scheduler, never to return.

  cprintf("PID : %d, priority : %d, proc\_tick : %d ticks, total\_cpu\_usage : %d ticks (3)\n",curproc->pid, curproc->priority,curproc->proc\_tick, curproc->cpu\_used);

  cprintf("PID : %d terminated\n",curproc->pid);

  pop\_proc(&runqueues[curproc->priority/4],curproc);

  curproc->state = ZOMBIE;

  sched();

  panic("zombie exit");

2-3. set\_sche\_info() 시스템 콜 추가 구현 (3번 요소)

설계 2 과제를 통한 시스템 콜 추가와 동일하게 구현하였습니다.

1.user.h : int set\_sche\_info(int , int)을 추가하였습니다. 유저 프로그램에서 시스템 콜을 호출하여서 사용할 수 있도록 하기 위함입니다.

2.usys.S : xv6의 시스템 호출 리스트에 추가로 SYSCALL(set\_sche\_info)를 추가하였습니다.

3.syscall.h : 시스템 호출 번호 매핑에 #define SYS\_set\_sche\_info 22을 추가하였습니다. 새로운 시스템 호출을 위해 22에 새로운 매핑을 추가한 것입니다.

4.syscall.c : 시스템 호출 인수를 구문 분석하는 함수 및 실제 시스템 호출 구현에 대한 포인터에 새로운 시스템 호출 함수 포인터를 추가하였습니다. [SYS\_set\_sche\_info] sys\_set\_sche\_info 와 extern int sys\_set\_sche\_info(void);를 추가하였습니다.

5.sysproc.c : 시스템 호출 코드를 추가하였습니다. int sys\_set\_sche\_info(void) 추가

6. proc.c : set\_sche\_info(int,int) 함수를 구현하였습니다. wait() 나 kill() 같은 함수들과 마찬가지로, 커널 모드로 모드 전환을 한 후 sysproc.c 내 시스템 콜 핸들러 함수가 내부 적으로 set\_sche\_info(int,int) 를 호출합니다.

텍스트, 도표, 스크린샷, 평행이(가) 표시된 사진

자동 생성된 설명

2-4. Makefile 내 -D DEBUG를 통한 스케쥴러 분석 추가 (4번 요소)

2-4-1. 스케쥴러 정책 결정

2-5 수행 전 기존 XV6 스케쥴러와 SSU\_SCHEDULER 비교를 위해

Makefile 내 해당 line 추가 및 CFLAGS에 -DDEBUG 와 같이 스케쥴러 정책을 미리 넣었습니다.

SCHED\_POLICY = SSU\_SCHEDULER # DEFAULT, SSU\_SCHEDULER 둘 중 하나의 정책을 골라 컴파일 후 실행하면 5번 성능 비교에 사용

CFLAGS = -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -D$(SCHED\_POLICY)

해당 Makefile 내용을 미리 SSU\_SCHEDULER로 결정해 놓은 이유는 만약 스케쥴러 정책을 make qemu 결정시에 하게 된다면 과제 채점 시, 명령어를 다르게 입력 해야 하므로 5번 성능 비교 시 수동으로 DEFAULT / SSU\_SCHEDULER 를 결정하여 성능 평가를 진행하였습니다. 과제 제출시에 낸 Makefile 상에 SCHED\_POLICY는 SSU\_SCHEDULER입니다.

2-4-2. DEFAULT (기존 XV6 스케쥴러) / SSU\_SCHEDULER

명세서에 언급되어 있는대로, debug =1 매개변수 전달을 통해

“스케쥴링 함수 에서 다음 실행될 프로세스를 선택할 떄마다” (2)번 출력 내용에 대해서 ifdef DEBUG로 감싸

debug=1 xv6 빌드 시, 이를 출력 할수 있도록 하였습니다. 해당 ifdef DEBUG는 scheduler() 함수에 포함하였습니다.

Makefile 내 추가

# debug 매개 변수를 통해 스케쥴링 함수에서 다음 실행될 프로세스를 선택할 떄마다 출력

# debug

ifeq ($(debug),1)

    CFLAGS += -DDEBUG

endif

proc.c scheduler() 함수 내

기존 스케쥴러 #ifdef DEFAULT 부분

#ifdef DEBUG

    if(p)

    {

      cprintf("PID : %d, priority : %d, proc\_tick : %d ticks, total\_cpu\_used : %d ticks (2)\n",p->pid, p->priority, p->proc\_tick, p->cpu\_used);

      cprintf("PID : %d, NAME : XV6 SCHEDULER.\n",p->pid);

    }

    #endif

SSU\_SCHEDULER #ifdef SSU\_SCHEDULER 부분

 #ifdef DEBUG

      if(p)

      {

        cprintf("PID : %d, priority : %d, proc\_tick : %d ticks, total\_cpu\_used : %d ticks (2)\n",p->pid, p->priority, p->proc\_tick, p->cpu\_used);

        cprintf("PID : %d, NAME : SSU\_SCHEDULER.\n",p->pid);

      }

    #endif

2-5. 기존 XV6 스케쥴러 와 SSU\_SCHEDULER 성능 비교 분석 (5번 요소)

기능 / 성능 의 비교

|  |  |  |
| --- | --- | --- |
|  | XV6 스케쥴러(Round-Robin) | SSU\_SCHEDULER |
| Startvation(기아) | X | O |
| 우선순위 고려 | X | O |
| 결정 모드  (선점 / 비선점) | 선점(preemption) 형 스케쥴러 | 선점(preemption) 형 스케쥴러 |
| Time Quantum  (Time Slicing) | 1 tick | 30 ticks |
| context  switching 횟수 | 많음  (Time quantum 영향이 가장 큼) | 적음  (Time quatum 영향이 가장 큼) |
| 평균 응답 시간  (Response time) | 좋음 | 좋음 |
| 평균 반환 시간  (Trun around time) | 나쁨 | 좋음 |
| 프로세스에게  미치는 영향 | 모든 프로세스에게 공정 | 긴 프로세스에게 불리 |

해당 표의 좋음 나쁨에 대한 비교는 비교 대상이 되는 XV6 스케쥴러 / SSU\_SCHEDULER에 대한 상대적인 값입니다.

( ex ) SSU\_SCHEDULER 1번 성능이 좋음 : 상대적으로 XV6 스케쥴러 보다 1번 성능이 좋다. )

XV6 스케쥴러에 경우 모든 프로세스에게 공정합니다. SSU\_SCHEDULER에 경우는 우선순위가 높은(우선 순위 값이 작은 프로세스) 가 우선적이므로 우선순위가 낮은(우선 순위 값이 큰 프로세스) 가 상대적으로 불리합니다. 이로 이냏

starvation이 발생 할 수 있습니다.

우선순위를 고려하여 스케쥴링을 하는 경우는 SSU\_SCHEDULER 만 해당합니다.

결정 모드(선점 / 비선점) 두 스케쥴러 XV6 스케쥴러와 SSU\_SCHEDULER 모두 선점형(preemption , 뺏을 수 있는, 빼앗 길 수 있는) 스케쥴러에 해당합니다.

Time Quantum에 경우에는 스케쥴러 정책과는 별개로 두 스케쥴러 모두 TQ를 30ticks / 1tick 적용할 수 있으나, 본 과제에서 기존 XV6 스케쥴러에 경우에는 타임 퀀텀이 1tick 이었고, 교체한 SSU\_SCHEDULER에 경우에는 30ticks에 해당합니다.

context switching 횟수 입니다. 두 스케쥴러 모두 선점형 스케쥴러 이므로, 문맥 교환 횟수가 많습니다. 하지만, 타임 퀀텀에 영향을 받아 타임퀀텀이 1tick 인 XV6 스케쥴러가 타임 퀀텀이 30tick인 SSU\_SCHEDULER 보다 많았습니다.

Round-Robin 스케쥴러인 XV6 스케쥴러는 응답 시간에 대해 설명하지 않겠습니다. 모든 프로세스에게 공정하게 타임 퀀텀 만큼씩을 번갈아 가며 수행하기 때문입니다.

SSU\_SCHEDULER는 pid 1,2 를 제외한 런큐에서 관리하는 우선순위 값 중 최소를 부여받습니다. 이를 통해 현재 실행 중인 우선순위 값이 가장 작은 프로세스 다음으로 스케쥴러에게 선택을 받을 수 밖에 없었습니다. scheduler\_test.c 유저 프로그램을 실행할 때에도 fork()를 통해 자식 프로세스를 초기값으로 만든 후, set\_sche\_info() 시스템 콜을 통해 우선순위와 수행 시간을 부여 받기 전에 현재 부모 프로세스가 수행 된 이후 바로 다음 수행을 보장 받습니다.(우선순위 값이 런큐에서 관리하는 모든 프로세스 중 최소 이기 때문입니다.) 두 스케쥴러 모두 응답시간은 매우 짧아 좋음 이었습니다.

그럼에도 예를들어 3tick 동안 XV6 스케쥴러는 3개의 프로세스를 모두 수행하여 응답이 빠르고, SSU\_SCHEDULER는 우선순위가 상대적으로 높은 프로세스들이 우선적 수행이므로 우선순위가 낮은 프로세스들은 이후 응답이 가능하므로 응답시간이 SSU\_SCHEDULER가 XV6 스케쥴러보다는 나쁘다고 측정하였습니다.

반환 시간은 XV6 스케쥴러는 나쁨으로 SSU\_SCHEDULER는 좋음으로 표기하였습니다.

Round-Robin에 경우 작업을 처음 시작(응답 시간)이 짧은 반면 스케쥴러가 관리하는 pTable에 모든 프로세스가 공정하게 타임 퀀텀(TQ) 만큼씩을 수행하므로 하나의 프로세스가 끝나는 시간은 길어질수 밖에 없습니다. 모든 프로세스들이 모두 안좋은 반환 시간을 갖게 됩니다.

그에 반해, SSU\_SCHEDULER는 우선순위를 고려합니다. 우선순위가 높은 프로세스들을 우선적으로 끝내기 때문에 상대적으로 짧은 프로세스들이 초기 우선순위를 부여받고, 빠르게 작업을 완료하여 반환 시간 또한 좋습니다. 하지만, 상대적으로 긴 프로세스들은 점점 우선순위가 낮아지고, 늦게 작업을 완료합니다. 그럼에도, Round-Robin 과 비교하였을때는 좋은 반환시간을 가지게됩니다.

마지막으로, 프로세스에게 미치는 영향입니다. starvation을 갖는 SSU\_SCHEDULER는 긴 프로세스에게 불리합니다. 이는 60tick(프로세스가 차지하는 cpu 시간) 마다 우선순위를 재 계산하는 로직으로 인합니다. 긴 프로세스에 경우 우선순위를 재 계산 당할 일이 더 많이 생기고, 우선순위가 점점 밀려 새로 들어오는 프로세스들에 비해 상대적으로 낮은 우선순위를 가질 수 밖에 없기 때문에 긴 프로세스들이 불리합니다. 기존 XV6 스케쥴러에 경우에는 그러한 우선순위를 고려하지 않기 때문에 모든 프로세스에게 공정합니다.

작업 도착시간은 부모 프로세스가 fork() 후 set\_sche\_info()를 통한 프로세스의 우선순위값 설정 이후입니다.

2-5-1. set\_sche\_info(1,110), set\_sche\_info(10,60), set\_sche\_info(11,60)

SSU\_SCHEDULER 평균 응답 시간 (0 tick)

SSU\_SCHEDULER 평균 반환 시간 (170tick)

작업 완료 순 1,2,3

전체 작업 완료 시간 230 ticks

|  |  |  |  |
| --- | --- | --- | --- |
|  | #1 (1,110) | #2 (10,60) | #3 (11,60) |
| 도착 시간 | 0 | 30 | 60 |
| 응답 시간 | 0 | 0 | 0 |
| 반환 시간 | 170 | 170 | 170 |

XV6 스케쥴러 평균 응답 시간 (0 tick)

XV6 스케쥴러 평균 반환 시간 (196.3333 ticks)

작업 완료 순 2,3,1

전체 작업 완료 시간 230ticks

|  |  |  |  |
| --- | --- | --- | --- |
|  | #1 (1,110) | #2 (10,60) | #3 (11,60) |
| 도착 시간 | 0 | 1 | 2 |
| 응답 시간 | 0 | 0 | 0 |
| 반환 시간 | 230 | 179 | 178 |

2-5-2. set\_sche\_info(1,110), set\_sche\_info(22,200), set\_sche\_info(11,250)

SSU\_SCHEDULER 평균 응답 시간 (0 tick)

SSU\_SCHEDULER 평균 반환 시간 ( 383.333333 tick)

전체 작업 완료 시간 560 ticks

작업 완료 순 1,3,2

|  |  |  |  |
| --- | --- | --- | --- |
|  | #1 (1,110) | #2 (22,200) | #3 (11,250) |
| 도착 시간 | 0 | 30 | 60 |
| 응답 시간 | 0 | 0 | 0 |
| 반환 시간 | 170 | 530 | 450 |

XV6 스케쥴러 평균 응답 시간 (0 tick)

XV6 스케쥴러 평균 반환 시간 ( 465. 66667 ticks)

전체 작업 완료 시간 560 ticks

작업 완료 순 1,2,3

|  |  |  |  |
| --- | --- | --- | --- |
|  | #1 (1,110) | #2 (22,200) | #3 (11,250) |
| 도착 시간 | 0 | 1 | 2 |
| 응답 시간 | 0 | 0 | 0 |
| 반환 시간 | 330 | 509 | 558 |

2-5-3. 2-5-1 과 2-5-2 지표를 통한 그래프

일단 XV6 스케쥴러의 작업 완료 순은 작업 시간이 짧은 순이었습니다. SSU\_SCHEDULER의 작업 완료 순은 작업 우선순위가 높은 순이었습니다. 작업 우선순위로 인해 후순위로 밀려난 작업들의 반환시간이 높아 SSU\_SCHEDULER의 반환시간이 더 나쁠 것으로 예상했으나, Round-Robin으로 기존 XV6 스케쥴러에 경우에는 모든 작업이 전반적으로 나쁜 반환시간을 가져 더 평균 반환 시간이 안 좋았습니다.

전체 작업들의 작업 시간이 점점 커질수록, Round-Robin(기존 XV6 스케쥴러)의 반환속도는 점점 더 나빠졌습니다.

다른 작업들과 함께 번갈아 작업을 해야 하기 때문입니다.

두 스케쥴러의 비교를 명확히 보여주는 반환시간에 대한 그래프입니다.

x 축은 전체 프로세스들의 작업 시간의 총 합 입니다.

y 축은 평균 Turn around time(반환 시간) 입니다.

y=0.0013455x^{2}+0.19572727x // 기존 XV6 스케쥴러 빨간 그래프

y=0.00023384x^{2}+0.3891947x // SSU\_SCHEDULER 파랑 그래프

평균 반환 시간은 기존 XV6 스케쥴러 보다 SSU\_SCHEDULER가 더 좋습니다.

라인, 그래프, 도표, 평행이(가) 표시된 사진

자동 생성된 설명

3. 실행 결과

3-1. set\_sche\_info(1,110), set\_sche\_info(22,200), set\_sche\_info(11,250) (debug=1)

텍스트, 스크린샷, 메뉴, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 메뉴, 폰트이(가) 표시된 사진

자동 생성된 설명

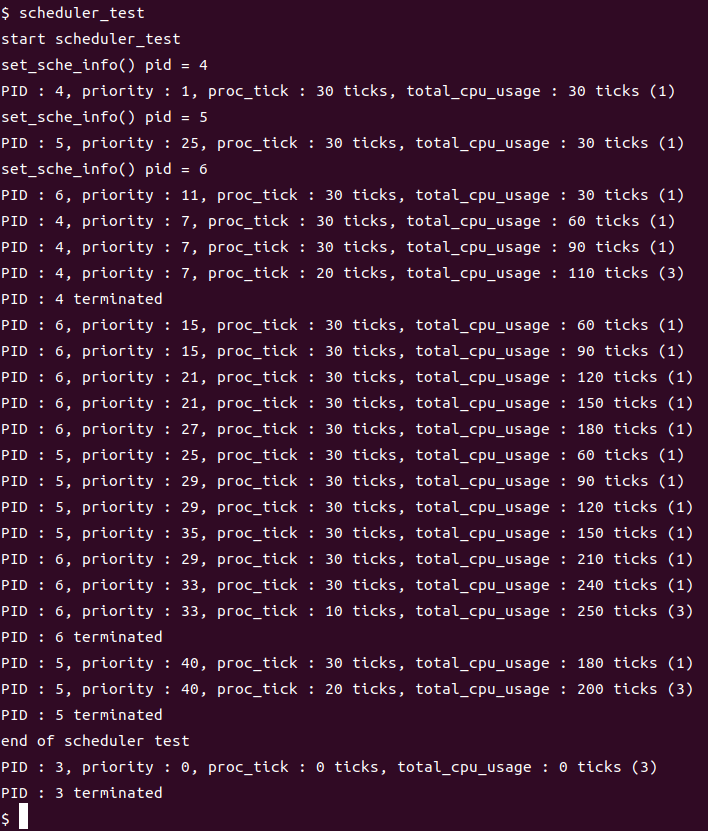
텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

3-2. set\_sche\_info(1,110), set\_sche\_info(22,200), set\_sche\_info(11,250)



3-3. set\_sche\_info(1,110), set\_sche\_info(10,60), set\_sche\_info(11,60) (debug=1)

텍스트, 스크린샷, 메뉴이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 메뉴이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

3-4. set\_sche\_info(1,110), set\_sche\_info(10,60), set\_sche\_info(11,60)

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

3-5 ~ 3-6. 5번 요소 XV6 스케쥴러와 SSU\_SCHEDULER 비교를 위한 출력(추가)

기존 XV6 스케쥴러의 타임퀀텀은 1이기 때문에 출력요소가 너무 많아 중간 중간 생략을 하고,

작업이 처음 응답하였을때와 작업을 마쳤을때를 캡쳐하였습니다. 3-5 와 3-6은 모두 debug=1 디버그 옵션을 줬습니다.

사실 기존 XV6 스케쥴러에 경우에는 우선순위를 재계산하면 안되나, 프로세스가 차지하는 60틱마다 재계산이 이루어지는 코드는 그대로 둔 채, 스케쥴러만 바꾸어 테스트하여 우선순위가 바뀌어 출력됩니다.

하지만, 기존 XV6 스케쥴러가 Round-Robin 형태로 우선순위를 고려하지 않는 것은 유지되어 우선순위가 바뀌더라도 영향이 없습니다.

3-5. XV6 스케쥴러 set\_sche\_info(1,110), set\_sche\_info(22,200),set\_sche\_info(11,250)

3번 부모 프로세스가 4,5,6 자식 프로세스를 만든 후, 4,5,6 프로세스가 공평하게 1틱씩을 수행하고 타이머 인터럽트를 통해 다음 프로세스를 수행하는 모습

작업 완료 순서는 pid 4,5,6 순이므로, 뒤에 4번이 종료 되기 직전에는 4,5,6 프로세스가 5번이 종료 되기 직전에는 5,6 프로세스가 번갈아 수행 6번이 종료 되기 직전에는 6프로세스만 수행 되는 모습을 볼수 있습니다.

텍스트, 스크린샷, 메뉴이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

3-6. XV6 스케쥴러 set\_sche\_info(1,110), set\_sche\_info(10,60), set\_sche\_info(11,60)

텍스트, 스크린샷, 메뉴이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

4. 소스 코드

4-1. defs.h

struct buf;

struct context;

struct file;

struct inode;

struct pipe;

struct proc;

struct rtcdate;

struct spinlock;

struct sleeplock;

struct stat;

struct superblock;

// bio.c

void            binit(void);

struct buf\*     bread(uint, uint);

void            brelse(struct buf\*);

void            bwrite(struct buf\*);

// console.c

void            consoleinit(void);

void            cprintf(char\*, ...);

void            consoleintr(int(\*)(void));

void            panic(char\*) \_\_attribute\_\_((noreturn));

// exec.c

int             exec(char\*, char\*\*);

// file.c

struct file\*    filealloc(void);

void            fileclose(struct file\*);

struct file\*    filedup(struct file\*);

void            fileinit(void);

int             fileread(struct file\*, char\*, int n);

int             filestat(struct file\*, struct stat\*);

int             filewrite(struct file\*, char\*, int n);

// fs.c

void            readsb(int dev, struct superblock \*sb);

int             dirlink(struct inode\*, char\*, uint);

struct inode\*   dirlookup(struct inode\*, char\*, uint\*);

struct inode\*   ialloc(uint, short);

struct inode\*   idup(struct inode\*);

void            iinit(int dev);

void            ilock(struct inode\*);

void            iput(struct inode\*);

void            iunlock(struct inode\*);

void            iunlockput(struct inode\*);

void            iupdate(struct inode\*);

int             namecmp(const char\*, const char\*);

struct inode\*   namei(char\*);

struct inode\*   nameiparent(char\*, char\*);

int             readi(struct inode\*, char\*, uint, uint);

void            stati(struct inode\*, struct stat\*);

int             writei(struct inode\*, char\*, uint, uint);

// ide.c

void            ideinit(void);

void            ideintr(void);

void            iderw(struct buf\*);

// ioapic.c

void            ioapicenable(int irq, int cpu);

extern uchar    ioapicid;

void            ioapicinit(void);

// kalloc.c

char\*           kalloc(void);

void            kfree(char\*);

void            kinit1(void\*, void\*);

void            kinit2(void\*, void\*);

// kbd.c

void            kbdintr(void);

// lapic.c

void            cmostime(struct rtcdate \*r);

int             lapicid(void);

extern volatile uint\*    lapic;

void            lapiceoi(void);

void            lapicinit(void);

void            lapicstartap(uchar, uint);

void            microdelay(int);

// log.c

void            initlog(int dev);

void            log\_write(struct buf\*);

void            begin\_op();

void            end\_op();

// mp.c

extern int      ismp;

void            mpinit(void);

// picirq.c

void            picenable(int);

void            picinit(void);

// pipe.c

int             pipealloc(struct file\*\*, struct file\*\*);

void            pipeclose(struct pipe\*, int);

int             piperead(struct pipe\*, char\*, int);

int             pipewrite(struct pipe\*, char\*, int);

//PAGEBREAK: 16

// proc.c

int             cpuid(void);

void            exit(void);

int             fork(void);

int             growproc(int);

int             kill(int);

struct cpu\*     mycpu(void);

struct proc\*    myproc();

void            pinit(void);

void            procdump(void);

void            scheduler(void) \_\_attribute\_\_((noreturn));

void            sched(void);

void            setproc(struct proc\*);

void            sleep(void\*, struct spinlock\*);

void            userinit(void);

int             wait(void);

void            wakeup(void\*);

void            yield(void);

int             set\_sche\_info(int, int);

void            calculate\_new\_priority();

// swtch.S

void            swtch(struct context\*\*, struct context\*);

// spinlock.c

void            acquire(struct spinlock\*);

void            getcallerpcs(void\*, uint\*);

int             holding(struct spinlock\*);

void            initlock(struct spinlock\*, char\*);

void            release(struct spinlock\*);

void            pushcli(void);

void            popcli(void);

// sleeplock.c

void            acquiresleep(struct sleeplock\*);

void            releasesleep(struct sleeplock\*);

int             holdingsleep(struct sleeplock\*);

void            initsleeplock(struct sleeplock\*, char\*);

// string.c

int             memcmp(const void\*, const void\*, uint);

void\*           memmove(void\*, const void\*, uint);

void\*           memset(void\*, int, uint);

char\*           safestrcpy(char\*, const char\*, int);

int             strlen(const char\*);

int             strncmp(const char\*, const char\*, uint);

char\*           strncpy(char\*, const char\*, int);

// syscall.c

int             argint(int, int\*);

int             argptr(int, char\*\*, int);

int             argstr(int, char\*\*);

int             fetchint(uint, int\*);

int             fetchstr(uint, char\*\*);

void            syscall(void);

// timer.c

void            timerinit(void);

// trap.c

void            idtinit(void);

extern uint     ticks;

void            tvinit(void);

extern struct spinlock tickslock;

// uart.c

void            uartinit(void);

void            uartintr(void);

void            uartputc(int);

// vm.c

void            seginit(void);

void            kvmalloc(void);

pde\_t\*          setupkvm(void);

char\*           uva2ka(pde\_t\*, char\*);

int             allocuvm(pde\_t\*, uint, uint);

int             deallocuvm(pde\_t\*, uint, uint);

void            freevm(pde\_t\*);

void            inituvm(pde\_t\*, char\*, uint);

int             loaduvm(pde\_t\*, char\*, struct inode\*, uint, uint);

pde\_t\*          copyuvm(pde\_t\*, uint);

void            switchuvm(struct proc\*);

void            switchkvm(void);

int             copyout(pde\_t\*, uint, void\*, uint);

void            clearpteu(pde\_t \*pgdir, char \*uva);

// number of elements in fixed-size array

#define NELEM(x) (sizeof(x)/sizeof((x)[0]))

4-2. Makefile

SCHED\_POLICY = SSU\_SCHEDULER # DEFAULT, SSU\_SCHEDULER 둘 중 하나의 정책을 골라 컴파일 후 실행하면 5번 성능 비교에 사용

OBJS = \

    bio.o\

    console.o\

    exec.o\

    file.o\

    fs.o\

    ide.o\

    ioapic.o\

    kalloc.o\

    kbd.o\

    lapic.o\

    log.o\

    main.o\

    mp.o\

    picirq.o\

    pipe.o\

    proc.o\

    sleeplock.o\

    spinlock.o\

    string.o\

    swtch.o\

    syscall.o\

    sysfile.o\

    sysproc.o\

    trapasm.o\

    trap.o\

    uart.o\

    vectors.o\

    vm.o\

    node.o\

# Cross-compiling (e.g., on Mac OS X)

# TOOLPREFIX = i386-jos-elf

# Using native tools (e.g., on X86 Linux)

#TOOLPREFIX =

# Try to infer the correct TOOLPREFIX if not set

ifndef TOOLPREFIX

TOOLPREFIX := $(shell if i386-jos-elf-objdump -i 2>&1 | grep '^elf32-i386$$' >/dev/null 2>&1; \

    then echo 'i386-jos-elf-'; \

    elif objdump -i 2>&1 | grep 'elf32-i386' >/dev/null 2>&1; \

    then echo ''; \

    else echo "\*\*\*" 1>&2; \

    echo "\*\*\* Error: Couldn't find an i386-\*-elf version of GCC/binutils." 1>&2; \

    echo "\*\*\* Is the directory with i386-jos-elf-gcc in your PATH?" 1>&2; \

    echo "\*\*\* If your i386-\*-elf toolchain is installed with a command" 1>&2; \

    echo "\*\*\* prefix other than 'i386-jos-elf-', set your TOOLPREFIX" 1>&2; \

    echo "\*\*\* environment variable to that prefix and run 'make' again." 1>&2; \

    echo "\*\*\* To turn off this error, run 'gmake TOOLPREFIX= ...'." 1>&2; \

    echo "\*\*\*" 1>&2; exit 1; fi)

endif

# If the makefile can't find QEMU, specify its path here

# QEMU = qemu-system-i386

# Try to infer the correct QEMU

ifndef QEMU

QEMU = $(shell if which qemu > /dev/null; \

    then echo qemu; exit; \

    elif which qemu-system-i386 > /dev/null; \

    then echo qemu-system-i386; exit; \

    elif which qemu-system-x86\_64 > /dev/null; \

    then echo qemu-system-x86\_64; exit; \

    else \

    qemu=/Applications/Q.app/Contents/MacOS/i386-softmmu.app/Contents/MacOS/i386-softmmu; \

    if test -x $$qemu; then echo $$qemu; exit; fi; fi; \

    echo "\*\*\*" 1>&2; \

    echo "\*\*\* Error: Couldn't find a working QEMU executable." 1>&2; \

    echo "\*\*\* Is the directory containing the qemu binary in your PATH" 1>&2; \

    echo "\*\*\* or have you tried setting the QEMU variable in Makefile?" 1>&2; \

    echo "\*\*\*" 1>&2; exit 1)

endif

CC = $(TOOLPREFIX)gcc

AS = $(TOOLPREFIX)gas

LD = $(TOOLPREFIX)ld

OBJCOPY = $(TOOLPREFIX)objcopy

OBJDUMP = $(TOOLPREFIX)objdump

CFLAGS = -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -D$(SCHED\_POLICY)

CFLAGS += $(shell $(CC) -fno-stack-protector -E -x c /dev/null >/dev/null 2>&1 && echo -fno-stack-protector)

ASFLAGS = -m32 -gdwarf-2 -Wa,-divide

# FreeBSD ld wants ``elf\_i386\_fbsd''

LDFLAGS += -m $(shell $(LD) -V | grep elf\_i386 2>/dev/null | head -n 1)

# Disable PIE when possible (for Ubuntu 16.10 toolchain)

ifneq ($(shell $(CC) -dumpspecs 2>/dev/null | grep -e '[^f]no-pie'),)

CFLAGS += -fno-pie -no-pie

endif

ifneq ($(shell $(CC) -dumpspecs 2>/dev/null | grep -e '[^f]nopie'),)

CFLAGS += -fno-pie -nopie

endif

# debug 매개 변수를 통해 스케쥴링 함수에서 다음 실행될 프로세스를 선택할 떄마다 출력

# debug

ifeq ($(debug),1)

    CFLAGS += -DDEBUG

endif

xv6.img: bootblock kernel

    dd if=/dev/zero of=xv6.img count=10000

    dd if=bootblock of=xv6.img conv=notrunc

    dd if=kernel of=xv6.img seek=1 conv=notrunc

xv6memfs.img: bootblock kernelmemfs

    dd if=/dev/zero of=xv6memfs.img count=10000

    dd if=bootblock of=xv6memfs.img conv=notrunc

    dd if=kernelmemfs of=xv6memfs.img seek=1 conv=notrunc

bootblock: bootasm.S bootmain.c

    $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c

    $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S

    $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o

    $(OBJDUMP) -S bootblock.o > bootblock.asm

    $(OBJCOPY) -S -O binary -j .text bootblock.o bootblock

    ./sign.pl bootblock

entryother: entryother.S

    $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c entryother.S

    $(LD) $(LDFLAGS) -N -e start -Ttext 0x7000 -o bootblockother.o entryother.o

    $(OBJCOPY) -S -O binary -j .text bootblockother.o entryother

    $(OBJDUMP) -S bootblockother.o > entryother.asm

initcode: initcode.S

    $(CC) $(CFLAGS) -nostdinc -I. -c initcode.S

    $(LD) $(LDFLAGS) -N -e start -Ttext 0 -o initcode.out initcode.o

    $(OBJCOPY) -S -O binary initcode.out initcode

    $(OBJDUMP) -S initcode.o > initcode.asm

kernel: $(OBJS) entry.o entryother initcode kernel.ld

    $(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother

    $(OBJDUMP) -S kernel > kernel.asm

    $(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .\* / /; /^$$/d' > kernel.sym

# kernelmemfs is a copy of kernel that maintains the

# disk image in memory instead of writing to a disk.

# This is not so useful for testing persistent storage or

# exploring disk buffering implementations, but it is

# great for testing the kernel on real hardware without

# needing a scratch disk.

MEMFSOBJS = $(filter-out ide.o,$(OBJS)) memide.o

kernelmemfs: $(MEMFSOBJS) entry.o entryother initcode kernel.ld fs.img

    $(LD) $(LDFLAGS) -T kernel.ld -o kernelmemfs entry.o  $(MEMFSOBJS) -b binary initcode entryother fs.img

    $(OBJDUMP) -S kernelmemfs > kernelmemfs.asm

    $(OBJDUMP) -t kernelmemfs | sed '1,/SYMBOL TABLE/d; s/ .\* / /; /^$$/d' > kernelmemfs.sym

tags: $(OBJS) entryother.S \_init

    etags \*.S \*.c

vectors.S: vectors.pl

    ./vectors.pl > vectors.S

ULIB = ulib.o usys.o printf.o umalloc.o

\_%: %.o $(ULIB)

    $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^

    $(OBJDUMP) -S $@ > $\*.asm

    $(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .\* / /; /^$$/d' > $\*.sym

\_forktest: forktest.o $(ULIB)

    # forktest has less library code linked in - needs to be small

    # in order to be able to max out the proc table.

    $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o \_forktest forktest.o ulib.o usys.o

    $(OBJDUMP) -S \_forktest > forktest.asm

mkfs: mkfs.c fs.h

    gcc -Werror -Wall -o mkfs mkfs.c

# Prevent deletion of intermediate files, e.g. cat.o, after first build, so

# that disk image changes after first build are persistent until clean.  More

# details:

# http://www.gnu.org/software/make/manual/html\_node/Chained-Rules.html

.PRECIOUS: %.o

UPROGS=\

    \_cat\

    \_echo\

    \_forktest\

    \_grep\

    \_init\

    \_kill\

    \_ln\

    \_ls\

    \_mkdir\

    \_rm\

    \_sh\

    \_stressfs\

    \_usertests\

    \_wc\

    \_zombie\

    \_scheduler\_test\

fs.img: mkfs README $(UPROGS)

    ./mkfs fs.img README $(UPROGS)

-include \*.d

clean:

    rm -f \*.tex \*.dvi \*.idx \*.aux \*.log \*.ind \*.ilg \

    \*.o \*.d \*.asm \*.sym vectors.S bootblock entryother \

    initcode initcode.out kernel xv6.img fs.img kernelmemfs \

    xv6memfs.img mkfs .gdbinit \

    $(UPROGS)

# make a printout

FILES = $(shell grep -v '^\#' runoff.list)

PRINT = runoff.list runoff.spec README toc.hdr toc.ftr $(FILES)

xv6.pdf: $(PRINT)

    ./runoff

    ls -l xv6.pdf

print: xv6.pdf

# run in emulators

bochs : fs.img xv6.img

    if [ ! -e .bochsrc ]; then ln -s dot-bochsrc .bochsrc; fi

    bochs -q

# try to generate a unique GDB port

GDBPORT = $(shell expr `id -u` % 5000 + 25000)

# QEMU's gdb stub command line changed in 0.11

QEMUGDB = $(shell if $(QEMU) -help | grep -q '^-gdb'; \

    then echo "-gdb tcp::$(GDBPORT)"; \

    else echo "-s -p $(GDBPORT)"; fi)

ifndef CPUS

CPUS := 1 # 설계과제 명세서에 의해 단일 CPU

endif

QEMUOPTS = -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp $(CPUS) -m 512 $(QEMUEXTRA) -D $(SCHED\_POLICY)

qemu: fs.img xv6.img

    $(QEMU) -serial mon:stdio $(QEMUOPTS)

qemu-memfs: xv6memfs.img

    $(QEMU) -drive file=xv6memfs.img,index=0,media=disk,format=raw -smp $(CPUS) -m 256

qemu-nox: fs.img xv6.img

    $(QEMU) -nographic $(QEMUOPTS)

.gdbinit: .gdbinit.tmpl

    sed "s/localhost:1234/localhost:$(GDBPORT)/" < $^ > $@

qemu-gdb: fs.img xv6.img .gdbinit

    @echo "\*\*\* Now run 'gdb'." 1>&2

    $(QEMU) -serial mon:stdio $(QEMUOPTS) -S $(QEMUGDB)

qemu-nox-gdb: fs.img xv6.img .gdbinit

    @echo "\*\*\* Now run 'gdb'." 1>&2

    $(QEMU) -nographic $(QEMUOPTS) -S $(QEMUGDB)

# CUT HERE

# prepare dist for students

# after running make dist, probably want to

# rename it to rev0 or rev1 or so on and then

# check in that version.

EXTRA=\

    mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\

    ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\

    printf.c umalloc.c scheduler\_test.c\

    README dot-bochsrc \*.pl toc.\* runoff runoff1 runoff.list\

    .gdbinit.tmpl gdbutil\

dist:

    rm -rf dist

    mkdir dist

    for i in $(FILES); \

    do \

        grep -v PAGEBREAK $$i >dist/$$i; \

    done

    sed '/CUT HERE/,$$d' Makefile >dist/Makefile

    echo >dist/runoff.spec

    cp $(EXTRA) dist

dist-test:

    rm -rf dist

    make dist

    rm -rf dist-test

    mkdir dist-test

    cp dist/\* dist-test

    cd dist-test; $(MAKE) print

    cd dist-test; $(MAKE) bochs || true

    cd dist-test; $(MAKE) qemu

# update this rule (change rev#) when it is time to

# make a new revision.

tar:

    rm -rf /tmp/xv6

    mkdir -p /tmp/xv6

    cp dist/\* dist/.gdbinit.tmpl /tmp/xv6

    (cd /tmp; tar cf - xv6) | gzip >xv6-rev10.tar.gz  # the next one will be 10 (9/17)

.PHONY: dist-test dist

4-3. node.c

#include "types.h"

#include "defs.h"

#include "param.h"

#include "memlayout.h"

#include "mmu.h"

#include "x86.h"

#include "proc.h"

#include "spinlock.h"

#include "node.h"

void push\_proc\_back(queue\* q, struct proc\* p)

{

    if(q->head == NULL)

    {

        q->head = p;

        q->tail = p;

    }

    else

    {

        q->tail->next = p;

        q->tail = p;

    }

}

struct proc\* get\_priority\_proc(queue\* q)

{

    if(q->head == NULL)

        return NULL;

    struct proc\* now\_proc = q->head;

    struct proc\* most\_priority = NULL;

    // O(큐에 있는 노드 수)

    while(now\_proc != NULL)

    {

        if(now\_proc->state != RUNNABLE)

        {

            now\_proc = now\_proc->next;

            continue;

        }

        // 우선순위는 값이 낮을 수록 우선순위가 높다.

        if(most\_priority == NULL || now\_proc->priority < most\_priority->priority)

            most\_priority = now\_proc;

        now\_proc = now\_proc->next;

    }

    return most\_priority;

}

struct proc\* pop\_proc(queue\* q, struct proc\* p)

{

    struct proc\* now\_proc = q->head;

    struct proc\* prev\_proc = NULL;

    while(now\_proc != NULL)

    {

        // 해당하는 프로세스 찾았다.

        if(now\_proc->pid == p->pid)

        {

            // 첫번째 프로세스 였다

            if(prev\_proc == NULL)

            {

                q->head = now\_proc->next;

                // 마지막 프로세스 였다.

                if(now\_proc->next == NULL)

                   q->tail = NULL;

            }

            else

            {

                prev\_proc->next = now\_proc->next;

                // 마지막 프로세스 였다.

                if(now\_proc->next == NULL)

                    q->tail = prev\_proc;

            }

            now\_proc->next = NULL;

            return now\_proc;

        }

        prev\_proc = now\_proc;

        now\_proc = now\_proc->next;

    }

    return NULL;

}

int find\_proc(queue\* q, struct proc\* p)

{

   struct proc\* now\_proc = q->head;

    while(now\_proc != NULL)

    {

        if(now\_proc->pid == p->pid)

            return 1;

        now\_proc = now\_proc->next;

    }

    return -1;

}

void print\_queue(queue\* q)

{

    int count = 1;

    struct proc\* now\_proc = q->head;

    while(now\_proc != NULL)

    {

        cprintf("NODE# : %d, PID : %d, priority : %d\n",count,now\_proc->pid, now\_proc->priority);

        count++;

        now\_proc = now\_proc->next;

    }

}

void init\_queue(queue\* q)

{

    q->head = NULL;

    q->tail = NULL;

}

int get\_least\_priority\_value(queue\* q)

{

    const int INF = 987654321;

    int least\_priority\_value = INF;

    // 큐에 노드가 없다.

    if(q->head == NULL)

        return -1;

    struct proc\* now\_proc = q->head;

    while(now\_proc != NULL)

    {

        if(now\_proc->pid == 1 || now\_proc->pid == 2)

        {

            now\_proc = now\_proc->next;

            continue; // PID 1 ,2 무시

        }

        if(now\_proc->priority < least\_priority\_value)

            least\_priority\_value = now\_proc->priority;

        now\_proc = now\_proc->next;

    }

    if(least\_priority\_value == INF)

        return -1;

    else

        return least\_priority\_value;

}

4-4. node.h

#ifndef NODE

#define NODE

    #define RUN\_QUEUE\_SIZE 25

    #define NULL            0

// run queue 구현을 위한 큐 구조체 해당 큐를 통해 프로세스의 링크드 리스트를 구현합니다.

// 각 큐는 프로세스를 가르킬 head와 tail이 있습니다.

typedef struct queue{

    struct proc\* head;

    struct proc\* tail;

}queue;

void   push\_proc\_back(queue\* q, struct proc\* p);     // queue tail에 삽입

struct proc\*  get\_priority\_proc(queue\* q);           // queue 에서 가장 우선순위 높은 node 뽑기(우선순위 값 최소)

struct proc\*  pop\_proc(queue\* q, struct proc\* p);    // queue 에서 프로세스 노드 제거

int    find\_proc(queue\* q, struct proc\* p);   // queue 에서 해당하는 프로세스 노드 찾기

void   print\_queue(queue\* q);                  // 큐에 있는 모든 노드 출력

void   init\_queue(queue\* q);                  // queue 초기화

int    get\_least\_priority\_value(queue\* q);    // 큐에 노드가 없을 경우 -1, 그 외 큐에서 가장 작은 우선순위 값 반환

#endif

4-5. proc.c

#include "types.h"

#include "defs.h"

#include "param.h"

#include "memlayout.h"

#include "mmu.h"

#include "x86.h"

#include "proc.h"

#include "spinlock.h"

#include "node.h"

struct {

  struct spinlock lock;

  struct proc proc[NPROC];

} ptable;

static struct proc \*initproc;

int ptable\_index; // ptable에 해당하는 인덱스의 노드가 RUNNABLE이라면 런큐에 넣는다.

queue runqueues[RUN\_QUEUE\_SIZE]; // 링크드리스트를 관리하는 큐의 배열

int nextpid = 1;

extern void forkret(void);

extern void trapret(void);

static void wakeup1(void \*chan);

void

pinit(void)

{

  initlock(&ptable.lock, "ptable");

}

// Must be called with interrupts disabled

int

cpuid() {

  return mycpu()-cpus;

}

// Must be called with interrupts disabled to avoid the caller being

// rescheduled between reading lapicid and running through the loop.

struct cpu\*

mycpu(void)

{

  int apicid, i;

  if(readeflags()&FL\_IF)

    panic("mycpu called with interrupts enabled\n");

  apicid = lapicid();

  // APIC IDs are not guaranteed to be contiguous. Maybe we should have

  // a reverse map, or reserve a register to store &cpus[i].

  for (i = 0; i < ncpu; ++i) {

    if (cpus[i].apicid == apicid)

      return &cpus[i];

  }

  panic("unknown apicid\n");

}

// Disable interrupts so that we are not rescheduled

// while reading proc from the cpu structure

struct proc\*

myproc(void) {

  struct cpu \*c;

  struct proc \*p;

  pushcli();

  c = mycpu();

  p = c->proc;

  popcli();

  return p;

}

//PAGEBREAK: 32

// Look in the process table for an UNUSED proc.

// If found, change state to EMBRYO and initialize

// state required to run in the kernel.

// Otherwise return 0.

static struct proc\*

allocproc(void)

{

  struct proc \*p;

  char \*sp;

  acquire(&ptable.lock);

  int index = 0;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)

    if(p->state == UNUSED)

  {

    ptable\_index = index; // 해당 인덱스의 프로세스가 RUNNABLE되면, node로 초기화후, 런큐 삽입

      goto found;

  }

  release(&ptable.lock);

  return 0;

found:

  p->state = EMBRYO;

  p->pid = nextpid++;

  release(&ptable.lock);

  // Allocate kernel stack.

  if((p->kstack = kalloc()) == 0){

    p->state = UNUSED;

    return 0;

  }

  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.

  sp -= sizeof \*p->tf;

  p->tf = (struct trapframe\*)sp;

  // Set up new context to start executing at forkret,

  // which returns to trapret.

  sp -= 4;

  \*(uint\*)sp = (uint)trapret;

  sp -= sizeof \*p->context;

  p->context = (struct context\*)sp;

  memset(p->context, 0, sizeof \*p->context);

  p->context->eip = (uint)forkret;

  if(p->pid == 1 ||p->pid == 2)

    p->priority = 99;

  else

  {

    int i,queue\_min = 0;

    int find = 0;

    for(i = 0; i < RUN\_QUEUE\_SIZE; i++)

    {

      if((queue\_min = get\_least\_priority\_value(&runqueues[i])) < 0)

      continue;

      else

      {

        p->priority = queue\_min;

        find = 1;

        // 현재 큐에서 우선순위 값을 얻었다면 다음

        // 다음 큐보단 값이 작을 것이 자명하기 때문에 break

       break;

      }

    }

    if(!find)

      p->priority = 0; // PID 1 , 2 외에 값을 찾지 못함 기본값인 0 설정

  }

  p->proc\_tick = 0;

  p->cpu\_used  = 0;

  p->is\_time\_set = 0;

  p->remaining\_ticks =0;

  p->priority\_tick = 0;

  p->next = NULL;

  return p;

}

//PAGEBREAK: 32

// Set up first user process.

void

userinit(void)

{

  struct proc \*p;

  extern char \_binary\_initcode\_start[], \_binary\_initcode\_size[];

  p = allocproc();

  initproc = p;

  if((p->pgdir = setupkvm()) == 0)

    panic("userinit: out of memory?");

  inituvm(p->pgdir, \_binary\_initcode\_start, (int)\_binary\_initcode\_size);

  p->sz = PGSIZE;

  memset(p->tf, 0, sizeof(\*p->tf));

  p->tf->cs = (SEG\_UCODE << 3) | DPL\_USER;

  p->tf->ds = (SEG\_UDATA << 3) | DPL\_USER;

  p->tf->es = p->tf->ds;

  p->tf->ss = p->tf->ds;

  p->tf->eflags = FL\_IF;

  p->tf->esp = PGSIZE;

  p->tf->eip = 0;  // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));

  p->cwd = namei("/");

  // this assignment to p->state lets other cores

  // run this process. the acquire forces the above

  // writes to be visible, and the lock is also needed

  // because the assignment might not be atomic.

  acquire(&ptable.lock);

  int i;

  for(i = 0; i < RUN\_QUEUE\_SIZE; i++)

  {

  // runqueues의 모든 큐 초기화

  init\_queue(&runqueues[i]);

  }

  p->state = RUNNABLE;

  release(&ptable.lock);

}

// Grow current process's memory by n bytes.

// Return 0 on success, -1 on failure.

int

growproc(int n)

{

  uint sz;

  struct proc \*curproc = myproc();

  sz = curproc->sz;

  if(n > 0){

    if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)

      return -1;

  } else if(n < 0){

    if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)

      return -1;

  }

  curproc->sz = sz;

  switchuvm(curproc);

  return 0;

}

// Create a new process copying p as the parent.

// Sets up stack to return as if from system call.

// Caller must set state of returned proc to RUNNABLE.

int

fork(void)

{

  int i, pid;

  struct proc \*np;

  struct proc \*curproc = myproc();

  // Allocate process.

  if((np = allocproc()) == 0){

    return -1;

  }

  // Copy process state from proc.

  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){

    kfree(np->kstack);

    np->kstack = 0;

    np->state = UNUSED;

    return -1;

  }

  np->sz = curproc->sz;

  np->parent = curproc;

  \*np->tf = \*curproc->tf;

  // Clear %eax so that fork returns 0 in the child.

  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)

    if(curproc->ofile[i])

      np->ofile[i] = filedup(curproc->ofile[i]);

  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;

}

// Exit the current process.  Does not return.

// An exited process remains in the zombie state

// until its parent calls wait() to find out it exited.

void

exit(void)

{

  struct proc \*curproc = myproc();

  struct proc \*p;

  int fd;

  if(curproc == initproc)

    panic("init exiting");

  // Close all open files.

  for(fd = 0; fd < NOFILE; fd++){

    if(curproc->ofile[fd]){

      fileclose(curproc->ofile[fd]);

      curproc->ofile[fd] = 0;

    }

  }

  begin\_op();

  iput(curproc->cwd);

  end\_op();

  curproc->cwd = 0;

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().

  wakeup1(curproc->parent);

  // Pass abandoned children to init.

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

    if(p->parent == curproc){

      p->parent = initproc;

      if(p->state == ZOMBIE)

        wakeup1(initproc);

    }

  }

  // Jump into the scheduler, never to return.

  cprintf("PID : %d, priority : %d, proc\_tick : %d ticks, total\_cpu\_usage : %d ticks (3)\n",curproc->pid, curproc->priority,curproc->proc\_tick, curproc->cpu\_used);

  cprintf("PID : %d terminated\n",curproc->pid);

  pop\_proc(&runqueues[curproc->priority/4],curproc);

  curproc->state = ZOMBIE;

  sched();

  panic("zombie exit");

}

// Wait for a child process to exit and return its pid.

// Return -1 if this process has no children.

int

wait(void)

{

  struct proc \*p;

  int havekids, pid;

  struct proc \*curproc = myproc();

  acquire(&ptable.lock);

  for(;;){

    // Scan through table looking for exited children.

    havekids = 0;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

      if(p->parent != curproc)

        continue;

      havekids = 1;

      if(p->state == ZOMBIE){

        // Found one.

        pid = p->pid;

        kfree(p->kstack);

        p->kstack = 0;

        freevm(p->pgdir);

        p->pid = 0;

        p->parent = 0;

        p->name[0] = 0;

        p->killed = 0;

        p->state = UNUSED;

        release(&ptable.lock);

        return pid;

      }

    }

    // No point waiting if we don't have any children.

    if(!havekids || curproc->killed){

      release(&ptable.lock);

      return -1;

    }

    // Wait for children to exit.  (See wakeup1 call in proc\_exit.)

    sleep(curproc, &ptable.lock);  //DOC: wait-sleep

  }

}

//PAGEBREAK: 42

// Per-CPU process scheduler.

// Each CPU calls scheduler() after setting itself up.

// Scheduler never returns.  It loops, doing:

//  - choose a process to run

//  - swtch to start running that process

//  - eventually that process transfers control

//      via swtch back to the scheduler.

void

scheduler(void)

{

  struct cpu \*c = mycpu();

  c->proc = 0;

  for(;;)

  {

  sti();

  acquire(&ptable.lock);

  // Makefile SCHED\_POLICY 를 DEFAULT / SSU\_SCHEDULER 로 선택 하느냐에 따라 스케쥴러 선택 가능

  # ifdef DEFAULT

  struct proc \*p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)

  {

    if(p->state != RUNNABLE)

      continue;

    c-> proc = p;

    switchuvm(p);

    p->state = RUNNING;

    #ifdef DEBUG

    if(p)

    {

      cprintf("PID : %d, priority : %d, proc\_tick : %d ticks, total\_cpu\_used : %d ticks (2)\n",p->pid, p->priority, p->proc\_tick, p->cpu\_used);

      cprintf("PID : %d, NAME : XV6 SCHEDULER.\n",p->pid);

    }

    #endif

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

    switchkvm();

    c->proc = 0;

  }

#elif SSU\_SCHEDULER

  struct proc\* p;

  int index = 0;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++, index++)

  {

      if(p->state != RUNNABLE)

        continue;

    ptable\_index = index;

    // RUNNABLE인 프로세스가 런큐에 없다면 삽입한다.

    if(find\_proc(&runqueues[p->priority/4],p) < 0)

      push\_proc\_back(&runqueues[p->priority/4], p);

  }

  int i;

  for(i = 0; i < RUN\_QUEUE\_SIZE; i++)

  {

    // runqueue 에서 가장 우선순위 값이 작은(우선순위가 높은) 프로세스 선택한다.

    if((p = get\_priority\_proc(&runqueues[i])) == NULL)

      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;

    switchuvm(p);

    p->state = RUNNING;

    #ifdef DEBUG

      if(p)

      {

        cprintf("PID : %d, priority : %d, proc\_tick : %d ticks, total\_cpu\_used : %d ticks (2)\n",p->pid, p->priority, p->proc\_tick, p->cpu\_used);

        cprintf("PID : %d, NAME : SSU\_SCHEDULER.\n",p->pid);

      }

    #endif

    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;

    break;

  }

#else

#endif

  release(&ptable.lock);

  }

}

// Enter scheduler.  Must hold only ptable.lock

// and have changed proc->state. Saves and restores

// intena because intena is a property of this

// kernel thread, not this CPU. It should

// be proc->intena and proc->ncli, but that would

// break in the few places where a lock is held but

// there's no process.

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; // 인터럽트 가능 여부를 intena 변수에 저장한다.

  swtch(&p->context, mycpu()->scheduler); // swtch() : 스케쥴러와 문맥교환 담당

  mycpu()->intena = intena; // 인터럽트 가능 여부 복구

}

// Give up the CPU for one scheduling round.

void

yield(void)

{

  acquire(&ptable.lock);  //DOC: yieldlock

  myproc()->state = RUNNABLE;

  sched();

  release(&ptable.lock);

}

// A fork child's very first scheduling by scheduler()

// will swtch here.  "Return" to user space.

void

forkret(void)

{

  static int first = 1;

  // Still holding ptable.lock from scheduler.

  release(&ptable.lock);

  if (first) {

    // Some initialization functions must be run in the context

    // of a regular process (e.g., they call sleep), and thus cannot

    // be run from main().

    first = 0;

    iinit(ROOTDEV);

    initlog(ROOTDEV);

  }

  // Return to "caller", actually trapret (see allocproc).

}

// Atomically release lock and sleep on chan.

// Reacquires lock when awakened.

void

sleep(void \*chan, struct spinlock \*lk)

{

  struct proc \*p = myproc();

  if(p == 0)

    panic("sleep");

  if(lk == 0)

    panic("sleep without lk");

  // Must acquire ptable.lock in order to

  // change p->state and then call sched.

  // Once we hold ptable.lock, we can be

  // guaranteed that we won't miss any wakeup

  // (wakeup runs with ptable.lock locked),

  // so it's okay to release lk.

  if(lk != &ptable.lock){  //DOC: sleeplock0

    acquire(&ptable.lock);  //DOC: sleeplock1

    release(lk);

  }

  // Go to sleep.

  p->chan = chan;

  p->state = SLEEPING;

  sched();

  // Tidy up.

  p->chan = 0;

  // Reacquire original lock.

  if(lk != &ptable.lock){  //DOC: sleeplock2

    release(&ptable.lock);

    acquire(lk);

  }

}

//PAGEBREAK!

// Wake up all processes sleeping on chan.

// The ptable lock must be held.

static void

wakeup1(void \*chan)

{

  struct proc \*p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)

  {

    if(p->state == SLEEPING && p->chan == chan)

  {

      p->state = RUNNABLE;

    if(p->pid != 1 && p->pid != 2) // pid 1 2를 제외한 프로세스 wake up 시 우선순위 0

    p->priority = 0;

  }

  }

}

// Wake up all processes sleeping on chan.

void

wakeup(void \*chan)

{

  acquire(&ptable.lock);

  wakeup1(chan);

  release(&ptable.lock);

}

// Kill the process with the given pid.

// Process won't exit until it returns

// to user space (see trap in trap.c).

int

kill(int pid)

{

  struct proc \*p;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

    if(p->pid == pid){

      p->killed = 1;

      // Wake process from sleep if necessary.

      if(p->state == SLEEPING)

        p->state = RUNNABLE;

      release(&ptable.lock);

      return 0;

    }

  }

  release(&ptable.lock);

  return -1;

}

//PAGEBREAK: 36

// Print a process listing to console.  For debugging.

// Runs when user types ^P on console.

// No lock to avoid wedging a stuck machine further.

void

procdump(void)

{

  static char \*states[] = {

  [UNUSED]    "unused",

  [EMBRYO]    "embryo",

  [SLEEPING]  "sleep ",

  [RUNNABLE]  "runble",

  [RUNNING]   "run   ",

  [ZOMBIE]    "zombie"

  };

  int i;

  struct proc \*p;

  char \*state;

  uint pc[10];

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

    if(p->state == UNUSED)

      continue;

    if(p->state >= 0 && p->state < NELEM(states) && states[p->state])

      state = states[p->state];

    else

      state = "???";

    cprintf("%d %s %s", p->pid, state, p->name);

    if(p->state == SLEEPING){

      getcallerpcs((uint\*)p->context->ebp+2, pc);

      for(i=0; i<10 && pc[i] != 0; i++)

        cprintf(" %p", pc[i]);

    }

    cprintf("\n");

  }

}

int set\_sche\_info(int priority, int remaining\_tick)

{

  if(priority < 0 || remaining\_tick < 0)

    return -1;

  struct proc\* p = myproc();

  cprintf("set\_sche\_info() pid = %d\n", p->pid);

  acquire(&ptable.lock);  //DOC: yieldlock

  // 프로세스 초기 priority 및 종료 타이머 인자 전달

  // 이전 런큐의 우선순위 배열에서 뽑아 새로운 인덱스의 큐로 옮겨주기

  pop\_proc(&runqueues[p->priority/4], p);

  p->priority = priority;

  p->is\_time\_set = 1;

  p->remaining\_ticks = remaining\_tick;

  p->priority\_tick = 0;

  push\_proc\_back(&runqueues[p->priority/4], p);

  release(&ptable.lock);

  return 0;

}

void calculate\_new\_priority()

{

  struct proc \* p;

  // 60tick 마다 trap.c에서 호출

    // Loop over process table looking for process to run.

    acquire(&ptable.lock);

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)

  {

      if(p->state == RUNNABLE || p->state == RUNNING || p->state == SLEEPING)

    {

    // IDLE 프로세스는 제외(PID 1,2) == 우선순위 99 고정

    // 예시에서 확인해본바로, SLEEPING -> RUNNABLE 에도 우선순위 99 고정

    if(p->pid == 1 || p->pid == 2)

      continue;

    // 우선 순위 재조정 및 변수 초기화

    pop\_proc(&runqueues[p->priority/4],p);

    int new\_priority = p->priority + (p->priority\_tick / 10);

    // 프로세스가 가질 수 있는  우선순위는 0 ~ 99 입니다.

    if(new\_priority > 99)

      new\_priority = 99;

    p->priority = new\_priority;

    push\_proc\_back(&runqueues[p->priority/4],p);

    // 우선순위 재조정 이후 관련 변수 초기화

    p->priority\_tick = 0;

    }

  }

  release(&ptable.lock);

}

4-6. proc.h

// Per-CPU state

struct cpu {

  uchar apicid;                // Local APIC ID

  struct context \*scheduler;   // swtch() here to enter scheduler

  struct taskstate ts;         // Used by x86 to find stack for interrupt

  struct segdesc gdt[NSEGS];   // x86 global descriptor table

  volatile uint started;       // Has the CPU started?

  int ncli;                    // Depth of pushcli nesting.

  int intena;                  // Were interrupts enabled before pushcli?

  struct proc \*proc;           // The process running on this cpu or null

};

extern struct cpu cpus[NCPU];

extern int ncpu;

//PAGEBREAK: 17

// Saved registers for kernel context switches.

// Don't need to save all the segment registers (%cs, etc),

// because they are constant across kernel contexts.

// Don't need to save %eax, %ecx, %edx, because the

// x86 convention is that the caller has saved them.

// Contexts are stored at the bottom of the stack they

// describe; the stack pointer is the address of the context.

// The layout of the context matches the layout of the stack in swtch.S

// at the "Switch stacks" comment. Switch doesn't save eip explicitly,

// but it is on the stack and allocproc() manipulates it.

struct context {

  uint edi;

  uint esi;

  uint ebx;

  uint ebp;

  uint eip;

};

enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// 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

  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)

  // 설계 과제 3을 위한 추가 변수

  int priority;                 // 프로세스의 우선순위

  int proc\_tick;                // 현재 프로세스가 차지한 CPU tick수

  int cpu\_used;                 // 총 CPU 이용 시간

  int priority\_tick;            // 마지막으로 우선순위 재조정 한 후 tick

  struct proc\* next;

  int is\_time\_set;              // set\_sche\_info() 초기 설정된 프로세스 인가?

  int remaining\_ticks;          // 남은 tick

};

// Process memory is laid out contiguously, low addresses first:

//   text

//   original data and bss

//   fixed-size stack

//   expandable heap

4-7. scheduler\_test.c

#include "types.h"

#include "stat.h"

#include "user.h"

#ifndef PNUM

    #define PNUM 3

#endif

void scheduler\_func(void)

{

    int pid,i = 0;

    printf(1,"start scheduler\_test\n");

    for(i = 0; i < PNUM; i++)

    {

        if((pid = fork()) < 0)

            exit();

        else if(pid == 0)

        {

            if((i % 3) == 0)

                set\_sche\_info(1, 110);

            else if((i % 3) == 1)

                set\_sche\_info(22, 200);

            else if((i % 3) == 2)

                set\_sche\_info(11, 250);

            while(1);

        }

    }

    for(i = 0; i < PNUM; i++)

        wait();

}

int main(void)

{

    scheduler\_func();

    printf(2,"end of scheduler test\n");

    exit();

}

4-8. syscall.c

#include "types.h"

#include "defs.h"

#include "param.h"

#include "memlayout.h"

#include "mmu.h"

#include "proc.h"

#include "x86.h"

#include "syscall.h"

// User code makes a system call with INT T\_SYSCALL.

// System call number in %eax.

// Arguments on the stack, from the user call to the C

// library system call function. The saved user %esp points

// to a saved program counter, and then the first argument.

// Fetch the int at addr from the current process.

int

fetchint(uint addr, int \*ip)

{

  struct proc \*curproc = myproc();

  if(addr >= curproc->sz || addr+4 > curproc->sz)

    return -1;

  \*ip = \*(int\*)(addr);

  return 0;

}

// Fetch the nul-terminated string at addr from the current process.

// Doesn't actually copy the string - just sets \*pp to point at it.

// Returns length of string, not including nul.

int

fetchstr(uint addr, char \*\*pp)

{

  char \*s, \*ep;

  struct proc \*curproc = myproc();

  if(addr >= curproc->sz)

    return -1;

  \*pp = (char\*)addr;

  ep = (char\*)curproc->sz;

  for(s = \*pp; s < ep; s++){

    if(\*s == 0)

      return s - \*pp;

  }

  return -1;

}

// Fetch the nth 32-bit system call argument.

int

argint(int n, int \*ip)

{

  return fetchint((myproc()->tf->esp) + 4 + 4\*n, ip);

}

// Fetch the nth word-sized system call argument as a pointer

// to a block of memory of size bytes.  Check that the pointer

// lies within the process address space.

int

argptr(int n, char \*\*pp, int size)

{

  int i;

  struct proc \*curproc = myproc();

  if(argint(n, &i) < 0)

    return -1;

  if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)

    return -1;

  \*pp = (char\*)i;

  return 0;

}

// Fetch the nth word-sized system call argument as a string pointer.

// Check that the pointer is valid and the string is nul-terminated.

// (There is no shared writable memory, so the string can't change

// between this check and being used by the kernel.)

int

argstr(int n, char \*\*pp)

{

  int addr;

  if(argint(n, &addr) < 0)

    return -1;

  return fetchstr(addr, pp);

}

extern int sys\_chdir(void);

extern int sys\_close(void);

extern int sys\_dup(void);

extern int sys\_exec(void);

extern int sys\_exit(void);

extern int sys\_fork(void);

extern int sys\_fstat(void);

extern int sys\_getpid(void);

extern int sys\_kill(void);

extern int sys\_link(void);

extern int sys\_mkdir(void);

extern int sys\_mknod(void);

extern int sys\_open(void);

extern int sys\_pipe(void);

extern int sys\_read(void);

extern int sys\_sbrk(void);

extern int sys\_sleep(void);

extern int sys\_unlink(void);

extern int sys\_wait(void);

extern int sys\_write(void);

extern int sys\_uptime(void);

extern int sys\_set\_sche\_info(void);

int (\*syscalls[])(void) = {

[SYS\_fork]    sys\_fork,

[SYS\_exit]    sys\_exit,

[SYS\_wait]    sys\_wait,

[SYS\_pipe]    sys\_pipe,

[SYS\_read]    sys\_read,

[SYS\_kill]    sys\_kill,

[SYS\_exec]    sys\_exec,

[SYS\_fstat]   sys\_fstat,

[SYS\_chdir]   sys\_chdir,

[SYS\_dup]     sys\_dup,

[SYS\_getpid]  sys\_getpid,

[SYS\_sbrk]    sys\_sbrk,

[SYS\_sleep]   sys\_sleep,

[SYS\_uptime]  sys\_uptime,

[SYS\_open]    sys\_open,

[SYS\_write]   sys\_write,

[SYS\_mknod]   sys\_mknod,

[SYS\_unlink]  sys\_unlink,

[SYS\_link]    sys\_link,

[SYS\_mkdir]   sys\_mkdir,

[SYS\_close]   sys\_close,

[SYS\_set\_sche\_info] sys\_set\_sche\_info,

};

void

syscall(void)

{

  int num;

  struct proc \*curproc = myproc();

  num = curproc->tf->eax;

  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {

    curproc->tf->eax = syscalls[num]();

  } else {

    cprintf("%d %s: unknown sys call %d\n",

            curproc->pid, curproc->name, num);

    curproc->tf->eax = -1;

  }

}

4-9. syscall.h

// System call numbers

#define SYS\_fork    1

#define SYS\_exit    2

#define SYS\_wait    3

#define SYS\_pipe    4

#define SYS\_read    5

#define SYS\_kill    6

#define SYS\_exec    7

#define SYS\_fstat   8

#define SYS\_chdir   9

#define SYS\_dup    10

#define SYS\_getpid 11

#define SYS\_sbrk   12

#define SYS\_sleep  13

#define SYS\_uptime 14

#define SYS\_open   15

#define SYS\_write  16

#define SYS\_mknod  17

#define SYS\_unlink 18

#define SYS\_link   19

#define SYS\_mkdir  20

#define SYS\_close  21

#define SYS\_set\_sche\_info 22

4-10. sysproc.c

#include "types.h"

#include "x86.h"

#include "defs.h"

#include "date.h"

#include "param.h"

#include "memlayout.h"

#include "mmu.h"

#include "proc.h"

int

sys\_fork(void)

{

  return fork();

}

int

sys\_exit(void)

{

  exit();

  return 0;  // not reached

}

int

sys\_wait(void)

{

  return wait();

}

int

sys\_kill(void)

{

  int pid;

  if(argint(0, &pid) < 0)

    return -1;

  return kill(pid);

}

int

sys\_getpid(void)

{

  return myproc()->pid;

}

int

sys\_sbrk(void)

{

  int addr;

  int n;

  if(argint(0, &n) < 0)

    return -1;

  addr = myproc()->sz;

  if(growproc(n) < 0)

    return -1;

  return addr;

}

int

sys\_sleep(void)

{

  int n;

  uint ticks0;

  if(argint(0, &n) < 0)

    return -1;

  acquire(&tickslock);

  ticks0 = ticks;

  while(ticks - ticks0 < n){

    if(myproc()->killed){

      release(&tickslock);

      return -1;

    }

    sleep(&ticks, &tickslock);

  }

  release(&tickslock);

  return 0;

}

// return how many clock tick interrupts have occurred

// since start.

int

sys\_uptime(void)

{

  uint xticks;

  acquire(&tickslock);

  xticks = ticks;

  release(&tickslock);

  return xticks;

}

int

sys\_set\_sche\_info(void)

{

  int priority, tick;

  if(argint(0, &priority) < 0 || argint(1, &tick) < 0)

    return -1;

  return set\_sche\_info(priority, tick);

}

4-11. trap.c

#include "types.h"

#include "defs.h"

#include "param.h"

#include "memlayout.h"

#include "mmu.h"

#include "proc.h"

#include "x86.h"

#include "traps.h"

#include "spinlock.h"

// 기존 XV6 스케쥴러와 SSU\_SCHEDULER를 선택함에 따라 TIME QUANTUM을 다르게 설정

#ifdef DEFAULT

 #define TIMEQUANTUM 1

#elif SSU\_SCHEDULER

 #define TIMEQUANTUM  30  // 기존 1 tick의 타임 슬라이싱을 30으로 맞추기 위한 매크로

#endif

// Interrupt descriptor table (shared by all CPUs).

struct gatedesc idt[256];

extern uint vectors[];  // in vectors.S: array of 256 entry pointers

struct spinlock tickslock;

uint ticks;

uint cpu\_used\_ticks;

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);

  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 T\_IRQ0 + IRQ\_TIMER:

    if(cpuid() == 0){

      acquire(&tickslock);

      ticks++;

      wakeup(&ticks);

      release(&tickslock);

    }

    lapiceoi();

    break;

  case T\_IRQ0 + IRQ\_IDE:

    ideintr();

    lapiceoi();

    break;

  case T\_IRQ0 + IRQ\_IDE+1:

    // Bochs generates spurious IDE1 interrupts.

    break;

  case T\_IRQ0 + IRQ\_KBD:

    kbdintr();

    lapiceoi();

    break;

  case T\_IRQ0 + IRQ\_COM1:

    uartintr();

    lapiceoi();

    break;

  case T\_IRQ0 + 7:

  case T\_IRQ0 + IRQ\_SPURIOUS:

    cprintf("cpu%d: spurious interrupt at %x:%x\n",

            cpuid(), tf->cs, tf->eip);

    lapiceoi();

    break;

  //PAGEBREAK: 13

  default:

    if(myproc() == 0 || (tf->cs&3) == 0){

      // In kernel, it must be our mistake.

      cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",

              tf->trapno, cpuid(), tf->eip, rcr2());

      panic("trap");

    }

    // In user space, assume process misbehaved.

    cprintf("pid %d %s: trap %d err %d on cpu %d "

            "eip 0x%x addr 0x%x--kill proc\n",

            myproc()->pid, myproc()->name, tf->trapno,

            tf->err, cpuid(), tf->eip, rcr2());

    myproc()->killed = 1;

  }

  // Force process exit if it has been killed and is in user space.

  // (If it is still executing in the kernel, let it keep running

  // until it gets to the regular system call return.)

  if(myproc() && myproc()->killed && (tf->cs&3) == DPL\_USER)

    exit();

  if(myproc() && myproc()->state == RUNNING && tf->trapno == T\_IRQ0 + IRQ\_TIMER && (tf->cs&3) == DPL\_USER)

  {

  cpu\_used\_ticks++;

  // 틱 관련 변수들 증가

  myproc()->proc\_tick++;

  myproc()->cpu\_used++;

  myproc()->priority\_tick++;

  // 60 tick 마다 priority 재 계산

  if((cpu\_used\_ticks % 60) == 0)

    calculate\_new\_priority();

  // set\_sche\_info() 통해 만든 지정한 시간 지나면 자식 프로세스들 종료

  if(myproc()->is\_time\_set && myproc()->remaining\_ticks)

  {

    myproc()->remaining\_ticks--;

    if(myproc()->remaining\_ticks == 0)

      myproc()->killed = 1;

  }

  }

  // 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)

  // 타임 슬라이싱 (TIMEQUANTUM == 30)

  if(myproc()->proc\_tick == TIMEQUANTUM)

  {

    cprintf("PID : %d, priority : %d, proc\_tick : %d ticks, total\_cpu\_usage : %d ticks (1)\n",myproc()->pid,myproc()->priority,myproc()->proc\_tick,myproc()->cpu\_used);

    myproc()->proc\_tick = 0;

    yield();

  }

  // Check if the process has been killed since we yielded

  if(myproc() && myproc()->killed && (tf->cs&3) == DPL\_USER)

    exit();

}

4-12. user.h

struct stat;

struct rtcdate;

// system calls

int fork(void);

int exit(void) \_\_attribute\_\_((noreturn));

int wait(void);

int pipe(int\*);

int write(int, const void\*, int);

int read(int, void\*, int);

int close(int);

int kill(int);

int exec(char\*, char\*\*);

int open(const char\*, int);

int mknod(const char\*, short, short);

int unlink(const char\*);

int fstat(int fd, struct stat\*);

int link(const char\*, const char\*);

int mkdir(const char\*);

int chdir(const char\*);

int dup(int);

int getpid(void);

char\* sbrk(int);

int sleep(int);

int uptime(void);

int set\_sche\_info(int, int);

// ulib.c

int stat(const char\*, struct stat\*);

char\* strcpy(char\*, const char\*);

void \*memmove(void\*, const void\*, int);

char\* strchr(const char\*, char c);

int strcmp(const char\*, const char\*);

void printf(int, const char\*, ...);

char\* gets(char\*, int max);

uint strlen(const char\*);

void\* memset(void\*, int, uint);

void\* malloc(uint);

void free(void\*);

int atoi(const char\*);

4-13. usys.S

#include "syscall.h"

#include "traps.h"

#define SYSCALL(name) \

  .globl name; \

  name: \

    movl $SYS\_ ## name, %eax; \

    int $T\_SYSCALL; \

    ret

SYSCALL(fork)

SYSCALL(exit)

SYSCALL(wait)

SYSCALL(pipe)

SYSCALL(read)

SYSCALL(write)

SYSCALL(close)

SYSCALL(kill)

SYSCALL(exec)

SYSCALL(open)

SYSCALL(mknod)

SYSCALL(unlink)

SYSCALL(fstat)

SYSCALL(link)

SYSCALL(mkdir)

SYSCALL(chdir)

SYSCALL(dup)

SYSCALL(getpid)

SYSCALL(sbrk)

SYSCALL(sleep)

SYSCALL(uptime)

SYSCALL(set\_sche\_info)