컴퓨터학부 20221867 이채은

**1. 개요**

-xv6 운영체제의 기존 스케줄러를 분석하고, 다단계 피드백 큐 스케줄링(MLFQ)을 동적 우선순위 조정 기능을 포함하여 구현한다.

* 1번에서는 xv6의 기존 스케줄러를 함수 단위로 분석하여 현재 스케줄링 기법의 동작 방식을 이해하고, 함수 간의 콜 그래프를 작성한다.
* 2번에서는 다단계 피드백 큐 스케줄링(MLFQ)을 구현하여 프로세스의 실행 패턴에 따라 우선순위를 동적으로 조정하고, Aging을 통해 낮은 우선순위의 프로세스도 적절한 시간 내에 실행되도록 한다.
* 3번에서는 구현한 SSU 스케줄러의 성능을 검증하기 위한 테스트 프로그램을 개발하여, 다양한 프로세스의 초기 조건을 설정하고 스케줄러의 동작을 관찰하여 시스템 성능을 분석한다. 더불어, 4번에서 기존 xv6 스케줄러와 새로 구현한 SSU 스케줄러의 기능적 차이와 성능적 차이를 비교 분석한다.

**2. 상세 설계 & 3. 실행 결과 & 4. 주석 달린 소스코드**

**- 1. 기존 xv6 스케줄러 분석**

✓ xv6 동작 방식과 scheduler() 찾는 과정

- xv6는 시스템의 모든 프로세스를 ptable에서 관리, ptable은 모든 프로세스의 정보가 저장된 배열이며, 최대 프로세스 수(NPROC)로 제한한다.

|  |
| --- |
| proc.c |
| A black background with white text  Description automatically generated |

- ptable.proc 배열에 각 프로세스가 들어 있으며, 각 프로세스는 struct proc 구조체로 관리, proc 구조체는 proc.h 헤더에 아래와 같이 선언(프로세스의 상태와 정보가 담겨 있다.

|  |
| --- |
| proc.h |
|  |

-눈에 띄는 것은 state이고, 프로세스가 어떤 상태인지 나타내는 아주 중요한 지표로, 열거형으로 선언되어 있다.



- xv6는 Round Robin 형태로 프로세스를 관리한다. 이를 구현하기 위해 앞서 언급한 proc 구조체가 필요한 것이다. NPROC은 xv6가 최대로 관리할 수 있는 프로세스의 수를 의미하며 default값은 64이다.(param.h)

-이제 운영체제의 부팅 과정과 커널 초기화를 담당하는 main.c의 코드를 보면 아래와 같다.

|  |
| --- |
| main.c |
| - 맨 아래의 mpmain()을 보면 여기에 scheduler()를 호출하는 것을 볼 수 있다. 이 함수는 proc.c 안에 있었다. |

✓ scheduler() 분석

- xv6 스케줄러의 핵심은 이 scheduler() 함수이다. Round Robin 으로 돌아가고 있고, 외부 인터럽트가 발생하지 않는 이상 조건 없는 for 루프를 돌면서 proc table 안의 프로세스 정보를 탐색한다.

|  |
| --- |
| scheduler() 코드에 주석 추가 |
| // Per-CPU process scheduler.  // 각 CPU가 실행할 프로세스를 선택하고 실행하는 스케줄러 함수입니다.  // 스케줄러는 절대 종료되지 않으며 무한 루프 안에서 계속 동작합니다.  void scheduler(void) {    struct proc \*p;              // 프로세스를 가리키는 포인터    struct cpu \*c = mycpu();      // 현재 이 스케줄러를 호출한 CPU 가져오기    c->proc = 0;                  // 초기화: 현재 CPU에 실행 중인 프로세스가 없음    for(;;){                      // 무한 루프: 스케줄러는 종료되지 않음      sti();                      // 인터럽트 활성화 (외부 이벤트 처리 가능)      acquire(&ptable.lock);      // 프로세스 테이블에 대한 락 획득 (동시 접근 방지)      // 프로세스 테이블 순회: 실행 가능한 프로세스 찾기      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){        if(p->state != RUNNABLE)  // RUNNABLE 상태가 아니면 건너뜀          continue;        // 선택된 프로세스를 현재 CPU에 할당        c->proc = p;        switchuvm(p);             // 유저 모드 메모리로 전환        p->state = RUNNING;       // 프로세스 상태를 RUNNING(실행 중)으로 변경        // 스케줄러의 컨텍스트에서 선택된 프로세스의 컨텍스트로 문맥 교환        swtch(&(c->scheduler), p->context);        switchkvm();              // 커널 모드 메모리로 복귀        // 실행이 끝난 후 현재 CPU에서 실행 중인 프로세스 초기화        c->proc = 0;      }      release(&ptable.lock);      // 락 해제 (다른 CPU가 접근 가능하도록 함)    }  } |

A screenshot of a computer program

Description automatically generated

- 순서도를 대략적으로 그리면 이런 구조이다.

- 그리고 scheduler() 안에 있는 swtch() 함수를 잠깐 확인하고 넘어가봤다.

|  |
| --- |
| swtch.S에 주석 추가 |
| # swtch.S: 문맥 교환(Context Switch) 구현  # void swtch(struct context \*\*old, struct context \*new);  # 현재 프로세스의 레지스터 상태를 스택에 저장한 후,  # 그 상태의 주소를 \*old에 저장합니다.  # 새로운 프로세스의 스택 포인터로 전환한 후,  # 새 프로세스의 레지스터를 복원하고 실행을 재개합니다.  .globl swtch # 외부에서 접근 가능한 전역 함수로 선언  swtch:  # 함수 호출 스택에서 첫 번째와 두 번째 인자 가져오기  # 첫 번째 인자 (old) 주소를 %eax에 저장  movl 4(%esp), %eax    # 두 번째 인자 (new) 주소를 %edx에 저장  movl 8(%esp), %edx  # 1. 현재 프로세스의 레지스터 상태를 저장 (callee-saved registers)  pushl %ebp # 현재 베이스 포인터(EBP) 저장  pushl %ebx # 범용 레지스터(EBX) 저장  pushl %esi # 소스 인덱스 레지스터(ESI) 저장  pushl %edi # 목적 인덱스 레지스터(EDI) 저장  # 2. 스택 포인터를 현재 프로세스의 컨텍스트에 저장  # 현재 프로세스의 %esp 값을 \*old가 가리키는 곳에 저장  movl %esp, (%eax)  # 3. 스택을 새 프로세스로 전환  # 새로운 프로세스의 스택 포인터 %esp로 전환  movl %edx, %esp  # 4. 새로운 프로세스의 레지스터를 복원  # 새 프로세스의 스택에 저장된 레지스터들을 pop하여 복구  popl %edi # 새 프로세스의 %edi 복원  popl %esi # 새 프로세스의 %esi 복원  popl %ebx # 새 프로세스의 %ebx 복원  popl %ebp # 새 프로세스의 %ebp 복원  # 5. 문맥 교환 완료 후 복귀  # 새 프로세스의 실행을 재개  ret # 스택에 저장된 복귀 주소로 점프 |

- 이 코드를 통해 문맥 교환의 전체 흐름을 알 수 있었다. 그리고 scheduler 및 sched 함수의 콜 그래프를 구현하면 아래와 같이 생겼고, 주요 함수의 세부 기능도 명시하였다.

A diagram of a computer network

Description automatically generatedA diagram of a switch

Description automatically generated

**scheduler:** 커널의 주요 스케줄링 함수로, 현재 프로세스를 중단하고 다른 프로세스를 실행하는 역할

-> 동작: 프로세스의 상태를 바꾸고, 스케줄링 가능한 프로세스를 선택하여 문맥 전환

**acquire:** 락을 획득하는 함수

-> 동작: 스케줄러에서 ptable에 접근하기 위해 락을 획득하여 다른 CPU가 동시에 접근하지 못하도록 보호

**release:** 락을 해제하는 함수

-> 동작: 스케줄러가 프로세스 테이블에 대한 작업을 마친 후, 락을 해제하여 다른 프로세스나 CPU가 테이블에 접근할 수 있도록 함.

**switchuvm:** 프로세스의 사용자 가상 메모리로 전환하는 함수

-> 동작: 현재 실행할 프로세스의 사용자 페이지 테이블로 전환하여, 해당 프로세스가 정상적으로 사용자 메모리에 접근할 수 있도록 함.

**switchkvm:** 커널 가상 메모리로 전환하는 함수

-> 동작: 스케줄링을 수행하기 위해 커널 모드에서 커널 페이지 테이블로 전환

**swtch:** 문맥 전환(context switch)을 수행하는 함수입니다.

-> 동작: 현재 실행 중인 프로세스의 CPU 상태를 저장하고, 새로운 프로세스를 실행할 수 있도록 CPU 상태를 복원

**pushcli:** 인터럽트 비활성화를 위한 함수

-> 동작: 현재 CPU의 인터럽트 플래그를 비활성화하여, 스케줄러가 실행되는 동안 인터럽트로 인해 문맥 전환이 방해되지 않도록 함

**popcli:** 인터럽트 활성화를 위한 함수

-> 동작: 스케줄링 작업이 끝난 후, CPU의 인터럽트 플래그를 다시 활성화하여 인터럽트를 허용

**holding**: 락이 현재 획득되어 있는지 확인하는 함수

-> 동작: 특정 락이 현재 프로세스나 CPU에 의해 유지되고 있는지 확인하는 용도로 사용

**getcallerpcs:** 호출 스택을 추적하는 함수

-> 동작: 함수가 호출된 스택 주소들을 가져와서, 디버깅 및 호출 경로를 확인하는 데 사용

**panic:** 커널 패닉을 발생시키는 함수

-> 동작: 치명적인 오류가 발생했을 때, 커널이 더 이상 안전하게 실행될 수 없음을 알리고 실행을 중단, 스케줄러에서 예상하지 못한 상황이나불가능한 상태에 도달했을 때 호출

**holding:** 현재 락을 보유 중인지 확인

-> 동작: sched 함수에서 락이 이미 획득된 상태인지 검사하여 안전하게 실행될 수 있도록 보장

A diagram of a computer system

Description automatically generated

- 교수님께서 참고하라고 주신 링크를 보며 예시 프로세스에 대한 대략적인 순서를 따라 그려보았다.

**- 2. 새로운 SSU Scheduler 구현**

**- (1) 큐와 프로세스 구조 변경**

✓ proc struct에 필요한 멤버 변수 추가

|  |
| --- |
| proc.h의 struct proc { } 안에 선언 추가 |
| A screen shot of a computer program  Description automatically generated |

**- (2) 다단계 피드백 큐 구성 & (3) 프로세스 큐 이동 및 우선 순위 조정**

✓ proc struct에 있는 필드 멤버를 이용하여 MLFQ처럼 동작하도록 구현했다. 이 코드 안에 Time Quantum 내에 실행을 완료하지 못하면, 한 단계 낮은 우선 순위 큐로 이동하는 기능이 있고, Aging도 있다. 주석을 상세하게 포함하였으며, 이 코드와 함께 스케줄러 코드를 함께 보아야 한다. 스케줄러 코드는 **(4) 스케줄러 변경** 쪽에 넣었다.

|  |
| --- |
| trap.c에 있는 trap 함수에서 수정한 부분 (주석 첨부) |
| void  trap(struct trapframe \*tf)  {  // 시스템 호출 처리  if(tf->trapno == T\_SYSCALL){  if(myproc()->killed) // 프로세스가 종료될 상태라면 exit 호출  exit();  myproc()->tf = tf; // 현재 프로세스의 trapframe 설정  syscall(); // 시스템 호출 처리  if(myproc()->killed) // 시스템 호출 후에도 종료될 상태라면 exit 호출  exit();  return; // 시스템 호출 처리 후 리턴  }  int tq[4] = {10, 20, 40, 80}; // 각 큐 레벨에 따른 타임퀀텀 값  switch(tf->trapno){  case T\_IRQ0 + IRQ\_TIMER: // 타이머 인터럽트 처리  if(cpuid() == 0){ // CPU 0에서만 ticks 증가 처리  acquire(&tickslock); // ticks 변수를 보호하기 위해 락 획득  ticks++; // 시스템 시간 증가  increase\_cpu\_wait(); // CPU 대기 시간 증가 처리 함수 호출, Aging도 처리    struct proc\* cur = myproc(); // 현재 실행 중인 프로세스 포인터  if(cur) {  cur->cpu\_wait = 0; // 현재 프로세스의 cpu\_wait 초기화  cur->cpu\_burst++; // 현재 프로세스의 CPU burst 증가  cur->cpu\_usage++; // CPU 사용량 증가  cur->is\_q\_timeout = 0; // 큐 타임아웃 상태 초기화  // 현재 프로세스가 PID 4 이상이고, 종료 시간이 설정되었으며,  // 해당 큐에서 주어진 타임퀀텀을 모두 사용했는지 확인  if(cur->pid >= 4 && cur->end\_time != -1 && cur->cpu\_burst / tq[cur->q\_level] >= 1) {  #ifdef DEBUG  cprintf("PID: %d uses %d ticks in mlfq[%d], total(%d/%d)\n", cur->pid, cur->cpu\_burst, cur->q\_level, cur->cpu\_usage, cur->end\_time);  #endif  cur->cpu\_burst = 0; // CPU burst 초기화  cur->is\_q\_timeout = 1; // 타임아웃 설정  // 큐 레벨 증가 (단, 가장 높은 큐는 3이므로 그 이상은 증가하지 않음)  if(cur->q\_level < 3)  cur->q\_level++;  }  // 프로세스가 CPU 사용량을 다 사용했는지 확인하여 종료 처리  if(cur->pid >= 4 && cur->end\_time != -1 && cur->cpu\_usage >= cur->end\_time) {  #ifdef DEBUG  cprintf("PID: %d uses %d ticks in mlfq[%d], total(%d/%d)\n", cur->pid, cur->cpu\_burst, cur->q\_level, cur->cpu\_usage, cur->end\_time);  #endif  cur->aging\_time = 1; // Aging 시간 설정  kill(cur->pid); // 프로세스 종료  cprintf("PID: %d, used %d ticks. terminated\n", cur->pid, cur->end\_time); // 종료 메시지 출력  }  }  wakeup(&ticks); // ticks에 대해 대기 중인 프로세스 깨우기  release(&tickslock); // 락 해제  }  lapiceoi(); // End of interrupt 신호 보내기  break;  // 다른 인터럽트 처리...  }  } |

✓ Aging 구현은 위의 trap 함수 코드와 아래의 함수들을 바탕으로 구현하였다.

|  |
| --- |
| proc.c에 추가한*struct* proc*\** get\_runnable\_procs(*struct* proc *\*runnable\_procs[]*, *int* *\*num\_runnable*) 함수 |
| *struct* proc*\** get\_runnable\_procs(*struct* proc *\*runnable\_procs[]*, *int* *\*num\_runnable*) {  *struct* proc *\**p;  *int* count = 0;  acquire(*&*ptable.lock);  *for* (p = ptable.proc; p < *&*ptable.proc[NPROC]; p++) {  *if* (p->state == RUNNABLE) {  runnable\_procs[count++] = p;  *if* (count >= NPROC) {  *break*;// *최대 러너블 프로세스 수를 초과 -> 종료*  }  }  }  release(*&*ptable.lock);  *\**num\_runnable = count;// *실제 러너블한 프로세스 수 반환*  *return* count > 0 ? runnable\_procs[0] : 0;// *첫 번째 러너블 프로세스 반환*  } |

- 러너블한 프로세스를 반환받기 위해 만들었다.

|  |
| --- |
| defs.h에 추가한get\_runnable\_procs함수 |
| *struct* proc*\** get\_runnable\_procs(*struct proc\**[], *int\**); |

- *void* yield(*void*); 아래에 추가했다.

|  |
| --- |
| trap.c에 추가한 increase\_cpu\_wait() 함수 |
| // *모든 러너블 프로세스의 cpu\_wait를 증가시키는 함수*  *void* increase\_cpu\_wait() {  *struct* proc *\**runnable\_procs[NPROC];  *int* num\_runnable = 0;  // *모든 러너블 프로세스를 가져오기*  get\_runnable\_procs(runnable\_procs, *&*num\_runnable);  // *각 러너블 프로세스의 cpu\_wait 증가*  *for* (*int* i = 0; i < num\_runnable; i++) {  runnable\_procs[i]->cpu\_wait++;  // *SLEEPING 상태인 경우 io\_wait 증가*  *if* (runnable\_procs[i]->state == SLEEPING) {  runnable\_procs[i]->io\_wait\_time++;  }  // *cpu\_wait가 250 이상*  *if* (runnable\_procs[i]->cpu\_wait >= 250) {  *if* (runnable\_procs[i]->q\_level > 0) {  runnable\_procs[i]->q\_level*--*;  #*ifdef* DEBUG  cprintf("*PID: %d Aging\n*",  runnable\_procs[i]->pid);  #*endif*  }  runnable\_procs[i]->cpu\_wait = 0;// *초기화*  for\_aging = 1;  }  }  } |

- 여기에 Aging 처리 부분을 구현했다.

✓ trap.c에서 큐에 할당된 시간을 다 썼으면 yield하게 하였다.

|  |  |
| --- | --- |
| proc.c의 yield 함수 수정 | trap.c의 trap 함수 수정 |
| **A computer code with text  Description automatically generated** | **A black background with white text  Description automatically generated** |

**- (4) 스케줄러 변경**

✓ 과제에서 주어진 대로 동작하기 위해 스케줄러 함수를 변경했다. 0레벨 큐부터 검사를 시행하고 있으며, test1-3의 출력을 위해 일부 바꾼 부분도 있다. trap.c와 함께 봐야한다.

|  |
| --- |
| scheduler 함수 변경 (주석 첨부) |
| *void* scheduler(*void*) {  *struct* proc *\**p, *\**selected\_proc;  *struct* cpu *\**c = mycpu();  c->proc = 0;  *for* (;;) {  sti();// *인터럽트 활성화*  acquire(*&*ptable.lock);  // *가장 최근에 수행된 프로세스*  // *초기값은 프로세스 테이블의 마지막 프로세스*  *struct* proc *\**recently\_p = p = *&*ptable.proc[NPROC *-* 1];  *for* (*int* level = 0; level < 4; level++) {  selected\_proc = 0;  // *역순으로 순회하여 높은 PID 순으로 선택*  *for* (p = recently\_p; p >= *&*ptable.proc[0];) {  // *RUNNABLE 아니거나 현재 큐레벨에 맞지 않는 경우 다음 프로세스 검색*  *if* (p->state != RUNNABLE || p->q\_level != level) {  p*--*;  *continue*;  }  selected\_proc = p;  // *프로세스 스케줄링*  c->proc = selected\_proc;  switchuvm(selected\_proc);  selected\_proc->state = RUNNING;  swtch(*&*(c->scheduler), selected\_proc->context);  switchkvm();  // *만약 Aging이 발생한 경우*  *if* (for\_aging) {  // *프로세스 마지막부터 다시 순회*  recently\_p = *&*ptable.proc[NPROC *-* 1];  level = 0;  for\_aging = 0;// *초기화*  *break*;  }  // *end\_time에 다다르면 p--*  *if* (selected\_proc->pid < 4 || (selected\_proc->pid >=4 && selected\_proc->aging\_time == 1)) {  p*--*;  }  // *end\_time에 다다르지 못했다면 현재 프로세스 실행 계속*  *else* {  recently\_p = p;  }  // *현재 CPU에서 실행 중인 프로세스 해제*  c->proc = 0;  }  }  release(*&*ptable.lock);  }  } |

**- (5) 프로세스 생성 시 초기화**

✓ allocproc 함수에 초기화 코드를 넣었다.

|  |
| --- |
| proc.c에 있는 allocproc 함수 수정 |
| // 따로 함수를 정의  *int strcmp\_xv6(const char \*s1, const char \*s2) {*  *while (\*s1 && (\*s1 == \*s2)) {*  *s1++;*  *s2++;*  *}*  *return \*(unsigned char \*)s1 - \*(unsigned char \*)s2;*  *}*  *static* *struct* proc*\**  allocproc(*void*)  {  … // 기존 코드와 동일  found:  p->state = EMBRYO;  p->pid = nextpid++;  // *프로세스 확인*  *if* (strcmp\_xv6(p->name, "*sh*") == 0 || strcmp\_xv6(p->name, "*idle*") == 0 || strcmp\_xv6(p->name, "*init*") == 0) {  p->q\_level = 3;// *shell, idle, init 프로세스는 항상 레벨 3*  } *else* {  p->q\_level = 0;// *이후 생성된 프로세스는 레벨 0에서 시작*  }  p->cpu\_burst = 0;  p->cpu\_usage = 0;  p->cpu\_wait = 0;  p->io\_wait\_time = 0;  p->end\_time = *-*1;  p->is\_q\_timeout = 0;  p->aging\_time = 0;  // 과제에서 주어진 값들 초기화 완료  // *프로세스 생성 시 출력 추가*  *if* (p->pid >= 4) {  #*ifdef* DEBUG  cprintf("*PID: %d created\n*", p->pid);  #*endif*  }  … // 기존 코드와 동일  } |

**- 3. 새로운 SSU Scheduler 테스트 프로그램**

**- (1) 스케줄링 테스트를 위해 프로세스의 초기 q\_level, cpu\_burst, cpu\_wait\_time, io\_wait\_time, end\_time 값을 설정할 수 있는 set\_proc\_info() 시스템 콜 추가**

✓ set\_proc\_info(int q\_level, int cpu\_burst, int cpu\_wait\_time, int io\_wait\_time, int end\_time);

|  |
| --- |
| proc.c에 추가 |
| *int* set\_proc\_info(*int* *q\_level*, *int* *cpu\_burst*, *int* *cpu\_wait\_time*, *int* *io\_wait\_time*, *int* *end\_time*) {  *struct* proc *\**curproc = myproc();// *현재 프로세스 가져오기*  curproc->q\_level = q\_level;  curproc->cpu\_burst = cpu\_burst;  curproc->cpu\_wait = cpu\_wait\_time;  curproc->io\_wait\_time = io\_wait\_time;  curproc->end\_time = end\_time;  #*ifdef* DEBUG  cprintf("*Set process %d's info complete\n*", curproc->pid);  #*endif*  *return* 0;  } |

|  |
| --- |
| sysproc.c에 추가 |
| *int* sys\_set\_proc\_info(*void*) {  *int* q\_level, cpu\_burst, cpu\_wait\_time, io\_wait\_time, end\_time;  *if* (argint(0, *&*q\_level) < 0 || argint(1, *&*cpu\_burst) < 0 ||  argint(2, *&*cpu\_wait\_time) < 0 || argint(3, *&*io\_wait\_time) < 0 ||  argint(4, *&*end\_time) < 0) {  *return* *-*1;  }  // *proc.c에 정의된 set\_proc\_info 호출*  set\_proc\_info(q\_level, cpu\_burst, cpu\_wait\_time, io\_wait\_time, end\_time);  *return* 0;  } |

|  |
| --- |
| proc.h에 추가 |
| *int* set\_proc\_info(*int q\_level*, *int cpu\_burst*, *int cpu\_wait\_time*, *int io\_wait\_time*, *int end\_time*); |

|  |
| --- |
| usys.S에 추가 |
| SYSCALL(set\_proc\_info) |

|  |
| --- |
| user.h에 추가 |
| *int* set\_proc\_info(*int q\_level*, *int cpu\_burst*, *int cpu\_wait*, *int io\_wait*, *int end\_time*); |

|  |
| --- |
| syscall.h에 추가 |
| #*define* SYS\_set\_proc\_info 22 |

|  |
| --- |
| syscall.c에 추가 |
| [SYS\_set\_proc\_info] sys\_set\_proc\_info, |
| *extern* *int* sys\_set\_proc\_info(*void*); |

✓ 해당 테스트 프로그램은 생성한 프로세스가 모두 종료된 후 종료되도록 구현, 디버깅 정보와 함께 테스트

- 각 테스트 프로그램의 코드와 실행 예시를 첨부하였다.

|  |
| --- |
| test1-1.c |
| A screen shot of a computer program  Description automatically generated |

|  |
| --- |
| test1-2.c |
| A computer screen shot of a program code  Description automatically generated |

|  |
| --- |
| test1-3.c (코드가 길어서 글로 첨부) |
| #*include* "*types.h*"  #*include* "*stat.h*"  #*include* "*user.h*"  #*define* DEBUG 1  *void* run\_child\_process(*int* *end\_time*) {  // *각 프로세스의 종료 시간 설정*  set\_proc\_info(2, 0, 0, 0, end\_time);  // *프로세스가 끝날 때까지 무한 루프 실행*  *while* (1) {  ;// *CPU 사용 부하를 유지하며 스케줄러에 의해 스케줄링되도록 함*  }  }  *int* main() {  printf(1, "*start scheduler\_test\n*");  *int* pid1 = fork();  *if* (pid1 == 0) {  run\_child\_process(300);  }  *int* pid2 = fork();  *if* (pid2 == 0) {  run\_child\_process(300);  }  *int* pid3 = fork();  *if* (pid3 == 0) {  run\_child\_process(300);  }  // *부모 프로세스가 모든 자식 프로세스가 종료될 때까지 대기*  *for* (*int* i = 0; i < 3; i++) {  wait();  }  printf(1, "*end of scheduler\_test\n*");  exit();  } |

|  |
| --- |
| test1-1, set\_proc\_info(0, 0, 0, 0, 500) |
| A screenshot of a computer program  Description automatically generated |

|  |
| --- |
| test1-2, set\_proc\_info(1, 0, 0, 0, 500) |
| A screenshot of a computer program  Description automatically generated |

|  |
| --- |
| test1-3, 첫 번쨰: set\_proc\_info(2, 0, 0, 0, 300), 두 번쨰: set\_proc\_info(2, 0, 0, 0, 300), 세 번쨰: set\_proc\_info(2, 0, 0, 0, 300), |
| A screenshot of a computer program  Description automatically generated |

+ 추가로 넣은 사항: 출력값은 예제와 미묘하게 다르나 출력만 다를 뿐, Aging 알고리즘은 구현에는 성공했다. 아래는 내 출력 코드에 대한 설명이다.

|  |
| --- |
| 출력 코드 설명 |
| PID: 4 created  PID: 5 created  PID: 6 created  Set process 6's info complete  PID: 6 uses 40 ticks in mlfq[2], total(40/300)  Set process 5's info complete  PID: 5 uses 40 ticks in mlfq[2], total(40/300)  Set process 4's info complete  PID: 4 uses 40 ticks in mlfq[2], total(40/300) 4의 큐 레벨 3으로 강등  PID: 4 uses 80 ticks in mlfq[3], total(120/300)  PID: 4 uses 80 ticks in mlfq[3], total(200/300)  PID: 6 Aging  PID: 5 Aging  PID: 4 uses 80 ticks in mlfq[3], total(280/300)  PID: 6 uses 40 ticks in mlfq[2], total(80/300) 6의 큐 레벨 3으로 강등  PID: 5 uses 40 ticks in mlfq[2], total(80/300) 5의 큐 레벨 3으로 강등 (6과 큐 레벨 같아짐)  PID: 5 uses 80 ticks in mlfq[3], total(160/300) 따라서 같은 레벨의 마지막 작업 5가 실행됨  PID: 5 uses 80 ticks in mlfq[3], total(240/300)  PID: 4 Aging  PID: 6 Aging  PID: 5 uses 60 ticks in mlfq[3], total(300/300)  PID: 5, used 300 ticks. terminated  PID: 6 uses 40 ticks in mlfq[2], total(120/300) 6의 큐 레벨 3으로 강등  PID: 4 uses 20 ticks in mlfq[2], total(300/300) 4의 큐 레벨 3으로 강등 (6과 큐 레벨 같아짐)  PID: 4, used 300 ticks. terminated  PID: 6 uses 80 ticks in mlfq[3], total(200/300)  PID: 6 uses 80 ticks in mlfq[3], total(280/300)  PID: 6 uses 20 ticks in mlfq[3], total(300/300)  PID: 6, used 300 ticks. terminated  end of scheduler\_test |

**- (2) 다음에 실행될 프로세스 선정 과정 스케줄링 과정 출력 구현**

✓ (예시 2)에서 보여주듯이, xv6 빌드 시 "debug=1" 매개변수 전달을 통해 프로세스가 생성된 시점, set\_proc\_info() 시스템 콜이 호출된 시점, CPU를 사용 중인 프로세스가 자신의 타임 슬라이스 만큼 CPU를 사용한 이후 시점, 프로세스가 CPU를 전부 사용하고 종료하는 시점에 프로세스의 정보를 출력했다.

- 출력 형식은 과제 명세에 따르도록 했다.

|  |  |
| --- | --- |
| Makefile에 디버깅 가능하도록 추가 | proc.c와 trap.c 상단에 추가 |
| A computer code with colorful text  Description automatically generated with medium confidence | A blue and purple text  Description automatically generated  -디버그 모드 활성화를 위해 |

-필요한 파일마다 위의 메시지를 추가하였고, 함수 안에 디버깅 메시지는 아래와 같이 넣었다.

|  |
| --- |
| set\_proc\_info 함수 안에 추가 |
|  |
| allocproc 함수 안에 추가 |
| A computer screen shot of a bug  Description automatically generated |
| trap 함수 안에 추가 (언제 추가했는지 대략적으로 알 수 있게 if문 시작과 디버깅 메시지 전 코드를 포함) |
| A computer code with colorful text  Description automatically generated |
| A computer code with colorful text  Description automatically generated |
| A blue background with yellow text  Description automatically generated |

**- 4. 위 1, 2, 3을 기반으로 기존 xv6 스케줄러와 SSU 스케줄러의 기능 및 성능 비교 분석**

✓ 기능 분석(도표 포함)

A screenshot of a white table with black text

Description automatically generated

- 어떤 기능이 구현되어 있는지 표로 나타내어 보았다. 둘을 최종 분석해보자면 다음과 같다.

* 기존 xv6 스케줄러: Round Robin 방식을 사용하여 모든 프로세스를 동일한 타임 슬라이스로 공정하게 실행한다. 이 방식은 모든 프로세스에 대해 동일한 우선순위를 적용하며, 프로세스 특성에 따른 차별화를 두지 않고 처리한다. 따라서 간단하고 공평하다는 장점이 있지만, CPU 사용량이 많은 프로세스와 I/O 중심 프로세스를 동일하게 처리하므로 응답성 측면에서 한계가 있다.
* SSU 스케줄러: MLFQ 방식을 사용해 프로세스의 특성에 따라 스케줄링을 최적화한다. 여러 개의 우선순위 큐를 사용하여 높은 우선순위 큐에서 짧은 타임 슬라이스를, 낮은 우선순위 큐에서 긴 타임 슬라이스를 설정함으로써, 인터랙티브 프로세스는 신속하게 처리하고 CPU-bound 프로세스는 차후에 처리하는 방식으로 반응성을 높인다. 또한, Aging 메커니즘을 통해 낮은 우선순위 큐에 오랫동안 머무는 프로세스의 우선순위를 동적으로 상승시켜 공정성을 유지한다. 이러한 방식으로 SSU 스케줄러는 시스템 성능과 응답성을 크게 개선할 수 있다.

✓ 성능 분석(그래프 포함)

- 성능 분석을 위해 응답 시간 및 대기 시간 측정을 하는 간단한 test.c 파일을 따로 만들었다.

|  |
| --- |
| test.c |
| // 성능 데이터 수집을 위한 xv6 사용자 프로그램 (응답 시간 및 대기 시간 측정)  #include "types.h"  #include "stat.h"  #include "user.h"  #define NUM\_PROCESSES 10  void workload() {  int i;  for (i = 0; i < 10000000; i++) {  asm volatile("nop"); // 간단한 반복 작업 (CPU 부하 생성)  }  }  int main() {  int pid, start\_time, end\_time, wait\_time;  int response\_times[NUM\_PROCESSES];  int wait\_times[NUM\_PROCESSES];  for (int i = 0; i < NUM\_PROCESSES; i++) {  start\_time = uptime(); // 프로세스 시작 시간 기록  pid = fork();  if (pid < 0) {  printf(1, "Fork failed\n");  exit();  } else if (pid == 0) {  // 자식 프로세스: 작업 수행 후 종료  workload();  exit();  } else {  // 부모 프로세스: 자식 프로세스의 응답 시간 및 대기 시간 측정  wait();  end\_time = uptime();  response\_times[i] = end\_time - start\_time;  wait\_time = end\_time - start\_time;  wait\_times[i] = wait\_time;  }  }  // 결과 출력  printf(1, "Response Times:\n");  for (int i = 0; i < NUM\_PROCESSES; i++) {  printf(1, "Process %d: %d ticks\n", i, response\_times[i]);  }  printf(1, "\nWaiting Times:\n");  for (int i = 0; i < NUM\_PROCESSES; i++) {  printf(1, "Process %d: %d ticks\n", i, wait\_times[i]);  }  exit();  } |

- 실행 결과는 다음과 같이 나왔다.

|  |  |
| --- | --- |
| 기존 xv6 스케줄러 | SSU 스케줄러 |
| A black screen with white text  Description automatically generated | A screen shot of a black background  Description automatically generated |

- 이 값을 바탕으로 그래프를 만들면 다음과 같이 나온다.  
A graph of progress and comparison

Description automatically generated with medium confidence

- 위 그래프에서는 xv6 스케줄러와 SSU 스케줄러의 각각 응답 시간과 대기 시간을 비교한다.

* 응답 시간 (Response Times) 비교
  + xv6 스케줄러에서는 프로세스 0의 응답 시간이 8 ticks로 길고, 다른 프로세스들은 상대적으로 짧은 응답 시간을 보인다.
  + SSU 스케줄러는 대부분의 프로세스가 비슷한 응답 시간(1~2 ticks)을 가진다. 이는 공정하게 프로세스를 스케줄링하는 결과를 보여준다.
* 대기 시간 (Waiting Times) 비교:
  + xv6 스케줄러에서 프로세스 0의 대기 시간이 길며, 다른 프로세스들은 대기 시간이 짧거나 거의 없다.
  + SSU 스케줄러에서는 모든 프로세스가 비슷한 대기 시간을 가지며, 이는 전반적으로 더 균형 잡힌 스케줄링을 보여준다.
* 결론

- xv6 스케줄러: 특정 프로세스(특히 0번)의 응답 및 대기 시간이 길어지는 경향이 있다.

- SSU 스케줄러: 더 균형 잡힌 응답 및 대기 시간을 제공하여 프로세스 간 공정성을 높인다.

**- 기타 참고 사항**

* 스케줄러 분석할 때는 맥 터미널 화면을 캡쳐하였고, 과제는 도커로부터 불러온 VS Code 화면을 따로따로 캡쳐하여 이미지의 차이가 조금 있을 수 있다.