**컴퓨터학부, 20190511, 배준형**

# 기존 스케쥴러 분석 : xv6

|  |
| --- |
| **[전체적으로 xv6가 스케쥴링 되는 과정]**  xv6는 기본적으로 handmade Code로 init 프로세스가 생성된 후 (pid=1) 초기 부팅이 진행된다.  초기에 가장 먼저 실행되는 함수는 pinit() 로 프로세스테이블 락을 초기화한다. 이때 실행되는 코드는 pid가 없다. 이후 실행되는 userinit() 내부에서는 시작 이진코드를 실행한다. 그리고 **initcode.S** 로 들어가서 xv6 **init.c 프로그램 실행 (==shell) 그리고** 초기 프로세스 디렉토리를 / 로 설정한다. (디렉토리 inode 가져옴)  이후 allocproc() 를 호출하여 **1번 프로세스를 생성한다. 1번 프로세스는 forkret() 를 실제적으로 실행시키는 프로세스로, forkret() 에서 fs.c : iint() 를 호출하여 디스크 앞단에 있는 superblock으로부터 inode와 데이터블록의 메타데이터를 받아온다. 이후 저널링(Journaling)** 기법으로 log.c : initlog() 를 호출하여 메타대데이터 및 데이터정보를 이전 로그로부터 갱신받고 로그블록은 초기화한다.  1번프로세스를 생성하자마자 initProc는 스케쥴링 함수를 호출하게된다. scheduler() 그리고 끊임없이 스케쥴링함수의for문을 돌며 스케쥴링을 시행한다.  이후 shell 프로세스 (pid=2) 를 init.c 를 실행시켜 생성한 후 (proc.c : fork()를 통한 호출)  생성된 1,2번 프로세스는 일반적인 상태에선 sleep 상태로 존재하게 된다. 2번 프로세스 sh(쉘) 은 $ 뒤에 사용자 입력이 올 때까지 기다리게 되고 사용자 입력이 들어온다면 proc.c : fork() 를 수행한다.  **[xv6 스케쥴러 메커니즘 (Mechanism)]**  xv6 스케쥴러는 기본적으로 context-switch 를 통해 cpu를 Time Sharing 방식으로 작동하기 위해 생성된 특별한 함수이다. 이 스케쥴러함수는 스케쥴링의 핵심 중 하나인 Context Switch (vm.c : swtch() 를 호출) 를 진행하며return하지 않고 스케쥴러 내부함수에서 usermode 로 , usermode에서 kernelmode로 스케쥴러 내부함수로 복귀하며 끊임없이 for(;;) 루프를 돈다.  xv6스케쥴러가 switch하는 방식은 굉장히 단순하다.  스케쥴러가 프로세스를 하나 뽑으면 해당 프로세스의 상태를 Running을 변경한다.  이후 switchuvm(p) 를 통해 해당 프로세스의 페이지테이블을 가져온 뒤  swtch(스케쥴러, 프로세스); 를 통해 스케쥴러에서 프로세스로 Context Switch를 진행한다.  그러면 cpu는 Time Quantum 동안 사용자프로그램(User program) 을 쭉 실행하게된다.  그러다가 프로세스가 스케쥴이 모종의 이유(Interrupt(비동기) or system call(동기,명시적)) 로 호출되게되면 trap.c : trap(struct trapframe\* tf) 함수로 trap을 처리하게 된다.  예를 들어서 Time Quantum 이 1tick 이 지나서 Timer Interrupt 가 발생한다면trap (Timer Interrupt 등) 으로 proc.c : yield() 호출되어 swtch(이전프로세스, 스케쥴러) 로 Context Switch 되어proc.c scheduler() 함수로 다시 되돌아오게된다.  yield()가 호출되게되면 해당 프로세스상태는 RUNNING에서 RUNABLE으로 변경되게 된다.  그리고 내부에서 sched() 가 호출되어 **swtch(프로세스, 스케쥴러)**로 다시 스케쥴러 함수로 복귀하게되고  복귀하자마자 즉시 switchkvm()를 호출하여 Kernel(OS) 의 가상페이지에 대한 페이지테이블을 다시 적재한 뒤  다음에 실행될 프로세스를 스케쥴링하게 된다.  추가적으로, xv6 스케쥴러 함수가 호출이 될 때는 기본적으로 sched() 라는 함수가 호출이 된다.  이 sched() 함수는 exit(), yield(), sleep() 내부에서 호출됨을 알 수 있다.  여기서 **exit(), sleep() 공통점은 현재 자신의 프로세스는 사용하지 않기 때문에 호출되고 yield는 timer Interrupt 가 발생하여 호출된다는 점이 특징이다.**  즉, xv6 던 현대 OS던 cpu 1개를 tick 단위로 **시분할 (TSS : Time Sharing System)** 방식으로 스케쥴링을 하고 있다.  여기까지가 xv6의 스케쥴링 메커니즘에 대한 설명이었고 **이는 뒤에 구현할 SSU Scheduler에서 사용되는**  **메커니즘과 거의 동일한 점을 볼 수 있다.**  **[xv6 스케쥴러 정책 (Policy)]**  다음으로, **xv6 스케쥴러의 정책(Policy) 에대해서 설명하자면** xv6의 스케쥴링 방식은 **RR (Round Robin**) 과 거의 흡사한 모습을 보여주고 있다. **xv6에서는 ptable 내부의 64개의 proc\* 구조체가 RUNNABLE인지 확인하여 원형 큐 처럼 계속계속 도는 구조로 되어있다.** (spin 형태로 ptable에 존재하는 proc 배열(Vector) 에서 RUNNABLE 상태인 프로세스를 뽑는다. ) 물론 완전한 Queue는 아니라서 ptable 사이의 중간이 비게된다면 새로운 프로세스가 맨 뒤가 아닌 중간에 삽입될 여지가 있지만, 일반적으로는 xv6 스케쥴러는 뒤에온 프로세스는 뒤에 삽입되도록 설계되어있기 때문에 전반적으로 RR 과 성능이 비슷한 점을 볼 수 있다.  즉, xv6 기본 스케쥴링은 1 Tick 마다 일종의 Round Robin 비슷한 형태로 스케쥴링이 진행된다는 말이다.  그렇기에 xv6의 Response Time은 좋은편이지만, 평균 TurnAround Time은 매우 형편이 없음을 뒤에서 설명할 예정이다.  **[xv6 스케쥴러 구현]**  **xv6 scheduler() 내부에서 사용하는 메인 함수들은 아래와 같다.**  **acquire() / release(), || sti(), || swtch(), swtchuvm(), swtchkvm()**  || 를 전후로 관련있는 내용을 묶어 설명하겠다.  우선 acquire과 release 는 가상 cpu의 0,1 번 코어를 사용하는 xv6 는 병행성 문제를 해결하기 위해 spinlock 형태로 while() 문들 돌며 lock을 해제할 때 까지 기다려주는 락을 사용한다. 프로세스를 뽑는 과정 자체 역시 **원자적연산(atomic)연산 즉 스케쥴링 과정에서 Interrupt 로 인하여 중단되고 다시 돌아올 때 스케쥴링을** 보장해주어야하므로 lock을 획득하고 스케쥴링을 진행한 후 다시 lock을 해제하는 방식으로 스케쥴링을 진행한다. 이러한 이유로 스케쥴러가 for문 시작부분에서 **acquire(&ptable.proc)** 을 한 뒤 **release(&ptable.proc)** 으로 락을 푸는 모습을 볼 수 있다.  다음으로 sti() 는 **interrupt를 활성화 시켜주는 함수**로 **apic로부터 cpu로 Timer Interrupt**를 받을 수 있도록 하기위함이다. sti() 는 보통 schedule() 함수가 처음 호출되면 가장 먼저 sti(); 를 설정한다.  두번째로는 **Context-Switch** 를 지원하기 위해 특별한 함수를 호출한다. switchuvm(p) 과 swtch(스케쥴러, p->context), switchkvm() 를 내부적으로 호출함을 알 수 있다.  수업시간에 말한 **Dual-Mode**를 지원해주기 위해 switch를 하는 프로세스와 **실제로 context를 교환 및 지원을 해주는 함수들이다.**  **switchuvm(p)** 는 스케쥴링된 프로세스의 **페이지테이블을 가져오는 역할**을 한다..  **switchkvm()** 는 sched() 로 호출받아 스케쥴러 함수로 다시 복귀했을 때 커널의 가상메모리에 대한 페이지테이블을 받아오는 함수이다.  **swtch()** 부분은 실제로 Context 즉, 처리기(CPU) 가 뽑힌 프로세스의 Register, Memory, System Context를 교환하는 함수로, 실제로 이 때, 스케쥴링이 끝나고 프로세스가 실행되게 된다. swtch() 함수는 C언어로 구현된 것이 아닌 **swtch.S 라는** 어셈블리어로 구현이 되어 레지스터를 교환해준다. |

# [SSU Scheduler V.S. xv6 Original Scheduler (1 Tick Round Robin) 성능분석

|  |  |  |  |
| --- | --- | --- | --- |
| xv6 기존 스케쥴러는 공정성 (fairness) 가 좋은 스케쥴러로 정확하게는 Round Robin이라고 하기는 애매하지만 스케쥴링 시ptable 구조체를 초기부터 계속 무한히 돈다는 점에서 RR과 성능이 비슷한 스케쥴링 기법을 사용한다.  반면, SSU Scheduler 의 경우 UNIX 초기 스케쥴러와 비슷하게 cpu decay(CPU 사용량) 에 따라 우선순위가 변경되며 I/O 위주의 작업이 CPU 위주의 작업보다 우선순위가 높게 설정되어있는 스케쥴링 기법이다.  이로인해서 다수의 I/O위주의 작업 (Interactive Job) 이 들어와서 우선순위가 낮은 CPU 위주의 작업 (Batch Job) 이 스케쥴링이 안될 수 있는 기아 문제 (Starvation Problem) 이 발생할 수 있다.  이러한 점을 바탕으로 스케쥴링 성능을 분석할 수 있도록 사용자 프로그램에서 시작, 끝 Ticks을 잴 수 있도록 시스템콜 myticks() 를 하나 추가하였다. 그리고 스케쥴링 성능을 분석해보기 위해 원본 xv6와 ssu\_scheduler의 PNUM 개수를 바꿔가며 myticks() 차이를 비교해볼 예정이다.  이 점을 바탕으로 xv6 기본 스케쥴러도 몇 가지 커스터마이징을 통해 ssu\_scheduler 에서 만든 scheduler\_test 명령 (ticks수를 정해주고 해당 tick 이후 프로세스가 죽게해주는 시스템콜) 을 구현하였다.  아래 출력되는 tick은 ( (1) : Time Quantum 이 끝나 yield() 호출 직전, (2) 스케쥴링 되는 시점 (3) 프로세스가 죽기 직전 으로 호출을 시도해보았다.  본격적인 스케쥴링 기법을 수행하기 전 QEMU 에뮬레이터로 출력되는 결과의 수만개의 Tick을 일일이 세는 점은 불가능하다고 판단되어 **간단한 C프로그램** **(extractTicks.c)**을 작성하여 QEMU xv6 결과를 test.txt로 리다이렉션을 통해 csv파일로 (Process ID, PNUM, 프로그램이 실행되는 TICK수, Schedule 되도록 설정된 시간, TurnAround Time, Response Time 계산 , 스케쥴링이 호출된 횟수, 스케쥴링이 시작된 시점, 프로세스가 들어온 시점, 프로세스가 종료되는 시점 Tick 등을 자동으로 추출해주도록 설계한 후 해당 프로그램을 이용하여 기존스케쥴러와 SSU 스케쥴러 차이를 비교해볼 예정이다.    예상했던 결과와 맞게 1Tick 마다 스케쥴링이 되는 모습을 보여주며 6->5->4 순서로 반복되게 호출됨을 볼 수 있다. (Round Robin과 비슷한 성질의 스케쥴링)  <**Tick 검출 프로그램 extractTick.c** : 출력되는 Tick을 I/O Redriection 을 받아와 입력받은 데이터를 읽어옴.>   * 다순 Pipe로 구현하려다가, 결과를 직접 보고 판단하고 싶어 test.txt로 결과를 받아온 값을 extractTick.c 프로그램이 재분석 (중요 부분만 수록 – 맨 뒤에 소스코드 수록하였습니다.)  |  | | --- | | ...  system("make clean");  system("make analy=1 qemu > test.txt")  ...  q\* findTotalTicks(FILE\* fp) //Tick 검출 코드  {  q\* head = NULL;  q\* tail = NULL;  char tempbuf[LINEBUF] = {0,};  int curPNUM = -1;  //Dummy 노드 생성  head = (q\*)malloc(sizeof(q));  head->line = 0;  head->next = NULL;  tail = head;  while (!feof(fp)) {  fgets(tempbuf, 1024, fp);  if (strstr(tempbuf, START\_SCHED)) { //스케쥴러 시작지점 검출  sscanf(tempbuf, "scheduler\_test start[PNUM:%d]", &curPNUM);  }  else if (strstr(tempbuf, INIT\_PID)) { //PID가 생성되었을 때 Tick 검출  int pid, ticks;  sscanf(tempbuf, "PID : %d, %d (0)", &pid, &ticks);  pidInfo[pid].create\_ticks = ticks;    }  else if (strstr(tempbuf, FIND\_PID)) { //스케쥴러 호출마다 PID, Tick 검출  int pid, total\_ticks = 0;  int tmp1, tmp2, tmp3;  sscanf(tempbuf, "PID : %d, priority : %d, proc\_tick : %d ticks, total\_cpu\_usage : %d ticks, totalTicks : %d (2)",  &pid, &tmp1, &tmp2, &tmp3, &total\_ticks);    if (pidInfo[pid].pid == 0) { //pid가 처음으로 호출되었으니 Response time을 위해 sched timer검출  pidInfo[pid].pnum\_id = curPNUM;  pidInfo[pid].pid = pid;  pidInfo[pid].start\_ticks = total\_ticks;  pidInfo[pid].CallCount = 1;  }  else {  pidInfo[pid].end\_ticks = total\_ticks; //계속 갱신하면 마지막에 갱신된 값이 마지막 스케쥴링 tick  pidInfo[pid].CallCount++;  }  }  else if (strstr(tempbuf, DIED\_PID)) { //해당 프로세슷 스케쥴링 끝났을 때, T-Time, R-TIme 갱신  int pid, ticks;  int tmp1, tmp2, tmp3;  sscanf(tempbuf, "PID : %d, priority : %d, proc\_tick : %d ticks, total\_cpu\_usage : %d ticks, totalTicks : %d (3)",  &pid, &tmp1, &tmp2, &tmp3, &ticks);  pidInfo[pid].end\_ticks = ticks;  if (ticks != 0) {  pidInfo[pid].turnAroundTime = pidInfo[pid].end\_ticks - pidInfo[pid].create\_ticks;  pidInfo[pid].responseTime = pidInfo[pid].start\_ticks - pidInfo[pid].create\_ticks;  }    }  else if (strstr(tempbuf, END\_SCHED)) { //스케쥴링이 끝난 시점에서 해당 PNUM에 대한 tick 저장  int sched\_end\_ticks;  sscanf(tempbuf, "end of scheduler\_test[PNUM>%d] : %d ticks", &curPNUM, &sched\_end\_ticks);  q\* newNode = (q\*)malloc(sizeof(q));  sprintf(newNode->lineBuf, "%d,%d", curPNUM, sched\_end\_ticks);  newNode->line = tail->line+1; //라인수 정해줌  newNode->commaCount = 1;  newNode->next = NULL;  //연결리스트 재연결  tail->next = newNode;  tail = newNode;  }  else if (strstr(tempbuf, PID\_TIMER)) { //set\_sche\_info() 시스템 콜 시 우선순위와 timer 저장  int pid, s\_ticks, priority;  sscanf(tempbuf, "[SET] pid : %d, priority : %d, schedule\_ticks : %d", &pid, &priority, &s\_ticks);  pidInfo[pid].priority = priority;  pidInfo[pid].sched\_time = s\_ticks;  //[SET] pid : 4, schedule\_ticks : 591  }  if (feof(fp))  break;  }  return head->next;  } |   **[A. xv6 스케쥴러와 SSU 스케쥴러의 기능 차이 분석]**  **-** xv6스케쥴러와 ssu 스케쥴러 기능의 가장 큰 차이는 **우선순위**에 따라 프로세스 호출순서가 바뀐다는 점이다. 이 점을 확인해보기 위해 scheduler\_test.c 사용자 프로그램의 아래 함수를 제작하여 실험해보았다.   |  | | --- | | /\*\*  \* CPU 위주 작업, I/O (sleep) 위주 작업 구성 및 우선순위가 다른 프로세스의 실행 방식  \*/  void scheduler\_testing\_3(int cpu\_process, int io\_process) {  uint start, end;  int pnum = cpu\_process + io\_process;  int pid, j, ticks;  start = myticks();  //cpu 위주 작업에 대한 세팅 (400Ticks로 기본)  printf(1, "scheduler\_test start[PNUM:%d]\n", pnum);  for (j = 0 ; j < cpu\_process ; j++) {  pid = fork();  if (pid == 0) {  set\_sche\_info(80, 400);  while(1);  }  }  //IO 위주 작업에 대한 세팅  for (j = 0 ; j < io\_process ; j++) {  pid = fork();  if (pid == 0) {  set\_sche\_info(2, 400);  ticks = myticks();  while(1) {  //무한 sleep으로 1 tick 도 지나지 않아 스케쥴링이 되지않는 점 방지를 위한 코드  if (ticks + 1 > myticks()) {  ticks = myticks();  sleep(5);  }  }  }  }  for (j = 0 ; j < cpu\_process + io\_process ; j++)  wait();  end = myticks();  printf(1, "end of scheduler\_test[PNUM>%d] : %d ticks\n",pnum, end-start);  } |  * 해당 코드를 scheduler\_testing\_3(5,15) 로 호출하여, CPU위주 작업 5개, sleep 위주 작업 15개에 대하여   우선순위가 I/O 위주 작업이 높으면서 다수의 프로세스를 차지하는 경우 **SSU 스케쥴러는 기아문제 (Starvation) 이 발생**할 수 있음을 보여주는 결과이기도 하다.   * R-Time : Response Time, T-Time : TurnAround Time을 축약한 것.     **[CPU위주작업 5개(우선순위80), sleep 위주 작업 15개 (우선순위2) 로 Tick을 측정해본 결과]**  결과를 잠깐 분석해보면 xv6스케쥴러와 SSU 스케쥴러 기능의 차이가 확연하게 볼 수 있다.  PID의 번호는 PID에 따른 생성 순서를 알려준다. xv6스케쥴러의 경우 RR방식과 유사하기 때문에 프로세스가 생성되자마자 빠른 시간내에 최소한 한 번 스케쥴링이 됨을 도표로부터알 수 있다. 하지만, SSU 스케쥴러는 시스템콜을 구현하자마자 해당 우선순위가 적용되는 것이 아닌 **우선순위 재갱신 과정에서 우선순위 갱신이 발생한다.** 그렇기 때문에 초기에 생성된 프로세스가 대략적으로30ticks 를 사용한 후 스케쥴링이 발생하는 점을 볼 수 있다. **그런데 sleep()을 사용하면 프로세스가 짧은 시간동안 수면상태 (Block)에 들어갔다가 해제되기 때문에 30ticks을 그대로 기다리지 않고 바로바로 스케쥴링이 되는 모습을 도표로부터 알 수 있다.** 그런 점을 **통해 xv6가 SSU 스케쥴러보다 더 공평한 프로세스**임을 알 수 있다.  또한 xv6와 SSU 스케쥴러의 가장 큰 차이인 **우선순위마다 호출 순서가 다르기 때문에** TurnAround Time의 결과가 xv6는 대체적으로 유사한 TurnAround를 가지는 반면, xv6 스케쥴러는 우선순위가2, 80으로 극단적으로 차이나는 스케쥴러이므로 9~23 (sleep 다수사용) 의 프로세스가 모두 종료한 후 그때부터 스케쥴링이 시작되어 1000Ticks (4~8) 과 (9~23) 프로세스들 집합군체에서 대략 2000Ticks 이나 차이가 나는 점을 볼 수가 있다. **이러한 특성으로 RR과 유사한 xv6의 TurnAround Time 성능은 매우매우 떨어지는 반면 xv6의 TurnAround의 성능은 xv6보다 1500Ticks이나 차이가나는 굉장한 성능 차이를 보여주고 있다.**  이번 실험에서는 I/O 위주 작업 대신 sleep 위주작업이라고 하였으나, 실제로 I/O위주 작업이 진행되면 sleep 상태로 들어가기 때문에 상황적으로는 유사하게 연출하였다. ssu scheduler 는 sleep상태에서 깨어날 때 RunQueue 에 있는 우선순위 중 가장 작은 값으로 재할당받기 I/O 위주작업도 sleep() 과 비슷한 결과를 가질 것이다.  이를 통해 xv6와 다르게 SSU 스케쥴러는 우선순위만 잘 조절한다면 **TurnAround Time이** xv6보다 확연히 좋아짐을 알 수 있다. 반면, I/O 위주 작업 (Sleep 위주작업) 이 다수가 존재한다면 CPU 위주 작업이 굶어죽는 이른바 **기아상태(Starvatio)**가 발생할 수 있다는 점도 또한 볼 수 있는 실험이었다.  **[B. xv6스케쥴러와 SSU 스케쥴러의 성능 차이 분석]**  xv6 스케쥴러와 SSU 스케쥴러의 성능 차이를 보기위해서 PNUM을 3 ~ 19까지 증가시켜가며 위의 프로그램으로 PCB정보를 추적해보았다. 실험코드는 아래와 같다.   |  | | --- | | void scheduler\_func\_V2(int pnum)  {  printf(1, "scheduler\_test start[PNUM:%d]\n", pnum);  int pid;  int i;  uint start, end;  start = myticks();  for (i = 0; i < pnum; i++) {  pid = fork();  if (pid == 0) {  set\_sche\_info(i+6\*i/2, (i+1)\*40);  while(1);  exit(); }  }  for (i = 0; i < pnum; i++) {  wait();  }  end = myticks(); //끝 틱 특정.  printf(1, "end of scheduler\_test[PNUM>%d] : %d ticks\n",pnum, end-start);  }  void scheduler\_testing(int N) {  int i;  for (i = 3 ; i < N ; i++) { // PNUM을 3부터 19까지 순차적으로 증가 실험.  printf(1, "\n\n\n========================================\n");  scheduler\_func\_V2(i);  printf(1, "========================================\n");  }  } |     [PNUM을 달리하였을 때 측정되는 전체 Tick와 R,T-Time 성능 차이]  <전체 Ticks 에대한 성능 차이>  실제로 위의 예제 프로그램으로 성능차이를 분석해본 결과 전체 Ticks 은 xv6나 SSU 스케쥴러나 비슷비슷함을 알 수 있었다. 하지만SSU\_scheduler 성능이 기존 xv6 스케쥴러보다 근소하지만 조금 더 좋은 성능을 가짐을 알 수 있었다.  이는 이론적으로 스케쥴링 시간은 매우 trivial 하여 무시되지만, 수만번 스케쥴링이 호출된다면 최소한 몇 Tick 정도의 시간차이가 발생한다고 생각된다. 그런점에서 기존스케쥴러가 SSU 스케쥴러보다 우위를 가지는 것은 스케쥴링 과정이 매우 단순 (ptable 의 프로세스를 단순히 spin하여 RUNNABLE 프로세스를 찾는 방식)하여 스케쥴링 자체 시간에 대한 Overhead가 매우 작다는 점이다. 반면 SSU 스케쥴러가 기존스케쥴러보다 꽤 복잡하게 다수의 Queue가 존재하며, 해당 Queue로 진입하는 시간 등을 고려한다면 스케쥴러 내부의 스케쥴링 과정에서만 Overhead가 꽤 발생할 수 있는 요소가 있다.  그럼에도 기존 스케쥴러보다 SSU 스케쥴러가 근소하게나마 Ticks가 빠를 수 있었던 이유는 스케쥴링 횟수와 관련이 있다고 생각된다. xv6스케쥴러는 1Tick마다 스케쥴링이 발생하지만, SSU 스케쥴러는 30Tick 마다 스케쥴링이 발생하여 스케쥴링자체가 짧은 스케쥴러를 30배 곱한 것과 스케쥴링 자체의 overhead가 조금 있는 SSU 스케쥴러가 전체Tick에 대한 속도가 더 빠른것이 아닌가 생각이된다.  단 SSU 스케쥴러와 xv6의 TurnAround Time이 전반적으로 비슷한 이유는 priority 가 일반적으로 4씩 차이나도록 설계해놨기 때문에 거의 RR처럼 동작하기 때문으로 생각이된다.  이와 반대로 xv6 스케쥴러의 Response time 은 fairness 한 편인데, 이와 반대로 **SSU 스케쥴러는 프로세스 생성순서대로 Responsetime이 30씩 증가하게 된다.** 그 이유는 예시프로그램에선 sleep() 등과 같이 프로세스 스케쥴링이 30ticks 전에 preemption 할 방법이 없어 앞서 생성된 우선순위가 높은 프로세스가 계속해서 30TIcks씩 사용된 이후 자신의 첫 스케쥴링 시간이 돌아오기 때문으로 Response time이 크게 증가하는 불공정함 측면을 보여주고 있음을 볼 수 있다.  따라, xv6에 비해 SSU 스케쥴러가 성능자체는 준수하다면 준수하다고 할 수 있지만, 기아문제, 공정성에선 불리하게 작용할 수 있다. |

[**SSU Scheduler 구현내용]**

# 과제 개요

|  |
| --- |
| 기존 스케쥴러에서는 ptable 을 순회하며 RUNNABLE 프로세스를 뽑는 RR 방식의 스케쥴러이다. Response Time 자체는 준수하여 공정성은 어느정도 TurnArund Time 를 보장하기 힘들다는 단점이 있다. 이러한 점을 반영하여 SSU Scheduler 는 프로세스마다 Priority를 가지고 있고, 해당 priority는 60ticks마다 재갱신된다.  또한 25개의 RunQueue 를 통해 Queue 내부에서 priority가 작은 순서의Queue 번호부터 스케쥴링이 진행됨을 알 수 있다. 해당 스케쥴러 방식은 일종의 우선순위 Queue 방식이며 Boosting이 없기 때문에 스케쥴러 기아상태가 발생할 수 있다는 단점이 존재한다.  **하지만, RR방식에서 발생하는 낮은 성능의 Turn Around를 보완할 수 있다는 점에서 기존 스케쥴러보다 우수한 성능을 보여주기도 한다. 특히 현재 중요한 프로세스의 우선순위를 임의로 올려줄 수 있는 기법 (LINUX 에서는 nice) 을 도입하고있기에 프로세스 관리에 용이하계 설계되어있다는 점이 흥미로운 점으로 꼽힌다.** |

# 상세설계

|  |
| --- |
| proc.h에서 추가한 내용  struct cpu {  ....  uint scheduler\_flag; //P3 스케쥴러 갱신여부 플래그 -> 갱신해야될 때 1, 아닐 때 0  };  strcut proc {  ....  //P3 추가 멤버  int priority; //우선순위 그자체  uint priority\_tick; //우선순위 재조정 되기 전 까지 CPU 사용시간  uint proc\_tick; //스케쥴링 될 때마다 측정하는 Ticks  uint cpu\_used; //CPU 총 사용시간  uint proc\_deadline; //프로세스 데드라인 (set\_sche\_prio 시스템콜에서 사용)  struct proc\* next; //RunQueue에서 연결할 포인터  struct proc\* prev; //RunQueue 에서 연결할 포인터  };   * SSU 스케쥴러의 Queue에서 직접적으로 동작하는 노드 역할은 **PCB가 직접한다.** 그렇기에 PCB 내부에 연결리스트 next, prev 가 구성되어있어 실제로 **동적할당을 하는 부분은 없다. (이미 PCB가 할당되어있기 때문 (내부의 ptable.proc)** . 그래서 프로세스가 unused 가 되었을 때 연결리스트 부분이 삭제될 수 있으므로 kill() 등에서 삽입하지 않도록 주의해서 설계하였습니다. * SSU 스케쥴러 기본 동작원리   **[공통 구현]**  <Makefile>   1. SSU Scheduler는 기본적으로 cpu를 1개만 사용하기 때문에 Makefile 내부 CPUS 값을 1로 조정한다.   ... Makefile 중  ifndef CPUS  CPUS := 1 #명세대로 내용 수정  endif  ...  # 예시 1-3 등에서 사용될 더 정확한 예시 ticks 분석을 위해 추가  ifeq ($(debugs), 1)  CFLAGS += -DDEBUGS  endif  #실제 제출 시 debug=1 일 때 tick 을 출력해주기 위해 추가된 부분  ifeq ($(debug), 1)  CFLAGS += -DDEBUG  endif  # tick 분석을 할 때 extractTick.c 에서 내용을 분석할 때 사용하는 매크로  ifeq ($(analy), 1)  CFLAGS += -DANALY  endif  # 기본형(A형 스케쥴러) 에서 응용형 (4 Queues in 1 RunQueue idx) 로 전환할 때 사용하는 매크로  ifeq ($(news), 1)  CFLAGS += -DNEWS  endif  **[스케쥴링]**   * 프로세스가 생성된 후 RunQueue에 삽입/삭제할 인터페이스를 제공합니다. -> appendProc() * RunQueue에 삽입되고 스케쥴링이 될 때 getHighPri() 를 호출하여 프로세스를 선택합니다. * 해당 프로세스가 swtch를 통해 되돌아오면 다시 appendProc() 로 해당 프로세스를 Queue에 넣고 getHighPri() 를 호출합니다. * (반복)   **[재갱신]**  우선순위 갱신 방식은 trap에서 프로세스가 유효할 때 TIMER\_INTERRUPT 가 발생할 때마다 trap.c 내부에서 myproc() && myproc()->state == RUNNING && tf->trapno == T\_IRQ0+IRQ\_TIMER 조건일 때 scheduler\_tick (을 증가시켜가며 해당 ticks이 60ticks 이상일 때 scheduler\_tick을 0으로 초기화하고 **mycpu 구조체에 새롭게 추가된 플래그 (scheduler\_flag)를 1로 만들고 yield()를 호출**   * **★ 여기서 yield()는 스케쥴러 양보를 위한 yield가 아닙니다. (스케쥴링 없이 바로 실행되던 프로세스를 재실행하도록 설계하였습니다. 함수 내부구현이 동일하여 새롭게 만들지 않았습니다.)** * yield() 내부에서 ptable.lock을 관리하고 -> sched() 를 통해 스케쥴러 함수로 돌아갈 수 있음. 이후 mycpu()->scheduler\_flag를 확인 후 update() 를 진행완료합니다. update가 끝나자마자 바로 다시 usermode로 돌아갈 수 있도록 내부 설계   **[시스템 콜 구현]**  **> 시스템 콜 set\_sche\_info() 구현**  - 앞 전 시스템 콜 구현과 동일하게 sysproc.c, syscall.c, syscall.h, user.h, usys.S 를 수정하여 시스템 콜 구현  > proc.h 의 PCB 내용 중 p->proc\_deadline 을 사용자로부터 입력받아 trap.c 에서 process ticks을 증가시켜가며 deadline을 넘어가는지 확인후 프로세스 종료 (초기값 -1) <- trap.c에서 -1이면 강제종료 시키지 않도록 설계함.  > 사용자로부터 priority를 입력받아 현 프로세스의 priority를 해당 값으로 변경 -> 60ticks이 지나면 해당 프로세스는 ticks대로 RunQueue에 삽입되게됨.   * **시스템콜 myticks() 구현**   프로세스 스케쥴링 시 ticks를 측정하기 위해 trap.c의 ticks 를 락을 통하여 병행성 문제를 해결한 상태로 user program에게 tick정보를 전달하는 시스템콜  **[사용자 프로그램 scheduler\_tick.c]**  **scheduler() 의 여러가지 상황을 비교하기 위해 구현되어있습니다.**   * **void scheduler\_func();** // PNUM이 3개일 때 조건을 비교해보기 위해 설계된 함수 * **void scheduler\_func\_V2(int pnum);** //pnum을 조절하여 3 ~ pnum만큼 fork() 를 연속적으로 진행하여 프로세스 성능을 비교합니다. * **void scheduler\_testing\_3(int cpu\_process, int io\_process);** //cpu\_process 와 io\_process(sleep) 개수를 조절하여 cpu bound job과 io bound job의 성능을 비교합니다. * **SHORT 매크로**를 이용해서 빠른시간내에 명세 스케쥴링을 테스트할 수 있게 구현되어있습니다.     [A,B 구현방식에 대한 그림, (**현재 스케쥴러는 A방식으로 기본설정**, news=1옵션으로 변경가능)]  **[RunQueue에 대한 2가지 형태로 구현 :**  **RunQueue 인덱스 내부에 큐 1개(기본) v.s 4개 Queue(NEWS)]**   * RunQueue 분석과정에서 RunQueue 구현형태가 2개가 될 수 있어 2개 모두 구현하였습니다.   실제로 제출한 스케쥴러는 A형이고, B형 스케쥴러는 위의 분석과정에서 사용되었습니다.  코드상에 포함되어있어 A,B 구현 모두 설명하겠습니다.   * **기본적으로 A형 (기본형 RunQueue 인덱스 하나당 큐 1개) 프로세스가 실행되며 make옵션이 news=1 일 경우 응용형 (RunQueue 당 4개의 큐)가 실행되도록 설정되어있음.**  1. (기본형-현재 제출) RunQueue 내부에 1개의 Queue가 우선순위가 낮은 프로세스가 head쪽에 오도록 설계. (각 RunQueue의 우선순위가 가장 낮은 프로세스는 tail에 붙어있음)  * **A방식이 현재 제출한 형태로 작성된 코드입니다.**  1. (응용형) RunQueue 내부에 4개의 Queue가 따로 구현되어 있고 해당 Queue들의 프로세스 총합을 관리하는 middleCount를 통해 Queue 접근을 최소화하는 기법   <proc.c : RunQuene 삽입 요소를 제외한 구현>   * proc.c : scheduler() 함수 내부에서 호출되는 함수들   **void updateQueue();** // 60Ticks마다 우선순위 재계산함수   * 현재 내부의 큐 중에서 proc\_tick이나, 현재 큐에 맞지않는 priority를 가진 Queue를 우선순위 계산 (priority = priority + cpu\_times/10) 후 재삽입하는 함수   priority 계산은 60ticks 마다 일어나게 하며 해당 60ticks 는 단순히 trap.c에서 증가시키는 ticks가 아닌 프로세스의 proiroty\_tick (현재 proc\_tick 으로 구현) 의 합이 60이 넘었는가 확인 후 60이 넘었으면 counter를 초기화하고 재갱신 함수 **updateQueue();** 를 호출함.  **proc\* getHighPri();** //RunQueue에서 RUNNING 상태이면서 우선순위 가장 높은 프로세스 선택  **return 현재 RunQueue의 Prority 가 가장작고 (우선순위가 가장 높고) RUNNABLE 상태의 프로세스 리턴.**   * Return에 적힌 설명대로 RunQueue에서 Head부터 탐색을 하며 우선순위가 가장 낮으면서 RUNNALBE 상태의 프로세스가 추출될 수 있도록 설계하였음.   **struct proc\* deleteQueue (procQ\* queue, struct proc\* retProc);** //프로세스를 RunQueue에서 삭제  -> 해당 retProc 프로세스를 queue로부터 빼서 리턴  - queue 내부에서 retProc를 빼기 쉽게하기 위해서 구현되어있습니다.  <wakeup1(), allocproc() 에서 호출되는 함수>  **int getSmalleastPri();**  **return RunQueue에서 가장 작은 우선순위(Priorirty)를 리턴받음**   * 주의할 점은 해당 함수에서 유휴프로세스(99) 는 뽑히지 않도록 설정되어 있음. * 프로세스가 생성, wake 시 0으로 생성하면 기아상태가 너무 잦게 발생하므로 RunQueue 우선순위 중 가장 작은 값으로 설정되도록 함. * wakeup1(), allocproc() {[fork()]} 에서 fork()는 내부에 allocproc() 를 통해 priority를 최솟값으로 설정하므로 priority를 일부러 설정하지 않았습니다.   **[기타사항]**   * RunQueue에서 프로세스가 Schedule 되면 RunQueue에서 delete 되도록 설계하였습니다. 그렇기 때문에, 일부로 프로세스가 종료 나 스케쥴 전에 다시 재삽입을 해주어야 합니다. 그렇기 때문에 userinit(void), fork(), yield(), wakeup(), kill()에서 SLEEP중이던 프로세스를 재삽입 (appendProc()) 하도록 설계하였습니다. * **기본적으로 process 구조체 내부에 prev, next 포인터를 가지고 있습니다. 프로세스는 상태가 unused 가 될 때 0으로 모두 초기화 되니 이 점에 각별히 주의해서 unused 가 될 때는 큐에 재삽입하지 않도록 설계하였습니다. \*보통 ZOMBIE --> 부모가 회수 --> UNUSED 이렇게 흘러가기 때문.** * Runqueue 내부에는 **Prioritiy 라는** Process 리스트를 관리하는 각 큐가 들어있습니다. (아래에서 자세히 후술) 해당 Prcoess 는 A,B 방식 내부 구현은 다르지만, 1개 자체의 큐인지 계층적인 큐인지로 구분되어있습니다. * RunQueue 내부에는 Queue 에 들어있는 프로세스의 수 (count) 가 들어있어 디버깅에 용이하도록 설계하였습니다.   [**A방식 (Runqueue에 queue 1걔) proc.c 구현]**   * 기본형에서는 **RunQueue 1개당 하나의 Queue에 4개의 우선순위가 혼재하여** 들어갑니다. 하지만 무작위로 넣으면 코드 성능이 현저하게 떨어지므로 위의 삽입 (appendProc()) 시 **자기보다 우선순위가 높거나 같은 노드 (prioirity가 작으면) 의 맨 끝에 삽입**하도록 설계되어 있습니다.   **[B방식 (RuneQueue에 Queue 4개) proc.c 구현]**   * 응용형에서는 **RunQueue 1개당 4개의 우선순위에 대해 각각의 Queue**가 들어있습니다. 물론 모든 큐의 접근으로 성능저하를 줄이기 위해 개의 큐의 개수를 총괄하는 종합 Counter=middleCounter 가 존재하여 Scanning으로 인하여 각 큐에 접근하여 낭비가 나지 않도록 설계하였습니다. * **이 점으로 인하여 삽입 시 A방식에서 Scanning으로 자신의 우선순위보다 아래인 프로세스를 찾을 필요 없이 각 큐의 뒷부분에 바로 삽입할 수 있도록 설계되어있습니다.** * 노드를 찾는 과정도 낮은 우선순위부터 Scanning하여 찾으며 A 방식과 시간복잡도는 비슷합니다.     [**SSU scheduler가 실행되는 모습**] |

# 결과

|  |
| --- |
| * set\_sche\_info(1,110), set\_sche\_info(10,60), set\_sche\_info(11,60) 호출 시.      * 현재 결과를 만들기 위해 sh 프로세스가 생성되고 첫 번째 프로그램이 돌아간 후로부터 스케쥴링 재갱신 ticks가 증가하도록 설계하여 결과가 명세결과가 완전히 동일하게 나옴.   **[set\_sched\_info(1,300), set\_sched\_info(22,600), set\_sched\_info(34,600)]**         * 현재 구현된 scheduler 에서 set\_sche\_info() 시스템콜의 우선순위 재갱신 반영은 최대 60ticks이 지나서야 반영이 되기 때문에 프로세스가 호출 즉시의 상태는 priority가 0으로 나올 수 있습니다. 이후 스케쥴링이 우선순위에 따라 번갈아가며 pid=3의 경우에는 **scheduler\_test**를 실행시킨 프로세스로써 부모 프로세스라 wait를 하면서 기다리게 됩니다. 그렇기에 다른 자식 **pid가 종료될 때 exit()를 호출하게되면 해당 proc.c 내부 exit() 코드에서 ZOMBIE 가된 프로세스가 있으면 sleep 코드를 깨우는 과정이 있습니다. 그렇기 때문에 자고있던 3번 pid가 깨어나게되며 가장 작은 priority를 받아 29, 72, 0 등의 다양한 pid를 확인할 수 있습니다. (그리고 가장 작은 priority가 RUNNABLE 상태에서 여러개가 존재할 때 cpu\_used가 적은 순으로 뽑히는 점을 적용하여 pid=3 이 타 프로세스보다 뽑힐 확률이 높아져 더욱 잘 수행하는 모습을 볼 수 있습니다.)** |

# 소스코드

## **proc.h (전체수록) [Process Scheduler 코드]**

|  |
| --- |
| // 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  **uint scheduler\_flag; //P3 스케쥴러 갱신여부 플래그 -> 갱신해야될 때 1, 아닐 때 0**  };  extern struct cpu cpus[NCPU];  extern int ncpu;  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)  //P3 추가 멤버  **int priority; //우선순위**  **uint priority\_tick; //우선순위 재조정 되기 전 까지 CPU 사용시간**  **uint proc\_tick; //스케쥴링 될 때마다 측정하는 Ticks**  **uint cpu\_used; //CPU 총 사용시간**  **uint proc\_deadline; //프로세스 데드라인**  **struct proc\* next;**  **struct proc\* prev;**  };  // Process memory is laid out contiguously, low addresses first:  // text  // original data and bss  // fixed-size stack  // expandable heap |

## **proc.c (전체 수록 – 전반적으로 고친부분이 많음) [Process Scheduler 코드]**

|  |
| --- |
| #include "types.h"  #include "defs.h"  #include "param.h"  #include "memlayout.h"  #include "mmu.h"  #include "x86.h"  #include "proc.h"  #include "spinlock.h"  #ifndef NULL  #define NULL ((void \*)0)  #endif  #ifndef true  #define true 1  #define false 0  #endif  #define JHS 0  #define MAX\_IDX 25  // ifdef NEWS -> 1 Queue in 1 RunQueue  // else -> 4 Queues in 1 RunQueue (기본형)  #if ORIGIN  #else  #define SUB\_IDX 4  #endif  static struct proc \*initproc;  **#ifndef NEWS**  //Priroity 내부에 4개의 우선순위를 모두 갖는 1개의 Queue  typedef struct {  struct proc\* head;  struct proc\* tail;  int queueCount;  }Priority;  **#else**  //Process 내부에는 각 Priority에 해당하는 procQ가 4개씩 존재함  typedef struct{  struct proc\* head;  struct proc\* tail;  int queueCnt;  }procQ;  typedef struct {  procQ queue[4];  int middleCnt;  }Priority;  **#endif**  //병행성 처리를 위한 락  struct {  struct spinlock lock;  struct proc proc[NPROC];  } ptable;  int nextpid = 1; //pid 번호는 순차적으로 전달  extern void forkret(void); //프로세스가 처음 실행될 때 실행되는 코드 -> first 변수 존재  extern void trapret(void);  static void wakeup1(void \*chan);  Priority RunQueue[MAX\_IDX]; //RunQueue  //추가. -> 배준형(20190511)  // default 경우 -> 1Queues in 1 RunQueue  // NEWS=true -> 4Queues in 1 RunQueue  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **#ifndef NEWS**  struct proc\* deleteQueue(Priority\* queue, struct proc\* ptr)  {  if (ptr == queue->head && ptr == queue->tail)  { // p 뿐일 때  queue->head = queue->tail = NULL;  }  else if (ptr == queue->head)  { // p가 시작부분일 때  queue->head = ptr->next;  ptr->next->prev = NULL;  }  else if (ptr == queue->tail)  { // p가 마지막 부분일 때  queue->tail = ptr->prev;  ptr->prev->next = NULL;  }  else  { // p가 중간부분일 때 중간부분 연결  ptr->prev->next = ptr->next;  ptr->next->prev = ptr->prev;  }  queue->queueCount--;  return ptr;  }  struct proc\* getHighPri()  {  int idx = 0;  Priority \*queue;  struct proc\* retProc, \*temp;  //Queue를 순회하며 가장 우선순위 작은 프로세스 찾기  for (idx = 0 ; idx < MAX\_IDX; idx++) {  if (RunQueue[idx].head == NULL)  continue;  queue = &RunQueue[idx]; //포인터를 하기쉽게 queue를 간이적으로 할당  struct proc\* ptr = queue->head; //Queue 이동을 위한 pointer  for (; ptr != NULL; ptr = ptr->next) {  if (ptr->state == RUNNABLE) //만약 상태가 RUNNABLE 이면 가장 작은 우선순위 찾은것임  break;  }  //뽑은 큐가 RUNNABLE 상태가 아니고 tail이라서 뽑힌경우  if (ptr == NULL || ptr->state != RUNNABLE)  continue;    // 11.11 조현웅 학우분 반영 추가 : 우선순위가 동일할 시 CPU\_USED 가 적은 순서가 우선이다.  temp = ptr;  if (temp->pid != 1 && temp->pid != 2) { //1,2번 프로세스는 일단 무조건 뽑힌것으로 간주  while (temp->next != NULL && temp->priority <= ptr->priority) {  temp = temp->next;  //우선순위가 동일한데 cpu 이용률이 더 적으면 ptr재갱신  if (temp->cpu\_used < ptr->cpu\_used && temp->priority <= ptr->priority) {  ptr = temp;  }  }  }  retProc = ptr;  deleteQueue(queue, retProc); //뽑힌 프로세스는 RunQueue에서 삭제  return retProc;  }  return NULL;  }  /\*\*  \* process 를 우선순위 고려해서 삽입  \*/  void appendProc(struct proc\* process)  {  Priority\* queue = &RunQueue[process->priority/4]; //RunQueue에서 해당 프로세스 위치찾기  //RunQueue가 비어있는 첫 번째 노드라면?  if(queue->head == NULL && queue->tail == NULL) {  //첫 번째 노드로 갱신  process->next = process->prev = NULL;  queue->head = queue->tail = process;  queue->queueCount++;  return;  }  else { //RunQueue가 첫 번째 노드가 아님  struct proc\* ptr;  for (ptr = queue->tail ; ptr->priority > process->priority && ptr != queue->head ; ptr = ptr->prev) {  }  //ptr보다 앞에 삽입되어야하만다면?  if (ptr->priority > process->priority) {  if (ptr == queue->head) {  //심지어 head위치에 삽입되어야하면? head위치로 삽입  process->prev = NULL;  process->next = ptr;  queue->head = process;  ptr->prev = process;  }  else {  //이외에는 ptr 앞에 process 끼어넣기  process->prev = ptr->prev;  process->next = ptr;  ptr->prev->next = process;  ptr->prev = process;  }  }  else {  //ptr보다 뒤에 삽입되어야 한다면?  if (ptr == queue->tail) {  //ptr보다 뒤에 있어야되는데 ptr이 tail이면 ptr 재갱신  process->next = NULL;  process->prev = ptr;  queue->tail = process;  ptr->next = process;  }  else {  //이외에는 ptr뒤에 process 끼어넣기  process->prev = ptr;  process->next = ptr->next;  ptr->next->prev = process;  ptr->next = process;  }  }  queue->queueCount++;  return;  }    }  void updateQueue()  {  //ptable.proc[0].priority = ptable.proc[1].priority = 99;  //ptable.proc[0].proc\_tick = ptable.proc[1].proc\_tick = 0;  int i;  struct proc \*p, \*ptr = NULL, \*tail = NULL;  Priority\* queue;  //RunQueue 순회  for (i = 0 ; i < MAX\_IDX ; i++) {  queue = &RunQueue[i];  //우선순위 갱신이 필요하면 Queue에서 빼서 queue 리스트로 연결  for (p = queue->head ; p != NULL ;) {  ptr = p;  p = ptr->next; //다음 노드로 이동  if (ptr->pid == 1 || ptr->pid == 2) { //하필이면 뽑힌 프로세스가 pid: 1,2면 패스  ptr->priority\_tick = 0;  continue;  }  //검사해보니까 priority\_tick도 사용해있거나 우선순위가 맞지않으면 재갱신 시도  if (ptr->priority\_tick != 0 || ptr->priority/4 != i) {  deleteQueue(queue, ptr);  //우선순위 재갱신  //우선순위 갱신은 바로바로 한는게 아니라 한꺼번에 간이 리스트로 연결해뒀다가 한 번에 연결  ptr->prev = ptr->next = NULL;  if (tail == NULL) {  tail = ptr;  }  else {  ptr->prev = tail;  tail = ptr;  }  }  }  }  //tail에 연결된 연결리스트 순회하며 출력  while (tail != NULL)  {  #if JHS  struct proc\* tmp = tail;  for (; tmp != NULL ; tmp = tmp->prev) {  cprintf("%d(%d)->", tmp->pid, tmp->priority\_tick);  }  cprintf("\n");  #endif    //tail 위치를 다음꺼로 연결하기 위해 tail을 미리작업  ptr = tail;  tail = tail->prev;  ptr->prev = ptr->next = NULL;  //우선순위 priority += priority\_ticks / 10 으로 재갱신  ptr->priority = ptr->priority + ptr->priority\_tick / 10;  ptr->priority = ptr->priority > 99 ? 99 : ptr->priority;  ptr->priority\_tick = 0;  // 재삽입  appendProc(ptr);  }  }  /\*\*  \* 가장 작은 우선순위를 찾아내는 함수 없을 시 0 리턴  \*/  int getSmallestPri()  {  int idx = 0;  struct proc\* p;  //Queue 순회를하며 가장 작은 우선순위 pickup  for (; idx < MAX\_IDX ; idx++) {  p = RunQueue[idx].head;  while (p != NULL) {  //찾은 프로세스가 RUNNABLE이면 최소의 priority라 간주하고 리턴  if (p->state == RUNNABLE) {  //찾은 우선순위가 1,2 pid(idle) 를 제외한 99 일 때는 받을 수 있도록 업데이트  while (p != NULL && (p->pid == 1 || p->pid == 2)) {  p = p->next;  }  if (p != NULL && p->state == RUNNABLE) //1,2를 제외한 타프로세스가 99면 99를 받음  return p->priority;  else if (p == NULL)  break;  else  p = p->next;  //cprintf("[%d] -> priority : %d\n", p->pid, p->priority);  }  else {  p = p->next;  }  }  }  return 0;  }  **#else**  int getSmallestPri()  {  int i,j;  struct proc\* p;  for (i = 0 ; i < MAX\_IDX ; i++) {  if (!RunQueue[i].middleCnt)  continue;  //4개의 Queue 중 서브 Queue 순회  for (j = 0 ; j < SUB\_IDX ; j++) {  //SubQueue가 개수가 0개 이상인경우 확인  if (!RunQueue[i].queue[j].queueCnt) {  continue;  }  p = RunQueue[i].queue[j].head;  while (p != NULL) {  //SubQueue의 번호를 확인  if (p->state == RUNNABLE) {  // 받아온 우선순위가 1,2 를 제외한 99일 때를 받아오기 위해 추가된 코드  while (p != NULL && (p->pid == 1 || p->pid == 2))  p = p->next;  if (p != NULL && p->state == RUNNABLE) {  return i \* 4 + j;  }  else if (p == NULL)  break;  else  p = p->next;  }  else  p = p->next;  }  //cprintf("[%d] state : %d\n", RunQueue[i].queue[j].head->pid, RunQueue[i].queue[j].head->state);  }  }  return 0;  }  struct proc\* deleteQueue(procQ\* queue, struct proc\* retProc)  {  //사실상 위의 deleteQueue와 매커니즘 동일함  if (queue == NULL || retProc == NULL)  return NULL;  //Queue요소개수가 1개면 head,tail에 연결  if (queue->head == queue->tail) {  queue->head = queue->tail = NULL;  queue->queueCnt = 0;  }  else if (retProc == queue->head) //Queue의 맨 앞을 삭제하는 경우  {  retProc->next->prev = NULL;  queue->head = queue->head->next;  }  else if (retProc == queue->tail) //Queue의 맨 뒤를 삭제하는 경우  {  retProc->prev->next = NULL;  queue->tail = queue->tail->prev;  }  else //중간 노드를 삭제하는 경우  {  retProc->prev->next = retProc->next;  retProc->next->prev = retProc->prev;  }  retProc->next = retProc->prev = NULL;  queue->queueCnt--;  return retProc;  }  struct proc\* getHighPri()  {  int i, j;  struct proc\* retProc, \*temp;  procQ\* queue;  // Queue를 처음부터 순회  for (i = 0 ; i < MAX\_IDX ; i++) {  //Queue의 중간 Queue들의 각 개수를 확인하는 부분  if(!RunQueue[i].middleCnt)  continue;  //SubQueue를 차례대로 순회하며 process 선택 준비  for (j = 0 ; j < SUB\_IDX ; j++) {  queue = &(RunQueue[i].queue[j]);  if (!queue->queueCnt)  continue;  // Sub Queue에서 RUNNING인 prcess를 찾을때까지 iteration 진행  for (retProc = queue->head ; retProc && retProc->state != RUNNABLE ; retProc = retProc->next);  if (!retProc)  continue;  // retProc가 RUNNABLE 상태인 경우  else if (retProc->state == RUNNABLE) {  // 11.11 조현웅 학우분 반영 추가 : 우선순위가 동일할 시 CPU\_USED 가 적은 순서가 우선이다.  temp = retProc;  //1,2번 PID는 선택되면 무조건 리턴 (idle는 초반에 실행되고 실행이 되지않지만, 초반엔 무조건 실행되어야함)  if (temp->pid != 1 && temp->pid != 2) {  while (temp->next != NULL && temp->priority <= retProc->priority) {  temp = temp->next;  //우선순위가 동일하거나 (작으면서) CPU사용시간이 적으면 해당 노드로 retProc 갱신  if (temp->cpu\_used < retProc->cpu\_used && temp->priority <= retProc->priority) {  retProc = temp;  }  }  }  //cprintf("%d state %d\n", retProc->pid, retProc->state);  //다음에 스케쥴될 Process는 RunQueue에서 삭제  retProc = deleteQueue(queue, retProc);  if (retProc == NULL)  return NULL;  RunQueue[i].middleCnt--;  return retProc;  }  }  }  return NULL;  }  /\*\*  \* RunQueue에 proc 를 추가하는 함수  \*/  void appendProc(struct proc\* proc)  {  if (proc == NULL)  return;  //삽입될 위치 확인  procQ\* queue = &(RunQueue[proc->priority/4].queue[proc->priority % 4]);  if (queue->head == NULL && queue->tail == NULL) {  //만약 RunQueue가 비어있다면 초기세팅구성  proc->next = proc->prev = NULL;  queue->head = queue->tail = proc;  queue->queueCnt = 1;  }  else {  //RunQueue가 비어있지않다면 tail 위치에다가 Proecss 삽입  proc->prev = queue->tail;  proc->next = NULL;  queue->tail->next = proc;  queue->tail = proc;  queue->queueCnt++;  }  RunQueue[proc->priority/4].middleCnt++;  }  //프로세스 초기화부분을 세팅하는 함수  //원래 배치될 부분은 scheduler()실행 전인 userinit()에 넣는게 맞음  void initQueue()  {  int i, j;  for (i = 0 ; i < MAX\_IDX ; i++) {  RunQueue[i].middleCnt = 0;  for (j = 0 ; j < SUB\_IDX ; j++) {  RunQueue[i].queue[j].head = RunQueue[i].queue[j].tail = NULL;  RunQueue[i].queue[j].queueCnt = 0;  }  }  }  /\*\*  \* Priority 재갱신이 필요할 때 호출하는 함수  \* 스케쥴러 내부에서 호출될 예정  \*/  void updateQueue()  {  int i, j;  procQ\* queue;  struct proc\* tmp, \*updateNode;  //RunQueue Index를 차례차례 순회  for (i = 0 ; i < MAX\_IDX ; i++) {  //RunQueue내부의 4개의 큐가 모두 비어있음을 확인하는 middleCnt 확인  if (!RunQueue[i].middleCnt)  continue;  //RunQueue 내부 Queue하나하나에 접근  for (j = 0 ; j < SUB\_IDX ; j++) {  queue = &(RunQueue[i].queue[j]);  if (!queue->queueCnt)  continue;  //RunQueue를 순회하며 업데이트해야될 process를 찾는과정  for (tmp = queue->head ; tmp != NULL ;) {  //prioriy\_tick도 0 tick이 아니면서 우선순위도 맞지 않는 경우를 찾음  if (tmp->priority\_tick == 0 && tmp->priority/4 == i) {  tmp = tmp->next;  continue;  }  //update 할 노드를 찾고 해당 노드를 재삽입을 위한 삭제를함  //그냥 삭제+삽입 할 경우 연결리스트의 구조가 깨지기 때문에 tmpNode 따로 두기  updateNode = tmp;  tmp = tmp->next;  //update를 위해 해당 노드를 Queue에서 빼냄  updateNode = deleteQueue(queue, updateNode);  RunQueue[i].middleCnt--;  //update진행  if (updateNode != NULL) {  //우선순위 재계산 후 업데이트 진행  updateNode->priority = updateNode->priority + updateNode->priority\_tick/10;  updateNode->priority = updateNode->priority > 99 ? 99 : updateNode->priority;  updateNode->priority\_tick = 0;  appendProc(updateNode);  }  }  }  }  }  **#endif**  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  //pTable락을 초기화하는 부분  void  **pinit(void)**  {  //cprintf("pinit : %d\n", myproc()->pid);  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(); //인터럽트 불능화  c = mycpu();  p = c->proc;  popcli(); //인터럽트 불능화 해제  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;  //초기 락을 설정  acquire(&ptable.lock);  //ptable을 단순순회하며 UNUSED 프로세스가 있으면 found로 이동  //goto를 쓴 이유는 모르곘음 (안정성 떨어져보임)  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)  if(p->state == UNUSED)  goto found;  release(&ptable.lock);  return 0;  found:  //아직 프로세스가 할당받기전이라 EMBRYO 상태  p->state = EMBRYO;  p->pid = nextpid++;  release(&ptable.lock);  // Allocate kernel stack. 프로세스 커널스택을 하나 할당받음  if((p->kstack = kalloc()) == 0){  p->state = UNUSED;  return 0;  }  sp = p->kstack + KSTACKSIZE; //프로세스 kernel 스택 위로 이동  // Leave room for trap frame.  //kernel stack 맨 윗부분에는 trapframe을 설정  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; //1번 프로세스처럼 가장 처음에 실행되는 프로세스는 iinit(), initlog() 설정을 위한 이동  sp -= sizeof \*p->context;  p->context = (struct context\*)sp;  memset(p->context, 0, sizeof \*p->context);  p->context->eip = (uint)forkret;  //extractTick.c 에서 프로세스 분석을 위해 넣어둔 코드 (프로세스 생성 tick 시간 확인)  #if ANALY  acquire(&tickslock);  cprintf("PID : %d, %d (0)\n", p->pid, ticks);  release(&tickslock);  #endif  //P3 과제를 위한 프로세스 설정  p->proc\_tick = 0; //생성된 시점에서 proc\_tick=0으로 설정  p->priority\_tick = p->cpu\_used = 0;  p->proc\_deadline = -1;  p->priority = getSmallestPri();  //프로세스 우선순위는 0,1,2 ilde를 제외한 가장 작은 값으로 설정  if (p->pid == 0 || p->pid == 1 || p->pid == 2)  p->priority = 99;  /\*  else if (p->priority == 99) //만약 99로 설정된다면 0으로 설정  p->priority = 0;  \*/  appendProc(p);  return p;  }  //PAGEBREAK: 32  // Set up first user process.  void  userinit(void)  {  //cprintf("userinit : %d\n", myproc()->pid);  struct proc \*p;  extern char \_binary\_initcode\_start[], \_binary\_initcode\_size[];  p = allocproc();    initproc = p;  // 초기 kernel Process table을 설정하는 부분  if((p->pgdir = setupkvm()) == 0)  panic("userinit: out of memory?");  inituvm(p->pgdir, \_binary\_initcode\_start, (int)\_binary\_initcode\_size);  //initcode.S 로 실행하기 위해 담는 부분  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);  appendProc(p);  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){ //프로세스 생성  return -1;  }  // Copy process state from proc.  //페이지 디렉토리는 부모 디렉토리를 복사  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){  kfree(np->kstack);  np->kstack = 0;  np->state = UNUSED;  return -1;  }  //부모프로세스 trapframe 등을 다 복사  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;  //파일 디스크립터 부모꺼 복사  // 이부분을 수정하면 예전에 리시프에서 GETFD, SETFD 로 CLOSE\_ON\_EXEC 부분을 조절가능할 듯  for(i = 0; i < NOFILE; i++)  if(curproc->ofile[i])  np->ofile[i] = filedup(curproc->ofile[i]);  np->cwd = idup(curproc->cwd);  //부모 이름을 복사 (디버그시 사용)  safestrcpy(np->name, curproc->name, sizeof(curproc->name));  pid = np->pid; //자식 pid 리턴  acquire(&ptable.lock);  //appendProc(np);  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;  }  }  //Journaling 기법으로 File System이 구현되어있기 때문에  begin\_op(); //Commit 까지 일단 시행  iput(curproc->cwd);  end\_op(); //Commit 이 끝난점을 확인한 후 log 메타데이터 저널리  curproc->cwd = 0;  acquire(&ptable.lock);  // Parent might be sleeping in wait().  wakeup1(curproc->parent);  // Pass abandoned children to init.  //프로세스 테이블을 돌면서 죽은 자식이 있으면 부모에게 회수하도록 지시  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.  //현 프로세스는 append하면안됨 (죽었으니 돌아오지않음)  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; //아래 죽은 자식이 있는지 없는지 체크를 위함  if(p->state == ZOMBIE){  // 죽은 자식을 찾았으면 죽은 자식의 내부구조체를 모두 초기화시켜버림  pid = p->pid;  kfree(p->kstack);  p->kstack = 0;  freevm(p->pgdir);  p->pid = 0;  p->parent = 0;  p->name[0] = 0; //독특한 점은 memset이 아닌 초기값만 0으로 바꿔서 string이 안되도록 설정.  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.)  //자식이 올 때까지 lock을 풀어주고 sleep모드로 변경  sleep(curproc, &ptable.lock); //DOC: wait-sleep  }  }  //#define DEBUG  //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)**  {  #if NEWS  cprintf("in 4 Queues in 1 Queue Schdeuler\n");  #endif  struct proc \*p;  struct cpu \*c = mycpu();  //int ct = 0;  c->proc = 0;  for(;;){  // Enable interrupts on this processor.  sti(); //cpu가 apic로부터 TIMER INTERRUPT 를 받아 preemption을 가능하게 하기위한 sti()  acquire(&ptable.lock); //병행성 문제를 해결하기 위한 ptable Lock (Spinlock : while 락)  // Loop over process table looking for process to run.  //RunQueue에서 우선순위 높은 친구 뽑아옴 (만약 같을 시 cpu 사용량 적은 친구로 뽑기..)  if (!(p = getHighPri())) {  release(&ptable.lock);  continue;  }  //기본 설계과제 제출용 출력부분  #ifdef DEBUG  cprintf("PID : %d, priority : %d, proc\_tick : %d ticks, total\_cpu\_usage : %d ticks\n",  p->pid, p->priority, p->proc\_tick, p->cpu\_used);  #endif  //예시 1-3 번등을 위해 설정한 매크로  #ifdef DEBUGS  cprintf("PID : %d, priority : %d, proc\_tick : %d ticks, total\_cpu\_usage : %d ticks (2)\n",  p->pid, p->priority, p->proc\_tick, p->cpu\_used);  #endif  //extractTick.c 에서 프로세스 분석을 위해 넣어둔 코드 (프로세스 스케쥴링 호출시마다 출력)  #ifdef ANALY  acquire(&tickslock);  cprintf("PID : %d, priority : %d, proc\_tick : %d ticks, total\_cpu\_usage : %d ticks, totalTicks : %d (2)\n",  p->pid, p->priority, p->proc\_tick, p->cpu\_used, ticks);  release(&tickslock);  #endif  c->proc = p; //스케쥴링 뽑인 녀석을 cpu 의 프로세스로 설정  switchuvm(p); //바뀐 p의 pagetable을 가져오는 함수  p->state = RUNNING;  swtch(&(c->scheduler), p->context); //CPU에게 현재 proc.c schdeuler 스케쥴러에서 프로세스 context로 전환  switchkvm(); // 스케쥴러로 돌아왔으므로 다시 Kernel Pagetable loading  //현재 cpu에 재갱신 플래그 (Scheduler\_flag 가 설정되어있는 경우)  if (mycpu()->scheduler\_flag)  {  mycpu()->scheduler\_flag = 0; //플래그 끄기  updateQueue(); //재갱신 진행시켜  //Queue에서 삽입해야함 (바로 다시 스케쥴리을 할 것이기 때문)  **#ifndef NEWS**  //RunQueue 용 Switch인 경우엔 재삽입을 위한 프로세스 삭제 (갱신된 후 다시 빼는 과정)  deleteQueue(&RunQueue[p->priority/4], p);  **#else**  deleteQueue(&(RunQueue[p->priority/4].queue[p->priority % 4]),p);  **#endif**  //다시 실행중이던 프로세스로 복귀 (해당 프로세스도 업데이트 됨)  switchuvm(p);  p->state = RUNNING;  swtch(&(c->scheduler), p->context);  //해당 프로세스가 종료하면 해당위치부터 실행 (추가하기전이랑 사실상 동일한 위치)  switchkvm();  }  c->proc = 0;  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)) //락을 가지고 있는지 확인  panic("sched ptable.lock");  if(mycpu()->ncli != 1) //interrupt 불능화 시켜놨는데 1로설정됨?  panic("sched locks");  if(p->state == RUNNING) //process를 yield에서 RUNABLE로 바꿨는데 RUNNING인 경우  panic("sched running");  if(readeflags()&FL\_IF)  panic("sched interruptible");  intena = mycpu()->intena;  swtch(&p->context, mycpu()->scheduler); //스케쥴러로 복귀  mycpu()->intena = intena;  }  **void**  **update(void)**  {  acquire(&ptable.lock); //DOC: yieldlock  myproc()->state = RUNNABLE;  sched();  release(&ptable.lock);  }  // Give up the CPU for one scheduling round.  **void**  **yield(void)**  {  acquire(&ptable.lock); //DOC: yieldlock  appendProc(myproc());  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); //파일 시스템 디스크 첫번째 superblock 받아오기  initlog(ROOTDEV); //journaling 기법으로 메타데이터 혹시 있는지 복원 (그리고 해당 블록초기화)  }  // 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)  {  //cprintf("sleep start\n");  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.  // sleep 전의 락을 걸기위해 서로간의 약속된 lk락을 버리고 ptable락으로 설정  if(lk != &ptable.lock){ //DOC: sleeplock0  acquire(&ptable.lock); //DOC: sleeplock1  release(lk);  }  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;  p->priority = getSmallestPri(); //sleep->깨어났을 때 RunQueue에서 가장 작은 우선순위를 갖도록 설정  if (p->pid == 0 || p->pid == 1 || p->pid == 2)  p->priority = 99;  //스케쥴링이 될 수 있또록 appendProc() 삽입  appendProc(p);  }  }  // Wake up all processes sleeping on chan.  **void**  **wakeup(void \*chan)**  {  //병행성 문제를 해결하기 위해 ptable lock으로 둘러쌈  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);  //ptable을 순회하며 혹시 SLEEPING 프로세스가 있으면 (xv6에서 sleeping하는 프로세스는 자식 프로세스를 기다리는 프로세스임)  //RUNNABLE로 변경하고 wait에서 깨어날 수 있도록함  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;  //해당 부분에서 가장 작은 값을 가지도록 설정함 보통 init sh 에서 다음에 실행시킨 3번 pid등이 주로 걸림  p->priority = getSmallestPri();  //근데 idle 프로세스면 99로 설정  if (p->pid == 0 || p->pid == 1 || p->pid == 2)  p->priority = 99;  //sleep에서 꺠어났으니 스케쥴링이 될 수 있도록 다시 RunQueue에 삽입  appendProc(p);  }  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.  ///디버그용 함수 CTRL+P를 누르면 프로세스 상태를 알 수 있음  /\*\*  \* 프로세스 상태가 죽었는지 확인하는 방법  \* CTRL+P : 프로세스 스케쥴러가 RunQueue에 삽입은 되었는데 스케쥴링이 되지 않거나 비어있는 경우 CTRL+P를 하면 프로세스 상태를 알려줌  \* trap 14 : NULL 프로세스의 포인터를 참조하거나 이럴 때 주로 발생 (페이지폴트니까.. --> RunQueue 부분 다시 살펴볼 것)  \* xv6 멈춤 현상 : 보통 이부분은 커널 update함수나 delete함수에 while(1); 에서 무한루프를 돌아서 찍히지 않는 경우가 대부분  \*/  **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");  }  } |

## **trap.c (전체 수록) [Process Scheduler 코드]**

|  |
| --- |
| #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"  // 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 scheduler\_tick; //스케쥴러 재갱신 전용 tick  int proc\_tick\_lock = 0; //스케쥴러 재갱신 tick을 증가시키기위한lock (pid 3이상부터 측정 -> 제일처음 실행되는 프로그램부터..)  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)  {  //시스템 콜일 때 trap  if(tf->trapno == T\_SYSCALL){  //중간에 프로세스 죽었으면 종료시킴  if(myproc()->killed) {  exit();  }  //현 프로세스의 trapframe을 갱신시켜줌 -> eax로 syscall 찾기가능  myproc()->tf = tf;  syscall();  if(myproc()->killed){  exit();  }  return;  }  switch(tf->trapno){  case T\_IRQ0 + IRQ\_TIMER:  if(cpuid() == 0){  acquire(&tickslock);  ticks++;  // 스케쥴링 시간동안 cpu\_used를 증가.  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();  }  //프로세스 Timer Interrupt인 경우에만 진입  if (myproc() && tf->trapno == T\_IRQ0 + IRQ\_TIMER) {  if (myproc()->pid >= 3 && proc\_tick\_lock == 0) {  //만약에 pid가 3이상 (userprogram 시작) 인데 lock이 안 풀려있으면 lock을 풀어줌  proc\_tick\_lock = 1;  }  myproc()->cpu\_used++;  myproc()->priority\_tick++;  myproc()->proc\_tick++;  //process pid 가 3이상일 때 스케쥴링 시간을 측정함  if (myproc()->pid != 1 && myproc()->pid != 2 && proc\_tick\_lock)  scheduler\_tick++;  //cprintf("my cpu ticks : %d\n", mycpu()->proc\_tick);  }    //P3 -> TimeQuantum 60s 로 변경  if(myproc() && myproc()->state == RUNNING && tf->trapno == T\_IRQ0+IRQ\_TIMER) {  //CPU 사용예약시간 초과 시 종료처리  //Process Deadline 넘어가면 종료시키기  if (myproc() && myproc()->proc\_deadline != -1 && myproc()->proc\_deadline <= myproc()->cpu\_used) {    //예시 1-3 등에서 프로세스 분석을 위해 넣어둔 코드 (프로세스 종료(퇴장) tick 시간 확인)  #ifdef DEBUGS  cprintf("PID : %d, priority : %d, proc\_tick : %d ticks, total\_cpu\_usage : %d ticks (3)\n",  myproc()->pid, myproc()->priority, myproc()->proc\_tick, myproc()->cpu\_used);  #endif  //extractTick.c 에서 프로세스 분석을 위해 넣어둔 코드 (프로세스 정료 tick 확인)  #ifdef ANALY  acquire(&tickslock);  cprintf("PID : %d, priority : %d, proc\_tick : %d ticks, total\_cpu\_usage : %d ticks, totalTicks : %d (3)\n",  myproc()->pid, myproc()->priority, myproc()->proc\_tick, myproc()->cpu\_used, ticks);  release(&tickslock);  #endif  cprintf("PID : %d terminated\n", myproc()->pid);  exit();  }  //cpu 시간이 30 지나면 yield(); -> Scehduler진입  if (myproc() && myproc()->proc\_tick >= 30) {  //예시 1-3 번등에서 스케쥴링이 종료되는 순간 출력되도록 설정  #ifdef DEBUGS  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);  #endif  myproc()->proc\_tick = 0; //스케쥴링이 완료되었으니까 process 스케쥴링 tick은 초기화  yield();  }  //scheduler\_tick 프로세스 tick을 측정하고 있다가 60이 넘어가면 cpu에 재갱신 플래그를 만들고 진입  if (scheduler\_tick >= 60) {  mycpu()->scheduler\_flag = 1; //cpu에다 나 우선순위 재갱신 하겠다 알려주는 부분  scheduler\_tick = 0; //trap.c scheduler 재갱신 tick 전역변수 초기화  yield(); //priority 재갱신하러 진입  }  }  // Check if the process has been killed since we yielded  if(myproc() && myproc()->killed && (tf->cs&3) == DPL\_USER)  exit();  } |

## **scheduler\_test (전체 수록) [xv6 user program]**

|  |
| --- |
| #include "types.h"  #include "stat.h"  #include "user.h"  #define PNUM 3  #define SHORT 1  void scheduler\_func(void)  {  int pid;  int i;  #if SHORT  //스케쥴링 userprogram 시간을 줄이고 3개 fork만 조절하기 위해 특별히 제작  printf(1, "scheduler\_test start\n");  pid = fork();  if (pid == 0) {  set\_sche\_info(1, 300);  while(1);  }  pid = fork();  if (pid == 0) {  set\_sche\_info(22, 600);  while(1);  }  pid = fork();  if (pid == 0) {  set\_sche\_info(34, 600);  while(1);  }  #else  //일반적으로 선형적인 구조로 PNUM을 실행시키기 위해 제작  printf(1, "scheduler\_test start : PNUM > %d\n", PNUM);  uint start, end;  start = myticks();  //int schedule\_list[6] = {1, 300, 24, 600, 34, 600}; // 각 프로세스의 priority 및 종료 타이머  //int schedule\_list[6] = {1, 110, 10, 60, 11, 60}; // 각 프로세스의 priority 및 종료 타이머  for (i = 0; i < PNUM; i++) {  pid = fork();  if (pid == 0) {  set\_sche\_info(i+6\*i/2, i\*100);  //set\_sche\_info(schedule\_list[i\*2], schedule\_list[i\*2+1]);  while(1);  exit();  }  }  #endif  //남은 프로세스들을 기다리는 부분 (부모가 자식 기다림)  for (i = 0; i < PNUM; i++) {  wait();  }  #if SHORT  printf(1, "end of scheduler\_test\n");  #else  end = myticks(); //스케쥴링이 끝난 시점 틱 측정  printf(1, "end of scheduler\_test : %d ticks\n", end-start);  #endif  }  /\*\*  \* pnum 만큼 3~pnum 만큼 연속적으로 fork하여 전체성능을 보는 함수  \*/  void scheduler\_func\_V2(int pnum)  {  printf(1, "scheduler\_test start[PNUM:%d]\n", pnum);  int pid;  int i;  uint start, end;  start = myticks(); //시작 틱 측정  //int schedule\_list[6] = {1, 300, 24, 600, 34, 600}; // 각 프로세스의 priority 및 종료 타이머  //int schedule\_list[6] = {1, 110, 10, 60, 11, 60}; // 각 프로세스의 priority 및 종료 타이머  for (i = 0; i < pnum; i++) {  pid = fork();  if (pid == 0) {  set\_sche\_info(i+6\*i/2, (i+1)\*40); //priority가 대략 4씩 증가하도록 설정  //set\_sche\_info(schedule\_list[i\*2], schedule\_list[i\*2+1]);  while(1);  exit();  }  }  //남은 프로세스들을 기다리는 부분 (부모가 자식 기다림)  for (i = 0; i < pnum; i++) {  wait();  }  end = myticks(); //끝난 시점에서 tick 측정  printf(1, "end of scheduler\_test[PNUM>%d] : %d ticks\n",pnum, end-start);  }  void scheduler\_testing(int N) {  int i;  for (i = 3 ; i < N ; i++) {  printf(1, "\n\n\n========================================\n");  scheduler\_func\_V2(i);  printf(1, "========================================\n");  }  }  /\*\*  \* 2개는 CPU 위주 작업, 10개는 I/O (sleep) 위주 작업 구성  \*/  void scheduler\_testing\_3(int cpu\_process, int io\_process) {  uint start, end;  int pnum = cpu\_process + io\_process;  int pid, j, ticks;  start = myticks(); //시작 tick 측정  printf(1, "scheduler\_test start[PNUM:%d]\n", pnum);  //cpu위주 작업 while(1); 만 반복하는 부분  for (j = 0 ; j < cpu\_process ; j++) {  pid = fork();  if (pid == 0) {  set\_sche\_info(80, 400);  while(1);  }  }  //io 위주의 작업을 시행하는 부분  for (j = 0 ; j < io\_process ; j++) {  pid = fork();  if (pid == 0) {  set\_sche\_info(2, 400); //실행시간을 400ticks으로 고정  ticks = myticks();  while(1) {  //그냥 while(1)만 하면 1tick이 증가하기도 전에 io를 하러가는 점 방지코드  if (ticks + 1 > myticks()) {  ticks = myticks();  sleep(5);  }  }  }  }  for (j = 0 ; j < cpu\_process + io\_process ; j++)  wait();  end = myticks();  printf(1, "end of scheduler\_test[PNUM>%d] : %d ticks\n",pnum, end-start);  }  /\*\*  \* ticks 제한 없이 무한정 sleep하게 하는 경우  \*/  void scheduler\_testing\_4(int cpu\_process, int io\_process) {  //scheduler\_testing\_3 과 동일한데 i/o(sleep) 작업에서 무한 sleep() 이 발생할 수 있음  uint start, end;  int pnum = cpu\_process + io\_process; //PNUM 개수  int pid, j;  start = myticks(); //시작시점 tick 측정  printf(1, "scheduler\_test start[PNUM:%d]\n", pnum);  for (j = 0 ; j < cpu\_process ; j++) {  pid = fork();  if (pid == 0) {  set\_sche\_info(80, 400); //CPU 위주 작업 세팅  while(1);  }  }  for (j = 0 ; j < io\_process ; j++) {  pid = fork();  if (pid == 0) {  set\_sche\_info(2, 400); //IO 위주 작업 세팅  while(1) {  // 1tick 이 끝나기도 전에 sleep을 하러가는게 문제  sleep(1);  }  }  }  for (j = 0 ; j < cpu\_process + io\_process ; j++)  wait();  end = myticks(); //끝난 시점의 tick 측정  printf(1, "end of scheduler\_test[PNUM>%d] : %d ticks\n",pnum, end-start);  }  int main(void)  {  scheduler\_func();  //scheduler\_testing(20); //PNUM을 3~20 연속 수행하여 전체 tick 성능 분석  //scheduler\_testing\_4(5,15); //sleep위주작업 vs cpu 위주작업 측정  exit();  } |

## **sysproc.c (추가된 부분만 수록) [시스템콜 추가]**

|  |
| --- |
| ...  // 프로세스 우선순위와 tick 설정 (바로 설정이 아닌 재갱신 때 반영될 예정)  **int**  **sys\_set\_sche\_info(void)**  {  int priority, tcks;  argint(0, &priority); //0번 argument 가져옴  argint(1, &tcks); //1번 정수형 arguement 가져움  if (priority < 0)  priority = 0;  else if (priority > 99)  priority = 99;  //프로세스 tick 설정  if (tcks < 0) { //ticks 음수 처리 -> 실패로 간주  return -1;  }  cprintf("set\_sche\_info() pid : %d\n", myproc()->pid);  #ifdef ANALY  cprintf("[SET] pid : %d, priority : %d, schedule\_ticks : %d\n", myproc()->pid, priority, tcks);  #endif  myproc()->proc\_deadline = tcks; //timer ticks 설정  myproc()->priority = priority; //우선순위 설정  /\*\* cpu값을 바로 적용하면..? 11.12 \*/  // 예시 결과와도 차이가 나고, 이전 방식이 더 안정적이라 일단 폐기  /\*  pushcli(); //Interrupt 활성화  mycpu()->scheduler\_flag = 1; //현 cpu에게 update flage 설정  popcli(); //interrupt 끄기  yield(); //update를 위한 yield();  \*/  return -1;  }  //현재 ticks를 알려주는 단순한 코드  **int**  **sys\_myticks(void)**  {  int tck;  acquire(&tickslock); //병행성 처리를 위해 둘러싼 ticks  tck = ticks;  release(&tickslock);  return tck;  } |

## **usys.S 추가 + syscall.c + syscall.h (추가된 부분만 수록) + user.h**

|  |
| --- |
| * **usys.S 추가된 부분**   ...  SYSCALL(set\_sche\_info)  SYSCALL(myticks)   * **syscall.c 추가된 부분**   extern int sys\_set\_sche\_info(void);  extern int sys\_myticks(void);  ...  static int (\*syscalls[])(void) = { ....  [SYS\_set\_sche\_info] sys\_set\_sche\_info,  [SYS\_myticks] sys\_myticks,  };   * **syscall.h 추가된 부분**   ..  #define SYS\_set\_sche\_info 22  #define SYS\_myticks 23   * **user.h추가된 부분**   ...  int set\_sche\_info(int,int);  int myticks(void); |

## **Makefile (추가된 부분만 수록) [Makefile]**

|  |
| --- |
| ....  ifeq ($(debugs), 1)  CFLAGS += -DDEBUGS  endif  ifeq ($(debug), 1)  CFLAGS += -DDEBUG  endif  ifeq ($(analy), 1)  CFLAGS += -DANALY  endif  ifeq ($(news), 1)  CFLAGS += -DNEWS  endif  ...  ifndef CPUS  CPUS := 1 #명세대로 내용 수정  endif |

## **extractTicks.c [Tick분석 프로그램)**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <string.h>  #define FILENAME "test.txt"  #define EXTRACTFILE "middlefile.txt"  #define START\_SCHED "scheduler\_test start"  #define END\_SCHED "end of scheduler\_test"  #define INIT\_PID "(0)"  #define DIED\_PID "(3)"  #define FIND\_PID "(2)"  #define PID\_TIMER "[SET]"  #define DEBUG 1  #define LINEBUF 1024  #define MAX\_PID 512  struct {  int pid;  int pnum\_id;  int sched\_time; //프로세스 예상 실행시간  int create\_ticks; //프로세스 생성 시점 Tick  int start\_ticks; //프로세스 첫 스케쥴링 tick  int end\_ticks; //프로세스 스케쥴링 종료 Tick  int CallCount; //프로세스 스케쥴링 호출횟수  int priority; //프로세스 세팅 우선순위  int turnAroundTime; //Turn around time  int responseTime; //Response Time  }pidInfo[MAX\_PID];  typedef struct q\_ {  int line;  int commaCount;  struct q\_\* next;  char lineBuf[LINEBUF];  }q;  //char\* findTick(char\* line, char\* filebuf);  void printList(q\* q); //디버그용 화면 출력  void extractBuffer (q\* q, FILE\* fp); //만들어진 연결리스트를 csv파일로 실제로 추출하는 함수  q\* makeCSV (q\* s); //분석된 tick pidInfo를 바탕으로 csv파일로 잘 정리해서 추출해주는 함수  q\* findTotalTicks(FILE\* fp); //결과 txt로부터 틱 분석 함수  int main(void)  {  FILE\* fp1, \*fp2;  char filebuf[LINEBUF];  char csvname[LINEBUF];  if ((fp1 = fopen (FILENAME,"w+")) == NULL) {  fprintf(stderr, "fopen error : %s\n", FILENAME);  exit(0);  }  //시스템 함수를 통해 xv6 실행 및 리다이렉션은 자동화시켜줌  system("make clean");  system("make analy=1 qemu > test.txt"); //Makefile에 analy=1를 옵션추가 시 플래그가 설정되어 tick분석을 할 수 있도록함  //만들어진 test.txt로부터 csv파일을 생성하기 위한 이름 (csv없이 출력해야함)  printf("input your csv file name : ");  fgets(csvname, LINEBUF, stdin);  strcpy(csvname + strlen(csvname) - 1, ".csv");  if ((fp2 = fopen (csvname,"w")) == NULL) {  fprintf(stderr, "fopen error : %s\n", EXTRACTFILE);  exit(0);  }  //차례대로 tick 분석 실행  q\* tickQ = findTotalTicks(fp1); //tick을 분석하여 pidInfo등 갱신  printList(tickQ); //일단 한 번 출력  tickQ = makeCSV(tickQ); //분석된 tick정보를 csv파일로 변환  extractBuffer(tickQ, fp2); //csv 파일버퍼를 진짜 csv로 출력  fclose(fp1); //오픈된 파일스트림 닫기 fp1,fp2  fclose(fp2);  exit(0);  }  //연결리스트 s를 fp csv파일로 추출하는 함수  void extractBuffer (q\* s, FILE\* fp) {  printf("Extract file to : %d\n", fp->\_fileno);  q\* delNode = s;  //그냥 연결리스트 순회하면서 파일로 추출  while (s != NULL) {  fprintf(fp, "%s\n", s->lineBuf);  s->commaCount = 0;  s->line = 0;  memset(s->lineBuf, 0 , LINEBUF);  delNode = s;  s = s->next;  free(delNode); //다 쓴 연결리스트는 free (안해줘도 OS가 해주긴함)  }  }  /\*\*  \* csv파일로 추출하여 그래프, 도표등을 보기쉽게 표현할 수 있도록 자동화한 함수  \*/  q\* makeCSV (q\* s)  {  //Queue가 비었으면 Queue 생성  if (s == NULL) {  fprintf(stderr, "queue is NULL\n");  s = (q\*)malloc(sizeof(q));  s->line = 1;  memset(s->lineBuf, 0, LINEBUF);  strcpy(s->lineBuf, ",");  s->commaCount = 1;  }  //csv파일의 제목을 달아줌  char title[LINEBUF] = "PNUM,Total\_tick,,PNUM,PID,  schedTime,TurnAround,Response,CallCount,CreateTick,firstSchedTick,EndSchedtick,priority";  int commCount = 1;  q\* ptr = s, \*saved = s;  for (int i = 0 ; i < MAX\_PID ; i++) {  //pidInfo[i].end\_ticks 가 0이면 측정할 프로세스대상이 아님 (init 혹은 sh 프로세스)  if (pidInfo[i].pid != 0 && pidInfo[i].end\_ticks != 0 && pidInfo[i].pnum\_id != -1) {  //Queue가 비었으면 앞단에 PNUM 전체 ticks + 한 칸 띄우고 정보를 저장하도록 설계  if (ptr == NULL)  {  ptr = (q\*)malloc(sizeof(q));  memset(ptr->lineBuf, 0, LINEBUF);  //앞단에 콤마가 몇개까지 있었는지 갱신  ptr->commaCount = saved->commaCount - commCount;  saved->next = ptr; //연결리스트 연결  ptr->line = saved->line + 1;  ptr->next = NULL;  //앞단 콤마까지 콤마추가  for (int comm = 0; comm < commCount; comm++)  strcat(ptr->lineBuf, ",");  }  //리스트 정보갱신  sprintf(ptr->lineBuf + strlen(ptr->lineBuf), ",,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d",  pidInfo[i].pnum\_id, pidInfo[i].pid, pidInfo[i].sched\_time,  pidInfo[i].turnAroundTime, pidInfo[i].responseTime, pidInfo[i].CallCount,  pidInfo[i].create\_ticks, pidInfo[i].start\_ticks, pidInfo[i].end\_ticks,  pidInfo[i].priority  );  saved = ptr;  ptr = ptr->next; //리스트 연결 및 순회  }  }    //리스트 추가가 완료되었으면 리스트의 가장 앞단에 제목을 추가해서 리스트갱신  q\* init = (q\*)malloc(sizeof(q));  init->line = 0;  init->commaCount = -1;  strcpy(init->lineBuf, title);  init->next = s;  return init;  }  void printList(q\* s) {  for (q\* ptr = s ; ptr != NULL ; ptr = ptr->next) {  printf("[%d] %s\n", ptr->line, ptr->lineBuf);  }  for (int i = 0 ; i < 512 ; i++) {  if (pidInfo[i].pid == 0)  continue;  #if DEBUG  printf("=============================\n");  printf("[pid : %d] info\n", pidInfo[i].pid);  printf("pnumID : %d\n", pidInfo[i].pnum\_id);  printf("create\_ticks : %d\n", pidInfo[i].create\_ticks);  printf("start\_ticks : %d\n", pidInfo[i].start\_ticks);  printf("end\_ticks : %d\n", pidInfo[i].end\_ticks);  printf("\n");  printf("Process Priority : %d\n", pidInfo[i].priority);  printf("sched\_time : %d\n", pidInfo[i].sched\_time);  printf("call\_count : %d\n", pidInfo[i].CallCount);  printf("Turnaround Time : %d\n", pidInfo[i].turnAroundTime);  printf("Response Time : %d\n", pidInfo[i].responseTime);  printf("=============================\n\n");  #endif  }  }  q\* findTotalTicks(FILE\* fp)  {  q\* head = NULL;  q\* tail = NULL;  char tempbuf[LINEBUF] = {0,};  int curPNUM = -1;  //Dummy 노드 생성  head = (q\*)malloc(sizeof(q));  head->line = 0;  head->next = NULL;  tail = head;  while (!feof(fp)) {  fgets(tempbuf, 1024, fp);  if (strstr(tempbuf, START\_SCHED)) { //초기 스케쥴러가 시작되어 PNUM을 알아낼 때 사용  sscanf(tempbuf, "scheduler\_test start[PNUM:%d]", &curPNUM);  }  else if (strstr(tempbuf, INIT\_PID)) { //프로세스가 가장 초기에 allocProc에서 생성될 때 tick 측정  int pid, ticks;  sscanf(tempbuf, "PID : %d, %d (0)", &pid, &ticks);  pidInfo[pid].create\_ticks = ticks;    }  else if (strstr(tempbuf, FIND\_PID)) { //프로세스가 스케쥴러가 호출될 때마다 갱신되는 정보  int pid, total\_ticks = 0;  int tmp1, tmp2, tmp3;  sscanf(tempbuf, "PID : %d, priority : %d, proc\_tick : %d ticks, total\_cpu\_usage : %d ticks, totalTicks : %d (2)",  &pid, &tmp1, &tmp2, &tmp3, &total\_ticks);  //가장 초기에 호출된 프로세스면 내부적으로 첫 스케쥴링 타이밍을 측정  if (pidInfo[pid].pid == 0) {  pidInfo[pid].pnum\_id = curPNUM;  pidInfo[pid].pid = pid;  pidInfo[pid].start\_ticks = total\_ticks;  pidInfo[pid].CallCount = 1;  }  else {  //스케쥴링이 끝나는 시점 측정 -> 실제로는 프로세스 종료시점을 측정해야하는데 일단 계속 업데이트  //++ 스케쥴러가 호출되는 횟수를 기록하는 변수 갱신  pidInfo[pid].end\_ticks = total\_ticks; //계속 갱신하면 마지막에 갱신된 값이 마지막 스케쥴링 tick  pidInfo[pid].CallCount++;  }  }  else if (strstr(tempbuf, DIED\_PID)) { //프로세스가 죽었을 때 T-Time, R -Time을 측정하기 위해 호출하는 부분  int pid, ticks;  int tmp1, tmp2, tmp3;  sscanf(tempbuf, "PID : %d, priority : %d, proc\_tick : %d ticks, total\_cpu\_usage : %d ticks, totalTicks : %d (3)",  &pid, &tmp1, &tmp2, &tmp3, &ticks);  pidInfo[pid].end\_ticks = ticks;  if (ticks != 0) { //ticks이 없다는건 일단 우리가 관심이 있는 프로세스는 아니라는 뜻  pidInfo[pid].turnAroundTime = pidInfo[pid].end\_ticks - pidInfo[pid].create\_ticks; //T-time 계산  pidInfo[pid].responseTime = pidInfo[pid].start\_ticks - pidInfo[pid].create\_ticks; //R-Time 계산  }    }  else if (strstr(tempbuf, END\_SCHED)) { //scheduler\_test 의 PNUM이 종료될 때마다 호출하는 한 프로세스 과정 전체의 ticks 측정  int sched\_end\_ticks;  sscanf(tempbuf, "end of scheduler\_test[PNUM>%d] : %d ticks", &curPNUM, &sched\_end\_ticks);  q\* newNode = (q\*)malloc(sizeof(q));  sprintf(newNode->lineBuf, "%d,%d", curPNUM, sched\_end\_ticks);  //해당 ticks들은 연결리스트로 가장 앞단에 추가하여 분석이 쉽게 함  newNode->line = tail->line+1; //라인수 정해줌  newNode->commaCount = 1;  newNode->next = NULL;  //연결리스트 재연결  tail->next = newNode;  tail = newNode;    //newNode->lineBuf  //end of scheduler\_test[PNUM:3] : 302 ticks  }  else if (strstr(tempbuf, PID\_TIMER)) {  int pid, s\_ticks, priority;  sscanf(tempbuf, "[SET] pid : %d, priority : %d, schedule\_ticks : %d", &pid, &priority, &s\_ticks);  pidInfo[pid].priority = priority;  pidInfo[pid].sched\_time = s\_ticks;  //[SET] pid : 4, schedule\_ticks : 591  }  if (feof(fp))  break;  }  return head->next;  } |