컴퓨터학부, 20190511, 배준형

# 기존 스케쥴러 분석 : xv6

|  |
| --- |
| xv6는 기본적으로 handmade Code로 init 프로세스가 생성된 후 (pid=1) 초기 부팅이 진행된다. 이후 커널 영역을 할당하고 H/W 내부 APIC로부터 올라오는 펄스 신호 (pulse = Timer Interrupt) 등을 처리하기 위해 커널(OS) 전용 IDT (Interrupt Description Table 및 Interrupt Vector Table) 를 만드는 등의 작업을 한다.  이후 shell 프로세스 (pid=2) 를 init.c 를 실행시켜 생성한 후 (proc.c : fork())  proc.c : scheduler() 함수에서 끊임없이 스케쥴링을 하게된다.  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 정책]  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.  프로세스가 스케쥴이 모종의 이유(Interrupt) 로 끝나게 되면 trap.c : trap(struct trapframe\* tf) 함수로 trap을 처리하게 되고 특정 trap (Timer Interrupt 등) 으로 proc.c : yield() 가 발생하게 되면 proc.c : sched() 스케쥴러 내부에서 vm.c : swtch(prev Context, Next Context); 으로 proc.c scheduler() 함수로 돌아오게된다.  이후 함수 switchkvm() 으로 커널 메모리영역을 다시 적재한다.  이때 Time Quantum( tq=1 Tick) 이 지나서 스케쥴링이 시작되었다고 본다.  스케쥴링을 진행하기 위해 내부에서 for문을 계속 돌며 spin 형태로 ptable에 존재하는 proc 배열(Vector) 에서 RUNNABLE 상태인 프로세스를 뽑는다.  즉, xv6 기본 스케쥴링은 1 Tick 마다 일종의 (정확하게는 ptable이 들어온 순서부터 할당되지 않고 unused 부분에 채워질 수 있어 아니지만) Round Robin 형태로 스케쥴링이 진행된다는 말이다.  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 호출된다.**  [xv6 스케쥴러 구현]  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.  그렇기 때문에 for(;;) 내부에서는 **acquire() 와 release() 함수가** 내부적으로 Call 되어 진행된다는 것을 알 수 있다.  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 함수와 **실제로 context를 교환하는 함수**이다.  switchuvm(p) 는 스케쥴링된 프로세스의 가상메모리와 페이지테이블을 할당하는 역할을 한다.  switchkvm() 는 커널모드로 진입한다. 인터럽트로 인하여 프로세스가 스케쥴링이 끝난다면 sched() 가 실행될 것이다. 이때 커널프로세스는 switchkvm 을 통해서 커널모드로 복귀함을 알 수 있다.  swtch() 부분은 실제로 Context 즉, 처리기(CPU) 가 뽑힌 프로세스의 Register, Memory, System Context를 교환하는 함수로, 실제로 이 때, 스케쥴링이 끝나고 프로세스가 실행되게 된다.  세번째는 스케쥴러 함수 내부에서는 이제 sti(); 를 설정해두어 프로세서가 Interrupt가 허용되도록 설정해두었다. |

[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.    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 코드 : 수작업으로 ticks 수 등을 세면서 발생하는 인위적인 오류를 막고자 제작.>   |  | | --- | | #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <string.h>  #define FILENAME "test.txt"  #define EXTRACTFILE "middlefile.txt"  #define FIND "end of scheduler\_test"  int main(void)  {  FILE\* fp1, \*fp2;  char filebuf[1024];  if ((fp1 = fopen (FILENAME,"w+")) == NULL) {  fprintf(stderr, "fopen error : %s\n", FILENAME);  exit(0);  }  system("make qemu > test.txt");  sleep(2);  fprintf(fp1, "scheduler\_test\n");  char filename[1024];  printf("파일 이름 입력 : ");  scanf("%s", filename);  while(fgetc(stdin) != '\n');  strcat(filename, ".csv");  if ((fp2 = fopen (filename,"w")) == NULL) {  fprintf(stderr, "fopen error : %s\n", EXTRACTFILE);  exit(0);  }  while(fgets(filebuf, 1024, fp1) != NULL) {    if (feof(fp1))  break;  else {  char \*f\_ptr;  if (strstr(filebuf, FIND)) {  char pnum[20] = {0,}, ticks[20] = {0,};  char \*s = filebuf + strlen(FIND);  for (; \*s != '\0'; s++) {  int k;  if (\*s == '[') {  s++;  for(k=0;\*s != ']';s++,k++) {  pnum[k] = \*s;  }  }  if (\*s == ':') {  s++;  for(k=0;\*s != 't';s++,k++) {  ticks[k] = \*s;  }  break;  }  }  fprintf(fp2, "%s, %s\n", pnum+5, ticks+1);  }  }  }    fclose(fp1);  fclose(fp2);  exit(0);  } |   **[A. CPU위주 프로세스만 존재하는 프로그램에서의 성능차이.]**  - 실제로 위의 예제 프로그램으로 성능차이를 분석해본 결과  SSU\_scheduler 성능이 생각보다 ticks 수가 현저히 많은 점을 알 수 있다.  이는 단순배열에서 RR (Round Robin) 방식으로 스케쥴링을 하는 것보다 연결리스트를 순회하며 priority를 계산하는 과정에서 발생하는 overhead가 크기 때문으로 생각된다. |

[**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

|  |
| --- |
| [2가지 형태로 구현]   * 본의아니게 RunQueue 형태를 2형태로 해석하여 2가지 방향으로 구현해보았습니다.. 어느 하나를 선택하기엔 구현한 점이 아깝기도 하여 두 구현 방식 모두 설명하겠습니다. * 기본적으로B구현이 함수가 더 많으며 가독성을 위해 B구현, A구현 기능이 동일한 함수는 이름이 같도록 설계하였음 * **기본적으로 ORIGIN=1 매크로 설정이 되어있으며, ORIGIN=1 일 경우 (RunQueue 당 1개의 큐) 기본형(A형) 이 실행되며, ORIGIN=0 일 시 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) 2. (응용형) RunQueue 내부에 4개의 Queue가 따로 구현되어 있고 해당 Queue들의 프로세스 총합을 관리하는 middleCount를 통해 Queue 접근을 최소화하는 기법   **[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  <proc.c : RunQuene 삽입/삭제 요소를 제외한 구현>   * proc.c : scheduler() 함수 내부에서 호출되는 함수들   **void updateQueue();** // 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();**  **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);**  -> 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.   <시스템 콜 구현>   1. 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)  > 사용자로부터 priority를 입력받아 현 프로세스의 priority를 해당 값으로 변경 -> 60ticks이 지나면 해당 프로세스는 ticks대로 RunQueue에 삽입되게됨.  <사용자 프로그램 : scheduler\_test.c 구현>   1. 스케쥴링 확인 프로그램 : scheduler\_test  * 앞 전과제와 동일하게 Makefile을 수정하고 + 파일 구현   해당 파일에서 scheduler\_func() 를 호출하여 PNUM만큼 fork() 후 set\_sche\_info() 를 호출함.   * 주의할 점, 그냥 fork()를 중첩해서 달면 2^PNUM 만큼 fork가 발생하므로 pid==0 이 자식프로세스임을 이요하여 구현. * 초기 부모프로세스는 wait시스템콜을 통해 자식 프로세스 수만큼 기다림.   [공통구현방식]   * **Update Priority : 우선순위 갱신** * 우선순위 갱신 방식은 trap에서 프로세스가 유효할 때 TIMER\_INTERRUPT 가 발생할 때마다 trap.c 내부의 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 * 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. |

# 결과

|  |
| --- |
| * set\_prio.. |

# 소스코드

## proc.c

|  |
| --- |
|  |

## trap.c