|  |
| --- |
| Operating Systems  Spring, 2023  School of Software, CAU  **Project #2**  **- Page Allocation and Task Scheduling –**  **[프로젝트 보고서]**  학번: 20192513  이름: 오 양 호 |

**1. 서론**

본 보고서에서는 페이지 할당자와 스케줄러의 구현 방법과 결과에 대해 기술한다.

* 1. **Page Allocation**

PintOS에 구현된 Page Allocation은 palloc.c 파일의 palloc\_get\_multiple 함수에 구현되어 있다. palloc\_get\_multiple 함수는 파라미터로 플래그와 페이지 개수를 받는다. 플래그는 유저풀과 커널풀 중 어떤 풀의 페이지를 할당할 것인지 결정하고, 해당 풀에 요청한 페이지 개수만큼 페이지를 할당한다. 할당한 페이지는 페이지가 사용 중인지 아닌지 확인하기 위한 비트맵으로써 각 풀이 하나씩 가지고 있다. palloc\_get\_multiple에서 First Fit 방식으로 페이지를 할당하는 것은 bitmap\_scan\_and\_flip이라는 함수의 작동 방식에 의한 것이다. bitmap\_scan\_and\_flip은 파라미터로 네 가지 인자를 받는다. 하나는 어떤 풀의 비트맵을 바꿀 것인지, 두 번째는 해당 비트맵의 어느 인덱스부터 스캔을 진행할 것인지, 세 번째는 페이지를 몇 개 할당할 것인지, 네 번째는 해당 영역을 0으로 바꿀 것인지 또는 1로 바꿀 것인지를 정하는 인자이다. 아무것도 하지 않은 상태의 palloc.c 파일은 bitmap\_scan\_and\_flip 함수에 pool->used\_map, 0, page\_춧, false를 인자로 넣어 사용하고 있으므로, palloc\_get\_multiple이 받은 플래그로 유저풀과 커널풀 중 어떤 풀을 선택할지 결정하고 비트맵의 처음부터 페이지 카운트까지 쭉 스캔을 한다. 스캔 도중 인자로 받은 페이지 개수만큼의 자리가 존재하면 그 부분에 할당한다. 이것은 First Fit 방식이다. 따라서 주어진 과제를 수행하기 위해서는 이 부분을 Buddy System 알고리즘 규칙을 따르도록 수정하면 된다. 또한 페이지를 할당 해제하는 것도 비슷한 방식으로 palloc\_free\_multiple함수에서 일어나므로 이 부분도 수정이 필요하다.

* 1. **Task Scheduling**

PintOS에서 기존에 구현된 Round Robin 방식의 스케줄링을 이해하기 위해서는 init.c의 메인 함수가 어떻게 동작하는지 알 필요가 있었다. init.c의 메인함수가 동작하는 과정 중 기존 RR 스케줄을 MFQ 스케줄러로 교체하기 위해서 가장 중요한 부분은 timer\_init 함수를 호출하는 부분이었다. timer\_init 함수는 하드웨어 인터럽트 핸들러로 timer\_interrupt를 등록하고 이후 시스템이 부팅되고 동작하는 동안 매 틱마다 timer\_interrupt함수를 호출하게 한다. timer\_interrupt함수는 호출될 때마다 틱을 1 증가시키고 thread\_tick 함수를 호출한다. 이어서 thread\_tick 함수에서 thread\_ticks이 정의해둔 TIME\_SLICE에 다다르면 다음 스레드에게 CPU를 양보하는 intr\_yield\_on\_return함수를 호출한다. (intr\_yield\_on\_return함수는 현재 스레드를 다른 스레드로 context switching할 것을 지시하는 역할을 한다.) 따라서 thread\_tick 함수가 RR 방식의 스케줄링을 하는 역할을 담당하고 있다. 또한 init.c의 메인함수는 thread\_start함수를 호출하는데 이 함수는 interrupt를 활성하해서 timer\_init함수가 시간 interrupt로서 계속 틱을 세도록 하고, 이는 preemptive 스케줄링이 가능하도록 돕는다. 기존의 RR 스케줄링 방식을 MFQ 방식으로 교체하기 위해서는 먼저, 기존에 사용하던 ready\_list(ready queue 역할)를 4개로 늘리고 스레드가 과제의 요구사항에 맞게 각 리스트에 들어가고 나오도록 해야한다.

**2. 본론**

본론에서는 구현 방법에 대해 상세히 기술한다.

* 1. **Page Allocation**

먼저, Buddy System을 구현하기 위해 페이지 개수를 입력하면 가장 가까운 2의 거듭제곱를 찾는 함수를 구현했다. 이름은 powerOfTwo로 정했다. 구현은 입력 받은 정수에 가장 가까운 2의 거듭제곱을 찾을 때까지 시프트 연산을 하는 방식으로 구현했다. 그후에 palloc\_get\_multiple함수 내의 bitmap\_scan\_and\_flip 함수를 대체하기 위한 bitmap\_buddy\_allocate 함수를 palloc.c 파일 내에 구현하였다. 이 함수는 파라미터로 비트맵 포인터와 페이지 개수를 받는다. bitmap.c 파일 내에 정의돼 있는 bitmap\_size 함수로 비트맵의 전체 사이즈를 구한다. 이는 사용가능한 페이지 수와 같을 것이다. 그리고 Buddy System은 전체 사이즈가 2의 거듭제곱이어야 하므로 max\_size로 전체 사이즈에 가장 가까운 2의 거듭제곱을 찾아 2를 나눠준다. 예를 들면 356개의 사용가능한 페이지가 있으면 Buddy System을 구현하기 위해 256페이지를 max\_size로 정하고 여기서부터 Buddy System을 적용하여 2의 거듭제곱으로 블록을 나눠 할당하겠다는 이야기이다. Buddy System에서 요청한 페이지에 가장 가까운 2의 거듭제곱을 저장할 변수 buddy\_size를 정의하고 그 값을 할당한다. bitmap\_buddy\_allocate의 구현은 bitmap.c에 정의된 bitmap\_none이라는 함수를 이용하여 비트맵의 첫 번째 부분부터 buddy\_size 만큼의 공간이 있는지 탐색한다. 만약 스캔 중에 단 하나라도 1이 있다면 그 공간을 들어갈 수 없다고 판단하고, buddy\_size만큼 떨어진 지점에서 다시 탐색한다. 이런 방식으로 하면 Buddy System의 방식으로 페이지를 할당할 수 있다. 기존의 bitmap\_scan\_and\_flip함수를 대체하는 함수이므로 할당에 성공할 시 비트맵에 할당했다고 표시해주기 위해 bitmap\_set\_multiple함수로 0을 1로 바꿔주도록 했고, 할당에 성공하면 성공한 지점의 인덱스를 반환하고 할당에 실패하면 BITMAP\_ERROR를 반환하도록 구현했다. 본 과제에 대한 테스트는 유저풀에 페이지를 할당하는 것으로 진행했는데, 할당할 수 있는 페이지가 없을 때 그것을 확인하기 위해 palloc\_get\_multiple함수에 flags & PAL\_USER가 참일 때도 PANIC 상태에 빠지도록 수정했다. 페이지를 할당 해제하는 것은 기존 방식은 요청받은 페이지 개수만큼만 페이지 할당 해제를 하도록 되어있는데 그 부분을 buddy\_size만큼 해제하도록 수정했다. 마지막으로 과제에서 요구했던 palloc\_get\_status 함수는 단순히 플래그로 받은 풀에 해당하는 비트맵을 쭉 스캔하면서 printf로 출력했다. 한 줄에 32비트씩 출력해야 하므로 그것에 대한 것도 if문으로 처리했다.

* 1. **Page Allocation Test**

페이지 할당 방식이 Buddy System으로 잘 작동하는지 테스트하기 위해 pa.c에 직접 palloc\_get\_multiple함수를 사용하였다. 페이지 할당은 유저풀에서 진행했고, numA부터 numE까지 정수형 변수에 요청할 페이지 개수를 할당하여 palloc\_get\_multiple에 파라미터로 넘겨줬다. 페이지 요청과 해제는 교재에 나와있는 Buddy System을 설명하는 그림과 동일하게 진행했다. 다만 교재에서는 max\_size가 1M인 상황에서 설명했고 과제에서 사용가능한 페이지 수에 가장 가까운 2의 거듭제곱이 256페이지라 교재의 예제에서 4를 나눈 값으로 설정했다. 프로세스가 페이지를 요청할 때마다 페이지가 할당되는 것을 보이기 위해 palloc\_get\_multiple함수를 호출할 때마다 구현했던 palloc\_get\_status 함수를 같이 호출해서 잘 되고 있음을 보였다. 또한 페이지 요청과 프로세스 해제를 할 때마다 해당 작업이 일어나고 있음을 printf로 출력해서 보기 쉽게 했다.

**2.1 Task Scheduling**

MFQ 스케줄링을 구현하기 위해 실행했던 순서는

다음과 같다:

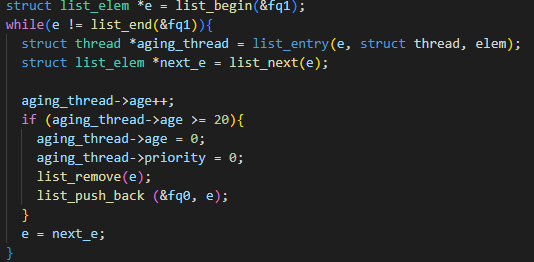
1. 4개의 Multi-level queue 구현
2. Aging 구현
3. Queue마다 time slice를 다르게 적용

이러한 순서로 구현을 진행했다.

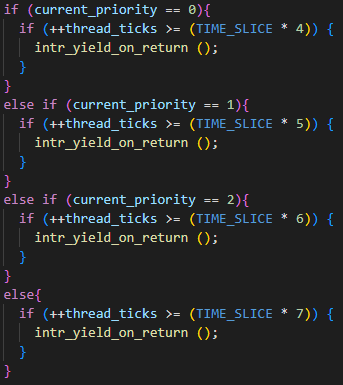
먼저 4개의 Multi-level queue를 thread.c에 선언했다. 이름은 fq0, fq1, fq2, fq3로 했고 과제의 설명과 동일하게 fq0를 가장 높은 우선순위 큐로, fq3를 가장 낮은 우선순위 큐로 정했다. ready\_list를 대체할 자료구조이기 때문에 기존에 ready\_list에서 thread의 elem 필드를 받던 함수를 모두 수정해줬다. 수정한 함수의 목록은 다음과 같다:

1. next\_thread\_to\_run()
2. thread\_yield()
3. thread\_unblock()
4. thread\_init()

각 함수를 어떻게 수정했는지 설명하겠다. next\_thread\_to\_run은 기존에 ready\_list에서 스레드를 꺼내 schedule함수에게 전달하는 역할을 하고 있었다. 간단히 말하자면 ready\_list는 이미 Round Robin 방식으로 스레드가 들어갔다 나오고 있고, next\_thread\_to\_run은 스케줄링 방식이 어떻게 되던 그저 ready\_list에서 꺼내기만 한다. 따라서 기존에 ready\_list에서만 꺼내던 방식을 가장 높은 우선순위 큐인 fq0부터 스레드를 꺼내보고 해당 큐가 비었으면 fq1에서 찾고, 이런 식으로 높은 우선순위 큐를 먼저 보고 비어 있을 때 하위 우선순위 큐에서 꺼내보도록 했다. 다음은 thread\_yield 함수를 수정했는데, 이 함수는 지금 실행중인 스레드를 ready\_list에 넣는 역할을 하고 있었다. 과제의 요구사항에서 MFQ 방식을 구현하기 위해 현재 실행중인 스레드는 다시 원래 큐로 들어가는 것이 아니라 한 단계 낮은 우선순위의 리스트로 들어가야 하므로 switch 문으로 현재 실행중인 스레드가 우선순위를 한 단계 낮추고 우선순위에 해당하는 큐로 들어가도록 수정했다. 다음 thread\_unblock 함수는 블록 상태에 있던 스레드가 블록 해제됐을 때 다시 ready\_list에 들어가도록 하고 있었는데, 과제의 요구사항에 블록 상태에 있던 스레드가 블록 해제된다면 age를 0으로 초기화하고 우선순위를 한 단계 높여야 하므로 thread\_yield를 수정했던 방식과 유사하게 switch문으로 수정했다. 또한 thread\_creat함수가 스레드를 생성할 때 unblock을 한 번 해주는 것 때문에 수정한 thread\_unblock 함수가 모든 스레드의 우선순위를 1 낮추고 시작하게 되어서 이 부분은 thread\_unblock\_first라는 함수를 따로 만들었다. 이 함수는 unblock 될 때 상위큐로 올라가는 부분을 삭제하고 원래 큐로 들어가도록 구현했으며 이 함수를 thread\_create에 thread\_unblock 대신 사용했다. 마지막으로 thread\_init에서 기존에 ready\_list를 초기화하는 작업이 일어나고 있었는데 여기에 fq0~fq3 리스트를 초기화하도록 수정했다. 여기까지 구현하면 Multi-level queue로 우선순위에 스레드의 우선순위에 따라 해당하는 큐에 스레드를 넣고 한 번 실행된 스레드는 우선순위가 한 단계 떨어지고, 블록됐던 스레드는 우선순위가 올라가게 되어 I/O-bound 스레드과 Processor-bound 스레드가 공평하게 CPU를 사용할 수 있을 것이다. 두번째 목표로 Aging을 구현했는데 현재 실행중인 스레드의 하위 큐에 있는 모든 스레드들의 age가 매 틱마다 1씩 증가해야 한다. init.c 파일과 timer.c 그리고 interrupt.c를 읽고 나면 매틱마다 thread\_tick함수가 호출된다는 것을 알 수 있다. 따라서 일단 thread 구조체에 age 필드를 추가하고, 매 틱마다 age를 증가시키기 위해 thread\_tick 함수에 Aging 구현을 추가했다. 구현 방식을 설명하자면, 실행 중인 스레드보다 낮은 우선순위의 스레드들이 모두 age가 오를 수 있도록 thread\_tick 함수에 반복문을 통해서 하위 우선순위 큐를 순회하며 각 스레드마다 직접 age++를 해줬다. 현재 실행 중인 스레드의 우선순위가 0이면 fq1, fq2, fq3에 있는 스레드의 age가 올라야 하므로 반복문 세 개를 넣었고, 현재 실행 중인 스레드의 우선순위가 1이면 f2, f3에 있는 스레드의 age가 올라야 하므로 반복문 두 개를 넣고, 이런 식으로 하드코딩을 했다. 보고서에 소스코드를 직접 넣으면 매우 보기 힘드므로 짧게 코드를 보이자면 다음과 같다:



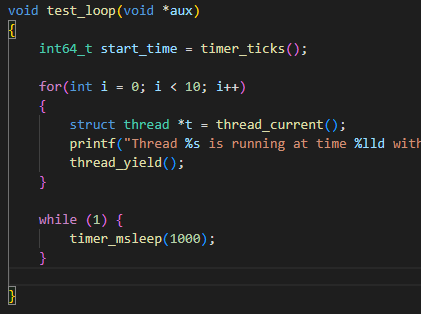
반복문으로 age를 올리면서 해당 스레드의 age가 20을 넘는지도 확인하도록 했다. 만약 20을 넘는다면 age를 0으로 초기화하고 우선순위를 올리고 현재 큐에서 삭제하고 상위 큐에 스레드를 삽입했다. 현재 큐에서 스레드를 삭제하는 과정에서 스레드에 대한 참조를 잃어버릴 수 있으므로 e = next\_e로 그 부분을 해결했다. 마지막으로 큐마다 time slice가 다르게 적용되는 것도 thread\_tick 함수 내에서 구현했다. 먼저 4로 정의돼 있던 TIME\_SLICE를 1로 수정하고, 현재 실행 중인 스레드의 우선순위를 확인한 후 TIME\_SLICE \* (주어진 시간)만큼만 실행된 후 양보하도록 구현했다.



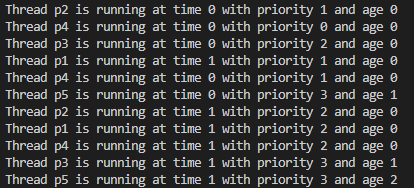
과제의 요구사항처럼 FQ0, FQ1, FQ2, FQ4를 각각 4, 5, 6, 7틱으로 설정했다. 이렇게 하면 MFQ의 구현이 모두 끝났다. 이 스케줄링 방식은 우선순위대로 해당하는 큐에 들어가고, 실행되고 나면 우선순위를 한 단계 낮추며 블록상태에서 풀려나면 age가 0으로 초기화 된 후 한 단계 상위 우선순위로 변한다. 또한 하위 큐들의 Starvation을 예방하기 위해 매 틱마다 실행되고 있는 스레드의 하위 우선순위 스레드들이 age를 증가시켜 특정 age가 넘으면 상위 우선순위로 올 수 있게 하여 Starvation을 예방한다. 마지막으로 큐마다 다른 Time Slice를 적용하여 공정성을 더했다. (근데 보통 높은 우선순위의 큐가 더 많은 시간을 보장받는 것으로 배웠는데 과제의 요구사항은 반대라서 의아했다.)

**2.2 Task Scheduling Test**

MFQ를 구현하는 것보다 테스트를 하는 것이 더 어려웠다. 특히 Aging과 Time Slice가 제대로 작동되는지 확인하기가 상당히 어려웠다. 때문에 test\_loop는 단순히 test\_loop가 돌 때 해당 스레드에서 반복문으로 여러 번 해당 스레드의 상태를 출력하도록 하고, thread\_yield 함수를 통해 time slice가 다 끝나지 않아도 스레드를 교체하여 출력 결과를 보고 age가 증가하고 있는지, 우선순위가 변경되고 있는지, 변경한 스케줄링 방식대로 스레드 순서가 변화하는지를 확인하게 했다. 다음은 test\_loop 소스코드의 일부분이다.



그리고 다음은 실행 결과의 일부분이다:



스레드가 시작한 시점과 우선순위, age를 출력하도록 했고 age와 우선순위가 변화하며, 스레드가 공정하게 스케줄링 됨을 확인할 수 있었다.

**3. 결론**

교수님께서 저번 과제보다 쉽게 내주실 줄 알았지만 큰 착각이었다. 저번 과제도 내 능력의 최대치에 가까운 난이도라고 생각했는데 이번 과제는 더 어려웠다. 과제를 해결하면서 PintOS가 어떤 방식으로 스케줄링을 하고 있었는지 각각의 함수가 어떤 역할을 분담하고 서로 어떻게 돕는지 파악하며 코드를 파악하는 실력이 상승한 것 같다. 이외에도 MFQ 스케줄러를 구현하기 위해 어느 부분을 수정할지 고민하고 어떤 방식으로 디버깅하고 검증할까 고민하면서 더 많은 지식을 얻게 된 것 같아 즐거웠다.