|  |
| --- |
| **Report** |

|  |  |  |
| --- | --- | --- |
| **학과** | 컴퓨터공학부 | |
|  |  |  |
| **과목** | 운영체제 | |
|  |  |  |
| **조 번호** | 15조 | |
|  |  |  |
| **성 명** | 임기찬, 천영재, 문태현 | |
|  |  |  |
| **제출일** | 2018.06.09 | |
|  |  |  |
| **담당 교수** | 김성조 교수님 | |
|  |  |  |



|  |
| --- |
| **스케줄링 알고리즘**  **RR 스케줄링 분석 리포트** |

1. Pintos 의 타이머 인터럽트는 1초에 100회 발생한다. 준비 큐에 있다가 Dispatcher에 의해 실행이 시작된 스레드는 다른 스레드가 실행되기 전까지 몇 초 동안 실행되는가?

if (++thread\_ticks >= TIME\_SLICE)

intr\_yield\_on\_return ();

time slice는 4라고 정의되어 있기 때문에, 스레드 틱은 0에서 시작. 총 4회 진행된다.

1. RR 스케줄러는 어떤 자료구조를 이용하여 준비 큐에 머무르고 있는 스레드를 관리하는가?

링크드 리스트를 사용한다. 그 후, 그 포인터들을 이용하여 list\_end, list\_next 등의 함수를 사용하여 관리한다.

1. threads/thread.c 소스 코드에서 어떤 함수들이 질문 2에서 언급된 자료구조에 접근하는가?

Thread\_init 함수 ->처음 만들어지는 스레드를 초기화 시켜준다.

Thread\_unblock 함수 -> block된 스레드를 unblock하여 ready list에 넣어준다.

Thread\_wakeup-> sleep list에 있던 하나의 스레드를 깨워 ready list에 넣어준다.

Thread\_yield-> 현재 진행되는 스레드를 멈추고 ready list에 넣은 후, 다른 스레드를 부름.

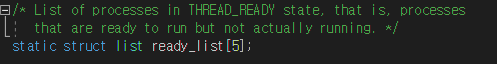
Next\_tread\_to\_run-> 다음 진행될 스레드를 ready list에서 불러온 뒤 리턴한다.

1. RR 스케줄러는 스레드 우선순위를 사용하는가?

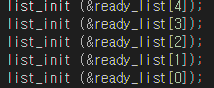
사용하지 않는다. Priority가 구현되어 있고, 스레드 구조에 들어가지만, priority를 이용하여 구현하는 것이 아닌, 그냥 시간의 틱에 따라서 디스패치되는 것만 이용한다. 결국 FIFO를 만족시킨다.

|  |
| --- |
| **스케줄링 알고리즘**  **MFQ 스케줄러 구현 리포트** |

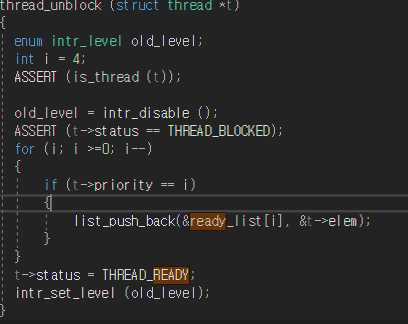
1. threads/thread.c 와 threads/thread.h에서 RR스케줄러를 교체하기 위해 수정 및 추가한 부분.



리스트를 배열로 5개 생성. 총 5개의 우선순위를 가지고, 높은 우선순위의 배열 값이 먼저 실행된다.



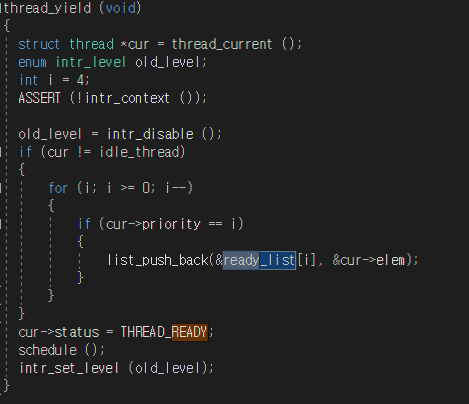
5개의 배열이므로, 5개를 전부 초기화.



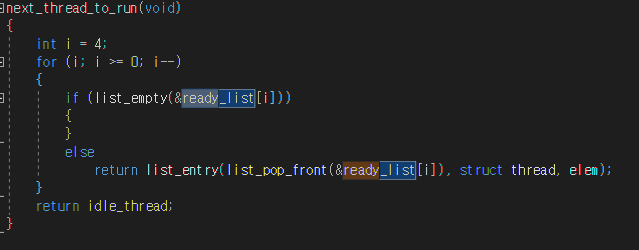
Thread\_unblock 함수에서는 우선순위에 따라 리스트에 넣어주는 포문과 이프문 작성.

우선순위와 배열의 번호는 같다. 그러므로 for문을 사용하여, 배열에 push해주는 것이 가능하다.

|  |
| --- |
| **스케줄링 알고리즘**  **MFQ 스케줄러 구현 리포트** |



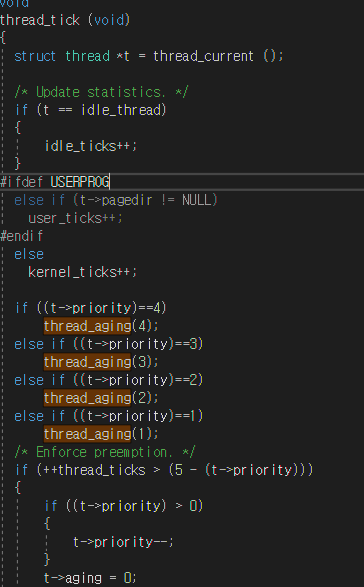
Thread\_yield함수에서도 마찬가지로 작성. Thread\_unblock 과 수정된부분에 차이는 거의 없다.



Next\_thread\_to\_run 함수에서는 비어 있지 않은 최고 우선 순위 값 리스트 부터 하나 하나 리턴하게 바꿈. 4가 비어있다면 3에서 pop해서 리턴, 그다음 3이 비어있으면 2에서 pop해서 리턴, 반복하여

|  |
| --- |
| **스케줄링 알고리즘**  **MFQ 스케줄러 구현 리포트** |

모두 비어있다면 idle\_thread를 리턴해 준다.



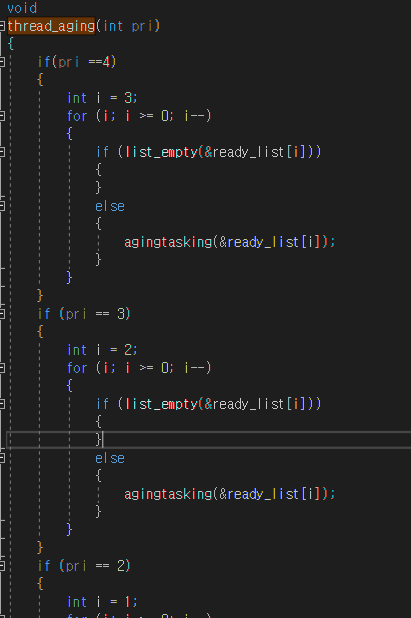
Thread\_tick 함수에서는 aging을 구현하기 위해 만든 aging 함수를 우선순위에 따라 호출하고, 또한 tick이 (5-(t->priority))+1만큼 실행되게 만듬. (thread tick의 시작은 0이기 때문) 또한 할당된 시간 틱을 모두 실행하면 aging을 초기화.

2. 준비 큐에 머무르고 있는 스레드를 관리하기 위해 사용한 자료구조.

마찬가지로 리스트형태. 리스트형태의 5개의 배열로 만들어 각각 우선순위를 배분하였다.

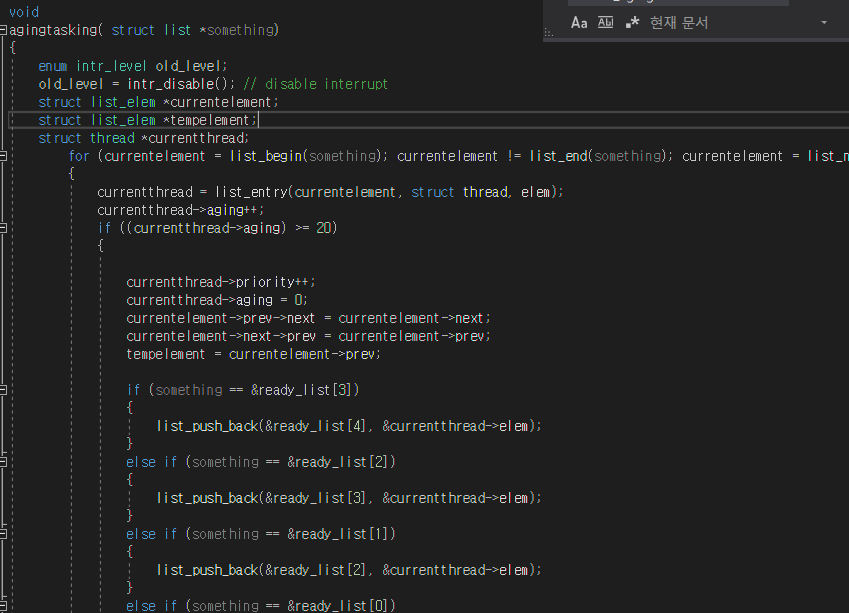
|  |
| --- |
| **스케줄링 알고리즘**  **MFQ 스케줄러 구현 리포트** |

3. 스레드 aging 구현 방법.



Aging을 구현하기 위해 만든 thread\_aging 함수. 인자로 우선순위를 int값으로 받고, 그 인트 값보다 낮은 우선순위를 가지고 있는 list들의 agingtasking을 실행한다.

|  |
| --- |
| **스케줄링 알고리즘**  **MFQ 스케줄러 구현 리포트** |



List 포인터를 인자값으로 받는 함수. 그 리스트를 처음부터 돌면서 전부 aging을 +1 시켜주고, 만약 20이상의 aging이 된다면 그 순간 그 리스트에서 제거하고, 한단계 위의 리스트안에 가장 뒷부분에 넣는다.

1. MFQ 스케쥴러의 동작을 확인하기 위한 테스트 방법.

기본적으로 제공되는 scheduling C를 약간 수정한다. 틱의 수정이아닌, 우선순위값을 PRI\_DEFAULT 가아닌, 0~4사이의 정수로 제공하게 바꾼다. 그 후 실행화면에서 총 몇틱이 실행되는지, 그리고 몇 개가, 그리고 그보다 낮은 우선수위에 있는 스레드가. 제대로 에이징되서 올라오는지 확인한다. 에이징의 확인방법은 맨처음 넣은 PRI가 실행되는 틱보다 더 적은 수의 틱이 실행되면 aging이 된 것으로 확인한다.

|  |
| --- |
| **메모리 할당**  **메모리/페이지 할당 분석 리포트** |

1. Pintos 메모리 시스템의 기본 페이지 크기는 몇 바이트인가? 4Kbytes

palloc.c에서 PGSIZE를 봤고, 이게 구현되어 있는 vaddr.h를 찾아갔더니 PGSIZE = (1<<PGBITS)였는데 이때 PGBITS = 12여서 offset이 12비트를 사용함을 알 수 있었다. 따라서 페이지의 크기는 4K(2^12)bytes가 된다.

1. Threads/malloc.c에서 기본 페이지보다 작은 크기의 메모리 영역을 할당하기 위해 어떤 방법이 사용되고 있는가?

16, 32, 64, 128, …, 1024 Byte 크기의 Descriptor를 먼저 생성한다.

그리고 malloc(size\_t size)을 할 때 요청된 size를 만족하는 제일 작은 크기의 Descriptor에 할당합니다. 이때 Descriptor는 사용 가능한 블록 목록을 유지 하는데 사용 가능한 블록 목록이 비어 있지 않으면 해당 블록 중 하나를 사용합니다.

그렇지 않으면, "arena"라고 불리는 새로운 메모리 페이지가 page allocator로부터 얻어진다. (사용할 수 있는 페이지가 없다면, malloc()은 null 포인터를 반환한다). 새로운 arena는 블록으로 분할되며, 모두 블록의 Descriptor의 사용 가능한 블록 목록에 추가됩니다. 그런 다음 새로운 블록 중 하나를 반환합니다.

블록을 해제하면 Descriptor의 사용 가능한 블록 목록에 블록을 추가합니다. 그러나 블록이 있던 arena에 사용중인 블록이 없다면, 우리는 사용 가능한 블록 목록에서 모든 arena 블록을 제거하고 arena를 page allocator에게 돌려줍니다.

이 일련의 과정을 통하여 2KB보다 큰 블록을 처리 할 수는 없다. 왜냐하면 블록이 너무 커서 한 페이지에 하나의 Descriptor가 들어갈 수 없기 때문입니다. 우리는 page allocator로 인접한 페이지를 할당하고 할당 된 블록의 arena header 시작 부분에 할당 크기를 붙임으로써 이를 처리합니다.

|  |
| --- |
| **메모리 할당**  **메모리/페이지 할당 분석 리포트** |

1. Threads/palloc.c와 lib/kernel/bitmap.c 내의 어떤 함수에 페이지 할당 알고리즘(First Fit)이 구현되어 있는가?

palloc.c에서 palloc\_get\_page(enum palloc\_flags flags)함수를 보면 palloc\_get\_multiple (flags, 1)을 반환한다. 이 때 page 1개를 할당하기 위해서 page\_cnt를 1로 설정한다.

그래서 palloc\_get\_multiple (enum palloc\_flags flags, size\_t page\_cnt)을 보면 pages를 pool->base + PGSIZE \* page\_idx로 만들고 이를 반환한다. page\_idx는 bitmap\_scan\_and\_flip (pool->used\_map, 0, page\_cnt, false)을 호출해서 찾는다

이를 찾는 과정에서는 bitmap\_scan\_and\_flip (struct bitmap \*b, size\_t start, size\_t cnt, bool value)을 보게 된다. 여기서 반환되는 idx는 bitmap\_scan (b, start, cnt, value)을 통해서 찾는다. 이 경우 찾은 사용 가능한 페이지의 idx에 다른 페이지가 할당되지 않게 하기위해 bitmap\_set\_multiple (b, idx, cnt, !value)을 해준다.

그리고 bitmap\_scan (const struct bitmap \*b, size\_t start, size\_t cnt, bool value)를 보면 bitmap b를 처음부터 끝까지 검색하는데 만약 연속된 cnt개수 만큼의 사용 가능한 페이지를 최초로 찾았을 때 이 index를 반환한다.

|  |
| --- |
| **메모리 할당**  **메모리/페이지 할당 알고리즘 구현 리포트** |

1. threads/palloc.c와 lib/kernel/bitmap.c에서 페이지 할당 알고리즘 구현을 위해 수정 및 추가한 부분

pallocator를 가져와서 사용하기 위해서 bitmap.c에 아랫 줄 추가(7 line)

#include "threads/palloc.h"

next fit을 위해 reference\_bit 추가(26 line)

size\_t reference\_bit = 0;

bitmap\_scan 수정 및 추가(303~373 line)

bitmap\_scan(const struct bitmap \*b, size\_t start, size\_t cnt, bool value)

{

ASSERT(b != NULL);

ASSERT(start <= b->bit\_cnt);

if (cnt <= b->bit\_cnt)

{

size\_t last;

size\_t i, j, k = b->bit\_cnt + 1, min\_block = 9999;

if (pallocator == 0) {

last = b->bit\_cnt - cnt;

for (i = start; i <= last; i++)

if (!bitmap\_contains(b, i, cnt, !value))

return i;

}

else if (pallocator == 1) {

start = reference\_bit;

last = b->bit\_cnt - cnt;

for (i = start; i <= last; i++)

{

if (!bitmap\_contains(b, i, cnt, !value)) {

reference\_bit = i + cnt;

return i;

}

}

for (i = 0; i<start; i++)

{

if (!bitmap\_contains(b, i, cnt, !value)) {

reference\_bit = i + cnt;

return i;

}

}

}

else if (pallocator == 2) {

last = b->bit\_cnt - cnt;

|  |
| --- |
| **메모리 할당**  **메모리/페이지 할당 알고리즘 구현 리포트** |

for (i = start; i <= last; i++) {

if (!bitmap\_contains(b, i, cnt, !value))

{

j = i + cnt;

for (j = i + cnt; j < b->bit\_cnt; j++) {

if (bitmap\_contains(b, j, 1, !value)) {

break;

}

}

if ((j - i - cnt) < min\_block) {

k = i;

min\_block = j - i - cnt;

}

i = j;

}

}

if (k <= b->bit\_cnt) {

return k;

}

}

else if (pallocator == 3) {

for (i = 1; i <= b->bit\_cnt && i < cnt; i = i \* 2) {

}

last = b->bit\_cnt - i;

for (j = 0; j <= last; j += i) {

if (!bitmap\_contains(b, j, i, !value)) {

printf("%d\n", j);

return j;

}

}

}

}

return BITMAP\_ERROR;

}

Buddy System을 위해서 bitmap\_set\_multiple 수정 및 추가(220~236 line)

/\* Sets the CNT bits starting at START in B to VALUE. \*/

void

bitmap\_set\_multiple (struct bitmap \*b, size\_t start, size\_t cnt, bool value)

{

size\_t i;

ASSERT (b != NULL);

ASSERT (start <= b->bit\_cnt);

ASSERT (start + cnt <= b->bit\_cnt);

if (pallocator == 3) {

for (i = 1; i <= b->bit\_cnt && i < cnt; i = i \* 2) {

|  |
| --- |
| **메모리 할당**  **메모리/페이지 할당 알고리즘 구현 리포트** |

}

cnt = i;

}

for (i = 0; i < cnt && i < b->bit\_cnt; i++)

bitmap\_set (b, start + i, value);

}

1. 각 페이지 할당 알고리즘의 구현 방법
   1. Next fit

최초의 reference\_bit는 0으로 설정된다.

그리고 page를 할당할 때 먼저 reference\_bit을 가져와서 여기서부터 스캔을 시작한다. 만약 last bit가 되기전에 할당 될 수 있는 공간이 있으면 할당한다. 만약 그렇지 않으면 처음부터 reference\_bit의 위치까지 스캔을 해서 할당 될 수 있는 공간이 있으면 할당하고 없으면 BITMAP\_ERROR가 발생한다.

* 1. Best fit

처음부터 비어 있는 공간들을 스캔하는데, 사용 가능한 공간들 중에서 가장 작은 것을 선택한다. 먼저 사용 가능한 공간을 찾으면, 그 다음의 사용할 수 없는 bit를 찾는다. 그래서 이 공간의 크기를 측정하고, 만약 이게 그 전에 선택된 공간보다 작으면 이 공간을 선택한다. 이 과정을 last bit까지 반복하고나서 선택된 공간에 page를 할당한다.

* 1. Buddy system

먼저 할당해야 하는 page의 크기를 k라고 할 때, 2i-1 < k < 2i 인 i를 찾는다. 그리고 cnt가 아닌 2i 크기 단위로 스캔을 해서 공간을 선택해서 할당한다. 0~2i, 2i~(2i \* 2), (2i\* 2)~(2i \* 3), … 단위로 할당 할 수 있는지 스캔함으로써 bitmap을 계속해서 1/2로 나누지 않고 buddy system을 구현하였다.

1. 페이지 할당 알고리즘의 동작을 확인하기 위한 테스트 방법
   1. Next fit

테스트 코드

#include <stdio.h>

#include <string.h>

#include "threads/thread.h"

|  |
| --- |
| **메모리 할당**  **메모리/페이지 할당 알고리즘 구현 리포트** |

#include "threads/malloc.h"

#include "threads/palloc.h"

#include "projects/memalloc/memalloctest.h"

void run\_memalloc\_test(char \*\*argv UNUSED)

{

size\_t i;

char\* dynamicmem[12];

dynamicmem[0] = (char \*)malloc(262144);

memset(dynamicmem[0], 0x00, 262144);

dynamicmem[1] = (char \*)malloc(262144);

memset(dynamicmem[1], 0x00,262144);

dynamicmem[2] = (char \*)malloc(131072);

memset(dynamicmem[2], 0x00, 131072);

free(dynamicmem[1]);

dynamicmem[3] = (char \*)malloc(131072);

memset(dynamicmem[3], 0x00, 131072);

free(dynamicmem[2]);

dynamicmem[4] = (char \*)malloc(12000);

memset(dynamicmem[4], 0x00, 12000);

dynamicmem[5] = (char \*)malloc(32768);

memset(dynamicmem[5], 0x00, 32768);

dynamicmem[6] = (char \*)malloc(500000);

memset(dynamicmem[6], 0x00, 500000);

dynamicmem[7] = (char \*)malloc(16384);

memset(dynamicmem[7], 0x00, 16384);

dynamicmem[8] = (char \*)malloc(65536);

memset(dynamicmem[8], 0x00, 65536);

free(dynamicmem[6]);

free(dynamicmem[8]);

dynamicmem[9] = (char \*)malloc(26214);

memset(dynamicmem[9], 0x00, 26214);

dynamicmem[10] = (char \*) malloc (131072);

memset (dynamicmem[10], 0x00, 131072);

dynamicmem[11] = (char \*)malloc(26200);

memset(dynamicmem[11], 0x00, 26200);

printf ("dump page status : \n");

palloc\_get\_status (0);

thread\_sleep (100);

for (i=0; i<12; i++) {

if(i!=1 && i != 2 && i != 6 && i != 8) {

free(dynamicmem[i]);

printf ("dump page status : \n");

palloc\_get\_status (0);

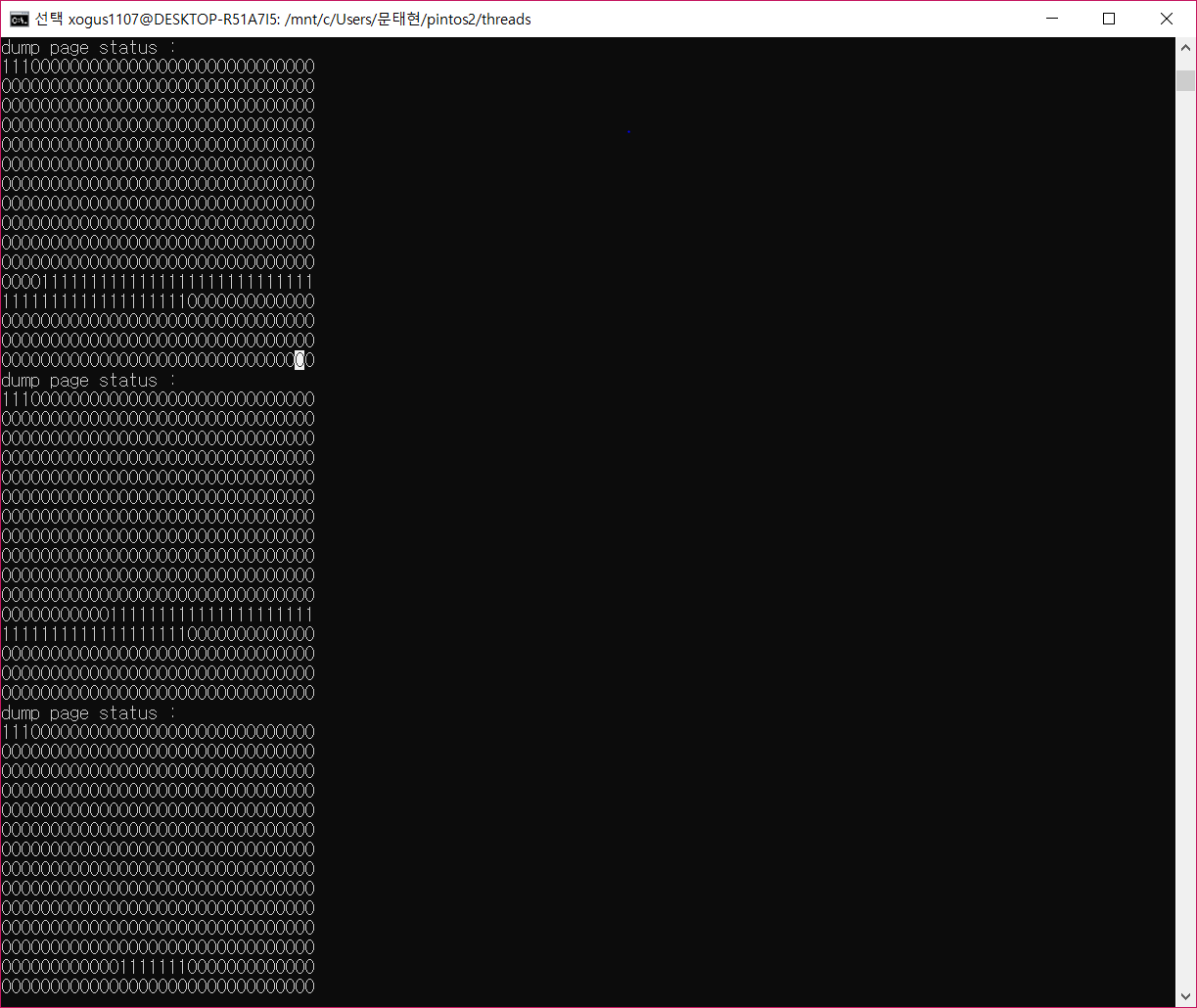
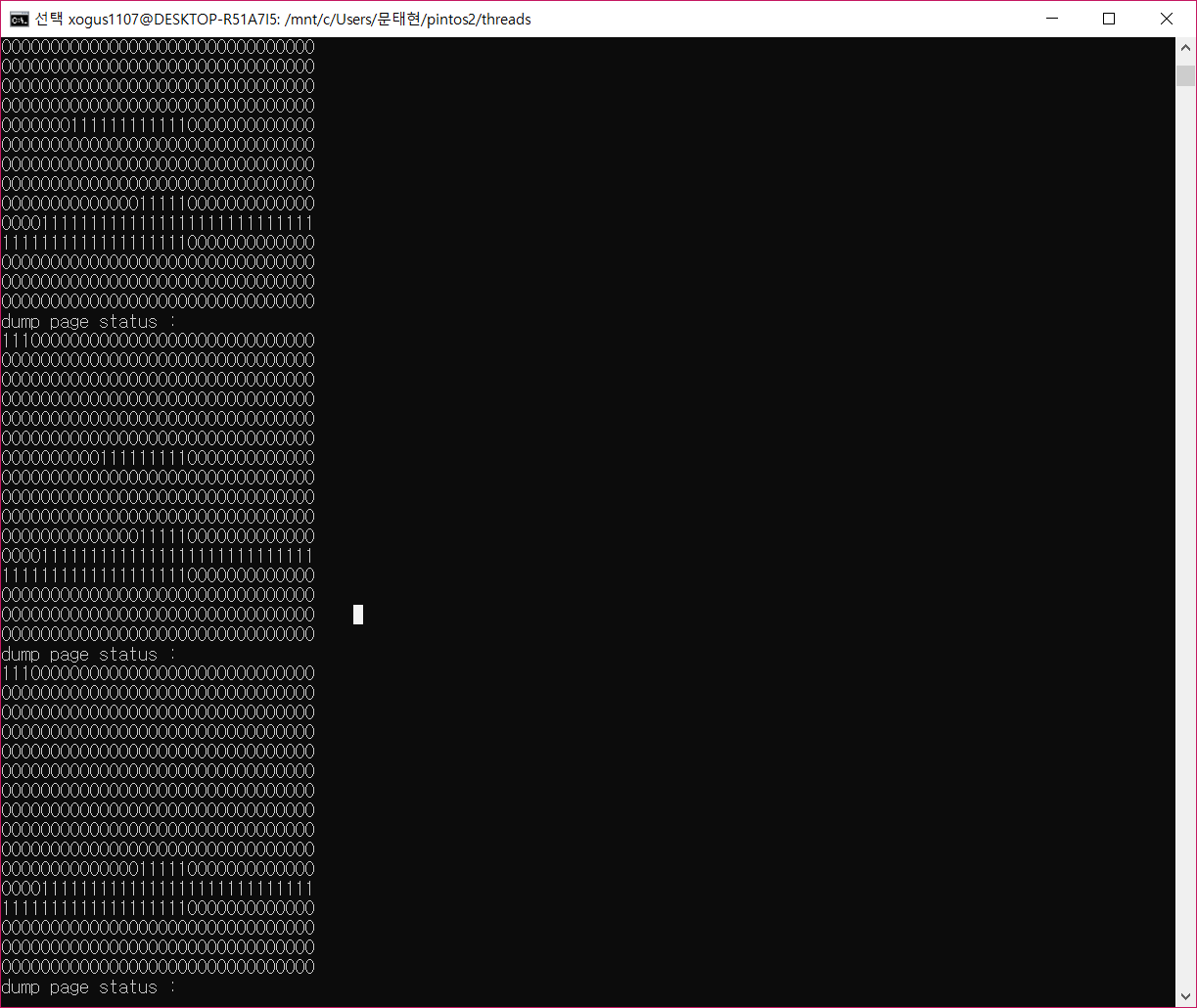
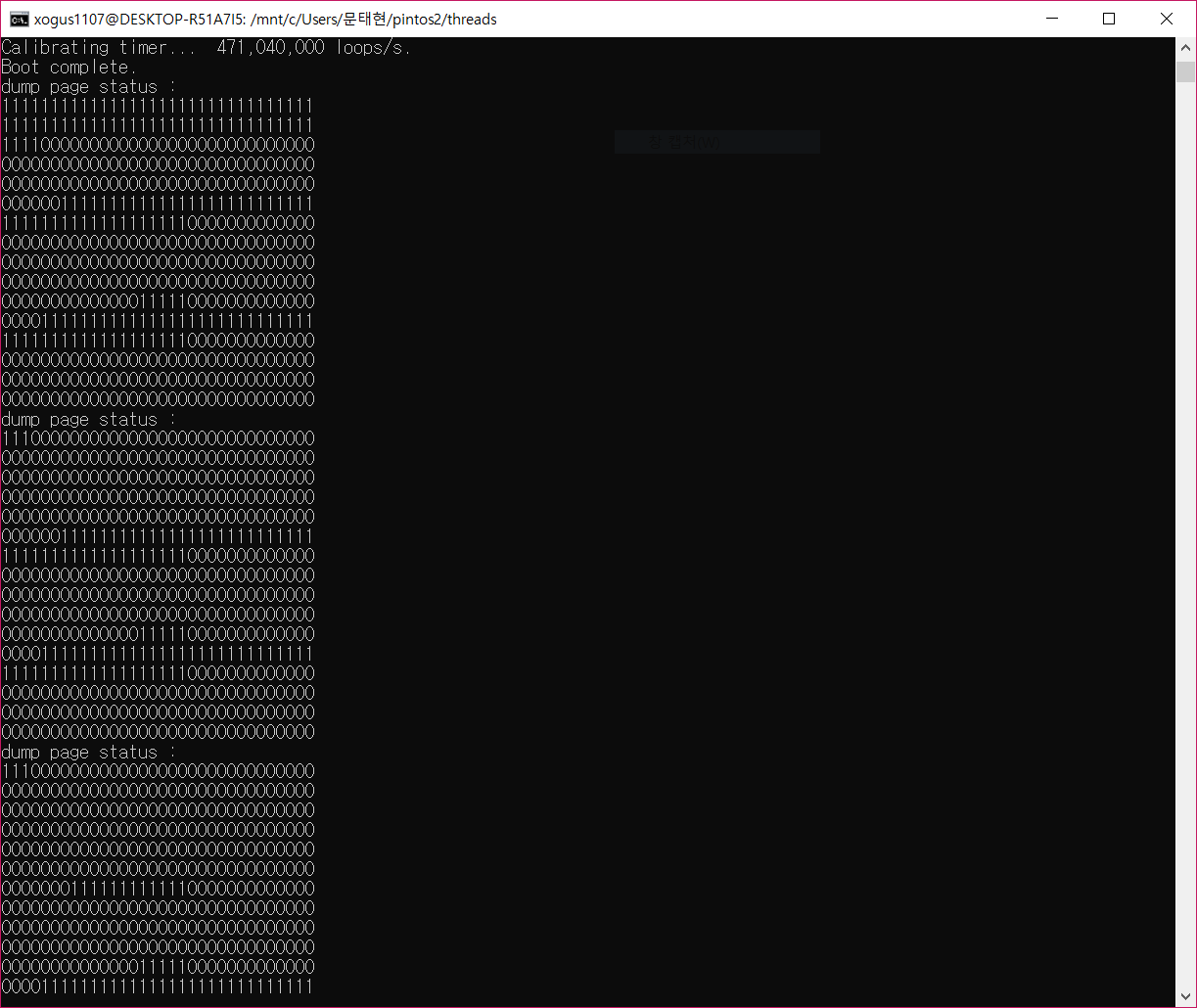
}

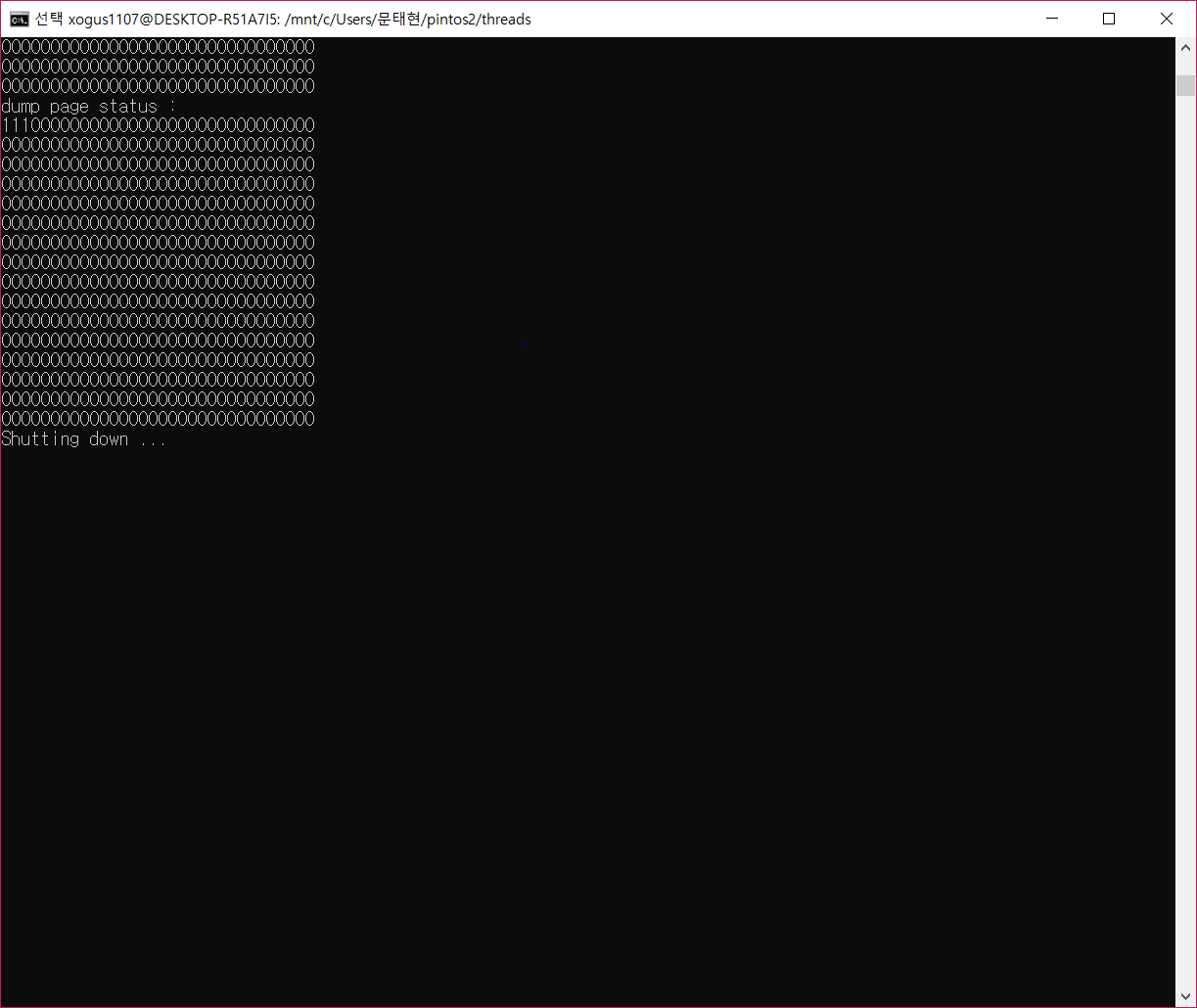
}

}

|  |
| --- |
| **메모리 할당**  **메모리/페이지 할당 알고리즘 구현 리포트** |

테스트 결과



* 1. Best fit

테스트 코드

#include <stdio.h>

#include <string.h>

#include "threads/thread.h"

#include "threads/malloc.h"

#include "threads/palloc.h"

#include "projects/memalloc/memalloctest.h"

void run\_memalloc\_test(char \*\*argv UNUSED)

{

size\_t i;

char\* dynamicmem[12];

|  |
| --- |
| **메모리 할당**  **메모리/페이지 할당 알고리즘 구현 리포트** |

dynamicmem[0] = (char \*)malloc(26214);

memset(dynamicmem[0], 0x00, 26214);

dynamicmem[1] = (char \*)malloc(50000);

memset(dynamicmem[1], 0x00,50000);

dynamicmem[2] = (char \*)malloc(13100);

memset(dynamicmem[2], 0x00, 13100);

free(dynamicmem[1]);

dynamicmem[3] = (char \*)malloc(48000);

memset(dynamicmem[3], 0x00, 48000);

//free(dynamicmem[2]);

dynamicmem[4] = (char \*)malloc(12000);

memset(dynamicmem[4], 0x00, 12000);

free(dynamicmem[0]);

dynamicmem[5] = (char \*)malloc(20000);

memset(dynamicmem[5], 0x00, 20000);

dynamicmem[6] = (char \*)malloc(50000);

memset(dynamicmem[6], 0x00, 50000);

dynamicmem[7] = (char \*)malloc(16384);

memset(dynamicmem[7], 0x00, 16384);

dynamicmem[8] = (char \*)malloc(65536);

memset(dynamicmem[8], 0x00, 65536);

free(dynamicmem[6]);

free(dynamicmem[8]);

dynamicmem[9] = (char \*)malloc(40000);

memset(dynamicmem[9], 0x00, 40000);

dynamicmem[10] = (char \*) malloc (131072);

memset (dynamicmem[10], 0x00, 131072);

dynamicmem[11] = (char \*)malloc(26200);

memset(dynamicmem[11], 0x00, 26200);

printf ("dump page status : \n");

palloc\_get\_status (0);

thread\_sleep (100);

for (i=0; i<12; i++) {

if(i!=0 && i != 1 && i != 6 && i != 8) {

free(dynamicmem[i]);

printf ("dump page status : \n");

palloc\_get\_status (0);

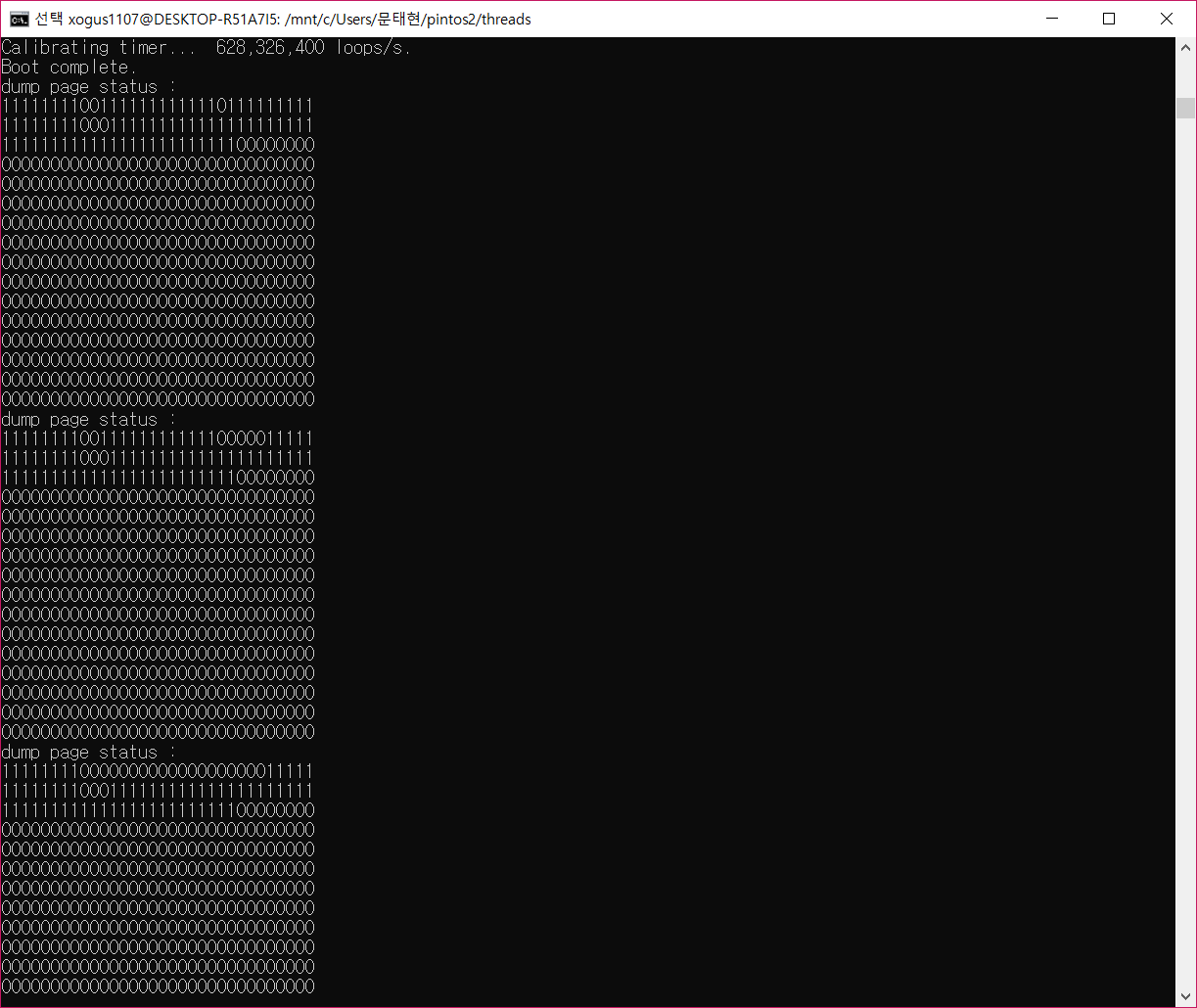
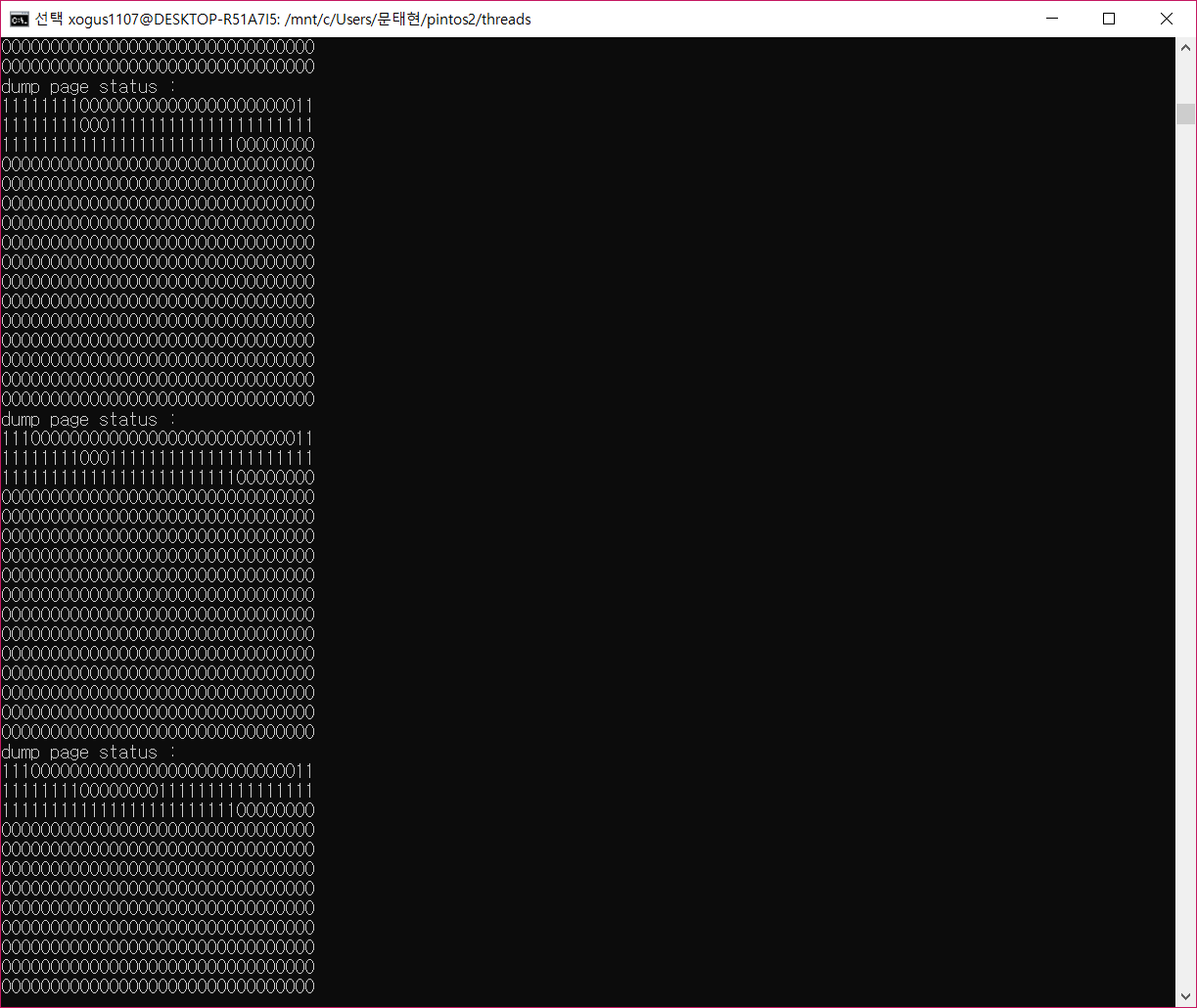
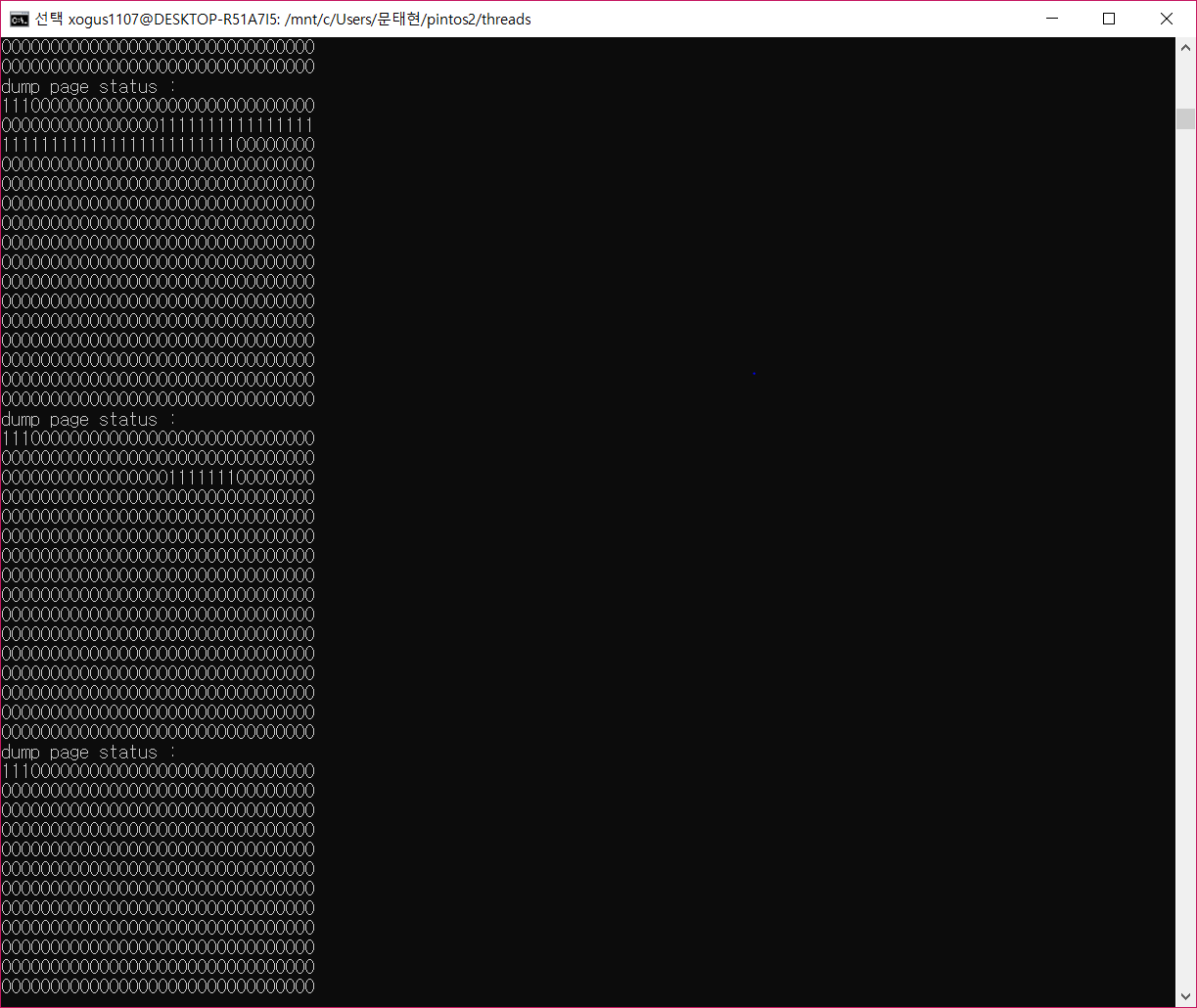
}

}

}

|  |
| --- |
| **메모리 할당**  **메모리/페이지 할당 알고리즘 구현 리포트** |

테스트 결과

Dynamicmem[1] = 50000을 free하고나서 Dynamicmem[3] = 48000을 할당하면 그 위치에 Dynamicmem[3]이 할당된다. 그리고 Dynamicmem[0] = 26214를 free해주고나서 Dynamicmem[5] = 20000을 할당하면 그 위치에 Dynamicmem[5]가 할당된다. Dynamicmem[6] = 50000, Dynamicmem[8] = 65536을 free해주고 나서 Dynamicmem[9] = 40000을 할당하면 Dynamicmem[8]이 아니라 Dynamicmem[6]이 있던 위치에 할당된다. 이로써 best fit이 구현 되었음을 알 수 있었다.

|  |
| --- |
| **메모리 할당**  **메모리/페이지 할당 알고리즘 구현 리포트** |

* 1. Buddy System

테스트 코드

#include <stdio.h>

#include <string.h>

#include "threads/thread.h"

#include "threads/malloc.h"

#include "threads/palloc.h"

#include "projects/memalloc/memalloctest.h"

void run\_memalloc\_test(char \*\*argv UNUSED)

{

size\_t i;

char\* dynamicmem[12];

dynamicmem[0] = (char \*)malloc(4096);

memset(dynamicmem[0], 0x00, 4096);

dynamicmem[1] = (char \*)malloc(65536);

memset(dynamicmem[1], 0x00, 65536);

dynamicmem[2] = (char \*)malloc(131072);

memset(dynamicmem[2], 0x00, 131072);

dynamicmem[3] = (char \*)malloc(8192);

memset(dynamicmem[3], 0x00, 8192);

dynamicmem[4] = (char \*)malloc(131072);

memset(dynamicmem[4], 0x00, 131072);

dynamicmem[5] = (char \*)malloc(32768);

memset(dynamicmem[5], 0x00, 32768);

dynamicmem[6] = (char \*)malloc(65536);

memset(dynamicmem[6], 0x00, 65536);

dynamicmem[7] = (char \*)malloc(16384);

memset(dynamicmem[7], 0x00, 16384);

dynamicmem[8] = (char \*)malloc(65536);

memset(dynamicmem[8], 0x00, 65536);

dynamicmem[9] = (char \*)malloc(262144);

memset(dynamicmem[9], 0x00, 262144);

dynamicmem[10] = (char \*) malloc (4096);

memset (dynamicmem[10], 0x00, 4096);

dynamicmem[11] = (char \*)malloc(32768);

memset(dynamicmem[11], 0x00, 32768);

printf ("dump page status : \n");

palloc\_get\_status (0);

thread\_sleep (100);

for (i=0; i<12; i++) {

|  |
| --- |
| **메모리 할당**  **메모리/페이지 할당 알고리즘 구현 리포트** |

free(dynamicmem[i]);

printf ("dump page status : \n");

palloc\_get\_status (0);

}

}

|  |
| --- |
| **메모리 할당**  **메모리/페이지 할당 알고리즘 구현 리포트** |

테스트 결과

