컴퓨터학부 20160548 이승현 나반

1. 과제 개요

malloc(3) 및 free(3)와 같은 함수를 대체하여 메모리를 동적으로 할당하는 메모리 관리자를 구현한다. 1번 문제에서는 페이지 1개(4KB)의 메모리를 관리하는 간단한 메모리 관리자를 구현한다. 2번 문제에서는 4개의 페이지(16KB)를 프로세스가 요청 시에만 할당하는 확장 가능한 heap을 구현한다.

- **구현 방법**

1. 간단한 메모리 관리자(동적으로 메모리 할당 및 해제) 구현

할당된 메모리 주소 영역과 할당되지 않은 메모리 주소 영역을 링크드 리스트를 이용하여 관리하는 메모리 관리자를 구현하였다. 이 때 링크드 리스트를 위해 한 페이지를 추가로 mmap(2)을 이용해 할당하여 사용하였다. 메모리 할당은 최초 적합 방식으로 할당하도록 했다.

링크드 리스트의 노드는 다음과 같이 구성된다.

```
typedef struct node {
        int is_valid;
        int start_addr;
        int size;
        struct node *next_node;
        struct node *prev_node;
} Node;
```

여기서 is_valid는 해당 노드가 유효한지 나타내는 멤버 변수이다. 유효하지 않다면 해당 노드에 덮어쓰기 해 재사용하게 된다. start_addr은 할당된 메모리 영역의 시작 주소를 나타낸다. 페이지 첫 부분이 주소 0이고, 바이트 단위로 1씩 증가한다. size는 할당된 메모리의 크기를 나타낸다. next_node는 링크드 리스트 내에서 다음 노드를 가리키고, prev_node는 이전 노드를 가리킨다.

이러한 노드로 이루어진 링크드 리스트를 할당된 메모리 영역, 할당되지 않은 메모리 영역 각각 하나씩, 총 두개를 만들어 메모리 관리에 사용했다.

2. 확장 가능한 Heap 구현

확장 가능한 heap은 1번 문제에서 구현했던 간단한 메모리 관리자를 이용해서 구현하였다. 4개의 페이지 각각은 1번 문제에서 구현한 함수들을 이용하여 관리하도록 했다. 이를 위해 1번문제에서 사용했던 함수들의 이름 뒤에 "_one_page"를 덧붙여 사용했고, 각 노드에 몇 번째 페이지의 메모리 영역을 가리키는 것인지 확인할 수 있도록 page_index 멤버 변수를 추가하였다. 링크드 리스트 역시 페이지별로 두개씩(할당되지 않은 메모리, 할당된 메모리

링크드 리스트) 두고 페이지 내의 메모리만 관리하도록 했다. 대신 노드들을 할당할 메모리 영역은 1번문제에서와 같이 한페이지(4KB)만 두고 모든 페이지 관리 리스트들이 함께 사용하도록 하였다. 메모리 할당 방식은 최초 적합 방식을 사용하였다. 첫번째 페이지에서 할당 가능한 메모리 영역을 찾고, 찾지 못하면 그 다음 페이지에서 찾는 방식으로 하였다. 이 때 페이지가 아직 할당 되어있지 않으면, 새로운 페이지를 할당하는 방식으로 확장 가능한 heap이 되도록 하였다.

2. 결과

- 1. 간단한 메모리 관리자(동적으로 메모리 할당 및 해제) test

```
shlee@shlee-virtual-machine:~/workspace/ssuos/project5$ gcc test_alloc.c alloc.c
shlee@shlee-virtual-machine:~/workspace/ssuos/project5$ ./a.out
Hello, world! test passed
Elementary tests passed
Starting comprehensive tests (see details in code)
Test 1 passed: allocated 4 chunks of 1KB each
Test 2 passed: dealloc and realloc worked
Test 3 passed: dealloc and smaller realloc worked
Test 4 passed: merge worked
Test 5 passed: merge alloc 2048 worked
shlee@shlee-virtual-machine:~/workspace/ssuos/project5$
```

모든 테스트 통과하였음

```
shlee@shlee-virtual-machine:~/workspace/ssuos/project5$ gcc test_alloc.c alloc.c
shlee@shlee-virtual-machine:~/workspace/ssuos/project5$ ./a.out
start:VSZ:4489216
Hello, world! test passed
should increase:VSZ:4624384
Elementary tests passed
should not change:VSZ:4624384

Starting comprehensive tests (see details in code)
Test 1 passed: allocated 4 chunks of 1KB each
should not change:VSZ:4624384

Test 2 passed: dealloc and realloc worked
should not change:VSZ:4624384

Test 3 passed: dealloc and smaller realloc worked
should not change:VSZ:4624384

Test 4 passed: merge worked
should not change:VSZ:4624384

Test 5 passed: merge alloc 2048 worked
should not change:VSZ:4624384

shlee@shlee-virtual-machine:~/workspace/ssuos/project5$
```

테스트 수행 도중 vsz 확인.

- 2. 확장 가능한 Heap test

```
shlee@shlee-virtual-machine:~/workspace/ssuos/project5$ gcc test_ealloc.c ealloc.c
shlee@shlee-virtual-machine:~/workspace/ssuos/project5$ ./a.out

Initializing memory manager

Test1: checking heap expansion; allocate 4 X 4KB chunks
start test 1:VSZ:4620288
should increase by 4KB:VSZ:4624384
should increase by 4KB:VSZ:4628480
should increase by 4KB:VSZ:4632576
should increase by 4KB:VSZ:4636672
should not change:VSZ:4636672
Test1: complete

Test2: Check splitting of existing free chunks: allocate 64 X 256B chunks
start test 2:VSZ:4636672
should not change:VSZ:4636672
should not change:VSZ:4636672
Test2: complete

Test3: checking merging of existing free chunks; allocate 4 X 4KB chunks
start test 3:VSZ:4636672
should not change:VSZ:4636672
should not change:VSZ:4636672
should not change:VSZ:4636672
should not change:VSZ:4636672
should not change:VSZ:4636672
Test3: complete

All tests complete
shlee@shlee-virtual-machine:~/workspace/ssuos/project5$
```

테스트 결과 정상적으로 작동됨을 확인하였음.

3. 소스코드

- **1. 간단한 메모리 관리자(동적으로 메모리 할당 및 해제)**

- **alloc.c**

```
#include "alloc.h"


typedef struct node { // 메모리 관리에 사용할 링크드 리스트의 노드 구조체
        int is_valid; // 해당 노드가 유효하면(해당 노드 사용중) 1, 아니면 0. 이게 1이면 이 노드가 저장된 메모리를 다른 노드가 사용하면 안됨.
        int start_addr; // 할당된 메모리 영역의 시작 주소(0 부터 시작, 1바이트 단위)
        int size; // 할당된 메모리 영역의 크기
        struct node *next_node; // 다음 메모리 관리 노드의 주소
        struct node *prev_node; // 이전 메모리 관리 노드의 주소
} Node;


typedef struct mem_list { // 메모리 관리 링크드 리스트들 저장하는 구조체
        Node *mem_in_use_head; // 할당되어 사용중인 메모리 영역 관리 링크드 리스트
        Node *mem_not_in_use_head; // 사용중이지 않은 메모리 영역 관리 링크드 리스트
} MemLinkedList;


char *mem; // heap 메모리 영역
Node *management_mem; // 메모리 관리 링크드 리스트의 노드들을 할당할 메모리 영역
MemLinkedList mem_linked_list; // 메모리 관리 링크드 리스트들 구조체


Node *getNewNode(); // 새로운 노드를 메모리에 할당하여 리턴하는 함수
void removeNode(Node *node); // 더이상 사용하지 않는 노드를 메모리에서 해제하는 함수


int init_alloc() {
```

```
	Node *new_node;

	mem = mmap(NULL, PAGESIZE, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0); // heap으로 사용할 메모리 영역을 mmap으로 할당함
	if (mem == MAP_FAILED) return -1; // mmap 실패시 -1 리턴

	management_mem = mmap(NULL, PAGESIZE, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0); // 메모리 관리 링크드 리스트에 사용할 메모리 영역을 mmap으로 할당
	if (management_mem == MAP_FAILED) return -1; // mmap 실패시 -1 리턴

	// 초기에는 메모리의 모든 공간이 할당 가능하므로 사용하지 않는 메모리 영역의 링크드 리스트에 메모리 전체 크기를 나타내는 노드를 넣는다
	new_node = getNewNode(); // 새로운 노드 생성
	new_node->start_addr = 0; // 시작 주소 0으로 설정
	new_node->size = PAGESIZE; // 크기는 PAGESIZE로 설정
	new_node->next_node = NULL; // 다음 노드 없음
	new_node->prev_node = NULL; // 이전 노드 없음

	mem_linked_list.mem_in_use_head = NULL; // 사용중인 메모리 없음
	mem_linked_list.mem_not_in_use_head = new_node; // 사용중이지 않은 메모리 관리 리스트에 위에서 생성한 노드를 넣는다.

	return 0;
}

int cleanup() {
	Node *node;
```

```
	// mem in use list 정리
	node = mem_linked_list.mem_in_use_head;
	while (node) {
		removeNode(node);
		node = node->next_node;
	}

	// mem not in use list 정리
	node = mem_linked_list.mem_not_in_use_head;
	while (node) {
		removeNode(node);
		node = node->next_node;
	}

	if (!munmap(management_mem, PAGESIZE)) return -1;
	if (!munmap(mem, PAGESIZE)) return -1;

	return 0;
}

char *alloc(int size) {
	Node *new_node;
	Node *new_alloced_mem;
	Node* mem_not_in_use;

	if (size % MINALLOC) { // 요청된 크기가 8의 배수가 아니면 NULL을 리턴한다
		return NULL;
```

```
	}

	mem_not_in_use = mem_linked_list.mem_not_in_use_head;
	while (1) {
		if (!mem_not_in_use) { // 여유 공간이 부족한 경우 NULL을 리턴한다
			return NULL;
		}

		if (mem_not_in_use->size > size) {
			// 사용할 만큼 할당
			// not_in_use Node의 size 변경
			mem_not_in_use->size -= size;

			// in_use list에 넣을 새로운 노드 생성
			new_node = getNewNode();
			new_node->size = size;
			new_node->start_addr = mem_not_in_use->start_addr + mem_not_in_use->size; // 앞에서 mem_not_in_use의 size 줄였기 때문에 그냥 size만 더해주면 됨
			new_node->prev_node = NULL;
			new_node->next_node = mem_linked_list.mem_in_use_head;

			// 새로운 노드를 in_use_list의 앞부분에 넣음
			if (mem_linked_list.mem_in_use_head) {
				mem_linked_list.mem_in_use_head->prev_node = new_node;
			}
			mem_linked_list.mem_in_use_head = new_node;
```

```
            new_alloced_mem = new_node;


            break;
    } else if (mem_not_in_use->size == size) {
            // not_in_use 리스트에서 해당 노드 제거
            if (mem_not_in_use->prev_node) {
                    mem_not_in_use->prev_node->next_node = mem_not_in_use->next_node;
            } else {
                    mem_linked_list.mem_not_in_use_head = mem_not_in_use->next_node;
            }


            if (mem_not_in_use->next_node) {
                    mem_not_in_use->next_node->prev_node = mem_not_in_use->prev_node;
            }



            // in_use로 이동할 노드 setting
            mem_not_in_use->prev_node = NULL;
            mem_not_in_use->next_node = mem_linked_list.mem_in_use_head;



            // in_use_list의 앞부분에 집어넣음
            if (mem_linked_list.mem_in_use_head) {
                    mem_linked_list.mem_in_use_head->prev_node = mem_not_in_use;
            }
            mem_linked_list.mem_in_use_head = mem_not_in_use;
```

```
                new_alloced_mem = mem_not_in_use;



                break;
            }

            mem_not_in_use = mem_not_in_use->next_node; // 다음 노드에 할당 가능한지 확인
        }


        return mem + new_alloced_mem->start_addr; // 할당된 메모리 주소를 리턴한다
}

void dealloc(char *dealloc_ptr) {
        int mem_index = dealloc_ptr - mem; // dealloc할 메모리의 주소
        Node *node;
        Node *mem_not_in_use;
        Node *tmp_node;
        Node *prev_node = NULL;

        if (mem_index < 0 || 4096 <= mem_index) { // 범위를 벗어난 주소가 전달됐을 때
            return;
        }


        node = mem_linked_list.mem_in_use_head;
        while (node) {
```

```
        if (node->start_addr == mem_index) {
                break;
        }

        node = node->next_node;
}


if (!node) return; // 인자로 전달된 주소에 해당되는 노드를 찾지 못했으면 종료

// mem_in_use 리스트를 정리한다
if (node->prev_node) { // 헤드노드가 아닐 때
        node->prev_node->next_node = node->next_node;
} else { // 헤드노드일 때
        mem_linked_list.mem_in_use_head = node->next_node;
}

if (node->next_node) {
        node->next_node->prev_node = node->prev_node;
}


// mem_not_in_use 리스트에 넣는다
mem_not_in_use = mem_linked_list.mem_not_in_use_head;
if (!mem_not_in_use) {
        node->prev_node = NULL;
        node->next_node = NULL;
```

```
        mem_linked_list.mem_not_in_use_head = node;
} else {
        while (mem_not_in_use) {
                if (mem_not_in_use->start_addr > node->start_addr) {
                        // mem_not_in_list에 node 삽입
                        node->prev_node = mem_not_in_use->prev_node;
                        node->next_node = mem_not_in_use;
                        if (node->prev_node) {
                                node->prev_node->next_node = node;
                        } else {
                                mem_linked_list.mem_not_in_use_head = node;
                        }
                        mem_not_in_use->prev_node = node;


                        // 뒤쪽 노드와 합칠 수 있는지 확인
                        if (node->next_node) {
                                if (node->start_addr + node->size == node->next_node->start_addr) {
                                        tmp_node = node->next_node;
                                        node->size += node->next_node->size;
                                        node->next_node = node->next_node->next_node;
                                        if (node->next_node) {
                                                node->next_node->prev_node = node;
                                        }

                                        // 필요 없어진 노드 제거
                                        removeNode(tmp_node);
```

```
                }
            }


            // 앞쪽 노드와 합칠 수 있는지 확인
            if (node->prev_node) {
                if (node->prev_node->start_addr + node->prev_node->size == node->start_addr) {
                    node->prev_node->next_node = node->next_node;
                    if (node->next_node) {
                        node->next_node->prev_node = node->prev_node;
                    }
                    node->prev_node->size += node->size;

                    // 필요 없어진 노드 제거
                    removeNode(node);
                }

            }

            break;
        }

        prev_node = mem_not_in_use;
        mem_not_in_use = mem_not_in_use->next_node;
    }
```

```
            if (!mem_not_in_use) {
                // mem_not_in_use list의 맨 끝에 삽입해야 하는 경우
                prev_node->next_node = node;
                node->prev_node = prev_node;
                node->next_node = NULL;

                // 앞쪽 노드와 합칠 수 있는지 확인
                if (node->prev_node) {
                    if (node->prev_node->start_addr + node->prev_node->size == node->start_addr) {
                        node->prev_node->next_node = node->next_node;
                        node->prev_node->size += node->size;

                        // 필요 없어진 노드 제거
                        removeNode(node);
                    }

                }
            }
        }


        return;
}

Node *getNewNode() { // 새로운 노드 할당하고 주소 리턴하는 함수
        while (management_mem->is_valid) { // 해당 메모리가 사용중이라면
            management_mem += 1; // 다음 위치 확인
```

```
        }
        management_mem->is_valid = 1; // 새로 할당할 위치 사용중이라고 표시

        return management_mem; // 주소 리턴
}

void removeNode(Node *node) { // 사용 끝난 노드 메모리 해제하는 함수
        node->is_valid = 0; // 유효하지 않다고 표시하면 끝
}

void printAllNode() { // 링크드 리스트의 모든 노드 내용 출력하는 함수, 디버깅용으로 사용
        Node *node;
        printf("₩n******************print all nodes******************₩n");
        printf("mem in use:₩n");
        node = mem_linked_list.mem_in_use_head;
        while(node) {
                printf("index: %d, size: %d₩n", node->start_addr, node->size);

                node = node->next_node;
        }

        printf("mem not in use:₩n");
        node = mem_linked_list.mem_not_in_use_head;
        while(node) {
                printf("index: %d, size: %d₩n", node->start_addr, node->size);

                node = node->next_node;
```

```
        }

        printf("*****************print all nodes end***************₩n₩n");


        return;

}
```

- **2. 확장 가능한 Heap**

- **ealloc.c**

```
#include "ealloc.h"


#define MAX_PAGE_COUNT 4


typedef struct node { // 메모리 관리에 사용할 링크드 리스트의 노드 구조체
        int page_index; // 해당 노드가 가리키는 메모리 영역이 속해있는 페이지의 번호
        int is_valid; // 해당 노드가 유효하면(해당 노드 사용중) 1, 아니면 0. 이게 1이면 이 노드가 저장된 메모리를 다른 노드가 사용하면 안됨.
        int start_addr; // 할당된 메모리 영역의 시작 주소(0 부터 시작, 1바이트 단위)
        int size; // 할당된 메모리 영역의 크기
        struct node *next_node; // 다음 메모리 관리 노드의 주소
        struct node *prev_node; // 이전 메모리 관리 노드의 주소
} Node;


typedef struct mem_list { // 메모리 관리 링크드 리스트들 저장하는 구조체
        Node *mem_in_use_head; // 할당되어 사용중인 메모리 영역 관리 링크드 리스트
        Node *mem_not_in_use_head; // 사용중이지 않은 메모리 영역 관리 링크드 리스트
} MemLinkedList;


char *mem[MAX_PAGE_COUNT]; // heap 메모리 영역 (페이지 4개)
MemLinkedList memLinkedLists[MAX_PAGE_COUNT]; // 메모리 관리 링크드 리스트들 구조체(페이지당 하나씩)
Node *management_mem; // 메모리 관리 링크드 리스트의 노드들을 할당할 메모리 영역


int init_alloc_one_page(int i);
int cleanup_one_page(int i);
```

```
char *alloc_one_page(int i, int size);

void dealloc_one_page(int i, char *dealloc_ptr);

Node *getNewNode();

void removeNode(Node *node);

int checkallocedatpage(int page_num, char *addr);

void printallnode(int i);


void init_alloc() {

	management_mem = mmap(NULL, PAGESIZE, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0); // 메모리 관리에 사용할 메모리 영역을 할당받는다

}




char *alloc(int size) {

	int i;

	char *new_alloced_mem = NULL;


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

		if (!mem[i]) { // 해당 heap 메모리 페이지가 아직 할당되지 않았다면

			init_alloc_one_page(i); // 한페이지 할당

		}


		new_alloced_mem = alloc_one_page(i, size); // i번 페이지에서 메모리 할당


		if (new_alloced_mem) { // 할당 되었다면

			break; // break

		}
```

```
                // 할당 안됐으면 다음 페이지에서 할당 시도
        }

        return new_alloced_mem;
}

void dealloc(char *dealloc_ptr) {
        int i;

        for (i = 0; i < MAX_PAGE_COUNT; ++i) {
                if (checkallocedatpage(i, dealloc_ptr)) { // 해제할 메모리가 해당 페이지에 있는지 확인
                        dealloc_one_page(i, dealloc_ptr); // 해당 페이지에서 메모리 해제
                        break;
                }
        }
}


void cleanup() {
        int i = 0;
        // 메모리 가리키는 포인터와 관리 링크드 리스트들 초기화
        for (i = 0; i < MAX_PAGE_COUNT; ++i) {
                mem[i] = NULL;
                memLinkedLists[i].mem_in_use_head = NULL;
                memLinkedLists[i].mem_not_in_use_head = NULL;
                management_mem = NULL;
        }
```

```
        return;
}

int checkallocedatpage(int page_num, char *addr) { // 전달된 주소가 해당 페이지에 할당되어 있는 메모리 영역의 주소인지 확인하는 함수
        int i;
        Node *node = memLinkedLists[page_num].mem_in_use_head;
        char *base = mem[page_num];

        if (!base) return 0;

        while (node) {
                if (base + node->start_addr == addr) { // 주소 일치하면
                        return 1; // 1리턴
                }

                node = node->next_node; // 다음 노드 확인
        }

        return 0; // 일치하는 주소 없으면 0 리턴
}

//////////////////////////////////////////////////////
// 아래는 간단한 메모리 관리자 코드를 재활용함

int init_alloc_one_page(int i) {
```

```
	Node *new_node;

	mem[i] = mmap(NULL, PAGESIZE, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
	if (mem[i] == MAP_FAILED) return -1;

	new_node = getNewNode();
	new_node->page_index = i;
	new_node->start_addr = 0;
	new_node->size = PAGESIZE;
	new_node->next_node = NULL;
	new_node->prev_node = NULL;

	// 해당 페이지에 대한 리스트를 따로 만들어 관리
	memLinkedLists[i].mem_in_use_head = NULL;
	memLinkedLists[i].mem_not_in_use_head = new_node;

	return 0;
}

int cleanup_one_page(int i) {
	Node *node;

	// mem in use list 정리
	// 해당 페이지에 대한 리스트를 정리하도록 수정
	node = memLinkedLists[i].mem_in_use_head;
	while (node) {
		removeNode(node);
```

```
                node = node->next_node;
        }

        // mem not in use list 정리
        node = memLinkedLists[i].mem_not_in_use_head;
        while (node) {
                removeNode(node);
                node = node->next_node;
        }

        return 0;
}

char *alloc_one_page(int i, int size) {
        Node *new_node;
        Node *new_alloced_mem;
        Node * mem_not_in_use;

        if (size % MINALLOC) {
                return NULL;
        }

        mem_not_in_use = memLinkedLists[i].mem_not_in_use_head;
        while (1) {
                if (!mem_not_in_use) {
                        return NULL;
                }
```

```
if (mem_not_in_use->size > size) {
        // 사용할 만큼 할당
        // not_in_use node의 size 변경
        mem_not_in_use->size -= size;

        // in_use list에 넣을 새로운 노드 생성
        new_node = getNewNode();
        new_node->size = size;
        new_node->start_addr = mem_not_in_use->start_addr + mem_not_in_use->size; // 앞에서 mem_not_in_use의 size 줄였기 때문에 그냥 size만 더해주면 됨
        new_node->prev_node = NULL;
        new_node->next_node = memLinkedLists[i].mem_in_use_head;

        // 새로운 노드를 in_use_list의 앞부분에 넣음
        if (memLinkedLists[i].mem_in_use_head) {
                memLinkedLists[i].mem_in_use_head->prev_node = new_node;
        }
        memLinkedLists[i].mem_in_use_head = new_node;

        new_alloced_mem = new_node;

        break;
} else if (mem_not_in_use->size == size) {
        // not_in_use 리스트에서 해당 노드 제거
        if (mem_not_in_use->prev_node) {
                mem_not_in_use->prev_node->next_node = mem_not_in_use->next_node;
```

```
                } else {
                        memLinkedLists[i].mem_not_in_use_head = mem_not_in_use->next_node;
                }

                if (mem_not_in_use->next_node) {
                        mem_not_in_use->next_node->prev_node = mem_not_in_use->prev_node;
                }



                // in_use로 이동할 노드 setting
                mem_not_in_use->prev_node = NULL;
                mem_not_in_use->next_node = memLinkedLists[i].mem_in_use_head;

                // in_use_list의 앞부분에 집어넣음
                if (memLinkedLists[i].mem_in_use_head) {
                        memLinkedLists[i].mem_in_use_head->prev_node = mem_not_in_use;
                }
                memLinkedLists[i].mem_in_use_head = mem_not_in_use;

                new_alloced_mem = mem_not_in_use;

                break;
        }

        mem_not_in_use = mem_not_in_use->next_node;
}
```

```
	return mem[i] + new_alloced_mem->start_addr;
}

void dealloc_one_page(int i, char *dealloc_ptr) {
	int mem_index = dealloc_ptr - mem[i];
	Node *node;
	Node *mem_not_in_use;
	Node *tmp_node;
	Node *prev_node = NULL;

	if (mem_index < 0 || 4096 <= mem_index) {
		return;
	}


	node = memLinkedLists[i].mem_in_use_head;
	while (node) {
		if (node->start_addr == mem_index) {
			break;
		}

		node = node->next_node;
	}


	if (!node) return; // 인자로 전달된 주소에 해당되는 노드를 찾지 못했으면 종료
```

```
// mem_in_use 리스트를 정리한다
if (node->prev_node) { // 헤드노드가 아닐 때
        node->prev_node->next_node = node->next_node;
} else { // 헤드노드일 때
        memLinkedLists[i].mem_in_use_head = node->next_node;
}

if (node->next_node) {
        node->next_node->prev_node = node->prev_node;
}

// mem_not_in_use 리스트에 넣는다
mem_not_in_use = memLinkedLists[i].mem_not_in_use_head;
if (!mem_not_in_use) {
        node->prev_node = NULL;
        node->next_node = NULL;
        memLinkedLists[i].mem_not_in_use_head = node;
} else {
        while (mem_not_in_use) {
                if (mem_not_in_use->start_addr > node->start_addr) {
                        // mem_not_in_list에 node 삽입
                        node->prev_node = mem_not_in_use->prev_node;
                        node->next_node = mem_not_in_use;
                        if (node->prev_node) {
                                node->prev_node->next_node = node;
                        } else {
```

```
            memLinkedLists[i].mem_not_in_use_head = node;
        }
        mem_not_in_use->prev_node = node;


        // 뒤쪽 노드와 합칠 수 있는지 확인
        if (node->next_node) {
            if (node->start_addr + node->size == node->next_node->start_addr) {
                tmp_node = node->next_node;
                node->size += node->next_node->size;
                node->next_node = node->next_node->next_node;
                if (node->next_node) {
                    node->next_node->prev_node = node;
                }

                // 필요 없어진 노드 제거
                removeNode(tmp_node);
            }
        }

        // 앞쪽 노드와 합칠 수 있는지 확인
        if (node->prev_node) {
            if (node->prev_node->start_addr + node->prev_node->size == node->start_addr) {
                node->prev_node->next_node = node->next_node;
                if (node->next_node) {
                    node->next_node->prev_node = node->prev_node;
```

```
                                        }

                                        node->prev_node->size += node->size;


                                        // 필요 없어진 노드 제거

                                        removeNode(node);

                                }


                        }


                        break;

                }


                prev_node = mem_not_in_use;

                mem_not_in_use = mem_not_in_use->next_node;

}


if (!mem_not_in_use) {

                // mem_not_in_use list의 맨 끝에 삽입해야 하는 경우

                prev_node->next_node = node;

                node->prev_node = prev_node;

                node->next_node = NULL;


                // 앞쪽 노드와 합칠 수 있는지 확인

                if (node->prev_node) {

                        if (node->prev_node->start_addr + node->prev_node->size == node->start_addr) {

                                node->prev_node->next_node = node->next_node;

                                node->prev_node->size += node->size;
```

```
                                        // 필요 없어진 노드 제거
                                        removeNode(node);
                                    }

                                }
                            }
                        }

        return;
}

Node *getNewNode() {
        while (management_mem->is_valid) {
                management_mem += 1;
        }
        management_mem->is_valid = 1;

        return management_mem;
}

void removeNode(Node *node) {
        node->is_valid = 0;
}

void printAllNode(int i) {
```

```
	Node *node;
	printf("₩n*****************print all nodes*****************₩n");
	printf("mem in use:₩n");
	node = memLinkedLists[i].mem_in_use_head;
	while(node) {
		printf("index: %d, size: %d₩n", node->start_addr, node->size);

		node = node->next_node;
	}

	printf("mem not in use:₩n");
	node = memLinkedLists[i].mem_not_in_use_head;
	while(node) {
		printf("index: %d, size: %d₩n", node->start_addr, node->size);

		node = node->next_node;
	}
	printf("*****************print all nodes end****************₩n₩n");

	return;
}
```