Lecture 11

동적 메모리 할당

리뷰

- 표준 라이브러리
  - **파일 조작**: `fopen()`, `freopen()`, `fflush()`, `remove()`, `rename()`, `tmpfile()`, `tmpnam()`, `fread()`, `fwrite()`, `fseek()`, `ftell()`, `rewind()`, `clearerr()`, `feof()`, `ferror()`
  - **문자 분류**: `isalpha()`, `isdigit()`, `isalnum()`, `iscntrl()`, `islower()`, `isupper()`, `isprint()`, `ispunct()`, `isspace()`
  - **메모리**: `memcpy()`, `memmove()`, `memcmp()`, `memset()`
  - **변환**: `atoi()`, `atol()`, `atof()`, `strtol()`, `strtoul()`, `strtod()`
  - **유틸리티**: `rand()`, `srand()`, `abort()`, `exit()`, `atexit()`, `system()`, `bsearch()`, `qsort()`
  - **진단**: `assert()`, `__FILE__`, `__LINE__` 매크로

리뷰

- 표준 라이브러리
  - **매개변수 리스트**
    - 함수 선언에서 "…"을 사용함

```
int printf(const char *fmt, …);
```

    - `va_list ap`로 접근, `va_start`로 초기화, `va_arg`로 다음의 매개변수 선정, `va_end`로 종료
  - **시간**: `clock()`, `time()`, `difftime()`, `mktime()`, `asctime()`, `localtime()`, `ctime()`, `strftime()`

프로그램의 메모리 구성

- C 프로그램 동작 시 차지하는 메모리 구성

높은 메모리 주소

낮은 메모리 주소

프로그램의 메모리 구성

- C 프로그램 동작 시 차지하는 메모리 구성
  - 메모리 구성 확인을 위해 `size` 명령을 사용할 수 있음

```c
int main() {
    return 0;
}
```

```
D:\>size test.exe
   text    data     bss     dec     hex filename
  11184    2232     384   13800    35e8 test.exe
```

```c
int gvar;
int main() {
    return 0;
}
```

```
D:\>size test.exe
   text    data     bss     dec     hex filename
  11184    2232     416   13832    3608 test.exe
```

```c
int gvar=10;
int main() {
    return 0;
}
```

```
D:\>size test.exe
   text    data     bss     dec     hex filename
  11184    2248     384   13816    35f8 test.exe
```

프로그램의 메모리 구성

- C 프로그램 동작 시 차지하는 메모리 구성
  - 메모리 구성 확인을 위해 `size` 명령을 사용할 수 있음

```c
int main() {
    return 0;
}
```

```
D:\>size test.exe
   text    data     bss     dec     hex filename
  11184    2232     384   13800    35e8 test.exe
```

```c
void func() {
    int lvar=1;
    printf("%d\n",lvar);
}

int main() {
    return 0;
}
```

```
D:\>size test.exe
   text    data     bss     dec     hex filename
  11424    2248     384   14056    36e8 test.exe
```

정적 메모리 할당

- 배열의 크기를 컴파일 때 고정해야 함

```
int main() {
    int n;
    scanf("%d", &n);
    int arr[n]; // 에러
    return 0;
}
```

C90 표준과의 컴파일

```
gcc -std=c90 -pedantic-errors
    -Wall main.c -o main
```

- C99 표준부터 VLA(Variable Length Array)은 지원됨
  - 위 코드는 C99 이상인 표준으로 컴파일될 수 있음
  - 배열 `arr`는 스택에서 할당되므로 할당해줄 수 있는 크기에 한계가 있음

1989	1990	1999	2011	2017
C89	C90	C99	C11	C17

동적 메모리 할당

- 프로그램 실행 중에 필요한 만큼 메모리를 확보하고 해제할 수 있음

`malloc(size)`	`size` 바이트만큼 메모리 할당, 초기화 안 됨
`calloc(n, size)`	`n*size` 바이트만큼 메모리 할당 후 0으로 초기화
`realloc(ptr, size)`	기존 `ptr`가 가리키는 주소의 메모리를 `size`만큼 재조정
`free(ptr)`	할당한 메모리를 해제

```
int *a;
a = (int*)malloc(10*sizeof(int));
```

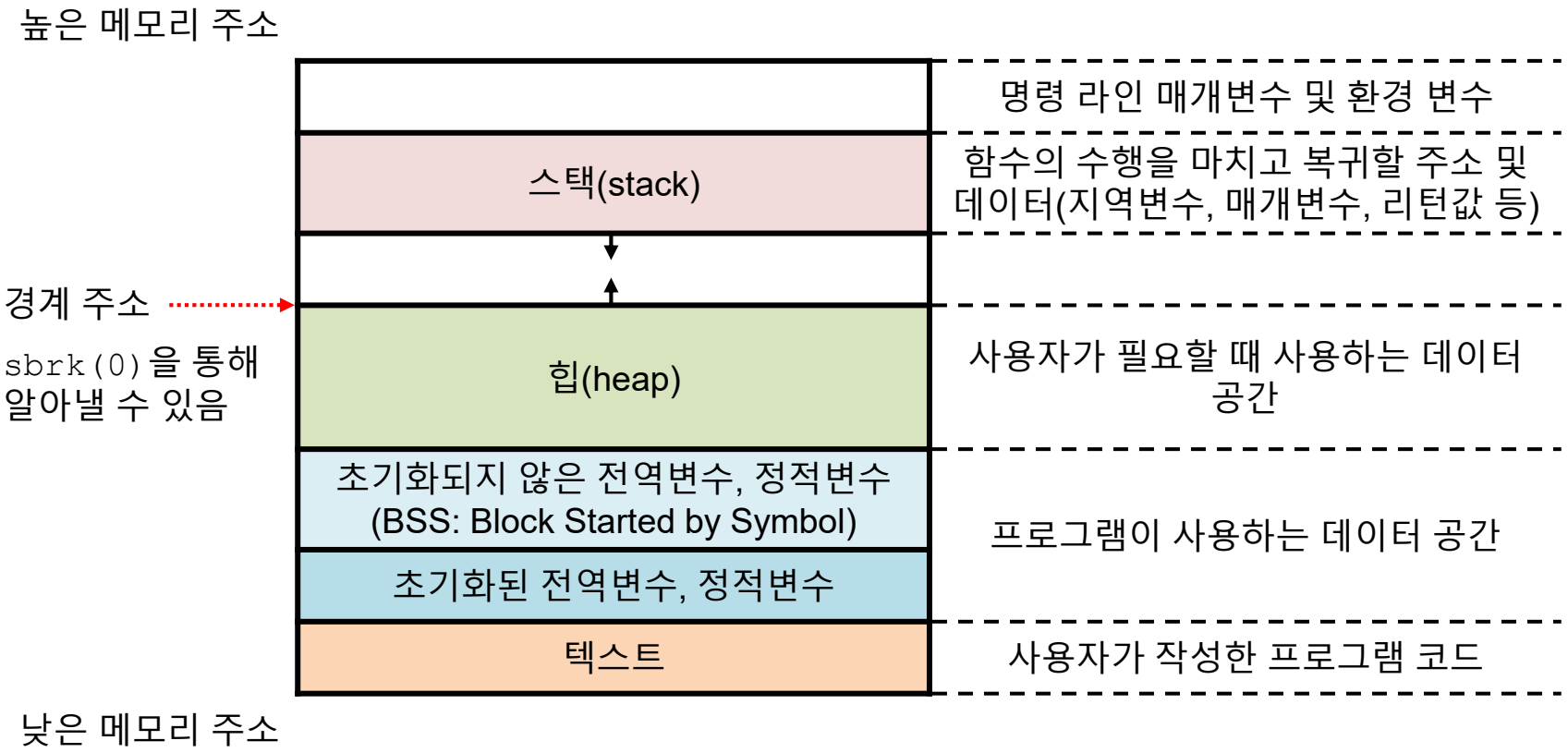
- `int` 형 포인터 변수 `a`는 스택 영역에 저장됨
- 포인터 `a`가 가리키는 `40` 바이트 메모리 블록은 힙 영역에 할당됨
- 할당된 메모리는 더 이상 필요하지 않으면 `free(a)`로 해제해줘야 함

메모리 할당 관련 시스템 콜

- `mmap()`
  - 파일 또는 기타 데이터를 메모리에 매핑하여 데이터를 효율적으로 읽고 쓸 수 있음
- `munmap()`
  - 매핑된 메모리를 해제함
- `brk()` 및 `sbrk()`
  - 할당된 메모리와 할당되지 않은 메모리 사이의 경계를 조정해줌
  - `brk(addr)`
    - 지정할 `addr`까지 현재의 경계 주소를 증가하거나 축소함
    - 성공 시 `0`, 에러 시 `-1`을 리턴함
  - `sbrk(incr)`
    - 지정할 `incr`만큼 현재의 경계 주소를 증가하거나 축소함
    - 성공 시 이전의 경계 주소, 에러 시 `(void*)-1`을 리턴함

메모리 할당 관련 시스템 콜

- 할당된 메모리와 할당되지 않은 메모리 사이의 경계



동적 메모리 할당

- C 표준 라이브러리 함수
  - `malloc, calloc, realloc, free`
  - 이식성(portability)이 있음
  - 실제 하드웨어 대신 운영체제에 메모리를 요청함
- 운영체제 시스템 콜
  - `mmap, munmap, brk, sbrk`
  - 프로세스의 가상 메모리 공간을 관리함
  - 위의 C 표준 라이브러리 함수들의 **빌딩 블록**이 됨
- 예, `malloc`의 동작
  - 작은 메모리 요청(몇 KB 정도): `brk`, `sbrk`를 호출하여 힙을 조정함
  - 큰 메모리 요청(보통 128KB 이상): `mmap`, `munmap`을 호출하고 OS에 메모리 블록을 요청하거나 반환함

메모리 단편화

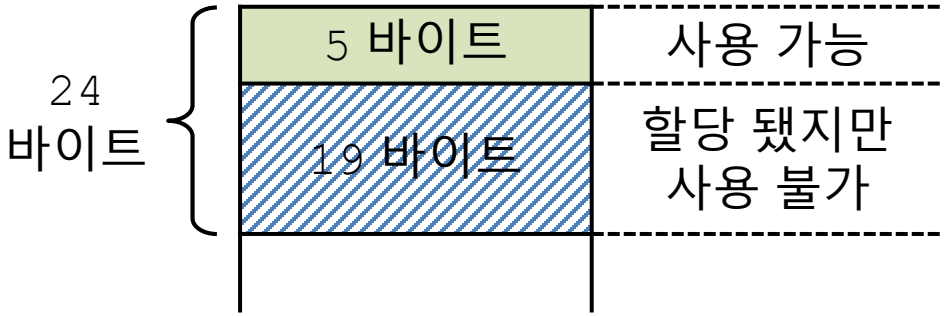
- 메모리 단편화(memory fragmentation)
  - 힙 메모리 공간이 파편화되는 현상이며, 힙 영역에서 메모리가 남아있지만 블록을 할당하기 어려운 상황
  - 2가지 종류
    - **내부 단편화**(internal fragmentation)
      - 요청한 메모리보다 더 큰 블록이 할당되어 생기는 낭비된 공간
      - 원인
        - CPU가 메모리를 8, 16, 32 바이트 단위로 맞춤
        - 블록 크기, 상태 등의 메타데이터(metadata) 저장
        - 메모리를 미리 정해진 크기 블록(bucket)으로 관리
    - **외부 단편화**(external fragmentation)
      - 메모리 전체 사용 가능 공간이 충분하나, 연속된 큰 블록이 없어 요청을 바로 할당할 수 없는 상황
      - 원인: 메모리 블록이 할당/해제 되면서 작은 빈 공간들이 분산됨

메모리 단편화

- 내부 단편화
  - 리눅스 환경에서 내부 단편화를 확인할 수 있은 예시 코드

```
char *p = (char*)malloc(5); // 5 바이트 요청
size_t actual = malloc_usable_size(p); // 실제 할당 바이트
                                       // malloc.h에 제공
```

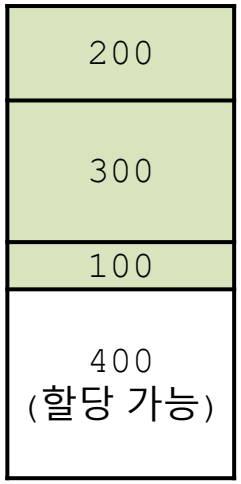
```
dat@dat-VirtualBox:~/Downloads/test_C_memory_allocation$ vim test.c
dat@dat-VirtualBox:~/Downloads/test_C_memory_allocation$ gcc -Wall test.c -o test
dat@dat-VirtualBox:~/Downloads/test_C_memory_allocation$ ./test
requested = 5
actual    = 24
```



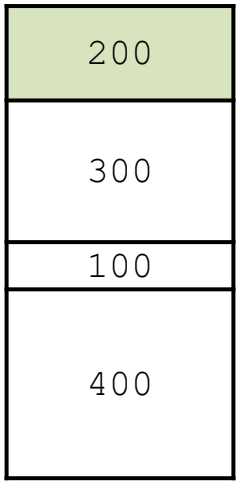
메모리 단편화

- 외부 단편화
  - 다음의 간소화된 예시를 참고함

① 초기 상태:
1000 바이트



② `malloc(200);`
`malloc(300);`
`malloc(100);`



할당이 가능한 공간은 800 바이트이지만 이전의 할당/해제로 인해 300, 100, 400 블록으로 나누게 됨

③ `free(300);`
`free(100);`
총 800 바이트 할당 가능

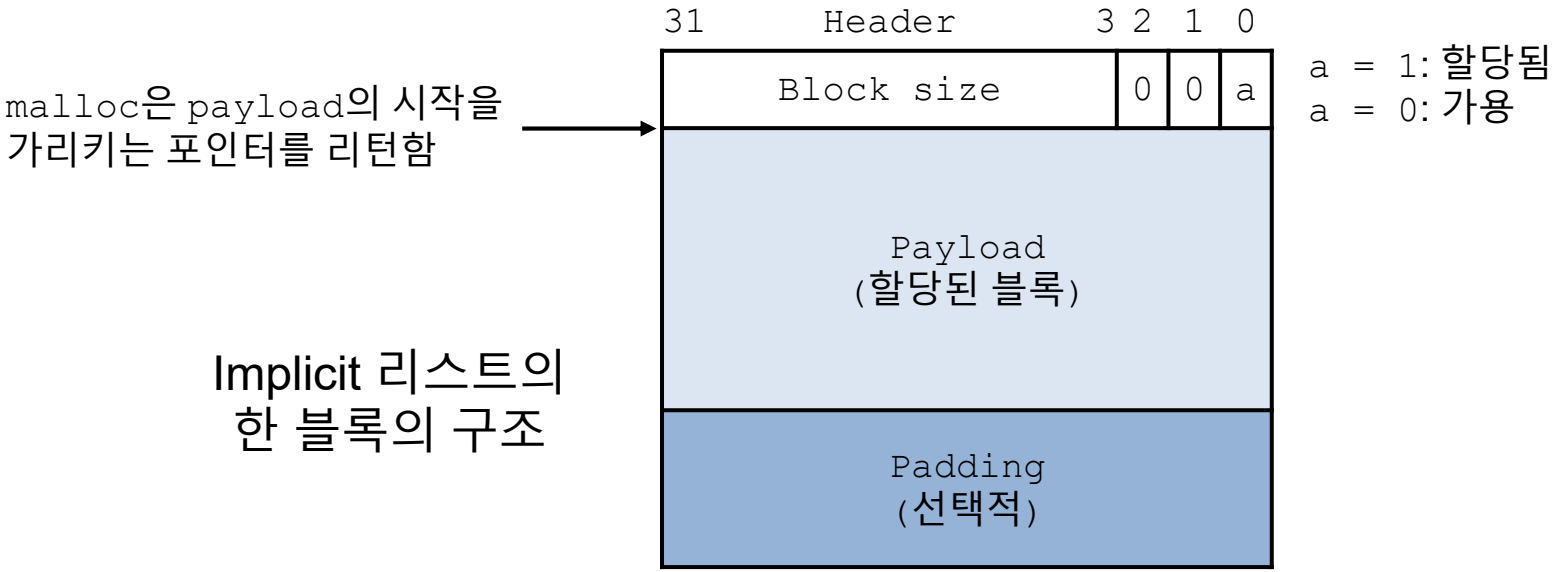
④ `malloc(500);`
→ 외부 단편화 발생

메모리 할당기

- 메모리 할당기는 다음 이슈들을 고려해야 함
  - **Tracking**: 가용 블록을 어떻게 추척할까?
  - **Placement**: 새로 할당된 블록을 배치할 적절한 가용 블록을 어떻게 선택할까?
  - **Splitting**: 새로 할당된 블록을 일부 가용 블록에 배치한 후, 나머지 가용 블록은 어떻게 정리할까?
  - **Coalescing**: 해제된 블록은 어떻게 정리할까?

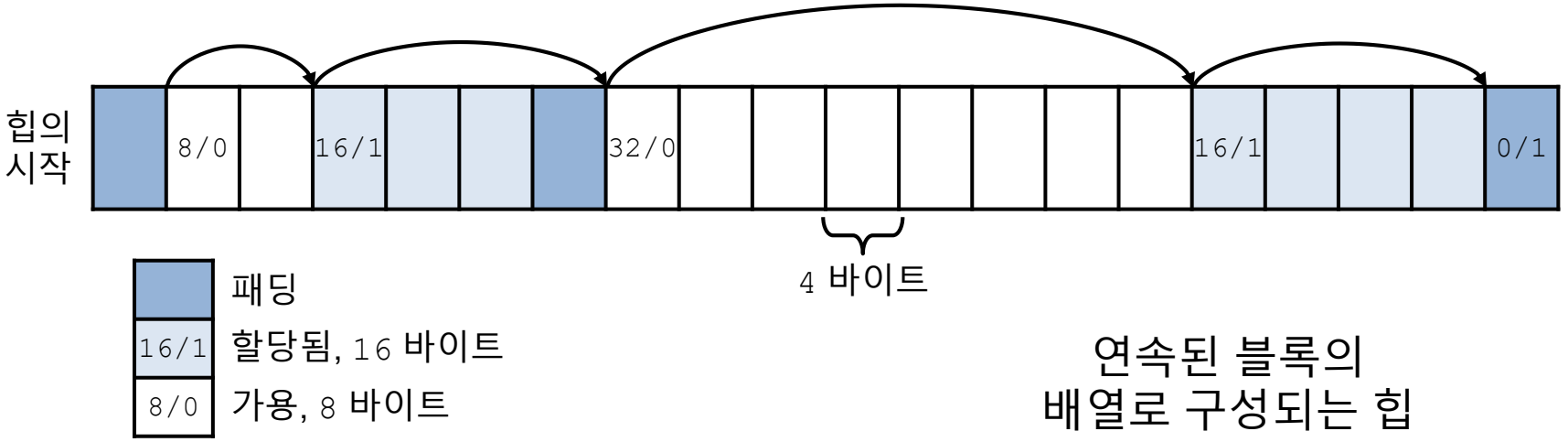
메모리 할당기

- Tracking
  - Implicit free list(묵시적 가용 리스트)
    - 할당된 블록과 가용 블록이 모두 연결되어 있음
    - 할당된 블록과 가용 블록을 구분하는 데이터는 블록 내에 내장됨



메모리 할당기

- Tracking
  - Implicit free list(묵시적 가용 리스트)
    - 할당된 블록과 가용 블록이 모두 연결되어 있음
    - 할당된 블록과 가용 블록을 구분하는 데이터는 블록 내에 내장됨



메모리 할당기

- Tracking
  - Explicit free list(명시적 가용 리스트)
    - 가용 블록끼리만 연결되어 있음
    - 메모리 할당: 리스트에서 블록을 삭제함
    - 메모리 해제: 리스테에 블록을 추가함

메모리 할당기

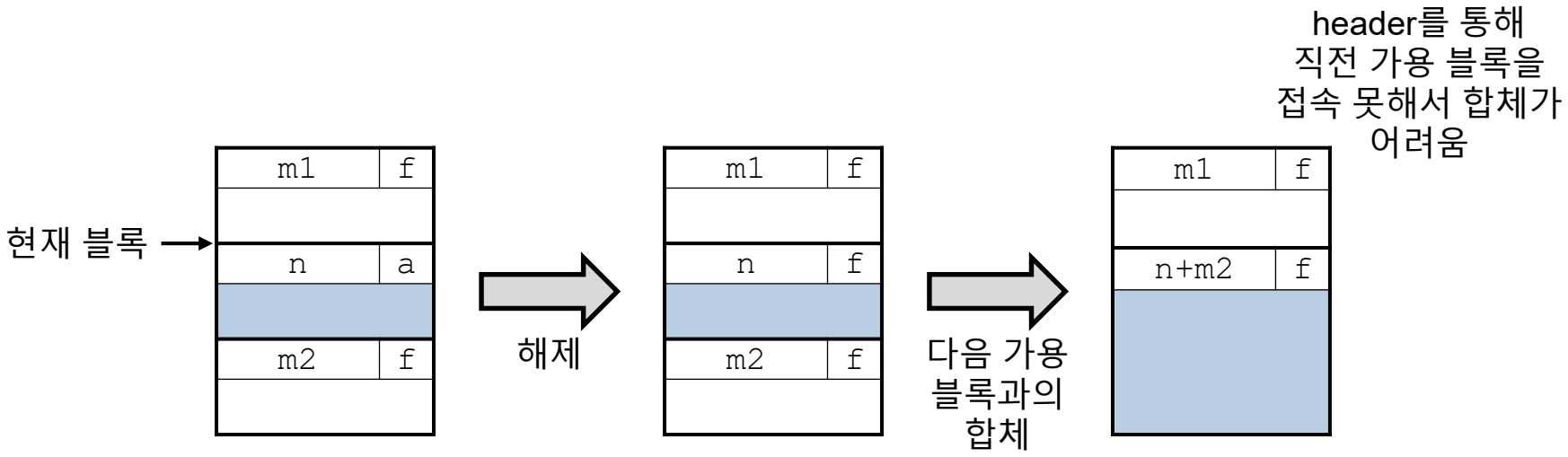
- Placement
  - First-fit
    - 처음부터 탐색하고 크기가 맞는 첫 가용 블록을 선택함
    - ☺ 장점: 리스트의 끝 부분에 큰 가용 블록을 갖게 되는 경향
    - ☹ 단점: 리스트의 앞 부분에 작은 가용 블록 조각을 남겨두는 경향
  - Next-fit
    - 처음부터 탐색하는 대신 이전 검색이 중단된 시점에서부터 검색함
    - ☺ 장점: first-fit보다 훨씬 빨리 실행될 수 있음
    - ☹ 단점: 해당 지점에서 마지막까지 찾지 못했다면, 결국 처음부터 찾아야 함
  - Best-fit
    - 모든 가용 블록을 검사하여 크기에 맞는 가용 블록 중 가장 작은 블록을 선택함
    - ☺ 장점: 메모리 활용도가 높음
    - ☹ 단점: 시간이 오래 걸림

메모리 할당기

- Splitting
  - 가용 블록 전체 할당
    - 간단하고 빠르게 구현할 수 있음
    - 내부 단편화를 도입함
  - 둘로 나눠 할당
    - 할당할 가용 블록을 두 부분으로 분할함
    - 첫 부분은 할당된 블록이 되고, 나머지 부분은 새로 가용 블록이 됨

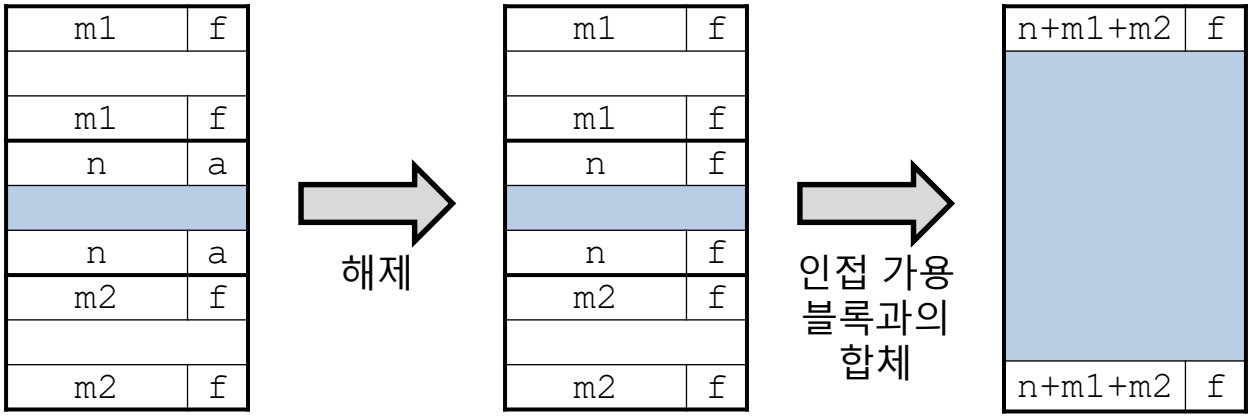
메모리 할당기

- Coalescing
  - 인접한 가용 블록을 합체함
  - 현재 블록을 해제하고 다음 가용 블록과 합체하는 것은 간단하지만, 직전 가용 블록과 합체하는 것은 어려움
    - 전체의 리스트를 검색해야 하기 때문에 시간이 오래 걸림



메모리 할당기

- Coalescing
  - 직전 가용 블록과의 합체를 위해 경계 태그(boundary tag) 활용
    - 블록 끝에 header와 같은 정보를 갖는 footer를 사용함
    - Footer를 통해 직전 블록 정보를 접속할 수 있으므로 직전 가용 블록과의 합체를 빠르게 할 수 있음

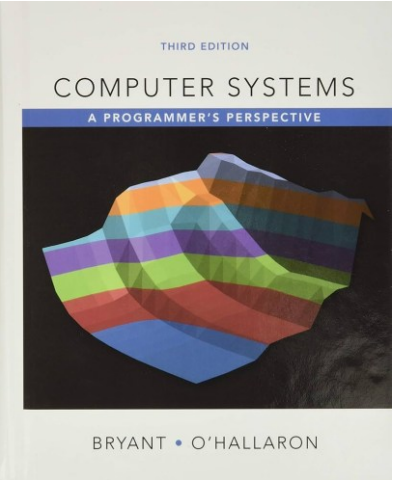


메모리 할당기

- Coalescing
  - 경계 태그를 사용하는 방법은 간단하지만 메모리 활용도에 좋지 않음
  - 실제적으로 효율적인 병합을 위해 doubly linked list(`next`와 `prev`의 포인터 2개는 header에서 저장)를 활용함
  - 즉시 병합: 가용 블록 사용될 때마다 병합 호출
  - 지연 병합: 필요할 때까지 병합을 연기함으로써 성능 개선

메모리 할당기

- 간소화한 메모리 할당기 구현
  - *Computer Systems: A Programmer's Perspective*에서
    제9장 *Virtual Memory*
    9.9 *Dynamic Memory Allocation*
    9.9.12 *Implementing a Simple Allocator*



- Title: *Computer Systems: A Programmer's Perspective 3rd Edition*
- Authors: *Randal Bryant, David O'Hallaron*
- Publisher: *Pearson*
- Date: *2015.03.02.*
- ISBN-10 *013409266X*
- ISBN-13 *978-0134092669*

malloc()

- GNU libc (glibc)에서의 malloc 구현
  - Tracking
    - 작은 블록(≤ 256KB): segregated linked list
    - 큰 블록(> 256KB): 트리 기반 자료구조(보통 빠른 탐색을 위해 BST 사용)
    - Header 정보
      - 블록 크기
      - 플래그: allocated 또는 free
      - Doubly linked list 포인터
  - Placement
    - 작은 블록 요청: first-fit
    - 큰 블록 요청: best-fit
  - Splitting
    - 요청 크기는 가용 블록 크기보다 작으면 블록 분할을 적용함
  - Coalescing
    - 지연 병합

동적 할당 관련 팁

- 메모리 단편화를 줄이기 위해 `malloc` 사용을 권장
- 할당된 메모리를 조절할 때 `realloc` 사용
- 동적 메모리 할당/해제를 많이 사용하면 불필요한 splitting/coalescing으로 인해 프로그램의 성능이 저하됨
- 유닉스 또는 유닉스 계열 운영체제에서 메모리 누수를 검사하려면 `valgrind`를 사용할 수 있음

```
gcc -Wall test.c -o test
valgrind --leak-check=yes ./test
```

동적 할당 관련 팁

- valgrind 사용 예시

```c
#include <stdlib.h>
void func() {
    int *x = (int*)malloc(10*sizeof(int));
    x[10] = 0;
}
int main() {
    func();
    return 0;
}
```

```
HEAP SUMMARY:
    in use at exit: 40 bytes in 1 blocks
  total heap usage: 1 allocs, 0 frees, 40 bytes allocated

40 bytes in 1 blocks are definitely lost in loss record 1 of 1
   at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
   by 0x10915E: func (in /home/dat/Downloads/test_C_valgrind/test)
   by 0x109185: main (in /home/dat/Downloads/test_C_valgrind/test)

LEAK SUMMARY:
   definitely lost: 40 bytes in 1 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 0 bytes in 0 blocks
        suppressed: 0 bytes in 0 blocks

For lists of detected and suppressed errors, rerun with: -s
ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

쓰레기 수집기(garbage collector)

- 해제되지 않은 메모리는 프로그램 종료 때까지 가상 메모리 공간에서 남아 있음
- C 언어에서 쓰레기 수집기는 구현되지 않음
  - 할당: `malloc()`, `calloc()`, `realloc()`
  - 해제: `free()`
  - 메모리 누수는 프로그래머 책임
- 필요 시 쓰레기 수집기를 구현해야 함
  - 참조 카운팅(reference counting)
  - Mark-and-Sweep
  - 보수적 쓰레기 수집기(conservative garbage collector)

쓰레기 수집기(garbage collector)

- 참조 카운팅(reference counting)
  - 객체마다 참조 횟수를 저장함
  - 참조 횟수가 0이 되면 메모리 해제
  - ☺ 장점: 간단하고 결정적임
  - ☹ 단점: 순환 참조(circular reference)는 처리 불가
- Mark-and-Sweep
  - 힙을 탐색하여 도달 가능한 객체를 표시함(mark)
  - 표시되지 않은 객체를 해제함(sweep)
  - ☺ 장점: 순환 참조는 처리 가능
  - ☹ 단점: 시간이 오래 걸림
- 보수적 쓰레기 수집기(conservative garbage collector)
  - 스택과 힙을 스캔하여 포인터처럼 보이는 값을 살아있는 참조로 판단됨
  - ☺ 장점: 기존 C 코드 수정 필요 없이 사용 가능
  - ☹ 단점: 일부 메모리를 불필요하게 유지할 수 있음