

Lab. 4 메모리

1) 메모리 관리와 통계 정보

: free 명령어로 시스템의 총 메모리 양과 사용중인 메모리의 양을 측정할 수 있다. 자신의 시스템 메모리를 측정한다.

```
$ free
```

	① total	used	② free	shared	③ buff/cache	④ available
Mem:	32942000	337640	30641272	18392	1963088	32000464
Swap:	0	0	0			
\$						

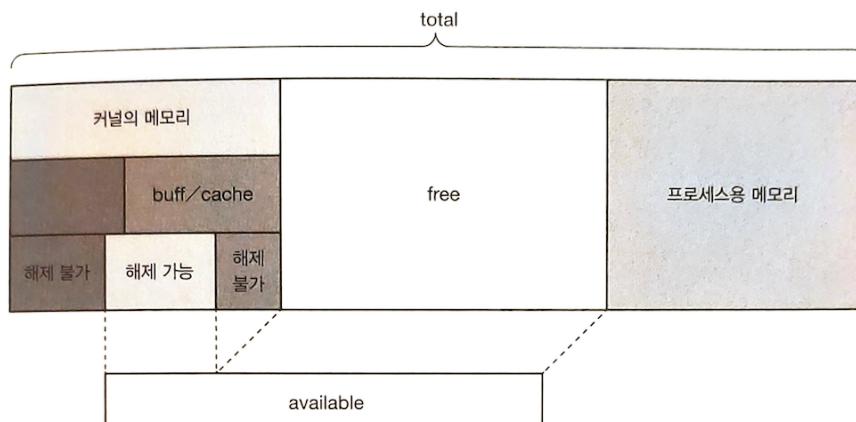
① **total** : 시스템에 탑재된 전체 메모리 용량입니다. 위의 예에서는 32기가바이트입니다.

② **free** : 표기상 이용하지 않는 메모리입니다(available 필드 참조).

③ **buff/cache** : 버퍼 캐시 또는 페이지 캐시(6장에서 설명)가 이용하는 메모리입니다. 시스템의 빈 메모리 (free 필드의 값)가 부족하면 커널이 해제합니다.

④ **available** : 실직적으로 사용 가능한 메모리입니다. free 필드값의 메모리가 부족하면 해제되는 커널 내의 메모리 영역 사이즈를 더한 값으로, 해제될 수 있는 메모리에는 버퍼 캐시나 페이지 캐시의 대부분 혹은 다른 커널 내의 메모리 일부가 포함됩니다.

free 명령어의 각 내용을 아래의 그림으로 표현 할 수 있다.



sar 명령어를 사용하면 실시간으로 (두 번째 파라미터 지정 값 간격) 통계 정보를 얻을 수 있다. 직접 실행해 본다.

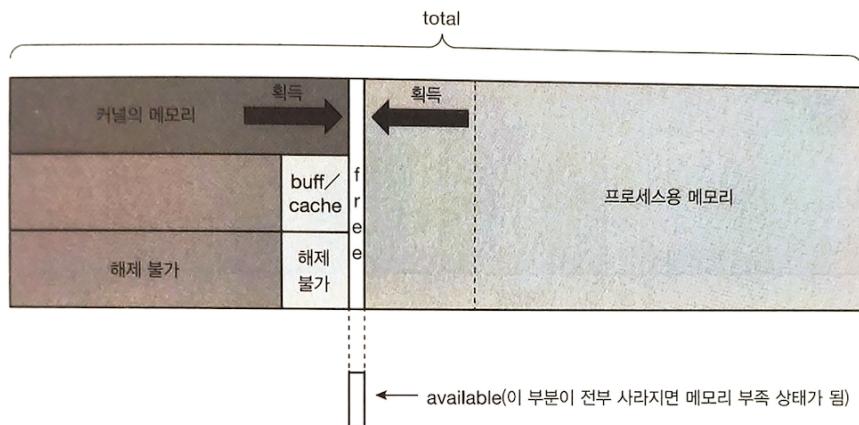
```
$ sar -r 1
```

	08:19:40	kbmemfree	kbmemused	%memused	kbbuffers	kbcached	kcommit	%commit
(중략)								
kbactive	28892368	4049632	12.29	5980	3117188	2127556	6.46	
kbinact	2413616	937524	112					

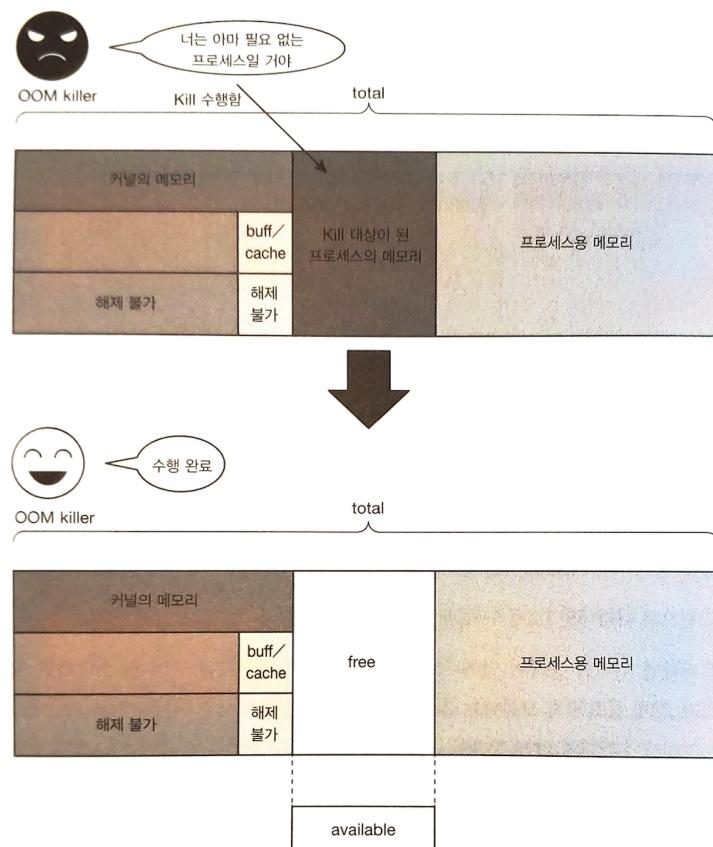
free와 sar 명령어에서 사용된 각 필드를 의미별로 비교하면 다음 표와 같다.

free 명령어	total	free	buff/cache	available
sar -r 명령어	없음	kbmemfree	kbbuffers + kbcached	없음

커널과 사용자 프로세서가 계속 메모리 사용량을 늘린다면 아래 그림과 같이 메모리 부족 상태가 발생한다.



메모리 부족 상태가 발생하면 메모리 관리 시스템은 적절한 프로세스를 선택해서 강제종료하고 메모리를 해제하는 OOM Killer라는 기능을 사용할 수 있다.



잘못된 메모리 주소에 접근할 때 발생하는 오류를 직접 관찰하는 실험을 수행하기 위해 아래 코드를 수행한다.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *p = NULL;
    puts("before invalid access");
    *p = 0;
    puts("after invalid access");
    exit(EXIT_SUCCESS);
}
```

표 1 segv.c

```
$ cc -o segv segv.c
```

테스트 결과를 분석하고 해석해서 보고서에 기록한다.

2) 메모리 할당 실험

프로세스를 생성할 때 메모리가 할당된다. 실행 파일에 기록된 정보를 사용해서 메모리 크기 를 판단하고 필요한 메모리 사이즈를 구하게 된다.

- 필요한 메모리 크기 = 코드 영역 크기 + 데이터 영역 크기

다음의 mmap.c 프로그램을 통해 메모리 할당 동작을 실험해 본다. 프로그램은 아래와 같이 동작한다.

- 프로세스의 메모리 맵 정보(/proc/pid/maps) 출력 /* pid는 각 프로세스의 번호 */
- 메모리를 새로 100 메가바이트 확보한다
- 다시 메모리 맵 정보를 표시한다.

```
#include <unistd.h>
#include <sys/mman.h>
#include <stdio.h>
#include <stdlib.h>
#include <err.h>

#define BUFFER_SIZE 1000
#define ALLOC_SIZE (100*1024*1024)

static char command[BUFFER_SIZE];

int main(void)
```

```

{
    pid_t pid;

    pid = getpid();
    snprintf(command, BUFFER_SIZE, "cat /proc/%d/maps", pid);

    puts("*** memory map before memory allocation ***");
    fflush(stdout);
    system(command);

    void *new_memory;
    new_memory = mmap(NULL, ALLOC_SIZE, PROT_READ | PROT_WRITE,
MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (new_memory == (void *) -1)
        err(EXIT_FAILURE, "mmap() failed");

    puts("");
    printf("*** succeeded to allocate memory: address = %p; size = 0x%lx ***\n",
           new_memory, ALLOC_SIZE);
    puts("");

    puts("*** memory map after memory allocation ***");
    fflush(stdout);
    system(command);

    if (munmap(new_memory, ALLOC_SIZE) == -1)
        err(EXIT_FAILURE, "munmap() failed");
    exit(EXIT_SUCCESS);
}

```

표 2 mmap.c

mmap() 함수는 페이지 단위로 메모리를 할당하는 시스템 콜을 호출한다. system() 함수는 파라미터에 지정된 명령어를 수행한다.

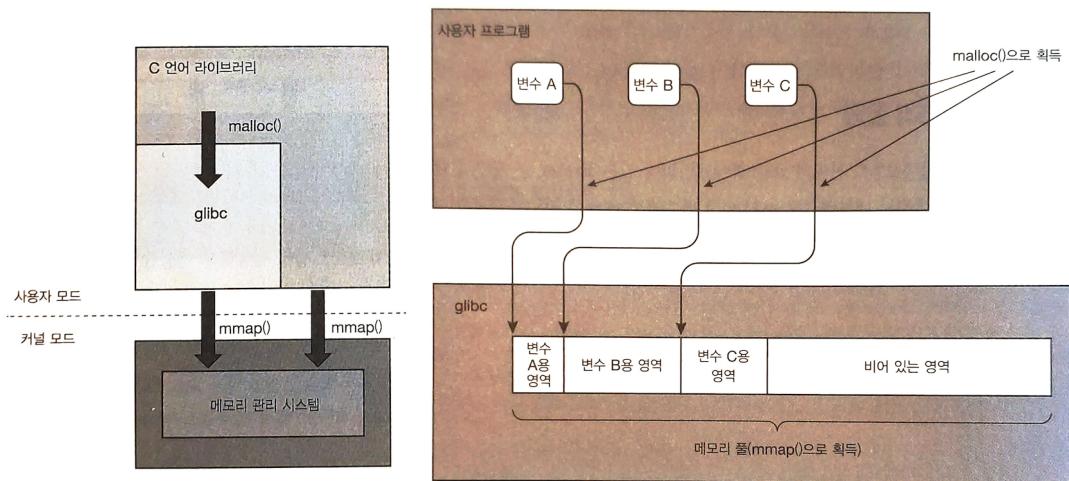
```

$ cc -o mmap mmap.c
$ ./mmap

```

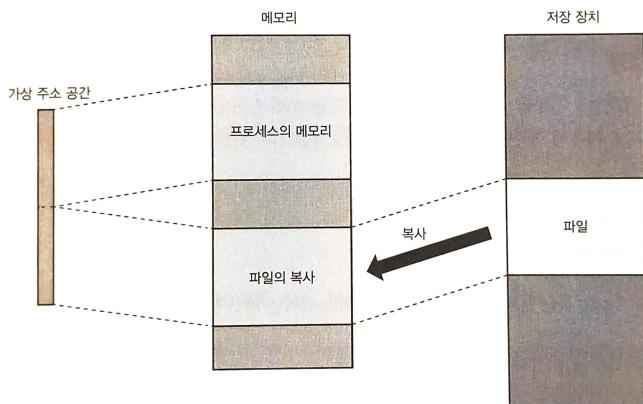
실험을 수행한 후 메모리 주소를 중심으로 어느 주소에 새 메모리가 할당되었는지 분석하고 보고서에 기록한다.

C 언어와 같은 고수준 언어에서는 malloc() 함수를 통해 바이트 단위로 메모리를 할당 할 수 있다. mmap()과 malloc() 함수의 메모리 할당 단위가 다르므로 glibc는 사전에 mmap()으로 메모리를 할당해 두고 필요한 양 만큼 사용한다. 또한 파일과 같이 직접 메모리를 관리하지 않는 스크립트 언어의 오브젝트 생성도 최종적으로 내부에서 C 언어의 malloc() 함수를 사용하고 있다.



3) 파일 맵 실험

파일 맵(file map)은 가상 메모리를 응용한 리눅스의 중요 동작 중 하나이다. 프로세스가 파일을 연 뒤에 저장 장치의 파일을 직접 읽고, 쓰기 할 수 있지만 메모리에 매핑(저장) 한 후 사용할 수도 있다. 수정된 메모리의 파일 내용은 적절한 시점에 저장 장치에 저장된다. 아래 그림은 파일 맵을 그린 그림이다.



파일 맵 기능을 사용하는 프로그램을 통해 파일 매핑의 수행과 내용에 접근이 가능한지 테스트 한다. 아래와 같은 항목을 확인한다.

- 파일이 가상공간에 매핑되어 있는가
- 매핑된 영역을 읽으면 파일이 실제로 읽어지는가
- 매핑된 영역에 쓰기를 하면 실제 써지는가

테스트를 위해 먼저 파일을 작성한다.

```
$ echo hello > testfile
```

filemap.c 프로그램은 아래와 같은 사양을 가진다.

- 프로세스의 메모리 맵 정보(/proc/pid/mmaps)를 출력

- testfile을 열어둔다
- 파일을 mmap()으로 메모리 공간에 매핑한다
- 프로세스의 메모리 맵 정보를 다시 출력한다
- 매핑된 영역의 데이터를 읽어 들여 표시
- 매핑된 영역의 데이터를 덮어쓴다

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <err.h>

#define BUFFER_SIZE      1000
#define ALLOC_SIZE       (100*1024*1024)

static char command[BUFFER_SIZE];
static char file_contents[BUFFER_SIZE];
static char overwrite_data[] = "HELLO";

int main(void)
{
    pid_t pid;

    pid = getpid();
    snprintf(command, BUFFER_SIZE, "cat /proc/%d/maps", pid);

    puts("*** memory map before mapping file ***");
    fflush(stdout);
    system(command);

    int fd;
    fd = open("testfile", O_RDWR);
    if (fd == -1)
        err(EXIT_FAILURE, "open() failed");

    char * file_contents;
    file_contents = mmap(NULL, ALLOC_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
    if (file_contents == (void *) -1) {
        warn("mmap() failed");
        goto close_file;
    }

    puts("");
    printf("*** succeeded to map file: address = %p; size = 0x%x ***\n",
file_contents, ALLOC_SIZE);

    puts("");
    puts("*** memory map after mapping file ***");
    fflush(stdout);
    system(command);
```

```

puts("");
printf("*** file contents before overwrite mapped region: %s", file_contents);

// overwrite mapped region
memcpy(file_contents, overwrite_data, strlen(overwrite_data));

puts("");
printf("*** overwritten mapped region with: %s\n", file_contents);

if (munmap(file_contents, ALLOC_SIZE) == -1)
    warn("munmap() failed");

close_file:
if (close(fd) == -1)
    warn("close() failed");
exit(EXIT_SUCCESS);
}

```

표 3 filemap.c

```

$ cc -o filemap filemap.c
$ ./filemap

```

`mmap()` 함수의 실행과 `testfile`의 데이터 주소가 어디에 매핑되었는지 확인해야 한다. `mmap()` 함수에 의해 추가된 공간과 거기에 `testfile`이 매핑된 결과를 확인하고 데이터를 읽고 덮어쓴 흔적을 확인할 수 있어야 한다. 확인은 다음과 같이 한다. 결과를 분석한다.

```
$ cat testfile
```

4) 스와핑 실험

스와핑은 메모리가 부족할 때 선정된 프로세스를 저장장치의 스왑 영역으로 보관하고 메모리 영역을 확보하는 전략이다. 메모리가 항상 부족한 시스템일 경우 프로세스가 메모리에 접근할 때마다 스와핑이 발생하여 스왑 인, 스왑 아웃이 반복되는 스래싱 현상이 발생할 수 있다. 이때 시스템 성능이 심각하게 저하되어 사용자 입력이 반응할 수 없게 된다.

시스템의 스왑 영역은 아래와 같이 확인할 수 있다.

```

$ swapon -show
저장장치 어느 영역에 스왑 영역이 설정되어 있는지 확인해 보자.

스왑 영역은 free 명령어로도 확인이 가능하다. 실행해 본다.

$ free

```

시스템에 부하가 걸릴 때는 아래와 같이 스와핑 발생 상태를 확인해보는 것도 도움이 된다.

```
$ sar -W 1
```

```
$ sar -S
```

위 명령어에서도 `kbswpused` 필드가 나타내는 스왑 영역의 사용량 추이가 점점 증가하고 있

다면 위험도가 증가하고 있다는 의미이다.