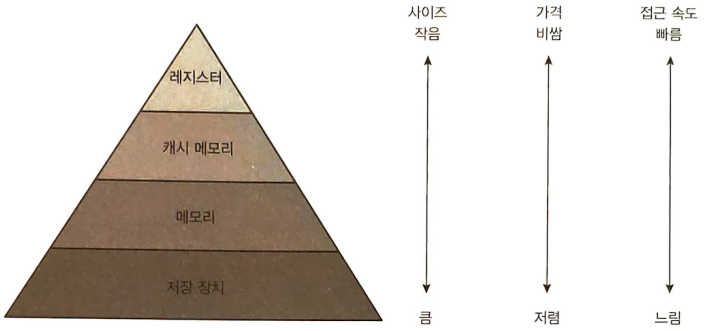
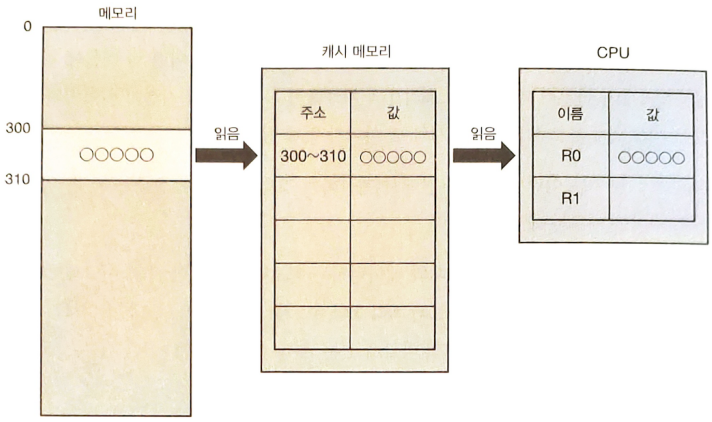
Lab. 5 메모리 계층

: 프로세스와 레지스터의 처리속도는 DRAM 메모리에 비해 매우 빠르다. 모든 프로세스를 실행할 때 메모리에 데이터와 소스코드가 저장되어 있으므로 프로세서는 메모리에서 데이터와 소스 코드를 읽고 써야하는데 메모리의 늦은 처리 속도가 실행속도를 늦추는 결정적인 방해 요소가 된다. 이러한 문제점을 완화하기 위해서 아래 그림과 같은 계층적 메모리 구조를 사용한다.



아래 그림은 캐시 메모리가 적용된 상태에서 프로세서가 데이터를 읽을 때 과정을 보여준다. 반대 순서는 쓰기 과정이 된다.



1) 캐시 메모리 실험

캐시 메모리도 계층형 구조로 만들어지는 경우가 많다. 각 계층마다 크기, 응답 시간 CPU간 공유 등이 다르다. 계층형 케시는 L1, L2 등의 이름이 붙고, L1처럼 숫자가 낮으면 CPU에 더 가깝고 빠르며 크기는 더 작다. 캐시 메모리 정보는 /sys/devices/system/cpu/cpu0/cache/index0/[1] 디렉터리에 파일 내용을 살펴보면 파악할 수 있다. 예제로 살펴보면 아래 그림과 같다. 자신의 시스템에 대해 파악하고 아래와 같은 표를 작성해 보자

1) cpu0, L1 캐시의 경우임.

파일명	이름	종류	공유하는 논리 CPU	사이즈 (킬로바이트)	캐시 라인 사이즈 (바이트)
index0	L1d	데이터	모든 논리 CPU별로 존재	32	64
index1	L1i	코드	모든 논리 CPU별로 존재	64	64
index2	L2	데이터와 코드	모든 논리 CPU별로 존재	512	64
index3	L3	데이터와 코드	0-3, 4-7 공유	8192	64

캐시 메모리의 영향으로 프로세스가 접근하는 데이터의 크기에 따라 데이터를 읽고 쓰는 속도의 변화를 측정하는 실험을 수행한다. 프로그램 cache.c 는 다음의 기능을 수행한다.

- 명령의 첫 번째 파라미터 입력 값을 크기로 (단위는 킬로바이트) 메모리를 확보
- 확보한 메모리 영역 안에 정해진 횟수만큼 순차 접근을 수행
- 한 번 접근할 때마다 걸린 소요 시간을 표시 ($\frac{\text{윗줄 소요시간(나노초)}}{\text{윗줄 접근 횟수}}$)

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

#define CACHE_LINE_SIZE 64
#define NLOOP           (4*1024UL*1024*1024)
#define NSECS_PER_SEC   1000000000UL

static inline long diff_nsec(struct timespec before, struct timespec after)
{
        return ((after.tv_sec * NSECS_PER_SEC + after.tv_nsec)
                - (before.tv_sec * NSECS_PER_SEC + before.tv_nsec));
}

int main(int argc, char *argv[])
{
        char *progname;
        progname = argv[0];

        if (argc != 2) {
                fprintf(stderr, "usage: %s <size[KB]>₩n", progname);
                exit(EXIT_FAILURE);
        }

        register int size;
        size = atoi(argv[1]) * 1024;
        if (!size) {
                fprintf(stderr, "size should be >= 1: %d₩n", size);
                exit(EXIT_FAILURE);
        }

        char *buffer;
```

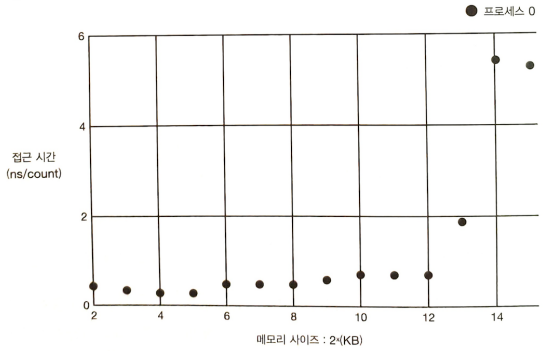
```
        buffer = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE |
MAP_ANONYMOUS, -1, 0);
        if (buffer == (void *) -1)
                err(EXIT_FAILURE, "mmap() failed");

        struct timespec before, after;

        clock_gettime(CLOCK_MONOTONIC, &before);

        int i;
        for (i = 0; i < NLOOP / (size / CACHE_LINE_SIZE); i++) {
                long j;
                for (j = 0; j < size; j += CACHE_LINE_SIZE)
                        buffer[j] = 0;
        }

        clock_gettime(CLOCK_MONOTONIC, &after);

        printf("%f₩n", (double)diff_nsec(before, after) / NLOOP);

        if (munmap(buffer, size) == -1)
                err(EXIT_FAILURE, "munmap() failed");
        exit(EXIT_SUCCESS);
}
```

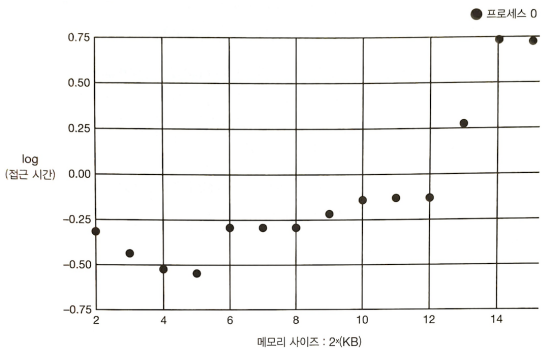
표 1 cache.c

$ cc -O3 -o cache cache.c
컴파일할 때 미세한 시간 차이를 측정하기 위해서 최적화 옵션 -O3를 넣어야 한다. 실험은 4 킬로바이트부터 32메가바이트까지 데이터 크기를 두 배씩 늘리면서 수행한다.

$ for i in 4 8 16 3264 128 256 1024 2048 4096 8192 16384 32768 ; do ./cache $i ; done

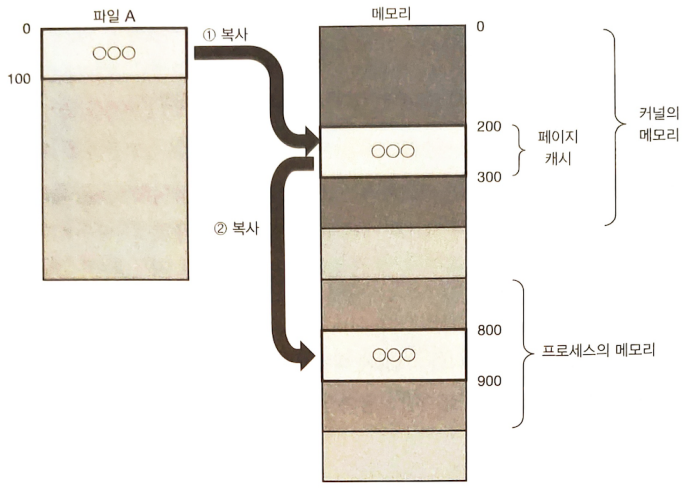
아래 2개의 그림은 위 수행 결과 - 접근 시간을 y축으로, 메모리 크기를 x축으로 표시한 것이다. 아래 우측 그림은 y축을 log 단위로 변경해서 차이가 잘 보이도록 조정한 것이다. 여러분의 시스템에서 측정하고 그 결과를 아래와 같은 그래프로 표시하고 분석해서 보고서를 작성한다.





2) **페이지 케시 실험**

*페이지 캐시*는 메모리 접근 속도와 하드 디스크와 같은 저장 장치의 접근 속도가 매우 큰 차이가 있어서 메모리 영역에 마련한 중재 구역이다. 프로세스가 파일의 데이터를 읽어 들일 때 커널은 프로세스의 메모리에 직접 파일 데이터를 복사하는 대신 커널 메모리 영역의 페이지 캐시에 복사한 뒤 프로세스 메모리에 복사한다. 전원이 강제로 꺼지는 경우에는 페이지 캐시의 내용은 사라지게 된다. 이러한 일이 절대 발생하면 안되는 경우에는 open() 함수를 호출할 때 O_SYNC 플래그 설정을 통해 항시 저장장치와의 동기화를 강제할 수 있다.



페이지 캐시의 성능을 알아보기 위해 같은 파일 읽기를 2번 실행하고 소요시간을 비교해 본다. 1기가 바이트 크기의 testfile 파일을 만든다.

$ dd if=/dev/zero of=testfile oflag=direct bs=1M count=1K
여기서 'oflag=direct'는 다이렉트 I/O 방법을 사용해서 파일을 쓰도록 한다. 즉, 페이지 캐시를 사용하지 않도록 한다. 따라서 이 시점에는 페이지 파일이 없다.

아래 명령으로 testfile을 읽고 페이지 캐시 사용량을 측정한다.

$ free
$ time cat testfile > /dev/null
$ free
time 명령에 따라 파일을 읽는데 걸린 소요시간이 나타났을 것이다. 'real' 소요시간 중에 'sys'가 얼마의 비중을 차지하는가? 처음 파일에 접근하기 때문에 저장 장치를 읽기 위해 시스템 콜을 호출해서 CPU가 처리하는 시간이 'sys'에 나타나는 것이다. 나머지 시간 'real' - 'sys'는 읽기가 끝나도록 저장장치를 기다리는 시간이다. 이 값은 여러분의 시스템에서 얼마인가? 또한 free 명령을 통해 메모리 크기를 볼 때 'buff/cache' 크기가 증가하고 있다. 증가한 크기는 어느 정도이며, 이것은 어떤 의미인지 분석한다.

이제 두 번째 읽기를 수행한다.

$ time cat testfile > /dev/null
$ free

두 번째 실험 결과는 어떻게 변화했는가? 변화된 결과에서 페이지 캐시의 작용으로 설명할 수 있는 부분은 무엇인지 분석하고 기록한다.

실험 후 파일은 삭제한다.

$ rm testfile

위의 테스트를 수행할 때 시스템의 통계 정보를 확인해 본다. 아래의 세 가지 정보를 가져온다.
- 저장 장치로부터 페이지 캐시에 데이터를 읽은 페이지 인 횟수
- 페이지 캐시로부터 저장 장치에 페이지를 쓴 페이지 아웃 횟수
- 저장 장치에 대한 I/O의 양

테스트의 편의를 위해 아래의 스크립트를 통해 수행한다.

```
#!/bin/bash

rm -f testfile

echo "$(date): start file creation"
dd if=/dev/zero of=testfile oflag=direct bs=1M count=1K
echo "$(date): end file creation"

echo "$(date): sleep 3 seconds"
sleep 3

echo "$(date): start 1st read"
cat testfile >/dev/null
echo "$(date): end 1st read"

echo "$(date): sleep 3 seconds"
sleep 3

echo "$(date): start 2st read"
cat testfile >/dev/null

echo "$(date): end 2nd read"

rm -f testfile
```

표 2 read-twice.sh

스크립트를 아래와 같이 수행하고, 동시에 페이지 인, 아웃 정보를 얻기 위해서 다른 터미널을 열어서 sar -B 명령을 수행한다.
$ ./read-twice.sh

(다른 터미널에서) $ sar -B 1

두 개 명령의 출력을 비교하면 페이지 인, 아웃 발생 숫자를 파악할 수 있습니다. 또한 파일 작성시, 첫 번째로 파일을 읽을 때, 두 번째로 읽을 때를 구분해서 페이지 인, 아웃 발생을 파악할 수 있습니다. 이러한 점을 분석해서 보고서를 작성한다.

다음으로 저장 장치레 발생한 I/O의 양을 확인한다. 'sar -d -p' 명령을 이용해서 저장 장치별로 통계 정보를 표시한다. 일단 쓰기의 대상이 되는 파일 시스템, 즉 루트 파일시스템이 존재하는 장치의 이름을 확인한다.

```
$ mount | grep "on /"
/dev/sda5 on  / type btrfs (rw, ... /* 이하 생략 */ )
$
```

예제의 시스템에서는 '/dev/sda5'이다. sda 저장 장치의 5 파티션을 의미한다. 아래와 같은 명령으로 데이터를 감시한다. 아래 명령을 다른 터미널에서 실행하면서 동시에 read-twice.sh을 다시 실행한다.

```
$ ./read-twice.sh
(다른 터미널에서) $ sar -d -p 1
```

sar 명령의 결과에서 rd_sec/s와 wr_sec/s가 각각 1초당 개별 저장 장치 (여기서는 sda)에 대해 읽고 쓴 데이터의 양이다. 단위는 섹터이며 512바이트이다. 결과를 해석하고 보고서를 작성한다. 참고로 %util은 측정 시간 (명령에서 1초) 동안 저장장치에 접근한 시간의 할당량은 의미한다.