

**프로파일링**

**Profiling**

**2024 겨울 자료구조스터디**

**김태완**

**2024/02/15**

- 시간 측정 방법
- Valgrind

# 시간 측정 방법

# 시간 측정 방법

## time 라이브러리

- 시간을 측정할 코드에서 시간 측정 함수를 직접 추가해서 결과를 출력하는 방식
- **<time.h>** 헤더파일을 include해야 사용 가능함.

### time\_t 타입

- POSIX Time(1970년 1월 1일 0시 0분 0초)을 기준으로 현재 시점까지 시간을 초 단위의 값으로 표현하는 타입 (long long)

### time\_t time(time\_t \*pTime)

- 매개변수: 현재 시점의 시간을 저장할 time\_t 타입의 포인터 변수
- 리턴값: 현재 시점의 시간을 time\_t 타입으로 반환

### clock\_t 타입

- 프로세스가 실행되는 동안 소모되는 CPU의 클럭 수 (long)
- 이를 **CLOCKS\_PER\_SEC(1,000,000)** 매크로 상수로 나누면 초 단위로 변환된 시간을 구할 수 있음.

### clock\_t clock(void)

- 리턴값: 현재 시점의 시간을 clock\_t 타입으로 반환, 실패 시 -1 반환

### time()과 clock()의 차이

- sleep() 등 함수를 사용하거나 scanf()로 사용자 입력을 받아 프로세스가 실행을 멈출 때, clock()은 멈춘 시간을 포함하지 않음.
- time()은 초 단위, clock()은 클럭 틱 단위(마이크로초)로 측정함.

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

int main(void){
    time_t start, end;
    double elapsed_time;

    time(&start); // 시간 측정 시작

    // 시간을 측정할 부분

    time(&end); // 시간 측정 종료

    // 경과 시간 계산 (초 단위)
    elapsed_time = difftime(end, start);
    printf("Execution time: %.f seconds\n",
        elapsed_time);
```

**time 함수를 사용하여 시간을 측정하는 예시**

## time 라이브러리

**int clock\_gettime(clockid\_t clk\_id, struct timespec \*tp);**

- 나노초(10의 -9승) 단위로 시간을 구할 수 있는 함수
- 파라미터
  - **clk\_id**: 측정할 클럭 시간의 종류를 지정
  - **\*tp**: 측정한 시간을 저장할 구조체
- 리턴값: 성공 시 0, 실패 시 -1 반환

### clockid\_t 타입의 종류

- **CLOCK\_REALTIME**: 시스템 전역의 실제 시간(POSIX Time)을 구함.
- **CLOCK\_MONOTONIC**: 부팅 이후로부터 흐른 시간을 구하는 단조 시계로, 두 이벤트의 시간 차이를 구할 때 주로 사용
- 나머지 타입은 **man clock\_gettime** 명령어를 입력하여 확인 가능함.

### timespec 구조체

```
struct timespec
{
    time_t tv_sec; // Seconds - >= 0
    long tv_nsec; // Nanoseconds - [0, 999999999]
};
```

## time 라이브러리

### 예제 (최성현 님의 queue 문제 풀이 코드)

- 시간을 측정할 부분의 앞뒤에 시간 측정 함수를 사용

```
#include "LinkedList.h"
#include <time.h>

int main(void){
    struct timespec start, end;
    double elapsed_time;
    int NumberOfCard;

    // 기존 코드
    // 시간 측정 시작
    clock_gettime(CLOCK_MONOTONIC, &start);

    //여기서부터 문제 해결
    // 기존 코드

    // 시간 측정 종료
    clock_gettime(CLOCK_MONOTONIC, &end);

    // 경과 시간 계산 (밀리초 단위)
    elapsed_time = ((end.tv_sec - start.tv_sec) * 1e3) + ((end.tv_nsec - start.tv_nsec) / 1e6);

    printf("Execution time: %.3f milliseconds\n", elapsed_time);
    printf("마지막 카드의 숫자는 %d입니다.", Queue->Front->Data);
}
```

## 유닉스 time 명령어

### 사용 방법

```
$ /usr/bin/time -p ./Problem1    # 시스템의 time 명령어
$ time -p ./Problem1            # shell에 포함된 time (덜 유용한 버전)
```

### 출력 결과

```
ktw@ktw-RTES:~/Documents/DataSet-Structure-Study/qpn$ time -p ./Problem1
카드 장수를 입력하십시오 : 20000000

Execution time: 1.167183352 seconds
마지막 카드의 숫자는 6445568입니다 .real 5.20
user 1.04
sys 0.14
ktw@ktw-RTES:~/Documents/DataSet-Structure-Study/qpn$ /usr/bin/time -p ./Problem1
카드 장수를 입력하십시오 : 20000000

Execution time: 1.195088538 seconds
마지막 카드의 숫자는 6445568입니다 .real 4.23
user 1.03
sys 0.18
```

### p 옵션

- **real**: 명령어가 호출부터 종료될 때까지 소요된 시간 (I/O 대기시간 포함)
- **user**: CPU가 커널 함수 외 작업을 처리할 때 소비한 시간
- **sys**: 커널 함수를 수행하는 데 소비한 시간 (File 접근, I/O 관리, 메모리 접근 등을 위한 system call)

**Valgrind**



## Valgrind란?

- C/C++ 소프트웨어 개발에서 프로그램의 성능을 최적화하는 데 사용되는 동적 분석 도구로, Linux에서 사용 가능함.
- ※ **동적 분석 도구**: 컴파일된 실행 파일을 사용하여, 프로그램을 직접 실행하면서 발생 가능한 문제점들을 찾아내는 도구
- ※ **정적 분석 도구**: 코드 파일을 사용하여, 코드 자체에서 원인을 분석하는 도구
- 모듈식 구조로 사용자가 목적에 맞는 도구를 선택할 수 있음.
  - **Memcheck**: 메모리 관리 관련 오류 검출
  - **Cachegrind**: 캐시 및 분기 예측 프로파일러
  - **Helgrind**: 스레드 오류 탐지기로, 멀티 스레드 프로그램 개발 시 사용됨.
  - 이밖에도 다양한 도구들을 제공함.
- 다음 명령어로 설치할 수 있음.
  - **sudo apt install valgrind**

## 공식 홈페이지

- <https://valgrind.org/> 에서 지원하는 플랫폼 및 아키텍처를 확인할 수 있고, 사용자 매뉴얼을 제공함.

## 실행 방법

- 코드 작성 후 컴파일
  - **gcc -g -o <실행파일 이름> <코드 경로>**
  - -g 옵션을 붙여야 문제가 발생한 코드 라인을 확인 가능함.
- Valgrind 명령어로 프로그램을 실행
  - **valgrind <tool 옵션> <command line 옵션> <프로그램 경로> <인자>**
  - **tool 옵션**: Valgrind에서 사용할 도구를 지정하며, 생략 시 기본값은 Memcheck임.

## Memcheck

- 프로그램이 할당한 모든 힙 블록을 추적하여, 프로그램 종료 시 해제되지 않은 블록을 파악 가능

### 유용한 command line 옵션

- **--leak-check=yes**
  - definitely lost, possibly lost 에 해당하는 각 블록에 대한 정보를 더 자세히 보여 줌
- **--log-file=<filename>**
  - Valgrind의 모든 메시지를 지정된 파일로 보냄
  - 특수 형식 지정자를 사용하여 로그 파일명을 동적으로 생성할 수 있음.

### 실습

- 오른쪽 그림의 코드(test.c)를 작성 후 컴파일
  - **gcc -g -o test ./test.c**
- Memcheck로 분석
  - **valgrind --leak-check=yes ./test**

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

void f(void){
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;
}

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

## 분석 메시지 해석

### 문제 1(Memory error)

#### 프로세스 ID

```

==35888== Memcheck, a memory error detector
==35888== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==35888== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==35888== Command: ./test
==35888==
==35888== Invalid write of size 4
==35888==    at 0x10916B: f (test.c:6)
==35888==    by 0x109180: main (test.c:10)
==35888== Address 0x4a59068 is 0 bytes after a block of size 40 alloc'd
==35888==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-
gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==35888==    by 0x10915E: f (test.c:5)
==35888==    by 0x109180: main (test.c:10)
==35888==
==35888==

```

**Memory error의 종류를 알려 줌  
(40바이트로 할당된 메모리 블록을 초과한 위치에 접근)**

**코드의 어느 부분에 문제점이 있는지 보여줌(stack trace)**

## 분석 메시지 해석

## 문제 2(Memory leak)

```
==35888== HEAP SUMMARY:
```

```
==35888==      in use at exit: 40 bytes in 1 blocks
```

프로그램이 종료될 때 40바이트가 사용 중

```
==35888==    total heap usage: 1 allocs, 0 frees, 40 bytes allocated
```

```
==35888==
```

```
==35888== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
```

```
==35888==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-  
gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
```

```
==35888==    by 0x10915E: f (test.c:5)
```

```
==35888==    by 0x109180: main (test.c:10)
```

```
==35888==
```

Memory leak이 일어난 위치를 보여 줌

```
==35888== LEAK SUMMARY:
```

```
==35888==    definitely lost: 40 bytes in 1 blocks
```

```
==35888==    indirectly lost: 0 bytes in 0 blocks
```

```
==35888==    possibly lost: 0 bytes in 0 blocks
```

```
==35888==    still reachable: 0 bytes in 0 blocks
```

```
==35888==    suppressed: 0 bytes in 0 blocks
```

```
==35888==
```

Memory leak을 항목별로 요약해서 보여 줌

```
==35888== For lists of detected and suppressed errors, rerun with: -s
```

```
==35888== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

## 분석 메시지 해석

- 다음과 같이 코드 수정 후, 컴파일하여 Valgrind로 분석해보기

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

void f(void){
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;
}

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



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

void f(void){
    int* x = malloc(10 * sizeof(int));
    free(x);
}

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

- 분석 메시지에서 오류가 사라진 것을 확인할 수 있음.

==36867== HEAP SUMMARY:

==36867== in use at exit: 0 bytes in 0 blocks

==36867== total heap usage: 1 allocs, 1 frees, 40 bytes allocated

==36867==

==36867== All heap blocks were freed -- no leaks are possible

==36867==

==36867== For lists of detected and suppressed errors, rerun with: -s

==36867== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

## 코드 피드백

## 김도협 님의 tree 문제 풀이 코드 분석 결과

## ■ 문제 1(Memory error)

총 노드의 개수 입력:

5

1번 노드의 자식 노드를 입력(-1일 경우 자식 없음): 2 3

==42967== Conditional jump or move depends on uninitialised value(s)

==42967== at 0x1096D1: main (ex01.c:145)

==42967==

==42967== Conditional jump or move depends on uninitialised value(s)

==42967== at 0x1096EB: main (ex01.c:145)

==42967==

2번 노드의 자식 노드를 입력(-1일 경우 자식 없음): -1 -1

3번 노드의 자식 노드를 입력(-1일 경우 자식 없음): 4 5

4번 노드의 자식 노드를 입력(-1일 경우 자식 없음): -1 -1

5번 노드의 자식 노드를 입력(-1일 경우 자식 없음): -1 -1

==42967== Conditional jump or move depends on uninitialised value(s)

==42967== at 0x109751: main (ex01.c:157)

==42967==

==42967== Conditional jump or move depends on uninitialised value(s)

==42967== at 0x109764: main (ex01.c:157)

==42967==

구슬을 몇 번 떨어뜨릴건가요?: 5

2

Memory error의 종류를 알려 줌  
(초기화되지 않은 값에 의해 분기함)

코드의 어느 부분에 문제점이 있는지 보여줌(stack trace)

## 코드 피드백

### 김도협 님의 tree 문제 풀이 코드 분석 결과

#### ■ 문제 2(Memory leak)

```

==42967== HEAP SUMMARY:
==42967==    in use at exit: 20 bytes in 1 blocks → 프로그램이 종료될 때 20바이트가 사용 중
==42967== total heap usage: 9 allocs, 8 frees, 2,228 bytes allocated
==42967==
==42967== 20 bytes in 1 blocks are definitely lost in loss record 1 of 1
==42967==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-
gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==42967==    by 0x10938C: main (ex01.c:79)
==42967==
==42967== LEAK SUMMARY:
==42967==    definitely lost: 20 bytes in 1 blocks
==42967==    indirectly lost: 0 bytes in 0 blocks
==42967==    possibly lost:  ==43103== Uninitialised value was created by a heap allocation
==42967==    still reachable: ==43103==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-
==42967==    suppressed:  ==43103== gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==42967==    ==43103== by 0x10938C: main (ex01.c:79)
==42967==
==42967== Use --track-origins=yes to see where uninitialised values come from
==42967== For lists of detected and suppressed errors, rerun with: -s
==42967== ERROR SUMMARY: 7 errors from 5 contexts (suppressed: 0 from 0)

```

--track-origins=yes 옵션을 사용하여 분석하면 코드의 어느 부분에서 초기화가 이루어지지 않았는지 알려 줌 (문제 1)

## 코드 피드백

### 분석 결과를 바탕으로 코드 수정

#### ■ 문제 1(Memory error)

- Line 2

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

- Line 79

```
// 노드의 번호를 중복 체크하기 위한 배열
int *recorded_nodes = (int *)malloc(sizeof(int) * total_Node);
memset(recorded_nodes, 0, sizeof(int) * total_Node);
```

- Line 156

```
// 구슬을 얼마나 떨어뜨릴 것인지 저장하는 변수
unsigned long long k;
if(k < 1 || k > 1000000000000000000ULL){
    return -1;
}
printf("구슬을 몇 번 떨어뜨릴건가요?: ");
scanf("%llu", &k);
```

표시한 부분을 scanf 뒤로 이동

#### ■ 문제 2(Memory leak)

- Line 199

```
// 메모리를 해제
SBT_DestroyTree(nodes[0]);
free(nodes);
free(recorded_nodes);
```



## 코드 피드백

### 분석 결과를 바탕으로 코드 수정

- 앞에서 언급된 문제가 해결되었음
- **주의사항:** Valgrind는 실행 중 분석을 진행하는 동적 분석 도구이기 때문에, 사용자 입력이 달라지면 이전에는 테스트되지 않았던 코드 영역이 실행되어 추가적인 문제가 나타날 수 있음.

총 노드의 개수 입력:

5

1번 노드의 자식 노드를 입력(-1일 경우 자식 없음): 2 3

2번 노드의 자식 노드를 입력(-1일 경우 자식 없음): -1 -1

3번 노드의 자식 노드를 입력(-1일 경우 자식 없음): 4 5

4번 노드의 자식 노드를 입력(-1일 경우 자식 없음): -1 -1

5번 노드의 자식 노드를 입력(-1일 경우 자식 없음): -1 -1

구슬을 몇 번 떨어뜨릴건가요?: 5

2

```
==43733== HEAP SUMMARY:
```

```
==43733==      in use at exit: 0 bytes in 0 blocks
```

```
==43733==    total heap usage: 9 allocs, 9 frees, 2,228 bytes allocated
```

```
==43733==
```

```
==43733== All heap blocks were freed -- no leaks are possible
```

```
==43733==
```

```
==43733== For lists of detected and suppressed errors, rerun with: -s
```

```
==43733== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

끝

