17

막간: 메모리 관리 API

이번 장에서는 UNIX의 메모리 관리 인터페이스에 대해 논의한다. 제공되는 인터페이스들은 비교적 간단하기 때문에 이 장은 짧고 핵심적인 내용만 다룬다[1]. 여기서의 메모리라 함은 사용자 주소 공간을 의미한다. 우리가 해결하고자 하는 문제는 다음과 같다:

핵심 질문: 어떻게 메모리를 할당하고 관리해야 하는가
UNIX/C 프로그램에서 메모리를 할당하고 관리하는 방법을 이해하는 것은 강력하고 안정적인 소프트웨어를 구축하는 데 중요하다. 일반적으로 어떤 인터페이스가 사용되는가? 어떤 실수를 해서는 안 되는가?

17.1 메모리 공간의 종류

C 프로그램이 실행되면, 두 가지 유형의 메모리 공간이 할당된다. 첫 번째는 스택(stack) 메모리라고 불리며 할당과 반환은 프로그래머를 위해 컴파일러에 의해 *암묵적으로* 이루어진다. 이러한 이유 때문에 때로는 **자동**(**automatic**) 메모리라고 불린다.

C 프로그램에서 스택에 메모리를 선언하는 것은 쉽다. 예를 들어, `func()`라는 함수 안에서 `x`라 불리는 정수를 위한 공간이 필요하다고 하자. 이러한 메모리를 선언하려면 다음과 같이 하면 된다.

```
void func() {
     int x; // 스택에 int 형을 선언
     ...
}
```

컴파일러가 나머지 작업을 수행하여, `func()`가 호출될 때 스택에 공간을 확보한다. 함수에서 리턴하면 컴파일러는 프로그래머 대신에 메모리를 반환한다. 함수 리턴 이후에도 유지되어야 하는 정보는 스택에 저장하지 않는 것이 좋다.

오랫동안 값이 유지되어야 하는 변수를 위해 힙(`heap`) 메모리라고 불리는 두 번째 유형의 메모리가 필요하다. 모든 할당과 반환이 프로그래머에 의해 명시적으로 처리된다.

1) 사실 우리는 모든 장이 그러길 바란다. 그러나 우리 생각엔 이 장이 유난히 짧고 더 핵심적이다.

당연히 무거운 책임! 그리고 많은 버그의 원인. 그러나 조심하고 주의를 기울이면 큰 문제 없이 올바르게 인터페이스를 사용할 수 있다. 다음 코드는 정수에 대한 포인터를 힙에 할당하는 예를 보여 준다.

```
void func() {
     int *x = (int *) malloc(sizeof(int));
     ...
}
```

이 코드에 대한 주의 사항이 몇 개 있다. 첫째, 한 행에 스택과 힙 할당이 모두 발생한다. 우선 컴파일러가 포인터 변수의 선언(`int *x`)을 만나면 정수 포인터를 위한 공간을 할당해야 한다는 것을 안다. 프로그램이 `malloc()`을 호출하여 정수를 위한 공간을 힙으로부터 요구한다. `malloc()`은 그 정수의 주소를 반환한다(성공한 경우, 실패한 경우에는 `NULL`을 반환). 이 반환된 주소는 스택에 저장되어 프로그램에 의해 사용된다.

이런 명시적 성질과 다양한 쓰임새 때문에 힙 메모리의 사용은 사용자와 시스템 모두에게 어려운 숙제다. 남은 논의는 힙 메모리에 초점을 맞출 것이다.

17.2 malloc() 함수

`malloc()` 호출은 매우 간단하다. 힙에 요청할 공간의 크기를 넘겨 주면, 성공했을 경우 새로 할당된 공간에 대한 포인터를 사용자에게 반환하고 실패했을 경우 `NULL`을 반환한다[2].

매뉴얼 페이지는 `malloc`을 사용하는 데 필요한 일을 보여 준다. 명령어 라인에 `man malloc`이라고 입력하면 다음과 같은 내용을 볼 것이다.

```
#include <stdlib.h>
...
void *malloc(size_t size);
```

이 정보로부터 `malloc`을 사용하기 위해서 할 일은 헤더 파일 `stdlib.h`를 소스 코드에 포함시키는 것 뿐이라는 것을 알 수 있다. 사실 이 일도 할 필요 없다. 모든 C 프로그램에 디폴트로 링크되는 C 라이브러리는 `malloc()` 함수를 가지고 있다. 헤더 파일을 추가하면 `malloc()`을 올바르게 호출했는지 컴파일러가 검사할 수 있다. 전달된 인자의 개수가 맞는지 올바른 데이터 타입의 인자를 전달했는지 검사하게 된다.

`malloc()`의 인자는 `size_t` 타입의 변수이고 이 변수는 필요 공간의 크기를 바이트 단위로 표시한 것이다. 대부분의 프로그래머는 숫자를 직접 입력하지 않는다(10 등). 사실 그런 식으로 입력하는 것은 좋지 않다. 대신 다양한 루틴과 매크로가 활용된다. 예를 들어, `double precision`의 부동 소수점 값을 위한 공간을 확보하기 위해서 다음과 같이 하면 된다.

```
double *d = (double *) malloc(sizeof(double));
```

2) C 언어에서 `NULL`은 특별한 것이 전혀 아니다. 단순히 0 값을 나타내는 매크로일 뿐이다.

팁: 의심스러우면 한 번 써 본다

당신이 사용하고 있는 루틴 또는 연산자가 어떻게 동작하는지 확신할 수 없으면, 실행시켜 보고 예상한 대로 동작하는지 확인하는 방법 이외의 대안은 없다. 매뉴얼 페이지나 기타 문서를 읽는 것이 유용하긴 하지만, 실제 동작을 확인하는 것이 중요하다. 코드를 작성해서 실험해 보라. 당연히 이 방법이 작성한 코드가 예상대로 동작하는지 확인하는 가장 최선의 방법이다. 사실 우리도 `sizeof()`에 대해 우리가 말했던 내용이 사실이라는 것을 재차 확인하기 위하여 실제로 실행시켜 보았다.

와, `double`이 엄청 많네! 이 `malloc()` 호출에서는 정확한 크기의 공간을 요청하기 위하여 `sizeof()` 연산자를 사용한다. C 언어에서 이 `sizeof()`는 통상 컴파일 시간 연산자이다. 인자의 실제 크기가 컴파일 시간에 결정된다. `sizeof()`는 숫자(이 경우 `double`의 크기인 8)로 대체되어 `malloc()`에 전달된다. 이러한 이유로, `sizeof()`는 연산자로 간주되는 게 맞으며 함수 호출이 아니다(함수 호출은 실행 시간에 일어난다).

데이터 타입뿐 아니라 변수의 이름도 `sizeof()`의 인자로 전달할 수 있다. 그러나 원하는 결과를 얻지 못할 때도 있으므로 조심해야 한다. 예를 들어, 다음과 같은 코드를 살펴보자.

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));
```

첫 번째 행에서, 정수형 원소 10개를 가지는 배열을 위한 공간을 선언하였다. 멋진 코드. 그러나 다음 줄에서 `sizeof()`를 사용하면 4(32비트 컴퓨터에서) 또는 8(64비트 컴퓨터에서) 값을 반환한다. 이 경우 `sizeof()`는 동적으로 할당받은 메모리의 크기가 얼마인지가 아니라, 정수를 가리키는 포인터의 크기가 얼마인지 물어본다고 생각하기 때문이다. 그러나 때때로 `sizeof()`는 기대한 대로 동작할 때도 있다:

```
int x[10];
printf("%d\n", sizeof(x));
```

이 경우에는 변수 x에 40바이트가 할당되었다는 것을 컴파일러가 알 수 있는 정적인 정보가 충분하다.

또 하나 조심해야 하는 경우는 문자열을 다룰 때이다. 문자열을 위한 공간을 선언할 때에는 다음과 같은 문장을 사용한다: `malloc(strlen(s) + 1)`. 이 문장은 `strlen()` 함수를 사용하여 문자열의 길이를 얻어낸 뒤 문자열-끝을 나타내는 문자를 위한 공간을 확보하기 위해 1바이트를 더한다. `sizeof()`의 사용은 여기서 문제를 일으킬 수 있다.

`malloc()`은 `void` 타입에 대한 포인터를 반환한다는 것을 알았을 것이다. 그렇게 하는 것은 주소만 넘겨주고 해당 주소 공간에 어떤 타입의 자료를 저장할 지는 프로그래머가 결정하게 하는 전형적인 C의 방식이다. 프로그래머는 **타입 변환**(**type casting**)을 이용하여 공간 활용을 결정한다. 위의 예에서 프로그래머는 `malloc()`이 반환한 데이터 타입을 `double` 형을 가리키는 포인터 타입으로 변환하였다. 캐스팅은 실제

무언가를 하지는 않고, 컴파일러와 다른 프로그래머에게 "예, 제가 알아서 할게요"라고 말할 뿐이다. `malloc()`의 결과에 타입을 명시함으로써, 프로그래머는 정확한 타입을 "확인"한 정도이며, 프로그램이 제대로 동작하는 데 있어서 캐스트가 반드시 필요한 것은 아니다.

17.3 `free()` 함수

"메모리 할당"은 우리가 고민하고 있는 문제들 중 쉬운 쪽이다. 할당된 메모리를 언제, 어떻게 해제하고 더욱이 해제 여부를 확인하는 것이 더 어려운 문제이다. 더 이상 사용되지 않는 힙 메모리를 해제하기 위해 프로그래머는 `free()`를 호출한다.

```
int *x = malloc(10 * sizeof(int));
...
free(x);
```

한 개의 인자, `malloc()`에 의해 반환된 포인터를 받는다. 할당된 영역의 크기는 전달되지 않는다. 할당된 메모리의 크기는 메모리 할당 라이브러리가 알고 있어야 한다.

17.4 흔한 오류

`malloc()`과 `free()`를 사용하는 데 흔히 발생하는 오류가 많다. 학부 운영체제를 가르치면서 반복적으로 보아 왔던 오류 중 일부를 소개하겠다. 모든 예들은 컴파일러 최적화 없이 컴파일되고 실행되었다. C 프로그램을 컴파일하는 것은 제대로 돌아가는 C 프로그램을 제작하는 데 반드시 필요하지만, 그것만으로는 충분하지 않다는 것을 배우게 될 것이다.

올바른 메모리 관리가 그렇다. 많은 새로운 언어들이 **자동 메모리 관리**(**automatic memory management**)를 지원한다. 그러한 언어들에서는 메모리를 할당하기 위하여 `malloc()`과 유사한, 보통 **new** 또는 새 객체를 할당하기 위한 비슷한 루틴을 호출하는 반면에, 공간을 해제하기 위해서는 아무것도 호출하지 않는다. 쓰레기 수집기(garbage collector)가 실행되어 참조되지 않는 메모리가 찾아 알아서 해제한다.

메모리 할당 잊어버리기

많은 루틴은 자신이 호출되기 전에 필요한 메모리가 미리 할당되었다고 가정한다. 예를 들어, `strcpy(dst, src)` 루틴은 소스 포인터에서 목적 포인터로 문자열을 복사한다. 그러나 주의하지 않으면, 다음과 같이 코드를 작성할 수 있다.

```
char *src = "hello";
char *dst;        // 아뿔싸! 할당이 안 되어 있네
strcpy(dst, src); // segfault 그리고 죽는다
```

팁: 컴파일되었다 또는 실행되었다 ≠ 원하는 결과다

단지 프로그램이 컴파일(!) 또는 심지어 한 번 또는 여러 번 정확하게 실행되었다고 하더라도 프로그램이 정확하다는 것을 의미하지는 않는다. 많은 사건이 어우러져 프로그램이 잘 동작한다고 믿게끔 만들 수 있지만, 뭔가가 바뀌면 동작을 중지하게 된다. 대부분 학생들의 반응은 "그러나 전에는 동작했단 말이에요"라고 말하거나 소리지르고, 컴파일러, 운영체제, 하드웨어, (감히 말하건대) 심지어는 교수를 비난한다. 그러나 문제는 보통 거기 있을 것이라고 생각하는 바로 그 곳, 당신의 코드 안에 있다. 다른 요소를 비난하기 전에 자리로 돌아가서 디버깅해라.

이 코드를 실행하면 **세그멘테이션 폴트(segmentation fault)**[3]가 발생할 가능성이 높다. 이 폴트는 "**네가 메모리 관련 무언가를 잘못했어, 이 바보 같은 프로그래머야, 그래서 나 화났거든**"이라는 메시지이다.

이 경우, 올바른 코드는 다음과 같을 수 있다:

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src) + 1);
strcpy(dst, src); // 제대로 동작
```

대안으로, `strdup()`을 사용하여 훨씬 편하게 할 수 있다. 더 많은 정보를 원한다면 `strdup`의 `man` 페이지를 읽으시오.

메모리를 부족하게 할당받기

이 오류는 메모리를 부족하게 할당받는 것으로, 때때로 **버퍼 오버플로우(buffer overflow)**라고 불린다. 앞의 예에서, 흔한 오류는 목적지 버퍼 공간을 약간 부족하게 할당받는 것이다.

```
char *src = "hello"
char *dst = (char *) malloc(strlen(src)); // 너무 작다!
strcpy(dst, src); // 제대로 동작
```

이상하지만 `malloc`이 구현 방식과 기타 많은 세부 사항에 따라서는 이 프로그램이 제대로 동작하는 것처럼 보이는 경우가 종종 있다. 어떤 경우에는 문자열 복사가 실행될 때 할당된 공간의 마지막을 지나쳐 한 바이트 만큼 더 공간을 사용한다. 이 공간이 더 이상 사용되지 않는 변수 영역이기 때문에 덮어쓰더라도 아무 피해가 발생하지 않을 때도 있다. 다른 때에는 이러한 오버플로우가 매우 유해할 수 있고 사실 많은 시스템에서 보안 취약점의 원인이다 [Wer06]. 어떤 경우에는 `malloc` 함수 라이브러리가 여분의 공간을 할당하고 프로그램은 다른 변수의 값을 덮어쓰지 않고 잘 동작한다. 또 다른 경우에는 프로그램은 고장을 일으키고 크래시된다. 다음과 같은 또다른 중요한 교훈을 얻는다: 프로그램이 한 번 올바르게 실행된다고 하더라도, 프로그램이 올바르다는 것을

3) 용어가 신비롭게 들리겠지만, 이러한 불법적인 메모리 접근이 왜 세그멘테이션 폴트라고 불리는지 곧 배우게 될 것이다. 이게 계속 책을 읽을만한 인센티브가 아니라면, 뭐를 하면 계속 책을 읽을래?

의미하지는 않는다.

할당받은 메모리 초기화하지 않기

이 오류의 경우, `malloc()`을 제대로 호출했지만 새로 할당받은 데이터 타입에 특정 값을 넣는 것을 잊는 것이다. 절대로 이러지 마라. 초기화하지 않는다면, 프로그램은 결국 **초기화되지 않은 읽기**(**uninitialized read**), 즉 힙으로부터 알 수 없는 값을 읽는 일이 생긴다. 그 영역에 어떤 값이 들어 있을지 누가 알겠는가? 재수가 좋으면 프로그램이 여전히 잘 동작할 수 있는 값일테고, 그게 아니라면 임의의 값이나 해로운 값이 읽힐 것이다.

메모리 해제하지 않기

다른 일반적인 오류는 **메모리 누수**(**memory leak**)다. 메모리 해제를 잊었을 때 발생한다. 장시간 실행되는 응용 프로그램이나 또는 운영체제 자체와 같은 시스템 프로그램에서는 큰 문제이다. 메모리가 천천히 누수되면 결국 메모리가 부족하게 되고 시스템을 재시작 할 수밖에 없기 때문이다. 메모리 청크의 사용이 끝나면 반드시 해제해야 한다. 쓰레기 수집 기능이 있는 언어도 이 문제에는 도움이 되지 않는다. 메모리 청크에 대한 참조가 존재하면, 어느 쓰레기 수집기도 그 청크를 해제하지 않을 것이고, 따라서 이런 현대적인 언어에서도 메모리 누수는 여전히 문제가 된다.

경우에 따라서 `free()`를 호출하지 않는 것이 타당하게 보이는 경우도 있다. 예를 들어, 프로그램 실행 시간이 짧아, 실행 시작 후 곧 종료한다고 하자. 이 경우 프로세스가 죽으면, 운영체제는 할당된 페이지를 모두 정리하고 따라서 메모리 누수는 일어나지 않는다. 이 가정이 맞을 수도 있지만, 들여서는 안 되는 나쁜 버릇으로서, 이와 같은 전략은 매우 조심해서 사용해야 한다 (다음 여담을 참조하시오). 장기적으로 프로그래머로서 당신의 목표는 좋은 습관을 만드는 것이다. 중요한 습관 중 하나가 C 같은 언어에서 어떻게 메모리를 관리하는지 이해하고 할당받았던 메모리 블럭을 해제하는 것이다. 메모리 해제 없이도 그럭저럭 개발자로 지낼 수 있겠지만 한 바이트라도 명시적으로 할당받았으면 해제하는 습관을 들이도록 하자.

메모리 사용이 끝나기 전에 메모리 해제하기

때때로 프로그램은 메모리 사용이 끝나기 전에 메모리를 해제한다. 그런 실수는 **dangling pointer**라고 불리며 심각한 실수이다. 차후 그 포인터를 사용하면 프로그램을 크래시시키거나 유효 메모리 영역을 덮어쓸 수 있다. 예를 들어, `free()`를 호출하고, 그 후 다른 용도로 `malloc()`을 호출하면 잘못 해제된 메모리를 재사용한다.

반복적으로 메모리 해제하기

프로그램은 가끔씩 메모리를 한 번 이상 해제하며 **이중 해제**(**double free**)라 불린다. 결과는 예측하기 어렵다. 상상할 수 있듯이 메모리 할당 라이브러리는 어찌할 바 모르게

여담: 프로세스가 종료하면 왜 메모리 누수가 일어나지 않는가

실행 시간이 짧은 프로그램을 작성할 때 `malloc()`을 이용하여 일부 공간을 할당할 수 있다. 프로그램은 실행 후 곧 종료할 것이다. 종료 직전에 `free()`를 호출할 필요가 있을까? 호출하지 않으면 잘못한 것처럼 보이지만 실제로는 손해보지 않는다. 이유는 간단하다. 시스템이 실제 두 단계로 메모리를 관리하기 때문이다. 메모리 관리의 첫 번째 단계는 운영체제에 의해 수행된다. 프로세스가 실행할 때 메모리를 프로세스에게 건네 주고 프로세스가 종료하거나 다른 이유로 죽을 때 메모리를 되돌려 받는다. 두 번째 단계는 각 프로세스 내에서, 예를 들면, `malloc()`과 `free()`를 호출할 때 힙 내에서 수행된다. `free()`를 호출하지 못했더라도 그래서 메모리가 누수되었더라도 프로세스가 종료할 때, 운영체제는 프로세스의 모든 메모리를 회수한다. 프로세스의 코드, 스택, 및 여기서 논의 중인 힙 등 모든 페이지가 포함된다. 프로세스 주소 공간의 힙이 어떤 상태가 되더라도, 운영체제는 프로세스가 죽을 때 그 모든 페이지를 회수한다. 당신이 해제하지 않았더라도 메모리가 누수되지 않는다.

이처럼 실행 시간이 짧은 프로그램의 경우, 나쁜 형태의 코드라고 생각되지만 메모리 누수는 어떠한 동작적인 문제도 일으키지 않는 경우가 대부분이다. Web 서버나 데이터베이스 관리 시스템 등의 결코 종료하지 않는 장시간 실행되는 서버(웹서버나 데이터베이스 관리 시스템과 같이 절대 종료하지 않는 것)의 코드를 작성할 때, 누수 메모리는 훨씬 심각한 문제이고 응용 프로그램이 메모리를 다 소진하게 되면 결국 크래시가 발생한다. 물론, 메모리 누수는 한 가지 특정 프로그램에서 더 큰 문제가 된다: 운영체제 자신. 이러한 사실은 커널 코드를 작성하는 사람들이 가장 힘들고 어려운 일을 한다는 걸 우리에게 다시 한 번 상기시킨다.

되고, 모든 종류의 이상한 일을 하게 된다. 가장 흔히 일어나는 결과는 크래시다.

`free()` 잘못 호출하기

마지막 문제점은 `free()`를 잘못 호출하는 것이다. `free()`는 `malloc()` 받은 포인터만 전달될 것으로 예상한다. 그 이외의 값을 전달하면 문제가 발생한다. **유효하지 않은 해제(invalid frees)**는 매우 위험하고 당연히 피해야 한다.

요약

메모리를 오용하는 방법은 많다. 메모리 관련 오류가 자주 발생하기 때문에, 코드에서 그런 오류를 찾아내는 도구의 생태계가 생겼다. **purify** [HJ92]와 **valgrind** [SN05]를 검토해 보라. 메모리 관련 문제의 원인을 찾아내는 훌륭한 도구들이다. 이 강력한 도구의 사용에 익숙해지면 그동안 그들없이 어떻게 살았나 하는 의문이 들 것이다.

17.5 운영체제의 지원

`malloc()`과 `free()`를 논의해 오면서 시스템 콜에 관해서는 한 번도 언급하지 않았다. 이유는 간단하다: 그들은 시스템 콜이 아니라 라이브러리 함수이기 때문이다! `malloc` 라이브러리가 프로세스 가상 주소 공간 안의 공간을 관리하지만 라이브러리 자체는 시스템에게 더 많은 메모리를 요구하고 반환하는 시스템 콜을 기반으로 구축된다.

그런 시스템 콜 중 하나가 `brk`라고 불리는 시스템 콜로서, 프로그램의 **break** 위치를 변경하는 데 사용된다. `break`는 힙의 마지막 위치를 나타낸다. `brk`는 새로운 `break` 주소를 나타내는 한 개의 인자를 받는다. 새로운 `break`가 현재 `break`보다 큰지 작은지에 따라 힙의 크기를 증가시키거나 감소시킨다. `sbrk`는 증가량만을 받아들이는 것을 제외하고 비슷한 용도로 사용된다.

`brk` 또는 `sbrk`를 직접 호출해서는 안 된다는 것에 주의하라. 이들은 메모리 할당 라이브러리에 의해 사용된다. 직접 사용하면 감당할 수 없는 결과가 발생할 것이다. 반드시 `malloc()`과 `free()`를 사용하라.

마지막으로, `mmap()` 함수를 사용하여 운영체제로부터 메모리를 얻을 수도 있다. 올바른 인자를 전달하면 `mmap()`은 프로그램에 **anonymous**의 메모리 영역을 만든다. `anonymous` 영역은 특정 파일과 연결되어 있지 않고 **스왑 공간**(**swap space**)에 연결된 영역을 말한다. 이 영역에 대해서는 나중에 가상 메모리에서 논의한다. 이 메모리는 힙처럼 취급되고 관리된다. 자세한 내용은 `mmap()`의 매뉴얼 페이지를 읽어라.

17.6 기타 함수들

메모리 할당과 관련하여 다른 함수들이 몇 개 더 있다. 예를 들어, `calloc()`은 메모리를 할당 영역을 0으로 채워서 반환한다. 이 함수는 초기화하는 것을 잊어버리는 오류를 방지한다(앞에서 설명한 "초기화되지 않은 읽기"에 관한 문단을 참조하라). `realloc()`은 이미 할당된 공간에 대해(예를 들어, 배열) 추가의 공간이 필요할 때 유용하다. `realloc()`은 더 큰 새로운 영역을 확보하고 옛 영역의 내용을 복사한 후에 새 영역에 대한 포인터를 반환한다.

17.7 요약

우리는 메모리 할당을 다루는 몇 가지 API를 소개하였다. 기본적인 내용만을 다루었다. 자세한 내용은 다른 곳에서 찾을 수 있다. 더 많은 정보를 원한다면 C 책 [KR88]과 Stevens [SR05] (7장)를 읽어라. 많은 오류들을 자동적으로 감지하고 수정하는 방법에 대한 Novark의 논문을 읽기 바란다 [NBZ07]. 이 논문에는 자주 발생하는 문제들과 그것들을 발견하고 수정하는 방법에 잘 요약해 놓았다.

참고 문헌

[HJ92] **"Purify: Fast Detection of Memory Leaks and Access Errors"**
R. Hastings and B. Joyce
USENIX Winter '92
멋진 Purify 도구에 관한 논문. 현재는 상업용 제품이다.

[KR88] **"The C Programming Language"**
Brian Kernighan and Dennis Ritchie
Prentice-Hall 1988
C 언어 개발자의 C 책. 한 번 읽은 후에 프로그래밍을 해 보라. 그런 후 다시 읽어라. 프로그램 할 때마다 책상 가까운 곳에 두어라.

[NBZ07] **"Exterminator: Automatically Correcting Memory Errors with High Probability"**
Gene Novark, Emery D. Berger, and Benjamin G. Zorn
PLDI 2007
메모리 오류를 자동적으로 찾아 수정하는 것에 관한 멋진 논문. 또한 C와 C++ 프로그램의 일반적인 오류에 관한 개괄도 제공한다.

[SN05] **"Using Valgrind to Detect Undefined Value Errors with Bit-precision"**
J. Seward and N. Nethercote
USENIX '05
특정 유형의 오류를 발견하는 것이 목적인 경우의 valgrind 사용법.

[SR05] **"Advanced Programming in the Unix Environment"**
W. Richard Stevens and Stephen A. Rago
Addison-Wesley, 2005
예전에도 이야기 했지만 다시 말한다. 이 책을 여러 번 읽어라 그리고 의문이 생길 때마다 참고 문헌으로 사용하라. C 프로그래밍 경험이 많음에도 불구하고 저자들은 책을 읽을 때마다 어떻게 새로운 것을 배울 수 있는지 항상 놀란다.

[Wer06] **"Survey on Buffer Overflow Attacks and Countermeasures"**
Tim Werthman
URL: `www.nds.rub.de/lehre/seminar/SS06/Werthmann_BufferOverflow.pdf`
버퍼 오버플로우와 그들이 일으키는 보안 문제에 관한 훌륭한 요약서. 유명한 취약점 공격을 보려면 참조하시오.

숙제 (코드)

이 과제에서 메모리 할당에 대한 어느 정도의 지식을 얻을 수 있다. 먼저 몇몇 버그 있는 프로그램을 작성한다(재미있겠지!). 다음, 당신이 삽입한 버그를 발견하는 데 도움을 주는 몇 가지 도구를 사용한다. 그러면 이러한 도구가 얼마나 대단한지 알게 될 것이고 미래에 그들을 사용하게 되어 행복하고 생산적인 프로그래머가 될 것이다.

사용할 첫 번째 도구는 `gdb` 디버거이다. 이 디버거에 대해 배울 것이 많다. 여기서는 단지 수박 겉핥기식으로 넘어갈 것이다.

두 번째 사용할 도구는 `valgrind` [SN05]이다. 이 도구는 프로그램의 메모리 누수와 다른 교묘한 메모리 문제를 발견하는 데 도움을 준다. 만약 설치되어 있지 않다면 다음 web 사이트에 가서 설치하라.

`http://valgrind.org/downloads/current.html`

문제

1. `null.c`라는 간단한 프로그램을 작성하라. 이 프로그램은 정수를 가리키는 포인터를 만들고 NULL로 초기화하고 그것을 역참조하려고 한다. `null`이라는 실행 파일이 생성되도록 컴파일하라. 프로그램을 실행시키면 어떤 일이 발생하는가?
2. 다음에 이 프로그램을 심볼 정보가 포함되도록 컴파일하라(`-g` 플래그 사용). 실행 파일에 더 많은 정보를 추가하여 디버거가 변수 이름 등의 유용한 정보를 접근할 수 있게 한다. 디버거에서 프로그램을 실행하라(`gdb null`). `gdb`가 실행되면, `run` 명령어를 입력하라. `gdb`는 무엇을 보여주는가?
3. 마지막으로, 이 프로그램에 `valgrind` 도구를 사용하라. 우리는 `valgrind`의 일부인 `memcheck` 도구를 사용하여 어떤 일이 일어나는지 분석할 것이다. 다음과 같은 명령어를 실행시켜라: `valgrind --leak-check=yes null`. 이를 실행하면 어떻게 되는가? 출력을 해석할 수 있는가?
4. `malloc()`을 사용하여 메모리를 할당하지만 종료 전에 해제하는 것을 잊어버리는 간단한 프로그램을 작성하라. 이 프로그램을 실행시키면 무슨 일이 일어나는가? 이 프로그램의 문제를 gdb를 사용하여 발견할 수 있는가? `valgrind`는 어떤가(다시 `--leak-check=yes` 플래그를 주고 실행)?
5. 크기가 100인 `data`라는 이름의 정수 배열을 만드는 프로그램을 작성하라. 그런 후 `data[100]`을 0으로 클리어 하라. 이 프로그램을 실행하면 어떤 일이 벌어지는가? `valgrind`를 사용하여 프로그램을 실행하면 어떤 일이 일어나는가? 프로그램은 정확한가?
6. 정수 배열을 할당하고(위와 동일), 다시 해제하고, 배열의 원소 하나의 값을 출력하려고 시도하는 프로그램을 작성하라. 프로그램이 실행되는가? `Valgrind`를 사용하여 실행하면 어떤 일이 발생하는가?

7. `free`에 위에서 할당한 배열의 중간을 가리키는 포인터와 같은 황당한 값을 전달하라. 어떤 일이 벌어지는가? 이런 종류의 문제를 발견하기 위해 도구가 필요한가?

8. 메모리를 할당할 때 다른 인터페이스를 사용해 보라. 예를 들어, 간단한 벡터-비슷한 자료 구조를 만들고 벡터를 관리하기 위하여 `realloc()`을 사용하는 루틴을 만들어라. 벡터 원소를 저장하기 위하여 배열을 사용하라. 사용자가 벡터에 항목을 추가할 때, `realloc()`을 사용하여 필요 공간을 확보하라. 이 벡터 프로그램은 얼마나 잘 동작하는가? 연결 리스트와 비교하면 어떠한가? 버그를 발견하기 위하여 valgrind를 사용하라.

9. `gdb`와 `valgrind` 사용법을 배우기 위하여 시간과 노력을 투자하라. 도구를 잘 아는 것은 매우 중요하다. UNIX와 C 환경에서 디버거 전문가가 되기 위해 시간과 노력을 투자하라.