실제 응용프로그램들의 메모리 릭 측정

최진선° 이인환
한양대학교 전자통신컴퓨터공학과
jschoi@csl.hyu.ac.kr° ihlee@hanyang.ac.kr

Measuring memory leak in real applications

Jinsun Choi°, Inhwan Lee
Dept. of Electronic and Computer Engineering, Hanyang University

요 약

C/C++언어로 구현된 응용프로그램들은 언어적인 특성으로 메모리 릭에 취약하다고 알려져왔다. 이에 이러한 언어적인 약점을 보완하기위해 가비지 컬렉션 등과 같은 기술이 연구/발표되어왔다. 그러나 릭이 실제 응용프로그램 사이에서 얼마나 발생이 되고 있으며 얼마나 심각한지 발표된 자료는 찾을 수 없었다. 제안된 보완기술들 조차 실제 응용프로그램을 적용하여 테스트한 사례는 찾을 수 없었다. 따라서, 본 논문에서는 실제 응용 프로그램을 선정하여 메모리 릭의 발생 정도를 측정하고, 발생 원인은 무엇이고 C/C++ 언어간의 릭 발생 특징은 존재하는지 조사해 보고자 한다. 또한 Valgrind 툴을 이용하면서 발견된 문제점을 토론함으로써 향후 더 우수한 동적 메모리 디버깅 툴을 개발하는데 기여하고자 한다.

1. 서 론

C/C++에서 객체 또는 변수 사용을 위해 메모리 공간을 할당할 수 있다. 동적으로 할당된 객체의 사용이 끝나면 할당된 메모리는 객체를 통해 명시적으로 반환처리되어야 한다. 특히 객체에 대한 포인터들은 스코프를 벗어나기 전에 메모리 반환을 수행하여야 한다[1]. 그러한 처리가 미흡할 때 메모리 릭을 초래하게 된다. 서버와 같이 오래 실행되는 프로그램이나 막대한 메모리를 사용하는 프로그램들은 메모리 릭이 존재하면 성능이 심각하게 저하되거나 심지어는 시스템이 크래쉬 될 수도 있다.

메모리 릭은 자동화된 가비지 컬렉션을 갖고 있지 않은 C/C++ 같은 언어를 사용할 때 자주 발생한다고 알려져 있다. 이러한 메모리 릭이 어떤 경우에 훨씬 심각한 상황을 초래할 수 있는지 살펴보면, 첫째, 프로그램이 오랜 기간 계속 실행되어야하는 백그라운드 태스크, 서버 프로그램을 예로 들 수 있다. 둘째, 공유 메모리를 요청하여 사용하는 경우로, 심지어 프로그램이 종료되어도 자동으로 메모리를 반환시키지 않아 시스템이 살아있는 동안 계속 사용할 수 없는 메모리 영역이되어 시스템 전반적으로 성능을 떨어뜨리는 원인이 될 수 있다. 셋째, 운영체제 자체에서도 릭이 발생될 수 있다. 마지막으로 임베디드 시스템이나 휴대용 장비와 같은 메모리가 매우 제한된 경우에도 심각한 상황을 초래할 수 있다[2].

2. 관련 연구

자바는 개발자가 메모리를 관리해주지 않아도 된다[3]. 자바는 가상 머신이 사용되지 않는 메모리 영역을 자동으로 찾아 해제해주는 기술이 반영되어있기 때문이다. 이러한 자동화된 가비지 컬렉션 기술이 자바내에 기본적으로 반영되어있으나, C/C++은 효율성, 성능에 더 비중을 두어 설계된 언어이기 때문에, 메모리 할당 뿐만 아니라 해제까지도 개발자가 책임지도록 설계되어있다. 이러한 C/C++의 언어적 약점을 성능과 안정성면에서 보완하기 위해 여러 가비지 컬렉션 기법이 제안되어왔다. 그러나 C/C++로 개발된 어플리케이션이 메모리 릭을 자주 발생시킨다고 주장할만한 실제 측정 자료는 찾을 수 없었다. 제안된 자율적 가비지 컬렉션[4] 연구에서는 테스트용으로 개발된 testAppl을 실험한 결과만 제시하고 있으며, 연구개발된 iWatcher 디버깅 툴[5]은 gzip, cachelib, bc 등의 어플리케이션을 실험한 자료가 있지만, 실험 결과는 수량적 측정치가 아닌, 버그 탐지 여부만을 제시하였다. 그 밖의 여러 연구들도 실제 어플리케이션을 대상으로 테스트한 사례는 찾을 수 없었다.

본 논문에서는 이러한 이유로 실제 우리가 이용하고있는 C 또는 C++로 개발된 프로그램들 중 첫번째로 메모리 릭 현황을 조사하고, 두번째, 메모리 릭을 발생시키는 몇가지 경우를 조사하여 특징을 살펴보고, 마지막으로 실험에 이용된 Valgrind 툴을 통한 추적 과정에서의 문제점에 관해 토론하고자 한다.

3. 메모리 릭 측정 실험 디자인

메모리 릭을 측정하기위해서는 실험 환경을 구축하여야 한다. 이에 몇 가지 고려해야할 사항들이 있다.

3.1 플랫폼 선정 기준

실험이 메모리 릭만을 체크하는 것에서 그치지 않고, 원인을 추적하고 수정하여 결과를 확인할 수 있는 것이 바람

직하다. 그러한 면에서 상대적으로 오픈 소스가 많은 리눅스를 선정하게 되었다.

3.2 대상 어플리케이션 선정 기준

어플리케이션 선정 기준은 첫째, 장기간 실행되어 있어야 하는 어플리케이션을 고려하였다. 이러한 성격의 프로그램은 메모리 릭이 점차적으로 증가할 때 치명적일 수 있기 때문이다. 리눅스에 기본적으로 설치되어있는 데몬들을 그러한 이유에서 테스트해보았으나 메모리 릭을 거의 발견하지 못하였다. 그 중 파일 공유 기능을 제공하는 samba 서버에서 메모리 릭 1건을 발견하여 그 원인을 추적하고자 선정하게 되었다. 둘째, 초기 테스트 작업 중 데몬보다는 GUI 성격의 프로그램이 메모리 할당 빈도가 월등히 높다는 것을 알게 되었다. 이에 상대적으로 간단한 기능을 제공하는 GUI 어플리케이션을 선정하여 테스트하는 것이 흥미로울 것으로 생각되었다. 셋째, 사전 조사 중 읽게된 논문[3]에, OOP 언어로 개발된 프로그램들은 일반적으로 대량의 메모리 할당이 수행된다고 언급하고 있다. 그러한 언급을 바탕으로 메모리 할당 수행 빈도가 C보다 C++가 월등히 많은지, 그리고 메모리 릭 또한 C++에서 더 많이 발생하는지 궁금하였다. 그래서 C와 C++로 구현된 프로그램을 각각 나누어 선정하도록 하였다. 결과적으로 GUI가 없는 응용프로그램으로 samba 서버와 lshw를 선정하였고, GUI가 있는 프로그램으로는 xmms, xpdf, worker를 선정하였다. 또한, samba 서버와 xmms는 C로 구현된 프로그램이고, 나머지는 C++로 구현된 프로그램이다.[6]

3.3 측정 도구 선정

메모리 릭 체크를 위한 툴은 크게 정적 툴과 동적 툴로 나눌 수 있다. 본 논문에서 관심있는 분야는 실행중 메모리가 꾸준히 감소되는 상황을 추적하는 것이 목적이기 때문에 동적 툴을 조사하였고, 그 조사 내용은 표 1과 같다. 이러한 동적 툴 중 선택 기준은 오픈 소스로 구현된 소스이고 실험하려는 대상 프로그램에대해 재컴파일을 요구하지 않는 툴을 선택하고자 하였다. 그에 맞는 툴이 memprof와 Valgrind였고, 그 중 발표된 논문[5] 중 Valgrind 툴이 언급되어있어, Valgrind를 선택하였다.

표 1 동적 툴 리스트

툴 명	라이센스	오픈 소스	재컴파일 요구
FunctionCheck 프로파일러	Yannick Perret	Yes	Yes
ccmalloc	Armin Biere	Yes	Yes
memprof	Red Hat, Inc, Kristian Rietveld	Yes	No
valgrind	valgrind.org	Yes	No
purify	IBM Rational	No	No
boundschecker	Compuware	No	No
insure++	Parasoft	No	No

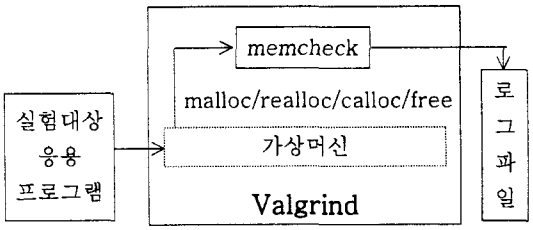

그림 1 메모리 릭 실험 동작 방식

Valgrind 툴은 패키지로 되어있어, 많은 기능을 지원하고 있다. 제공하는 기능 중 본 논문에서는 "memcheck"라는 툴을 통해 메모리 릭을 체크하게되었으며, 단순 malloc/free 횟 수만을 비교하는 것이 아니라, 반환되지 않은 블록 중에서 "definitely lost"라고 표시되어있는 결과에 대해서만 메모리 릭 추적을 수행하였다. 이 툴에서 "definitely lost"로 표시되는 경우는 메모리 반환 정보를 잃은 블록을 말하며 그 블록이 할당되기까지 호출 함수 경로를 알려준다. 이러한 부분은 메모리 릭의 징후로 간주되어 꼭 수정되도록 요구된다.[7]

3.4 메모리 릭 실험 동작

메모리 릭 실험에 있어 결정된 사항을 바탕으로 먼저 실험 동작 방식을 그림 2에 소개하였다. 실험은 Valgrind를 실행하면서 대상 실행 파일을 인자로 전달함으로써 시작된다. Valgrind는 가상 머신 방식을 이용하여 실험 대상 응용 프로그램에서 실행코드를 읽어, malloc/realloc/calloc/free 등과 같은 메모리 할당과 관련된 함수를 만나게 되면, memcheck 로 보내어 관리한다. 그 후 읽기/쓰기 명령을 수행할 때, memcheck에서 관리되는 정보를 비교하여 판단하는 방식으로 동작된다.

4. 메모리 릭 실험 결과 및 토론

실험한 결과는 표 1과 같다. 실험 결과에서 메모리 릭에 해당하는 내용만 요약하여 만든 표로, 두 번째 열은 응용에서 각각 메모리 할당 수행 횟 수를 나타내고, 세 번째 열은 명백한 메모리 릭 징후에 해당하는 횟 수를 나타내며, 마지막 열은 측정한 상황을 나타내었다.

표 2 Valgrind의 memcheck 툴을 수행한 결과

응용	할당 횟수	명백한 릭 횟수	측정 상황
samba	6,453	1	client 1대가 접속된 상태에서 측정함.
lshw	9,337	4	잘못된 부분을 수정한 후 측정함.
xpdf	202,366	9	Pdf file 1개를 open한 상태에서 측정함
xmms	25,159	6	Wave file 1개를 open한 상태에서 측정함
worker	16,191	3	

표 2에서 볼 수 있는 주목할만한 사항은 앞 절에서 설명한 바와 같이 GUI 프로그램에서 메모리 할당을 수행한 횟수가 GUI가 아닌 프로그램보다 월등히 많다는 것을 알 수 있다. 또한, 전체적으로 메모리 릭이 실제로는 매우 적게 발생한다는 사실도 알 수 있다.

이와같은 응용들을 측정한 후 명백한 메모리 릭이라고 조사된 로그를 추적하여 얻어진 결과를 다음절부터 구체적으로 설명해보고자 한다.

4.1 samba : 파일 공유 서버 데몬

장시간 실행되는 samba 서버에서 메모리 릭이 1회 발생되었으나, 추적결과 종료시에 string_set() 함수에서 호출된 string_init() 함수에서 할당된 메모리 블럭이 반환되지 않은 상태로 종료되어 나타난 현상으로 전혀 주목할만한 사항이 아니었다. 그러므로 samba 서버에서는 꾸준히 증가하는 메모리 릭이 존재하지 않았다. 그 밖에 데몬들에서 주목할만한 릭을 발견하지 못한 이유는 이들의 소스가 공개되어있어 지속적인 테스트를 거쳐 안정화되었을 것으로 생각된다. 따라서, 데몬보다는, 표 2를 통해 GUI 에서 메모리 할당이 훨씬 활발하게 일어나고 있다는 사실을 기반으로 간단한 GUI 프로그램을 선정하여 테스트해보았다. lshw, xpdf, xmms, worker 가 그러한 관점에서 테스트된 프로그램이다.

4.2 lshw : 하드웨어 리소스 체크 리스터

이 프로그램에서는 4번의 메모리 릭 징후가 탐지되었다. 그 중 2번은 공유 라이브러리(libc-2.3.2.so)에서 제공하는 함수중의 getline()함수에서 내부적으로 호출되는 getdelim() 함수에 문제가 있다는 것을 발견하였다. getline() 함수는 궁극적으로 오픈된 파일에서 한 줄의 내용을 읽어오는 함수로, getline()함수 내에서는 무조건 getdelim()함수를 호출하게 되어있다. getdelim()함수는 기본적으로 120 바이트를 할당하도록 되어있다. 그런데, getdelim()함수가 한 라인읽기 실패로 -1을 리턴할 경우, 리턴전에 할당했던 120 바이트를 반환시켰다면 릭이 일어나지 않았을 것이다. 그러나 그 경우 조차도 반환 처리 책임을 개발자에게 맡김으로써 릭이 발생되었다. 이 부분은 오히려 개발자의 오류라기보다는 공유라이브러리 개발자가 실수한 것이다. 왜냐하면 매뉴얼[8]에조차 -1로 리턴될 때도 반환 처리를 꼭 수행해야한다는 안내가 없었기 때문에 개발자로서는 이러한 사실을 알 수 없기 때문이다.

나머지 2번은 동일한 라이브러리에서 제공하는 scandir() 함수와 관련하여 발생하는 오류로, scandir()함수는 지정경로를 스캔하고 인자로 전달받은 namelist 를 배열형태로 메모리를 할당하고 관련정보를 채워 리턴하는 함수이다. 이 함수 호출 이후, 메모리 반환 책임은 전적으로 개발자에게 있다. 그러나 이 프로그램에서는 개발자가 scandir() 함수를 통해 전달받은 namelist 배열의 데이터를 for 루프 안에서 문자열 매칭하여, 매칭 성공시 데이터의 사용과 반환 처리를 수행하고, 매칭 실패시 아무런 처리도 수행하지 않게 되어있다. 따라서, 매칭 실패에도 반환 처리는 수행해야함에도 그러한 처리를 생략함으로써 릭이 발생하게 된 것이다.

이 실험을 통해 메모리 릭의 특징을 일반화할 수는 없지만, 라이브러리에서든, 응용 프로그램에서든 정상적인 로직에서는 문제 발생 확률이 적고, 에러로 리턴 될 때 좀 더 섬세하게 처리되어야함을 보여주고 있다.

4.3 xmms / xpdf / worker

xmms는 오디오플레이어, xpdf는 PDF 파일 뷰어, worker는 파일매니저 프로그램이다. 이 세 프로그램은 GUI 형태의 응용 프로그램으로, GUI가 아닌 응용프로그램보다 릭 원인을 찾기위한 추적이 매우 어려웠다. 추적이 어려운 이유는 첫째로 C++로 구현된 응용의 경우 클래스가 갖는 특징에 따라, 컨스트럭터(constructor)와 상속(inheritance)의 특성을 내포하고 있고 다수의 X 라이브러리들이 링크되어 있어, 하나의 인터페이스 호출이 실제로는 내부적으로 여러단계의 함수 호출로 이어져 추적이 매우 어려웠다. 그림 2는 xpdf의 로그 일부로, 원인 추적이 얼마나 어려운지 잘 나타내주고 있다. ①~④으로 표시된 것은 소스 그룹단위로 볼 때 외부 인터페이스 호출을 의미하며, 그 그룹 내부에서도 여러 함수 호출이 연쇄적으로 호출되고 있다. 추적을 위해서는 공유 라이브러리의 소스 조차 모두 보유하고 있어야만 추적이 용이하다는 것을 알 수 있다. 둘째로 이 로그의 내용은 메모리 릭 발생 지점을 표시해주는 것이 아니라 어디에서 할당된 메모리가 릭 징후를 보이는지 알려주는 것에 그친다. 즉, 그 메모리 포인터가 정작 반환처리할 정보를 잃은 지점을 알려주지 못하기 때문에, 이 로그만으로는 추적의 아주 작은 단서는 얻을 수 있으나, 결정적인 단서는 되지 못한다. 마지막으로 메모리 릭이라고 표시된 로그가 모두 문제가 되는 것은 아니라는 것을 알게되었다. 즉, 프로세스가 종료될 때 그 프로세스를 위한 가상메모리 영역도 같이 해지된다는 것은 모두가 아는 사실이다. 따라서, 종료될 때 마지막으로 썼던 메모리 영역을 반환하지 않고 종료되는 것은 문제가 되지 않는다. 메모리 릭이 실행 중에 메모리 반환을 위한 정보를 계속해서 잃는 것이 문제가 되는 것이다. 그러나, Valgrind 로그에서는 그 두 가지를 구분하여 알려주지 않고 있다. 따라서 로그만으로는 꼭 수정해야할 것과 무시해도 되는 것을 구분할 수 없기 때문에 수정이 필요한지에 대한 결정을 신속히 내리기 어렵다.

```
316 bytes in 7 blocks are definitely lost in loss record 336 of 430
   at 0x401A6A2: malloc (vg_replace_malloc.c:149)
   by 0x4346AFB: XQueryTree (in /usr/X11R6/lib/libX11.so.6.2) -----①
   by 0x4167784: XmGetVisibility (in /usr/X11R6/lib/libXm.so.3.0.1) ---②
   by 0x41673CE: (within /usr/X11R6/lib/libXm.so.3.0.1)
   by 0x4165D3E: _XmNavigSetValues (in /usr/X11R6/lib/libXm.so.3.0.1)
   by 0x41633C7: (within /usr/X11R6/lib/libXm.so.3.0.1)
   by 0x42B5762: (within /usr/X11R6/lib/libXt.so.6.0)          -------③
   by 0x42B571B: (within /usr/X11R6/lib/libXt.so.6.0)
   by 0x42B5CA2: XtSetValues (in /usr/X11R6/lib/libXt.so.6.0)
   by 0x42C8D9B: XtVaSetValues (in /usr/X11R6/lib/libXt.so.6.0)
   by 0x80C6D97: XPDFViewer::updateCbk(void*, GString*, int, int, char*)
(PDFDoc.h:211) -------④
```

그림 2 xpdf의 로그 중 일부

```
struct A {
private:
    int* pLost;
    int* pNoFree;
public:
    void f();
    void g();
};
void A::f()
{
    pLost = (int*) malloc (sizeof(int)*10);     ---------①
}
void A::g()
{
    int nVal = 15;
    pLost = &nVal;              ----------------------②

    pNoFree = (int*) malloc (sizeof(int)*5);    ---------③
}
int main ()
{
    A a;
    a.f();
    a.g();
}
```

그림 3 테스트 프로그램 예

앞의 상황과 관련된 이해를 위해 그림 3에 간단한 프로그램을 구현하여 테스트해보았다. A클래스안에 f() 함수와 g()함수를 각각 선언하였다. f()함수는 메모리 할당 주소를 pLost 라는 포인터 변수에 저장하고, g()함수는 로컬 변수 nVal의 주소를 pLost 포인터 변수에 저장하도록 함으로써 메모리 반환 정보를 잃게 만들었다. 또한 pNoFree 포인터 변수에 메모리를 할당하여 주소를 저장하도록 하여, 종료시 메모리 반환을 수행하지 않고 종료되도록 하였다. 이러한 내용으로 메모리 릭을 체크한 결과의 일부가 그림 4의 내용이다. 그림 4의 ①은 g()함수에서 malloc() 함수를 호출하고, ②는 f()함수에서 malloc() 함수를 호출하였다는 로그만 나와있다. 즉, g()함수에서 malloc에 의한 포인터 변수가 반환처리를 하지 않았기 때문에 그 변수의 메모리 할당 수행 위치가 ①로 표시되었다. 또한, g()함수내에서 pLost 포인터 변수가 메모리 할당 주소를 잃게 되었기 때문에 그 pLost 포인터 변수의 메모리 할당 수행한 위치가 ②로 표시된 것이다. ①과 ②의 릭 원인이 다름에도 로그는 동일하게 표시되어 오류에 관한 추측이 난해함을 알 수 있다.

그 밖에 C와 C++로 구현된 응용 프로그램에서 메모리 할당 빈도와 메모리 릭의 관점에서 테스트한 결과 언어적인 차이점은 발견되지 않았다. 앞에서 소개한 논문[4]에서의 메모리 할당의 빈도가 C보다 C++에서 높다는 언급은 잘못된 것이라는 사실을 알 수 있었다. 그 이유는 C에서와 마찬가지로 C++도 명시적으로 메모리 할당을 위해서는 malloc()이나 calloc()과 같은 함수와 new를 통해 수행되는 것으로 C++라고 단순히 그 횟수가 높아질리 없다. 다만, C++에서는 클래스 개념이 있어, 클래스를 사용할 때 오브젝트 선언 후 사용하도록 되어있기 때문에 스택 메모리 사용은 C보다 더 활발할 수 있겠지만, 그것은 메모리 릭과는 무관하다. 따라서 C++가 메모리 릭을 발생시키기 더쉽다고 생각하는 것은 잘못된 생각이다. 오히려 메모리 반환을 개발자에게 전적으로 맡기기보다는 클래스의 디스트럭터(destructor)를 이용하여 릭을 방지할 수 있어 C++가 메모리 릭 측면에서는 더 안정된 장치를 제공하는 언어라고 생각된다.

```
20 bytes in 1 blocks are definitely lost in loss record 1 of 2
    at 0x401A6A2: malloc (vg_replace_malloc.c:149)
    by 0x8048544: A::g() (in /work/Class)     ------------①
    by 0x804857A: main (in /work/Class)

40 bytes in 1 blocks are definitely lost in loss record 2 of 2
    at 0x401A6A2: malloc (vg_replace_malloc.c:149)
    by 0x8048517: A::f() (in /work/Class)     ------------②
    by 0x804856B: main (in /work/Class)
```

그림 4 메모리 릭 체크 결과

5. 결 론

본 논문에서 목표한 실제 응용에서의 메모리 릭 발생 정도가 어느 정도인지를 몇 가지 어플리케이션을 통해 확인하였다. 확인된 사항을 정리해보면, 첫째, C/C++의 언어적인 특성으로 메모리 릭이 많이 발생한다는 주장과는 달리 실제 사용되는 프로그램들에서는 우려할만한 메모리 릭은 발견하기 어려웠다. 둘째, 몇몇 프로그램에서 오류 처리에 대한 미흡한 처리로 메모리 릭이 발생하는 것을 발견하게 되었고, 그러한 실수는 공유 라이브러리 개발자나 어플리케이션 개발자나 마찬가지였다. 셋째, C와 C++언어 사이에 메모리 릭 측면과 메모리 할당 횟수 측면에서 특별한 차이가 없음을 확인하였다. 마지막으로 Valgrind 툴로 메모리 릭의 횟 수는 발견하기 편리하였으나 원인을 추적하기에는 다소 어려웠다. 메모리 릭 징후를 보이는 포인터의 메모리 할당 위치를 제시는 해주나, 실제로 반환 정보를 잃게되는 위치는 알려주지 않아 가능성을 모두 적용해보야 한다는 번거로움이 있으며, 반환 처리되지 않는 릭 징후와 실행 중 계속 반환 정보를 잃는 것과의 구분된 로그가 제공되지 않아 불편했다. 이러한 과정을 통해 동적 메모리 디버깅 툴이 어떤 정보를 제공해야하는지를 알게 되었고, 개선의 필요성도 알게 되었다. 향후 이러한 문제를 개선하여 좀 더 추적에 도움이 되는 툴을 개발할 계획이다.

참고문헌

[1] http://cplus.about.com/od/cprogramin1/1/bldef_memleak.htm
[2] http://en.wikipedia.org
[3] 정의현,김성진, 자바2 JDK1.3, p9, 2002
[4] Brian Willard, Ophir Frieder "Autonomous Garbage Collection:Resolving Memory Leaks in Long Running Network Applications", Computer Communications and Networks, 1998
[5] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas, "iWatcher: Efficient Architectural Support for Software Debugging", International Symposium on Computer Architecture(ISCA'04), 2004
[6] lshw, xpdf, xmms, worker, http://freshmeat.net
[7] http://valgrind.org/, valgrind documentation, p49, June, 2006
[8] http://www.die.net/doc/linux/man/man3/getline.3.html