동적 삽입 및 제거가 가능한 사용자 수준의 모듈 프레임워크 설계

임성락*, 유영창*
*호서대학교 컴퓨터공학과
e-mail:srrim@hoseo.edu

A Design of User-level Module Framework for Dynamic Insertion and Removal

Seong-Rak Rim*, Young-Chang Yoo*
*Dept of Computer Engineering, Hoseo University

요 약

본 논문에서는 실행 중인 모듈 프로그램에 새로운 모듈의 삽입 및 제거가 가능한 사용자 수준 모듈 프레임워크(UMF: User-level Module Framework)를 제시한다. 제시한 UMF은 하나의 메인 모듈과 여러 개의 서브 모듈들로 구성되며 서브 모듈은 메인 모듈에 동적으로 삽입 및 제거된다. 제시한 UMF의 타당성을 검토하기 위하여 리눅스 환경에서 GCC 컴파일러의 PIE(Position Independent Executables)옵션을 이용하여 사용자 수준의 메인 모듈과 서브 모듈을 생성하여 동적 삽입 및 제거 기능을 실험한다.

1. 서론

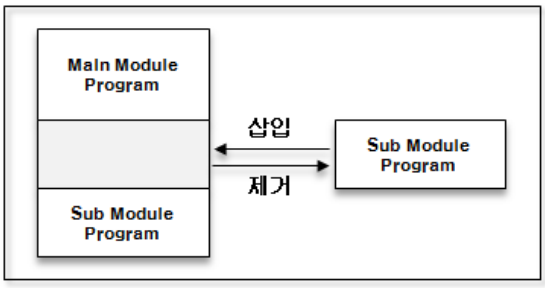
프로그램은 개발, 운용, 유지, 확장 등의 생명 주기 동안 구성 소스 크기는 커지고 분석하기 어려워지는데 이런 문제를 소프트웨어 위기(software crisis)라는 용어로 표현하고 있다[1]. 이러한 위기를 극복하기 위해서는 프로그램 개발 과정에서 기능의 정확성은 물론 기능의 확장성과 디버깅의 용이성을 고려해야 한다.

일반적으로 기능적 확장이나 디버깅 과정에서 수정된 기능의 적용을 위해서는 프로그램의 중지와 재시작이라는 반복적인 작업이 필수적으로 수반된다. 이 방법은 무정전 시스템, 웹 서비스 그리고 온라인 게임과 같이 시스템의 동작 정지 시간을 최소화하기를 요구하는 경우에는 적절한 해결책으로 사용되지 못하고 있다.

이러한 문제점을 해결하기 위한 방법으로 리눅스 환경에서 커널 모듈의 동적 삽입 및 제거 기능을 제공하고 있다[2]. 본 논문에서는 커널 수준이 아닌 사용자 수준 모듈의 동적 삽입 및 제거가 가능한 사용자 수준의 모듈 프레임워크(UMF: User level Module Framework)를 제시한다.

2. UMF(User-level Module Framework) 설계

제시한 UMF의 기본 개념은 (그림 1)과 같이 하나의 메인 모듈과 여러 개의 서브 모듈로 구성된다. 기본적으로 메인 모듈 프로그램이 실행되고 있는 동안 새로운 서브 모듈 프로그램을 삽입하거나 메인 모듈에 삽입된 서브 모듈 프로그램을 제거할 수 있도록 하는 것이다.



(그림 1) 모듈의 동작 개념도

(그림 1)에서 메인 모듈 프로그램을 먼저 실행시킨다. 실행 중인 메인 모듈은 임의의 서브 모듈로부터 '삽입' 혹은 '제거' 요청을 받아들인다. '삽입' 요청일 경우 지정한 서브 모듈 프로그램을 메인 모듈에 삽입시키고, '제거' 요청일 경우 지정한 서브 모듈 프로그램을 메인 모듈로 부터 제거시키도록 한다.

2.1 모듈 프레임워크 구성요소

메인 모듈은 모듈 관리에 필요한 자료 구조를 초기화하고 프로세스 간 통신을 위한 서버 기능을 구축한 후, 서브 모듈의 삽입 및 제거 요청을 받아들이기 위한 대기상태로 진입한다. 서브 모듈로부터 삽입 및 제거 요청을 하면 이에 대한 처리를 수행하고 다시 요청 대기 상태로 진입한다.

서브 모듈로부터의 요청에 따라 동적 삽입 및 제거 기

능을 수행하기 위한 메인 모듈의 프레임워크 다음과 같은 블록으로 구성한다.

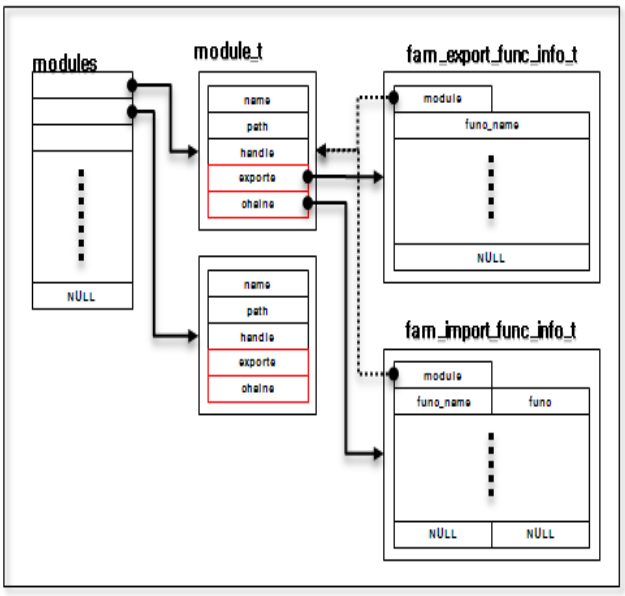
- 모듈을 관리하기 위한 자료 구조 데이터 블록
- 모듈을 관리하기 위한 함수 코드 블록
- 삽입 및 제거 요청을 받아들이기 위한 IPC 블록
- 모듈 실행을 위한 main()함수를 포함한 블록

서브 모듈은 메인 모듈에게 삽입 및 제거를 요청하며 메인 모듈에 삽입되어 자신의 기능을 수행하게 된다. 동적 삽입 및 제거 기능을 수행하기 위한 서브 모듈의 프레임워크 다음과 같은 블록으로 구성한다.

- Sub 모듈은 모듈의 기능 구현을 위한 기능 코드 블록
- 다른 모듈에서 사용할 수 있도록 선언된 함수들의 정보를 가지고 있는 EXPORT 블록
- 다른 모듈에서 구현되어 있어서 연결되어야 하는 함수들의 정보를 가지고 있는 IMPORT 블록
- 삽입 요청 처리를 위한 초기화 코드 블록
- 제거 요청 처리를 위한 종료 코드 블록
- 모듈 실행을 위한 main()함수를 포함한 블록

2.2 모듈 프레임워크 자료구조

메인 모듈 프로그램은 모듈을 관리하기 위해서 (그림 2)와 같은 구조체 형태의 자료구조를 가진다. 서브 모듈은 modules이라는 배열 형태의 리스트에 의해서 관리한다.



(그림 2) 모듈 관리 자료 구조

모듈을 관리하기 위한 하나의 배열 요소는 module_t라는 자료 구조를 통해서 표현되며 하나의 모듈을 표현하는 자료구조 module_t는 (그림 3)과 같은 구조체 형태로 설계한다.

```
struct module_t
{
   s8     name[FAM_MODULE_NAME_MAX];  ---------- ①
   s8     path[FAM_MODULE_PATH_MAX];  ---------- ②
   void * handle;                     ---------- ③

   fam_export_func_info_t * exports;  ---------- ④
   fam_import_func_info_t  * chains;  ---------- ⑤
};
```

(그림 3) 모듈 구조체

(그림 3)에서 name은 각각의 모듈을 구별하기 위한 문자열이고(①), path는 모듈 파일 위치를 나타내는 시스템 상의 정보 문자열이다(②). handle은 path와 dlopen()함수를 이용하여 모듈 파일을 공유 라이브러리로 열고 반환된 핸들 값을 저장한다(③). exports는 모듈이 외부에 제공하는 함수 정보를 관리하기 위해서 등록 요청되었을 때 할당되는 정보의 메모리 선두 주소이다(④). chain은 모듈이 참조하는 외부 함수 정보를 관리하기 위해서 등록 요청되었을 때 할당되는 메모리 선두 주소이다(⑤).

2.2.1 exports 구조체 fam_export_func_t

다른 모듈에서 참조 가능한 함수에 대한 정보를 제공하기 위한 구조체(fam_export_func_info_t)는 (그림 4)와 같이 설계한다.

```
struct fam_export_func_info_t
{
  module_t *  module;
  s8 *        func_name;      ---------- ①
                              ---------- ②
} ;
```

(그림 4) EXPORT 자료 구조

(그림 4)에서 module은 함수를 외부에 공개하는 모듈의 정보이고(①), func_name은 함수의 이름 문자열이다(②).

2.2.2 chains 구조체 fam_import_func_t

모듈이 참조하는 외부 함수 심볼 목록과 함수를 동적으로 연결하기 위한 함수 포인터 목록을 포함한 구조체(fam_import_func_info_t)는 (그림 5)와 같이 설계한다.

```
struct fam_import_func_info_t
{
   module_t * module;          ---------- ①
   s8 *       func_name;       ---------- ②
   void *     (**func)(void);  ---------- ③
} ;
```

(그림 5) IMPORT 자료 구조

(그림 5)에서 module은 함수를 사용하는 모듈의 정보이고(①), func_name은 함수의 이름 문자열이다(②). func은 실제 함수 포인터 변수의 주소이다(③).

2.2.3 함수 연결 루틴

함수의 실제 주소를 가져오는 루틴은 서브 모듈이 동적 라이브러리 특성을 가지는 것을 이용하여 (그림 6)과 같이 설계한다.

```
void * (*dl_func)(void);
char * error;

dl_func = dlsym(module->handle, (const char *) chain->func_name );-------- ①
if ((error = dlerror()) != NULL)
{
   fputs(error, stderr);                                                  ②
   return NULL;
}

*chain->func = dl_func;  ------------------------------------------------- ③
```

(그림 6) 함수 연결 루틴

(그림 6)에서 dlsym()함수는 동적 라이브러리 핸들 값인 module->handle과 chain->func_name에 포함된 함수 심볼 문자열을 인자로 호출한다. 그 결과 값인 함수 주소를 dl_func 변수에 저장한다(①). 만약 핸들이 유효하지 않거나 함수 심볼이 존재하지 않아 함수 주소를 얻어 올 수 없다면 에러를 표시하고 NULL값을 반환하여 에러 상태를 표시한다(②). 정상적으로 함수 주소를 얻어 왔다면 chain->func에 저장한다(③).

2.2.4 함수 제거 루틴

모듈이 제거되면 모듈 간에 연결된 함수들의 연결을 제거하기 위해 (그림 7)과 같이 설계한다.

```
int                     lp;                                              ①
module_t                *work_module;
fam_chain_func_info_t   *work_chains;

work_module = NULL;
for( lp=0;lp< module_list->fcount; lp++ )
{
  work_module   = get_module( module_list, lp );
  work_chains   = work_module->chains;

  while(( work_chains != NULL)&&(work_chains->func_name!=NULL))
  {                                                                       ②
    if(strcmp(work_chains->func_name,func_name ))
    {                                                                     ③
      *(work_chains->func) = __fam_call_default;
    }
    work_chains++;
  }
}
```

(그림 7) 함수 제거 루틴

(그림 7)에서 존재하는 모든 모듈을 찾고, 모듈 간에 연결된 함수를 해제하기 위해 module_t 구조체 포인터 변수 work_module과 fam_chain_func_info_t 구조체 포인터 변수 work_chains 변수를 사용한다(①). get_module()함수와 work_module변수 사용하여 모듈 전체를 검색한다(②). 모듈 내부에서 연결된 함수 정보를 work_module->chains와 work_chains 변수를 이용하여 검색한다(③). 검색 과정에서 연결 제거 대상 함수명 문자열을 가지고 있는 func_name 변수와 work_chains->func_name 변수가 동일하다면 해당 연결 함수 포인터 변수 work_chains->func에 __fam_call_default함수를 대입한다. 이 과정은 연결이 해제된 함수의 참조에 의한 메모리 폴트 에러를 방지하기 위한 것이다.

3. 구현 및 실험

제시한 UMF의 타당성을 검토하기 위하여 리눅스 환경에서 GCC 컴파일러의 PIE(Position Independent Executables)옵션[3]을 이용하여 메인 모듈(app_base), 그리고 서브 모듈1(module_l1)과 서브 모듈2(module_l2)를 생성한다. 서브 모듈1에 api1()과 api2()함수를 작성하고 메인 모듈에 삽입한 후 서브 모듈2에서 api1()함수와 api2()함수를 호출하여 연결된 함수가 호출되는지를 검사한다.

서브 모듈1은 다른 모듈이 참조할 수 있는 함수목록을 제공하기 위하여 구조체(fam_export_func_info_t)를 (그림 8)과 같이 구현한다.

```
fam_export_func_info_t fam_export_module_l1[] =
{
  {
    .module = NULL,                                                       ①
    .func_name = (s8 *) "m1_api1",
  },
  {
    .module = NULL,                                                       ②
    .func_name = (s8 *) "m1_api2",
  },
 { NULL,NULL },                                                           ③
}
```

(그림 8) EXPORT 구현

(그림 8)에서 m1_api1()함수를 외부에 공개하기 위해서 .module 필드와 .func_name 필드를 설정하고(①), m1_api2()함수를 외부에 공개하기 위해서 .module 필드와 .func_name 필드를 설정한다(②). .module 필드는 메인 모듈의 UMF의 모듈 관리 루틴에서 모듈이 삽입되면서 설정되므로 Sub 모듈은 NULL로 설정한다. .func_name 필드는 공개되는 함수의 문자열 주소를 설정한다. 공개되는 함수 목록의 자료의 끝을 나타내기 위해서 .module 필드와 .func_name 필드를 NULL로 구조체 배열 마지막 부분에 기술한다. (③).

서브 모듈2는 다른 모듈에서 참조해야 하는 함수 목록을 제공하기 위하여 구조체(fam_import_func_info_t)를 (그림 9)와 같이 구현한다.

```
void * (*m1_api1) ( void          ) = __fam_call_default; ------------------ ①
void * (*m1_api2) ( int a, int b  ) = __fam_call_default; ------------------ ②
fam_chain_func_info_t fam_chain_module_l1[] =
{
  {
    .module   = NULL,
    .func_name = (s8 *) "m1_api1",                                        ③
    .func      = (void *(**)(void)) &m1_api1,
  },
  {
    .module = NULL,
    .func_name = (s8 *) "m1_api2",                                        ④
    .func = (void *(**)(void)) &m1_api2,
  },
  { NULL,NULL },  ------------------------------------------------------- ⑤
};
```

(그림 9) IMPORT 구현

Sub 모듈간의 함수 호출은 정적 호출 방식이 불가능하다. UMF에서는 함수 포인터 변수를 이용하여 호출하는

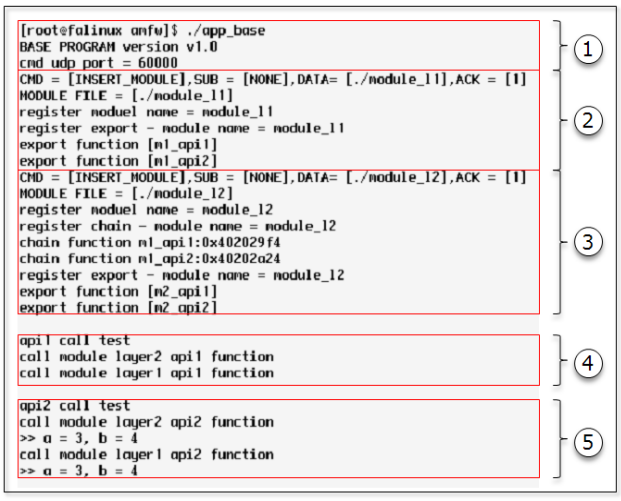
방식으로 구현한다. Sub 모듈의 IMPORT 구현은 함수 포인터 변수 구현 부분과 외부에 공개하기 위한 구조체 자료 구조 구현부분으로 나누어진다.

(그림 9)에서 m1_api1(), m1_api2()함수를 제공하기 위해 함수 포인터변수를 선언하며, 다른 모듈에서 해당 함수를 제공하지 않을 경우 함수 호출과 동작의 실패를 방지하기 위해 __fam_call_default()함수를 초기 값으로 설정한다(①②).

참조되는 m1_api1()함수를 외부에 공개하기 위해서 .module,.func_name,.func 필드를 설정하고(③), m1_api2()함수를 외부에 공개하기 위해서 .module, .func_name, .func 필드를 설정한다(④). .module 필드는 메인 모듈의 UMF의 모듈 관리 루틴에서 모듈이 삽입되면서 설정되므로 Sub 모듈은 NULL로 설정한다. .func_name 필드는 참조되는 함수의 문자열 주소를 설정한다. 참조되는 함수 목록의 자료 끝을 나타내기 위해서 .module, .func_name, .func 필드를 NULL로 구조체 배열 마지막 부분에 기술한다. (⑤).

실행 가능한 공유라이브러리를 만들기 위해서는 "gcc -fPIE -c -g -\all lib_main.c" 와 같은 형식으로 컴파일하여 오브젝트를 만들고 "gcc -o run_lib -pie -rdynamic lib_main.o"와 같은 형식으로 실행파일을 만든다. 이렇게 만들어진 실행 가능한 공유 라이브러리는 "gcc -o call_lib call_main.c ./run_lib"와 같은 형식으로 링크하여 사용할 수 있다.

(그림 10)에서 메인 모듈 프로그램인 app_base는 실행 후 UDP포트로 대기 상태로 진입한다(①). 서브 모듈1(module_l1)이 삽입되어 모듈명과 외부 공개 함수가 등록된다(②). 서브 모듈2(module_l2)가 삽입되어 모듈명과 외부 공개 함수가 등록되고 서브 모듈1의 함수와 연결된다(③). 서브 모듈2에서 모듈 api1()함수를 호출하면 연결된 서브 모듈1의 api1()함수가 호출된다(④). 서브 모듈2에서 api2()함수를 호출하면 연결된 서브 모듈2의 api2()함수가 호출된다(⑤). 따라서 모듈이 등록되고 상위 모듈에 연결된 하위 모듈의 함수가 호출되는 것을 확인 할 수 있다.



(그림 10) 실험 결과

4. 결론

프로그램의 디버깅, 기능개선 및 확장은 프로그램 개발 이후에 필수적으로 발생하는 소프트웨어 생명 주기이다. 시스템의 정지기간의 최소화는 서버 시스템뿐만 아니라 임베디드 소프트웨어까지도 요구되는 사항이다.

본 논문에서는 이에 대한 대안으로 동적 삽입 및 제거가 가능한 사용자 수준의 모듈 프레임워크를 제시하였다. 링크 구조가 단순한 C 언어에서 PIE옵션의 컴파일 옵션을 이용하여 구현하여 제시한 모듈 프레임워크의 동적 삽입이 가능함을 실험하였다.

참고문헌

[1] NATO Software Engineering Conference 1968.

[2] Daniel P. Bovet, Marco Cesati 지음, 박장수 옮김, "리눅스 커널의 이해 (Understanding the Linux Kernel)", 3rd edition, 한빛미디어, 2006

[3] 타카바야시 사토루, 우카이 후미토시, 사토 유스케, 하마지 신이치로, 슈토 카즈유키 공저/진명조 역, BINARY HACKS, Page 144