**Pintos Project 1: User Program (1)**

담당 교수 : 문의현

학번 / 이름 : 20171759 / 신재현

개발 기간 : 20221005 ~ 20221031

1. **개발 목표**

본 프로젝트에서는 간단한 시스템 콜을 구현하는 기능을 개발하는 것을 목표로 한다. 이를 위해서 사용자가 전달 한 argument의 스택을 쌓는 과정을 우선적으로 구현한다. 또, 이 과정에서 서로 다른 프로세스 간의 race condition을 막고, 부모-자식 프로세스 관계를 설정하는 방법을 익힌다. 마지막으로, 시스템 콜을 실제로 구현해서, 사용자 프로세스가 커널을 사용할 수 있도록 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Argument Passing

사용자가 프로그램을 실행하기 위해서 커널에게 각종 명령어를 전달한다. 이 때, 사용자가 전달 한 명령어를 스택에 쌓기 위해서 esp 포인터를 사용해서 스택에 쌓는다. 이 때, 사용자가 전달 한 명령어 및 해당 명령어에 대한 주소 등이 page directory의 page에 정해진 규칙을 기반으로 쌓이게 되고, 마지막 명령어를 esp가 가리키는 상태로 마친다.

1. User Memory Access

서로 다른 프로세스가 동시에 스택을 쌓을 수도 있다. 또, 부모 프로세스가 자식 프로세스가 끝날 때 까지 기다려서, 자식 프로세스가 사용하던 자원을 정리해야 한다. 즉, 공유 영역에 접근하는 임계 영역(critical section)에 서로 다른 process가 들어가지 않도록 설계해야 한다. 또, 부모 프로세스가 자식 프로세스를 기다려야 한다. 이를 구현하기 위해서 동기화 기법 중 semaphore을 사용해서 개발하는 것으로 한다.

한편, 공유 자원에 접근하는 동안에는 커널 영역의 메모리에 접근하지 않도록 사용자가 접근하려는 메모리의 유효성을 검증해야 한다. 만약 사용자가 적절치 못한 메모리에 접근하거나, 존재하지 않는 메모리 주소에 접근하려고 하는 경우, 커널의 보호를 위해서 프로세스의 진행을 막아야 한다.

1. System Calls

System call에는 크게 9 가지 종류를 구현하도록 한다. 크게 halt, exit, exec, wait, read, write, fiboancci, max\_of\_four\_int를 구현한다. 현재 탬플릿 코드는, 사용자가 시스템 콜을 호출했을 때, 그 어떠한 시스템 콜도 이를 받지 않도록 되어 있다. 이를 해결하기 위해서 위에 명시 한 9가지 종류의 system call을 구현해서, 각각의 명령이 사용자 프로세스를 통해서 시스템 콜에 전달되었을 때, 커널 영역에서 적절 한 명령이 수행되도록 설계한다.

* 1. **개발 내용**
* Argument Passing

Thread\_create를 통해서 스레드가 생성되면, 스케쥴링을 거쳐서 start\_process가 실행된다. Start\_process에서는 인수로 전달받은 argument를 파싱하도록 명령한다. 이를 위해서 intr\_frame이 생성되고, esp와 eip가 load 함수로 전달된다. Load 함수는 전달받은 명령문을 파싱한다. 이 때, strtok 함수를 사용해서 띄어쓰기(“ “)를 delimiter로 설정해서 각 단어를 파싱했다. 각 argument는 새로운 배열인 argv에 저장 되었다. 이후, load 함수에서 page가 생성되고, 해당 page에 esp 포인터가 움직이면서 argument를 실제로 스택에 쌓게 된다.

스택에 쌓는 과정은 총 6개의 절차로 구성된다. 우선, 각각의 파싱 된 argument를 esp를 이용해서 스택에 쌓는다. 이 때, 각 명령어의 길이를 저장한다. 모든 명령어를 stack에 쌓게 되면, 4byte 단위로 다음 정보를 stack에 쌓기 위해서 4byte에서 부족한 만큼을 word-align을 위해 0으로 채운다. 또, 명령어의 공간이 끝났음을 표시하기 위해 NULL pointer sentinel을 추기한다. 역시 4byte 0이다. 그 다음에는 차례로 각 명령어의 주소를 넣고, 이후 명령어의 시작 주소를 넣는다. 마지막으로 명령어의 개수를 넣고 return address를 넣고 스택 쌓기는 끝난다. 이 때, 핀토스는 하나의 스레드로 이루어진 OS 이므로 return address는 0으로 항상 전달한다.

* User Memory Access

핀토스에서는 default로 PHYS\_BASE를 0xC0000000로 설정한다. 위 주소 이하는 사용자의 프로그램이 자유롭게 사용할 수 있는 user area이며, 위 주소 이상은 커널이 배타적으로 사용하는 kernel area다. 만약 user area가 kernel area를 침범하려고 할 경우, 잘못된 메모리를 엑세스 하는 것으로 판단하고, 해당 명령어를 중단시켜야 한다. 이를 구현하기 위해서 본 프로젝트에서는 syscall.c에 check\_address라는 함수를 구현했다. 위 함수는 사용자가 스택에 넣고자 하는 명령어들의 주소를 확인한다. 만약 전달 받은 주소가 page에 매핑되지 않은 가상 주소거나, 주소가 커널 영역에 속할 경우 즉시 exit(-1)을 호출해서 커널에게 실패로 인한 exit임을 알려준다. 이를 구현하기 위해서는 userprog/pagedir.h에 속한 pagedir\_get\_page를 이용해서 매핑 된 주소인지를 확인한다. 또, 커널 영역에 속한 주소인지를 확인하기 위해서 threads/vaddr.h에 속한 is\_user\_vaddr을 이용한다.

* System Calls

시스템 콜은 사용자 프로세스가 커널 영역에 접근하기 위해서 필요하다. 앞서 설명한 바와 같이, 사용자 프로세스에서 커널 메모리 주소에 접근할 경우 오류를 발생하도록 시스템이 설계되어 있다. 그래서 커널 영역에 속한 프로세스의 기능을 사용하기 위해서는, 커널에서 적절한 API를 제공해야 한다. 이를 시스템 콜이라고 부른다. 사용자 프로세스에서는 시스템 콜을 호출할 수 있고, 이 때 OS는 커널 모드로 진입해서 합법적으로 커널 프로세스를 진행하게 된다.

본 프로젝트에서는 총 9가지의 시스템 콜을 구현한다.

1. Halt : 핀토스를 종료하는 시스템 콜이다. shutdown\_power\_off를 호출해서 핀토스를 종료한다.
2. Exit : 사용자의 프로세스를 종료하는 시스템 콜이다. 현재 스레드에 대한 정보를 가져온 뒤, 스레드의 exit status를 확인해서 사용자에게 exit 코드를 프롬프트 한 뒤 thread\_exit을 호출해서 최종적으로 스레드를 종료한다.
3. Exec : 프로세스를 실제로 실행하는 시스템 콜이다. 사용자가 전달한 argument를 process\_execute를 통해서 전달하고, 해당 argument를 stack에 쌓은 뒤, 자식 프로세스가 실행될 때 까지 기다렸다가 종료하는 일종의 thread lifecycle을 돌리기 위해서 구현 된 시스템 콜이다.
4. Wait : 자식 프로세스가 종료 될 때 까지 기다리도록 하는 시스템 콜이다. Process\_wait를 통해서 구현한다.
5. Read/Write : 사용자로부터 데이터를 읽는 시스템 콜이다. 현재 프로젝트에서는 standard input/output에서만 데이터를 읽고 쓰도록 설계되었다. 다음 프로젝트에서 이를 file system에서 읽고 쓰는 것으로 변경 한다.
6. Fibonacci : 피보나치 함수를 구현하는 시스템 콜이다.
7. Max of four : 전달 된 네 개의 명령 인자 중에서 가장 큰 수를 찾는 프로세스다. Fibonacci와 Max of four 모두 특별한 커널의 API를 사용하지 않기에 따로 라이브러리를 호출하지는 않는다.

사용자 영역에서는 우선적으로 main 함수가 실행된다. Main 함수가 실행되면 run\_actions에서 main으로부터 명령어가 들어있는 argv를 받는다. 그리고 run\_task에서 process\_wait와 process\_execute를 호출해서 사용자의 시스템 콜에 해당하는 process를 생성하고 구동한다. Process\_execute에서는 사용자가 전달한 명령어를 적절히 파싱해서, 스택에 쌓게 된다. 이후, load 함수에서 명령어를 스택에 쌓은 뒤 시스템 콜을 구현하는 함수(syscall.c)를 호출한다. 이 때, syscall n이 호출되는데, 이 때 n에는 명령어의 개수가 들어간다. 이후 intr\_handler를 거쳐syscall\_handler를 통해서 system call이 작동되고, 이 때 스택의 마지막을 가리키는 esp를 통해서 명령어를 가져와서 실행한다.

모든 시스템 콜이 수행된 뒤, 시스템 콜은 유저 레벨로 적절한 리턴 값을 반환하거나 혹은 시스템을 종료한다. 리턴 값을 반환할 때는 eax 레지스터에 값을 저장하는 방식을 택한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

1차 개발작업 : 우선적으로 사용자가 보내는 명령어의 파싱을 진행한다. 파싱을 한 뒤, stack에 쌓는 작업까지 진행해서 hex\_dump를 호출했을 때, 명령어의 누락이 발생하지 않는 것을 확인한다. 현 개발 단계까지 마무리되면, echo를 통해서 어떤 명령어를 전달하더라도 모두 정상적으로 dumping이 되는 것을 확인할 수 있다.

2차 개발작업 : 시스템 콜을 구현하기 위해서는 부모-자식 프로세스의 생성 작업이 수반되어야 한다. 여기에서 가장 중요한 것은, 부모 프로세스 입장에서 자식 프로세스가 종료될 때 까지 기다려야 한다. 즉, wait을 구현하고, wait을 진행 할 동안 자식 프로세스의 메모리 로드가 완료되었는지를 확인하는 load 세마포어도 사용해서, 프로세스의 계층 구조를 완성한다. 현 단계를 완성하게 되면, process wait에 무한 루프가 돌지 않아도 정상적으로 자식 프로세스가 종료될 때 까지 부모 프로세스가 기다리는 것을 확인할 수 있다.

3차 개발작업 : 마지막으로 시스템 콜을 구현한다. 시스템 콜을 구현하기 위해서는 syscall.c에 시스템 콜을 구현하는 함수를 선언해야 한다. 또, 사용자 정의 시스템 콜을 구현하기 위해서 몇 가지 시스템 콜을 구현하고 등록한다. 4개의 인자를 받을 수 있는 syscall4를 구현하고, 이를 max\_of\_four\_int와 연결해서 정상적으로 작동하는 것을 확인한다.

개발 검증 : 모든 개발이 완료되었으며, make grade 명령어를 통해서 정상적으로 핀토스가 작동하는 것을 확인한다. 만약 page fault 등이 뜰 경우, 세마포어 구현 부분에서 누락이 없는지 확인한다. 특히, child process를 구현하는 과정에서 page 생산이 많았으므로, 정상적으로 메모리가 동적 해제되어서 메모리 누수가 없는 지를 면밀하게 검토한다.

* 1. **개발 방법**

가장 중요한 것은 프로세스의 구조체를 수정하는 것이다. Thread.h의 struct thread에는 다음과 같은 요소가 추가되어야 한다.

* + - 부모-자식 프로세스를 list 자료구조로 연결하기 위해 필요한 요소. 부모 프로세스의 구조체 포인터, 자식 프로세스를 저장하는 list 자료구조, list 자료구조를 구현하기 위한 list\_elem가 이에 해당한다.
    - 현재 프로세스의 상태 관련 요소. 여기에는 세마포어와 메모리 적재 및 종료 여부, 그리고 종료 status 등이 포함된다. 세마포어는 메모리 적재 및 종료를 나타내도록 두 종류로 운영한다.

이를 기반으로 init\_thread 함수에서 thread initialize를 진행한다. 각각의 세마포어의 초기화를 진행하고, 상태 관련 요소를 0 혹은 알맞은 값으로 설정하는 작업을 진행한다. 또, thread\_exit에서는 exit 세마포어를 up 해서 exit이 완료되었음을 나타내도록 한다.

Process.c에서는 두 종류의 함수를 선언했다.

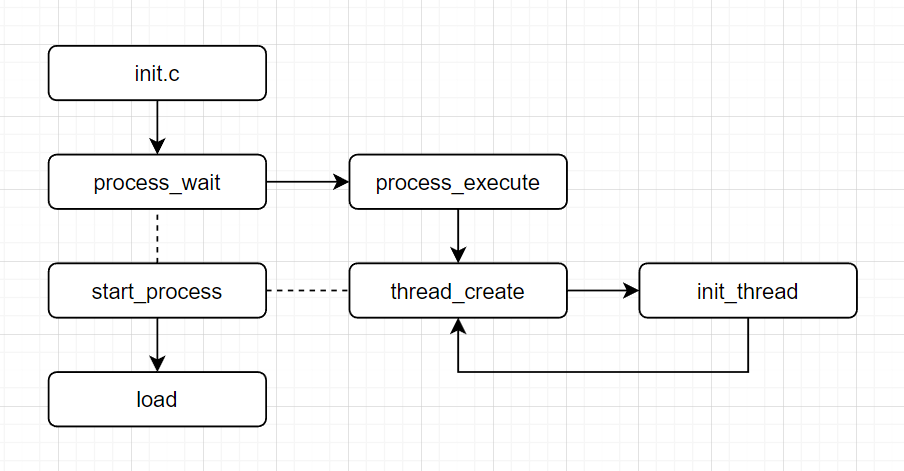
1. get\_child\_process : pid가 주어지면, 해당 pid가 현재 실행중인 프로세스의 자식 프로세스인지를 확인한다. 자식 프로세스가 아닐 경우, NULL을 리턴한다,
2. remove\_child\_process : 주어진 프로세스를 삭제하고, 프로세스가 담긴 page를 삭제해서 메모리 누수를 막는다.

또, Process.c에서 아래와 같은 내용을 추가한다.

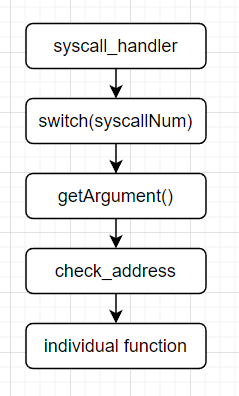
* + - start\_process : 사용자가 전달한 명령어를 스택에 쌓고, load sema\_up을 진행해서 다음 작업이 일어날 수 있도록 한다. 만약 메모리가 정상적으로 로드되지 않았다면, 프로세스를 종료한다.
    - process\_wait : exit 시스템 콜 혹은 오류로 인해서 thread\_exit이 호출 될 경우, exit 세마포어가 up 된다. 이 함수에서는 세마포어가 up 될 때까지 기다린 뒤, 자식 프로세스가 완료된 것을 확인하고, 자식 프로세스를 제거하는 작업을 진행한다.
    - process\_exit : 자식 프로세스가 종료 된 경우, 자식 프로세스를 zombie process로 만들지 않기 위해서 child의 parent를 NULL로 설정한다.
    - load : 명령어를 파싱해서 스택에 쌓는 작업을 진행한다.

이외에도 syscall.c에서는 시스템 콜을 구현하기 위한 함수를 모두 구현해야 하며, lib/user/syscall.c에서는 시스템 콜을 구현하기 위한 syscall 어셈블리 코드를 작성하고, syscall-nr.h에서는 사용자 정의 시스템 콜을 등록한다.

1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing



1. User Memory Access / System Call



* 1. **제작 내용**

1. Argument Passing

본 프로젝트에서 Argument를 passing 하는 과정에서는 특별히 다른 코드를 사용하지 않았다. Argument는 parsing 하는 코드는 아래와 같다.

텍스트, 화면, 스크린샷, 은색이(가) 표시된 사진

자동 생성된 설명

위 코드에서 볼 수 있다시피, 각각의 argument를 띄어쓰기를 delimiter로 지정해서 단어 별로 분리하였다. 이후, argv 배열에 추가했는데, 편의 상 범용적인 변수명을 사용했다. 또, 추후에 argc가 필요하기에, 현 단계에서 미리 구했다.

텍스트이(가) 표시된 사진

자동 생성된 설명

스택을 쌓기 위해서는 새로운 문자열 배열인 argAddrInStack을 선언했다. 이 배열에는 각 명령어가 몇 번 주소에 들어가 있는지를 저장한다. 그리고 esp를 arg의 길이 + 1만큼 떨어뜨려놓고 바로 주소를 저장했는데, +1을 한 이유는 \0까지 저장하기 위함이다. NULL pointer sentinel 혹은 argument address, argc 저장 등과 같은 부수적인 코드는 실제 코드를 참조하면 된다. Stack을 쌓는 과정에서는 내장 함수 혹은 라이브러리를 사용하지 않았으며, 파싱을 하기 위해서 strtok을 사용했다.

일련의 과정을 거친 뒤, esp는 return address를 가리키고 있다.

1. User Memory Access

텍스트이(가) 표시된 사진

자동 생성된 설명

위 코드는 syscall.c에 존재한다. 매 번 새로운 argument의 주소가 들어 올 때마다 주소의 유효성을 검증해야 하기에 별도의 check\_address 함수를 설계했다. 위 함수는 사용자가 사용하려는 주소를 받아서 총 3 가지를 검증한다. 우선, page의 가상 메모리에 매핑이 되어 있는 주소인지를 확인한다. 둘째, 메모리 주소가 NULL이 아닌지를 검사하며, 마지막으로 user address인지를 확인한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

각각의 argument는 getArgument에서 가져온다. getArgument는 시작할 주소 esp와 argument를 저장할 arg 배열 및 몇 개의 argument를 가져올지에 대한 정보인 count가 포함되어서 전달된다. 이 때, 각각의 주소를 arg에 넣기 전에 check\_address를 수행하며, 만약 check\_address 과정에서 유효성이 결여되어 있다 판단되면 그 즉시 exit 시스템 콜을 호출하도록 설계하였다.

이를 구현하기 위해서는 몇 가지 내장 라이브러리들을 사용했다. 코드에서 볼 수 있다시피, pagedir\_get\_page와 is\_user\_vaddr을 사용했으며, 각각 매핑과 user 영역의 메모리인지 여부를 검사한다. 자세한 설명은 이전 section에서 진행했으므로, 생략한다.

1. System Calls

시스템 콜을 호출하기 위해서는 시스템 콜 호출 – lib/user/syscall.c에서 시스템 콜 핸들링 – inter\_handler 호출 – syscall\_handler 호출을 통해서 진행된다. syscall\_handler 이전까지는 명세서에 자세히 설명되어 있으므로 설명을 생략한다. syscall.c에서 시스템 콜이 호출되면, 어떤 시스템 콜을 호출할지 고유 번호를 함께 전달하게 된다. syscall\_handler에서는 해당 번호를 토대로 switch 문을 돌려서 어떤 명령을 수행할지를 결정한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

위 코드는 syscall.c에 있는 syscall\_handler의 일부다. syscallNum에는 최초에 system call이 호출될 때 함께 커널에 전달 된 시스템 콜의 고유 번호가 포함되어 있다. 가령 SYS\_HALT와 같이 아무런 인자가 필요 없는 시스템 콜의 경우, 커널 API를 호출하고 마무리 한다. 하지만 SYS\_EXEC와 같은 시스템 콜은 반드시 argument를 수반한다. 이를 받기 위해서 우선 getArgument를 호출한다. getArgument에서 어떤 일을 진행하는지는 위에서 설명했다. Argument를 가져 온 뒤에는 필요한 kernel API를 호출하는데, 이 때 SYS\_EXEC에서는 exec 시스템 콜을 호출한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

exec에서는 process execute 함수를 호출하는데, 이 때, exec에서는 child process를 생성하고, sema\_down(load)를 기다린다. 즉, child process가 메모리에 올라갈 때 까지 기다리는 세마포어를 사용해서, 프로세스의 순서를 맞추는 동기화 작업을 진행했다. 그리고 child process의 생성이 완료되면, child pid를 리턴하는 방식으로 제작했다. 한편, child process pid가 exec에서 return 되면, syscall\_handler에서는 child process pid를 받게 된다. 이 값은 eax에 저장되어서, 다시 user에게 전달된다.

그 밖의 시스템 콜 역시 exec와 비슷한 lifecycle을 토대로 작동하며, 구체적인 작동 방식은 코드를 참조하면 된다.

1. Additional System calls

추가적으로 과제에서 요구하는 system call을 구현하고자 우선 lib/user/syscall.c를 수정하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

앞서 구현된 syscall을 참조해서 작성했다. 이 함수의 경우 max of four int에서 4개의 인수를 받아야 하기 때문에 만들게 되었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Fibonacci는 1개의 인수만 전달되므로 syscall1을 사용했고, max\_of\_four\_int에서는 syscall4를 사용했다. 한편, lib/syscall-nr.h에는 각각의 사용자 정의 시스템 콜을 USR\_XXX의 형식으로 등록했다. 자세한 사항은 코드를 참조하면 된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

앞서 설명한 시스템 콜의 lifecycle을 수행한 뒤, switch 문에 들어와서 각각의 시스템 콜이 위의 코드를 수행한다. syscall.c 코드 내에 fibonacci와 max\_of\_four\_int 함수가 적혀 있으니 참조하면 된다. getArgument와 eax에 리턴값을 저장하는 부분은 앞서 설명 완료했기에 생략한다.

* 1. **시험 및 평가 내용**

