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

담당 교수 : 김영재

조 / 조원 : 이도안

개발 기간 : 2024. 09. 12. ~ 2024. 10. 04.

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

1. User program Project(1) 에서는 함수 호출에 따른 Argument Passing을 구현하여, 커널 영역의 스택에 직접 push하고 이를 확인할 것이다. 또한 사용자 메모리 접근의 유효성을 검증하고, 시스템 콜을 요청 받았을 때 이것을 커널에서 직접 처리하는 것을 구현하여 커널과 사용자 프로그램 영역의 의사소통 개념을 이해하고 운영체제에서의 처리과정을 학습, 확장하는 것을 목표로 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

사용자의 프로그램 입력을 argument로 받아 Parsing한 후, 이를 커널에 정확하게 전달하는 과정을 수행함.

1. User Memory Access

때때로 프로그램이 유효하지 않은 데이터 영역에 접근하려고 할 때 이를 커버하는 과정이 필요함. 그것을 방지하는 과정이고 유효하지 않은 메모리 접근 시 해당 프로그램을 exit하게 됨.

1. System Calls

사용자 프로그램이 exit, halt, wait 등 커널의 함수들을 사용할 수 있도록 시스템 콜을 구현한다. 그러한 시스템 콜을 받았을 때의 수행을 syscall.c 파일에서 처리한다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정 설명

유저 프로그램 실행 시 argument는 space를 기준으로 parsing한다. 잘라진 arguments는 차례대로 argv[0] … 으로 저장되고, 이를 카운트하는 argc 변수도 저장한다. 그리고 이 arguments는 user stack에 차례대로 쌓이게 된다.

이를 위해서 stack을 초기화하고 사용할 수 있게 해주는 setup\_stack 함수를 호출해준다.

텍스트, 영수증, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

위 그림과 같이, 각 arguments 문자열을 스택에 push하고, word-align(padding)과 null ptr를 각각 push 이후 각 arguments가 저장된 스택 포인터를 push해준다. 마지막으로 argv가 시작되는 스택 주소, argc, return address를 push한다. 이때 push하는 항목의 size를 잘 고려하여, push 전 esp를 감소시켜 stack의 사용 공간을 잘 확보한다.

* User Memory Access
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명
  + Invalid memory access를 어떻게 막을 것인지 설명

Invalid memory access는 프로세스가 스스로에게 할당된 유효한 메모리 영역인 user space를 벗어난 페이지에 접근하려고 할 때 발생한다. 포인터가 가리키는 주소가 유효하지 않거나, 부적절한 외부의 영역(커널)에 접근하려고 할 때 발생하기도 한다.

이를 막기 위해서 syscall 처리 시 받은 argument의 주소가 유효한 것인지 검증해야 한다. 이를 위해 제공된 함수인 is\_user\_vaddr 함수를 응용하여 처리하려고 한다. 만약 타당하지 않은 주소를 참조하려고 한다면, 이 함수가 false를 반환하게 되고, 해당 프로그램을 종료하여 Invalid memory access를 막을 수 있을 것이다. 또한 접근하려는 메모리의 유효성 검증을 위해서 pagedir\_get\_page() 함수도 호출하여 둘 중 하나라도 false가 반환되면 프로그램을 종료하면 되겠다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

사용자 프로그램은 직접 하드웨어에 접근하거나 커널의 기능을 사용할 수 없기 때문에 system call의 도움을 받아 kernel과 소통하여 IO와 같은 기능을 사용하거나 하드웨어에 접근할 수 있다. 메모리 안정성을 위해 분리된 kernel 과 user memory 간 서로 침범하지 않지만 시스템 콜을 사용하여 kernel에 접근할 수 있게 한다.

* + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명 (하나의 시스템 콜 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)

SYS\_HALT

Pintos를 종료하기 위한 시스템 콜로, sys\_halt 함수에서 shutdown\_power\_off() 를 호출해 시스템을 종료한다.

SYS\_EXIT

User 프로그램을 종료하고 프로그램의 status를 커널에 반환하는 시스템 콜로, sys\_exit 함수가 thread\_exit()를 호출하여 실행된다.

SYS\_EXEC

자식 스레드를 생성하여 프로그램을 실행하는 시스템 콜로, sys\_exec 함수에서 process\_execute()를 사용해 구현된다.

SYS\_WAIT

부모 스레드가 자식 스레드보다 먼저 종료되지 않도록 대기하는 시스템 콜이며, sys\_wait 함수에서 process\_wait()를 호출해 작동한다.

SYS\_READ

파일에서 데이터를 읽는 시스템 콜로, STDIN만 처리하며 sys\_read 함수가 input\_getc()를 호출해 입력을 받는다.

SYS\_WRITE

파일에 데이터를 출력하는 시스템 콜로, STDOUT만 처리하며 sys\_write 함수가 putbuf()를 호출해 데이터를 출력한다.

SYS\_FIBONACCI

주어진 숫자의 피보나치 수를 계산하는 시스템 콜로, sys\_fibonacci 함수에서 재귀 대신 for문을 사용해 값을 계산한다. 입력받은 수 N의 피보나치 수열 값을 계산하여 반환한다.

SYS\_MAX\_OF\_FOUR\_INT

네 개의 정수 중 최댓값을 반환하는 시스템 콜로, sys\_max\_of\_four\_int 함수에서 처리한다.

입력받은 4개의 arguments중 가장 큰 값을 반환한다.

* + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

예를 들어 유저가 exit 시스템 콜 API를 호출한다면, argument passing을 동반한 매크로인 syscall1이 실행된다. 이러한 함수는 syscall number와 arguments를 동반하며, intr\_frame에서 이것들을 받아들이고, 커널은 이를 감지하여 syscall\_handler를 실행하게 된다. Syscall\_number에 따라서 커널 레벨의 syscall\_handler는 각 함수를 적절히 수행하고 함수의 반환값이 존재하는 경우 f->eax에 저장되며, 유저 레벨로 돌아간다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **II. A.의 개발 범위를 포함하여 구현 내용에 대한 일정 작성**

총 개발 기간: 2024. 09. 12. ~ 2024. 10. 04.

* 9/12 ~ 9/20: Pintos manual과 가이드pdf 를 통해 Pintos 구조 파악
* 9/22 ~ 9/24: Argument Passing, User Memory Access 구현
* 9/25 ~ 9/27: System Call 구현
* 9/28 ~ 9/28: Additional System Call (FIBONACCI, MAX\_OF\_FOUR\_INT) 구현
* 9/29 ~ 10/03: 보고서 작성
  1. **개발 방법**
* **II. B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

1. Argument Passing

[userprog/process.c] 의 bool load 함수에서 처리하게 된다.

Argv 스트링 배열과 argv\_ptr 배열을 각각 선언하고 strtok\_r 함수를 사용하여 공백 단위로 입력받은 arguments를 끊어서 argv 배열에 넣어준다.

Setup\_stack 이후 각 argv 배열의 크기만큼 스택의 공간을 할당하고, push한다. Push 할때마다 각 스택에서의 주소값을 argv\_ptr에 저장한다. Word align을 위한 패딩과 null ptr까지 스택에 push한 후 argv\_ptr를 push, 마지막으로 argv의 주소, argc, return address를 push 하면 Argument passing의 구현이 완료된다.

1. User Memory Access

[userprog/process.c] 의 load 함수에서 실행파일명인 argv[0]을 filesys\_open(argv[0])으로 열었을 때 null값을 반환한다면 잘못된 파일명이므로 exit해준다.

[userprog/syscall.c] 의 check\_addr 함수는 is\_user\_vaddr함수와 pagedir\_get\_page 함수를 사용해 메모리의 적합성을 판단한다.

is\_user\_vaddr(addr) 함수는 addr이 사용자 공간(User Space)에 속하는지 확인한다.

pagedir\_get\_page(thread\_current()->pagedir, addr) 함수는 해당 주소(addr)가 페이지 테이블에 유효한지 확인한다. 이를 통해 kernel panic을 방지할 수 있게 된다.

위 함수들 중 하나라도 false를 반환한다면 exit(-1)로 비정상종료를 함으로서 커널 패닉 또는 시스템 오류를 막을 수 있다.

1. System Call

[userprog/syscall.c] 파일에서 syscall\_handler 함수를 조작한다. User Memory Aceess의 validation이 완료되면 인터럽트 프레임에 전달된 syscall number에 따라 행동을 달리 한다. 각 수행은 위에서 말한 system call의 개발 과정과 같고, 반환된 함수의 return 값은 f->eax 레지스터에 저장된다.

추가적으로 구현한 FIBONACCI, MAX\_OF\_FOUR\_INT system call에 해당하는 system call number는 [lib/syscall-nr.h]에 새로 정의한다.

[lib/user/syscall.c]의 매크로 목록에서 syscall3까지 존재하는데 MAX\_OF\_FOUR\_INT는 인자가 4개이므로 syscall4를 구현해야 한다.

[threads/thread.h]에 추가적으로 semaphore : mem\_lock, child\_lock을 선언하고, 이를 이용해 process.c에서 synchronization 구현을 한다. Thread.c에서 또한 sema\_init을 통해 semaphore를 초기화시켜준다.

1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

스크린샷, 텍스트, 도표이(가) 표시된 사진

자동 생성된 설명

1. User Memory Access

스크린샷, 텍스트, 도표, 디자인이(가) 표시된 사진

자동 생성된 설명

1. System Calls

스크린샷, 텍스트, 도표이(가) 표시된 사진

자동 생성된 설명

* 1. **제작 내용**
* **II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.**
* **구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명.**
* **개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.**

1. Argument Passing

Parsing :

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

유저로부터 입력받은 문자열을 strtok\_r 함수를 응용해 파싱하는 과정이다. 인자의 개수는 몇 개가 들어올 지 몰라 넉넉하게 128개를 상한으로 설정하였다.

문제 및 해결 : argv[argc]에서 각 문자열에 대한 데이터 공간이 할당되지 않아서 segfault가 발생하는 상황에 처했다. Malloc을 통해 각 문자열만큼의 공간을 할당해주었다.

Stack :

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

Stack 파트에서는 파싱한 argv, argc를 이용하여 80x86 convention에 적합하게 stack을 구현한다. 각 argv인자와 그 스택 포인터를 push하는데, 그 사이에 4byte align을 위한 padding과 null ptr가 들어가게 된다.

문제 및 해결 : word align 시 비트 곱 연산을 통하여 // \*esp = (void \*) ((unsigned int)(\*esp) & 0xfffffffc); //word align 과 같이 구현하니, 스택 앞의 내용이 손실되는 상황이 발생하였다. Argv를 스택에 쌓아 내려올 때, cnt를 세어 끝에 padding bytes를 붙이는 것으로 바꾸어 구현하였다.

1. User Memory Access



Check\_addr :

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

syscall.c에 추가한 check\_addr 함수로 user adress 범위 내에 있는지 확인하여 오류를 방지한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

문제 및 해결 :

Is\_user\_vaddr() 함수로만 검증한 결과 kernel panic을 막을 수 없는 상황이 발생했다. 따라서 addr가 가리키는 페이지의 메모리 영역이 타당한지 체크하는 pagedir\_get\_page() 함수 또한 사용해서 memory access를 확인해주어야 했다.

1. System Calls

* **이번 프로젝트에서 개발한 시스템 콜을 구현 관점에서 상세히 서술.**

syscall 내부 함수 개발 :

* void sys\_halt (void){
* shutdown\_power\_off();
* }
* void sys\_exit (int status){
* printf("%s: exit(%d)\n", thread\_name(), status);
* thread\_current()->exit\_status = status;
* thread\_exit();
* }
* pid\_t sys\_exec (const char \*cmdline){
* check\_addr(cmdline);
* return process\_execute(cmdline);
* }
* int sys\_wait (pid\_t pid){
* return process\_wait(pid);
* }
* HALT system call : 내장 함수인 shutdown\_power\_off()를 사용하여 현재 실행 중인 프로그램을 종료시키는 역할을 수행하도록 하였다.
* EXIT system call : thread\_exit()을 실행시켜 exit하기 전에 current thread에 있는 exit status 값을 알맞게 바꿔주고, exit 문구를 syntax에 맞게 출력한다.
* EXEC system call : process\_execute()로 thread를 생성하고 반환값을 전달한다.
* WAIT system call : process\_wait()의 반환값을 전달한다. 인자로 전달한 pid에 해당하는 child process를 설정한 semaphore를 통해 기다리고, semaphore가 up되면 종료 상태를 반환하는 함수이다.

Sys\_read, sys\_write :

int sys\_read (int fd, void \*buffer, unsigned size){

  //lock?

  if (!fd) {

    for (int i = 0; i < size; i++) {

      \*((uint8\_t \*)buffer + i) = input\_getc();

    }

    return size;

  }

  //lock?

  return -1;

}

int sys\_write (int fd, void \*buffer, unsigned size){

  //lock?

  check\_addr(buffer);

  check\_addr(buffer + size);

  if (fd == 1) {

    putbuf(buffer, size);

    return size;

  }

  //lock?

  return -1;

}

이 프로젝트에서 Read와 write는 standard I/O만 구현하였다.

* READ system call : file descriptor (fd)가 STDIN(0)인 경우 한 문자씩 입력을 받는다.
* WRITE system call : file descriptor (fd)가 STDOUT(1)인 경우 한 문자씩 putbuf()를 통해 출력한다.

Syscall\_handler:

static void

syscall\_handler (struct intr\_frame \*f UNUSED)

{

  // printf ("system call!\n");

  //2

  //intr\_frame에 esp 멤버가 있고 이를 통해 스택에 접근이 가능함.

  //Arguments 있으면?? 그것들은 esp+4이렇게감.

  //!pagedir\_get\_page 해야하나? O

  check\_addr(f->esp);

  int syscall\_num = \*(int\*) f->esp; //\*(int\*) f->esp는 syscall\_number임.

  // //3 syscall handler

  switch(syscall\_num){

    case SYS\_HALT:

      shutdown\_power\_off();

      break;

    case SYS\_EXIT:

      check\_addr(f->esp + 4);

      sys\_exit(\*(int\*)(f->esp+4));

      break;

    case SYS\_EXEC:

      check\_addr(f->esp + 4);

      //create child process

      f->eax = sys\_exec(\*(uint32\_t \*)(f->esp+4));

      break;

    case SYS\_WAIT:

      check\_addr(f->esp + 4);

      //wait child process until process finishes its work.

      f->eax = sys\_wait(\*(uint32\_t \*)(f->esp + 4));

      break;

    case SYS\_READ:

      check\_addr(f->esp + 4);

      check\_addr(f->esp + 8);

      check\_addr(f->esp + 12);

      f->eax = sys\_read(\*(uint32\_t \*)(f->esp + 4), \*(uint32\_t \*)(f->esp + 8), \*(uint32\_t \*)(f->esp + 12));

      break;

    case SYS\_WRITE:

      check\_addr(f->esp + 4);

      check\_addr(f->esp + 8);

      f->eax = sys\_write(\*(uint32\_t \*)(f->esp + 4), \*(uint32\_t \*)(f->esp + 8), \*(uint32\_t \*)(f->esp + 12));

      break;

    case SYS\_FIBONACCI:

      check\_addr(f->esp + 4);

      f->eax = sys\_fibonacci(\*(int\*)(f->esp + 4));

      break;

    case SYS\_MAX\_OF\_FOUR\_INT:

      check\_addr(f->esp + 4);

      check\_addr(f->esp + 8);

      check\_addr(f->esp + 12);

      check\_addr(f->esp + 16);

      f->eax = sys\_max\_of\_four\_int(\*(uint32\_t \*)(f->esp + 4), \*(uint32\_t \*)(f->esp + 8), \*(uint32\_t \*)(f->esp + 12), \*(uint32\_t \*)(f->esp + 16));

      break;

  }

}

메모리 접근 검증이 완료되면 Syscall\_num를 f->eax로부터 받아들여 해당하는 num에 대하여 syscall을 수행한다. 해당 num이 존재하지 않으면 아무런 수행 없이 종료된다. 추가적으로, 인자가 추가되는 system call에 대해서는 각 주소값에 대해 check\_addr를 수행하고 이를 통해 메모리 접근 오류를 방지하게 하였다.

1. Additional System calls

* **새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

MAX\_OF\_FOUR\_INT에서는 총 4개의 인자를 이용하여 연산하므로 추가적인 syscall4 매크로의 선언이 필수적이었다. 다른 syscall0~3의 서식과 동일하게 작성해주었다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Syscall을 부르는 새로운 API함수들을 선언해주었다.

int sys\_fibonacci(int n){

  int prev,now;

  prev=1;

  now=1;

  int tmp;

  if(n==0) return 0;

  n-=2;

  while(n--){

    tmp=now;

    now=prev+now;

    prev=tmp;

  }

  return now;

}

int sys\_max\_of\_four\_int(int a,int b,int c,int d){

  int m1,m2;

  m1 = a > b ? a : b;

  m2 = c > d ? c : d;

  return m1 > m2 ? m1 : m2;

}

Syscall.c 에서 system call number로 위에 상응하는 값을 받았을 때 수행할 syscall 내부 함수이다. 각각 n에 해당하는 fiboncci 수열의 값과 4개의 int값 중 최대값을 찾아 반환하는 함수의 기능을 하게 구현하였다.

case SYS\_FIBONACCI:

      check\_addr(f->esp + 4);

      f->eax = sys\_fibonacci(\*(int\*)(f->esp + 4));

      break;

    case SYS\_MAX\_OF\_FOUR\_INT:

      check\_addr(f->esp + 4);

      check\_addr(f->esp + 8);

      check\_addr(f->esp + 12);

      check\_addr(f->esp + 16);

      f->eax = sys\_max\_of\_four\_int(\*(uint32\_t \*)(f->esp + 4), \*(uint32\_t \*)(f->esp + 8), \*(uint32\_t \*)(f->esp + 12), \*(uint32\_t \*)(f->esp + 16));

      break;

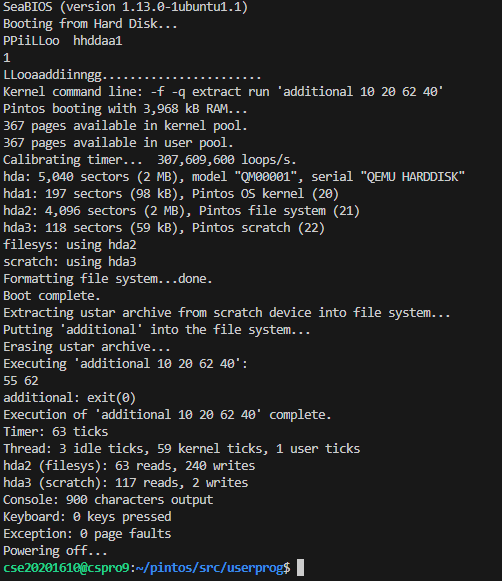
Syscall.c 에서 system call number로 위에 상응하는 값을 받았을 때 수행하게 하는 switch문의 일부이다. 각 number는 아래와 같이 syscall-nr.h에 추가로 선언해주었다.

    /\*USERFROG\*/

    SYS\_FIBONACCI,

    SYS\_MAX\_OF\_FOUR\_INT

* 1. **시험 및 평가 내용**
* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부.**

****

**예상 결과인 55 62로 출력되며 additional system call이 잘 수행되었으며 Powering off… 로 핀토스가 잘 종료됨을 볼 수 있다.**