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

담당 교수 :박성용

조 / 조원 :박성현

개발 기간 :2022/09/20 ~ 2022/09/30

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**
* 기본적인 system call과 additional system call을 구현한다. 기본적인 system call은 halt, exit, exec, wait, read, write이며 additional system call은 fibonacci와 max\_of\_four\_int이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

User stack에 인자로 받은 argument들이 뒤에서부터 들어간다. Argument가 들어간 후, 4byte align을 해주고 NULL값을 넣는다. 그리고 앞에서 넣은 인자들의 주소값과 이 주소들의 시작주소를 넣고 argument의 개수 argc와 return address가 차례로 들어간다.

1. User Memory Access

주어진 스택의 stack pointer가 들어왔을 때, user 영역인지 kernel 영역인지 판단해 kernel 영역이면 exit(-1)과 함께 종료된다.

1. System Calls

Additional을 포함해 총 8개의 system call을 구현한다. Halt는 pintos 시스템 종료, exit은 실행중인 프로세스 종료, exec은 프로그램 실행, wait은 child가 종료될 때까지 parent가 기다릴 수 있도록 한다. 또한, read는 stdin에서 읽어오고, write는 stdout으로 쓴다. 추가적인 fibonacci와 max\_of\_four\_int는 인자로 받은 4개의 숫자를 사용하여 피보나치와 가장 큰 숫자를 찾아 출력한다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정 설명
    - Command를 받았을 때, start\_process함수를 거쳐 load함수로 가게 된다. 여기서 stack을 위한 virtual memory를 생성하기 위해 pagedir\_create가 실행된다. 이후, command에서 앞의 명령어만 parsing한 후, filesystem에서 명령어를 찾아 불러온다. 그 후, make\_stack이라는 사용자 정의함수로 이동해 stack을 쌓는다. 가장먼저, space로 parsing을 하고 argument의 개수만큼 문자열 배열을 동적할당한다. 그리고 문자열을 담을 배열 char \*\*buf에 하나씩 담는다. Stack pointer esp를 buf의 크기+1만큼 감소시키면서 자른 문자열을 하나씩 뒤에서부터 넣는다. +1은 자른 문자열의 뒤에 항상 \0이 들어가야 하기 때문이다. Stack에 값을 넣으면서 문자열이 들어간 stack의 주소도 address배열에 저장하고 총 길이도 count해 total\_len에 저장한다. 값을 다 넣은 후, 4 byte align을 위해 total\_len을 사용해 4의 배수로 맞춰서 esp값을 감소시킨다. 그 후, 4만큼 한번 더 줄이고 NULL 값을 넣어준다. 아까 저장했던 address배열을 뒤에서부터 stack에 넣는다. 문자열의 시작주소는 마지막으로 주소를 넣었던 stack의 주소 값을 넣으면 되므로 esp+4의 값을 넣는다. 그리고 문자열의 개수 argc를 넣어주고 마지막으로 0, 즉 return address를 넣는다. buf배열을 free 시킨 후, make\_stack함수는 종료된다.
* User Memory Access
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명
    - Virtual memory는 User memory와 kernel memory로 나누어져 있다. User level에서 kernel 영역을 access하게되면 시스템이나 다른 process에 damage를 줄 수 있기 때문에 이것을 방지해야 한다. 이것이 invalid memory access이다. Esp의 값이 top of the stack, 즉 PHYS\_BASE의 값보다 크게 되면 kernel memory라 생각한다.
  + Invalid memory access를 어떻게 막을 것인지 설명
    - Threads/vaddr에 있는 is\_user\_vaddr 함수를 사용한다. 이 함수는 esp의 값을 받아서 PHYS\_BASE의 값과 비교한다. 이 값은 top of the stack, 즉 kernel 메모리의 시작점이므로 이 값보다 작은 값이 들어오면 user memory access라 판단한다. Syscall.c에서 system call 함수들을 수행하기 전에, 넘겨주는 esp의 값들을 하나하나 체크해서 user memory가 아니면 exit(-1)을 실행한다. 또한 exception.c의 page\_fault 함수에서 이미 구현되어있는 user, write, 그리고 not\_present변수를 사용해 kernel 영역인지 판단해 종료시키는 조건문도 추가한다.
* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명
    - System call은 user 영역에서 kernel 영역의 기능이 필요할 때 사용한다. User level에서 system call을 통해 kernel 영역으로 접근해 kernel 영역의 기능들을 사용할 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)
    - Halt - pintos프로그램을 종료시킨다.
    - Exit – 현재 실행중인 process를 종료시킨다.
    - Exec – 명령어와 그 인자들로 새로운 프로세스에서 그 명령어에 해당하는 작업을 실행한다. 실패하면 -1을 return한다.
    - Wait – child process가 끝날 때까지 parent process가 기다리도록 한다. 또한 list에 저장되어 있던 child의 정보들을 제거한다.
    - Read - stdin으로부터 값을 읽어온다.
    - Write - stdout으로 값을 출력한다.
    - Fibo - 4개의 입력 받은 숫자 중, 첫번째 숫자에 대한 Fibonacci number을 출력한다.
    - Max\_four - 4개의 입력 받은 숫자 중, 가장 큰 숫자를 return한다.
  + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명
    - User program에서 system call이 필요한 함수를 호출하면 가장 먼저 C library에 있는 함수를 참조하게 된다. 여기서 이 함수는 interrupt를 발생시키는 instruction(0x30)를 macro로 선언되어 있는 syscall# 로 호출하게 된다. 여기서 syscall은 받은 인자의 개수로 나뉘어진다. 여기서 parameter들을 user stack에 넣는다. 그 후, 이 명령이 어떤 명령인지 syscall-nr-h에 선언되어 있는 enum을 통해 system call number를 kernel에 전달하게 된다. kernel에서는 interrupt vector을 통해 system call handler로 가게 되고 여기서 내부적으로 구현되어 있는 kernel의 함수를 실행한다. 실행 후, return값을 stack의 eax register에다 저장한 후, user level로 돌아가게 된다.

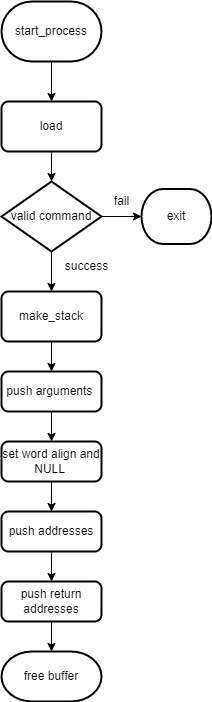
1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **II. A.의 개발 범위를 포함하여 구현 내용에 대한 일정 작성**
  + 9/20~9/23: Argument passing
  + 9/23: User memory access
  + 9/23 ~9/30: System call and additional system call
  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)**
  + Argument passing
    - Process.c
      * Make\_stack 함수를 통한 stack에 parameter 넣어주기.
      * Process\_execute에서 명령어 parsing 및 검사
      * Load에서 명령어 검사
  + User memory access
    - Syscall.c에서 is\_user\_add함수를 통해 esp값 검사.
    - Exception.c의 page\_fault에서 미리 선언된 user, not\_present변수를 통한 invalid memory access 검사.
  + System call
    - Syscall.c
      * Syscall\_handler에서 switch-case문을 통한 각 명령어에 대한 함수호출
      * 각 명령어에 대한 함수 구현 및 return 값 eax에 저장
    - Thread.h
      * Thread 구조체에 자식의 종료를 기다리는 child\_lock semaphore, 부모에서 list\_remove를 기다리는 free\_lock semaphore, 자식의 정보를 담은 child\_elem list\_elem, 자식 process들을 담고있는 child list, 그리고 exit\_stat 추가.
    - Thread.c
      * Thread\_create에 새로 만들어진 process의 정보를 child list에 넣어주기
      * Init\_thread에서 child list를 만들고, semaphore set및, 초기화.
  + Process.c
    - Process\_wait에서 자식의 pid를 받아오고 종료를 기다리는 semaphore설정.
    - Struct thread \*get\_child함수 구현을 통한 자식의 pid 반환
    - Process\_exit에서 프로세스 종료 시, sema\_up을 해주고 list\_remove를 위한 semaphore set.
  + Additional 구현을 위한
    - Additional.c 생성
    - Syscall.nr.h에 새로운 system call number 추가.
    - Lib/user/syscall.c에 syscall4 추가 및 새로운 function 정의.
    - Lib/user/syscall.h 에 새로운 function 선언
    - Syscall.h에 동작 구현에 대한 함수 선언

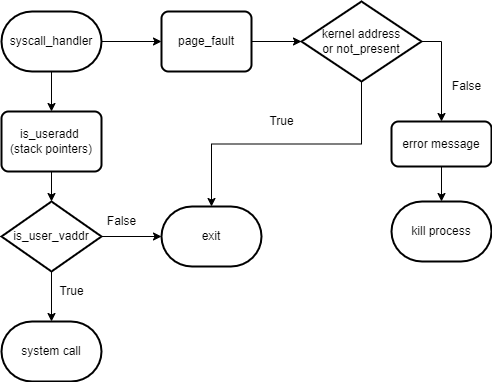
1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

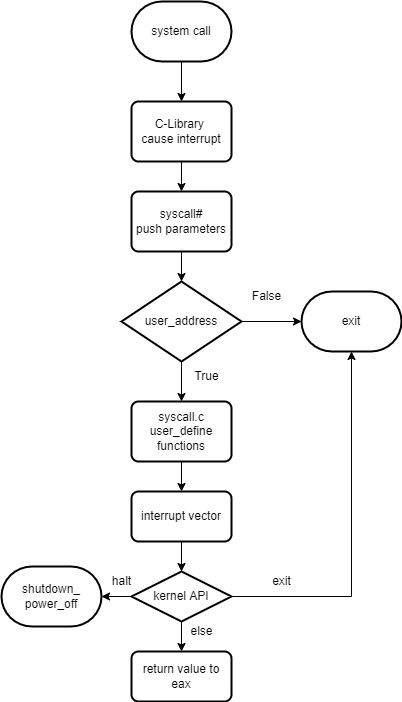
1. Argument Passing



1. User Memory Access



1. System Calls

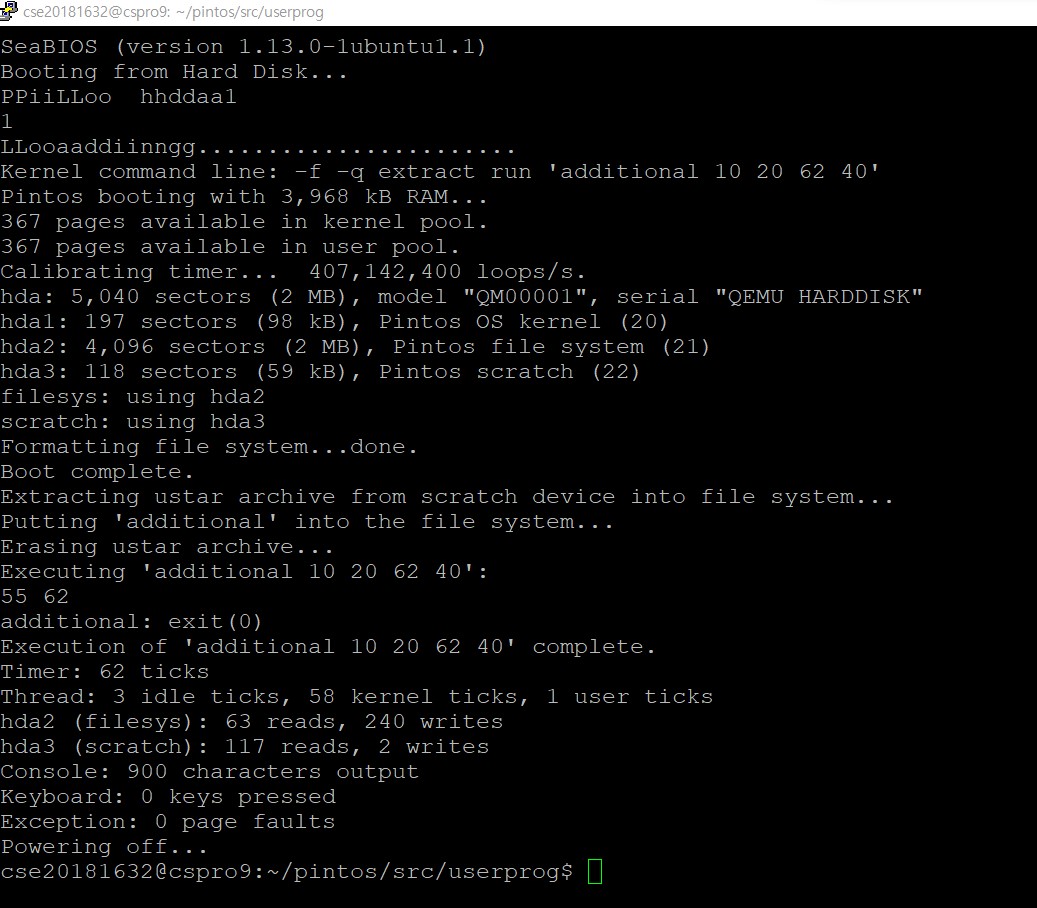


* 1. **제작 내용**
* **II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.**
* **구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명.**
* **개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.**

1. Argument Passing
   * 처음 프로그램이 시작되고 process가 실행되면 start\_process 함수가 실행된다. 여기서 받은 명령어와 인자들을 함께 load함수로 넘겨준다. load함수에서 가장 먼저 받은 인자들과 명령어들을 parsing한다. Parsing의 경우 띄어쓰기로 한다. 그 후, 명령어만 filesystem에 넣어 유효한 명령어인지 확인한다. 그 후, stack을 위한 page를 할당하고 make\_stack이라는 사용자 정의 함수로 받은 문자열과 명령어들과 stack pointer esp를 넘겨 stack을 구성한다. Make\_stack함수에서는 가장 먼저 filename을 parsing한다. 명령어와 문자열을 띄어쓰기로 parsing하면서 buf 배열에 넣어준다. Parsing된 문자열의 마지막에 ‘\0’문자도 같이 넣기 위해 길이를 하나 늘려 넣는다. 문자열을 넣으면서 address라는 배열에 주소값도 같이 넣어준다. 그 후, stack에 esp의 값을 4씩 줄이며 buf문자열 배열의 끝부터 하나씩 넣어준다. 다 넣어준 후, 4 byte align을 위해 총 문자열의 길이가 4의 배수가 되도록 esp의 값을 줄여준다. NULL값을 넣기 위해 0을 넣어준 후, 아까 저장해 두었던 address 배열의 끝부터 stack에 넣어준다. 그 후, 문자열 배열의 시작 주소를 넣어줘야 한다. 그 주소는 바로 위 스택의 주소, 즉 esp+4의 주소값이다. 이 주소값을 넣어준 후, parameter의 개수와 return address 0을 넣어준다. 마지막으로 buf배열을 free시키고 make\_stack함수는 종료된다.
2. User Memory Access
   * User memory access를 판별하는 법은 간단하다. Threads/vaddr.h에 있는 is\_user\_vaddr 함수를 사용하면 된다. 편의를 위해 syscall.c에 is\_useradd라는 사용자 정의 함수를 만들었다. 이 함수는 stack pointer를 받아 is\_user\_vaddr을 통해 user stack의 주소면 True, False면 exit(-1)를 수행하게 했다. Syscall.c 의 syscall\_handler함수는 esp로 switch-case문으로 이루어져 있다. 각 system call을 호출하기전, system call에 인자로 넘어가는 esp를 is\_useradd함수를 통해 모두 검사한다. 또한, exeption.c의 page\_fault함수에 user과 not\_present라는 변수가 선언되어 있다. User은 user영역을 의미하고 not\_present는 stack\_page가 존재하지 않는다는 걸 알려주는 변수이다. Page\_fault가 발생했을 때, 그 주소가 user이아니거나 존재하지 않는 stack의 주소라면 exit(-1)을 호출하는 조건문도 추가해주었다.
3. System Calls

* **이번 프로젝트에서 개발한 시스템 콜을 구현 관점에서 상세히 서술.**
* **Syscall.c 함수에서 syscall\_handler함수는 f->esp, 즉 stack pointer을 가지고 switch-case문으로 구성되어 있다. 각 system call에 대해 가독성을 위해 따로 함수를 구현하였다. 구현당시, system call number은 f->esp값에 저장되어 있고, esp값이 4씩 증가하면서 parameter들이 저장되어 있는 걸 직접 출력을 통해 알 수 있었다. 아래는 각 system call에 대한 구현 설명이다.** 
  + SYS\_HALT:
    - Halt 함수를 호출한다. 이 함수는 shutdown\_power\_off라는 함수를 통해 pintos프로그램을 종료시킨다.
  + SYS\_EXIT:
    - 주소 값을 검사 한 후, exit함수를 호출한다. exit함수는 현재 실행중인 process의 이름과 함께 termination message를 출력하고 그 thread의 eixt\_stat를 종료된 exit number로 변경시킨 후, thead\_exit을 통해 종료시킨다.
  + SYS\_EXEC:
    - 주소 값을 검사한 후, exec함수의 return 값을 f->eax에 저장한다. 명령어들을 실행시켜주는 함수이며 함수로 넘어가는 인자는 const char\*이고 반환형은 pid\_t이다.
    - exec함수에서는 process\_execute함수를 호출한다. Process\_execute함수는 process.c에 구현되어 있다. 이 함수에서는 filename을 인자로 받는다. 이 filename의 앞부분, 즉 명령어 부분만 parsing을 해준 후, filesytem에 존재하는지 확인한다. 확인 후, thread\_create를 통해 새로운 process를 만들어준 후, 그 pid를 exec함수로 반환해 준다.
  + SYS\_WAIT:
    - 주소 값을 검사한 후, wait함수로 (f->esp+4)를 넘겨준다. Wait는 부모 process가 자식 process의 종료를 기다리게 하는 system call이다.
    - Wait 함수에서 process.c의 process\_wait을 호출한다.
    - Process\_wait을 구현하기 전에, thread.h와 thread.c를 먼저 변경했다. Thread.h의 thread구조체에 child\_lock과 free\_lock이라는 semaphore을 추가했다. Child\_lock은 말 그대로 child의 종료를 기다리는 semaphore이고 free\_lock은 list의 메모리 제거를 위한 semaphore이다. 그리고 list에 들어갈 element를 저장한 child\_elem, 자식 process를 저장할 child list를 추가한다. 마지막으로 종료에 대한 정보를 담은 exit\_Stat이라는 변수도 추가한다. Thread.c에서는 thread\_create라는 함수가 process가 생성되면 가장 먼저 수행된다. 여기서 init\_thread라는 함수가 호출된다. 이 함수에서 child의 list를 생성하는 list\_init, 두개의 semaphore을 0으로 set하는 sema\_init을 구현했다. Thread\_create함수로 돌아온 후, list\_push\_back함수를 통해 list에 현재 thread의 child에 대한 정보 child\_elem을 넣어준다.
    - Process\_exit에도 몇가지 코드를 추가했다. Process를 종료할 때, 현재 thread의 child\_lock을 sema\_up을 통해 풀어 부모 프로세스가 알 수 있게 한다. 하지만 부모 프로세스가 list에 있는 child의 정보들을 제거하기 전 까지 child는 사라지면 안되므로 free\_lock을 sema\_down을 통해 잠궈준다.
    - Process\_Wait에서는 가장먼저 get\_child라는 사용자 정의 함수를 통해 자신의 자식 프로세스의 pid를 받아온다. Get\_child 함수는 넘겨받은 pid를 list안에 있는 thread를 돌며 비교해 일치하는 pid를 가진 thread를 반환한다. 만약 일치하는 자식이 존재하지 않으면 -1을 반환한다. 만약 자식이 존재한다면 child\_lock을 sema\_down 시켜줌으로써 child가 종료될 때 까지 기다린다. 종료되면 child의 exit\_stat을 변경시켜주고 list에서 child에 대한 값들을 list\_remove를 통해 모두 제거시킨다. 그 후, free\_lock을 sema\_up을 통해 풀어준다.
  + SYS\_READ:
    - Read는 인자로 3개를 받는다. Esp+4는 fd의 값, esp+8은 buffer의 주소, 그리고 esp+12는 크기이다. 이 3개의 esp를 모두 검사해 준 뒤, read 함수의 반환 갑을 f->eax에 저장한다.
    - Read 함수는 fd가 0일때만 실행된다. 아닐 경우, -1을 반환한다. Input\_getc()함수를 사용해서 NULL값이 나올 때까지 bytes\_read를 증가시킨다. 그 후, 총 읽은 byte수, 즉 bytes\_read를 반환한다.
  + SYS\_WRITE:
    - write함수 역시 read와 마찬가지로 3개의 인자를 받는다. 넘겨받는 인자역시 동일하다.
    - Write 함수내에서는 fd가 1, 즉 stdout일때만 작동한다. 넘겨받은 size만큼 putbuf함수를 사용해 출력한다. 그 후, 출력한 size를 return한다. 에러시, -1을 반환한다.

1. Additional System calls

* **새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술**
* Userprog/syscall.c에 int Fibonacci(int n), int max\_of\_four\_int(int a, int b, int c, int d)선언 추가
* Lib/syscall-nr.h에 SYS\_FIBO(13), SYS\_MAX\_FOUR(14) 추가해 system call number 추가
* Lib/user/syscall.c에 syscall4 새롭게 define
  + ARG3을 추가하고 addl $20으로 변경
  + fibonacci함수는 syscall1을 반환하고 max\_of\_four\_int는 새롭게 만든 syscall4를 반환하도록 생성
* Lib/user/syscall.h에 int Fibonacci(int n), int max\_of\_four\_int(int a, int b, int c, int d)선언 추가
* Userprog/syscall.c의 syscall\_handler함수에 두가지 case SYS\_FIBO와 SYS\_MAX\_FOUR 추가. SYS\_FIBO는 fibonacci함수를 호출하며 SYS\_MAX\_FOUR는 max\_of\_four\_int함수를 호출한다.
  + 두 함수 다 f->eax에 값을 저장하며 SYS\_MAX\_FOUR같은 경우에는 인자가 4개이므로 esp+4,8,12,16모두 검사한다.
  + int Fibonacci(int n) 함수 정의
    - n이 0이면 0, 1이나 2이면 1을 반환한다. 음수일 경우 -1을 반환하며 나머지의 경우 for문을 돌며 받은 정수만큼 fibonacci 수열을 계산해 값을 반환한다.
  + int max\_of\_four\_int(int a, int b, int c, int d)선언 추가
    - 4개의 숫자 중 가장 큰 숫자 반환.
* Src/examples에 additional.c추가
  + 5개의 index를 가진 argv 문자열 배열을 받는다. 5개가 아닐 경우, EXIT\_FAILURE반환.
  + Argv의 첫번째 인자는 명령어이므로 index 1 부터 atoi함수를 이용해 문자열로부터 숫자를 추출한다. fiboncci함수에는 첫번째 숫자를, max\_of\_four\_int함수에는 4개의 숫자를 모두 넘겨 반환값을 받아 주어진 형식대로 출력한다.
* Examples/makefile 수정
  + PROGS 태그에 additional 추가
  + Additional\_SRC = additional.c 추가
  1. **시험 및 평가 내용**
* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부.**