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

담당 교수 : 박성용 교수님

조 / 조원 : 20181617 김채연

개발 기간 : 2020/10/05-2020/11/1

1. **개발 목표**

* Pintos OS의 환경에서 제공되는 기본 코드를 사용하여 userprogram을 구현한다. argument passing, system call, User Memory Access 구현을 필요로 하며 이때 System call은 halt, exit, exec, read, write, wati와 같은 명령어에 추가적으로 fibonacci와 max of four int도 구현한다.

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

명령어를 통해 입력받은 argument가 tokenizing을 통해 나눠진 뒤 하나씩 stack에 저장된다.

1. User Memory Access

User가 Memory에 Access하는 과정에서 잘못된 주소가 넘겨지는 오류가 발생했을 때 예외처리를 통해 pintos 프로그램을 종료한다.

1. System Calls

System Call 명령어가 입력되었을 때 system call handler를 통해 stack에 쌓여있는 argument를 가져와 각 명령어에 맞게 처리된다.

* 1. **개발 내용**
* Argument Passing

먼저 입력받은 내용을 tokenizing을 한다. 이때 strtok\_r과 strlcpy를 이용하였다. 이렇게 나누어진 aragument들은 esp라는 stack pointer를 이용하여 하나씩 stack에 저장된다. stack은 process.c내에서 먼저 초기화가 된 뒤에 쌓인다. argument가 모두 stack에 저장이 되면 그다음 과정은 word\_align이다. 다음은 word\_align을 포함하여 저장된 stack의 argument들의 주소를 stack에 저장하고, 이후에는 argument의 개수, return address를 차례로 stack에 쌓는다.

* User Memory Access

invalid memory access를 막는다. 이때 invalid memory access란 NULL 포인터와 Unmapped virtual memory, 그리고 kernel address space의 pointer가 주소공간에 access할 때를 말한다. User process는 User memory 공간만 이용할 수 있기 때문에 Kernel memory의 접근을 막아야 하고 각종 에러에서 보호하기 위해 잘못된 주소에 대한 접근을 막아야 한다.

따라서 User process가 address space에 접근하기 전 pagedir\_get\_page 를 통해 Unmapped Virtual memory인지 검사를 하고 is\_user\_vaddr, is\_kernel\_vaddr을 통해 kernel address space의 포인터인지 검사를 한다. 이를 통해 Invalid memory access를 막을 수 있다.

* System Calls

운영체제는 kernel과 user 모두 두가지로 나누어져 있다. 프로그램이 구동될 때 많은 부분이 파일을 읽거나 쓰거나 메시지를 출력하는 등 kernel 모드를 사용하게 된다. System Call은 이러한 kernel 영역의 기능을 user 모드에서 사용가능하게 한다. 즉 프로세스가 하드웨어에 직접 접근해서 필요한 기능을 사용할 수 있게 하기 때문에 필요하다.

아래는 이번 프로젝트에서 개발한 system call 명령어이다.

* + 1. halt : shutdown\_power\_off()를 이용하여 Pintos를 종료한다.
    2. exit : 현재 userprogram을 종료하고 status를 kernel로 리턴한다. 만약 process의 parent가 wait하고 있다면 이 리턴값을 받는다. 결국 status 0은 성공을 의미하고, 0이 아니면 에러를 의미한다.
    3. exec : cmdline 을 통해 주어진 이름의 process를 실행시킨다. 즉 child process를 만든다. 그리고 새 process의 id를 return 한다.
    4. wait : 주어진 pid의 child process를 기다린다. child thread ID가 유효한지 검사를 하고 child의 exit status를 검색한다. wait 전에 process가 끝나는 것을 방지하기 위해 synchronization을 한다.
    5. read : open file fd를 buffer로 read를 한다. 실제로 읽은 byte의 수를 return한다. 현재는 STDIN을 통해 read한다.
    6. write : open file fd에 buffer의 size bytes만큼 write를 수행한다. 그러고 실제로 쓰인 byte의 수를 return 한다. 현재는 STDOUT을 통해 write한다.
    7. Fibonacci : 주어진 n번째 Fibonacci 수열을 return한다.
    8. max\_of\_four\_int : 주어진 4개의 int 변수에 대해 가장 큰 값을 return한다.

유저 레벨에서 시스템콜이 호출된다면 하드웨어가 유저모드에서 커널모드로 mode bit를 바꾼다. 하드웨어가 sys\_call이라는 커널 내의 trap handler로 가게 되고 지금까지 유저프로그램에서 진행했던 단계는 저장된다. system call number가 sys\_call table에 있는 번호인지 확인 후 해당 번호의 system call function의 주소를 가져와서 작업을 한다. 작업이 끝난다면 다시 시스템 콜을 호출했던 유저 레벨로 돌아가고 mode bit를 유저모드로 전환한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

10/5 ~ 10/12 : Argument passing 구현

10/13 ~ 11/19 : systell call 구현

10/20 ~ 10/26 : UserMemoryAccess구현 및 오류 해결

10/27 ~ 10/29 : 추가구현 및 디버깅

10/30 ~ 11/1 : 보고서 작성

* 1. **개발 방법**

process.h, process.c 에서 argument passing을 구현한다. 먼저 tokenizing을 한다. process\_execute, load에서 주어진 file\_name을 각 argument로 모두 parsing을 한 뒤 배열에 따로 저장을 한다. stack에 쌓는 과정은 setup\_stack이 이루어진 후 한다. construct\_stack 이라는 함수를 새로 만들어서 stack pointer esp를 통해 알맞은 주소 위치에 stack을 쌓는다.

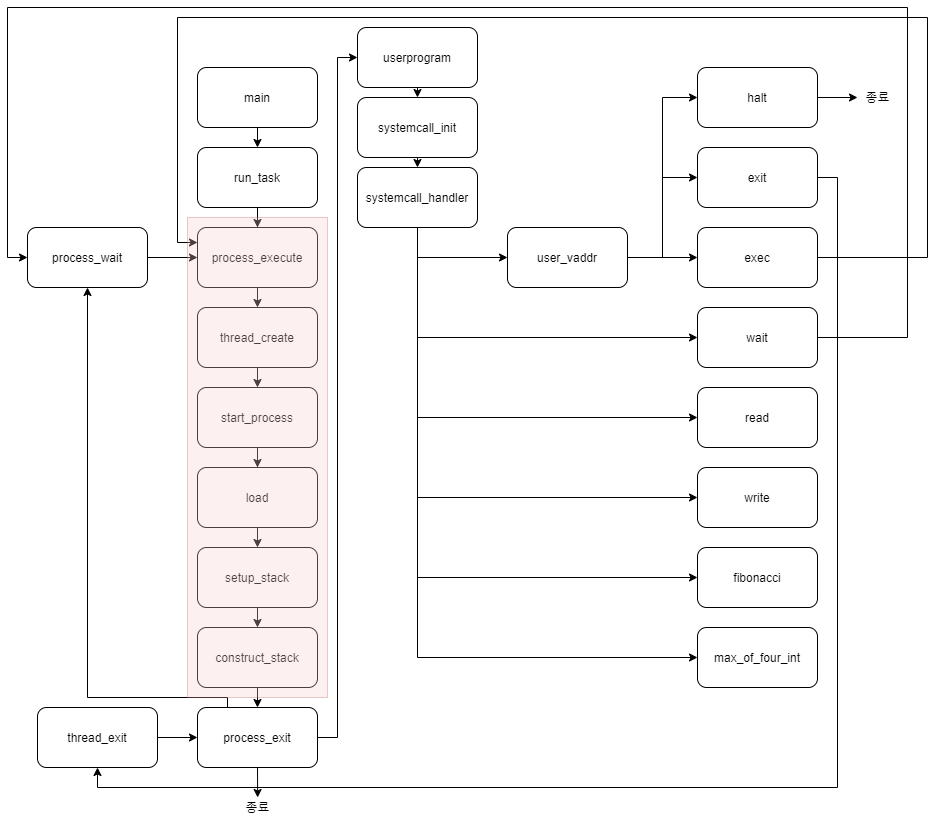
syscall.h, syscall.c 에서 system call을 구현한다. syscall\_handler 함수 내에서 switch문을 이용하여 명령어를 나눈다. 각 명령어에 맞는 함수를 호출하여 system call을 수행한다. 이때 parameter는 esp를 통해 stack에 쌓여있는 argument를 넘겨준다. 각 명령어에 대한 함수는 sys\_(명령어) 함수로 처리를 하였다. return value가 생긴다면 eax에 저장을 한다. system call에 해당하는 값은 syscall\_nr.h에 선언되어 있다.

syscall.c와 process.c, exception.c 에 user memory access에 대한 내용을 추가한다. syscall.c 에서는 exit, exec, wait 명령어에 대해 넘겨주는 값이 valid한지 검사를 한다. 이는 user\_vaddr함수를 새로 추가하여 is\_user\_vaddr 함수를 통해 구현한다. process.c에서는 process\_execute 함수 내부에서 검사를 한다. load에서 access를 검사하는 코드를 가져와서 filesys\_open 함수의 결과가 NULL이라면 에러로 처리한다. exception.c에서는 page\_fault 내부에서 is\_kernel\_vaddr함수를 통해 검사를 진행한다. 이때 발생하는 에러에 대해서는 프로그램을 종료하도록 return -1이나 sys\_exit(-1)을 한다.

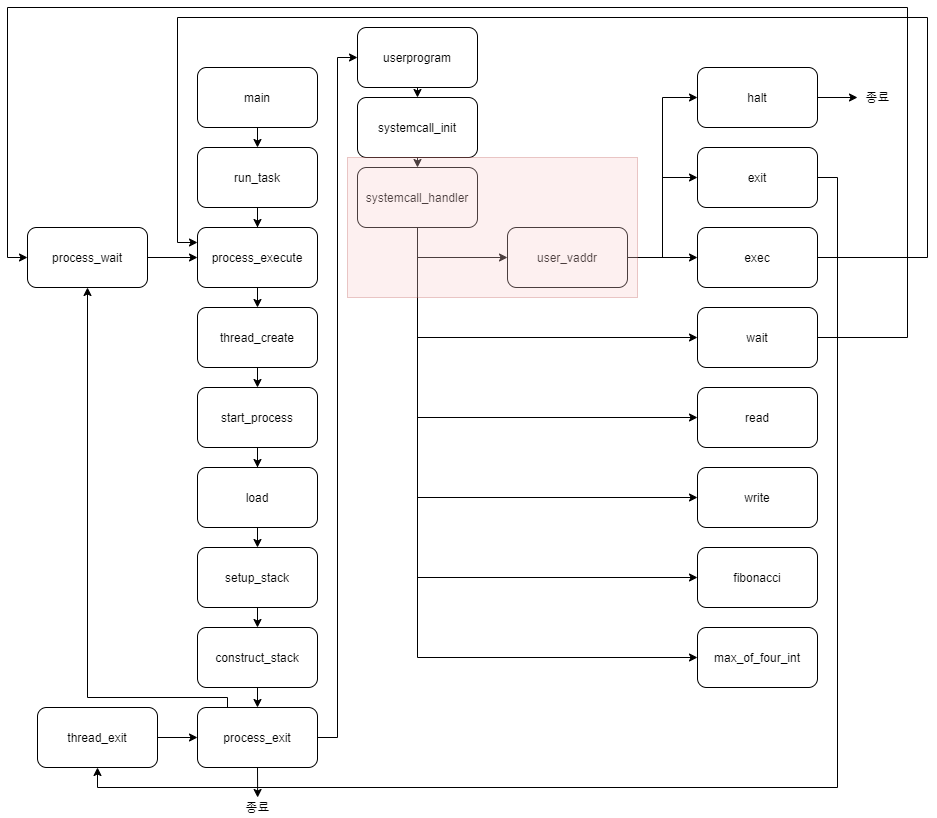
process.c와 thread.h, thread.c 에 child 가 죽을 때까지 parent는 기다려 줘야되기 때문에 synchronize를 위한 내용을 추가한다. thread.h 와 thread.c에 #ifdef USERPROG/ #endif를 이용하여 userprog에서 이용하는 구조체를 만든다. thread.c에서 init\_thread에서 추가된 구조체를 초기화하는 코드를 추가한다. process.c에서는 process wait, process exit 내에 세마포어를 이용하여 child와 parent의 진행 단계를 처리한다. parent는 child의 exit를 기다리도록 하고 child의 memory는 parent가 remove를 할 때까지 남겨두도록 한다.

additional systemcall을 구현하기 위해서는 먼저 lib/user의 syscall\_nr.h, syscall.h, syscall.c 에서 syscall4를 추가로 만들고 새로운 systemcall에 해당하는 값도 추가한다. 그다음 syscall.h와 syscall.h에 해당 system call을 처리하는 함수를 새로 선언, 구현하고 switch문에도 따로 분류를 한다. 그다음 마지막으로 example의 Makefile에 additional을 추가하고 additional.c파일을 새로 만들어서 출력문을 작성한다.

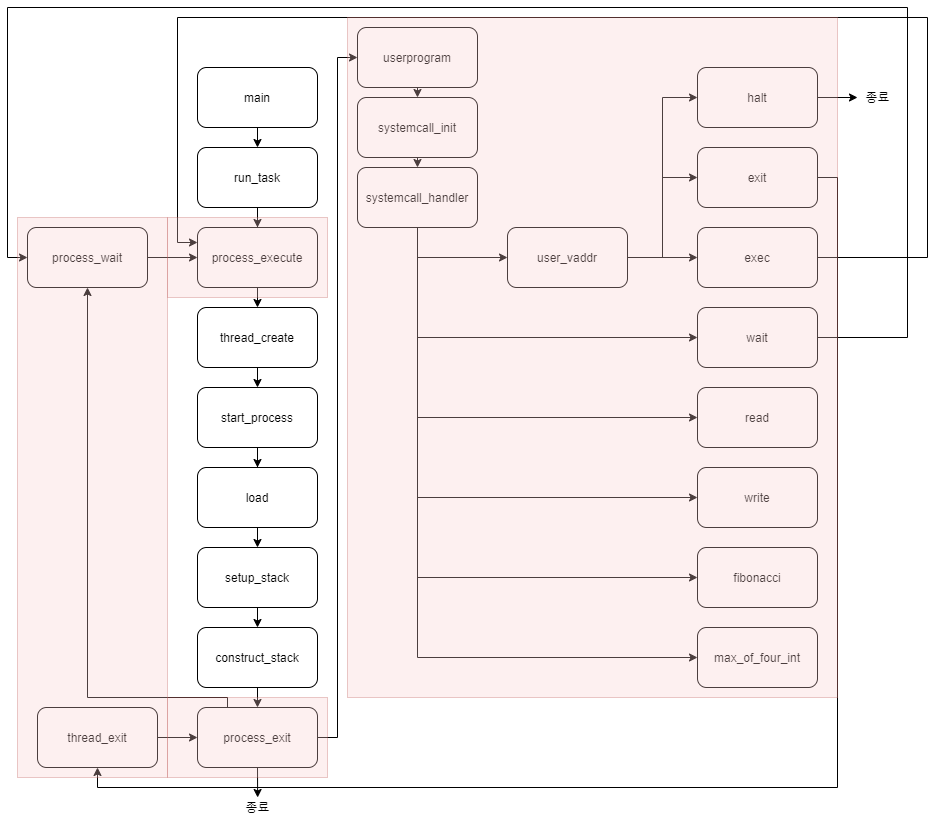
1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing



1. User Memory Acess



1. System Calls

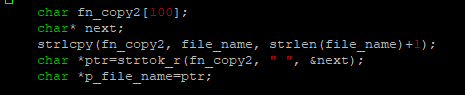


* 1. **제작 내용**

1. Argument Passing
2. userprog/process.c

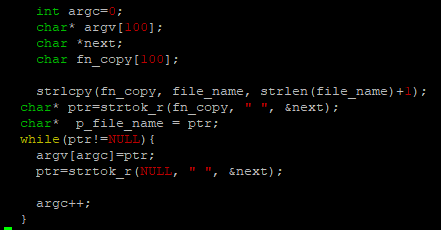
먼저 file\_name의 parsing에 대한 코드이다.

process\_execute 함수 내에 thread\_create함수에 넘겨주는 file\_name을 parsing하여 넘겨준다. strlcpy를 통해 file\_name을 fn\_copy2에 복사후 strtok\_r을 통해 공백을 기준으로 나눈다. 이 나눠진 부분은 p\_file\_name에 저장을 한다. 처음에는 process\_execute에서는 parsing을 하지 않았더니 계속해서 ‘echo x’를 실행하였을 때 echo x가 넘어가서 정상적으로 진행되지 않았다. 그래서 process\_execute 함수에서 parsing을 하고 parsing한 결과로 thread\_create를 하니 echo만 넘어가면서 정상적으로 진행되었다.



thread\_create에서 file\_name으로 넘겨주는 코드를 p\_file\_name으로 수정하였다.

load 함수 내에서도 file\_name을 parsing한다. argument를 저장하는 argv, argument 갯수를 저장하는 argc를 추가로 선언한다. 위에서 한 방법처럼 strlcpy와 strtok\_r의 반복을 통해 공백을 기준으로 나눈다. 나누어진 부분은 첫번째는 p\_file\_name에, 이후는 argv에 순서대로 저장한다. 그리고 argc를 하나씩 추가하여 argument 개수도 센다.

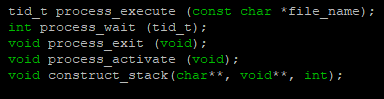


load 내의 file\_name을 다른 함수로 넘겨주느 코드는 모두 p\_file\_name으로 수정하였다. 이후 stack을 쌓는 것은 load 내에서 setup\_stack이 이루어진 후 construct\_stack함수를 선언하여 구현하였다. construct\_stack은 새로 추가한 함수이다.



앞에서 parsing을 하고 저장한 argv와 argc를 이용한다. esp는 setup\_stack을 통해 초기화 된 stack pointer이다. 먼저 argc만큼 argument를 저장한다. 이후 esp가 4의 배수가 되도록 word\_align을 하고 NULL, argv의 address, argv, argc, return address를 차례로 stack에 쌓는다.

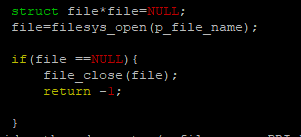
1. userprog/process.h



추가로 선언한 함수인 construct\_stack의 prototype을 추가했다.

1. User Memory Access
2. userprog/process.c

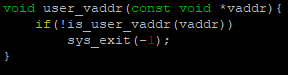
process\_execute 함수 내에서 file\_name에 대한 invalid memory access 처리가 없어서 새로 추가를 했다. parsing되어있는 file\_name인 p\_file\_name을 filesys\_open함수를 통해 return 값을 file에 저장을 한다. 만약 file이 NULL이라면 file을 닫고 프로그램을 종료한다.



이 코드를 thread\_create 하기 전에 추가를 하여 먼저 에러를 처리하도록 한다.

1. userprog/syscall.c

syscall\_handler함수 내에서 명령어를 처리하기 전에 memory접근이 invalid하지 않은지 검사를 한다. 따라서 switch문에서 exit, exec, wait 명령어를 처리하기 전에 user\_vaddr(f->esp+4)이라는 함수를 새로 추가하여 호출해준다. user\_vaddr은 is\_user\_vaddr함수를 이용하여 만약 user memory공간이 아니라면 sys\_exit(-1)을 호출하여 프로그램을 끝낸다.



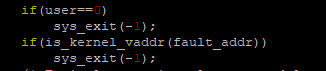
1. userprog/syscall.h



추가로 선언한 함수인 user\_vaddr의 prototype을 추가했다.

1. userprog/exception.c

page\_fault내에서 invalid memory access에 대한 처리를 추가한다. 만약 user가 0이거나 fault\_addr에 대한 kernel memory검사를 해서 kernel이라면 sys\_exit(-1)을 호출하여 종료시킨다. 이때 is\_kernel\_vaddr 함수를 이용하여 fault\_addr이 kernel 공간인지 검사했다.



위의 코드를 page\_fault 내에서 user가 할당된 이후에 추가를 해주었다.

1. System Calls
2. userprog/syscall.c

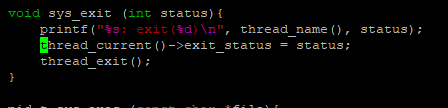
syscall\_handler내에서 switch문을 통해 명령어를 구분한다. 명령어는 uint32\_t를 통해 변환하였다. halt, exit, exec, wait, read, write로 구분하여 각 명령어를 처리하는 함수를 호출했다. 각 필요한 argument는 esp를 통해 stack에 접근하여 넘겨주었다.



halt는 shutdown\_power\_off 함수를 호출하여 처리하였다.



exit는 exit를 출력해준 뒤 현재 thread의 status를 변경해주고 threaada\_exit를 호출하여 처리하였다.



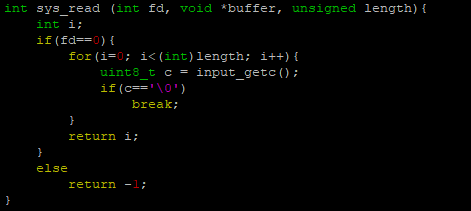
exec는 process\_execute 함수를 리턴하여 처리하였다.



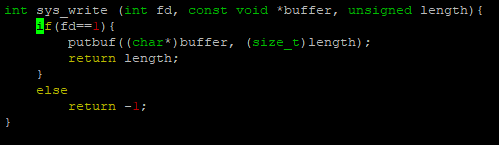
wait는 process\_wait 함수를 리턴하여 처리하였다.



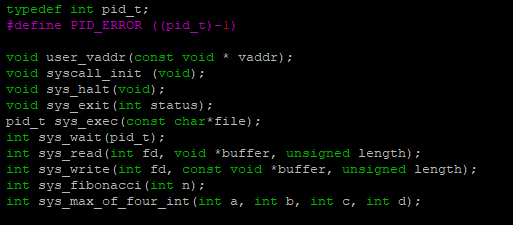
sys\_read는 fd가 0이라면 input\_getc함수를 이용하여 stdin 출력을 해주었다.



sys\_write는 fd가 1이라면 putbuf함수를 이용하여 stdout을 해주었다.



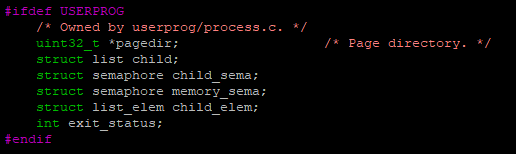
1. userprog/syscall.h



pid\_t에대해서 선언을 해주고 추가로 선언한 함수인 systemcall을 처리하는 함수들의 prototype을 추가했다.

1. threads/thread.h

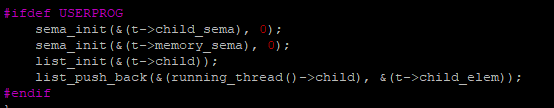
process가 wait, exit 등을 할 때 synchronization을 해주기 위해 thread구조체에 변수를 더 추가했다. pagedir은 page\_directory, child는 child thread, child\_sema 와 memory\_sema는 각각 child 의 상태와 memory에 대한 lock을 수행하는 semaphore 구조체, child\_elem은 chiild의 elem, 그리고 exit\_status를 추가했다.



이 코드를 thread 구조체 내에 추가했다.

1. threads/thread.c

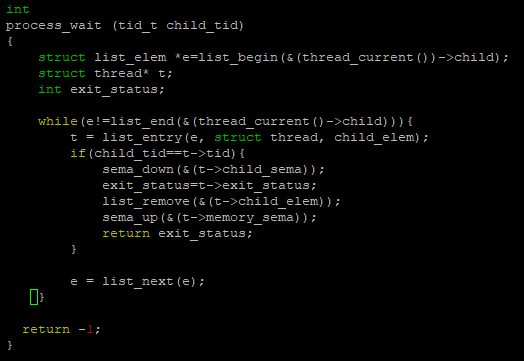
추가된 thread구조체 변수에 대해서 init을 추가로 해주어야 한다. 따라서 init\_thread함수 내부에 child\_sema, memory\_sema, child, child\_elem을 추가로 초기화해준다.



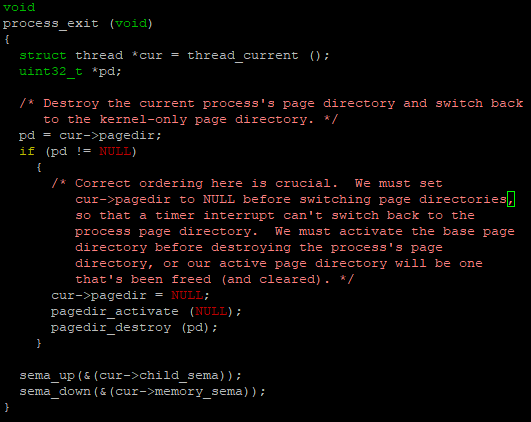
child\_sema와 memory\_sema는 sema\_init함수, child는 list\_init 함수를 이용하였다. child\_elem은 현재 running\_thread의 child에 list\_push\_back을 해주었다.

1. userprog.process.c

process\_wait와 process\_exit에 코드를 추가한다. 먼저 process\_wait 함수내에서는 해당 child\_tid가 죽을 때까지 기다려야 되고 죽으면 thread의 exit\_status를 return한다. 따라서 while문에서 child\_sema 세마포어 down하여 child를 기다린다. 그리고 memory도 lock되어야 하므로 meory\_sema 세마포어 up하여 메모리를 remove하기 전까지 남겨둔다. 처음에는 memory는 lock을 하지 않았더니 list\_remove에 필요한 t가 없어서 계속해서 오류가 발생하였다. 그래서 memory lock을 하는 semaphore를 추가하니 정상적으로 진행되었다.

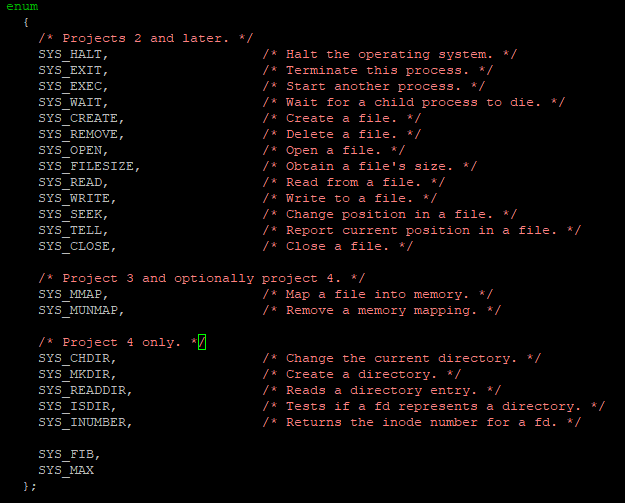


이와 반대로 process\_exit 함수 내에서는 child\_sema와 memory\_sema를 각각 up과 down을 한다.



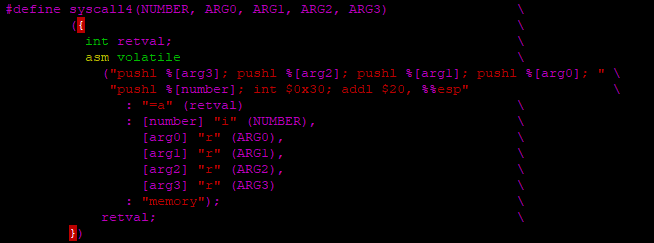
1. Additional System calls
2. lib/syscall\_nr.h

추가 구현할 fibonacci와 max of four int 에 대해서 enum에 값을 추가해준다.

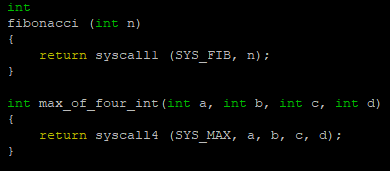


1. user/syscall.c

max of four int는 4개의 argument를 받기 때문에 syscall4를 새로 define해주어야 된다. 따라서 위의 형식에 맞추어 arg3이 하나더 추가된 syscall4를 구현한다.



fibonacci와 max\_of\_four\_int의 함수를 추가한다. return을 각각 syscall1(SYS\_FIB, n), syscall4(SYS\_FIB, a,b,c,d)로 systemcall number와 argument를 넘겨준다.



1. user/syscall.h



추가로 구현한 fibonacci와 max\_of\_four\_int 함수의 prototype을 선언한다.

1. examples/Makefile

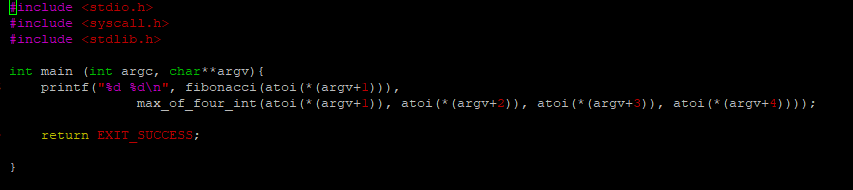
추가 구현한 systemcall에 대한 명령어를 additional로 추가한다.



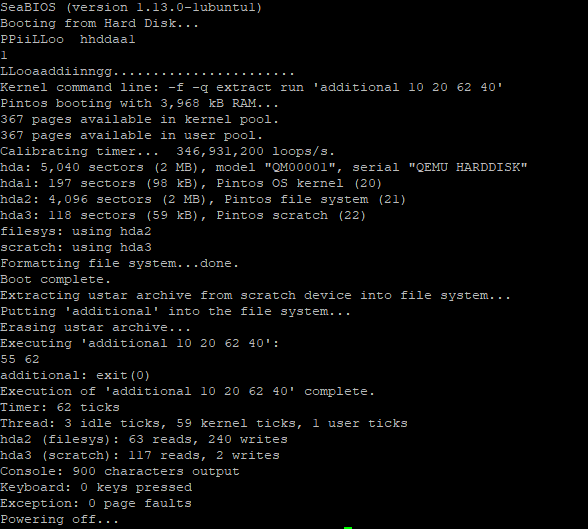


1. examples/additional.c

additional 에 대한 출력을 처리하기 위해 additional.c를 새로 만든다. 메인 함수에서 fibonacci의 결과와 max\_of\_four\_int의 결과를 호출하고 각각 출력한다.



* 1. **시험 및 평가 내용**



fibonacci와 max\_of\_four\_int에 대해서 정상적으로 작동하는 것을 볼 수 있다.