+---------------------------+
| CS 330 |
| PROJECT 2: USER PROGRAMS |
| DESIGN DOCUMENT |
+---------------------------+

---- GROUP ----

>> Fill in the names and email addresses of your group members.

김우진 <hyung1721@kaist.ac.kr>

김준범 <dungeon12345@kaist.ac.kr>

---- PRELIMINARIES ----

# of tokens to use: 0

Contribution

김우진: 50%

김준범: 50%

>> If you have any preliminary comments on your submission, notes for the
>> TAs, or extra credit, please give them here.

>> Please cite any offline or online sources you consulted while
>> preparing your submission, other than the Pintos documentation, course
>> text, lecture notes, and course staff.

ARGUMENT PASSING

---- DATA STRUCTURES ----

>> A1: Copy here the declaration of each new or changed `struct' or

>> `struct' member, global or static variable, `typedef', or
>> enumeration. Identify the purpose of each in 25 words or less.

없다.

---- ALGORITHMS ----

>> A2: Briefly describe how you implemented argument parsing. How do
>> you arrange for the elements of argv[] to be in the right order?
>> How do you avoid overflowing the stack page?

Argument parsing은 start_process() 안에서 진행한다. 먼저 strtok_r()를 사용해 공백을 기준으로 file_name을 자르며 argument의 개수를 센다.

계산한 argc값으로 argv에 필요한 메모리 공간을 할당하고 다시 한번 parsing을 진행하며 argv 배열 안에 넣어준다.

User program의 main에 필요한 argument(argc, argv)는 미리 stack에 push해야 한다. 이 작업은 push_arguments()에서 진행된다.

push_arguments()가 가장 먼저 진행하는 것은 argv 배열의 문자열들을 뒤에서부터 push하는 것이다. 이것으로 user program의 argv[]는 80x86 calling convention이 요구하는 순서를 만족하게 된다.

project 2의 document에 따라서 입력 받는 file_name은 128byte 보다 크지 않도록 하였다. 이 때문에 argument passing 과정에서는 overflow는 없게 된다.

또, user program이 다른 user program을 실행하는 경우 새로운 page를 allocate하고 esp가 page의 꼭대기로 초기화되기 때문에 반복적인 호출에서도 overflow가 일어나지 않게 된다.

---- RATIONALE ----

>> A3: Why does Pintos implement strtok_r() but not strtok()?

strtok()은 내부 implementation에서 static 변수를 활용한다. 따라서 여러 개의 thread가 동시에 strtok()를 호출하게 되는 경우에 해당 변수가 동시에 접근되면서 race가 발생할 가능성이 있다. 반면에 strtok_r()은 stack 내에서만 구동되어 여러 thread에 대해서 safe하다.

>> A4: In Pintos, the kernel separates commands into a executable name
>> and arguments. In Unix-like systems, the shell does this
>> separation. Identify at least two advantages of the Unix approach.

Shell이 command line을 parsing하여 kernel에 넘겨주는 경우, invalid command line에 의한 kernel panic을 방지할 수 있다.

최악의 경우 shell이 crash하겠지만 kernel이 panic하는 것 보다는 차라리 shell crash가 낫다. 그리고 kernel이 command parsing을 하지 않아 kernel의 overhead가 줄어들게 된다.

SYSTEM CALLS

---- DATA STRUCTURES ----

>> B1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration. Identify the purpose of each in 25 words or less.

```
struct thread
  {
    ...
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
    struct list children;               /* List of child process. */
    struct list_elem elem_child;        /* List element for children list. */
    struct file* fd_table[128];         /* Array of file descriptor.  */

    int status_exit;                    /* Exit status for exit() system call. */
    struct semaphore exit_sema;         /* Semaphore for synchronization of exiting
                                           child. */
    struct semaphore delete_sema;       /* Semaphore for synchronization of
                                           deleting child from children. */

    bool failed;      /* Boolean which represents failure of load() function. */
```

```
    bool loaded;                        /* Boolean which represents success of
                                           load() function. */
#endif
    ...
  };
```

struct list children은 현재 process가 exec한 자식 thread들이 담긴 list이다.

fd_table은 file descriptor와 file pointer를 mapping하는 배열이다.

exit_sema, delete_sema는 parent 프로세스가 child 프로세스를 기다릴 때 synchronization을 위해 도입했다.

boolean failed와 loaded는 user 프로그램이 정상적으로 load가 된 것을 확인한다.

>> B2: Describe how file descriptors are associated with open files.
>> Are file descriptors unique within the entire OS or just within a
>> single process?

우리의 디자인에서는 thread 구조체가 각각 독립적인 file descriptor table을 갖는다.

이 fd_table은 open된 파일의 file pointer를 요소로 가진 배열이다. fd_table의 사이즈는

128로 고정되며, file을 open할 때 새롭게 fd 값을 부여할 때는 사용 중이 아닌 fd를 2부터 중복되지 않게 부여한다.

결국 특정 파일을 open할 때 fd_table의 index와 부여된 fd를 매칭하여 file pointer를 넣게 되어 모든 프로세스가 각각 unique한 file descriptor table을 가진다.

---- ALGORITHMS ----

>> B3: Describe your code for reading and writing user data from the
>> kernel.

Data를 쓰는 경우, syscall_handler에서 syscall_write()가 호출되는데 fd가 1인 경우와 아닌 경우를 나누어 처리한다.

fd가 1인 경우 putbuf()를 활용해 console에 data를 쓴다. fd가 1이 아닌 경우 file_write()를 활용해 buffer에 size만큼을 쓴다.

Data를 읽는 경우, syscall_handler에서 syscall_read()가 호출되는데 fd가 0인 경우와 아닌

경우를 나누어 처리한다.

fd가 0인 경우 input_getc()로 keyboard에서 입력된 것을 받는다. fd가 0이 아닌 경우 file_read()를 활용해 size 만큼을 파일에서 읽는다.

kernel이 user data을 읽을 때는 invalid한 pointer를 받지 않도록 조심해야 한다.

이때 invalid pointer의 종류는 NULL pointer, kernel 영역을 가리키는 pointer, mapping 안된 영역을 가리키는 pointer이다.

각각의 경우를 check_invalid_pointer()가 확인하여 invalid pointer라면 exit(-1)을 한다.

>> B4: Suppose a system call causes a full page (4,096 bytes) of data
>> to be copied from user space into the kernel. What is the least
>> and the greatest possible number of inspections of the page table
>> (e.g. calls to pagedir_get_page()) that might result? What about
>> for a system call that only copies 2 bytes of data? Is there room
>> for improvement in these numbers, and how much?

Page 내에 4096bytes 크기의 단일 데이터로 있으면 1번의 inspection이 필요하다.

만약 1byte 크기의 데이터로 나뉘어 있다면 최대 4096번의 inspection이 필요하게 된다.

위와 마찬가지의 논리로 2byte 데이터의 경우에는 최소 1번, 최대 2번의 inspection이 요구된다.

우리의 디자인에서는 check_invalid_pointer()를 매 system call마다 부르기 때문에

improvement할 요소가 없다고 판단된다.

>> B5: Briefly describe your implementation of the "wait" system call
>> and how it interacts with process termination.

syscall_handler는 "wait" system call에 대한 syscall number를 받게 되면 syscall_wait()을 호출하고 이 함수의 리턴 값을 %eax 레지스터에 저장한다. syscall_wait()은 스택에서 뽑아낸 pid 인자와 함께 process_wait()를 호출한다.

process_wait()는 현재 실행 중인 process의 children을 순회하며 인자로 받은 pid를 가지는

child를 찾는다. 만약 찾지 못한다면 -1을 리턴하고, 찾는다면 다음의 작업을 수행한다.

1. child의 exit_sema를 down한다.

2. sema_down()이 수행되면 child를 현재 프로세스의 children에서 지운다.

3. child의 exit status를 회수한다.

4. child의 delete_sema를 up한다.

5. 회수한 exit_status를 리턴 한다.

1과 3과정은 process termination 작업과 synchronization하기 위해서 필요하다.
thread 구조체를 해제한 후에 exit status를 회수하면 안되기 때문에 동기화가 필요하다.
자세한 설명은 B8에 이어서 작성하였다.

위의 과정으로 parent는 child가 완벽히 exit하기를 기다리고, 안전하게 exit status를 회수한다.

```
>> B6: Any access to user program memory at a user-specified address
>> can fail due to a bad pointer value.  Such accesses must cause the
>> process to be terminated.  System calls are fraught with such
>> accesses, e.g. a "write" system call requires reading the system
>> call number from the user stack, then each of the call's three
>> arguments, then an arbitrary amount of user memory, and any of
>> these can fail at any point.  This poses a design and
>> error-handling problem: how do you best avoid obscuring the primary
>> function of code in a morass of error-handling?  Furthermore, when
>> an error is detected, how do you ensure that all temporarily
>> allocated resources (locks, buffers, etc.) are freed?  In a few
>> paragraphs, describe the strategy or strategies you adopted for
>> managing these issues.  Give an example.
```

우선 syscall_handler는 pointer형의 argument에 대해서 항상 validity를 체크한다.
예를 들어, read system call은 buffer라는 pointer형 argument를 필요로 한다.
syscall_handler는 %esp값으로부터 해당 argument에 접근하고 check_invalid_pointer() 함수를 호출하여 이 pointer의 validity를 체크한다.

check_invalid_pointer()는 현재 프로세스의 pagedir pointer, pd와 validity를 체크할 pointer, ptr을 인자로 받고, 다음의 세 가지 경우를 체크한다.

1. ptr이 NULL 포인터인가?
2. ptr이 kernel 영역의 주소인가?
3. ptr이 pd가 가리키는 페이지에 mapping되어 있지 않은 주소인가?

나열한 순서대로 validity를 체크하며, 하나라도 만족하면 syscall_exit(-1)을 호출하여 해당 process를 terminate한다. 모두 만족하지 않는다면, user program에서 access 가능한 pointer이므로 아무런 동작을 하지 않는다.

예를 들어, read system call이 buffer pointer로 NULL값 혹은 kernel 영역의 주소를 넘겨받았다면 check_invalid_pointer()에 의해서 해당 system call을 호출한 process를 terminate하게 된다.

syscall_exit(-1)이 호출되면 process가 열었던 file descriptor를 모두 닫아주고, child가 모두 exit할 수 있도록 체크해주며, pintos가 struct thread를 해제해주기 때문에 process에 할당된 data structure는 모두 해제된다.

---- SYNCHRONIZATION ----

>> B7: The "exec" system call returns -1 if loading the new executable
>> fails, so it cannot return before the new executable has completed
>> loading. How does your code ensure this? How is the load
>> success/failure status passed back to the thread that calls "exec"?

우리는 두 개의 boolean, loaded & failed, 변수와 thread_yield()를 통해서 simple synchronization for waiting loading을 구현하였다. 이는 syscall_exec() 함수와 start_process() in process.c에 걸쳐서 구현되어 있다.

우선 exec system call이 호출되면 syscall_handler는 syscall_exec()을 호출한다. syscall_exec()은 process_execute()를 호출하여 작동하기 때문에 새롭게 생성된 child는 start_process()를 실행함으로써 인자로 받은 cmd_line을 수행한다.

start_process() 내에 있는 load() 함수의 return 값이 false라면 load는 실패한 것이다. 이때 child의 failed 변수는 true로 바뀐다. 만약 return 값이 true라면 load는 성공한 것이므로 child의 loaded 변수가 true로 바뀐다. syscall_exec()에서는 child의 loaded가 false인 동안 thread_yield를 하며 child가 load되기를 기다린다. 만약 child가 load에 실패한다면 failed는 true로 바뀌기 때문에 -1을 리턴하고 child가 종료되도록 delete_sema 도 up을 해준다.

while 반복문과 thread_yield()를 통해서 child의 load status가 변하는지 계속 모니터링하기 때문에 child는 load가 실패하면 이를 parent의 exec system call에 알릴 수 있다. 따라서, 우리의 코드는 child의 load가 실패하면 parent의 exec은 -1을 리턴 한다는 것을 보장할 수 있다.

```
>> B8: Consider parent process P with child process C.  How do you
>> ensure proper synchronization and avoid race conditions when P
>> calls wait(C) before C exits?  After C exits?  How do you ensure
>> that all resources are freed in each case?  How about when P
>> terminates without waiting, before C exits?  After C exits?  Are
>> there any special cases?
```

다음은 parent가 child를 wait하는 과정에서 일어나는 동기화 작업에 대한 설명이다.

child가 exit하는 과정에서는 thread_exit() -> process_exit() 순서로 함수들이 호출된다. process_exit() 함수에서는 child가 생성될 당시 0으로 초기화되었던 exit_sema를 up하게 된다. parent는 process_wait()에서 child가 해당 sema를 up해주기를 기다려야 하기 때문에 child의 termination 과정이 시작해야만 children에서 child를 지우고 이 child가 exit했다는 것을 보장해줄 수 있다.

그 다음, child는 process_exit()에서 delete_sema를 down한다. parent는 child의 struct thread로부터 exit status를 회수하기 때문에 child의 data structure들이 해제되지 않은 상태임을 보장받아야 한다. 따라서 delete_sema를 down함으로써 child는 parent가 해당 sema를 up해주기를 기다려야 하고, parent는 sema_up() 이전에 exit status를 회수한다.

질문에서 제시한 첫 번째 문제를 생각해보자.

1. C가 exit하기 전에 P가 wait(C)를 호출한 경우

P는 C의 exit_sema가 up되어야만 exit status를 회수할 수 있다. 따라서 C가 exit되기를 기다릴 것이며, exit된다면 비로소 exit status를 회수하고 이후 작업을 진행한다.

2. C가 exit한 후에 P가 wait(C)를 호출한 경우

C가 먼저 exit를 하게 되면 C의 exit_sema는 up된 상태이지만 delete_sema를 down하는 작업에서 멈춰 있는 상태이다. 따라서 P가 wait(C)를 호출하여 exit_sema를 down하고, delete_sema를 up함으로써 C의 exit와 P의 wait(C)를 정상적으로 진행될 것이다.

여기서 쓰이는 두 개의 semaphore 변수들은 thread 구조체 내에 포함된 것이기 때문에 exit하는 과정에서 모두 free된다.

질문에서 제시한 두 번째 문제를 생각해보자. 이 문제는 parent가 먼저 exit해버리는 경우이다.

check_child_before_exit()라는 함수를 이용하여 이 문제를 해결할 수 있다.

check_child_before_exit() 함수는 process가 syscall_exit()를 통해 exit할 때,

thread_exit()를 호출하기 이전에 child를 모두 체크하며 child가 있다면 delete_sema를 up해준다.

1. C가 exit하기 전에 P가 wait하지 않고 terminate된 경우

P가 terminate될 때 호출되는 check_child_before_exit()에서 C의 delete_sema를 미리 up해준다. C는 delete_sema를 down하기 위해서 기다릴 필요 없이 정상적으로 exit될 수 있다.

2. C가 exit한 후에 P가 wait하지 않고 terminate된 경우

P가 terminate될 때 호출되는 check_child_before_exit()에서 C의 delete_sema를 up해준다.

현재 C는 exit를 할 때 delete_sema를 down하기 위해 막혀 있었기 때문에 delete_sema가 up 되면서 정상적으로 exit될 것이다.

C의 exit 여부와 P의 wait(C) 호출 여부에 따라서 발생 가능한 경우는 위에 열거한 4개이다.

---- RATIONALE ----

>> B9: Why did you choose to implement access to user memory from the
>> kernel in the way that you did?

Pintos 매뉴얼에서 제시한 방법 중, 우리는 invalid pointer를 방지하기 위해서 verify validity of user-provided pointer, then dereference 방법을 선택했다.

pagedir_get_page()와 is_kernel_addr() 함수를 적절히 활용하여 check_invalid_pointer()를 구현하였고, 결과적으로 직관적이고 알아보기 쉬운 방법이 되었기 때문에 선택했다. syscall_handler에서 각 system call에 대해서 항상 이 check_invalid_pointer()로 validity를 조사하기 때문에, 어떤 부분에서 validity를 체크했는지, 어떤 pointer가 잠재적 위험성을 가지는지 한눈에 알아볼 수 있다는 장점도 있다.

>> B10: What advantages or disadvantages can you see to your design
>> for file descriptors?

우선 우리는 process별로 file descriptor를 관리하기 위해 file descriptor table을 thread 구조체 내에 만들었다. Fixed-size array이며, array index는 fd 값을 나타내고 각 element는 file pointer이다.

1. 장점

- fd 값이 곧 array index이기 때문에 프로세스가 관리하는 file pointer에 접근하기 쉽다.

- 프로세스 별로 독립적인 file descriptor table을 가지기 때문에 다른 프로세스의 table을 고려하지 않아도 된다.

2. 단점

- STDIN, STDOUT에 할당된 fd 값이 모든 process의 file descriptor table에 존재한다. 이는 항상 두 개의 array element가 낭비될 수 있음을 의미한다.

- 한 프로세스가 열 수 있는 파일의 개수가 array의 사이즈(=128)로 제한된다. 이는 Pintos FAQ를 참고하여 설정한 max값이다.

>> B11: The default tid_t to pid_t mapping is the identity mapping.
>> If you changed it, what advantages are there to your approach?

우리는 바꾸지 않았다. tid_t를 곧 pid_t로 생각하고 과제를 진행하였다.

만약 둘 사이의 mapping을 바꾼다면, child가 생성될 때 parent의 tid에 특정 산술연산을 추가하여 parent-child 관계가 생기도록 child의 pid를 정하면 children 리스트를 따로 추가하지 않아도 되는 장점이 있다. 하지만, 여전히 child를 알아내거나 parent를 알아내려면 모든 thread들을 순회해야 하는 단점이 생길 것 같다.

SURVEY QUESTIONS
==========

Answering these questions is optional, but it will help us improve the course in future quarters. Feel free to tell us anything you want--these questions are just to spur your thoughts. You may also choose to respond anonymously in the course evaluations at the end of the quarter.

>> In your opinion, was this assignment, or any one of the three problems
>> in it, too easy or too hard? Did it take too long or too little time?

>> Did you find that working on a particular part of the assignment gave
>> you greater insight into some aspect of OS design?

>> Is there some particular fact or hint we should give students in
>> future quarters to help them solve the problems? Conversely, did you
>> find any of our guidance to be misleading?

>> Do you have any suggestions for the TAs to more effectively assist
>> students, either for future quarters or the remaining projects?

>> Any other comments?