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

---- GROUP ----

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

김건우 <kakaloto85@kaist.ac.kr> 55%
신정윤 <jugipalace@kaist.ac.kr> 45%
FirstName LastName <email@domain.example>

---- PRELIMINARIES ----

>> 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.
thread.h 파일에서 우리는 userprog을 위한 구조들을 추가했다.

process.c:
- process_execute():
  - fn_copy2 :process_execute에서 real name을 찾아내는 역할,strcpy를 이용해 file_name을 복사.
  - realname: fn_copy2에서 strtok_r로 찾아낸 함수의 이름(ex. 'echo')
- start_process():
  - char *next_ptr
  - char* token
    - -> 1.strtok_r를 이용하여 start_process의 인자의 argument들을 분리하여 token에 저장
  - int argc
  - char** argv[30]
    - -> 2. 1 과정에서 token들을 argv배열에 저장. argc는 넣을 argv에 token을 넣을때마다 1씩 증가시켜줌.

---- 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?
먼저, 우리는 start_process 안에 char** argv[30] 을 선언하여 최대 30개의 argument를 받을 수 있는 array를 만들었고,
우리는 strtok_r 함수를 이용하여 file_name을 공백(" ")을 기준으로 토큰화하여 각 토큰을 차례대로 array에 넣었다.

그 후, 첫번째 원소, argv[0] (즉, file_name 자체)에 대한 load가 성공하면, 그후 다른 인자들을 stack에 삽입하는 함수 parse_stack을 만들었다.

void parse_stack (char** argv, int argc, void** esp) 함수는 *esp 값을 낮춰가며 argv의 각 원소들, word-align을 위한 stack, 0으로 채워진 여분 argv, 그 후 앞서 채운 원소들이 저장된 주소, argv array의 시작 주소, argc 값, 그리고 마지막으로 return address 의 용도로 stack pointer를 한 번 더 낮추는 과정이다.

parse_stack의 각 삽입 과정에서 반복되는 과정들은 for문을 이용하였고, strlen 을 이용하여 *esp를 직접적으로 계산해주면서 필요한 data들을 삽입한다.

---- RATIONALE ----

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

strtok 함수는 정적 버퍼를 이용하여 한 thread에서 쭉 진행될 때는 상관이 없지만, 여러 개의 thread를 사용할 경우 다른 thread 에서도 같은 버프를 사용해 문제가 발생할 수 있다. 반면, strtok_r 는 save_ptr를 별도의 인자로 저장해, 서로 다른 thread로부터 interrupt 되어도 각각의 save_ptr 로 부터 current 위치를 얻을 수 있기에 multi-thread 를 이용하는 Pintos에서는 strtok_r() 를 사용하는 것이 strtok를 사용하는 것보다 용이하다.

>> 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.
-> kernel에서 argument passing을 하는 경우 유효하지 않은 argument가 들어왔을 때 커널 패닉이나, 시스템 적으로 오류가 날 수 있다. 하지만 shell에서 먼저 argument passing을 하는 경우 kernel에 오류를 넘겨주지 않기 때문에 더 안전하고, 먼저 처리 하기 때문에 좀 더 효율적이다.

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.
thread.c:

struct file* fd_table [130]; 이는 각 process에서 다루는 file의 포인터들을 fd 값에 해당하는 리스트에 저장해놓은 table로, open이면 파일의 fd가 설정되고 fd_table[fd] 에 file 이 저장된다. close(fd)가 될 시에는 fd_table[fd]에 NULL 이 저장되어, 추후 다른 파일이 해당 fd를 사용할 수 있다. 한 프로세스 당 최대 128개의 파일에 대한 정보를 저장한다. 0,1의 fd는 각각 stdin, stdout으로 쓰이기 때문에, 2 부터 채우도록 하였다.

tid_t pid 는 이후, syscall wait과 exec에서 사용하게 되는 process id 인데, 우리는 tid 값을 그대로 pid로 이용하였다.

struct semaphore wait_lock; wait_lock은 child_process 가 exit할때까지 parent_process를 wait하게 해주는 역할의 semaphore다. process_wait 에서 down 되고, process_exit에서 child process가 끝나면 parent의 wait_lock 을 sema_up 해주어, parent_process 가 다시 진행될 수 있게 한다.

struct semaphore load_lock; load_lock은 child_process 가 load를 마칠 때까지 parent_process 를 잠그는 역할을 한다. child_process가 load를 마치기 전에 parent process를 다시 실행하면, child_process의 파일을 변경하거나 child process가 load를 못하고 죽어버렸을 때 문제가 생길 수 있다.

struct list child_wait_list; child_wait_list는 아래에 있는 child_list_elem으로 부모 프로세스의 child_process들을 저장한 list이다. 즉, 부모 기준으로 child process(thread) 들을 모아놓은 list다.

struct list_elem child_list_elem; child thread를 부모의 child_wait_list에 넣을 수 있게 만들어준 list_elem 이다.

struct thread* parent; 각 child process의 parent process 를 기억한다.
int child_status;
int exit;
결국 child_status도 child가 exit을 잘했는지를 판단하기 위한 요소인데, 이거를 exit flag처럼 하나로 만들 수는 없을까? 같이 보면서 얘기해봐야할 것 같아요.

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

fd 값은 한 프로세스 내의 fd_table 에서만 구별되며, OS 전체에서는 겹치는 fd 들이 있을 수 있다. 그래서 우리는 thread_current->fd_table[fd]로 file을 찾는다. 각 프로세스의 fd_table은 처음에 모든 값이 NULL로 초기화되어있고, file이 open 되면 for문을 통해 fd_table에서 NULL로 되어있는 fd 값에 file을 할당해준다(2이상 129이하). 이후, void close(int fd)를 하면, fd_table에서 해당하는 fd의 file을 file_close를 이용해 닫아주고, fd_table[fd] 값을 NULL로 초기화해준다.

---- ALGORITHMS ----

>> B3: Describe your code for reading and writing user data from the
>> kernel.
int read(int fd, void *buffer, unsigned size)는 우선 buffer의 주소가 user 가상 메모리에 있는지 확인하고, 있다면 sema_down(&file_lock)을 통해 file_lock을

걸어준다. 우선 fd가 0이면 input_getc()로 읽는다. 그 외의 경우는, fd_table[fd]에 해당하는 file을 file_read 함수를 이용해 주어진 size 만큼 읽는다. 총 읽힌 byte 수를 return하고 sema_up을 통해 file_lock을 다시 풀어준다.

int write (int fd, const void *buffer, unsigned size)에서도 sema_down(&file_lock)을 통해 file_lock을 걸어주고 fd가 1일때는, putbuf로 buffer의 내용을 size만큼 console 에 적는다. 아닌 경우, fd에 해당하는 file을 찾고, 그 file가 write가 허용되는지를 deny_write 값으로 확인한다(이는 open에서 만약 현재 thread의 이름과 open하는 file의 이름이 같은 경우 file_deny_write 함수를 통해 deny_write를 1로 바꿔준다). write 가 허용된 경우, file_write를 통해 buffer에서 size만큼을 file에 쓰고 file_lock을 풀어준다.

>> 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?
현재는 user space에서 kernel로 mapping할 때 page단위로 inspection을 실행하고 있다.
만약 fullpage가 연속적인 경우,최소 한 번의 inspection(pagedir_get_page())만이 필요할 것이다.(user space data의 start address만 확인하면 됨)
그러나 4,096bytes가 1byte씩 불연속하게 다른 페이지에 저장되어 있는 경우에는 최대인 4,096번의 inspection이 필요할 것이다.
2 byes일 경우에도 마찬가지로, 최소 1번 최대 2번의 inspection이 필요할 것이다.
만약 user space에 data를 페이지 단위로 compact하게 저장한다면, inspection의 수를 줄일 수 있을 것이다.

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

```
int
wait (pid_t pid)
{
 return syscall1 (SYS_WAIT, pid);
}
```

을 통해 pid와 sys_wait이라는 call number를 커널에 넘겨준다.
userprog의 system call에서 sys_wait이라는 call number를 통해 wait함수를 실행시킨다.
이때 wait함수에서는 커널에서 pid값을 가져와서 인자로 사용하여

```
int wait (tid_t pid){
      return process_wait(pid);
}
```

를 실행한다.

process_wait(tid_t child_tid) 함수는, 다음과 같은 과정으로 child process가 죽기를 기다리고, exit status를 return 한다.

1. child_tid가 NULL 인지 확인하고, parent의 child_wait_list가 비어있지 않은지, 즉 현재 thread가 child가 있는지 확인한다.

2. for 문을 돌려서, 인자로 받은 child_tid를 tid값으로 가지는 child_process가 있는지 찾는다.
3. child_tid 와 tid가 같은 child_process가 있다면, current thread(parent process)의 wait_lock 을 sema_down 하여 child가 exit하면서 sema_up을 시켜줄 때까지 기다린다.
4. 이후, child_process 를 child_wait_list 에서 지우고, child의 exit status 를 저장한다.
5. child process의 page를 free 해주고, 저장해놓은 exit status 를 return한다(interrupt frame의 eax에 저장하여 유저로 넘겨줌).

>> 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.
시스템 콜을 실행할 때, syscall_handler에서 각각의 syscall에 대해, 먼저 현재 interrupt frame의 *esp가 적절한 위치에 있는지를 check_user_sp(const void *sp)를 통해 확인한다.또한 buffer 포인터를 argument로 받는 read와 write 시스템 콜의 경우, buffer가 가리키는 주소 또한, 유저 메모리에서 적절한 위치에 있는지 확인하기 위해 check_user_sp를 진행해준다. check_user_sp를 통해, 유저 메모리에서 벗어나는 위치에 있는 argument 가 있는 경우, exit(-1)을 호출하여 자원을 free해준다. 모든 프로세스는 exit을 하는 경우, 모든 열려있는 file들을 close 해주고 pagedir을 파괴한다.

이후, 부모 process에서 child process로부터 exit status를 전달받고 나서는 process_wait에서 부모 프로세스가 자식 프로세스를 palloc_free_page(child)를 해주어, 생성된 lock이나 fd_table 등의 structure 들을 free 해준다.

우리는 exception.c 의 page fault(struct intr_frame *f)에서도 exit(-1)을 호출하여 page fault를 일으키는 code나 data 에 대해서도 exit을 하여 자원을 초기화하는 과정을 거친다.

예시로는, tests 에 있는 read-bad-ptr.c 파일의 경우 read (handle, (char *) 0xc0100000, 123); 을 실행하는데, 이때는 buffer의 주소가 적절하지 않은 경우이므로, system call read에서 child_user_sp(buffer)를 해줘서, exit(-1)을 호출한다.

두 번째 예시로, bad-read.c 의 경우는 msg("Congratulations - you have successfully dereferenced NULL: %d", *(volatile int *) NULL); 을 호출하는데 이때 msg 를 통해, read를 하게 된다. 그러나, 이 경우 NULL pointer는 가리키는 값이 없으므로, mapping 이 되지 않은 주소에서 메모리를 읽으려 하는 과정이고, 우리는 이런 문제를 exception.c에서의 page fault를 수정함으로써 해결하였다.

---- 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"?

"exec" system call 은 process_execute(cmd_line)을 실행한다. 그 경우, child process를 생성하며 start_process 가 child process의 function으로 들어간다. loading 과정은 start_process 함수 내의 load 함수에서 발생한다.

start_process 함수에서 load 를 실패할 경우, thread_current-> exit = 1; 을 한다. exit은 load가 실패했을 때를 위한 flag다.(default는 0으로 설정/ load 실패 시 1로 설정)이런 경우, child 프로세스가 비정상적으로 끝났을 때, 우리는 process_execute에서 process_wait(tid)를 호출하여, load가 안되었을 때 메모리를 빠르게 지워주고, -1 값을 return 한다.

>> 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?
before C exits: P에서는 wait을 통해 process_wait이 불리고, process_wait에서는 sema_down(&thread_current()->wait_lock)을 통해 C가 exit할때까지 기다린다.
C가 exit할떄, sema_up(&parent->wait_lock)을 해주고, P가 종료된다.
After C exits: 이미 C가 exit할떄, sema_up(&parent->wait_lock)을 해준 상태이므로 P 의 process_wait에서 sema_down(&thread_current()->wait_lock)을 바로 탈출한다.

두 상황모두 process_wait이 정상적으로 실행되므로, palloc_free_page(child)를 통해 C가 정상적으로 free가 된다. (C의 pagedir은 C가 exit할때 process_exit에서 free해줌)

P가 C를 기다리지 않고 종료되면, C가 free되지 못한다.

---- RATIONALE ----

>> B9: Why did you choose to implement access to user memory from the
>> kernel in the way that you did?
user의 syscall.c에서 userprog의 syscall handler로 interrupt frame을 이용하여 syscall의 number와 argument들을 넘겨준다. 이때 각각의 syscall에 따라 argument의 개수가 다르기 때문에 이를 고려하여,interrupt frame의 *esp값을 4씩 증가해가며, 해당 *esp가 가리키는 값을 argument로 받는다. 이때 B6에서 말한 방식으로 invalid한 값들이 들어왔을때 exit(-1)을 하여 메모리 누수를 막는다.

>> B10: What advantages or disadvantages can you see to your design
>> for file descriptors?
우리의 디자인은 fd 값을 for 문을 이용해 각 프로세스의 fd_table array 에서 일치하는 값을 일일히 확인하여 찾아야한다. 기존에 우리가 생각했던 방법은, int next_fd라는 값을 저장하여, 중간에 close 되어 NULL로 초기화되어도 최댓값+1 만을 next_fd로 사용하는 방법이었다. 현재 우리가 생각한 방법은, 기존의 방법보다 속도 면에서는 비효율 적일 것이다.하지만 이 방법은 기존에 고려했던 방법보다 fd_table의 메모리 공간을 좀 더 효율적으로 사용할 수 있다는 장점이 있다.

>> B11: The default tid_t to pid_t mapping is the identity mapping.
>> If you changed it, what advantages are there to your approach?
핀토스에서는 thread와 process가 1대1 대응이기 때문에 tid_t와 pid_t를 identity mapping해도 문제가 없지만, 한 process가 multi thread로 이루어진 os의 경우에서는 multi thread를 고려하여 pid를 만드는 것이 더 효율적일 것이다.

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?