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

---- GROUP ----

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

Team 31
안동현 <segaukwa@kaist.ac.kr>
함동훈 <hdh8277@kaist.ac.kr>

0 tokens used

---- 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.

http://csl.skku.edu/SSE3044F14/Projects
의 PPT 를 참고하여 기본적인 이해를 하였습니다.

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.
None

---- 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?

스택 구조 안에 주어진 input 과 함께 그 주소들을 넣는게 목적이었다. 첫 번째 argument 는 command 로, 나머지는 command 를 위한 argument 로 취급하여 이후 start_process, load() 등에 쓰이게끔 했다. argv[]의 element 를 넣는 과정은 string 을 뒤에서부터 읽어 token 을 갖도록 했다. overflow 가 일어나지 않도록 스택에 어떤 값이 들어갈때 그 값이 들어갈 주소가 valid 한 주소인지 확인하는 작업을 진행하였다.

---- RATIONALE ----

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

Strtok_r()은 strtok()와 다르게 세번째 인자로 포인터 save_ptr 를 받는다. 이 포인터가 있어야 나중에 들어온 argument 를 언젠가 call 해줄 수 있고 사용할 수 있다.

>> 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.

1) 커널 내 연산의 속도가 빠르다.
2) 커맨드 분리를 구현한다면, 다양한 커맨드를 한번에 처리할 수 있다.

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.

<process.h>

```
struct file_descriptor

{

   int fd;                                 // fd number

   struct file* file;                      // 해당 구조체가 나타내는 파일

   struct list_elem fd_elem;               // 해당 구조체의 파일을 리스트에 넣기 위한 element

};
```

File 을 관리하는 file descriptor 구조체의 struct file_descriptor

<thread.h> - 추가된 부분

```
#ifdef USERPROG

      uint32_t *pagedir;              //Page directory. – checking whether valid page or not

      int exit_status;                //Exit status when exit. Tests requires -1 for abnormal cases

      int child_exec;                 /* child success to load exec file */

      bool wait_child;                /* child wait status */

      int orphan;                       /*To manage orphan threads, that parents exit first*/

      struct thread * parent;           /*each thread may contain information of parent thread*/

      struct list child_list;           /*each thread may contain child list called by current thread*/

      struct list open_file;            /*list of file descriptor, to manage opened file by this thread*/

      struct semaphore waitsema;          /* semaphore for wait child */

      struct semaphore loadsema;          /*semaphore for load processes in order*/

      struct lock loadlock;                /* lock for load file of child */

      struct condition loadcond;          /* condvar for signal load success */

      struct file * exec_file;            /*file that currently executing. Need for deny write*/

#endif
```

process 를 수행하는 과정에서 발생할 수 있는 file 처리를 위해 file descriptor 를 리스트로

가지고 있고, 또한 child thread 의 list 를 가지고 있어 parent thread 의 입장에서 child 를 관리할 수 있도록 해준다. 또한 execution 에서 발생할 수 있는 race condition issue 를 해결하기 위한 semaphore 들을 구현해주었다.

<thread.h>

```
struct child_thread                    /*The single child thread structure*/
  {
    tid_t tid;                         //tid of child
    struct thread* thr;                //child thread
    struct list_elem ch_elem;          //elem for child list
    int exit_status;                   //save exit status to report how to exit
    int dead;                          //0 for alive, 1 for dead thread
  };
```

Child thread 를 관리할 수 있게끔 child 의 정보를 담은 구조체이다. Thread 의 child_list 를 통해 관리될 것이다.

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

file descriptor 는 0,1,2 번 이후에는 열려있는 파일을 대응하는 매개체 역할을 한다. fd 는 OS 전체에서 유일하게 지정되어 있어, 열려있는 파일에 대한 정보를 모든 프로세스가 알 수 있도록 한다. 이 이유는 서로 다른 프로세스에서 동일 파일을 중복으로 'write'할 수 없게끔 해야하는데, 그러기 위해서는 write 중인 파일을 알아야 하기 때문이다.

---- ALGORITHMS ----

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

READ

우선 buffer 와 buffer+size 가 모두 valid pointer 에 속하는지 확인한다. 아닌 경우 exit(-1)
fd 가 0 인 경우 standard input 을 buffer 에 받는다. 그 과정에서 에러가 생기는 경우 exit(-1)
에러 없이 끝났으면 그렇게 받은 buffer 의 size 를 return.

Fd 가 0 이 아닌 경우, 보조함수 fd_to_file 을 사용하여 fd 로부터 읽어야할 파일을 가져온다.
이때 fd 가 1 또는 2 여서 혹은 기타 이유로 읽을 파일이 NULL 로 나타나는 경우 return -1

3 이상이고 읽을 파일을 제대로 가져왔으면 그 읽어야 할 파일의 사이즈를 return
WRITE
마찬가지로 valid pointer 체크를 실시하여, 불가능한 경우 exit(-1)
fd 가 1 인 경우 standard output 이므로 putbuf 를 통해 해당 buffer 부터 size 만큼을 받아 쓰고, return 으로는 size 를 내보낸다.
그 이외의 경우 마찬가지로 fd_to_file 을 사용하여 fd 로부터 읽어야할 파일을 가져온다. 이때 fd 가 0 또는 2 여서 혹은 기타 이유로 작성할 fill 이 NULL 로 나타나는 경우 return 0 을 한다.
3 이상이고 읽을 파일을 제대로 가져왔으면, file_write 를 콜하여 return 값을 반환한다.

>> 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?

Full page data 가 연속되었다면, 시작값과 끝값만으로 그 안의 space 를 배분이 가능하다. 애초에 paging 을 하는 이유이기도 하다. 사이즈가 4096 이든 2 이든 2 번안에 저장 공간 배분이 가능하다. 만일 page 가 저장공간을 띄엄띄엄 사용한다면 최대는 4096 이 될 수도 있을 것이지만, 우리가 본 프로젝트에서 구현한 내용은 page 내에서 주소값들이 연속되게끔 하였다. 따라서 본 프로젝트는 최대 2 번, 최소 2 번 안에 주소 할당이 가능하다. 이를 원활히 진행하기 위해 page allocation 과 deallocation 을 성실하게 해 주었다.

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

Wait 을 부르는 주요 이유는 parent 가 child 의 execute 에서 exit 을 반환할때까지 기다려 주는 역할이다. tid 에는 기다릴 child 의 tid 가 들어간다. syscall 에서는 그냥 process_wait 을 부르고 모든 것을 process_wait 에서 처리한다. 반환해야 할 것은 기다린 child 의 exit_status 이다. Wait 을 하는 것은 parent 의 입장으로서, 현재 thread 를 parent 라 생각한다. 그 thread 의 chlist 를 찾아보고 주어진 tid 와 매칭시켜서 list_entry 를 통해 child_thread 구조체를 얻어낸다. 만약 얻어내는데에 오류가 있다면(chlist 가 null 이거나, child 가 이미 죽었거나 등) 에러를 나타내는 -1 을 return 한다. 그 후 exit_status 를 반환하려면 일단 child 가 exit 를 해야 하므로 이를 sema_down 을 통해 조절한다. 그리고 exit()에는 child 기준으로 생각하여 parent 의 세마포어, 즉 방금 언급한 sema_down 된 세마포어를 exit()의 끝자락에 작성하여 wait 이 마저 실행되도록 한다. 이를 위해 parent 에 booltype 인 wait_child 를 보조적으로 활용하였다. 세마포어가 풀리면, wait_child 가 풀려났음을 선언하고, child_thread 구조체에 dead 속성값을 1 로 넣어준다.

>> 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
000>> 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 bad user memory access 를 address 부여 전에 체크함으로서 방지한다. Write 를 할 때 buffer 의 시작과 끝을 보고, 메모리 배분시 장애가 있으면 에러를 리턴한다. 그 이외에 bad-jump2-test 나 multi-oom 의 일부가 나타내는 *(int*)0xC0000000 = 42 같은 경우는 page fault 를 통해 처리하여 에러임을 나타냈다. Allocation resource 는 사용한 메모리마다 그 사용 가치가 모두 떨어졌으면 palloc_get_page(0)를 통해 배분된 메모리는 palloc_free_page 로, calloc 으로 배분된 메모리는 free 로 deallocation 하였다.

---- 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"?

parent 인 thread 에서 child list 를 관리하는데, exec 에서 preocess_exec()까지 해주고 그 중간 과정인 load 가 성공적으로 부여되기 전까지 loadlock 을 걸어 새 executable 이 load 를

완료하도록 해 준다. load 성공 여부는 parent 에서 관리되는 loadsema, loadlock, loadcond 를 사용하여 race condition 에 돌입하는 것을 막았다. Loadsema 의 경우는 1 로 initialize 되어 exec 할 때 down 으로 시작한다. exec 에서 down 이 되었다면, load 에서 up 을 시켜 다른 곳에서 exec 를 통해 sema_down 이 성사되기 전까지 다른 load 를 막는다. 이렇게 진행된 후에 lock 을 얻을 수 있는 기회를 부여하여, success 전달과 cond_signal 을 안전히 완수하도록 loadlock 의 lock 을 잠그고 작업이 끝나면 연다. 이를 통해 load 의 success/failure status 가 안전히 전달될 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?

thread 구조체 안의 waitsema 를 통해 조절한다. Wait 은 부모 입장, exit 는 child 입장이므로 wait 입장에서는 current thread 의 waitsema 를 semadown, exit 입장에서는 parent 의 waitsema 를 semaup 한다. exit 전에는 waitsema 가 down 되어 exit_status 가 바로 return 되는 것을 막고, exit 가 된 경우 waitsema 가 다시 up 이 되므로 그때의 exit_status 를 return 이 가능해지게 된다. Parent 가 바로 빠져나가 child 가 orphan 이 되는 경우도 있을 수 있는데, 이 경우는 임의의 thread 가 exit 하기 전 child 를 살펴 보아 살아있는 thread 가 있는지를 확인을 하는 작업을 거치는 것으로 해결한다. 만약 부모 thread 가 exit 해야하는데 살아있는 thread 가 있다면, orphan 값을 flag 와 같이 0,1 로 나타내어 주는 것으로 이 thread 가 orphan 임을 나타낸다. 그 후 부모 thread 는 exit 한다. 후에 orphan 인 thread 에서 exit 요청이 발생한 경우, 원래 메모리 할당 해제는 부모 thread 담당이지만 orphan 은 그럴 부모가 없으므로 직접 자기 메모리를 해제시키고 exit 한다.

---- RATIONALE ----

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

본 프로젝트에서는 kernel address 에 영향이 갈 수 있는 모든 system call 을 차단하였다. 그 이유는 당연하게도 User 의 행동이 kernel 에 영향을 미치게 된다면 시스템 자체가 망가져버릴 수 있기 때문이다. 이는 차단해야만 하는 일이다.

>> B10: What advantages or disadvantages can you see to your design
>> for file descriptors?
Advantages
1) Open 된 file 에 대한 management 가 쉬워진다.
2) filesys 에서 이미 구현된 함수들을 이용하여 이를 사용하면 file 을 열고 닫기 수월한데, 이것을 thread 에서 사용할 수 있는 매개체 역할을 한다.
Disadvantages.
1) 어쨌든 OS 가 열고있는 file 전체에 대해 manage 하므로, 용량을 많이 차지할 수 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?
바꾸지 않았다.
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? +--------------------------+