CS 330
PROJECT 2: USER PROGRAMS
DESIGN DOCUMENT

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

TEAM 17
Paul Kim 20150138 <paulkim3151@gmail.com>
Sangwoo Kim 20150144 <woodmac@kaist.ac.kr.>

---- 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.

https://github.com/ryantimwilson/Pintos-Project-2/blob/master/src/userprog/syscall.c (For getting reference of loading synchronization)

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.

We didn't declared any new data structure for argument passing. Argument passing is handled by parsing the whole input string in start_process method in process.c

---- 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?

We read argument strings twice, for counting argc and total size of stack we will need,which is called len. Then as putting copy argument strings into stack, we copy its address in appropriate place in stack, which is equal to rounddown(esp -len, 4) - 4*argc.
Our implementation appears after load function, which sets up stack and allocate some pages. So if there is any risk of overflowing stack page, load function will fail and avoid to put any arguments in stack.

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

strtok() method provide by default C library does not change input string itself, which is quite inconvenient way for parsing string. strtok_r() is hence useful while we are using the method in iteration statement.

>> 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.

Arguments handling takes long time. So if we use shell, we use less time in shell so that can improve performance of system.
And if we use shell, ill argument which can cause error will not fail kernel. So system becomes more concrete.

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

```
/* Process Loading state */
#define NOT_LOAD 0
#define LOAD_FAIL -1
#define LOAD_SUCESS 1

/* Process control blcok for process management */
struct process
{
   pid_t pid;                              /* Process identifier. */
   tid_t tid;                              /* Thread identifier that process
running on. */
   char name[16];                          /* Program Name. */

   int load;                         /* indicate process loading state */
   int exit_status;                  /* contains process exit status */
   int fd_num;                       /* For file descriptor allocation */

   struct process *parent;           /* Pcb for parent process */
   struct list child_list;                 /* Child list of current process */
   struct list_elem elem;                  /* List element for process_list */
   struct list_elem elem_heir;             /* List element for child_list */

   struct list fd_list;                    /* List of file desciptor (file_noe)
*/

};
```

```
/* File node for file descriptor list */
struct file_node {
       int fd;                      // File desciprtor number
       struct file* file;           // File structure (filesys/file.h)
       struct list_elem elem;       // List element
};
```

process.c

```
static struct list process_list; // Process list of running process
```

thread.c

```
static struct list all_list; // Thread list of live thread
```

thread.h

```
struct thread
{
  …
   int exit_status;          /* Thread exit status */

#ifdef USERPROG
   /* Owned by userprog/process.c. */
   pid_t pid;                          /* Process identifier */
   uint32_t *pagedir;                  /* Page directory. */
#endif
  …
}
```

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

Every process has its own fd_num, which is increased by every single open syscall. So regardless of which file is opening, or whether it is already open or not, newly opened file is allocated new file descriptor number. File descriptor is unique within a process.

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

We check validity of a user-provided pointer.
First of all, to check whether pointer is in the kernel memory, we use is_kernel_vaddr() in threads/vaddr.h file. This prevents to access kernel memory.
Second, if the user access memory that is not in the page table member, it causes user page fault. This is handled using pagedir_get_page() function.

Both case cause page fault exception, so program will be terminated.

>> 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?

For full page of data:
Least possible number = 1, Greatest possible number = 4096
If the full page memory in user space perfectly matches to single page memory in kernel, kernel can validate in single pagedir_get_page() call and use memcpy() to copy data into kernel. In order hands, if the full page memory is not contiguous, the kernel have to check the validity of every user-provided memory pointer value. So this may causes 4096 inspections of the page table.
If we assumes the memory is contiguous, the greatest possible number is 2 if kernel virtual address is not the page head. In this cases, we need to allocate other page to copy memory.

For the 2 byte of data:
Least possible number = 1, Greatest possible number = 2
This is the similar case to full page of data. Normal situations, the first inspection is enough to handle memory copy. But to handle two distinct byte memory, kernel needs to check every memory using pagedir_get_page() calls.

To improve this situations, we can use get_user() function and put_user() function. This uses assembly codes to check validity of memory.

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

Unless process is init process, it has own parent. If a process is terminated by exit system call, the thread that runs the process will be exit and free every resource it uses except pcb block. pcb block is freed by waiting process if child process is not alive.

>> 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.

We check address in syscall handler so we can decide whether address is valid or not. This will reduce whole lot of repeating checking codes. If system call handler is called, it checks whether stack pointer, address included in argument, or new stack pointer after syscall handling is valid using validate_memory() method. This method will check certain address is in user space or not null pointer.

Moreover, there is still be bad cases that causes errors. The test cases “bad-read” for example, the program attempts to read kernel memory by derefferencing the NULL pointer. These attepts crase the program by exception handler in exception.c. During killing process, we free allocated resource (file descriptors) in process_exit. The rest of resource is free during process_wait() of parent process.

---- 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() function waits until state of newly executed program is changed. We use while loop. If executable successes of fails, it changes its state so parent process can handle afterward.

>> 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?

If wait(C) is called before C exits, P will yield itself until C is exit.
If wait(C) is called after C exits, it will not yield and continues handling. In wait(C) method, resource used by C, which is pcb is freed.
If P terminates before C, parent of C becomes NULL. Then C free its PCB block itself.
If P terminates after C, pcb block is not free, so it will remain in the memory until OS halts.

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

We uses method to verify the validity of user-provided pointer before dereferencing. We uses this method because firstly there is a convenient method in vaddr.h called is_kernel_vaddr() and accessing page table is more safe way to handle memory access in general.

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

Advantages:
1. Kernel can manage every file descriptor of certain user process
2. If we want to uses file descriptor of certain process, kernel doesn’t have to consider about other process at all.

Disadvantages:
1) To avoid writing on opened executable, we have to check all process’ fd_list which is definitely longer than situation that we make fd_list per whole system instead of per process.
2) We used list, so identifying file from file descriptor needs long searching time.

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

We used same approach, that is identity mapping. But there was no pcb block which states parent-child hierarchy, we made the pcb block system. This system was very useful for exit-wait system since original thread system has no hierarchy for it uses queue instead of tree.