```
                     +--------------------------+

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

---- GROUP ----

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

Anirudh Nallana <anallana@buffalo.edu>
Aishwarya Chand <achand3@buffalo.edu>
Priyanka Garg <pgarg2@buffalo.edu>

---- PRELIMINARIES ----

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

>> Describe briefly which parts of the assignment were implemented by
>> each member of your team. If some team members contributed significantly
>> more or less than others (e.g. 2x), indicate that here.
Anirudh Nallana - Argument Passing, Safe Memory Access
Aishwarya Chand - Denying Writes to Executables, Design Document
Priyanka Garg - System Call infrastructure, Implement System Calls

>> 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.
process.c
1. Synchronizes program termination by ensuring orderly exit using semaphore.
static struct semaphore userprog_exit_sema;

---- 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 parse the arguments by splitting the command line into tokens using strtok_r, then
>>we allocated space for each argument on the stack. For avoiding stack overflow, we
>>calculated the total memory required for the arguments and added padding if needed.
>>The arguments argc and argv were placed last and arguments were pushed onto the stack
>>from the end.

---- RATIONALE ----

>> A3: Why does Pintos implement strtok_r() but not strtok()?
>> The optional placeholder(save_ptr) in strtok_r() allows us to put the arguments string
>> somewhere that is reachable which can be helpful.

>> 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. The Unix approach is safer since the executable commands can be checked above the kernel and avoided if they can lead to kernel failure.
2. The Unix approach can be updated or modified easily, since it is a user program and is not directly dependent on the kernel.
3. The Unix approach also promotes efficiency, since all user-programs are only loaded when necessary, and can free kernel memory/resources by not getting booted/loaded with the kernel.

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.h

```
/* Struct for holding information about child threads in parent thread */
struct child_thread {
   int child_tid; /* Holds tid of child thread */
   bool is_exit; /* Holds whether child has exited or not. */
   int exit_code; /* Holds the error code of child */
   struct list_elem elem; /* list elem for struct child_threads inside parent
```

```
thread */
};

+ List of structures being used by thread

    struct list files;                /* List of files being used by thread */
    int next_fd;                      /* Holds the next file descriptor for files
of thread */
    struct thread* parent_thread;     /* Parent thread attached to child threads
*/
    struct list child_threads;        /* List of child threads inside parent
thread */
    struct lock lock_child;           /* Lock when child is used for parent */
    struct condition cond_child;      /* Condition when child is used for parent
*/
    int locked_child_tid;             /* Holds the tid of child for whom the
parent should be locked for */
    int exit_code;                    /* Holds exit error code of the thread */



process.c
1. Iterate through the child thread list of the current thread to locate the child
thread
struct list_elem* elem;
  struct child_thread* child;
  for (struct list_elem* e = list_begin(&(thread_current()->child_threads)); e !=
list_end(&(thread_current()->child_threads)); e = list_next(e)) {
    struct child_thread* temp_child = list_entry(e, struct child_thread, elem);
    if(temp_child->child_tid == child_tid) {
      thread_current()->locked_child_tid = child_tid;
      child = temp_child;
      elem = e;
    }
  }

syscall.c

/* A Mapping between struct file * and file descriptors */
struct mapping_file {
  int fd; // File Descriptor
  struct file* file_; // file struct pointer
  struct list_elem elem; // List elem to store onto the list files in thread
};


>> 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 descriptors are unique only with each process, and are given/mapped to files using the variable `next_fd` which is incremented upon each new file opened.

---- ALGORITHMS ----

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

```
case SYS_READ:
      validate_safe_memory(f_esp + 7);
      validate_safe_memory(*(f_esp + 6));
      if(*(f_esp + 5) == 0) {
        uint8_t* buffer = *(f_esp + 6);
        int buffer_size = *(f_esp + 7);
        for (int i = 0; i < buffer_size; i++) {
          buffer[i] = input_getc();
        }

        f->eax = buffer_size;
      }
      else {
        struct mapping_file* read_file = files_search(&thread_current()->files,
*(f_esp+5));
        if(read_file != NULL) {
          lock_acquire_filesys();
          f->eax = file_read_at(read_file->file_, *(f_esp + 6), *(f_esp + 7), 0);
          lock_release_filesys();
        }
        else
          f->eax = -1;
      }
```

+ SYS_READ is used to indentify the syscall to read (a buffer). input functions 'input_getc' or 'file_read_at' are used to read.

```
    case SYS_WRITE:
      validate_safe_memory(f_esp + 7);
      validate_safe_memory(*(f_esp + 6));
      if(*(f_esp+5) == 1) {
        putbuf(*(f_esp+6), *(f_esp+7));
        f->eax = *(f_esp + 7);
      }
      else {
        struct mapping_file* write_file = files_search(&thread_current()->files,
*(f_esp+5));
        if(write_file != NULL) {
          lock_acquire_filesys();
          f->eax = file_write_at(write_file->file_, *(f_esp + 6), *(f_esp + 7), 0);
          lock_release_filesys();
        }
        else
```

```
          f->eax = -1;
      }
```

+ SYS_WRITE is used to identify the syscall to write. output functions 'putbuf' or 'file_write_at' are used for writing
+ NOTE :- Proper synchronization is needed for both reading/writing. And a common lock 'lock_filesys' structure is used to achieve this.

>> 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?
4,096 bytes
If the whole chunk is in one continuous memory page and is already mapped, only one call to pagedir_get_page() is needed.
If each byte is on a different memory page then we would need one call to pagedir_get_page() for each page.

2 bytes
If both bytes are on the same memory page then just one call to pagedir_get_page() is enough.
If the bytes are on two different pages then two calls to pagedir_get_page() are required.

Improvement:
Current approach should be efficient but if we still want to make more improvements then if systems cache the results of page table lookups then we can reduce the need for future lookups.

>> B5: Briefly describe your implementation of the "wait" system call
>> and how it interacts with process termination.
```
case SYS_WAIT:
      validate_safe_memory(f_esp + 1);
      f->eax = process_wait(*(f_esp + 1));
```

+ The waiting method is implemented using `process_wait()` since the userprograms are implemented through processes. The process_wait function is critical for execution of userprograms since, they are used to wait/lock for execution(implemented using process_execute) which leads into process_exit noting the termination of the userprogram. The process_wait must lock until all the threads created by the userprogram finish execution.

>> 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 used an approach where the user addresses(which were provided) are checked right before dereferencing or when the system call is executed. We check the contents of the pointer being NULL, and also check if the address is not in the kernel context/space, and finally we also check the page containing the address to see if there is a bad pointer or even an empty pointer.

---- 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"?

>> To ensure that the exec system call returns only after the new executable has completed loading
>> We used a combination of thread synchronization mechanisms to ensure that the parent process waits for the
>>child process to load the executable successfully or fails before returning from the exec system call.

>> In our implementation each child thread is associated with a child_thread structure. The process_execute function ensures
>>the parent waits for the child to finish loading the executable. We have also used a condition variable and loc for
>>synchronization between the parent and child and the child thread keeps updating its loading status and
>> signals the parent through the conditions variable we have used.
>>The parent thread uses the cond_wait mechanism we have implemented to block until the child thread signals the completion of the load process. If the
>>loading fails as indicated by the status set in the child thread the exec system call explicitly returns -1. So in case if the loading
>>is successful the parent thread resumes execution and completes the exec system call

>> 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?

>>To avoid race conditions and ensure proper synchronization when the parent process (P) calls wait on the child process
>>we used a multiple things for eg -synchronization primitives including multiple locks for each child and a condition variable. We even store the tid of the child if the parent needs to wait for the child to finish execution before exiting.

>>when P calls wait(C) before C exits, it would result in an exception, and the process P exits with status -1
>>once C has completed execution.
>>When P waits after C exits - P waits for C to exit and then exits

>> While when P terminates before C exits, C will be exit as well.
>> We also ensured, the P does not terminate until all its children terminate first using lock to each child with a condition structure.

---- RATIONALE ----

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

>>We implemented access to user memory through a validation function that checks if the user pointers lie within valid memory
>>regions before the kernel can accesses them. So the function we have implemented use page table lookups to verify the pointer
>>and if any pointer is invalid it terminates the offending process.
>>We chose this approach because it's simple and effective and ensuresthat only valid memory is accessed

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

>>We believe that our design simple and we have tried to make it efficient
>>Further our design for file descriptors is straightforward and efficient.
>>By assigning unique file descriptors for each opened file we tried to reduce the chances for any race conditions.
>>In our implementation each process has its own file descriptor table so this will ensure that resources are correctly managed
>> and file descriptors are properly cleaned up when a process terminates.
>>So tis will minimizes resource leaks and ensures that file handling is streamlined contributing to better overall system performance.

>> 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 chose to retain the default identity mapping between thread IDs and process IDs .
>>We believe that this approach would simplify the design by leveraging the

inherent uniqueness of thread IDs
>>thus removing the need for additional management or translation.

