```
             +--------------------------+
             |         CS 2042          |
             | PROJECT 2: USER PROGRAMS |
             |     DESIGN DOCUMENT      |
             +--------------------------+
```

---- GROUP ----

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

S.A.I.M. Perera <irash.21@cse.mrt.ac.lk>

---- 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.
in userprog/process.c,

- `char *save_ptr;` -
 A pointer used with `strtok_r` to tokenize the `file_name` and extract the program's name. The purpose is to isolate the program name from the command-line arguments.

- `char* temp` -
 A temporary buffer used to store a copy of the `file_name`. This copy is made to tokenize the input string without modifying the original `file_name`. The purpose is to safely extract the program name.

- `int argc` -
An integer variable to count the number of command-line arguments. It represents the number of elements in the `argv` array.

- `char *argv[30]` -
An array of character pointers to store the command-line arguments. It is used to hold the individual arguments extracted from the `file_name`.

- `char *token, *save_ptr;` -
Pointers used in a loop to tokenize the `temp` variable and populate the `argv` array with individual command-line arguments. `token` represents the current argument, and `save_ptr` is used with `strtok_r`.

- `int cnt` -
An integer variable to iterate through the command-line arguments while setting up the stack.

- `uint32_t *ret_address[argc]` -
An array of pointers to store the addresses of the command-line arguments on the stack. These addresses are used to retrieve the arguments when the program starts executing.

- The code in the `setup_stack` function is responsible for setting up the stack for the newly created process. It involves pushing arguments, their addresses, and other required information onto the stack.

---- 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?

1. **`process_execute` Function:**
   - A copy of `file_name` is made into the `temp` buffer, which is a temporary storage for tokenization without altering the original `file_name`.
   - `strtok_r` is used to tokenize `temp` by space characters, extracting each argument.
   - The first token extracted represents the program name and is stored in `file_name`.
   - The extracted tokens are stored in the `argv` array, and `argc` keeps track of the number of arguments.
   - The first token in `argv` is the program name, and subsequent tokens are the command-line arguments.

2. **`setup_stack` Function:**
   - The function is responsible for setting up the stack for the new process.
   - It calculates the addresses for the arguments and places them in the right order on the stack.

To avoid overflowing the stack page, the code takes several precautions:
   - It calculates the required stack space for each argument, considering the length of the argument strings.

- It ensures that the stack pointer is word-aligned by adding padding if necessary.
- It pushes the null pointer to indicate the end of the argument list.
- It sets up the addresses of arguments in the right order on the stack.
- It calculates and pushes the argument count to the stack.

---- RATIONALE ----

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

Pintos implements `strtok_r()` instead of `strtok()` because the kernel in Pintos separates commands into command line (executable name) and arguments. Therefore, it needs to store the addresses of the arguments to access them later. `strtok_r()` allows the caller to provide a save_ptr (placeholder), ensuring the thread-safe separation and storage of arguments, which is crucial in Pintos' multi-threaded environment.

>> 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. **Modularity and Extensibility:**Command-line parsing in Unix-like systems is handled by the shell, which provides modularity and flexibility. A variety of shells with varying features and parsing capacities are available for users to select from. It is simpler to expand or alter the behavior of the shell without changing the kernel because to this division of responsibilities.

2. **User Control and Flexibility:** The Unix method gives users the ability to specify how commands are to be understood. The ability to build intricate command pipelines, scripts, and aliases enables users to automate repetitive processes and carry out more sophisticated procedures. System admins and power users especially benefit from this flexibility.

## 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.

in threads/thread.h,

```c
struct thread
{
```

```
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

    int64_t waketick;

    // Check whether the thread is exited or not.
    bool ex;

    // Parent of the thread.
    struct thread* parent;

    // Exit code for the thread process.
    int exit_code;

    // The list of files opened by the thread.
    struct list files;
    int fd_count;

    // List to hold all the child processes.
    struct list child_processes;

    int child_load_status;
    int child_exit_status;

    // The executable file of the current thread.
    struct file* exec_file;
    struct semaphore waitThread;

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
};
```

in userprog/syscall.h,

```c
// Structure to store information about open files
struct file_details {
  int fd;           // File descriptor
  struct file *cur_file; // Pointer to the file structure
  struct list_elem elem; // List element for the file list
};
```

>> 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 linked to open files within a single process. Each process maintains a distinct collection of file descriptors that are utilized to identify files that are open within that specific process. The operating system as a whole does not share these file descriptors; instead, they are unique to the active process.

Put another way, different file descriptors within a process exist, but different processes may have different open files referenced by the same file descriptor value. This separation allows each process to have its own independent file descriptor and open files, preventing conflicts and misunderstandings regarding open files between processes.

---- ALGORITHMS ----

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

The code for reading and writing user data from the kernel is implemented in the `syscall_handler` function. It is responsible for handling all system calls and executing the appropriate code for each system call. The code for reading and writing user data is implemented in the `SYS_READ` and `SYS_WRITE` cases of the switch statement.

>> 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 a full page of data, the minimum number of checks required is 1 when we immediately get a valid page head, and the maximum number can be 4096 when the data is non-contiguous, necessitating checks for each address. In the case of contiguous

data, the maximum is 2 when we encounter a kernel virtual address that is not a page head, requiring checks for the start and end pointers of the full page data.

For 2 bytes of data, the minimum number of checks is 1 when we receive a kernel virtual address with more than 2 bytes of space to the end of the page. The maximum number is 2 when the data is not contiguous or when we encounter a kernel virtual address only 1 byte away from the end of the page, prompting inspection for the other byte's location.

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

The `wait` system call is implemented in the `process_wait` function. It is responsible for waiting until the child process with the given `pid` exits and then returning the child's exit status. The function iterates through the list of child processes and checks whether the child process with the given `pid` has exited. If the child process has exited, the function returns the child's exit status. Otherwise, it waits until the child process exits and then returns the child's exit status.

>> 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.

Pintos, the operating system, employs a two-pronged strategy to mitigate bad user memory access:

1. **Preventive Checks:** Pintos initiates a series of checks through the `is_valid_ptr` function before validating memory accesses. This function scrutinizes several key aspects of a pointer, including:
    - Whether the pointer is NULL.
    - Whether it points to a valid user address.
    - Whether it has been correctly mapped in the process's page directory.

    For instance, Pintos starts by confirming the integrity of pointers pointing to the stack and function arguments while executing the "write" system call. Pintos takes the decisive step of stopping the impacted process if any of these preliminary tests are unsuccessful. Additionally, Pintos performs additional checks within

particular methods to verify pointers such as the buffer's start and end addresses before to use.

2. **Error Handling:** In scenarios where preventive measures fall short and errors persist, Pintos has a contingency plan in the form of the `page_fault` exception handler. This handler is responsible for scrutinizing the validity of the faulting address (`fault_addr`) using the `is_valid_ptr` function. If the address is found to be invalid, the process in question is promptly terminated.

    As an illustrative example, consider a case where a program attempts to access an address such as `0xC0000000`, an address deemed invalid. In this instance, the `page_fault` exception handler, aided by the `is_valid_ptr` function, identifies the issue, assigns a return status of -1 to the process, and subsequently terminates it.

In addition to the above, Pintos also employs a `lock_release_all` function to release all locks held by a process before terminating it. This function is called by the `page_fault` exception handler to ensure that all locks are released before the process is terminated.

---- 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"?

Using the `process_execute` function, the `exec` system call is implemented. Its job is to load the new executable and notify the thread calling `exec` of the success or failure of the load. Following the creation of a new thread for the new executable, the function waits for it to finish loading. The `load_status` variable in the `thread` struct is used to report back the load success or failure status to the thread that calls `exec`.

>> 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?

When the parent process P calls `wait(C)` before C exits, the parent process P waits until the child process C exits and then returns the child's exit status. When the child process C exits, it notifies the parent process P by setting the `child_exit_status` variable in the `thread` struct. The parent process P then returns the child's exit status.

When the parent process P calls `wait(C)` after C exits, the parent process P returns the child's exit status immediately. The child process C has already exited, so the parent process P does not need to wait for the child process C to exit.

When the parent process P terminates without waiting, before C exits, the child process C is orphaned. The child process C is then adopted by the `init` process, which waits until the child process C exits and then frees all resources associated with the child process C.

When the parent process P terminates without waiting, after C exits, the child process C is orphaned. The child process C is then adopted by the `init` process, which frees all resources associated with the child process C.

When the parent process P terminates without waiting, before C exits, the child process C is orphaned. The child process C is then adopted by the `init` process, which waits until the child process C exits and then frees all resources associated with the child process C.

When the parent process P terminates without waiting, after C exits, the child process C is orphaned. The child process C is then adopted by the `init` process, which frees all resources associated with the child process C.

---- RATIONALE ----

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

We chose to implement access to user memory from the kernel in the way that we did because it is the most efficient and effective way to do so. The `pagedir_get_page` function is used to retrieve the kernel virtual address corresponding to a user virtual address. This function is used to check whether the user virtual address is valid and whether it has been correctly mapped in the process's page directory. If the user virtual address is invalid or has not been correctly mapped, the process is terminated.

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

The advantages of our design for file descriptors are as follows:

1. **Simplicity:** Our design is simple and easy to understand. It is also easy to implement and maintain.

2. **Efficiency:** Our design is efficient because it uses a list to store the file descriptors. This allows for fast insertion and deletion of file descriptors.

The disadvantages of our design for file descriptors are as follows:

1. **Memory Usage:** Our design uses a list to store the file descriptors. This requires additional memory to store the list.

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

did not change the default tid_t to pid_t mapping.

## 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?