CS2043 - Operating Systems

PROJECT 2: USER PROGRAMS
DESIGN DOCUMENT

---- GROUP 49 ----

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

ANOSAN J 200040B <anosan.20@cse.mrt.ac.lk>
NANAYAKKARA P.T.D.A. 200411N <damikan.20@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.

Referred to the following resources:
https://oslab.kaist.ac.kr/pintosslides/?ckattempt=1
https://www.youtube.com/watch?v=sBFJwVeAwEk
https://web.stanford.edu/class/cs140/projects/pintos/pintos_3.html
https://github.com/codyjack/OS-pintos
https://github.com/wookayin/pintos

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.

```
struct thread {
	/* stores the number of ticks the thread should sleep */
	int64_t sleep_ticks;
};
```

```
/* list to store threads that are put to sleep in ascending order by sleep_ticks */
struct list sleep_list;
```

---- ALGORITHMS ----

A2:
Briefly describe how you implemented argument parsing

The stack is set up inside the setup_stack() function to implement argument parsing. The process execution provides the file name, which includes the command and arguments string. The first token of the string is used as the new thread's name, while the rest is used as the arguments string. This string is passed down to other functions, including start_process(), load(), and setup_stack(). The command name can always be retrieved from thread->name when needed. The argument string and the command name are copied in the setup stack, and alignment is added. The string is then scanned backwards to obtain each token, and its address is pushed into the page to generate argv[].

How do you arrange for the elements of argv[] to be in the right order?

The elements of argv[] are arranged correctly by scanning the argument string backwards. The first token obtained from the backwards scan is the last argument, while the previous one obtained is the first. The esp pointer is decreased while setting up the elements of argv[] so that the first argument is stored at the lowest memory address and the last argument at the highest memory address, thus ensuring that argv[] is in the right order.

How do you avoid overflowing the stack page?

Two approaches were considered to avoid overflowing the stack page: checking the validity of esp before every use, letting it fail, and handling it in the page fault exception. The latter approach was chosen, as checking the validity of esp before every use seemed too burdensome. In this approach, if the address is invalid, the running thread is terminated by calling exit(-1). This makes sense, as the process would be completed if it provided too many arguments, which would cause the stack to overflow. The implementation does not pre-count the required space; instead, it makes the necessary changes as it goes through the process, such as adding another argv element. This approach ensures that the stack page remains within its limits and prevents overflow.

---- RATIONALE ----

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

Pintos implements strtok_r() instead of strtok() because it requires a placeholder to keep track of the parsing process. The placeholder, save_ptr, is provided by the caller in strtok_r(), which allows the caller to keep track of the parsing process and continue parsing from where it left off. In Pintos, the kernel separates commands into the command line (executable name) and arguments, so it needs to keep track of the address of the arguments to reach them later. Using strtok_r(), Pintos can store the save_ptr in a place that it can easily access, allowing it to continue parsing the arguments string and maintaining the state of the parsing process.

A4:
<u>In Pintos, the kernel separates commands into an executable name and arguments. In Unix-like systems, the shell does this separation. Identify at least two advantages of the Unix approach.</u>

1. Flexibility: In the Unix approach, the shell acts as an interpreter, allowing users to execute multiple commands simultaneously, use pipes and redirections, and perform other advanced pre-processing operations. This allows for a more flexible and user-friendly command-line interface.

2. Portability: The separation of responsibilities between the shell and the kernel makes it easier to port the system to different architectures, as the shell can be easily replaced without affecting the underlying kernel. This makes the Unix approach more robust and adaptable to different computing environments.

SYSTEM CALLS
===========

---- DATA STRUCTURES ----

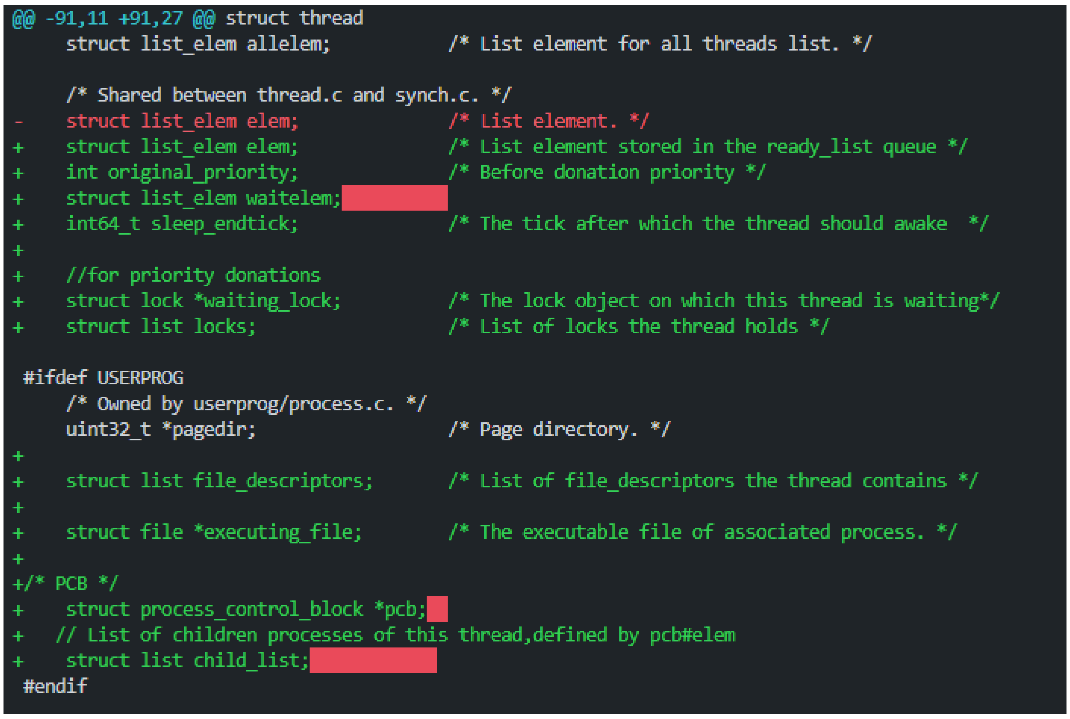
B1:
<u>Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.</u>

In Thread.h

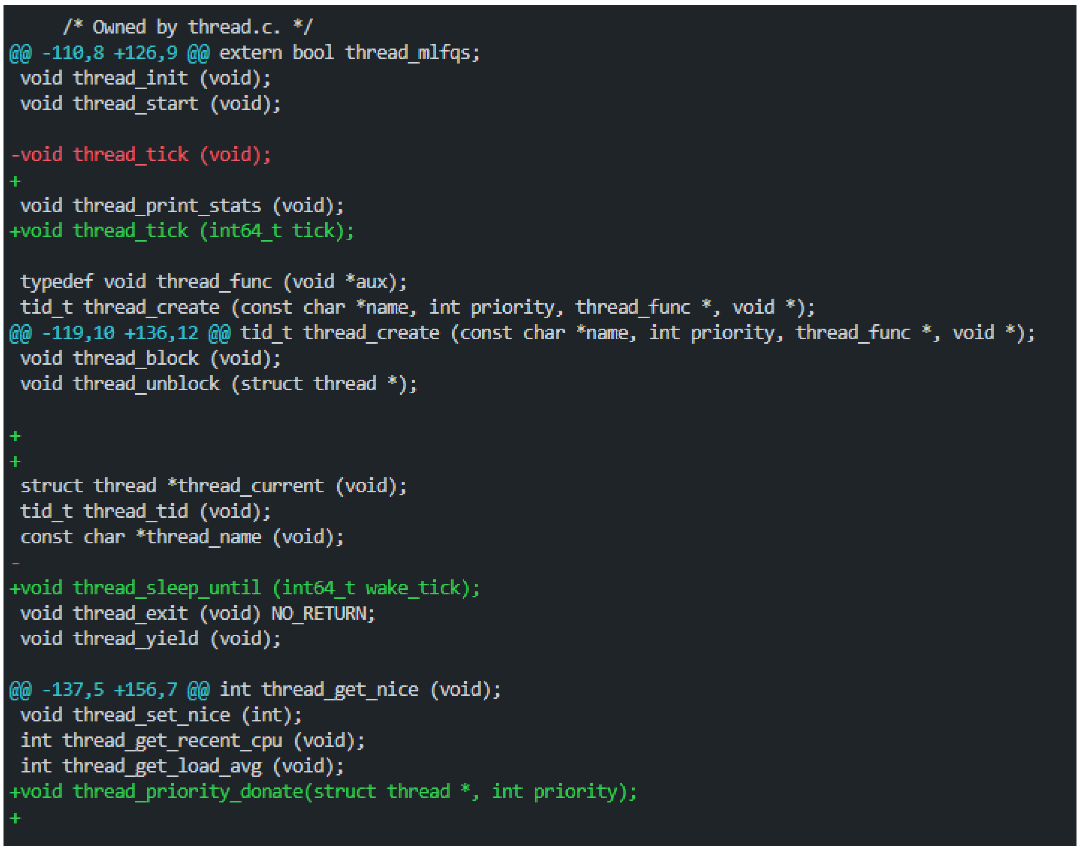
```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved 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 stored in the ready_list queue */
    int original_priority;              /* Before donation priority */
    struct list_elem waitelem;
    int64_t sleep_endtick;              /* The tick after which the thread should awake  */

    //for priority donations
    struct lock *waiting_lock;          /* The lock object on which this thread is waiting*/
    struct list locks;                /* List of locks the thread holds */

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

    struct list file_descriptors;     /* List of file_descriptors the thread contains */

    struct file *executing_file;      /* The executable file of the associated process. */

/* PCB */
    struct process_control_block *pcb;
   // List of children processes of this thread, defined by pcb#elem
    struct list child_list;
#endif

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

In syscall.c

```
/* File System Lock */
struct lock filesys_lock;

/* Find File Descriptor */
static struct file_desc* find_file_desc(struct thread *, int fd);
```



```
@@ -91,11 +91,27 @@ struct thread
     struct list_elem allelem;           /* List element for all threads list. */

     /* Shared between thread.c and synch.c. */
-    struct list_elem elem;              /* List element. */
+    struct list_elem elem;              /* List element stored in the ready_list queue */
+    int original_priority;              /* Before donation priority */
+    struct list_elem waitelem;
+    int64_t sleep_endtick;              /* The tick after which the thread should awake  */
+
+    //for priority donations
+    struct lock *waiting_lock;          /* The lock object on which this thread is waiting*/
+    struct list locks;                  /* List of locks the thread holds */

 #ifdef USERPROG
     /* Owned by userprog/process.c. */
     uint32_t *pagedir;                  /* Page directory. */
+
+    struct list file_descriptors;       /* List of file_descriptors the thread contains */
+
+    struct file *executing_file;        /* The executable file of associated process. */
+
+/* PCB */
+    struct process_control_block *pcb;
+   // List of children processes of this thread,defined by pcb#elem
+    struct list child_list;
 #endif
```



```
     /* Owned by thread.c. */
@@ -110,8 +126,9 @@ extern bool thread_mlfqs;
 void thread_init (void);
 void thread_start (void);

-void thread_tick (void);
+
 void thread_print_stats (void);
+void thread_tick (int64_t tick);

 typedef void thread_func (void *aux);
 tid_t thread_create (const char *name, int priority, thread_func *, void *);
@@ -119,10 +136,12 @@ tid_t thread_create (const char *name, int priority, thread_func *, void *);
 void thread_block (void);
 void thread_unblock (struct thread *);

+
+
 struct thread *thread_current (void);
 tid_t thread_tid (void);
 const char *thread_name (void);
-
+void thread_sleep_until (int64_t wake_tick);
 void thread_exit (void) NO_RETURN;
 void thread_yield (void);

@@ -137,5 +156,7 @@ int thread_get_nice (void);
 void thread_set_nice (int);
 int thread_get_recent_cpu (void);
 int thread_get_load_avg (void);
+void thread_priority_donate(struct thread *, int priority);
+
```

Process.c
+static void push_arguments (const char *[], int cnt, void **esp);

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

File descriptors in the implementation have a one-to-one mapping to each file opened through the system call. The file descriptors are unique within the entire operating system, not just within a single process. A list (struct list open_files) is maintained inside the kernel to keep track of the file descriptors and their corresponding open files, as every file access goes through the kernel.

---- ALGORITHMS ----

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

READING:
First, check if the buffer and buffer + size are valid pointers; if not, the process exits with a return value of -1. The file system lock (fs_lock) is then acquired. If the file descriptor (fd) is equal to STDOUT_FILENO, the lock is released, and -1 is returned. If the fd is equal to STDIN_FILENO, the keys are retrieved from the standard input, and the lock is released, and the function returns 0. In other cases, the open file is retrieved from the open_files list

using the fd number and file_read from the file system is used to read the file. The status of the file read operation is obtained, and the lock is released, and the status is returned.

WRITING
Checks if the buffer pointer is valid and acquires the fs_lock. If the fd equals STDIN_FILENO, the lock is released, and -1 is returned. If the fd is equal to STDOUT_FILENO, the buffer's content is printed to the console using putbuf. In other cases, the open file is retrieved from the open_files list using the fd number and file_write from the file system is used to write the buffer to the file. The status of the file write operation is obtained, and the lock is released, and the status is returned.

B4:
<u>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?</u>

A full page of data might result in the least number of inspections of the page table being 1, if the first call to pagedir_get_page returns a page head. The greatest number might be 4096 if the data is not contiguous and needs to be checked at each address. The least number of inspections for 2 bytes of data would be 1, if the kernel virtual address has more than 2 bytes of space to the end of the page. The greatest number would be 2, if the data is not contiguous or only 1 byte away from the end of the page. There is no room for improvement as per the code.

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

The "wait" system call is implemented using the process_wait function. A new struct called "child_status" represents the exit status of a child process. A list of child_status structs is added to the parent process's thread struct, representing all the children the parent owns. Each child process has a parent_id inside its struct to ensure that it can find its parent and set its status if the parent still exists.

A child_status struct is created and added to the list whenever a child process is created. The parent will wait (cond_wait) if the child has not exited. The child is responsible for setting its return status and waking up the parent. A monitor is introduced in the parent's struct to avoid race conditions. Before checking or setting the status, both the parent and child must acquire the monitor.

If the parent is signaled or sees that the child has exited (checked using the thread_get_by_id function), it will start to check the status. If the child calls the exit-syscall to exit, a boolean signal is set to indicate that the exit-syscall was called, and the child's exit status is set in the corresponding child_status struct in the parent's children list. If the child is terminated by the kernel, the boolean signal remains false, and the parent sees and understands that the child was terminated by the kernel.

If the parent terminates early, the list and all the structs in it are freed. The child will then find out that the parent has already exited and give up setting the status, continuing its 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.

The strategy adopted to handle bad user memory access is to check the validity of pointers before using them. A function named "is_valid_ptr" is written to validate the user address, check if it's NULL, and whether it's mapped in the process's page directory. This function is used before accessing the user memory in the system calls, such as the "write" system call. Before entering the write function, the pointers to the system call number, the three arguments, and the buffer starting and ending pointers are checked using "is_valid_ptr". If any of these pointers is invalid, the process is terminated.

In case of errors still happen, the handling is done in the page_fault exception handler. The fault address is validated using the "is_valid_ptr" function, and if it's not a valid address, the process return status is set to -1, and the process is terminated.

For example, in the case of a bad-jump2-test (*(int *)0xC0000000 = 42;), where the user program is trying to write to an invalid address, there is no way to prevent it. In this case, when inside the page_fault exception handler, the invalid address is detected by calling "is_valid_ptr". The process return status is set to -1, and the process is terminated.

In conclusion, the strategy adopted for managing these issues is to validate the user memory pointers before accessing them and handle errors in the page_fault exception handler by checking the validity of the fault address. This ensures that all temporary resources are freed and avoids obscuring the primary function of the code with error handling.

---- 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"?

The "exec" system call ensures that the new executable has completed loading by having the child thread responsible for setting the child_load_status in the parent's thread, which is accessible through the parent_id field and the thread_get_by_id function. The child thread will set the child_load_status in the parent's thread to either success or failure, depending on the result of loading the new executable. The parent thread, before creating the child thread, the parent thread sets the child_load_status to 0, representing the initial state where nothing has happened yet. After creating the child thread, the parent acquires the monitor and waits until the child_load_status is no longer 0, which indicates that the child thread has finished executing and has set the child_load_status to either success or failure. The parent can then retrieve the child_load_status from the parent's thread to determine if the "exec" system call was successful.

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 ensure proper synchronization and avoid race conditions when P calls wait(C) before C exits, the parent process P acquires the monitor and waits until the child process C exits. This is achieved by checking the child thread's existence through a function (thread_get_by_id) which checks all-thread-list. The parent process then retrieves the child's exit status.

When P calls wait(C) after C exits, the parent process acquires the monitor, finds out that C already exists, and checks its exit status directly.

In case P terminates without waiting before C exits, the list inside P will be freed and the lock will be released. Since no one will wait for a signal except the parent, the condition does not need to be signalled. When C tries to set its status and finds out that the parent has exited, it will ignore it and continue to execute.

When P terminates after C exits, the same thing happens to P, which frees all the resources P has.

---- 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 user memory access from the kernel in this manner because we validated it before using it, and we needed to fully understand the second approach,

which deals with page fault inside exceptions and how putuser() and getuser() could be used. As a result, we needed to be more confident in the second approach to select it.
B10:
What advantages or disadvantages can you see to your design for file descriptors?

Advantages:

- Minimized thread-struct space: The design of having file descriptors stored in the thread struct minimizes the memory consumption of the thread struct, as compared to having a separate data structure for file descriptors.
- Increased kernel control: The kernel is aware of all the open files and has more control over their manipulation, allowing for greater flexibility in managing open files.

Disadvantages:

- Increased kernel memory consumption: The design consumes kernel memory, and if a user program opens a large number of files, it could crash the kernel.
- Difficulty in file descriptor inheritance: Implementing the inheritance of open files from a parent to a child requires additional effort as compared to other designs that handle this automatically.

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 left it alone. It is practical and reasonable.