50007.1 - Laboratory - Task 2 - Design Document

Robbie Buxton, Jordan Hall, Bartłomiej Cieślar and Oliver Killane

08/11/21

Group

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

Name
Bartłomiej Cieślar
Jordan Hall
Oliver Killane
Robert Buxton

Preliminaries

Latex Formatting

To make this document easier to read, we have implemented a consistent style:

C4 (5 marks)

This is the question's text...

...it can go over several lines!
We also have syntax highlighting for:

- Functions such as **schedule** or **init_thread**.
- Variables such as **num_ready_threads** or **thread_mlfqs**.
- Structures of type definitions such as **thread** or **ready_queue**.
- Constants and values set in define such as **BINARY_POINT** and **PRI_MAX**.
- Files such as **thread.c** and **fixed-point.h**.

We also include code directly from this repository, so the contents, line numbers and file titles all match their respective places.

syscall.c

```
98  static int get_user(const uint8_t *uaddr);
99  static bool put_user(uint8_t *udst, uint8_t byte);
100
101 static inline struct file *get_file(unsigned fd);
102
103 void syscall_init(void)
104 {
105         /* Register the syscall handler for interrupts. */
106         intr_register_int(0x30, 3, INTR_ON, syscall_handler, "syscall");
107
108         /* Register syscall handlers. */
109         syscall_handlers[SYS_HALT] = halt;
110         syscall_handlers[SYS_EXIT] = exit;
```

This makes our document quicker and easier to read and mark.

Argument Passing

Data Structures

A1 (1 mark)

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 roughly 25 words.

Process info struct

```
31
32 /* The initial size of the bitmap of open files */
33 #define OPEN_FILES_SIZE 4
34
35 /* Default exit status on error */
```

The purpose of this struct is to pass in the necessary arguments to the **start_process()**. The arguments are as follows:

- The new stack for the process, complete with the arguments for the **main()** call.
- The size of the new stack; used to set the **esp** register.
- The manager of the child process for the waiting system.

Algorithms

A2 (2 marks)

How does your argument parsing code avoid overflowing the user's stack page? What are the efficiency considerations of your approach?

A stack page overflow is avoided by checking for the amount of free space left on the page every time we want to add a new string argument. This check accounts for the word alignment of the input strings, additional pointer in the **argv** array, a null entry at the end of **argv**, the size of **argc** and the size of the call return pointer. Additionally, there is a compile-time check for the size of the page in case it cannot accommodate even the function name. The process creation fails in case any of those size checks fail.

As for the efficiency of our approach, by setting up the arguments to the call in **process_execute()**, we avoid an unnecessary copy of that data into a new stack page, which allows for directly installing the page into the user page directory right after it is passed into the **start_process()**. Moreover, by setting up the **argv** argument during tokenization in one go, we avoid an additional pass for figuring out the pointer values to the specific string arguments. We set it up, along with the **argc** and the call return pointer, at the bottom of the page, because we do not know the size of the actual string values that are being set up at the same time at the top of the page and are growing in the opposite direction. After that, we use **strmov()** to safely move the data to its position just below the string data to avoid problems in case the move destination overlaps with the initial place where **argv** is set up. An optimization we could have performed would be to place the tokenized helper string with command data directly inside the page to save a **malloc()** and copying the data for **argv**; however, that would have required many additional moves and complexity in case of multiple spaces between the arguments to **main()** or cause the stack to store redundant data, so we did not follow through with that approach.

Rationale

A3 (2 marks)

*Pintos does not implement **strtok()** because it is not thread safe. Explain the problem with **strtok()** and how **strtok_r()** avoids this issue.*

The function **strtok()** is not thread-safe because it uses global data. It has a static pointer which it reuses when the function is called again. This means that you cannot parse 2 strings in parallel and therefore it is not thread safe. The function **strtok_r()** on the other hand is re-entrant ('_r') and it uses a locally-defined pointer. This means that no auxiliary data is globally defined and multiple threads can freely work on different strings in parallel.

A4 (3 marks)

In Pintos, the kernel separates commands into an executable name and arguments. In Unix-like systems, the shell does this separation. Identify three advantages of the Unix approach.

- **Separation of concerns**
  If argument passing was implemented in the kernel, changes to it could result in bugs and features of the kernel breaking, and if there is a bug in the argument passing, it could cause the kernel and hence the whole system to crash. This coupling is avoided by separating argument parsing to a program built specifically to excel at that task (a shell).
- **Extensibility of the shell**
  Having the shell doing argument parsing makes it easier to extend the functionalities that the user receives in the end. This is because it allows for implementing other features that would not necessarily fit in the kernel that wants to generalize its functionalities. One example is command aliasing, relative paths for scrubbing through the file system as a user, custom commands not packaged in separate programs, or splitting a process call into multiple ones with pipelining between them.
- **Modifiability at runtime**
  To update/change the kernel requires an OS reboot from the updated binary, this is not possible/convenient on some systems (for example cloud service infrastructure). By parsing arguments through a shell program, the shell can be shut down, updated and restarted without taking the whole system down.

System Calls

Data Structures

B1 (6 marks)

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 roughly 25 words.

thread.h

```
95  struct thread {
96          /* Owned by thread.c. */
97          tid_t tid; /* Thread identifier. */
```

```
98          enum thread_status status; /* Thread state. */
99          char name[16]; /* Name (for debugging purposes). */
100         uint8_t *stack; /* Saved stack pointer. */
101         int8_t priority; /* Thread's current computed priority */
102         union {
103                 struct {
104                         int8_t nice; /* Niceness ranges from 20 to -20 */
105                         fixed32 recent_cpu; /* Measure of cpu usage */
106                 };
107                 struct {
108                         /* Owned by donation.c */
109                         int8_t base_priority; /* Thread's base priority */
110                         struct lock *donee; /* The lock that receives the thread's
                                ↪ priority */
111                         struct list_elem donorelem; /* Used in donee's list of donors
                                ↪  */
112                         struct list donors; /* Locks donating their priority to the
                                ↪ thread */
113                 };
114         };
115         struct list_elem allelem; /* List element for all threads list. */
116         /* Shared between thread.c and synch.c. */
117         struct list_elem elem; /* List element. */
118
119 #ifdef USERPROG
120         /* Owned by userprog/process.c. */
121 #ifndef NDEBUG
122         bool may_page_fault; /* For debugging kernel */
123 #endif
124         uint32_t *pagedir; /* Page directory. */
125         struct vector open_files; /* Vector of open file structs */
126         struct list children; /* List of child processes */
127         struct child_manager *parent; /* Struct managing the child process */
128 #ifndef VM
129         struct file *exec_file; /* The program file the thread is running */
130 #else
131         struct vector mmapings; /* Maps mmap ids to malloced lists of pages is uses
                ↪ */
132         struct list exec_file_mmapings; /* list of executable pages' user_mmaps. */
133 #endif
134 #endif
```

thread has been altered to include the following fields:

may_page_fault	A flag for whether the kernel should panic during a page fault or assume that it is testing a user-provided pointer.
open_files	A vector containing pointers to the current open files.
open_files_bitmap	A bitmap to determine which indices in the **open_files** vector contain open files (**true**), and which are free spaces (**false**).
children	A **list** of all child processes spawned by the thread, that can be waited on.
parent	A pointer to the parent process manager (the child manager of the thread that created this process through **exec**).
exec_file	The program file that the process's code was loaded from, and that is kept open with writes denied for the duration of the process's runtime.

process.h

```
12 /* Process identifier. (duplicated code from src/user/syscall.h) */
13 typedef int pid_t;
14 #define PID_ERROR ((pid_t)-1)
15
16 /* Child process manager. */
17 struct child_manager {
18         bool release; /* Whether the other side should free it. 0 by default */
19         int exit_status; /* The status for the wait command. -1 by default */
```

The **child_manager** struct stores information on a child of a thread. It is allocated on the heap, rather than in the **thread** struct so that its existence is not dependent on that of the child or parent. This ensures that the exit codes of the children are preserved for the parent if it is running. It includes the following fields:

release	A boolean stating if the other side needs to free the struct from the heap. It is set to **false** by default.
exit_status	The exit code of the child process. It is set to **-1** by default.
wait_sema	A **semaphore** used for waiting on a child to exit. It is set to **0** by default.
elem	A **list_elem** used for the list of children in the parent.
tid	The tid of the child process. It is also used in the check for if the child has managed to load its code properly.

process.c

```
30 #endif
31
32 /* The initial size of the bitmap of open files */
33 #define OPEN_FILES_SIZE 4
34
35 /* Default exit status on error */
```

The **process_info** struct contains the setup information when creating a new process. It includes these attributes:

stack_template	A pointer to the new stack page template
stack_size	The size of the new stack for the child process
parent	A **child_manager** managing the new child process

syscall.c

```
30  */
```

A type for all syscall handlers. In order to reduce duplication all syscall handlers are declared with this typedef.

To reduce code duplication and avoid fetching unnecessary data from the interrupt frame, syscall handlers take a pointer to the **eax** member of the interrupt frame (to write result), and a pointer to the first of 3 possible arguments.

We use a **GET_ARG** macro to get each argument in order:

syscall.c

```
19 #include "vm/mmap.h"
20 #include "threads/malloc.h"
21
22 #define MAP_FAILED (-1)
23
24 #endif
```

An example of this is in the **seek** syscall handler:

syscall.c

```
426         if (!is_user_vaddr(buffer + size - 1))
427                 thread_exit();
428
429         if (fd < FD_START && fd != STDOUT_FILENO) {
```

We also have a return macro set the return value. By using a macro instead of an assignment, the code is made more readable as it is clear where the return value is set.

syscall.c

```
26 #define FILE_FAILED (-1)
```

syscall.c

```
44 /* intermediate page buffer for reading and writing to files */
45 int8_t *read_write_buffer;
46 struct lock read_write_buffer_lock;
47
48 /* Global lock for accessing files */
49 struct lock filesys_lock;
50
51 /* Lock file on entry */
52 void filesys_enter(void)
53 {
54         lock_acquire(&filesys_lock);
55 }
56
57 /* Unlock file on exit */
```

The prototypes for the syscall handling functions.

syscall.c

```
59 {
```

An array of syscall handling functions; upon initialization of the syscall handling, this array gets populated so that the index of a given handler is the syscall reference number.

syscall.h

```
6 void filesys_enter(void);
```

A lock used to ensure mutually exclusive accesses to the file system.

Algorithms

B2 (2 marks)

Describe how your code ensures safe memory access of user provided data from within the kernel.
We use several functions for verifying the safety of pointers to user provided data.

syscall.c

```
146                 thread_exit();
147
148         sys_handle *handler = syscall_handlers[syscall_ref];
149         handler(&f->eax, f->esp + sizeof(void *));
150 }
151
152 /* Checks if the buffer BUFFER of size SIZE can be safely accessed
153  * by the kernel.
154  */
155 bool check_user_buffer(const void *buffer_, unsigned size)
156 {
157         const uint8_t *buffer = buffer_;
158         if (!size)
159                 return true;
```

syscall.c

```
129         lock_init(&filesys_lock);
130
131         /* Palloc the buffer used to copy data */
132         read_write_buffer = palloc_get_page(0);
133         if (!read_write_buffer)
134                 PANIC("Failed to palloc a read/write buffer for filesys");
135         lock_init(&read_write_buffer_lock);
136 }
137
138 static void syscall_handler(struct intr_frame *f)
139 {
140         /* Get the args from the stack. */
141         if (!check_user_buffer(f->esp, sizeof(int)))
142                 thread_exit();
143         int syscall_ref = *(int *)(f->esp);
```

syscall.c

```
112         syscall_handlers[SYS_WAIT] = wait;
113         syscall_handlers[SYS_CREATE] = create;
114         syscall_handlers[SYS_REMOVE] = remove;
115         syscall_handlers[SYS_OPEN] = open;
116         syscall_handlers[SYS_FILESIZE] = filesize;
117         syscall_handlers[SYS_READ] = read;
118         syscall_handlers[SYS_WRITE] = write;
119         syscall_handlers[SYS_SEEK] = seek;
120         syscall_handlers[SYS_TELL] = tell;
121         syscall_handlers[SYS_CLOSE] = close;
122
123 #ifdef VM
124         syscall_handlers[SYS_MMAP] = mmap;
125         syscall_handlers[SYS_MUNMAP] = munmap;
126 #endif
```

These functions check that the maximum memory location of the provided data is a userspace virtual address (using **is_user_vaddr**). Then we check that the data is in valid pages by using:

- **get_user()(ptr)**
  Returns −1 if the address not is readable by the current process; otherwise, it returns the byte value under that pointer.
- **put_user()(ptr, val)**
  Returns if it has managed to write the **val** to the place **ptr** points to.

We have modified **page_fault()** handler to facilitate this behaviour, by checking if the page fault has been caused by the kernel in **put_user()** or **get_user()**.

exception.c

```
155
156 #endif
157
158         /* Turn interrupts back on (they were only off so that we could
159          * be assured of reading CR2 before it changed).
160          */
161         intr_enable();
```

B3 (3 marks)

*Suppose that we choose to verify user provided pointers by validating them before use (i.e. using the first method described in the spec). What is the least and the greatest possible number of inspections of the pagetable (e.g. calls to **pagedir_get_page()**) that would need to be made in the following cases?*

You must briefly explain the checking tactic you would use and how it applies to each case to generate your answers.

To check a user provided pointer to a buffer of a given size we complete the following steps:

1. **Check the largest address is in user memory**
   We check the highest memory address (a pointer to buffer plus size) is smaller than **PHYS_BASE** using **is_user_addr()**. Because the largest memory address in the buffer is in user virtual memory so is the rest of the buffer.
2. **Check if buffer's pointer is in the process's page directory**
   We check the buffer pointer is in a page in the page directory of the user using **pagedir_get_user()**.
3. **Check the start of all subsequent pages containing parts of the buffer**
   We can get the distance from the buffer pointer to the end of the page using:
   PGSIZE - **ptr** % **PGSIZE**

If the size is larger than this, we can skip to the start of this page. We use **pagedir_get_user()** to verify the page is in the current process's page directory.

We proceed in increments of **PGSIZE**, checking that each page is in the user's page directory until we reach the end of the buffer.

As a result, in every page that contains part of the buffer we will have called **pagedir_get_user()** once.

For null-terminated strings we do the following:

1. **Check first character**
   Check the first character is valid using **is_user_vaddr()** and **pagedir_get_user()**.

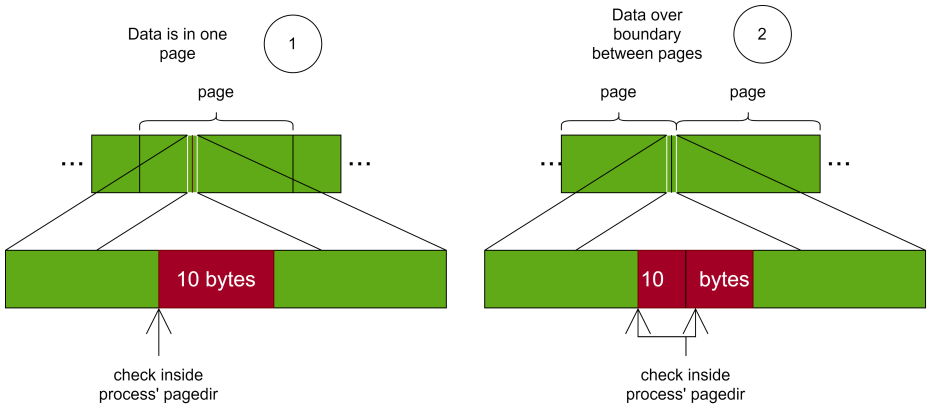
2. **Iterate over each character**
   We can interate over each character, up to the final **NULL**.

   For each character we must check the address is valid using **is_user_vaddr()**. If the address is at the start of a page we can use **pagedir_get_user()** to check that the page is in the current process's page directory.

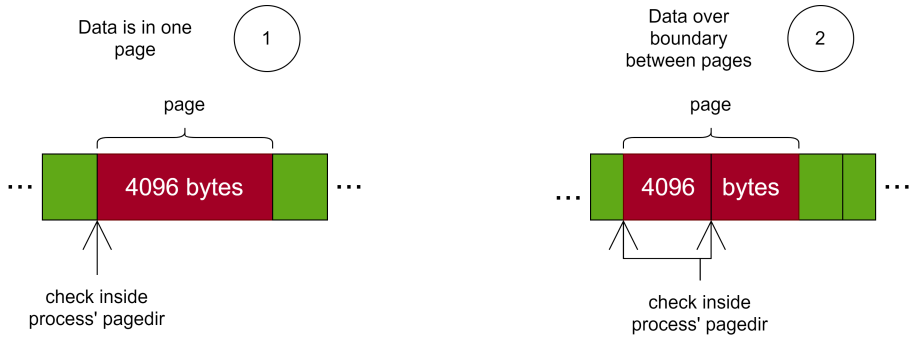
As a result for a given size we can calculate the number of calls to **pagedir_get_user()** as:

$$\frac{\textbf{size}}{\textbf{PGSIZE}} \text{ or } \frac{\textbf{size}}{\textbf{PGSIZE}} + 1$$

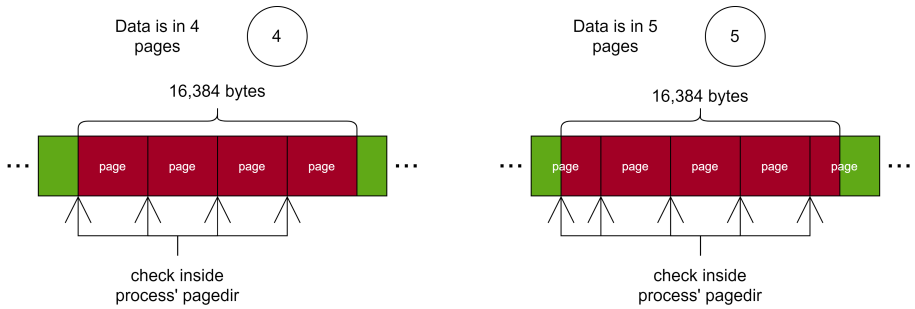
(a) *A system call that passes the kernel a pointer to 10 bytes of user data.*



(b) *A system call that passes the kernel a pointer to a full page (4,096 bytes) of user data.*



(c) *A system call that passes the kernel a pointer to 4 full pages (16,384 bytes) of user data*



B4 (2 marks)

When an error is detected during a system call handler, how do you ensure that all temporarily allocated resources (locks, buffers, etc.) are freed?

As **process_exit()** is always called from **thread_exit()** we exit the thread and deallocate all held resources there.

Our deallocation is split into 4 parts:

1. **Freeing the children managers**
   This is done for all threads, not just processes. A thread may have been created and waited on by a process - e.g main thread.

   The thread or process contains a **list** of **child_manager**. By using the atomic **test_and_set()** function, the release flag can be set synchronously without a race condition. This flag informs the child processes that the parent has terminated. They can then free their **child_manager** on termination.

2. **Freeing the parent manager**
   This is only done by process threads, so this happens inside the if that checks if there is any page directory installed for the thread. Like for freeing of children, this is done synchronously using the release flag and an atomic **test_and_set()**. The semaphore for the possibly waiting parent is upped beforehand.

3. **Closing open files**
   The process's open files are contained in a vector (**open_files**) with a bitmap (**open_files_bitmap**) determining which vector indexes contain open files.

   The bitmap is traversed, and for each **true** value, the corresponding open file is closed.

4. **Closing process executable**
   The process executable is kept open for the duration of the process's runtime to prevent writes to it. When the process terminates, this file is closed and the writes are allowed back again.

Filesystem Lock

The only other resource that syscalls use is a global filesystem lock, which gets released everywhere before the thread has a chance to exit, and does not need to be freed since it is global.

B5 (8 marks)

Describe your implementation of the "wait" system call and how it interacts with process termination for both the parent and child.

syscall.c

```
267  * Note that the process PID may have terminated before this function is run.
268  * Returns the exit status of the process.
269  */
270 void wait(uint32_t *ret, const void *args)
271 {
272         GET_ARG(args, pid_t, pid);
273         RETURN(ret, process_wait(pid));
274 }
```

The wait system call calls and returns the result of **process_wait()**.

process_wait() traverses the **children** list to get the **child_manager** struct of child. If there is no **child_manager** of the correct **tid** cannot be found, it returns **-1**.

The **child_manager** is allocated separately to both the parent and child processes. This ensures that if the child terminates, the parent can still access its exit code (for a **wait** syscall). Likewise, if the parent exits, this is visible to the child through **release**.

The **wait_sema** member of **child_manager** is used to block the parent thread as it waits on a child. The semaphore is initialised with a counter value of **0** and the parent calls **sema_down()**. **sema_up()** is called on the semaphore by the child on exit, which unblocks the parent.

If the parent or child accesses their **child_manager** and the other has terminated, **release** will be true. Release is set by the atomic **test_and_set()** operation to prevent a race condition. If this is the case, the remaining process can free the **child_manager**. In the case of the child, it does not need to place its exit status, or **sema_up()** the **wait_sema**. In the case of the parent, the **child_manager** must be removed from its list of children.

Synchronisation

B6 (2 marks)

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 syscall returns the output from a call to **process_execute()** using the infrastructure already provided in pintos. This is by setting the **eax** in the interrupt frame.

For the **process_execute()** the challenge is that the loading happens in another thread. This means some form of synchronization is needed in order to communicate if the child has failed to load. This is done by reusing the semaphore in the child manager struct in order to first wait for the child to set its **tid** there. This is set to **-1** in case that the loading fails. After the child has performed the load it sets its parent manager **tid** value and ups the semaphore there to inform the parent thread that it has performed that action.

B7 (5 marks)

Consider parent process P with child process C. How do you ensure proper synchronization and avoid race conditions when:

(i) *P calls wait(C) before C exits?*

(ii) *P calls wait(C) after C exits?*

(iii) *P terminates, without waiting, before C exits?*

(iv) *P terminates, without waiting, after C exits?*

Additionally, how do you ensure that all resources are freed regardless of the above case?

We handle the synchronization of the parent-child relationship by using an externally allocated struct called **child_manager**. Inside this struct there is a semaphore that is used by a parent process for waiting for a child to exit. The child calls **sema_up** when it exits and the parent calls **sema_down** when it waits for a child. This way the child can call **sema_up** upon exit, regardless of the parent's status, and the parent can call **sema_down** whenever it wants to wait on a child process. If the child has exited before the parent waits on it, the **sema_down** does not block the parent, otherwise the parent becomes blocked until the child call **sema_up** when the child exits.

The second part of synchronization is done to only free the child manager once both the parent and the child process exit. This is done by using a **bts** instruction that can save the value of a boolean into the carry flag and set it atomically. This is wrapped into a **test_and_set()** function which is called on a release flag inside the child manager. The release flag is set to false by default. If this is the first time **test_and_set()** is called on it, the return value from the call will be false, so the manager will not be freed. However, during the second call **test_and_set()** will return true indicating that the manager should actually be released. This will be done by either the parent or the child process, depending on which one is the latter to exit or wait. Since the setting and getting of the boolean is atomic we prevent a race condition here.

As for the order of specific cases asked for in the question:

(i) P calls wait(C) before C exits:

(a) P finds the child manager for C

(b) P downs the semaphore in the child manager of C

(c) C ups the semaphore in the parent manager, unblocking P

(d) C calls **test_and_set()** and does nothing since it returned false

(e) C exits

(f) P unblocks as a result of C upping the semaphore

(g) P calls **test_and_set()** and frees the child manager since it returned true

(h) P returns from **process_wait()**

Here, the order after P is unblocked is an example, since the interleaving of the two processes there can be arbitrary, but everything is synchronized in all cases.

(ii) P calls wait(C) after C exits

(a) C ups the semaphore in the parent manager

(b) C calls **test_and_set()** and does nothing since it returned false

(c) C exits

(d) P finds the child manager for C

(e) P downs the semaphore in the child manager of C

(f) P immediately passes through since the counter in the semaphore was 1 from C upping it

(g) P calls **test_and_set()** and frees the child manager since it returned true

(h) P returns from **process_wait()**

(iii) P terminates, without waiting, before C exits

(a) P gets to C while freeing its children

(b) P calls **test_and_set()** and does nothing since it returned false

(c) P exits

(d) C ups the semaphore in the parent manager

(e) C calls **test_and_set()** and frees the child manager since it returned true

(f) C exits

(iv) P terminates, without waiting, after C exits

(a) C ups the semaphore in the parent manager

(b) C calls **test_and_set()** and does nothing since it returned false

(c) C exits

(d) P gets to C while freeing its children

(e) P calls **test_and_set()** and frees the child manager since it returned true

(f) P exits

Rationale

B8 (2 marks)

Why did you choose to implement safe access of user memory from the kernel in the way that you did?

We chose the second approach suggested in the spec. This is because we will need to handle it this way for the third part of the project: the implementation of virtual memory.

It also makes the code for the buffer checking shorter and faster because we do not need to enter the user's page directory and check its contents, and can take advantage the MMU (Memory Management Unit) of the processor to check virtual addresses are valid more quickly.

B9 (2 marks)

What advantages and disadvantages can you see to your design for file descriptors?

Our file management design is focused on speed and memory efficiency. It uses resizable vectors to keep track of currently open files and a bitmap of the fields in a vector that contains open files in order to have the possibility to be able to reuse those entries once they get vacated by a file being closed. This ensures memory utilization equal to the largest number of files ever opened by the process. This can be expected to be similar to the number of files opened at any given time throughout the lifetime of the process. It also allows for a quick conversion between file descriptors and file struct pointers.

The disadvantage of this approach is that it requires 3 additional **malloc()**s and associated **free**s for allocating our vector of files and bitmap. This means our solution has higher memory usage than a list based implementation. We decided increased memory usage was well worth the increased speed of accessing files.

Another approach we considered that would have possibly saved us some file creation and deletion latency would be the usage of a hashmap. However, we decided against it, since opening the files is not a time-critical action that the user will perform multiple times. Moreover, since the number of files per-process is not expected to be high in such a small system. Therefore iterating through the bitmap is not going to add much overhead to the **open** and **close** syscalls.

A different improvement would be to replace the bitmap we use for reusing fields in the vector with an interval tree. However, using that structure would have added additional latency for smaller number of open files, as well as additional memory usage. Thus, we decided not to use it either, despite better asymptotic complexity.