CS330 Project 2: USER PROGRAMS

---- TEAM ----

Team 10

>> Fill in the names, email addresses and contributions of your team members.

Tuan Manh Lai <laituan245@kaist.ac.kr> (Contribution1 = 50%)

Nham Van Le <levn@kaist.ac.kr> (Contribution2 = 50%)

>> Specify how many tokens your team will use.

0 tokens

---- 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.

http://linux.die.net/man/3/strtok_r

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 did not add any new data structure for implementing argument passing.

---- 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 implemented argument passing in the *start_process()* function. More specifically, after the call to the function *load()*, if there is no problem, we will then start to push arguments on the stack (Obviously, if the loading fails, we won't have to do argument passing). First, we extract all the arguments from the command line in left-to-right order (using the function *strtok_r()*) and push them on the stack in order. Second, we round the stack pointer down to a multiple of 4 for best performance. After that, we push a null pointer sentinel and the address of each argument, on the stack. These are the elements of argv. Even though, we do not actually store the addresses of the arguments while we push them on the stack, but C string must end with \0 and the *strlen()* function can be used to find the length of a string, we can easily retrieve the address of each argument. Finally, we push argv (the address of argv[0]), argc (the number of arguments), and a fake "return address" (which is just 0x00000000).

We try to avoid overflowing by putting some limit on the size of the arguments.

---- RATIONALE ----

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

The man page says that "The *strtok()* function uses a static buffer while parsing, so it's not thread safe. Use *strtok_r()* if this matters to you". In other words, if there are two or more threads use the function *strtok()* at the same time, the final outcome can be incorrect. Because in our

system, there can be many threads (or processes) executing concurrently. That's why we need to use the thread-safe version, which is *strtok_r()*.

>> 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.

Two advantages:

+ Using the Unix approach can **lower the risk of crashing the entire system**. For example, while the shell is doing the separation, if there is some problem and the shell crashes, then it probably won't be a big problem. On the other hand, if the kernel crashes, the whole system will also crash. This is a bad thing.

+ Note that the shell can totally do the separation by itself without the help of the kernel (This is basically a string-processing problem). Probably the time it takes to do the separation should be the same whether the kernel does the separation or the shell does the separation. And so we should let the shell does the separation so that we can **reduce the time and memory used by the kernel** (Time and memory of kernel is really valuable and we should use them wisely)**.**

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.

For operating on files:

struct file_info{
  int fd;
  tid_t tid;
  struct file * file_ptr;
  struct list_elem elem;
}
	Each instance of this struct represents a mapping between a file descriptor number and an open file.

struct lock fd_lock;	This lock is used to prevent race conditions when multiple processes want to use the function *allocate_fd()* at the same

	time.
struct semaphore filesys_sema;	This semaphore is used to ensure that only one process at a time is executing the file system code (mutual exclusion).
struct list file_info_list;	The list that we will use to keep track of the info of all open files.

For executing and waiting processes:

struct pack{
  char * argv;
  struct semaphore * sema;
  bool * loaded;
}
	This struct contains all the variables we need to pass to the *start_process()* function.

struct exit_info{
  tid_t tid;
  int status;
  struct list_elem elem;
}
	To be able to return the exit status of a thread.

struct relationship_info{
  tid_t parent_tid;
  tid_t child_tid;
  struct list_elem elem;
}
	To be able to find a child process of a given process

static struct list exit_info_list	To manage all the instances of exit_info struct in the system
static struct list relationship_list	To manage all the instances of relationship_info struct in the system

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

Each instance of the struct *file_info* represents an association between a file descriptor number and an open file. In order to keep track of all the associations in the system, we used the list *file_info_list*, which basically consists of all the instances of the struct *file_info*. Whenever some process opens a file successfully, the size of the list will increase by 1. Similarly, whenever some process closes a file, the size of the list will decrease by 1.

In our current design, file descriptors are unique within the entire OS. That's why we have just one list for keeping track of all the mappings. If we chose to make the file descriptors to be unique just within a single process, then we may need to have one list for each process. Note

that whenever a process opens some file, we will call the function *allocate_fd()* to get a new file descriptor. The definition of the function is shown below.

```
static int
allocate_fd (void)
{
  static int next_fd = 2;
  int fd;

  lock_acquire (&fd_lock);
  fd = next_fd++;
  lock_release (&fd_lock);

  return fd;
}
```

Finally, it may be worth to mention that each instance of the struct *file_info* has a variable *tid*, which denotes the id of the process that the file descriptor belongs to. In this way, we can easily find the set of all file descriptors that belong to some particular process (if we want).

---- ALGORITHMS ----

>> B3: Describe your code for reading and writing user data from the kernel.
The most important thing to do is to check the validity of the stack pointer in the system calls. We have 2 functions to do that. The first one is to check the validity of one specific pointer address.

```
bool is_valid (void * pointer) {
  if (pointer == NULL)
    return false;
  if (!is_user_vaddr(pointer))
    return false;
  if (pagedir_get_page(thread_current()->pagedir, pointer) == NULL)
    return false;
  return true;
}
```

In this function, we check if this address is whether NULL, not in the user address space, or is in an unmapped area.

The second one is to check the validity of the arguments passed to the system calls. We will make use of the first one in the second one

```
bool are_args_locations_valid (void * esp, int argc) {
  void * tmpptr = esp;
  int i;
  for (i = 0; i < argc; i++) {
    tmpptr = tmpptr + 4;
    if (!is_valid(tmpptr))
      return false;
  }
  return true;
}
```

In this code, we will check the validity of the arguments one by one (hence we need to pass to it the number of arguments).
If any of these tests fails, we will simply terminate the process.
If everything is good, we can then proceed to use the value of the pointers in the system calls to read and write the data from the kernel.

>> 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?

If a system call causes 4096 bytes of data to be copied from user space into the kernel, then the least number of inspections of the page table will be 1. This is because if the first byte we check turns out to be not valid (it is unmapped) then we do not have to do any other checking (we usually should immediately terminate the offending process).

If the bytes of the data are not contiguous, the greatest number will be 4096 because we may need to inspect every single byte of the data. When the bytes are contiguous, the greatest number will be 2 because we just need to check the first byte and the last byte of the data. And actually for all the system calls we have to implement that actually can cause the copying of data, the bytes of the data to be copied will be all contiguous *(e.g., read, write, …)* and so for our case the maximum number of calls to pagedir_get_page() will be 2.

For the same reasoning, if 2 bytes of data are to be copied, the least number will be 1 and the greatest possible number of inspections will be 2 (whether the two bytes are contiguous or not).

We don’t think there is any other room for improvement in these numbers.

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

In our implementation of the “wait” system call, we will first check the validity of the argument pointer, and if it is not valid, we terminate the process immediately. Otherwise, we simply call the *process_wait()* function implemented in the file process.c. Now we will describe the implementation of the *process_wait()* function.

Whenever some process is about to exit (either normally or abnormally), we will store its exit status in an instance of the struct *exit_info* and insert the instance into the list *exit_info_list* (mentioned in B1) if its parent process is still running. In addition, we will “up” some special semaphore associated with the process (stored in the related instance of the struct *relationship_info* and was initialized to be 0).

When the *process_wait()* function is called, we will first check if the pid passed to the function is indeed the pid of some child process of the current process (by looking at the list *relationship_list*). If it is not, we immediately return -1. Otherwise, we will “down” the semaphore in the instance of the struct *relationship_info* that represents the relationship between the current process and the child process. If the child process has not terminated (hence hasn’t upped the semaphore), the current process will have to wait until the child process exits. After the waiting, we then actually look at the list *exit_info_list* to retrieve the exit status of the child process and return it. Note also that the appropriate instances of the structs exit_info and relationship_info will also be freed during the execution of *process_wait()* function so that there won’t be any memory leak..

>> 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.

As mentioned before, in order to read or write user data, we defined two new functions, which are the functions *is_valid()* and *are_args_locations_valid()*. Given some user-provided pointer, the function *is_valid()* checks whether the pointer is valid (i.e, the pointer should not be NULL and it should not point to unmapped virtual memory nor the kernel virtual address space). The function *are_args_locations_valid()* is implemented on top of the function *is_valid()* and we use the function for validating all the arguments passed to some system call. By using these two functions, we can avoid obscuring the primary function of code in a morass of error-handling.

For example, let's consider the code for handling the "write" system call. By using the two functions above, we first check the validity of the esp pointer and the three argument pointers. If there is something wrong, we will terminate the offending process by calling the function *terminate_process()*. Otherwise, we then check the validity of the buffer beginning pointer and the buffer ending pointer. If everything is ok, the code after this is mainly for handling the logic of the write system call (there won't be any code for handling errors due to bad pointers in this part). Note that at the beginning, there are just about 10 lines of code for error-handling.

In our current design, before allocating resources for handling the system call of some process, we actually try to check the validity of all the user-provided pointers. In this way, even if there is some error due to bad pointers, there isn't actually any resource for us to free. In addition, if a process should be terminated at some point due to some error, either the function *terminate_process()* or *exit()* in the file syscall.c will be called eventually. So in both of these functions, we actually have code for closing all the files that the process has opened and also freeing all the instances of struct *file_info* associated with the process. And actually there is no other type of resources we need to free in these two functions if some process is terminated due to an error.

By using the above strategy, we can be sure that if some error occurs, we won't "leak" resources. Also, by using this strategy, we were able to pass the test "multi-oom".

---- 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 new struct "pack" mentioned in B1, we are able to have the "loaded" status when creating a process.
Because of that, the parent process can have access to the loaded status and if the child process is failed to load, it can know that and return -1.

```
tid_t
process_execute (const char *argv)
{
  struct pack my_pack;
  struct semaphore child_load_sema;
  bool loaded = false;
  ...
  sema_init(&child_load_sema, 0);
  my_pack.sema = &child_load_sema;
  my_pack.argv = argv_copy;
  my_pack.loaded = &loaded;
  my_pack.parent_tid = thread_current()->tid;
```

```
  tid = thread_create (file_name, PRI_DEFAULT, start_process, &my_pack);
  if (tid == TID_ERROR)
    palloc_free_page (argv_copy);
  else {
   sema_down(&child_load_sema);
   if (!loaded)
     return -1;
  }
  return tid;
}
```

```
static void
start_process (void * aux)
{
  struct pack * my_pack = (struct pack *) aux;
  void * argv = my_pack->argv;
  struct semaphore * sema = my_pack->sema;
  bool * loaded = my_pack->loaded;
  ...
   *loaded = true;
  asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
  NOT_REACHED ();
}
```

Here we see that if the process is failed to start for any reason, the “loaded” variable will not be set to true, so process_execute will return -1.

>> 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?

Our answer to B5 actually already explained how we ensure proper synchronization and avoid race conditions when P calls wait(C) before C exits. The basic idea is when a child process terminates, it has to up some special semaphore; and when a parent process wants to wait for some child process, it has to down that semaphore. In our design, whether the child ups the semaphore first or the parent downs the semaphore first, it doesn’t matter. In both of these cases, the resources that are not needed anymore will be freed near the end of the *process_wait()* function.

In case when P terminates without waiting before C exits, the resources will be freed when C is about to exit. More specifically, when some process C is exiting, inside the function *process_exit()*, we first check if the parent P of C is still running, If not, then we won’t save the exit status of C into the list *exit_info_list.*

Also note that when a process C exits (whether its parent is still running or not), we will free all the instances of the struct relationship_info that represent relationships that C is a parent process of some child process near the end of the function *process_exit()*. We have tried to think of all the places where memory leaks can occur and handled all that cases.

---- RATIONALE ----

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

In the pintos manual, two ways of handling invalid pointers are mentioned. We chose to do the first way (verify the validity of a user-provided pointer before dereferencing it) because it is way simpler to implement and the logic is easier to understand. We acknowledge the fact that besides the potential performance problem, by using this approach, we will also need to put the sanity check at multiple places, making the code a bit more redundant.

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

Advantages:

+ This design is simple. As mentioned before, in our design, since file descriptors are unique within the entire OS, we just need to have one list for keeping track of all the mappings between file descriptors and open files. If we chose to make the file descriptors to be unique just within a single process, then we may need to have one list for each process and this may be more complicated.

Disadvantages:

+ Overflow can occur. Since we represent each file descriptor using the C int type, if the system call *open()* is called too many times, then at some point, something wrong can occur (e.g, there can be negative file descriptor numbers, or the same file descriptor may be used for more than one open file).
+ When a process calls some system call that is related to working with files (e.g., read, write, close, …), we will have to scan the entire big list of the file descriptors of all processes in the system. This can be slow and this can be avoided if we make file descriptors unique within just a single process and for each process we have a list of its own file descriptors.

>> 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 keep the default mapping. We did not change this because in this lab, we only need to deal with single-threaded processes, so we keep the mapping for the sake of simplicity. If we need to deal with a multi-threaded system, obviously we need to change the mapping to something similar to how we handle the mapping between file descriptors and file pointers.