CS330 Project 2: USER PROGRAMS DESIGN DOCUMENT

---- TEAM ----

>> Team name.

Team 19

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

Hyeonwoo Kang <kanglib@kaist.ac.kr> (50)
Young-geol Jo <rangewing@kaist.ac.kr> (50)

>> Specify how many tokens your team will use.

0

---- PRELIMINARIES ----

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

NONE

>> 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.

NONE

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.

NONE HERE

---- 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?

First we copied file_name into a newly allocated page, s, and tokenized it with strtok_r(). Then we added each token to argv[] before the user stack got prepared. They should be put into the stack in the reversed order, thus we reversed the order during the iteration over argv[] and strlcpy().
To avoid stack overflow, we used only safe string functions, such as strnlen() and strlcpy().

---- RATIONALE ----

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

It is because strtok_r() is reentrant and thread-safe, whereas strtok() not. strtok() stores its save pointer to a global variable, thus it may be overwritten by other threads. Therefore, it is not thread-safe and cannot be used in multiprogramming without some synchronizations.

>> 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.

- Kernel does not have to deal with string pointers, which is dangerous, in the Unix approach. Thus It is safer.
- Shell can handle very complex command lines that include quotes, I/O redirections, or newline characters better.

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.

syscall.h

```
/* Process identifier type. */
typedef int pid_t;
```

- pid_t is a type for process identifiers.

syscall.c

```
/* External lock for base file system. */
struct lock fs_lock;
```

- fs_lock is an external lock for base file system used by system calls. It ensures that only one process can access the file system at the same time.

thread.h

- struct child (new structure)

  ```
  #ifdef USERPROG
  /* Child process information.
     Used in thread structures. */
  struct child {
    tid_t tid;                          /* Thread identifier. */
    int status;                         /* Exit status. */
    struct semaphore sema;               /* Synchronization. */
    struct list_elem elem;               /* List element. */
  };
  #endif
  ```

  - child is a structure to store information of the child processes. It is used for proper executing and waiting of child processes.

- struct thread (members added)

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

```
    struct file *exe;                  /* Handle of executable. */
    struct file **file;                /* File descriptor mappings. */
    int file_n;                        /* Maximum used fds. */
    int file_alloc_n;                  /* Maximum allocated fds. */
    int exit_code;                     /* Exit status. */
    bool is_failed;                    /* Flag used for process_execute(). */
    struct semaphore sema1;            /* Synchronization. */
    struct semaphore sema2;            /* Synchronization. */
    struct thread *parent;             /* Parent process. */
    struct list child_list;            /* Child process information. */
#endif
```

- uint32_t pagedir: page directory of the user process.
- struct file *exe: The file pointer of the executable of the process.
- struct file **file: The file descriptor mapping table for the process. It maps the file descriptors to the corresponding file pointers.
- int file_n: The maximum file descriptor number used. It is incremented by one when a file is opened.
- int file_alloc_n: The maximum number of files that can be opened. It is used to manage the dynamic allocation of the file table.
- int exit_code: Indicator for the exit status of the process.
- bool is_failed: True if the loading succeeded in process_execute().
- struct semaphore sema1, sema2: Semaphores to ensure the proper return of process_execute() after the completion of load().
- struct thread *parent: A pointer to its parent process.
- struct list child_list: A list of its child processes.

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

When a file is opened, its pointer is mapped to the file descriptor with the resizable mapping table of thread. We can get the pointer of file by accessing the table with its file descriptor.
Our file descriptors are unique within a single process, because we use a thread-local mapping table.

---- ALGORITHMS ----

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

First it checks the user address whether it is valid. It should not access the invalid address, such as NULL pointer, kernel address or an unmapped page. Then it acquires the physical address using pagedir_get_page(). Finally, it reads from or write to the physical memory.

>> 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?

In both cases, our implementation inspects the page table once or twice to check the both ends of the memory. There is no room for improvement, because we have to check these two addresses necessarily.

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

When a wait system call occurs, it calls process_wait(). In process_wait(), it finds the corresponding child process from its child_list first, and downs the semaphore of the child.
When it acquires the semaphore to be down, it waits for its termination. It is because that the semaphore goes up when the process exits at process_exit(). After that, process_wait() can continue its execution to remove the child, free the memory and return the exit status.

```
int
process_wait(tid_t child_tid)
{
  struct list *list;
  struct list_elem *e;

  list = &thread_current()->child_list;
  for (e = list_begin(list); e != list_end(list); e = list_next(e)) {
    struct child *c = list_entry(e, struct child, elem);
    if (c->tid == child_tid) {
      int status;
      sema_down(&c->sema);
      status = c->status;
      list_remove(e);
      free(c);
      return status;
    }
  }
  return -1;
}
```

```
/* Free the current process's resources. */
void
process_exit (void)
{
  struct thread *curr = thread_current ();
  ... // THIS PART IS OMITTED
      if (curr->parent) {
        list = &curr->parent->child_list;
        for (e = list_begin(list); e != list_end(list); e = list_next(e)) {
          struct child *c = list_entry(e, struct child, elem);
          if (c->tid == curr->tid) {
            c->status = curr->exit_code;
            sema_up(&c->sema);
            break;
          }
        }
      }
      ... // THIS PART IS OMITTED
    }
}
```

>> 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 first filtered the arguments by checking whether their addresses are valid, which means they are pointing mapped user memory and not a NULL pointer. Then we can protect the invalid memory access by terminating the caller thread. A page fault also kills the thread.

When a thread is killed, thread_exit() will be called, which calls process_exit() also. In this case and the case that process is terminated, process_exit() frees all allocated resources and return the default exit status, -1.

For example, when a write system call is triggered with an invalid user address as a buffer, such like NULL, it calls handle_exit(-1) to terminate the process.

---- 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"?

It uses two semaphores to ensure it. First, process_execute() waits for sema1 after creating the thread. sema1 goes up in start_process(), when the result of loading is determined. If load() failed, it sets is_failed true and waits for sema2.
After sema1 goes up, process_execute() continues and check is_failed of the thread with making sema2 up. After that, if the loading failed (is_failed), process_execute() returns TID_ERROR and start_process() calls thread_exit(). Otherwise, process_execute() allocates the child process and adds it to the child_list.
Therefore, we can ensure the correct operation of the exec system call.

process_execute() will return the result of the process creation to the exec system call handler. The handler will set the eax register of interrupt frame with the result, in order to pass back the result.

>> 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 P calls wait(C) before C exits, then it waits for the termination of C, as we have discussed in B5. It uses a semaphore to ensure the proper operation, which waits for the termination of C. In this case, the resources of C will be freed on its exit.

When P calls wait(C) after C exits, then there is no C in child_list. Therefore, process_wait will carry out nothing in actually. In this case, C has been already freed when it exited.

When P terminates without waiting before C exits, P frees all child processes in child_list.

When P terminates without waiting after C exits, then there will not be C in child_list. C has been already freed when it exited.

---- RATIONALE ----

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

that you did?

We chose the way to verify the validity of a pointer, because the simplicity of the implementation.

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

It can hide kernel pointer so that it could improve the security of the kernel. Moreover, this design is simple and intuitive. However it may result in memory inefficiency when some processes open too many files.

>> 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 did not changed it.

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 two problems in it, too easy or too hard? Did it take too long or too little time?

multi-oom test was too hard to pass.

>> Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design?

Probably.

>> 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?

It would be great if some hints for multi-oom are provided.

>> Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects?

>> Any other comments?