PROJECT 2: User Program

DESIGN DOCUMENT

Project Name: [**PintOS**]

December 17, 2016

[GROUP]

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

```
Asmaa        Osama        <sh.asmosama@hotmail.com>
Shereen      Gamal        <eng.shereenaboeldhab@hotmail.com>
Mayar        Abd-Elaziz   <eng.mayar2018@outlook.com>
```

[PRELIMINARIES]

>> 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.

- https://github.com/ This reference has helped as in some points in argument passing.

[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.

No new struct, struct member, typedef or enumeration.

[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?

```
static void
start_process (void *file_name_)
{
      ……
 /* Step (1): Separate the executable name from the arguments */
  char *save_ptr;
  file_name = strtok_r((char *) file_name, " ", &save_ptr);
  success = load (file_name, &if_.eip, &if_.esp, &save_ptr);
      ……
}

bool
load (const char *file_name, void (**eip) (void), void **esp, char **save_ptr)
{
      ……

  /* Set up stack. */
  if (!setup_stack (esp,file_name, save_ptr)){
    goto done;
  }
      ……
}
```

```
static bool
setup_stack (void **esp, const char* file_name, char** save_ptr)
{

/* Step (2): allocate memory with DEFAULT_ARGV_SIZE = 2 as an initial size */

/* Step (3): loop over the arguments of the command line and for each argument

   1. reduce the esp with its size.
   2. store the address at which the argument should be push onto stack in argv.
   3. update number of arguments.
   4. push the argument itself onto the stack.

*/

/* Step (4): Store 0 at the last index of argv */

/* Step (5): calculate number of bytes needed to alignment and push zero at each
extra byte */

/* Step (6): push zero as a remark for end of variable and then push the saved
addresses onto the stack in reverse order */

/* Step (7): push the address of the first argument in stack */

/* Step (8): push number of arguments */

/* Step (9): push fake return address e.g 0 */

/* Step (10): free the memory used to save addresses of arguments */

}
```

Avoiding Overflow is done by checking if (esp + size) is a valid pointer or not if it's not a valid pointer we invoke exit(-1);

[RATIONALE]

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

Because strtok() uses global data. So, it's unsafe in threaded programs such as kernel. Instead we use strtok_r() and provides it with the save_ptr in order not to be changed.

>> A4: 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.

Pintos approach will expose the kernel to be error prone quickly because there is no guarantee that the executable file exists or a valid one and this will lead to kernel panic. Whereas Handling this check in the shell will pay the system less consequences by shell crash or alert in case of handling that error.

[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.

struct thread
  {
       ……

    /* file list for all files owned by the Process/Thread */
    struct list file_list;

    /* file descriptor for each file in the file list */
    int fd;

    /* executable file */
    struct file *executable;

      /*--------------------------------- */

    struct list children ;                /* list of children */

    struct list_elem child_elem ;         /* to insert in all and in children */

    struct semaphore sema ;               /* to wait for child to be fully loaded
*/

    struct semaphore wait ;               /* to "wait" for a child to die */

    bool loaded_success;                  /* if process is loaded successfully or
not*/

    int status_pro ;                     /* status of process */

    struct thread *parent ;

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

#endif
       ……
  };

/* to keep info about the thread after its death
```

```
  basically it's used to provide a tid-exit_status map */
struct process
{
  tid_t pid ;          /* tid of the thread that runs the process*/
  bool dead            /* tells if a process is dead or not */
  bool waited_upon ;   /* tells if the parent has already waited upon that process
*/
  int exit_status ;    /* keeps the exit_status even after death of the thread, -1
if it hasn't died yet */
  struct list_elem child_elem ;    /* to add to my paren's list of children */
};
/* An open file. */
struct file
{
    struct inode *inode;        /* File's inode. */
    off_t pos;                  /* Current position. */
    bool deny_write;            /* Has file_deny_write() been called? */
    /* List Element for all files owned by a Process/Thread */
    struct list_elem file_elem;

    /* fd descriptor to distinguish between different of process/thread */
    int fd;
};
```

>> 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 thread has a variable fd initialized with 2 and whenever an open system call occurs we assign the fd value of the thread to file->fd and increment the fd value of the thread.

No, file descriptors are unique for only a single not for the entire OS.

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

```
int
read(int fd, void *buffer, unsigned size)
{
        lock_acquire(&filesys_syscalls_lock);

        // read from the keyboard using input_getc()
        if(fd == STDIN_FILENO)
        {
                int i;
                uint8_t *buffer_ptr = (uint8_t *) buffer;
                for(i = 0; i < size; i++)
                {
                        buffer_ptr[i] = input_getc();
                }

                lock_release(&filesys_syscalls_lock);
                return size;
        }

        ……
}
```

In read syscall:
First of all, we acquire a lock. Then, we check if the fd (sent as a param) equals to STDIN_FILENO which is (0). If so read from the kernel into the sent buffer using input_getc() method and after finishing the process release the lock and return the size of the buffer.

```
int
write(int fd, const void *buffer, unsigned size)
{
        lock_acquire(&filesys_syscalls_lock);

        // write to the console using putbuf()
        if(fd == STDOUT_FILENO){

                putbuf(buffer, size);
                lock_release(&filesys_syscalls_lock);
                return size;
        }
        ……
}
```

In write syscall:
First of all, we acquire a lock. Then, we check if the fd (sent as a param) equals to STDOUT_FILENO which is (1). If so write the sent buffer to the kernel using putbuf() method and after finishing the process release the lock and return the size of the buffer.

>> 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 case of not using pagedir_get_page() in validation of pointers:
The minimum number of times this method invoked is 1 from install_page() method.
In this case, all the data are sored in a single page.
The maximum number of times this method invoked is 4k (4096).
In this case, the data are stored as bytes across 4k (4096) pages one for each byte.

** In case of using pagedir_get_page() in validation of pointers:
The minimum number of times this method invoked is 1 and the max will be twice the maximum number of pages.

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

* The coordination between process termination and having the parent wait
   for that process is resolved by having struct process, which basicallly keeps a tid-exit_status pair, along with a bool to tell if the process is dead or not.
   Each thread keeps a list of children "processes" .
   the adv. of having a separate struct is to keep the exit_status when the thread has already died. When the exit syscall is invoked, the exit status in the process is set
   to the status sent. By default exit_status is (-1); if the thread dies and the exit_status is still -1, this means that it was killed by the kernel.
   In the wait call : first, it's checked that that tid belongs to a child of the current_thread by traversing the children list, then it's checked if that child has been waited upon,
   finally, if the thread is dead, the exit_status kept in the process is returned, else the parent waits on a semaphore in the child's thread struct initialized to zero,

thid sema is upped when the child is about to die, then the exitstatus is returned .

>> 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 address of the esp sent is first checked when a system call is invoked, and also the addresses of the arguments of the call if it has,
  also if the arguments itself is a pointer, it's checked to make sure that the argument is in the user space.
  This all happens before calling the function corresponding to the system call. At this stage, no resources would have been reserved so no freeing is to be done.

>> 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"?

* How it waits : a semaphore in the child's thread struct is initialized to zero then downed before the return statement.
  when the child returns from the load fn. it ups its sema allowing the parent to continue.
 * How the load status is passed : a bool in the parent thread is first set to false, if loading succeeds, I go to my parent's bool and set it to true.
  When the parent is unblocked it checks that bool and sets it back to 0. Having s.th. like a global variable in this case is a joke if you allow concurrency in your code.

>> 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?

* Before : The bool 'dead' in struct process is checked , if it's not; a semaphore in that child's thread is downed after being initialized to zero,
  when that child is about to die, it ups the sema, this is when the parent retrieves the exit_status from the child "process".
 * After : if the thread has already died, which can be told from the bool in the process struct -note here that a thread dies, but it's representing process does not-, in this case the exit_status stored in the process is returned.
 * Nothing at all happens if P doesn't wait, it's just that the child ups a sema with no one waiting on it before it dies .

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

** Because this way is simpler than the second method in which we have to modify the code for page_fault() in "exception.c".
   Although the second technique is faster because it takes the advantage of the processor's MMU and it's used in real kernels,
   we chose the first approach for the sake of simplicity.

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

 * Advantages

1. We have added a fd variable into struct file instead of allocating an identity struct for each file.

2. There is no limit on the number of open file descriptors (under the condition of available memory)
   because each thread has its own list of files with unique fd.

 * Disadvantages

1. Time complexity of search into the list of files owned by the thread is O(n).
   It will be better to use a hashmap/array to search in O(1).

>> 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 didn't change tid_t to pid_t.