Lab 2 Design Doc

Project 2: User Programs

Preliminaries

> Fill in your name and email address.

Name: Zhang Lingxin

Email: 2100013018@stu.pku.edu.cn

> If you have any preliminary comments on your submission, notes for the TAs, 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.

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.

My implement of argument passing does not include struct, struct member, global or static variable, typedef or enumeration.

Instead, I use **local assisting variables (**a local `argc` and a local `argv[])` to store the values of real `argc` and `argv[]` temporarily. The assisting variables will be discarded as the function returns.

```
static void
start_process (void *cmdline_)
{
  ...
  int argc = 0; // The number of arguments
  void *argv[64];  // The value of arguments
  /* Note: cmdline is less than 128 bytes, so the number of
     arguments is less than 64 bytes. (A space is needed between
     any two arguments)*/
  ...
}
```

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?

Argument passing (parsing?) is implemented in `start_process()`.

1. The steps of argument passing are as below:
    a. Set the `%esp` to `PHYS_BASE`.

    ```
    /* Stack alignment */
      if_.esp -= ((uintptr_t)if_.esp) % 4;
    ```

    b. Parse the command line input into arguments and put them into the stack in right-to-left order. Store the address in `argv[]` for later use.

    ```
      /* Parse the command line input into arguments and
         put them into the stack in right-to-left order.
         Store the address in argv[] for later use. */
      cur_arg = strtok_r(cmdline_dup, " ", &save_ptr);
      while(cur_arg != NULL){
        int arg_len = strlen(cur_arg) + 1;
        if_.esp -= arg_len;
    ```

```
    argv[argc++] = if_.esp; // store the addr of args in argv
    memcpy(if_.esp, cur_arg, arg_len * sizeof(char));
    cur_arg = strtok_r(NULL, " ", &save_ptr);
  }
```

c. Make the stack word-aligned by rounding the stack pointer down to a multiple of 4.

```
/* Stack alignment */
  if_.esp -= ((uintptr_t)if_.esp) % 4;
```

d. Push the pointers of arguments (stored in `argv[]`) into the stack.
(Note: The size of a pointer is 4 bytes, so there is no need for alignment operation)

```
  /* Push the pointers into the stack*/
  if_.esp -= sizeof(void *);
  *(uintptr_t *)if_.esp = (uintptr_t) 0; // a NULL pointer sentinel
  for(int i = argc - 1; i >= 0; i--){
    if_.esp -= sizeof(void *);
    memcpy(if_.esp, &argv[i], sizeof(void *));
  }
```

e. Push the real `argv` (current `%esp`), `argc` and a fake return address 0 into the stack.

```
  /* Push argv (current %esp) and argc */
  if_.esp -= sizeof(void *);
  *(uintptr_t *)if_.esp = (uintptr_t)if_.esp + sizeof(void*);
  if_.esp -= sizeof(void *);
  *(int *)if_.esp = argc;

  /* Push 0 as a fake return address */
  if_.esp -= sizeof(void *);
  *(uintptr_t *)if_.esp = (uintptr_t) 0;
```

2. To make sure that the elements of `argv[]` to be in the right order, I push the pointers of arguments in **right-to-left** order.
When the process initializes, `argv[0]` is on the top of the stack, and then `argv[1]`, `argv[2]` and so on. The process can access the values of arguments by these

pointers.
The order of the arguments in the stack does not matter. What matters is the order of the pointers to the arguments. (Of course the pointers and the arguments must be correctly connected.)
Still I push the arguments into the stack in the same order for the simplicity of implement.

3. There is a struct member `magic` in the end of the struct `thread`,which is set to a magic number `THREAD_MAGIC`, `0xcd6abf4b`. If there is a stack overflow, the number will be destroyed. So we can check the number to know if the stack is corrupt, and terminate the thread if necessary.

```
static void
start_process (void *cmdline_)
{
  /* Argument passing...*/

  /* Check if there is any stack overflow. */
  if(!thread_check_magic())
    thread_exit();
  ...
}

/* In thread.c */
bool thread_check_magic(){
  return thread_current()->magic == THREAD_MAGIC;
}
```

RATIONALE

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

The prototypes of the two functions:

```
char *strtok_r(char *str, char *delim, char **saveptr);
char *strtok(char *str, const char *delim);
```

The two functions can both break a string `str` into a series of tokens using the delimiter `delim`. But there is difference:

- `strtok_r` is reentrant (thus thread safe) while `strtok` is not.
  Pintos includes interrupts and scheduling, which means that the process should be implemented as reentrant. So we choose `strtok_r` as our choice.
- `strtok`will change the delimiter char in `str` into `\0` while `strtok` won't.

  If we use `strtok`, the initial string `str` will be changed. This may result in some inconvenience if we need to use the initial string later.

  Instead, `strtok_r` never changes the initial string.`strtok_r` splits the string `str` into tokens with the delimiter `delim`, saving the remaining part in `saveptr` and use this pointer for later operation.

> 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.

1. The Unix approach allows the kernel to treat the argument parsing as a common running thread and schedule it as the kernel wants. So the argument parsing can pause when a thread/process of high priority is ready to run. An independent priority scheduling like Unix will allow the kernel to be more flexible and efficient with high-priority process.
2. The shell can be changed, which means that the kernel has better flexibility and scalability.
3. When the argument parsing happens in the shell, the kernel requires less code. So the kernel may load faster.

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> */

struct lock filesys_lock;     /**< A lock for the whole file system. */

/* <thread.h> */

struct thread{
  ...
  /* For file operation*/
  struct file *cur_exec_file; /**< Exec file of current process. */
  struct list file_list;      /**< A list of files the thread opens. */
  int fd_num;                 /**< The number of created fds. */
  ...
}

/* A struct to store the infomation of the thread's open files*/
struct file_info{
    int fd;
    struct file *f;
    struct list_elem elem;
};
```

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

1. The struct `file_info` includes:
   - the file descriptor
   - a pointer to the corresponding `file` struct

   As a file is open, a new file descriptor is allocated for the file. A `file_info` struct is created to store the file descriptor and the pointer to the `file` struct. Then the `file_info` struct is pushed into the `file_list` of current thread.

   A file can be accessed with a file descriptor with the helper function `fd_to_file_info()`. The function returns `NULL` if there is not such a file.

```
static void
syscall_open(struct intr_frame *f){
  ...
  /* Store the information of open file. */
  struct thread *cur = thread_current();
  struct file_info *fi =
          (struct file_info *)malloc(sizeof(struct file_info));
  fi->f = open_file;
  fi->fd = cur->fd_num++;
  list_push_back(&cur->file_list, &fi->elem);
  f->eax = fi->fd;
}

/* Transfer a file descriptor into a pointer to file_info. */
static struct file_info *
fd_to_file_info(int fd){
  struct thread *cur = thread_current();
  struct list_elem *e;
  for(e = list_begin(&cur->file_list); e != list_end(&cur->file_list);
      e = list_next(e)){
    struct file_info *fi = list_entry(e, struct file_info, elem);
    if(fi->fd == fd)
      return fi;
  }
  return NULL;
}
```

2. File descriptors are unique within a single process. (As the information of a opened file is stored in the `thread` struct, there is no need to distinguish between file descriptors of different threads.)

ALGORITHMS

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

1. `syscall_read()`

   The prototype: `int read (int fd, void *buffer, unsigned size)`.

   The system call handler calls `syscall_read()` when the system call number is `SYS_READ`.

   Here are the steps of `syscall_read()`:

- Argument parsing. We parse three arguments on the top the stack and check there validity with `check_ptr_more()` function.
- Judge the file descriptor `fd`.
    - If `fd == STDIN_FILENO`, we read from `STDIN` with `input_getc()`
    - If `fd == STDOUT_FILENO`, this is an illegal operation. We terminate the process with `exit_on_error()`
    - Else `fd` stands for a common file. We call `fd_to_file_info()` to transfer `fd` into a `file_info` struct pointer, check the validity and call `file_read()` in `file.c`. Lock acquisition is necessary.

```
/* 9. read*/
static void
syscall_read(struct intr_frame *f){
  int fd = *(int*)check_ptr_more(f->esp + sizeof(void*));
  void *buffer = *(void **)check_ptr_more(f->esp + 2 * sizeof(void*));
  unsigned size = *(unsigned *)check_ptr_more(f->esp + 3 * sizeof(void*));
  check_ptr_more(buffer);

  /* fd == STDIN: input_getc() */
  if(fd == STDIN_FILENO){
    uint8_t *b = (uint8_t *)buffer;
    for(int i = 0; i < size; i++)
      b[i] = input_getc();
    f->eax = size;
    return;
  }

  /* fd == STDOUT: terminate the process. */
  if(fd == STDOUT_FILENO){
    exit_on_error();
  }

  /* Other file descriptors: file_read() */
  struct file_info *fi = fd_to_file_info(fd);
  if(fi != NULL && fi->f != NULL){
    lock_acquire(&filesys_lock);
    f->eax = file_read(fi->f, buffer, size);
    lock_release(&filesys_lock);
    return;
  }

  /* Return -1 on failure. */
  f->eax = -1;
}
```

2. `syscall_write()`

   The prototype: `int write (int fd, void *buffer, unsigned size)`.

   The system call handler calls `syscall_write()` when the system call number is `SYS_WRITE`.

   Here are the steps of `syscall_write()`:

   - Argument parsing. We parse three arguments on the top the stack and check there validity with `check_ptr_more()` function.
   - Judge the file descriptor `fd`.
     - If `fd == STDIN_FILENO`, this is an illegal operation. We terminate the process with `exit_on_error()`
     - If `fd == STDOUT_FILENO`, we write to `STDOUT` with `put_buf()`
     - Else `fd` stands for a common file. We call `fd_to_file_info()` to transfer `fd` into a `file_info` struct pointer, check the validity and call `file_write()` in `file.c`. Lock acquisition is necessary.

```
/* 10. write */
static void
syscall_write(struct intr_frame *f){
  int fd = *(int*)check_ptr_more(f->esp + sizeof(void*));
  void *buffer = *(void**)check_ptr_more(f->esp + 2 * sizeof(void*));
  unsigned size = *(unsigned *)check_ptr_more(f->esp + 3 * sizeof(void*));
  check_ptr_more(buffer);

  /* If trying to write to STDIN, terminate */
  if(fd == STDIN_FILENO)
    exit_on_error();

  /* STDOUT: putbuf() */
  if(fd == STDOUT_FILENO){
    putbuf((char*)buffer, size);
    f->eax = size; // Return the number of chars
    return;
  }

  /* Other file descriptors: fild_write() */
  struct file_info *fi = fd_to_file_info(fd);
  if(fi != NULL){
    lock_acquire(&filesys_lock);
    f->eax = file_write(fi->f, buffer, size);
    lock_release(&filesys_lock);
    return;
  }

  /* Return -1 on failure. */
  f->eax = -1;
}
```

> 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?

A page has 4096 bytes, so a block of a page size can occupy at most two pages. If we are lucky, the 4096 bytes may be in the same page. So the least number of inspections of the page table is 1 and the greatest is 2.

For a system call that only copies 2 bytes, most of the time they are in the same page; but sometimes they may be in two pages. So the least number of inspections of the page table is 1 and the greatest is 2.

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

The prototype of system call `WAIT`: `int wait (pid_t pid)`.

The system call handler calls `syscall_wait()` when the system call number is `SYS_WAIT`. Then `syscall_wait()` calls `process_wait()` to finish a “wait” behavior.

There is a semaphore, `sema_child` in the struct `child_info`. The parent can use the semaphore to wait on a child:

1. When a parent tries to wait on a child process:

- The process searches the child list for the child with correct `pid`. If it doesn't find the child, the function returns -1.
- If the child has been waited by another process (`waited == 1`), the process return -1; else set `waited` as 1.
- If the child has not exited, use`sema_down(&c->sema_child)` to wait on the child and return `exit_code`.
- If the child has exited, return `exit_code`.

```
int
process_wait (tid_t child_tid)
{
  ..
  /* Search the child list for the child*/
  for(e = list_begin(&cur->child_list); e != list_end(&cur->child_list);
      e = list_next(e)){
    struct child_info *c = list_entry(e, struct child_info, elem);

    if(c->tid == child_tid){
      /* If the child has been waited on, return -1. */
      if(c->waited)
        return -1;
      c->waited = 1;

      /* If the child is alive and never waited on, wait on it. */
      if(!c->exited){
          sema_down(&c->sema_child); // waiting for the child
          return c->exit_code;
      }

      /* If the child has exited, return the exit code. */
      return c->exit_code;
    }
  }
  return -1;
  ..
}
```

2. When a child is about to exit:

- The child checks if a parent may be waiting on it (`cur->parent != NULL`). If yes, the child set the `exit_code` in the struct `child_info` ; If not, just free the struct `child_info`.

- The child checks if the parent is waiting on it (`cur->myinfo->waited == 1`). If yes, the child uses `sema_up(&cur->myinfo->sema_child)` to tell the parent that it has exit.

```
void
thread_exit (void)
{
  ..
  /* Check if the thread's parent has exited */
  if(cur->parent != NULL){
    cur->myinfo->exit_code = cur->exit_code;

    /* If the parent is waiting for the child, release the lock */
    if(cur->myinfo->waited)
      sema_up(&cur->myinfo->sema_child);
    cur->myinfo->child = NULL;
    cur->myinfo->exited = 1;
  }
  else
    free(cur->myinfo);
  ..
}
```

The parent and the child communicate by the semaphore `sema_child` and the struct `child_info`.

> 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.

1. Error-checking

   To make the design more simple and reduce repeated error-checking code, I use two helper functions `check_ptr()` and `check_ptr_more()`. They will check the validity of a pointer and return the pointer if the pointer is valid, or end the process if the pointer is invalid.

   The difference between the two is that `check_ptr()` only checks the pointer itself, while`check_ptr_more()` checks the first address and the last address the user program is trying to access.

```
static void *
check_ptr(void *ptr){
  /* Is the address legal? */
  if(ptr == NULL || !is_user_vaddr(ptr))
    exit_on_error();

  /* Is the address in the current thread page? */
  void *pgptr = pagedir_get_page(thread_current()->pagedir, ptr);
  if(pgptr == NULL)
    exit_on_error();

  /* Is the address readable?*/
  if(get_user((uint8_t *)ptr) == -1)
    exit_on_error();

  /* Everything checked. A safe pointer. */
  return ptr;
}
```

   So when I need to dereference an unknown pointer provided by the user program, I check its validity by calls `check_ptr()` or `check_ptr_more()`. If the pointer passes the check, the function returns a copy of the pointer; if the pointer is invalid, the process just terminates in the checking function.

   Here is an example:

```
/* 5. CREATE */
static void
syscall_create(struct intr_frame *f){
  char *name = *(char **)check_ptr_more(f->esp + sizeof(void *));
  check_ptr_more(name);
  unsigned ini_size =
        *(unsigned*)check_ptr_more(f->esp + sizeof(void *) * 2);

  lock_acquire(&filesys_lock);
  f->eax = filesys_create(name, ini_size);
  lock_release(&filesys_lock);
}
```

2. Resources freeing

   I write a helper function `exit_on_error()`. The function will make sure that the file system lock is release, set the `exit_code` and then calls `thread_exit()` to terminate the process. The freeing of other resources (e.g. malloced structures) is handled in `thread_exit()`.

   When a fatal error is detected(e.g. an invalid pointer or trying to write to `STDIN`), the function is called to terminate the process safely:

```
/* Terminate the process when a fatal error occurs. */
static inline void
exit_on_error(){
  if(lock_held_by_current_thread(&filesys_lock)){
    lock_release(&filesys_lock);
  }
  thread_current()->exit_code = -1;
  thread_exit();
  NOT_REACHED();
}
```

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"?

There is a semaphore, `sema_parent` , and a bool, `exec_success` in `thread` struct. The former is used for the communication of “wait”, and the latter stores the loading status of the child.

1. The parent:

   When a thread tries to create a child in the function `process_execute()`:

   - It does everything necessary including parameter initialization and child thread creating.
   - It uses `sema_down(&cur->sema_parent)` to wait on the child.

   ```
   tid_t
   process_execute (const char *cmdline)
   {
     ...
     /* Wait for its child. Return -1 if fails. */
     sema_down(&cur->sema_parent);

     /* Return -1 on failure. */
     if(!cur->exec_success)
       return -1;
     /* Reset the flag on success. */
     cur->exec_success = 0;

     return tid;
   }
   ```

2. The child:

   When the child tries to load in the function `start_process()`:

   - If the child doesn’t load correctly, it sets `exec_success` as 0 and sets `exit_code` as -1. Then the child uses `sema_up(&cur->parent->sema_parent)` to tell the parent about its failure;
   - If the child loads correctly, it sets `exec_success` as 1. Then the child uses `sema_up(&cur->parent->sema_parent)` to tell the parent that it has loaded correctly.

   ```
   static void
   start_process(void *cmdline_){
     /* After loading... */

     /* If load failed, quit. */
     if (!success) {
   ```

```
      ...
      /* Information update. */
      cur->myinfo->exited = 1;
      cur->exit_code = -1;

      /* Wake up the parent thread after setting the return value. */
      sema_up(&cur->parent->sema_parent);

      thread_exit ();
    }

    /* Load success. Wake up the waiting parent. */
    cur->parent->exec_success = 1;
    sema_up(&cur->parent->sema_parent);
  }
```

> 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?

1. How do you ensure proper synchronization and avoid race conditions when P calls `wait(C)` before C exits? After C exits?

   There is a semaphore `sema_child` initialized as 0 in struct `thread`. The parent calls `sema_down()` and the child calls `sema_down()` if it detects that the parent is waiting for it.

   - If P calls `wait(c)` before C exits, P calls `sema_down()` to wait on the child. Only after C calls `sema_up()` can P continues.
   - If P calls `wait(c)` after C exits, the value of `sema_child` is 1 because C has called `sema_up()`. So when P calls `sema_down()` to wait on the child, the function will return at once.

2. How do you ensure that all resources are freed in each case?

Only resources allocated by `palloc_get_page()` or `malloc()` need to be freed. In this case, we just need to free struct `child_info`.

Every `child_info` struct is linked with a specific parent-child relationship. It is no longer useful either the parent exits or the child exits.

In my implement, it is freed when a thread A exits in `thread_exit()`:

- The function goes through the child list of A and checks every child for its status. If a child has exited, it is removed from the list and the relevant `child_info` struct is freed.
- The function checks if the parent of A has exit. If yes, the relevant `child_info` struct is freed.

The two steps make sure that all the useless `child_info` structures are freed.

```
void
thread_exit (void)
{
  ...
  /* Go through the child list and set the child's parent as NULL */
  e = list_begin(&cur->child_list);
  while(e != list_end(&cur->child_list)){
    struct child_info *c = list_entry(e, struct child_info, elem);
    if(!c->exited){
      c->child->parent = NULL;
      e = list_next(e);
    }
    else{
      struct list_elem *ee = e;
      e = list_next(e);
      list_remove(ee);
      free(c);
    }
  }
  ...

  /* Check if the thread's parent has exited */
  if(cur->parent != NULL){
    cur->myinfo->exit_code = cur->exit_code;

    /* If the parent is waiting for the child, release the lock */
    if(cur->myinfo->waited)
      sema_up(&cur->myinfo->sema_child);
    cur->myinfo->child = NULL;
    cur->myinfo->exited = 1;
  }
  else
```

```
    free(cur->myinfo);
}
```

3. How about when P terminates without waiting, before C exits? After C exits?

   If P terminates without waiting, the struct member `waited` in `child_info` won't be set as 1.

   - If P terminates without waiting before C exits, C will know this when it checks `waited` in `child_info`. Then C won't do anything about "wait".
   - If P terminates without waiting after C exits, nothing about "wait" will happens.

   As for the resource freeing, P and C both checks if the `child_info` struct should be freed when exit. So the resource will always be freed correctly by the first to exit.

RATIONALE

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

I choose the second method: to check only that a user pointer points below `PHYS_BASE`, then dereference it.

To make it more simple, I use two helper functions `check_ptr()` and `check_ptr_more()`. They will check the validity of a pointer and return the pointer if the pointer is valid, or end the process if the pointer is invalid. The difference between the two is that `check_ptr()` only checks the pointer itself, while `check_ptr_more()` checks the first address and the last address the user program is trying to access.

```
static void *
check_ptr(void *ptr){
  /* Is the address legal? */
  if(ptr == NULL || !is_user_vaddr(ptr))
    exit_on_error();

  /* Is the address in the current thread page? */
  void *pgptr = pagedir_get_page(thread_current()->pagedir, ptr);
  if(pgptr == NULL)
    exit_on_error();
  /* Is the address readable?*/
  if(get_user((uint8_t *)ptr) == -1)
    exit_on_error();

  /* Everything checked. A safe pointer. */
  return ptr;
}
```

Here are the reasons:

- The method makes use of the processor's MMU, so it should be faster than the first method.
- The implement is not very complex, as two helper functions `get_user()` and `put_user()` have been provided.

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

I define a struct `file_info` to store all the information about a opened file.

Every `file_info` struct is an element of `file_list` in a specific thread, standing for an opened file in the thread. The struct links a file descriptor with a `file` struct. read

```
/* A struct to store the infomation of the thread's opened files*/
struct file_info{
    int fd;
    struct file *f;
    struct list_elem elem;
};
```

- Advantages:
  - Less synchronic problems. There is no need to synchronize any file information (e.g. file descriptors) between threads. If two processes tries to search for files in their own file list in the same time, there is not race condition.
  - The resources are easy to manage. Since every `file_info` struct is belonged to a specific thread, the resource malloced for an opened file can be freed when the thread terminates.

    - The implement is straight-forward and simple.
- Disadvantages:
    - Possible confusion in code writing. The file descriptors are per-process, which can result in some confusion in debugging and code writing.
    - Potential higher storage. The struct `file_info` is created for every file-thread pair, which may need more storage space than storing file information in global context.
    - Higher cost of file searching. As the `file_info` is stored in a list `file_list`, we have to go through the list to find a specific file.

> B11: The default `tid_t` to `pid_t` mapping is the identity mapping. If you changed it, what advantages are there to your approach?

I didn't change it. The mapping stays the same.